POLYGON RENDERING FOR INTERACTIVE VISUALIZATION ON MULTICOMPUTERS

by

David Allan Ellsworth

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1996

© 1996

David Ellsworth

ALL RIGHTS RESERVED

DAVID ALLAN ELLSWORTH. Polygon Rendering for Interactive Visualization on Multicomputers (Under the direction of Professor Henry Fuchs).

ABSTRACT

This dissertation identifies a class of parallel polygon rendering algorithms suitable for interactive use on multicomputers, and presents a methodology for designing efficient algorithms within that class. The methodology was used to design a new polygon rendering algorithm that uses the frame-to-frame coherence of the screen image to evenly partition the rasterization at reasonable cost. An implementation of the algorithm on the Intel Touchstone Delta at Caltech, the largest multicomputer at the time, renders 3.1 million triangles per second. The rate was measured using a 806,640 triangle model and 512 i860 processors, and includes back-facing triangles. A similar algorithm is used in Pixel-Planes 5, a system that has specialized rasterization processors, and which, when introduced, had a benchmark score for the SPEC Graphics Performance Characterization Group "head" benchmark that was nearly four times faster than commercial workstations. The algorithm design methodology also identified significant performance improvements for Pixel-Planes 5.

All fully parallel polygon rendering algorithms have a sorting step to redistribute primitives or fragments according to their screen location. The algorithm class mentioned above is one of four classes of parallel rendering algorithms identified; the classes are differentiated by the type of data that is communicated between processors. The identified algorithm class, called sort-middle, sorts screen-space primitives between the transformation and rasterization.

The design methodology uses simulations and performance models to help make the design decisions. The resulting algorithm partitions the screen during rasterization into adaptively sized regions with an average of four regions per processor. The region boundaries are only changed when necessary: when one region is the rasterization bottleneck. On smaller systems, the algorithm balances the loads by assigning regions to processors once per frame, using the assignments made during one frame in the next. However, when 128 or more processors are used at high frame rates, the load balancing may take too long, and so static load balancing should be used. Additionally, a new all-to-all communication method improves the algorithm's performance on systems with more than 64 processors.

ACKNOWLEDGEMENTS

I am grateful for the advice and support given by my advisor, Henry Fuchs, and by my committee members, Frederick Brooks, Jr, Kevin Jeffay, Anselmo Lastra, and John Poulton. Anselmo Lastra deserves extra thanks for reading several drafts of the dissertation. I have also profited from many discussions with other members of the Pixel-Planes project team. Special thanks go to Steve Molnar, Bill Mark, and Victoria Interrante for their discussions, and Carl Mueller and Vincent Illiano for reading a draft of the dissertation. I doubt that I would have finished without the support from those mentioned above.

I am also grateful for the support provided by the National Science Foundation and the Defense Advanced Research Projects Agency, and by an Office of Naval Research fellowship. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Initial access to the Delta was provided by Steve Taylor and Paul Messina of Caltech; the bulk of the work was done with access provided by DARPA. The NSF/DARPA Science and Technology Center for Computer Graphics and Scientific Visualization provided support for travel to the California Institute of Technology for my initial work on the Delta.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES xi

LIST OF ABBREVIATIONS xvii

LIST OF SYMBOLS xix

1 INTRODUCTION 1
- 1.1 Overview 1
  - 1.1.1 Targeted Parallel System Architecture 1
  - 1.1.2 Targeted Applications 2
- 1.2 Thesis Statements 2
- 1.3 Analysis of Redistribution Choices 2
- 1.4 Sort-Middle Design Methodology 4
  - 1.4.1 Design Choices for the Overall Algorithm Structure 4
  - 1.4.2 Design Choices for Load-Balancing the Rasterization 7
  - 1.4.3 Geometry Processing Design Choices 8
  - 1.4.4 Implementation to Validate the Design Choices 9
  - 1.4.5 Systems with Specialized Rasterization Processors 9
- 1.5 Description of the Target System 9
- 1.6 Structure of the Dissertation 10
- 1.7 Expected Reader Background 10

2 BACKGROUND 11
- 2.1 The Graphics Pipeline 11
- 2.2 Types of Parallelism 13
- 2.3 Algorithms for General-Purpose Parallel Computers 13
  - 2.3.1 Image-Parallel Polygon Algorithms 14
  - 2.3.2 Object-Parallel Polygon Algorithms 17
  - 2.3.3 Ray Tracing Algorithms 19
- 2.4 Hardware Graphics Systems 20
  - 2.4.1 Image-Parallel Systems 20
  - 2.4.2 Object-Parallel Architectures 27
- 2.5 Summary 29

3 CLASSES OF PARALLEL POLYGON RENDERING ALGORITHMS 30
- 3.1 Description of the Algorithm Classes 30
  - 3.1.1 Sort-First 31
  - 3.1.2 Sort-Middle 32
  - 3.1.3 Sort-Last 34
  - 3.1.4 Other Algorithms 35
- 3.2 Models of the Algorithm Classes 35
  - 3.2.1 Serial Algorithm Cost 36
  - 3.2.2 Sort-First Overhead 36
  - 3.2.3 Sort-Middle Overhead 37
  - 3.2.4 Sort-Last-Sparse Overhead 38
  - 3.2.5 Sort-Last-Full Overhead 38
- 3.3 Comparisons of the Classes 38
  - 3.3.1 Sort-First vs. Sort-Middle 39
  - 3.3.2 Sort-Middle vs. Sort-Last-Sparse 40
  - 3.3.3 Sort-Middle vs. Sort-Last-Full 40
- 3.4 Summary 41

4 SORT-MIDDLE ALGORITHMS 42

4.1 Assigning Stages to Processors ... 43
4.2 Scheduling the Stages ... 44
4.3 Region Assignment ... 44
4.3.1 Work-Queue Region Assignments ... 45
4.3.2 Assignments of Regions Between Geometry and Rasterization Stages ... 46
4.3.3 Assignments of Regions Between Frames ... 46
4.3.4 Static Region Assignment ... 46
4.3.5 Hybrids ... 46
4.3.6 Region Assignment Cost Comparison ... 47
4.3.7 Summary of the Region Assignment Styles ... 49
4.4 Redistribution ... 49
4.4.1 Number of Message Streams Sent Per Processor ... 50
4.4.2 Two-Step Redistribution ... 51
4.4.3 A Simple Method for Choosing the Number of Groups ... 52
4.4.4 Costs of Different Redistribution Methods ... 52
4.4.5 Redistribution Cost Examples ... 53
4.4.6 A Better Method for Choosing the Number of Groups ... 53
4.4.7 Message Size ... 57
4.5 Recommendations and Summary ... 59

5 LOAD-BALANCING GEOMETRY PROCESSING ... 61
5.1 Algorithms for Distributing Hierarchical Display Lists ... 61
5.1.1 Distribute-by-Structure ... 62
5.1.2 Distribute-by-Primitive ... 62
5.1.3 Summary of Strategies ... 64
5.2 Optimizing the Distribute-by-Primitive Algorithm ... 64
5.2.1 Removing Superfluous Attribute Changes ... 64
5.2.2 Performing the Transformation Concatenation During Distribution ... 64
5.2.3 Primitive Structures ... 65
5.3 Randomization to Further Remove Coherence ... 66
5.4 Evaluating the Algorithms ... 66
5.4.1 Simulator Description ... 67
5.4.2 Performance Model Used in the Simulator ... 67
5.4.3 Simulation Results ... 68
5.5 Summary ... 69

6 LOAD-BALANCING RASTERIZATION ... 70
6.1 Evaluation Methodology ... 70
6.1.1 Simulator Description ... 71
6.1.2 Performance Model ... 72
6.2 Load-Balancing Methods ... 79
6.2.1 Region Aspect Ratio ... 80
6.2.2 Load-Balancing Methods for Static Region Assignments ... 80
6.2.3 Load-Balancing Methods for Once-Per-Frame Assignments ... 82
6.2.4 Load-Balancing Methods for Work-Queue Assignments ... 89
6.3 A Tour Through the Simulation and Performance Model Results ... 89
6.3.1 Typical Simulation Results ... 89
6.3.2 Typical Performance Model Results ... 92
6.3.3 Size-Based Cost Estimates ... 94
6.3.4 Adaptive Region Boundary Methods ... 94
6.3.5 Load-Balancing Differences Between the Between-Stages and Between-Frames Assignment Styles ... 95
6.3.6 Components of the Performance Model ... 97
6.4 Best Load-Balancing Method for Each Assignment Style ... 98
6.4.1 Static Methods ... 99
6.4.2 Once-Per-Frame Methods ... 99
6.4.3 Load-Balancing for Work-Queue Assignments ... 102
6.5 Comparison of Region Assignment Styles ... 103
6.6 Summary and Conclusions ... 106

7 SORT-MIDDLE IMPLEMENTATION ON THE TOUCHSTONE DELTA 109
7.1 Implementation Description 109
7.1.1 Interface Features 110
7.1.2 Algorithm Description 110
7.1.3 Redistribution 111
7.1.4 Load-Balancing Implementation 115
7.1.5 Graphics Routines 116
7.2 Performance Results 117
7.2.1 Avoiding the Frame Buffer Bottleneck 117
7.2.2 Polygons per Second 117
7.2.3 Load-Balancing Bottleneck 118
7.2.4 Comparison with the Performance Model 119
7.2.5 Components of the Processing Time 120
7.2.6 Anti-Aliasing Performance 122
7.3 Evaluation of the Design Choices 123
7.3.1 Redistribution Methods 123
7.3.2 Granularity Ratio 125
7.3.3 Maximum Message Size 127
7.4 Summary and Discussion 129

8 USING SPECIALIZED RASTERIZATION HARDWARE 131
8.1 Parallel Rendering using Hardware Rasterization 131
8.1.1 Heterogeneous Processing 131
8.1.2 Work-Queue Region Assignments 132
8.1.3 Fixed Region Sizes 133
8.1.4 Effect of Faster Processors 134
8.1.5 Summary 135
8.2 PPHIGS Implementation on Pixel-Planes 5 135
8.2.1 Hardware Configuration 135
8.2.2 Pixel-Planes 5 Rendering Algorithm 136
8.2.3 Performance 142
8.3 MEGA Implementation on Pixel-Planes 5 150
8.3.1 MEGA Rendering Algorithm 150
8.3.2 MEGA Performance Limitations 151
8.4 The Value of Specialized Hardware 153
8.4.1 Rasterization Hardware Price-Performance Ratio 153
8.4.2 Comparison of Pixel-Planes 5 and the Touchstone Delta 154
8.5 Possible PPHIGS Future Work 156
8.6 Summary 157

9 CONCLUSIONS AND FUTURE WORK 158
9.1 Summary and Conclusions 158
9.2 Future Work 160

Appendix

A SAMPLE MODELS AND VIEWPOINTS 163
A.1 PLB Head 165
A.2 Terrain 167
A.3 CT Head 169
A.4 Polio 171
A.5 Well 173

B SOURCES OF DELTA TIMING VALUES AND SIZES 175
B.1 Timing Values 175
B.1.1 Communication Times 176
B.1.2 Geometry Times 177
B.1.3 Rasterization Times 177
B.2 Data Structure Sizes 178

C SIMULATION AND MODELING RESULTS 179

C.1 Simulation Results 179
C.1.1 PLB Head Model 180
C.1.2 Terrain Model 184
C.1.3 CT Head Model 188
C.1.4 Polio Model 192
C.2 Performance Model Results 196
C.2.1 PLB Head Model 197
C.2.2 Terrain Model 199
C.2.3 CT Head Model 201
C.2.4 Polio Model 203

D PIXEL-PLANES 5 BOARD COSTS 205

E GLOSSARY 207

References 209

LIST OF TABLES

Table 3.1: Symbols used in the analysis of the algorithm classes. 36

Table 3.2: Costs used in the analysis of the algorithm classes. The values are from an Intel Touchstone Delta implementation. 36

Table 3.3: Data sizes used in the analysis of the algorithm classes. 36

Table 3.4: Serial algorithm cost. 36

Table 3.5: Overhead costs in sort-first algorithms. 37

Table 3.6: Overhead costs in sort-middle algorithms. 38

Table 3.7: Overhead costs in sort-last-sparse algorithms. 38

Table 3.8: Overhead costs in sort-last-full algorithms. 38

Table 3.9: Overhead costs summary. 39

Table 4.1: Summary of geometry processing and rasterization stage scheduling characteristics. 45

Table 4.2: Costs of different rasterization region assignment styles for *r* regions and *N* processors. 47

Table 4.3: Symbols used for illustrating the region assignment costs. 48

Table 4.4: Characteristics of the region assignment styles. 49

Table 6.1: Timing values used in the simulator. 72

Table 6.2: Constants used in the performance model. 74

Table 6.3: Percentage of average difference of expected performance between methods using triangle counts and size-based cost estimates. 94

Table 6.4: Maximum relative processor utilization between corresponding load-balancing methods for the between-stages and between-frames assignment styles. 98

Table 6.5: Best load-balancing method for different models and number of processors between the modular and interleaved static load-balancing method. 100

Table 6.6: Percentage difference between the best static load-balancing method (I 12) and the best ones for each model and partition size as shown in table 6.5. 100

Table 6.7: Best load-balancing method for different models, number of processors, and between-stages and between-frames assignment styles. 101

Table 6.8: For both between-stages and between-frames assignment styles, percentage difference in estimated time between the best load-balancing method shown in table 6.7 and fixed region load-balancing with size-based cost estimates and $G = 6$. 102

Table 6.9: Best load-balancing method for different models and number of processors for the three work-queue load-balancing methods. 103

Table 6.10: Best region assignment style for different models and number of processors. 104

Table 6.11: Percentage of additional time taken when using between-frames assignment of regions compared to the best assignment style for each configuration. 104

Table 6.12: Percentage of additional time taken when using static assignment of regions to the best assignment style for each configuration. 104

Table 6.13: Percentage of additional time taken when using between-frames assignment of regions, with six fixed regions per processor and using at most costs from 32 processors compared to the best assignment style for each configuration. 105
Table 7.1: Performance model divided into the different categories. 121
Table 7.2: The faster redistribution method used for the different models, for a range of partition sizes. 126
Table 7.3: Predicted and actual granularity ratios with the highest average triangle rates for the four models and a range of partition sizes. 128
Table 7.4: Maximum message size, in kilobytes, which produces the highest average triangle rate. 129
Table 7.5: Number of bytes exchanged between processors (averages). 129
Table 8.1: Average number of triangles per second, in millions, rendered on Pixel-Planes 5 using 54 SGPs and 22 renderers, for each of the four models. 149
Table 8.2: Comparison of Pixel-Planes 5 and the Intel Touchstone Delta. 155
Table A.1: Statistics for the PLB Head model. 165
Table A.2: Statistics for the Terrain model. 167
Table A.3: Statistics for the CT Head model. 169
Table A.4: Statistics for the Polio model. 171
Table A.5: Statistics for the Well model. 173
Table B.1: Description of times, their values, and the source of the values. 175
Table B.2: Description of data structure sizes and the source of the sizes. 178
Table C.1: Overlap figures mapping table for the PLB Head model. 182
Table C.2: Overlap figures mapping table for the Terrain model. 186
Table C.3: Overlaps figure mapping table for the CT Head model. 190
Table C.4: Overlap figures mapping table for the Polio model. 194
Table D.1: Cost of a Pixel-Planes 5 renderer board. 205
Table D.2: Cost of a Pixel-Planes 5 Intel i860 processor (one half of a board). 206

LIST OF FIGURES

Figure 1.1: The three algorithm classes. 3

Figure 1.2: Taxonomy of the sort-middle algorithms encompassed by the design choices considered in the dissertation. The choices examined in the dissertation are indicated with solid lines. 5

Figure 1.3: The Intel Touchstone Delta configuration. 10

Figure 2.1: Example graphics pipelines for three types of shading. 12

Figure 2.2: Taxonomy of the parallel polygon rendering algorithms for general purpose systems covered in this section. 14

Figure 2.3: Regular screen subdivision. 14

Figure 2.4: Whitman's adaptive decomposition with 8 regions. 16

Figure 2.5: Data flow of Whitman's implementation. 16

Figure 2.6: Binary-swap composition example with four processors. 17

Figure 2.7: Taxonomy of the ray-tracing algorithm covered in this section. 19

Figure 2.8: Taxonomy of the hardware graphics systems discussed in this section. 21

Figure 2.9: Example of interleaved partitioning using four processors. 21

Figure 2.10: Architectures proposed by Fuchs and Parke. 22

Figure 2.11: Example of Whelan's adaptive subdivision. 23

Figure 2.12: The SAGE architecture. 23

Figure 2.13: (a) Silicon Graphics GTX architecture (b) Silicon Graphics VGX Architecture. 25

Figure 2.14: Silicon Graphics RealityEngine architecture. 26

Figure 2.15: Example of a video-output image composition system. 27

Figure 3.1: Graphics pipeline in a fully parallel rendering system (from [Moln94]). 30

Figure 3.2: Sort-first. Redistributes raw triangles during geometry processing (from [Moln94]). 32

Figure 3.3: Sort-middle. Redistributes screen-space triangles between geometry processing and rasterization (from [Moln94]). 33

Figure 3.4: Sort-last. Redistributes pixels, samples, or pixel fragments during rasterization (from [Moln94]). 34

Figure 4.1: Taxonomy of the sort-middle algorithms encompassed by the design choices considered in this chapter. 42

Figure 4.2: Total processing cost for load-balancing. 48

Figure 4.3: Two examples of grouping for two-step redistribution. 51

Figure 4.4: Comparison of communication costs as a function of number of processors for a 10,000 triangle scene for the four redistribution methods. 54
Figure 4.5: Comparison of communication costs as a function of number of processors for a 100,000 triangle scene for the four redistribution methods. 54
Figure 4.6: Comparison of communication costs as a function of number of processors for a 1,000,000 triangle scene for the four redistribution methods. 54
Figure 4.7: Predicted optimum group size for different model sizes and numbers of processors. 58
Figure 4.8: Fraction of time required for redistribution using the number of groups calculated using the performance model compared to the time required when using $\sqrt{N}$ groups. 58
Figure 4.9: Pseudocode for the basic algorithm. 60
Figure 5.1: Simple distribute-by-primitive example, where two structures are distributed across two processors. 63
Figure 5.2: Simple method for distributing a single structure across four processors. 63
Figure 5.3: Pseudocode for the optimized distribution algorithm. 65
Figure 5.4: Distributing a single structure across four processors with superfluous attributes omitted and concatenation transformations folded into replace transformations. 66
Figure 5.5: Processor utilization as predicted by simulation for the four models and two distribution methods: assignment to the processor with the minimum load and random assignment to the processor with the minimum load. These values are with back-face culling disabled. 68
Figure 5.6: Processor utilization as predicted by simulation for three models and two distribution methods: assignment to the processor with the minimum load and random assignment to the processor with the minimum load. These values are with back-face culling enabled. 68
Figure 6.1: Taxonomy of the design choices described in this section with the names of the load-balancing methods 79
Figure 6.2: Example of modular static load-balancing with $N = 4$, $G = 4$, and $\alpha = 5/4$. 81
Figure 6.3: Example of interleaved static load-balancing with $N = 4$, $G = 4$, and $\alpha = 5/4$. 81
Figure 6.4: Example of assigned-region load-balancing with $N = 4$, $G = 4$, and $\alpha = 5/4$. 83
Figure 6.5: Example of adjusting the region boundaries for eight frames. 85
Figure 6.6: Example of when-needed adjustment of region boundaries. 86
Figure 6.7: Cases where 50, 0, and 100% of the pixels in bounding box are rendered. 87
Figure 6.8: Comparison of the predictor of the number of pixels within the triangle against the actual number of pixels in the triangle, for triangles sampled from the Terrain model. 88
Figure 6.9: Comparison of the predictor of the number of pixels within the triangle against the actual number of pixels in the triangle, for triangles sampled from the Well model. 88
Figure 6.10: Per-frame values from a simulation run using the Terrain model with $N = 128$, $G = 4$, and between-stages assignment of fixed regions using triangle counts. 90
Figure 6.11: Simulated processor utilization for a series of frames and a range of granularity ratios, all using the Terrain model, $N = 128$, and between-stages assignment of fixed regions using triangle counts. 91
Figure 6.12: Simulated processor utilization for a series of frames and a range of number of processors, all using the Terrain model, $G = 4$, and current frame assignment of fixed regions. 91

Figure 6.13: Average processor utilization expected for the Terrain model, for between-stages assignment of fixed regions using triangle counts, and for a range of granularity ratios and numbers of processors. 92
Figure 6.14: Average overlap factor for the Terrain model, for between-stages assignment of fixed regions using triangle counts, and for a range of granularity ratios and numbers of processors. 92
Figure 6.15: Time predicted by the performance model for the Terrain data set, for between-stages assignment of fixed regions using triangle counts for a range of granularity sizes. 93
Figure 6.16: Relative times from the performance model for the Terrain data set, for between-stages assignment of fixed regions using the triangle counts for a range of granularity sizes. 93
Figure 6.17: Simulated processor utilization for the per-frame adaptive-region and when-needed adaptive-region load-balancing methods relative to the fixed region method, for the Terrain model for between-stages assignment using triangle counts and a range of granularity ratios. 95
Figure 6.18: Overlap factor values for the per-frame adaptive-region and when-needed adaptive-region load-balancing methods relative to the fixed-region method, for the Terrain model for between-stages assignment using triangle counts and a range of granularity ratios. 96
Figure 6.19: Predicted frame time for the per-frame adaptive-region and when-needed adaptive-region load-balancing methods relative to the fixed-region method, for the Terrain model for between-stages assignment using triangle counts and a range of granularity ratios. 97
Figure 6.20: Relative simulated processor utilization for the Terrain model between the between-stages and between-frames assignment styles using triangle counts and fixed regions, for a range of granularity ratios and numbers of processors. 98
Figure 6.21: Performance model components for the PLB Head and Polio models. 99
Figure 6.22: Average relative predicted frame times when using static region assignments. 100
Figure 6.23: Average relative predicted frame times when using the between-frames region assignment style. 101
Figure 6.24: Average relative predicted frame times when using the between-stages region assignment style. 101
Figure 6.25: Typical predicted frame times when using work-queue assignment. 102
Figure 6.26: Average relative predicted frame times when using work-queue region assignments. Each curve corresponds to the marked load-balancing method. 103
Figure 6.27: Comparison of five region assignment styles and load-balancing methods for the four models. 105
Figure 7.1: Pseudocode of the implementation's rendering loop. 111
Figure 7.2: Communication paths for the one-step stream-per-region communication method. 113
Figure 7.3: Communication paths for the one-step stream-per-processor communication method. 113
Figure 7.4: Communication paths for the two-step stream-per-region communication method. 114
Figure 7.5: Communication paths for the two-step stream-per-processor communication method. 115
Figure 7.6: Summing trees using eight processors with fan-ins of two and four. 115

Figure 7.7: Speed of the implementation on the Intel Touchstone Delta for the four models compared with a linear speedup. 118
Figure 7.8: Comparison of triangle rendering rates for three load-balancing methods for the four models. 119
Figure 7.9: Predicted versus actual total frame processing times (average frame time multiplied by the number of processors) for the four models. 120
Figure 7.10: Predicted versus actual total frame processing times for the four models when load-balancing information is read from a file. 121
Figure 7.11: Aggregate time spent on different tasks for the PLB Head model. 122
Figure 7.12: Time spent on different tasks for the PLB Head model, in percentages. 122
Figure 7.13: Aggregate time spent on different tasks for the Polio model. 123
Figure 7.14: Time spent on different tasks for the Polio model, in percentages. 123
Figure 7.15: Speed when performing anti-aliasing compared with the non-anti-aliasing speed. 124
Figure 7.16: Rendering rate with anti-aliasing enabled as a fraction of the point-sampled rendering rate. 124
Figure 7.17: Speed with the PLB Head and Terrain models, for four different communication methods. 125
Figure 7.18: Speed with the CT Head and Polio models, for four different communication methods. 126
Figure 7.19: Average number of triangles per second for granularity ratios (G) of 4, 6, and 8. 127
Figure 7.20: Normalized triangle rendering rates for granularity ratios of 4, 6, and 8. 128
Figure 7.21: Implementation performance as the maximum message size is varied. 129
Figure 8.1: Rasterization processor utilization predicted by simulation for 128 by 128 pixel regions for a range of number of processors, and for the four models. 134
Figure 8.2: Block diagram of the Pixel-Planes 5 system. 136
Figure 8.3: Pixel-Planes 5 pipelining with an editing stage bottleneck. 138
Figure 8.4: Pixel-Planes 5 pipelining with a geometry stage bottleneck. 138
Figure 8.5: Pixel-Planes 5 pipelining with a rasterization stage bottleneck. 139
Figure 8.6: Pixel-Planes 5 pipeline example with a copy stage bottleneck. 139
Figure 8.7: Pixel-Planes 5 pipelining with a rasterization stage bottleneck, rasterizing two frames at once. 140
Figure 8.8: Model used in the multipass transparency example in figure 8.9. 141
Figure 8.9: Multipass transparency example showing a single column of pixels from the model in figure 8.8. 141
Figure 8.10: Sample Pixel-Planes 5 frame times for the Terrain model using 18 SGPs (19 GPs) and 8 renderers for one high resolution run and three low resolution runs. 143
Figure 8.11: Pipelining example when rasterization of individual regions is the bottleneck. 143
Figure 8.12: Pixel-Planes 5 performance when displaying the PLB Head model (containing 59,592 triangles) at high resolution for different numbers of SGPs and renderers. 144
Figure 8.13: Pixel-Planes 5 performance when displaying the PLB Head model (containing 59,592 triangles) at low resolution for different numbers of SGPs and renderers. 144
Figure 8.14: Pixel-Planes 5 performance when displaying the Terrain model (containing 162,690 triangles) at high resolution for different numbers of SGPs and renderers. 145

Figure 8.15: Pixel-Planes 5 performance when displaying the Terrain model (containing 162,690 triangles) at low resolution for different numbers of SGPs and renderers. 145
Figure 8.16: Pixel-Planes 5 performance when displaying the CT Head model (containing 229,208 triangles) at high resolution for different numbers of SGPs and renderers. 146
Figure 8.17: Pixel-Planes 5 performance when displaying the CT Head model (containing 229,208 triangles) at low resolution for different numbers of SGPs and renderers. 146
Figure 8.18: Pixel-Planes 5 performance per renderer when displaying the PLB Head model (containing 59,592 triangles) at high resolution for ratios of SGPs to renderers of 0.5 to 4. 147
Figure 8.19: Pixel-Planes 5 performance per renderer when displaying the PLB Head model (containing 59,592 triangles) at low resolution for ratios of SGPs to renderers of 0.5 to 4. 147
Figure 8.20: Pixel-Planes 5 performance per renderer when displaying the Terrain model (containing 162,690 triangles) at high resolution for ratios of SGPs to renderers of 0.5 to 4. 148
Figure 8.21: Pixel-Planes 5 performance per renderer when displaying the Terrain model (containing 162,690 triangles) at low resolution for ratios of SGPs to renderers of 1 to 4. 148
Figure 8.22: Pixel-Planes 5 performance per renderer when displaying the CT Head model (containing 229,208 triangles) at high resolution for ratios of SGPs to renderers of 0.5 to 4. 149
Figure 8.23: Pixel-Planes 5 performance per renderer when displaying the CT Head model (containing 229,208 triangles) at low resolution for ratios of SGPs to renderers of 0.5 to 4. 149
Figure 8.24: Renderer utilization for MEGA using the current static assignments, between-frame assignments with at most 4 regions per processor, and between-frame assignments without limiting the number of regions per processor. 152
Figure 8.25: Performance comparison of Pixel-Planes 5 and the Intel Touchstone Delta. 155
Figure 8.26: Number of extra regions rendered when region bottlenecks are removed by splitting the region's triangles over more than one renderer averaged over the sequence of frames for the four models, and for different numbers of renderers. 157
Figure A.1: Selected frames from the PLB Head model sequence. 166
Figure A.2: Selected frames from the Terrain model sequence. 168
Figure A.3: Selected frames from the CT Head model sequence. 170
Figure A.4: Selected frames from the Polio model sequence. 172
Figure A.5: Selected frames from the Well model sequence. 174
Figure C.1: Simulated processor utilization for the PLB Head model and static region assignments. 180
Figure C.2: Simulated processor utilization for the PLB Head model using once-per-frame region assignments. 180
Figure C.3: Simulated processor utilization for the PLB Head model using work-queue region assignments. 181
Figure C.4: Simulated overlap factor values for the PLB Head model. 182
Figure C.5: Simulated processor utilization for the Terrain model using static region assignments. 184

Figure C.6: Simulated processor utilization for the Terrain model using once-per-frame region assignments. 184
Figure C.7: Simulated processor utilization for the Terrain model using work-queue region assignments. 185
Figure C.8: Simulated overlap factor values for the Terrain model. 186
Figure C.9: Simulated processor utilization for the CT Head model using static region assignments. 188
Figure C.10: Simulated processor utilization for the CT Head model using once-per-frame region assignments. 188
Figure C.11: Simulated processor utilization for the CT Head model using work-queue region assignments. 189
Figure C.12: Simulated overlap factor values for the CT Head model. 190
Figure C.13: Simulated processor utilization for the Polio model using static region assignments. 192
Figure C.14: Simulated processor utilization for the Polio model using once-per-frame region assignments. 192
Figure C.15: Simulated processor utilization for the Polio model using work-queue region assignments. 193
Figure C.16: Simulated overlap factor values for the Polio model. 194
Figure C.17: Expected relative time for the PLB Head model using static region assignments. 197
Figure C.18: Expected relative time for the PLB Head model using once-per-frame region assignments. 197
Figure C.19: Expected relative time for the PLB Head model using work-queue region assignments. 198
Figure C.20: Expected relative time for the Terrain model using static region assignments. 199
Figure C.21: Expected relative time for the Terrain model using once-per-frame region assignments. 199
Figure C.22: Expected relative time for the Terrain model using work-queue region assignments. 200
Figure C.23: Expected relative time for the CT Head model using static region assignments. 201
Figure C.24: Expected relative time for the CT Head model using once-per-frame region assignments. 201
Figure C.25: Expected relative time for the CT Head model using work-queue region assignments. 202
Figure C.26: Expected relative time for the Polio model using static region assignments. 203
Figure C.27: Expected relative time for the Polio model using once-per-frame region assignments. 203
Figure C.28: Expected relative time for the Polio model using work-queue region assignments. 204

LIST OF ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
AGP	After Geometry Processing
ALU	Arithmetic Logic Unit
API	Application-Programmer Interface
ASCII	American Standard Code for Information Interchange
AVS	Application Visualization System
BBN	Bolt Beranek and Newman
CM	Connection Machine
CPU	Central Processing Unit
CRT	Cathode Ray Tube
CT	Computed Tomography
DAC	Digital to Analog Converter
FIFO	First In First Out
Gbyte	1,073,741,824 (2^{30}) bytes when specifying storage sizes or amounts; 1,000,000,000 bytes otherwise
GP	Graphics Processor
GPCG	Graphics Performance Characterization Group
HIPPI	High Performance Peripheral Interconnect
Hz	hertz
I/O	Input/Output
K	1,024 when specifying storage sizes or amounts; 1,000 otherwise
Kbyte	1,024 bytes when specifying storage sizes or amounts; 1,000 bytes otherwise
LCD	Liquid Crystal Display
Mbyte	1,048,576 (2^{20})bytes when specifying storage sizes or amounts; 1,000,000 bytes otherwise
MGP	Master Graphics Processor
MHz	1,000,000 hertz
MIMD	Multiple Instruction Multiple Data
MIPMAP	*Multum in parvo* (many things in a small place) map, a multiple resolution texture map
msec, ms	millisecond
µsec, µs	microsecond
NASA	National Aeronautics and Space Administration
NCGA	National Computer Graphics Association
NP	Non Polynomial

NTSC	National Television Standards Committee
NUMA	Non-Uniform Memory Access
PC	Personal Computer
PHIGS	Programmer's Hierarchical Interactive Graphics Standard
PLB	Picture Level Benchmark
PPHIGS	Pixel-Planes Hierarchical Interactive Graphics System
RGB	Red, Green, and Blue
SAGE	Systolic Array Graphics Engine
sec	second
SGP	Slave Graphics Processor
SPEC	Standard Performance Evaluation Corporation
SIMD	Single Instruction Single Data
UNC	University of North Carolina
VLSI	Very Large Scale Integration

LIST OF SYMBOLS

α	screen aspect ratio
α_r	region aspect ratio
A	area of screen (number of pixels)
A_r	area of a region
a	number of pixels in a given triangle
$\bar{a}$	average number of pixels in each triangle
a_{bb}	number of pixels inside the intersection of a triangle's bounding box and a region
c	fraction of triangles that must be redistributed each frame in sort-first algorithms (amount of coherence)
$C(g)$	communication time when using g groups
$c(N_g)$	communication time for processors in a group of size N_g
f	fan-in (number of incoming messages) in summing tree
G	granularity ratio, r/N
g	number of groups
g_h	number of groups in horizontal dimension of a regular grid of groups
g_v	number of groups in vertical dimension of a regular grid of groups
H	height of a region
h	height (number of scanlines crossed) of a triangle
$\bar{h}$	average triangle height (number of scanlines crossed)
h_{bb}	number of scanlines both inside a triangle's bounding box and a region
$l_{broadcast}$	latency to broadcast region assignments to all processors
l_{gather}	latency for gathering per-region costs
m	number of messages
m_1	number of messages sent and received per processor when using one-step redistribution
m_{1_agp}	number of messages sent and received per processor after geometry processing when using one-step redistribution
m_{1r}	number of messages received per processor in the first step of two-step redistribution
m_{1reg}	number of messages sent per processor when using one-step stream-per-region redistribution
m_{1r_agp}	number of messages received per processor after geometry processing in the first step of two-step redistribution
m_{1s}	number of messages sent per processor in the first step of two-step redistribution
m_{1s_agp}	number of messages sent per processor after geometry processing in the first step of two-step redistribution

m_{2r}	number of messages received per processor in the second step of two-step redistribution
m_{2r_agp}	number of messages received per processor after geometry processing in the second step of two-step redistribution
m_{2s}	number of messages sent per processor in the second step of two-step redistribution
m_{2s_agp}	number of messages sent per processor after geometry processing in the second step of two-step redistribution
N	number of processors
N_G	number of geometry processors
N_g	number of processors in a group
N_R	number of rasterization processors
n	number of triangles
n_1	number of triangles sent and received per processor when using one-step redistribution
n_{1r}	number of triangles received per processor in the first step of two-step redistribution
n_{1r_agp}	number of triangles received per processor after geometry processing in the first step of two-step redistribution
n_{1s}	number of triangles sent per processor in the first step of two-step redistribution
n_{1s_agp}	number of triangles sent per processor after geometry processing in the first step of two-step redistribution
n_{1_agp}	number of triangles sent and received per processor after geometry processing when using one-step redistribution
n_{2r}	number of triangles received per processor in the second step of two-step redistribution
n_{2r_agp}	number of triangles received per processor after geometry processing in the second step of two-step redistribution
n_{2s}	number of triangles sent per processor in the second step of two-step redistribution
n_{2s_agp}	number of triangles sent per processor after geometry processing in the second step of two-step redistribution
n_{2s_buf}	number of triangles remaining in each reformatting buffer after geometry processing (only when using two-step redistribution)
n_{back}	number of back-facing triangles
n_{msg}	maximum number of triangles per message, $\lfloor s_{msg} / s_{prim} \rfloor$
n_{off}	number of triangles off-screen, i.e. outside the viewing frustum
n_{on}	number of on-screen triangles
n_{refmt}	number of triangles that a processor reformats
n_{refmt_agp}	number of triangles that a processor reformats after geometry processing
O	overlap factor, the average number of regions that triangles overlap
r	number of regions
r_h	number of regions in horizontal dimension of a regular grid of regions
r_v	number of regions in vertical dimension of a regular grid of regions
S	number of samples per pixel
s_{disp_tri}	size of a display (transformed) triangle, in bytes; same as s_{tri}
s_{full_samp}	size of a sample used in sort-last full, in bytes
s_{msg}	size of a message, in bytes
s_{tri}	size of a triangle, in bytes

s_{raw_tri}	size of a raw (untransformed) triangle, in bytes
s_{sparse_samp}	size of a sample used in sort-last-sparse, in bytes
t_1	frame time when using one-step redistribution
t_{1_agp}	frame time after geometry processing when using one-step redistribution
t_2	frame time when using two-step redistribution
t_{2_agp}	frame time after geometry processing when using two-step redistribution
t_{assign}	time to assign all regions to processors
t_{assign_1}	component of region assignment time that is independent of N
t_{assign_N}	component of region assignment time that scales with N
$t_{assign_r\log N}$	component of region assignment time that scales with $(r–N)\log_2 N$
t_{betw_fr}	frame time when using the between-frames region assignment style
t_{betw_stages}	frame time when using the between-frames region assignment style
$t_{broadcast}$	per-processor time to broadcast region assignments
t_{bucket}	time to determine buffer and copy triangle to it
t_{byte}	time to send and receive a byte
t_{byte_r}	time to receive a byte
t_{byte_s}	time to send a byte
t_{comp}	time to composite a pixel
t_{extra_comp}	extra time to composite a sort-last-sparse pixel compared to t_{comp}
t_{gather}	per-processor time to gather per-region costs
t_{geom}	time to transform, clip test, and light a triangle
t_{geom_back}	time to transform and clip test a back-facing triangle
t_{geom_off}	time to transform and clip test an off-screen triangle
t_{line}	time to do scanline interpolations and calculations
t_{msg}	time to send and receive a message
t_{msg_r}	time to receive a message (using a message handler)
t_{msg_s}	time to send a message
t_{pixel}	time to rasterize a pixel
t_{pre_xform}	time to transform a triangle to screen space
t_{tri}	time to set up triangle for rasterization
$t_{tri_receive}$	time to receive a triangle when in a 4096-byte message
t_{refmt}	time to reformat a triangle between messages
t_{reg_ovhd}	time to process each region (overhead)
t_{scan}	time to scan convert a pixel; does not include composition
t_{sort}	time to sort per-region costs
t_{sort_1}	component of per-region sorting time that is independent of r
t_{sort_r}	component of per-region sorting time that scales with r
$t_{sort_r\log r}$	component of per-region sorting time that scales with $r\log_2 r$
t_{work_queue}	frame time when using the work-queue region assignment style
U_G	geometry utilization, the fraction of time that a processor is busy during geometry processing

U_R	rasterization utilization, the fraction of time that a processor is busy during rasterization
W	width of a region
w	width of a triangle

CHAPTER ONE

INTRODUCTION

1.1 Overview

This dissertation was motivated by my work on the Pixel-Planes 5 system [Fuch89], a high-performance, parallel graphics supercomputer. The system was quite successful: in 1991, it demonstrated a new level of performance of 2.3 million on-screen, rendered triangles per second. This dissertation builds on my experience developing the system's parallel rendering algorithm in two ways.

First, I consider how new algorithms, ones similar to Pixel-Planes 5's basic rendering algorithm, can be used for interactive rendering on general-purpose parallel systems. I do this for a class of systems known as *multicomputers*, which are message-passing, multiple-instruction multiple-data (MIMD) parallel systems [Bell85]; this class includes the Pixel-Planes 5 system. I focus on systems with tens to hundreds of processors to find the limits of interactive polygon rendering on these types of systems, and because using many processors is harder and more interesting. Second, I make a more systematic analysis of the Pixel-Planes 5 rendering algorithm. This analysis includes a study of the system's performance and suggestions on how to improve the performance.

The appropriateness of interactive polygon rendering on multicomputers follows from the typical use of these systems. Most large systems are used for complex calculations, which typically produce so much data that the result must be viewed graphically. This *visualization* operation transforms the result of the calculations into a graphical representation. Interactive visualization is desirable [McCo87] because it should allow productivity improvements similar to the improvement gained from the switch from batch processing to time-sharing. I will focus on the visualization of large polygonal datasets, since small datasets can often be conveniently viewed on workstations. Large datasets, those containing more than 500,000 polygons, cannot usually be conveniently displayed on workstations. The data may not fit in the workstation's memory, or may take too long to transfer to the workstation. The data can be reduced by filtering or decimation, but these operations would result in a less accurate visualization.

1.1.1 Targeted Parallel System Architecture

The research uses a multicomputer that, when I started the research, was the most powerful system built: the Intel Touchstone Delta [Inte91a]. While shared-memory MIMD systems have become more popular than multicomputers, the largest parallel system currently being built is a multicomputer [Thom96]. Some definitions of "multicomputer" [Bell85, Atha88] include systems connected by local area networks, but I exclude those systems; they are more commonly categorized today as *distributed systems*. Distributed systems are likely to require different algorithms, since they have much higher communication costs. I exclude shared-memory systems, since their lower communication costs may also dictate using different algorithms.

I only consider systems with large-grain nodes, ones with workstation-class microprocessors and at least 8 Mbytes of memory per node. This restriction removes the cases where overcoming severe node-size constraints dominates the possible solutions (as severe memory constraints dominated programming in the 1960's). Besides the Touchstone Delta, some examples of commercial multicomputers with these characteristics are the Intel Paragon XP/S [Inte92], the Thinking Machines CM-5 [Hill93], and the Cray T3D [Kess93]. The Delta is described in more detail in section 1.5.

1.1.2 Targeted Applications

Some of the dissertation's design decisions are based on the characteristics of typical visualization applications. Techniques used by these applications include:

- Displaying isosurfaces from volume data by computing triangles with the marching cubes algorithm [Hans92, Lore87].
- Showing vector fields by injecting particles into the field and following them [Seth88] or calculating streamlines that simulate smoke streams traveling through the field [Kenw92, Hult92].
- Displaying values on a surface by modulating the surface color.

Other visualization examples can be found in the proceedings of the Visualization conference series [Vis92, Vis93, Vis94, Vis95].

These visualization applications have some common characteristics. First, the geometry often changes substantially from frame to frame, perhaps as the user adjusts the isosurface level or the simulation progresses through another time step. Second, the triangles in the model are often fairly uniform in size. Finally, the structure of the model is often simple, in that there are only a few independently moving objects. However, because all applications do not always have these characteristics, it is important that any algorithm have good overall performance.

In many applications, low display resolution (640x512) is acceptable. However, higher resolutions are usually preferred. Also, anti-aliasing is often needed to remove the distracting artifacts caused by rendering small triangles.

Traditionally, polygon rendering techniques include the rendering of a number of different primitives, including lines, polygons, triangle strips, etc. However, for simplicity I will only consider a single type of primitive: triangles. Nearly all the techniques described can be easily extended to handle a variety of simple primitives instead of only triangles. A few techniques will require additional work to handle complex primitives, such as triangle strips or curved surfaces, because those techniques work better if the complex primitives are divided into multiple smaller primitives. I will not consider other rendering methods, such as volume rendering, even though they are often used in visualization. Those methods require completely different approaches.

1.2 Thesis Statements

This dissertation's contributions can be summarized with the following thesis statements:

1. Redistribution of screen-space triangles is necessary for efficient parallel rendering for interactive visualization applications on multicomputers.
2. A new design methodology can identify promising new algorithms from this class, and then select the most efficient algorithm.

Some words and phrases may need explanation. *Efficient* refers to parallel efficiency: the speedup of a parallel implementation compared to a sequential implementation divided by the number of processors used. *Redistribution of screen-space triangles* means that an algorithm must communicate, or redistribute, screen-space triangles between processors. *Necessary* implies that redistributing screen-space triangles is the most efficient under sufficiently many conditions that it must be part of all efficient implementations. *Most efficient* means that a given algorithm has the highest average efficiency for a set of models and viewing configurations, with each run on a range of system sizes.

1.3 Analysis of Redistribution Choices

To achieve the highest performance, each step of a parallel rendering algorithm must have the associated data divided among multiple processors. I call such algorithms *fully parallel* algorithms. If a single processor handled all of the data for a portion of the rendering algorithm, the limited communication bandwidth of a single general-purpose multicomputer node would limit the overall performance when more than a handful of processors is used. This is true even at the beginning of the pipeline where the smallest amount of bandwidth is required.

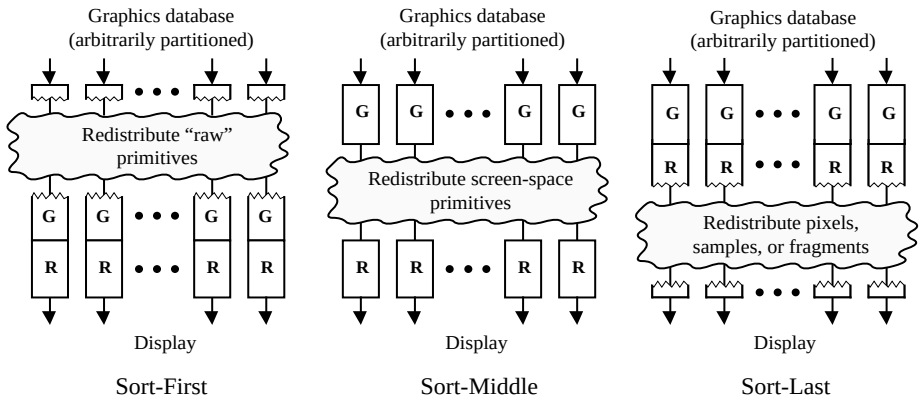


Figure 1.1 The three algorithm classes. Key: G = Geometry processing (transformation, clipping, lighting, etc.), R = Rasterization (from [Moln94]).

All fully parallel rendering algorithms must have a redistribution step, a step where data is reorganized among the processors. This can be seen by noting that, at the end of the graphics pipeline (during the hidden-surface elimination), fully parallel algorithms can only divide the data involved by having each processor be responsible for some subset of the screen's pixels. Each processor must have received some representation of the triangles that overlap its pixels to perform the hidden-surface elimination. However, a given processor cannot start a frame with the triangles necessary for the end of the algorithm since the triangles' screen-space locations have not yet been calculated. Instead, the algorithms' first step must assume that the triangles are divided among the processors without regard to the screen space location. Thus, a redistribution step is necessary, since the criterion for dividing the triangle data among the processors is different for the first and last algorithm steps. The redistribution can occur at various stages of the rendering algorithm. We will discuss this choice using a classification of fully parallel algorithms described in [Moln94].

The three main algorithm classes are shown in figure 1.1. The *sort-first* class of algorithms computes the triangles' screen-space locations and then redistributes untransformed, or raw, triangles (the transformed vertices may be sent as well, or discarded). *Sort-middle* algorithms redistribute triangles that have been transformed to screen-space. *Sort-last* algorithms redistribute pixels or polygon fragments, which can only be done after scan conversion has started. I consider two sort-last variants. The first, *sort-last-sparse*, has each processor redistribute only the pixels or samples generated by the scan conversion of its triangles without performing any hidden-surface elimination among those pixels or samples. The second, *sort-last-full*, has each processor first perform hidden-surface elimination among its triangles, and then has each processor redistribute the entire screen's pixels.

More details about the algorithm classification appear in Chapter 3. The classes are evaluated by calculating the amount of overhead required by all algorithms within each class. The overhead consists of two parts: the amount of communication required, and the amount of work performed that would not be done in a serial implementation, such as having to perform the rasterization setup for a given triangle on more than one processor. Since the overhead calculations cover classes of algorithms, they are fairly abstract, and do not account for load imbalances or the amount of global communication. Those two topics are covered in other discussions.

Comparing the overhead calculations for the four algorithm classes gives the following results. As will be shown in section 3.3.1, a straightforward implementation of sort-first requires between 30 and 120% more overhead computations than a sort-middle implementation, and requires 45 to 64% more communication; the values vary according to the assumed triangle size and number of screen subdivisions. However, sort-first algorithms may be able to take advantage of the frame-to-frame coherence of the triangles' locations on the screen and reduce the communication overhead, which may make them faster than sort-middle algorithms. Sort-first algorithms can only be faster when there is coherence in the retained geometry to exploit, which can only happen when the triangle data is relatively unchanged between frames. However, exploiting this coherence increases the cost of allowing changes to the geometry. This cost is expected to be so high that sort-first algorithms will be slower when extensive changes are made to the geometry.

Because visualizations applications commonly make extensive changes, a sort-middle implementation is preferable for an overall high-performance algorithm.

Under certain conditions, sort-last-sparse exhibits lower overhead than sort-middle. Since sort-last-sparse redistributes every pixel in each triangle, its redistribution cost is smaller when rendering small triangles. The overhead calculations in section 3.3.2 indicate that sort-last-sparse algorithms have less overhead than sort-middle algorithms when the average triangle covers fewer than 4.8 pixels. My sample datasets (described in Appendix A) have average triangle sizes near this value when they are rendered at a 640x512 resolution (the average sizes range from 2.9 to 6.7 pixels). The number of pixels or samples per triangle is larger than the crossover value when using a 1280 by 1024 screen resolution, or when performing anti-aliasing. Thus, sort-middle algorithms have lower overhead than sort-last-sparse algorithms in most of my cases.

Sort-last-full is different from the other classes in that the amount of per-processor communication required does not depend on the triangle size or the number of triangles. Instead, the communication amount depends on the screen resolution and the number of samples per pixel. This means that a performance increase does not require additional communication. However, sort-last-full requires much more communication compared to other algorithms running at interactive rates: a sort-last-full algorithm redistributes 2.6 Mbytes per processor per 640x512 frame, which is the same amount of data as 46,000 triangles. No single processor suitable for use in a general-purpose parallel system can today render 46,000 triangles per frame at fully interactive rates; that figure is larger than the number of triangles that can be rendered each second. The large communication overhead overwhelms the lesser amount of the other overhead component, repeated computation, required by sort-last-full.

1.4 Sort-Middle Design Methodology

Chapters 4 and 6 support the second thesis statement by describing a design methodology for sort-middle algorithms. The methodology consists of a series of design choices, and a procedure to make each choice. The choices can be divided into three categories. The first category determines the overall structure of the algorithm, and the second category determines how the rasterization is partitioned among the processors. The choices in these two categories interact, so not all combinations of choices are feasible. The feasible combinations are shown as a taxonomy in figure 1.2. The third category of choices is independent of the other choices, and determines how the geometry processing (the transformation, clipping, and per-vertex lighting) is divided among the processors.

For all three categories of choices, the methodology description is accompanied by an application of the methodology to one system, the Intel Touchstone Delta. The methodology is also used in Chapter 8 to analyze the Pixel-Planes 5 system. The design methodology is not exhaustive, since other reasonable algorithms are possible. The first category of design choices is fairly comprehensive, whereas the other two categories cover a small fraction of the overall spectrum of reasonable algorithms. An overview of this methodology and the associated implementation has been previously published [Ells93, Ells94].

1.4.1 Design Choices for the Overall Algorithm Structure

I consider in Chapter 4 four design choices that influence the algorithm's overall structure:

- which processors should run which graphics pipeline stages?
- when should the pipeline stages be run?
- which redistribution method should be used?
- when should the portions of the screen, the screen *regions*, be assigned to processors?

Sort-middle algorithms have three stages: geometry processing, redistribution, and rasterization. The first design choice is to determine the partitioning strategy: which processors run which stages. While all processors must participate in the redistribution, one approach would be to have all the processors perform both the geometry processing and rasterization. The second approach would be to have a portion of the processors perform the geometry processing, and the remainder perform the rasterization. The first approach is better since it avoids having to reassign processors between the stages to keep the workloads balanced as time required for the geometry processing changes relative to the time required for

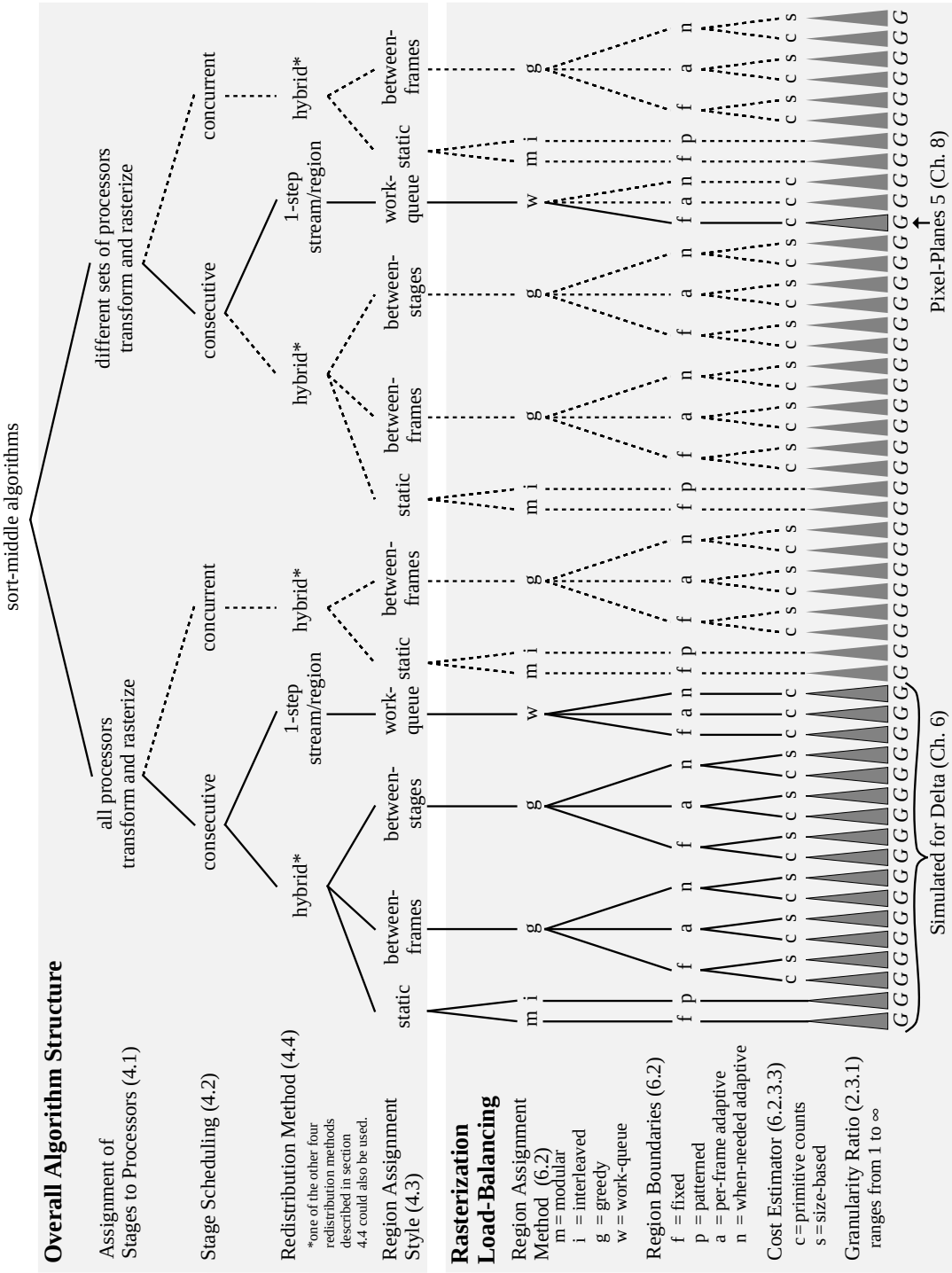


Figure 1.2 Taxonomy of the sort-middle algorithms encompassed by the design choices considered in the dissertation. The choices examined in the dissertation are indicated with solid lines. The numbers indicate the section containing the description of the design choice.

rasterization. However, the approach does increase the overhead, since each stage must be partitioned among more processors.

The second design choice is how to schedule when each stage should run. In order to maximize the overlap of communication and computations, the redistribution stage should be performed throughout the frame.

Given this determination, the remaining decision is whether to run the geometry processing and rasterization consecutively, one after the other, or concurrently, switching between the stages every few triangles. Running the stages consecutively gives more region assignment choices, as shown in figure 1.2. It also allows a higher amount of overlapped communication because the transformed triangles are generated earlier in the frame. However, it requires enough memory to store the entire frame's transformed triangles. Running the stages consecutively is the better option since it allows the next design choice's more desirable options to be used.

The third design choice is the redistribution technique. The redistribution is global: each processor sends transformed triangles to all other processors. The triangles are sent as a series of messages instead of a single large message so the redistribution can start before the end of the geometry processing, and to simplify memory management. Two techniques are possible:

- *stream-per-processor* redistribution: each processor sends a single series of messages to each other processor. Each message may contain triangles for several regions.
- *stream-per-region* redistribution: each processor sends one or more series of messages to every other processor, sending a series of messages for every region assigned to the other processor. Each message contains triangles for a single region.

The first technique requires a smaller minimum number of messages, which makes a difference when each processor sends only a handful of triangles. (The number of messages is determined by the maximum message size when the processors exchange many triangles.) Since each message has an associated cost (the *per-message overhead*), sending fewer messages will be faster. On the other hand, the second technique makes it much simpler to rasterize one region at a time, thus saving memory, since each message has triangles for a single region. The two techniques will respectively send a minimum of N^2 and rN messages, where N is the number of processors and r is the number of regions.

Modifications of the above techniques must be used with hundreds of processors, because the minimum number of messages can be quite large. Instead of sending the triangles directly to their final destination (called *one-step redistribution*), they are routed via intermediate processors (*two-step redistribution*). An example of this modified technique on a 2D mesh is to have the processors first exchange messages with the other processors in the same row, rearrange the triangles into new messages, and then exchange messages with the other processors in the same column. While this technique requires the data to be sent twice, doubling the per-byte communication costs, it reduces the minimum number of messages from N^2 to $2N\sqrt{N}$ for stream-per-processor redistribution, and from rN to $2N\sqrt{r}$ for stream-per-region redistribution.

No one redistribution technique gives the best performance. The one-step stream-per-region technique is best when using up to ten processors, since it reduces memory usage. With more processors, the one-step stream-per-processor technique is better, since it has less per-message overhead. When using more than 100 processors, the two-step stream-per-processor technique should be used to further reduce the per-message overhead. An implementation can automatically switch between the latter two techniques by evaluating a performance model at run time and selecting the faster one.

The last design choice that affects the overall algorithm structure involves the rasterization load-balancing algorithm. Sort-middle algorithms balance the rasterization loads by dividing the screen into regions, and assigning the regions to processors. The number of regions created is usually much greater than the number of processors. Each region is assigned to a single processor; each processor works on one or more regions. This design choice specifies one part of the load-balancing algorithm: the time that the region-processor assignments are made. The assignments can be made:

- dynamically during rasterization, assigning new regions to processors as they finish earlier ones; also known as a *work-queue* approach.
- once per frame, between the geometry processing and rasterization stages *(between-stages)*.
- once per frame, between successive frames *(between-frames)*.
- statically, at system initialization.

These four *region assignment styles* are ordered according to the cost of calculating and communicating the assignments as well as the evenness of the assigned processor loads. The first assignment style has the

highest costs as well as the most evenly balanced loads. The goal is to find the most efficient style: the one that has the best tradeoff between the cost to make the assignments and the higher processor utilization from the more evenly balanced workloads. Because the assignment costs increase with the number of processors, simulation studies show that no one assignment style is best. I show in Chapter 6 that the dynamic region assignment style is the best with two or four processors, that the between-frames once-per-frame is the best with more than four but less than 512 processors, and that the static assignment style is the best with 512 processors. The breakpoints between the different styles change with the size of the model.

1.4.2 Design Choices for Load-Balancing the Rasterization

The second group of design choices determine the load-balancing algorithm for the rasterization stage. The design choices are described in Chapter 6, and are:

- the method used to assign regions to processors.
- the procedure for calculating the region boundaries.
- the method used to compute an estimate of the time to rasterize a region.
- the *granularity ratio*, the number of regions per processor.

Since these design choices interact with the region assignment style, not all combinations are possible; figure 1.2 shows the possible combinations. The interactions make the evaluation of individual choices difficult, so all combinations were evaluated; the results appear at the end of this section. A terminology note: region assignment *styles*, discussed in the previous section, are not the same as the region assignment *methods* discussed in this section. The first term refers to the time that the assignments are made, and the second refers to the algorithm for making the assignments.

The first load-balancing design choice is the method for assigning regions to processors. The available choices are determined by the earlier choice of region assignment style (the time when the regions are assigned). When the regions are assigned statically, no information is available about the distribution of triangles on the screen. One static method assigns regions in a regular pattern, and a second assigns them sequentially to processors. Using a regular pattern gives more uniformly balanced loads, and gives slightly faster average performance. The two assignment styles that make region assignments once per frame (between-frames and between-stages) share one assignment method. That method (called *greedy*) gathers an estimate of the time to rasterize each region, and then makes greedy assignments: it steps through regions one-by-one, from the most time-consuming to the least, at each step giving the region to the processor that is most lightly loaded. The *work-queue* assignment method used with the work-queue assignment style is somewhat similar: the regions are sorted according to an estimate of the rasterization time, and then assigned from most time-consuming to the least. However, the actual assignments are performed as processors become idle, so the time estimates are only used to start the most time-consuming regions first.

The second design choice is the method to be used for calculating the region boundaries. The regions should be approximately square and the same size, since these properties minimize the probability that a triangle overlaps more than one region and must therefore be handled more than once. I only consider regular region boundaries, where the boundaries form a grid, cutting the screen from top to bottom and left to right. This constraint simplifies the per-triangle region classification; others [Whel85, Whit92] have tried and rejected more complex region boundaries. Given these desiderata, I consider three types of region boundaries:

- fixed region boundaries.
- adaptive region boundaries adjusted every frame.
- adaptive region boundaries adjusted when necessary.

The fixed region boundaries are calculated at system initialization. The per-frame adaptive region boundary method adjusts the region boundaries each frame so that the rasterization work is the same for each row and column of regions. The boundaries are adjusted using the per-region timing estimates that were gathered to perform the region assignments. The adjusted boundaries are used for the *next* frame.

The intent is to adjust the boundaries so each region has about the same amount of work, which should better balance the loads for a given granularity ratio compared to equally sized regions. The adaptive method does balance the loads better for a given granularity ratio. However, the adaptive method has higher overhead due to triangles that overlap multiple regions because the regions are small near congested areas of the screen. The adaptive region method is faster because the additional overhead is not large enough to outweigh the higher efficiency.

I tried the third region boundary method to avoid an inaccuracy with the per-frame adaptive boundary method. The per-frame method adjusts the boundaries every frame, which means that the current region time estimates do not correspond to the new region boundaries. This is a problem when the regions are assigned between successive frames, since the estimates are used to assign regions to processors. Although the timing estimates can be adjusted for the new boundaries, the adjustment is done as if work is uniformly distributed across each region. Since the distribution is not uniform, the adjustment process adds error to the timing estimates, which results in less well-balanced processor loads. The third region boundary method only adjusts the boundaries when one region takes so long to rasterize that it determines the overall rasterization time. Under certain conditions, this method only adjusts the regions every few frames, which thus often avoids the approximation error. The when-needed adaptive method improves performance over the per-frame adaptive method.

The third load-balancing design choice is part of the region assignment method. When regions are actively assigned, they must be assigned using some estimate of the time to rasterize each region. I consider two possible estimates: one uses a count of the number of triangles overlapping the region, and the other is based on the size of each triangle. The latter gives better estimates, but takes longer to compute and only gives at most a 3.4% speedup.

The last design choice is the granularity ratio, which specifies the number of regions per processor. I use a ratio because the number of regions must scale with the number of processors to avoid having too many regions when using a few processors, and too few regions with many processors. Using more regions makes the loads more evenly balanced, but also increases the communication and per-triangle rasterization setup overhead.

I evaluated the combinations of design choices using a simulator together with a model of the algorithm's performance. The simulator calculates the average amount of time that the processors are kept working during rasterization, as well as the average number of regions overlapped by the triangles. These two values are used in the performance model, which calculates the overall frame time.

This evaluation approach may make it easier to identify the best algorithm for other systems, because the simulations may not have to be repeated. Instead, the existing simulation results can be used in a performance model for the new system. The simulation results can be reused if, like the Delta, the new system's rasterization cost is primarily due to per-triangle costs instead of per-pixel costs, and if the per-region rasterization overhead is small compared to the other rasterization costs.

The performance model shows that the overall best load-balancing algorithm is a hybrid that uses two region assignment styles. When using up to 512 processors, the best algorithm uses four adaptively sized regions per processor that are actively assigned between frames. With 512 processors, the algorithm should switch to regions that are statically assigned in a regular pattern. The static assignment method uses 12 regions per processor because static assignment requires more regions to balance the processor workloads. The point at which to switch between the algorithms can be calculated automatically by evaluating a performance model that calculates whether there is enough time to perform the active load balancing.

1.4.3 Geometry Processing Design Choices

The last group of design choices, described in Chapter 5, determines the load-balancing algorithm for the geometry processing. These choices relate to how a hierarchical display list retained by the graphics system (either a hardware or software system) can be partitioned among the processors and traversed in parallel. I do not examine load-balancing algorithms suitable for graphics systems using an *immediate-mode* interface, where the application retains the geometric data and gives the graphics library the database each frame, even though this type of interface is more suitable for visualization applications. I do not treat these algorithms because correcting the load imbalances is so expensive that it only helps severe imbalances, which I expect to be rare.

A hierarchical display list organizes the data set into pieces called *structures.* Each structure is made up of a series of *structure elements,* which can be primitives, as described earlier; *attributes,* such as color changes and transformation matrices; and *execute elements,* which cause another structure to be traversed in a manner similar to a procedure call. During traversal, the set of currently active attributes forms an *attribute state*. That state is inherited by a called structure. One key issue is that the attribute state semantics obtained with a single-processor traversal must be preserved when the display list is traversed in parallel.

One method for dividing the display list is to put entire structures on different processors; this method is called *distribute-by-structure*. A second method, *distribute-by-primitive,* divides each structure independently so all processors traverse a smaller version of each structure: a version with a portion of the primitives and attributes. The second method is discussed in more detail, because it allows the partitioning to be done statically instead of dynamically, and because a simple algorithm suffices to preserve the attribute state semantics.

The distribute-by-primitive method divides each structure by storing successive primitives on different processors. The attribute elements can be sent to all processors, or to only the ones that have primitives affected by the attribute. The first technique makes it straightforward to make changes to the display list, but usually requires that the processors traverse many superfluous attribute elements. With the second technique, it is difficult to edit the display list, but removing the superfluous attributes allows efficient operation with hundreds of processors.

The distribute-by-primitive method can either give primitives to processors sequentially or randomly. Assigning primitives sequentially can result in unbalanced loads when the number of processors is close to, for example, a dimension of a large triangle mesh, or the number of triangles into which each primitive is tessellated. For example, if a model with spheres tessellated into 16 triangles is distributed across 16 processors, all the triangles on each processor will face the same direction. When back-face culling is enabled, some processors will not have any displayed triangles, and will have much less work to do. This type of load imbalance occurs surprisingly often with machine-generated databases. The random method greatly reduces the chance of an unbalanced load at only a slight speed penalty in other cases.

1.4.4 Implementation to Validate the Design Choices

The sort-middle-algorithm design methodology was developed concurrently with a sort-middle implementation on the Intel Touchstone Delta. The implementation supplies timing figures for the simulator and performance model, and is used in Chapter 7 to validate the methodology. The implementation does not exactly match the most efficient algorithm identified by the methodology because the work on the methodology continued after completing the major experiments with the implementation. Measured frame times are, on average, within 15% of the ones predicted by the performance model. The largest mispredictions occur when the overall network bandwidth constrains the performance. The performance model does not include network limitations, so its predictions are incorrect when running at high rendering rates. The highest measured performance was 3.1 million triangles per second when using 512 processors and the largest model (806,400 triangles).

1.4.5 Systems with Specialized Rasterization Processors

Chapter 8 discusses how the design methodology analysis changes when applied to a new type of system: systems that use specialized rasterization processors. The main difference is that there is an efficiency-cost tradeoff: a less efficient and less costly architecture may have a better price-performance ratio than a more efficient and more costly one. The chapter illustrates how the methodology can be used to identify a rendering algorithm for a single architecture by analyzing Pixel-Planes 5 and showing how its existing rendering algorithms can be improved. The chapter also makes two comparisons between systems with and without specialized hardware: Pixel-Planes 5 and the Touchstone Delta. The first comparison is the rendering rates of the systems. The second comparison is the price-performance ratio of the two system's rasterization processors. For the same cost, the specialized Pixel-Planes rasterization processors deliver three times the performance of the Delta's general-purpose i860 processors.

1.5 Description of the Target System

To allow more concrete conclusions, many of the arguments in this dissertation are made with a specific system in mind: the Intel Touchstone Delta at the California Institute of Technology [Inte91a].

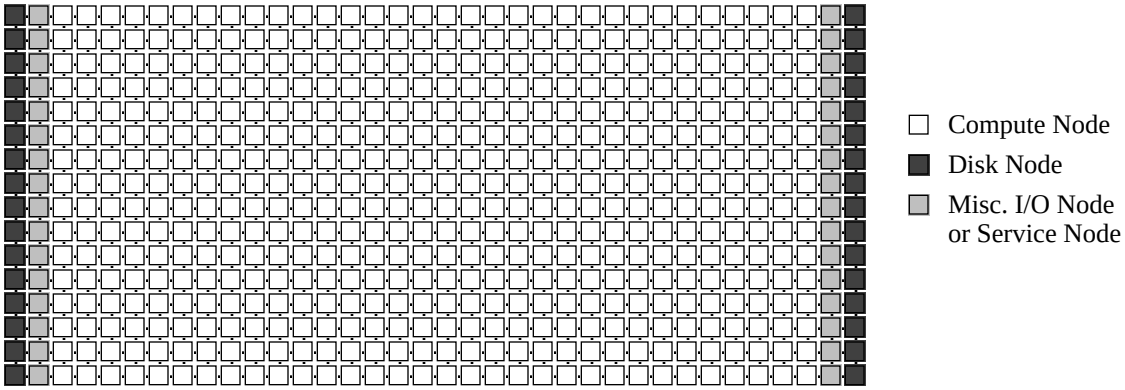


Figure 1.3 The Intel Touchstone Delta configuration.

The Delta is a one-of-a-kind prototype multicomputer, which is configured as a 16 by 36 2D mesh. The inner 16 by 32 nodes are the compute nodes, each having a 40 MHz Intel i860XR processor and 16 Mbytes of memory (see figure 1.3). The first and last columns of the mesh are the disk nodes, each with 2.8 Gbytes of storage. The second and next to last columns have service nodes, tape nodes, Ethernet nodes, and HIPPI nodes. The service nodes run a Unix-like operating system that allows users to run jobs on the compute nodes and do housekeeping tasks. The other I/O nodes are used for moving data on and off the system. There is no host processor, unlike some earlier multicomputers (e.g., the iPSC/860).

The 2D mesh network has bi-directional links in each dimension connecting the nodes. Each link can support 10 Mbytes/sec for long messages if the links have no conflicts [Litt92]. The network uses non-adaptive wormhole routing [Dall87]: messages are first routed in the long dimension, and then in the short dimension. The routing is performed in the network, so that processors on the intermediate nodes are not involved with the transfer, as in store-and-forward routing.

1.6 Structure of the Dissertation

Chapter 2 contains some background information and descriptions of previous work. Chapter 3 describes the rendering algorithm classification scheme in more detail, along with the analysis of the algorithm classes. Chapter 4 describes the sort-middle algorithm design choices that affect the overall algorithm structure (the top half of figure 1.2). Chapter 5 analyses the design choices for load-balancing the geometry processing stage, and Chapter 6 analyses the choices for load-balancing the rasterization (the bottom half of figure 1.2). Chapter 7 describes the implementation on the Touchstone Delta and the experiments made to validate the design methodology. Chapter 8 contains the analysis of algorithms to be used with specialized rasterization hardware, and Chapter 9 concludes with a summary and future work. The appendices contain descriptions of the sample datasets and how the timing values were measured, many charts of simulation data, and a glossary.

1.7 Expected Reader Background

I expect the reader to have an introduction to parallel processing and computer graphics, at a level of a single graduate-level course in each topic. The reader can gain the necessary graphics background by reading one of several books: [Roge85, Fole90, Fole94]. Several books provide an appropriate introduction to parallel processing, including the following: [Quin87, Braw89, Alma94, Kuma94].

CHAPTER TWO

BACKGROUND

This chapter contains some background information for the reader. It first introduces some terminology by describing the graphics pipeline, and also the four types of parallelism possible in graphics algorithms. Additional information on these topics can be found, respectively, in standard graphics texts [Fole90, Fole94 Roge85], and in Burke and Leler's annotated bibliography [Burk90].

The remaining sections describe earlier work in parallel graphics algorithms. Much of the work focuses on polygon rendering. I include some algorithms that render primitives other than polygons or perform ray tracing since many of their load-balancing algorithms are applicable to polygon rendering. Nearly all of the algorithms described are for MIMD systems, since that is the chosen architecture; the algorithms for SIMD systems are generally not applicable. The previous work first covers software algorithms for general-purpose systems, and then covers hardware algorithms and implementations. Because this dissertation focuses on software algorithms, they are more completely covered than hardware algorithms. The reader can find other surveys of parallel algorithms and architectures in [Burk90], [Whit92], and Chapter 18 of [Fole90].

2.1 The Graphics Pipeline

The graphics pipeline can be broken up several ways, depending on the type of shading algorithm and how it is to be performed. For Gouraud shading, the pipeline is as shown in figure 2.1a (this is similar to figure 8.3 in [Fole90]) The pipeline has the following stages:

- Traverse/Generate: if the graphics library stores the geometric model (using a retained-mode interface), the first stage traverses the retained model, processing state changes such as the transformation stack and the current color. If the application stores the model and uses a library with an immediate mode interface, the stage allows the user to generate the polygons using a series of procedure calls.

- Transform to Eye Space: transform the polygons to eye space, a space where the primitives have had all the modeling transformations applied and where the eye is always in the same place.

- Clip: clip the polygons to the hither plane and, if desired, to the far and side planes of the viewing frustum. The far plane clip is optional, and the side plane clipping need only be done if the rasterization technique requires it or if it is faster to do so. Polygons that are completely outside of the viewing frustum are rejected. If back-face culling is enabled, polygons with their back facing the eye are discarded here.

- Transform to Screen Space: perform the perspective division, and map the polygons to the appropriate window coordinates.

- Shade vertices: In Gouraud shading, the vertices are shaded according to the Gouraud shading model [Gour71]. The shading can be calculated using the dot product of the light vector(s) and normal vectors transformed into eye space, or the product of the light vector(s) back-transformed into modeling space and the untransformed normal vectors.

- Scan-convert: determine the interior of the polygon. This can be done by first calculating the first and last scanlines that the polygon crosses, and then, for each scanline, calculating the first and last pixels that are inside the polygon. The usual serial implementation loops for each pixel in the polygon and

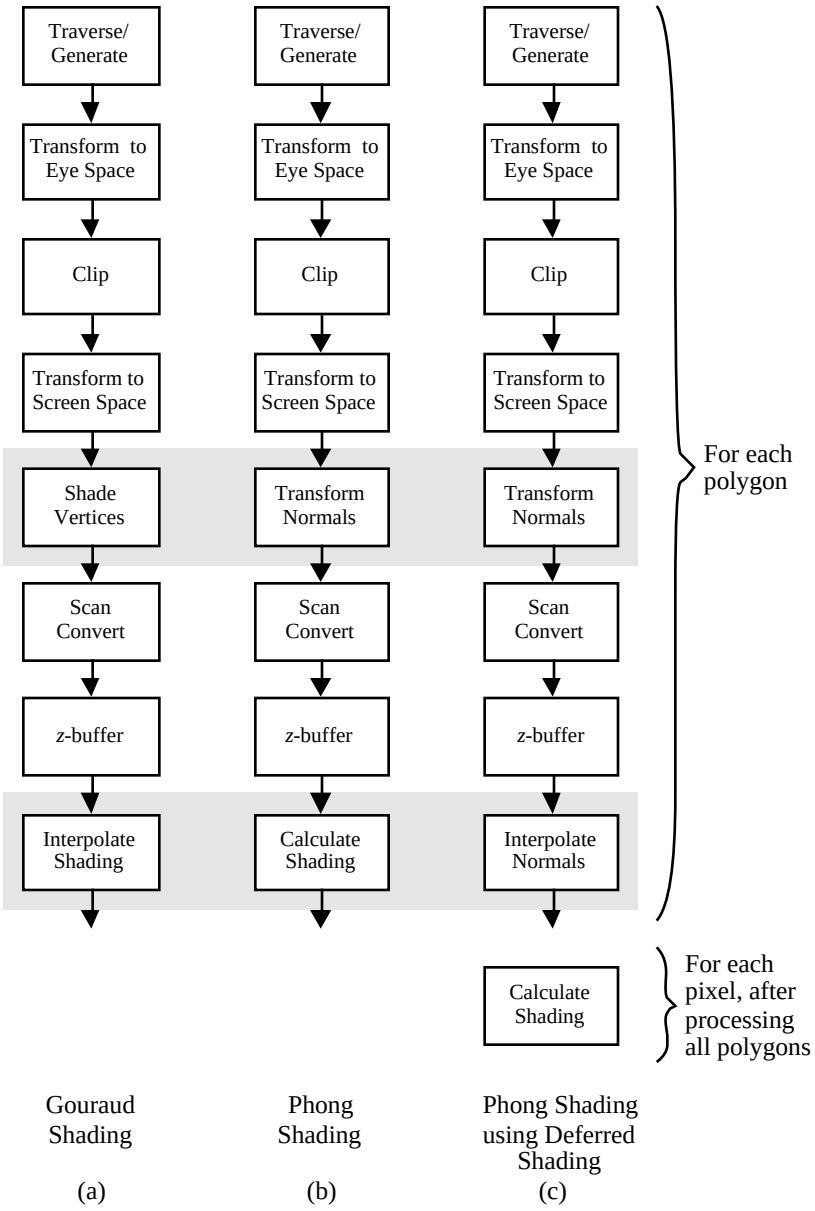


Figure 2.1 Example graphics pipelines for three types of shading.

does the scan conversion, *z*-buffering, and shading interpolation for each pixel before working on the next pixel.

- *z*-buffer: determine each pixel's visibility by comparing the pixel's depth with the stored *z* value.
- Interpolate Shading: interpolate the pixel's color values from the values calculated at the polygon vertices.

The first five steps together constitute the *geometry processing* part of the pipeline, and the last three steps constitute *rasterization*. When performing Phong [Buit75] shading and/or texture mapping, the last step is often modified to interpolate the normal vector and/or texture coordinates, and the shading is then calculated from the interpolated values (see Figure 2.1b).

However, some systems use a different algorithm for performing higher quality shading, using what is called *deferred shading;* see Figure 2.1c [Fuch89, Tebb90, Deer88]. This method interpolates the normal

vector and/or texture coordinates as before, but does not calculate the shading inside of the per-pixel scan conversion loop. Instead, the interpolated values, plus the texture identifier and the color of the polygon, are stored in per-pixel memory. After all of the polygons have been rasterized, the rasterization processor reads the stored values in each pixel, and computes the final color value. This method has the advantage that each pixel is only shaded once, instead of shading all pixels that pass the *z*-buffer test. Since it shades all of the pixels at once instead of just one polygon's pixels, this method has an advantage for some parallel systems. With a vector processor, this increases the vector length, boosting efficiency. With a SIMD processor, it increases the data size, which allows more processors to perform calculations instead of idling.

2.2 Types of Parallelism

Four types of parallelism can be used with the rendering process:

1. Functional Parallelism. The operations are performed by having different processors perform different functions. This method is also known as *operation parallelism*. Functional parallelism is used in pipelines of processors, where the operations to be performed are distributed among a set of different processors.

2. Frame Parallelism. Frame parallelism has different processors (or groups of processors) work on successive frames in an animation sequence or interactive session. The main advantage of frame parallelism is its linear speedup, as the usual performance is proportional to the number of processors. However, the performance increase is not accompanied by a reduction in latency, which is a drawback for interactive and real-time systems. This is less of a problem when the frame rate is high or the number of processors is small.

3. Image Parallelism. The screen is divided into regions, which are assigned to processors. Each processor then processes the primitives that overlap its regions. Primitives overlapping multiple regions are processed once for each. Image parallelism is often used for the rasterization part of the pipeline. The method for dividing the screen should handle the normal clustering of primitives on the screen so that the processors' loads are balanced, and it should also minimize having to process primitives more than once.

4. Object parallelism. The model is split up among the processors, and the processors perform operations on the primitives. The primitives can be split using random assignment, sequential assignment, or some other basis. Object parallelism is most often used for the initial portion of the pipeline because the splitting can be done without primitives' screen space locations.

Of the four types, frame parallelism is the least often used for interactive systems. More than one type of parallelism is usually used; some systems use all four types (e.g., the Silicon Graphics SkyWriter [SGI92]). Many systems use object parallelism for the front-end tasks (transformation, clipping, and lighting), and image parallelism for the rasterization and shading (e.g., [Whit92], [Croc93], and [Cox95]). Hardware systems usually use these two types of parallelism for the different tasks, but perform them on different processors, and thus also use functional parallelism (e.g., [Ghar88], the Silicon Graphics GTX [Akel88], Pixel-Planes 5 [Fuch89], and the Silicon Graphics RealityEngine [Akel93]). The next sections will show many of the different possibilities done to date.

2.3 Algorithms for General-Purpose Parallel Computers

The discussion is divided into three sections. The first two sections primarily describe algorithms organized around the traditional graphics pipeline; the first section describes image-parallel algorithms, and the second, object-parallel algorithms. Most of these algorithms only support a simple shading model, as they are optimized for speed. Also, most support Gouraud shading, although one also supports the Blinn [Blin77] and Cook-Torrance [Cook82] lighting models. The algorithms are not tailored for interactive use, although most of them will run at interactive rates when given an appropriate data set size and system size.

The third section contains a different class of algorithms: parallel ray-tracing algorithms. These compute high-quality images and support complex primitives. Both image- and object-parallel algorithms are covered.

It is difficult to compare the performance of these earlier works. Comparing triangles per second figures does not take into account differences in databases, rendering quality, or speed of the underlying machine.

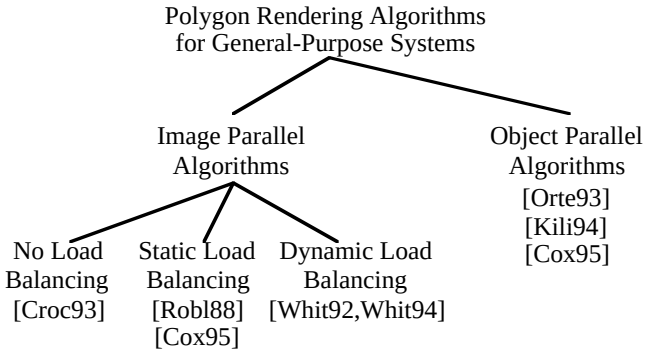


Figure 2.2 Taxonomy of the parallel polygon rendering algorithms for general purpose systems covered in this section.

Each can change the performance by at least a factor of ten. Comparing the speedups achieved over a single-processor implementation normalizes the computation power between different systems, but does not normalize the communication capacity, or the database and rendering quality differences. I will quote triangles per second figures for the interactive systems that use a simple shading model, and quote speedup figures for the non-interactive systems that use complex shading models. These figures are the most favorable for each type of implementation, and are the ones typically reported.

2.3.1 Image-Parallel Polygon Algorithms

The first four algorithms use image parallelism, where the screen is subdivided and the pieces are given to different processors. The screen subdivision can be regular subdivision, as shown in figure 2.3, where the screen is divided into a regular grid of regions. Or, the subdivision can be adaptive, where the regions are made smaller where the image is complex, and larger where it is simple. One important parameter is the number of regions. Several implementations use a constant number of regions per processor. I will call that constant factor the *granularity ratio*.

All screen subdivision methods have a redistribution step, where the polygons are routed to the processors rendering the regions that the polygon overlaps. A polygon must be routed to more than one processor when it straddles a region boundary and the regions are rendered by different processors. In an image-parallel implementation, part of the geometry processing classifies each polygon according to the regions it overlaps and places the transformed polygon in a data structure, or *bucket*, associated with each region. I call this process *bucketization*.

Roble. One of the early parallel polygon rendering systems was done by Roble on the iPSC Hypercube [Robl88]. This system uses an adaptive screen-space subdivision method. The regions are calculated by first using a regular grid of regions, with one region per processor, and counting the number of polygons in each region. The regions are then combined and split so that same number of regions remain, but with approximately the same number of primitives in each.

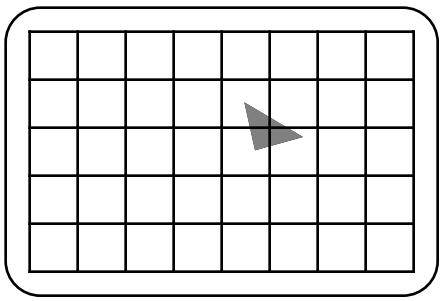


Figure 2.3 Regular screen subdivision. The triangle will be processed by the four processors that are assigned to the regions overlapping the triangle.

In this algorithm, the manager processor transforms the polygons and sends them to the node processors round-robin. The node processors calculate the number of their polygons in each of the initial regions and send the counts to the manager processor. The manager processor calculates the new region boundaries and broadcasts them to the node processors. The node processors redistribute the polygons, sending each polygon to all the processors that have a region overlapping it. Finally, the node processors use a scan-line algorithm to render their regions, and send the computed pixels to the frame buffer or disk.

The algorithm works well for a small number of processors, but becomes inefficient with more than 32 processors, since the load-balancing is done serially and can be a bottleneck. Also, the load-balancing method does not give especially high processor utilization (the percentage of time doing useful work) during the parallel rendering step—33 to 73% for the 16-processor cases.

Crockett and Orloff. Crockett and Orloff [Croc93] used a newer Intel MIMD system, the iPSC/860. They used a regular screen subdivision method, giving each processor the same number of scanlines to render. Since the amount of work in each scanline group can vary, this algorithm does not load-balance the rasterization process.

The algorithm starts by loading the data set onto the node processors, assigning polygons round-robin. The node processors transform and light the polygons, and decompose them into trapezoids with horizontal tops and bottoms (some tops and bottoms will have zero length). The trapezoids are redistributed to the appropriate processors and are rendered. The processors do the transformation and rendering semi-concurrently, switching from one to the other every few polygons. This reduces the memory requirements since buffering is only needed to hold a portion of a frame's trapezoids.

The implementers took care to optimize the polygon redistribution. The communication is done asynchronously to the computation to avoid waiting for the network. The communication is buffered, with the size of the buffers optimized to minimize the execution time. The algorithm description includes a performance model of the algorithm, which gives insight into the algorithm's behavior. A limitation of this algorithm is its omission of load-balancing, which reduces its performance: a test scene with 50,000 randomly-placed triangles runs at approximately 130,000 triangles per second when using 128 processors, while a realistic scene with 59,276 triangles runs at 100,000 triangles per second with 64 processors (with 128 processors, the algorithm runs at 90,000 triangles per second).

A later paper [Croc94] describes a version of the algorithm which redistributes spans instead of trapezoids. Since the decomposition of primitives into spans is done only once, the later version requires more communication and less computation. The new algorithm has higher performance, rendering nearly 300,000 triangles per second when running on a 192-processor Intel Paragon. Some of the performance increase is probably due to the Paragon's slightly faster processors and much faster communication.

Whitman. The next image-parallel polygon algorithm was implemented on the BBN Butterfly [Whit92, Whit94], a non-uniform memory access (NUMA) shared-memory system. Whitman investigated several methods of screen-space subdivision. All use dynamic scheduling, also known as work-queue scheduling. The subdivision methods are:

- scan-line decomposition: each processor renders a scan line at a time.
- regular rectangular regions, with different aspect ratios, and different numbers of regions per processor. He found that square regions and 12–24 regions per processor worked the best.
- adaptive decomposition. His method is somewhat similar to Whelan's (below). The method starts with a regular rectangular decomposition with 40 regions per processor, and counts the polygons in each region. The final regions are made by starting with a single full-screen region, and then repeatedly dividing the region with the largest number of polygons in half until there are 10 regions per processor. The value of 10 was determined experimentally (see figure 2.4).
- task adaptive. The screen is initially broken up into regular rectangular regions, and each processor works on a region at a time. When a processor is free and no more unassigned regions are available, the processor finds the processor that has the most work left undone, and takes half of it. Whitman experimented with different numbers of regions per processor, and found that two regions per processor worked the best.

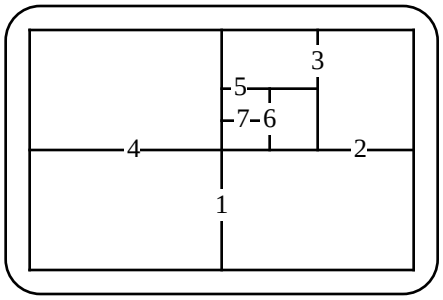


Figure 2.4 Whitman's adaptive decomposition with 8 regions. Each split is numbered by its order within the splitting.

All of the algorithms initially assign polygons to processors in round-robin order. They then bin-sort the polygons according to the regions they overlap, and place them into global shared memory. When a processor starts rendering a region it retrieves the associated polygons from global memory (see figure 2.5). Whitman experimented with leaving the polygons in global memory versus copying the polygons into local memory, and found the latter was faster due to reduced network contention.

Whitman's system achieved impressive parallel efficiencies: an average of 82% for 96 processors. This was due in part to the high quality rendering (the system produced images with anti-aliasing and Blinn shading [Blin77]); thus, the communication and load balancing overhead could be amortized over more calculations than would be if a simpler shading model was used. Also, shared memory systems allow low overhead communication, so load-balancing is somewhat cheaper than with most message-passing systems. However, the raw rendering rates are fairly low: up to 100,000 non-anti-aliased polygons per second using 96 processors.

Cox. In his dissertation, Cox [Cox95] describes three different parallel algorithms for the CM-5. The algorithms are parallel versions of the commercial RenderMan package [Upst89], a very high-quality, or photorealistic, renderer. The package supports curved surfaces as well as the traditional polygons, transparency, textures, and user-programmable shading.

Two of his algorithms do most of the work using image parallelism; a third uses object parallelism, and is described in the next section. The first two algorithms divide the screen into equal-sized, roughly square regions, and varied the number of regions per processor. Each processor is randomly assigned the same

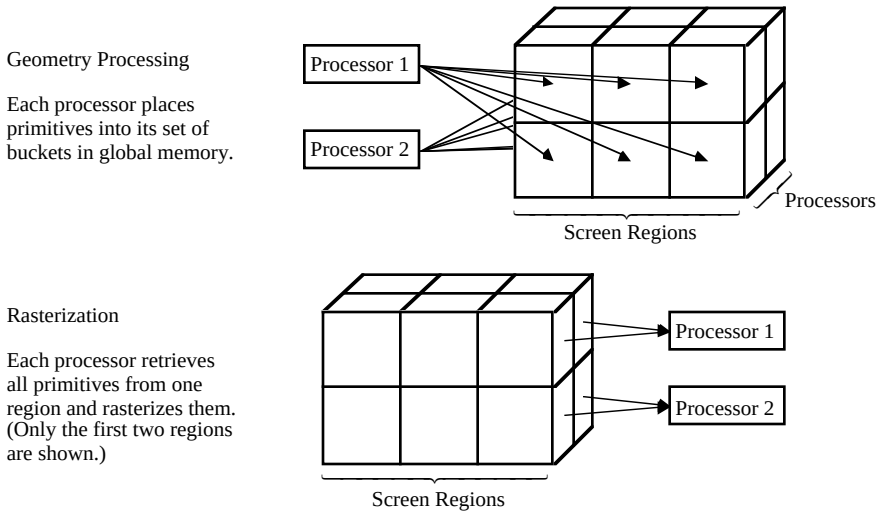


Figure 2.5 Data flow of Whitman's implementation. The box represents a 3D array of buckets in global shared memory. Two of the box's dimensions are for the 2D grid of regions, and the third is for the processors.

number of regions. In the first algorithm, the primitives (mostly curved surfaces) are initially assigned to processors *k* at a time; the assignment is either random or round-robin. Then, the processors transform the primitives to screen space, and send them to the appropriate processor(s). The final steps tessellate and shade the primitives, and then *z*-buffer them. The first algorithm reached a maximum speedup of 45 when using 512 processors. It was limited by the fairly large amount of repeated work and, to a smaller extent, load imbalances.

The second algorithm is a hybrid between the first algorithm and the third algorithm. The algorithm has the same initial steps as the first one. But, when a processor finishes its work from the initial assignment, it then "steals" work from other processors. It requests patches from still-busy processors, rasterizes them, and then returns the samples to the original processor. In most cases, the second algorithm was an improvement over the first, but it did not improve the maximum speedup.

2.3.2 Object-Parallel Polygon Algorithms

These algorithms do all of the work up to and including scan conversion using object parallelism before using another form of parallelism to perform the hidden surface elimination. The switch in parallelism methods means that samples and *z* values generated in the earlier stage need to be redistributed among the processors before they are combined, or composited, to create the final sub-image on each processor.

Cray Animation Theater. The polygon renderer for the Cray T3D system [Kili94] uses an image composition algorithm similar to the ones used by the hardware implementations described later in this chapter. It first divides the triangles arbitrarily among processors. Then, each processor renderers its triangles into a local color and *z*-buffer. Afterwards, the processors combine the local frame buffers by making a series of partial exchanges of the local color and *z* values with another processor so that each ends up with a few completely *z*-buffered scanlines (see figure 2.6). This is done by arranging the *N* processors into a conceptual hypercube with dimension $\log_2 N$. Then, the processors loop over each dimension in the hypercube. In each step, each processor sends half of its remaining color and *z*-buffer to the other processor in the current dimension, and composites its remaining buffer with the incoming data. One processor sends the top half of the remaining buffer, and the other sends the bottom half. The resulting smaller *z*-buffer is then used in the next iteration. At the end of the loop, the computed scanlines are broadcast to all processors using the algorithm in reverse, without composition. The broadcast is not strictly necessary; other methods could be used to transfer the image to the frame buffer. This renderer is currently the fastest one running on a general-purpose parallel processor, running at 4.5 million triangles per second with 256 processors.

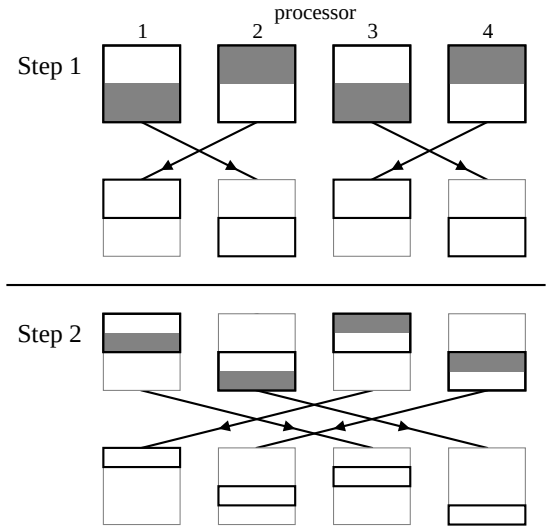


Figure 2.6 Binary-swap composition example with four processors. The gray rectangles indicate the portion of the image to be sent and the black outlines indicate the portion of the image stored by each processor.

Cox. Cox's third RenderMan implementation [Cox95] uses object parallelism for most of the work. The primitives are first distributed among the processors using the same method as the earlier algorithms. Then, each processor transforms, tessellates, shades, and samples its assigned primitives. The samples are then redistributed for the *z*-buffering using image parallelism. Each processor does the *z*-buffering for a constant number of approximately square regions; the regions are assigned randomly at the start of the frame.

Due to memory constraints, each processor can only receive samples for a limited number of regions at a time. This means that each processor starts the *z*-buffering of a few regions at the start of a frame. During a frame, a processor can only start *z*-buffering a new region when an old one finishes. The rendering must be synchronized with the *z*-buffering to ensure that only samples for the current regions are sent. A processor must wait if it generates samples for a region that is not currently being *z*-buffered.

Cox's measurements show that a large fraction of the time is spent synchronizing the per-region *z*-buffering. When combined with load imbalances as large as those of the first implementation, the maximum speedup is limited to two, even when using 64 or more processors. Thus, he found this strategy impractical.

Lee, Raghavendra, and Nicholas. A recent paper [Lee95] compared four methods for redistributing and combining the samples on the Intel Touchstone Delta. The earlier steps of the authors' implementations are the same as in the Cray Animation Theater: the triangles are first divided among the processors, and then each processor generates a full screen image of its triangles. All of the four methods carefully schedule the communication so that network contention is either reduced or completely avoided. The communication is performed by having each processor first exchange samples among processors in each column of the 2D mesh, combine the received samples, and then exchange samples among processors in each row. The details of the communication scheduling are too complex to be described here. A brief description of the methods follows:

1. Parallel Pipeline. Each processor sends a full screen image. The communication is broken up into a series of steps. In each step, each processor sends one portion of the screen, receives a different portion, and then composites the incoming data with the corresponding portion of its local frame buffer.

2. Parallel Pipeline with Bounding Boxes. This method is similar to the previous except that blank areas of the screen are not sent. The method only sends the portion of each region within a bounding box that contains all of the active pixels (pixels other than the background).

3. Direct Pixel Forwarding. This method only sends the active pixels. Each processor is given responsibility for an equal-sized region of the screen, and the active pixels are sent to it. The number of regions horizontally and vertically is the same as the number of columns and rows in the mesh of processors.

4. Direct Pixel Forwarding with Static Load Balancing. This method is the same as the previous method except that a different method of assigning pixels to processors is used, one that better balances the composition workloads. The screen is first divided horizontally by the number of columns in the mesh. Then, the scanlines in each column of pixels are assigned round-robin to the processors in the corresponding mesh, interleaving the scan lines among the processors. This balances the composition workloads better because each processor works on pixels distributed across the screen, which reduces the probability that one processor is given a region with too many active pixels.

The last method gives the best performance because it sends the smallest amount of data and also incorporates load balancing. When rendering Gouraud-shaded triangles, it runs at rates between 2.8 and 4 million triangles per second using 512 processors, depending on data set.

Ortega, Hansen, and Ahrens. While this algorithm [Orte93] was implemented on a MIMD system like the previous algorithms, it is very different as it is written using a data-parallel paradigm, like a SIMD system. However, it is an object parallel algorithm because the rasterization is done by dividing the data without regard to screen location. It was implemented on the CM-5, which has architectural and software support for data-parallel programming. Data-parallel programming uses the concept of virtual processors, where one programs as if there were a processor for each piece of data. Virtual processors are implemented by having each real processor store and process several pieces of data. Each real processor iterates over its set of virtual processor data when performing each operation.

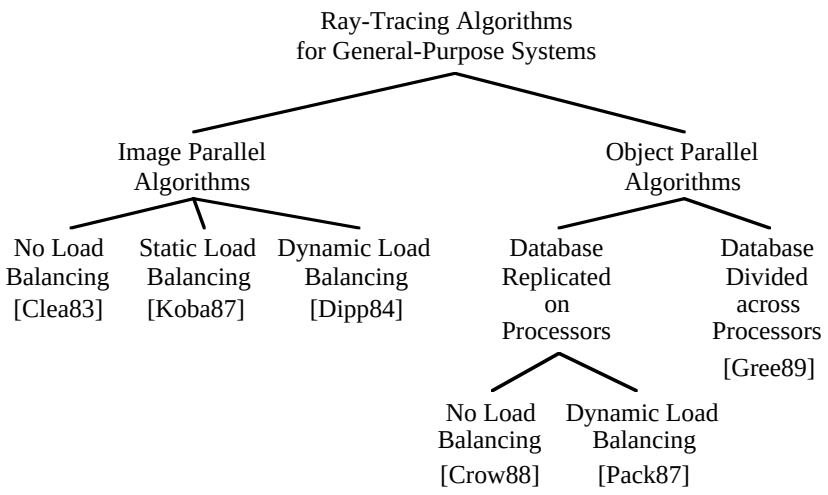


Figure 2.7 Taxonomy of the ray-tracing algorithm covered in this section.

Ortega et. al.'s algorithm works by first assigning each polygon to a distinct virtual processor. The processors transform the polygons, discard those outside the window, light the polygons, and scan convert the polygons into spans. The spans are routed to a second set of virtual processors, where the spans are clipped to the window, and then are interpolated to calculate the pixels. Finally, a third set of virtual processors, one for each screen pixel, is used for the *z*-buffering. Each processor is sent all the samples that overlap its pixel, and it determines the visible sample. The algorithm requires two communication steps, one for the transition from polygons to spans, and another for the transition from spans to pixels.

The basic algorithm works well when the polygons are nearly the same size, displaying nearly a million triangles per second, but its performance drops by a factor of ten when some larger polygons are included. An improved algorithm performs load-balancing steps between processing steps so the data can be moved from busy to free processors.

2.3.3 Ray Tracing Algorithms

There has been considerable interest in developing algorithms for parallel ray tracing that are suitable for general purpose parallel systems. Some of the algorithms use image parallelism like the polygon rendering algorithms mentioned above, while others use object parallelism. While many of these algorithms cannot be directly used for polygon rendering, some of their concepts can be reused, and thus I only report the earliest reported algorithm of each taxonomy type. Other algorithms can be found in [Burk90] and [Gree91].

Object-parallel ray tracing. In object-parallel ray tracing, the primitives are partitioned among the processors, and each processor intersects rays against its primitives. This is done by first dividing world space into volume elements (voxels), assigning one or more voxels to each processor, and giving each processor the primitives inside its voxels. Then, during rendering, rays are generated and sent to the processor that handles the first voxel encountered. As a ray progresses among the primitives, it is handled by the processor assigned to the associated voxel of world space. When a ray is traced into space handled by a new processor, it is sent to that processor.

Cleary et. al. [Clea83] first proposed this type of algorithm but did not perform load-balancing. Kobayashi et. al. [Koba87] used static load-balancing, giving each processor several voxels. Dippé and Swenson [Dipp84] proposed a custom architecture that used an algorithm with dynamic load-balancing. The latter algorithm's subdivision method cut the object space into an irregular 3D mesh. When a processor was overloaded it would move one of the corners of its voxel so that a neighboring processor would take on some of the load.

Image-parallel ray tracing. The other class of algorithms for ray tracing, the screen subdivision algorithms, can be divided into two subclasses: ones that do and do not replicate the model for every

processor. Replicating the model on every processor greatly simplifies the algorithm. Unlike polygon rendering, replicating the model is somewhat practical for ray tracing, as the models tend to be fairly compact because higher-order primitives (like patches and quadric objects) are often used, and because the slower speed of ray tracing makes large models less practical. One example of a image-parallel algorithm that replicates the model is one by Packer [Pack87] for an array of Transputers. This algorithm dynamically parcels out patches of the screen to processors, with each processor ray-tracing its patches independently. This work-queue approach automatically balances the workloads.

Another image-parallel algorithm is the one by Crow et. al [Crow88], which uses the first Connection Machine, the CM-1. The screen is first divided into 128x128 pixel regions. Each region is rendered by assigning a processor to each pixel, and then making a series of passes over the scene description. In each pass, each processor intersects its current ray with all of the primitives in the scene, finding the closest intersection. The first pass intersects the primitives with the ray starting from the processor's pixel, finds the closest primitive, and computes the ray reflected from the primitive. Later passes intersect the database with each processor's current reflected ray. This method did not perform load-balancing, instead using the large computational power of the machine for the algorithm's speed.

Other image-parallel algorithms do not replicate the model at each processor. Instead, these algorithms divide the model among the processors by dividing object-space into voxels, and distributing the voxels and associated primitives across the processors. When the processors trace rays through voxels, they retrieve the contents of voxels from the processor permanently storing them. To reduce communication, the voxels are cached: some number of the most recently used voxels are saved in local memory.

Green and Paddon [Gree89] used this method and achieved speedups of six with eight processors when at least 10% of the model could be stored at each node. They optimized the method by observing that a large percentage of the database queries were made to a small portion of the database. This observation led them to modify the caching strategy so that a portion of the cache storage is dedicated to statically store the most-often-referenced primitives. The modification improves the performance because accessing a static object is faster than searching the cache. The portion of the database to be stored statically is determined from statistics gathered when tracing a low resolution image; the primitives that are often used are then stored statically for the high resolution image. Their algorithm uses a work-queue to dynamically assign regions to processors.

While some of the parallel ray-tracing algorithms are similar to those used by the parallel polygon algorithms, it is hard to draw comparisons between the two classes of algorithms. To get its higher-quality shading, ray tracing requires much more computation per primitive than polygon rendering, which reduces the effects of overhead and communication. The additional computation implies that the best load-balancing methods should be more accurate, and also more expensive, than those used in parallel polygon rendering algorithms.

2.4 Hardware Graphics Systems

Hardware graphics systems have different constraints than software implementations using general purpose hardware. Their specialized design allows them to provide, for example, higher bandwidths than are generally found in general purpose systems. This difference means that image-parallel systems can use different screen subdivision methods, and also means that the higher bandwidths required by object-parallel systems are more practical. About half of the systems described have been or are being built; they are listed in the taxonomy in figure 2.8 with both their name and a reference to their description.

The image-parallel hardware systems use functional parallelism, where the front-end processing is done by one processor or set of processors, and the back-end processing is done on a separate processor or processors. This allows the processors to be specialized, which is useful because the work is also specialized: the front-end processing is floating-point-intensive while the back-end processing is both integer-operation- and memory-intensive. Object-parallel systems can also use this parallelism, but it is less visible because each front-end processor is tightly coupled to a back-end processor; the two processors can be thought of as a single "meta-processor".

2.4.1 Image-Parallel Systems

The main difference between hardware-and software-based image-parallel algorithms is that software systems usually assign a few large regions to each processor whereas hardware systems usually assign

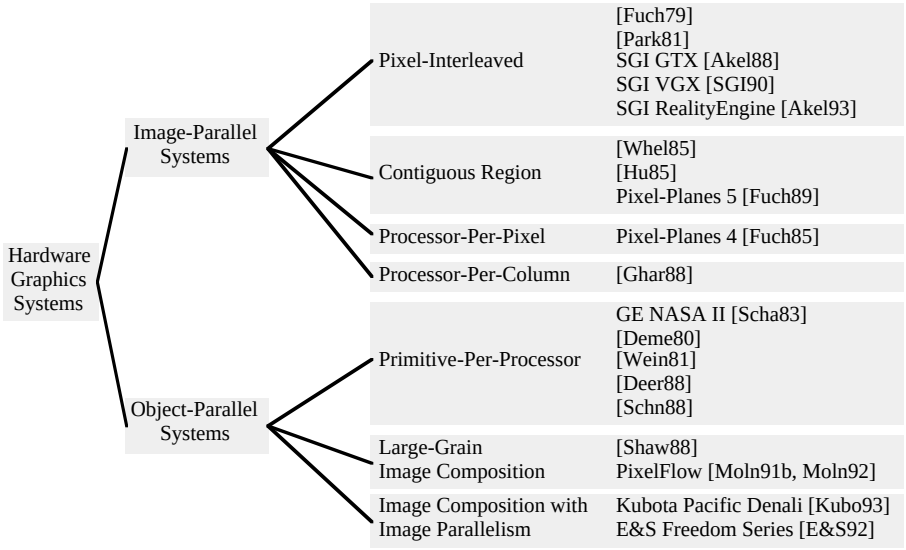


Figure 2.8 Taxonomy of the hardware graphics systems discussed in this section. Systems that have been built are shown with both the system's name and a reference to the description.

many small regions to each processor. The most common case is where the regions are the smallest possible: a single pixel. These "regions" are assigned to processors in an interleaved fashion (see figure 2.9). Values for i and j are chosen such that $ij = N$, and then each processor is given every i^{th} region horizontally and j^{th} region vertically.

If the regions are sufficiently small, all or nearly all the processors will work on each primitive, balancing the rasterization load. Usually, the rasterization processors do not perform the rasterization setup, as this would be expensive. Instead, the setup is done earlier in the pipeline. Using small interleaved regions means that each rasterization processor must accept a description of most of the primitives. In a specialized architecture, this bandwidth can be accommodated during the system design. Most general-purpose parallel processors do not have enough network bandwidth to accept a large fraction of the primitives, and must use much fewer regions. Some of the earlier hardware papers described systems using large, contiguous regions, which are more applicable for use on the general-purpose parallel systems considered in this dissertation.

Fuchs. One of the early parallel graphics systems was proposed by Fuchs [Fuch77, Fuch79]. This system has a central processor perform the geometry processing and the rasterization setup, and then uses a parallel subsystem to rasterize the polygons (see figure 2.10a). The processors are assigned portions of the screen in an interleaved fashion. All processors rasterize the same polygon at a time, waiting for the other processors to finish before going on to the next.

1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4

Figure 2.9 Example of interleaved partitioning using four processors. Each square usually represents a single pixel but can also represent a group of pixels. The numbers represent the processor assigned to the pixel or group of pixels.

Parke. Parke's paper [Park81] analyzed Fuchs's architecture as well as two other proposed architectures (see figure 2.10). The first proposed architecture used a screen subdivision method with one contiguous region per processor. The processors are organized in a pipeline by having a tree of splitter processors divide the incoming stream of polygons into a stream for each rasterization processor. The second proposed architecture is a hybrid between Fuchs's architecture and the first. The splitter tree is the same, but not as deep as before, giving a smaller number of contiguous regions. Each contiguous region is worked on by four processors in an interleaved subdivision method, as in Fuchs's architecture.

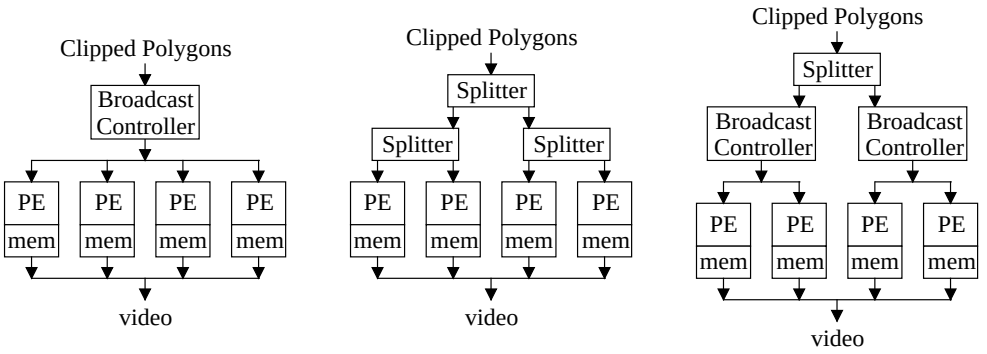
Parke derived performance models for the three architectures and also simulated the architectures. He simulated Fuchs's architecture for different interleave patterns, and found that the best interleave pattern is where the footprint approximates a square instead of one that is a single row or column. His models predicted that, for his test cases, the best architecture is the splitter architecture. However, that architecture suffers from load imbalances when the distribution of polygons on the screen is not uniform, and his performance model assumed a uniform polygon distribution. Because of this, he concluded that the hybrid architecture would be the best for most cases.

Parks. Parks [Park82] compared the three architectures proposed by Fuchs and Parke. He simulated each architecture for a series of scenes. He concluded that Parke's splitter architecture worked best with scenes having a uniform polygon distribution while Fuchs's interleave architecture works best with scenes having clusters of polygons. The hybrid algorithm had performance between the other two algorithms'. He also noted that the splitter architecture was the most sensitive to the screen position of the polygons; slight changes in the distance from the viewer to the model could make dramatic differences in the frame time. Fuchs's interleaved architecture was relatively insensitive the polygons' locations. These conclusions are for systems with 16 processors. The splitter algorithm is the best choice when using 256 processors because the per-polygon overhead dominates the time required by the interleaved method.

Whelan. In his dissertation [Whel85], Whelan simulated several different screen subdivision methods in order to find a suitable one for hardware implementation. He looked at three screen subdivision methods, dividing the screen into rows (i.e. regions extending the width of the screen), columns, and a equally-spaced grid of rectangular regions. His simulations included trying a range of region-processor ratios from 25 to 1. These methods used a static assignment of regions to processors, assigning the processors in an interleaved fashion. Each processor works on one or more contiguous regions of pixels.

Whelan also studied an adaptive screen subdivision method. The method generates 2^i regions in i steps. In each step, each existing region is cut in half so that each has half of the polygons (the location of a polygon is considered to be a single point, the centroid of the polygon). The odd steps cut the regions vertically, and the even steps cut the regions horizontally, as shown in figure 2.11.

His simulations indicated that the method that gave the highest parallel efficiency during rendering was the adaptive method, followed by the rectangular region method with more than one region per processor. However, he concluded that the rectangular region method was the most suitable for his hardware design, because calculating the region boundaries is quite expensive: "determining a partitioning ... probably



(a) Fuchs's architecture (b) Parke's splitter architecture (c) Parke's hybrid architecture

Figure 2.10 Architectures proposed by Fuchs and Parke.

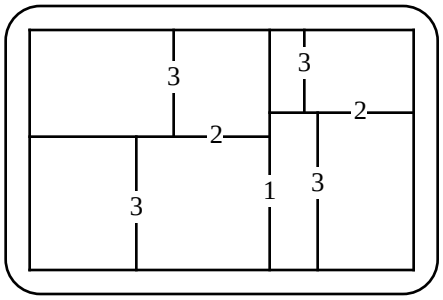


Figure 2.11 Example of Whelan's adaptive subdivision. Each line is numbered with the step where it was created.

requires as much computation as computing the image of the scene". Because his non-adaptive screen-space subdivision methods used static assignment of regions to processors, his methods were somewhat inefficient, using 16 processors with at most 70% efficiency .

Hu and Foley. A paper by Hu and Foley [Hu85] considered some similar methods for hardware implementation. They considered three architectures with a uniprocessor performing the transformation, lighting, and clipping, and then broadcasting the polygons to a parallel system for rasterization. Their simulations considered three methods of screen-space subdivision, all of which allocate scanlines:

- static assignment of a contiguous group of scanlines to each processor.
- static interleaved assignment of scanlines to processors.
- dynamic assignment of scanlines to processors. The uniprocessor handled the scanline assignment in this case. An omega network [Lawr75] handled communications between the processors and the shared frame buffer.

The authors simulated the three methods using each of two methods of broadcasting the primitives. The first method broadcasts the primitives one at a time, synchronizing the processors between polygons. The second method broadcasts all the primitives to the processors, so no synchronization is performed except at the end of processing. (The second method assumes each processor has sufficient storage to avoid delays.) They conclude that the dynamic assignment without per-primitive synchronization was the most practical. However, they do not take into account the increased cost of a system that supports dynamic assignment of scanlines compared to one with that has fixed assignments, or contention for the frame buffer.

SAGE. The SAGE (Systolic Array Graphics Engine) [Ghar88] is a rendering system which uses a systolic array [Kung82] of processors to generate the image a scanline at a time (see figure 2.12). A simple system has one processor for every column on the screen. Since Gharachorloo et. al. implemented the rasterizer in a VLSI chip that contains 256 processors, a complete rasterizer array requires only a few parts. Spans are input to the processor array, where each processor calculates whether the span overlaps it pixel location, and then passes the span to the next processor.

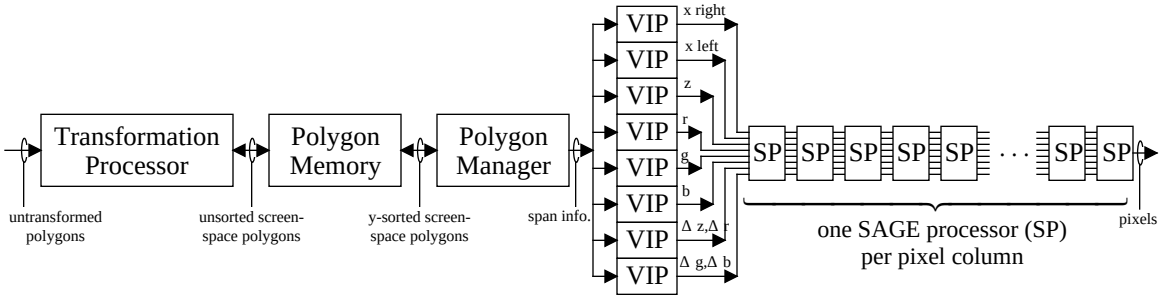


Figure 2.12 The SAGE architecture. Key: VIP = Vertical Interpolation Processor.

A full system has a parallel transformation engine to traverse the data set, generate screen-space polygons, and sort the polygons according to their topmost *y* value, placing them into a buffer. A second engine traverses the polygon buffer and computes the edge, color, and *z* slopes. It then loads the polygons and slope values into the Vertical Interpolation Processors (VIPs), also to be implemented as a VLSI chip. These processors interpolate the active polygons and generate the spans for the SAGE processors. When a scanline is rasterized, the spans for that scanline are fed into the systolic array and percolate through the processors. When all of the spans are processed, the computed pixels are read out and stored in a frame buffer.

The SAGE system promised rendering performance approaching one million polygons per second. The front-end processor has not been described.

Pixel-Planes 4. The Pixel-Planes 4 system [Fuch85] takes an extreme approach to screen subdivision, using a processor per pixel. The systems consists of a floating-point front end, plus a 512x512 SIMD array for the rasterization. A key feature of the SIMD processors is their linear expression tree, which enables the processors to quickly evaluate linear expressions in screen space. Linear expressions can be used to define polygon edges and to interpolate *z* and color values.

The front-end processor traverses the data set that is stored in its memory, and does the usual front end tasks: the transformation, clipping, and lighting calculations. It also generates instructions for the SIMD array, which involves calculating the slopes' values. Because the rasterization array is SIMD, it takes the same amount of time to rasterize a large polygon as it does to rasterize a small one. This characteristic is good if the data set has large polygons, but bad with many small polygons: most processors do no useful work when rasterizing small polygons. The system can display 37,000 Gouraud-shaded triangles per second.

Pixel-Planes 5. The Pixel-Planes 5 system [Fuch89] continues the use of SIMD processors for rasterization, but uses multiple SIMD arrays (called *renderers*) instead of one large one. The SIMD arrays only execute instructions sent to them, so instead of sending screen-space primitives to the renderers, a series of instructions are sent that perform the rasterization.

The Pixel-Planes 5 front end consists of several Intel i860 processors. Each stores a portion of the model, and transforms it during each frame, much like the Pixel-Planes 4 front-end processor. After the transformation is complete, the rasterization is performed, with the screen is subdivided into 128x128 pixel regions, the size of the SIMD array. The renderers are initially assigned to the regions with the most primitives, and each front-end processor sends the instructions to rasterize the primitives that overlap the assigned regions, sending them to the appropriate renderer. When all processors have sent their commands for a given region, that renderer is reassigned to the region with the next smaller primitive count. This continues until all of the regions are completed.

The Pixel-Planes 5 system uses the Phong shading model and supports procedural textures. MIPMAP textures are also supported, but at a lower speed. The system is described in detail in Chapter 8. The chapter also describes a second rendering algorithm, called MEGA, that uses static assignment of regions to renderers.

Silicon Graphics GTX. The Silicon Graphics GTX [Akel88] is a highly pipelined graphics system (see figure 2.13a). The front end calculations are performed in a pipeline of identical processors. The pipeline stages are as follows:

- Geometry Accelerator: buffers Graphics Library (GL) calls and converts data formats.
- Geometry Engines: The GTX has five Geometry Engines, each implementing a different part of the pipeline. Each engine has a first-in first-out buffer and a floating point processor. The different Geometry Engines split the front end calculations as follows:

  1. Vertex and normal transformation, and transformation matrix calculations.
  2. Lighting calculations.
  3. Clip testing and trivial rejection.
  4. Perspective division and clipping, when indicated by the previous step.
  5. Vertex transformation to the viewport, clamping of colors to the allowed range, and depth cueing.

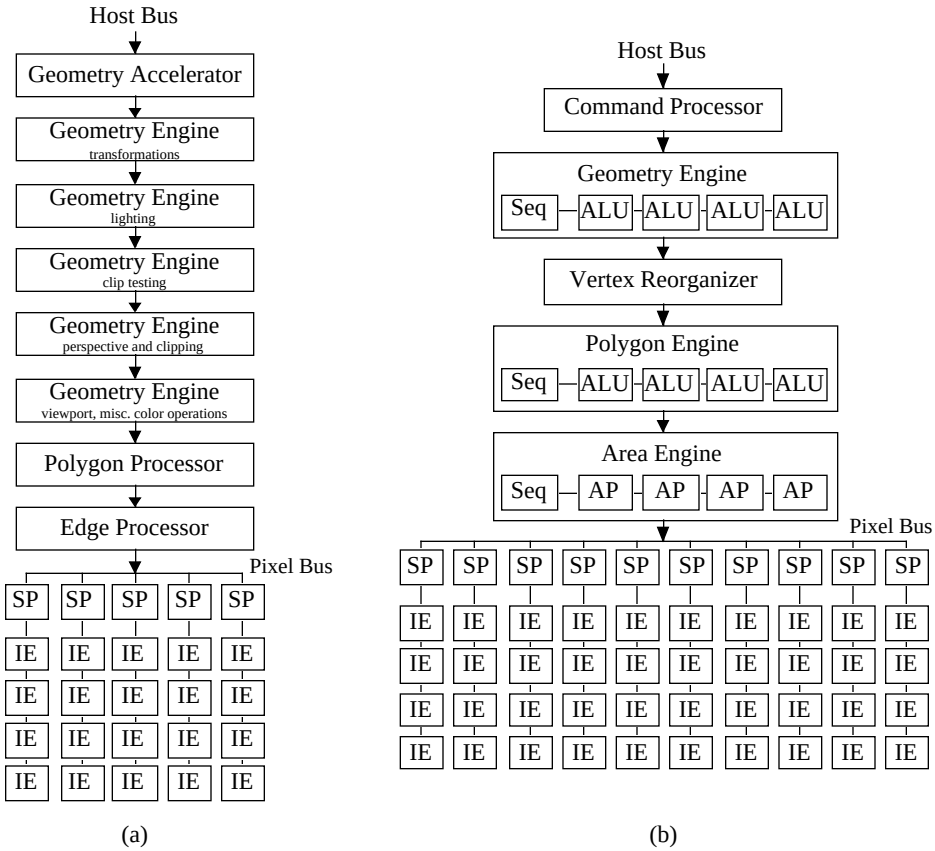


Figure 2.13 (a) Silicon Graphics GTX architecture (b) Silicon Graphics VGX Architecture. Abbreviations: SP - Span Processor; IE - Image Engine; Seq - Sequencer; ALU - Arithmetic Logic Unit (floating point); AP - Area Processor.

- Polygon Processor: decomposes polygons into screen-aligned trapezoids (the vertical edges are screen-edge aligned), and calculates parameter (r, g, b, y, z) slopes in the horizontal direction.
- Edge Processor: iterates horizontally along the trapezoids, generating vertical spans, and calculates the parameter slopes of the spans.
- Span Processors: The 5 Span Processors generate pixels from the vertical spans. Each Span Processor is responsible for every 5th column of pixels. The Span Processors operate independently, in MIMD fashion.
- Image Engines: Each Span Processor has four Image Engines associated with it that perform the z-buffering. Each MIMD Image Engine operates on every 4th pixel in the columns assigned to the associated Span Processor.

The GTX supports linearly-interpolated colors (Gouraud shading) using the Phong lighting model at each vertex.

Silicon Graphics VGX. The VGX [SGI90] is a pipelined graphics system similar to the GTX, but with a shorter and wider front-end pipeline, and with more processors in the rasterization system (see figure 2.13b). The pipeline stages are as follows:

- Command Processor: parses GL library calls and outputs complete primitives.
- Geometry Engine: A 4-way SIMD floating point processor that does the transformation, lighting, and clip testing, usually with one vertex per processor.

- Vertex Reorganizer: reorganizes vertex-per-processor data format from the Geometry Engine to the polygon-per-processor format needed by later processors.
- Polygon Engine: A 4-way SIMD processor that calculates polygon slopes and clips when necessary.
- Area Engine: A 4-way SIMD fixed-point processor that generates vertical spans from polygons.
- Span Processors: The 5 or 10 Span Processors generate pixels from the vertical spans. Each Span Processor is responsible for every 5th or 10th column of pixels. The Span Processors operate independently, in MIMD fashion.
- Image Engines: Each Span Processor has four Image Engines associated with it that perform the texture lookup and *z*-buffering. Each MIMD processor operates on every 4th pixel in the columns assigned to the associated Span Processor

The VGX supports Gouraud shading and Phong lighting, and can also do texture mapping. The texture mapping is not true MIPMAP texture mapping [SGI93]: it only does a bilinear interpolation from pixels using a single texture resolution for the polygon. It also only performs perspective correction on the texture coordinates at each vertex instead of at each pixel. The system supports automatic polygon subdivision to reduce errors from the two approximations to MIPMAP texturing. Later systems, the VGXT, SkyWriter and RealityEngine, perform true MIPMAP texturing.

Silicon Graphics RealityEngine. The SGI RealityEngine [Akel93] is similar to the VGX system but has an even shallower pipeline, and has a larger feature set. It can perform MIPMAP texture mapping and one-pass anti-aliasing using 4, 8, or 16 samples per pixel.

The pipeline stages are as follows:

- Command Processor: parses GL library calls and outputs complete primitives.
- Geometry Engines: The Geometry Engine stage has 6, 8, or 12 MIMD i860 processors that do the transformation, lighting, clipping, and calculate the triangle slopes.
- Fragment Generators: The Fragment Generators scan-convert the polygons, generating pixels or pixel fragments. It also does the MIPMAP texture mapping. A system can have 5, 10, or 20 Fragment

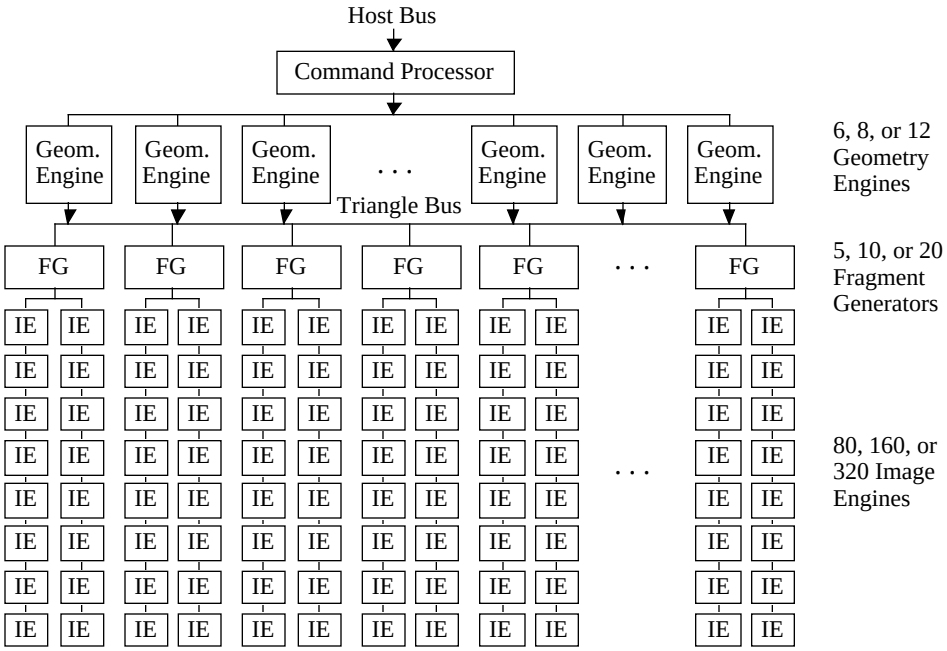


Figure 2.14 Silicon Graphics RealityEngine architecture. Abbreviations: Geom - Geometry; FG - Fragment Generator; IE - Image Engine.

Generators, each responsible for an equal number of pixels on the screen. The FG's are MIMD processors.

- Image Engines: The MIMD Image Engines takes the pixels and subpixel masks generated by the Fragment Generators and *z*-buffers each of the samples. Each Fragment Generator has 16 Image Engines.

The RealityEngine graphics subsystem has been succeeded by ones with higher-performance and similar feature sets, the RealityEngine^2 and the InfiniteReality.

2.4.2 Object-Parallel Architectures

The hardware-based object parallel systems generally have specialized interconnection networks to combine the samples generated by the rendering engines. The specialized networks are designed to handle the large bandwidths required, which can saturate general purpose networks. One common method used is to have each processor compute a full screen image of a cut-down version of the model. Then, the image from each processor is read out along with the *z* values, and the nearest primitive in each pixel selected by examining the *z* values of the frame buffers.

Figure 2.15 shows an example of this type of system where the rendering processors continually output the video image. The *z* and RGB values are sent through the composition tree and then to digital-to-analog converters and the monitors. This requires synchronizing both the frame buffer pixel scan-out and buffer swaps.

2.4.2.1 Primitive-Per-Processor Systems

The early work in image composition architectures was done using primitive-per-processor systems. These systems have rasterization processors that can only rasterize one primitive at a time. The image compositors are usually incorporated with the rasterizer, and the processors arranged in a pipeline (a maximally unbalanced tree). This arrangement makes it very easy to expand the system. Some systems time-share processors so that they can support more complex scenes, with more primitives than one per processor. These systems first sort all the primitives by their topmost *y* coordinate. Then, at the start of a new scanline, they load the primitives that have their top at the current scanline, and also they remove primitives that have their bottom at the previous scanline. Thus, a system can reuse processors during a frame; it only needs to have enough processors to handle the scanline that has the largest number of primitives crossing it.

The first implementation was the NASA II flight simulator built by General Electric [Scha83]. In this system, four circuit boards (one per edge) were used to determine whether the current pixel was within a quadrilateral. Many cards were used to support a complex scene. Here, the priority order was fixed, without regard to the *z* value. The lowest numbered card set with a polygon overlapping the current pixel determines the pixel's color. Primitives are assigned to modules according to their priority order by another processor. That processor determines the priorities using precomputed intra-object polygon ordering and inter-object ordering using separating planes.

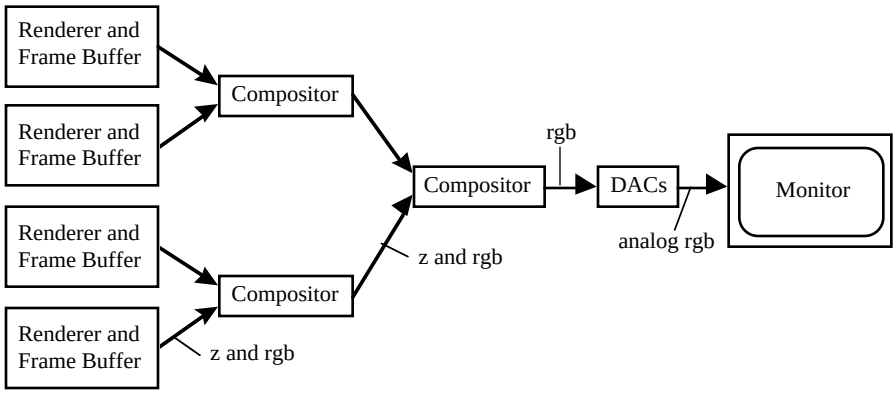


Figure 2.15 Example of a video-output image composition system.

A later system was proposed by Cohen and Demetrescu [Deme80], which used processors implemented in VLSI. The processors were arranged in a pipeline, and compared depth values. This idea was extended in the system proposed by Weinberg [Wein81] to support anti-aliasing using a variant of the A-Buffer algorithm [Carp84]. Instead of outputting a single *z* and RGB pair, each processor outputs a sequence of polygon fragments consisting of those objects that overlap the pixel. The compositor then merge these lists of fragments. Fragments that are not visible are not passed on, which helps keep the list size manageable. The output of the composition tree is sent to filter processors, which combine the fragments to arrive at the final pixel value.

Deering et. al. [Deer88] extended the basic model in a different direction by adding deferred shading, allowing Phong shading. In their proposed system, the processors output a packet containing *z*, material type, and normal vector. These values are compared and the front-most one given to a set of shading processor, which computes the color. Anti-aliasing is supported by supersampling.

Finally, Schneider and Claussen [Schn88] combined deferred shading and one-pass anti-aliasing in their design of the PROOF architecture. The pixel fragments lists are first merged in the composition tree. The fragments are then each shaded using shading processors, and finally filtered to the final pixel value in the filtering processors.

One common characteristic of the primitive-per-processor systems is that their performance drops considerably when there are more primitives than processors. When this happens, a multipass algorithm is used to handle the "overflow" primitives. In each pass, the *N* processors are loaded with the next *N* primitives, and the *z* and RGB values from the previous pass are passed through the processor array. Some systems place primitives in processors only when the primitive crosses the current scanline, reducing the need for the multipass algorithms.

2.4.2.2 Large-Grain Image Composition Systems

Some later image compositions used larger-grained processors, ones that process more than one primitive at a time. Shaw et. al. [Shaw88] proposed a system that supports anti-aliasing. Each renderer outputs a *z*, color, and coverage value for each pixel. The compositor combines the samples using a simplified version of Duff's composition algorithm [Duff85].

In his dissertation, Molnar [Moln91b] explored large-grained composition systems in detail. He examined several A-buffer type methods that were used previously, and concluded that they had objectionable artifacts. The only way to remove the artifacts is to include enough information in each fragment to calculate multiple samples per fragment in the filtering stage. Including that much information makes the composition bandwidth required approach the bandwidth required by a brute-force supersampling technique using five samples-per-pixel. (Molnar also investigated anti-aliasing methods that reduce the number of samples required [Moln91a]). The A-buffer method would also require complicated, high performance compositors and fragment combiner processors, while the supersampling method has trivial compositors and simple fragment combiners.

He developed a detailed system design called PixelFlow that used multiple-sample anti-aliasing and included support for deferred shading. A revised description of the system appears in [Moln92] and is currently being implemented.

2.4.2.3 Image Composition with Image Parallelism

Kubota Pacific Denali and Evans and Sutherland Freedom. Some recent workstations, the Kubota Pacific Denali [Kubo93] and the Evans and Sutherland Freedom Series [E&S92] perform most of the work using object parallelism. Unlike the earlier object-parallel systems, the hidden surface elimination is done using image parallelism. One set of processors executes the functions of the graphics pipeline through scan conversion, with the processors operating on different primitives in parallel. A second set of processors do the *z*-buffering, with each processor responsible for a portion of the screen.

The Denali system accepts commands from the host processor and sends them to one or more Transform Engine Modules (TEM). It broadcasts attribute changes to all modules, and sends primitives to a single module in round-robin order. The TEMs transform the primitives and rasterize them. They are then routed to a Frame Buffer Module (FBM) for *z*-buffering.

A Denali system can contain 5, 10, or 20 FBMs. Each holds the *z*-buffer and frame buffer for a portion of the screen. In a five FBM system, the first FBM does the first four pixels in a scanline, the next FBM does the next four pixels, etc. In a 10 FBM system, 5 FBMs handle the even scanlines, and the other handle the odd scanlines, with each set of 5 FBMs dividing up the scanline like the 5 FBM system. In a 20 FBM system each set of 5 FBMs works on every fourth scanline. The Denali system can perform Gouraud shading and Phong lighting, and can do MIPMAP texture mapping.

The Evans and Sutherland Freedom Series operate similarly. They have up to 16 front-end processors similar in function to the Denali TEMs. The systems have a routing network and a parallel frame buffer, but no details are available. The Freedom systems support texture mapping like the Denali and also support one-pass anti-aliasing using sub-pixel masks.

2.5 Summary

This chapter has described many different algorithms, most of which predicted or delivered good results. Because the algorithms were targeted for a variety of architectures and different image quality levels, it is difficult to evaluate their suitability for use for polygon rendering on a general-purpose multicomputer. The next chapter will take a step back and place many of the algorithms described here in a taxonomy and evaluate the branches of the taxonomy.

CHAPTER THREE

CLASSES OF PARALLEL POLYGON RENDERING ALGORITHMS

This chapter describes a classification of polygon algorithms and analyzes each to find the one most suitable for multicomputers. The classification used was developed at UNC by several members of the Pixel-Planes project, and has been described previously in Molnar's dissertation [Moln91b] and in a paper by Molnar, Cox, Ellsworth, and Fuchs [Moln94]. Much of the discussion and analysis in this chapter is based on the work in the latter paper.

There have been several classifications of polygon algorithms. Sutherland, Sproull, and Schumacker's classic paper [Suth74] characterizes ten uniprocessor polygon hidden-surface algorithms. Gharachorloo et al. [Ghar89] describes algorithms used in hardware systems, many of which are parallel. Whitman [Whit92] describes a taxonomy of parallel algorithms. However, there has not been a characterization of the different methods of parallelizing the entire rendering process until the Pixel-Planes characterization.

3.1 Description of the Algorithm Classes

This classification of parallel rendering algorithms only considers *fully parallel* rendering algorithms, where the data at each step in the pipeline is divided among multiple processors. Partially parallel algorithms can be considered as degenerate cases, but the analyses may not be correct since some overhead costs do not exist. Also, the classification only covers feed-forward algorithms, algorithms based on the traditional graphics pipeline. It does not cover ray tracing or certain volume-rendering algorithms. Figure 3.1 shows the graphics pipeline implemented in a fully parallel algorithm.

If, for simplicity, shading and anti-aliasing are omitted, the graphics pipeline is as follows. The pipeline starts by traversing the graphics database or by generating the triangles (for retained-mode and immediate-mode interfaces, respectively). The output of this step is *raw triangles*, 3D triangles in world space or modeling space that have yet to be transformed. The next step is the geometry processing, which consists of the transformation to eye space, clipping, transformation to screen space, and perhaps shading the vertices. The output of this step is triangles in screen space, or *display triangles*.

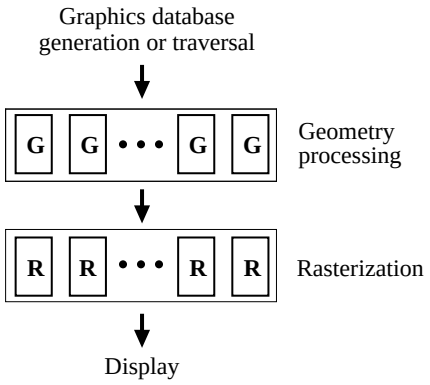


Figure 3.1 Graphics pipeline in a fully parallel rendering system. Processors *G* perform geometry processing. Processors *R* perform rasterization (after [Moln94]).

The final step turns the display triangles into the final pixel values. The triangles are scan converted, and each pixel is *z*-buffered and shaded. The display triangles are usually converted first to spans, then to pixels, and then to visible pixels.

In a fully parallel algorithm, the final step must be done using image parallelism, as all the potentially visible pixels must be examined to determine the final pixel value. However, image parallelism cannot be used for the entire graphics pipeline. The pipeline begins with raw triangles, which cannot be sorted in screen-space because the screen space location has yet to be calculated. The graphics pipeline must start with object parallelism, with the raw triangles distributed to the processors.

So, the triangles must be partitioned two different ways at different points of the pipeline. The change in partitioning the triangles requires that the triangles be sent among the processors, with the triangles sorted according to the location of triangles on the screen. The sort, or *redistribution*, can be made at many different points within the graphics pipeline. The data that is redistributed is different when the sort is done at different places in the pipeline. At different points in the pipeline, the data corresponding to a single triangle varies in size, which implies that performing the sort at different locations would require differing amounts of communication bandwidth. The placing of the sort also determines the amount of computation that is duplicated. Because of these factors, choosing where to perform the sort between partitionings is quite important when choosing a parallel algorithm.

Sort-first algorithms perform the sort as early as possible in the pipeline, before completing the geometry processing, and redistribute raw (untransformed) triangles. When the sort is performed later on in the pipeline, after all the geometry processing has been completed, display triangles are redistributed. These algorithms are called *sort-middle*. Finally, if the sort is done after rasterization has started, so that spans, pixels, or pixel fragments are redistributed, the algorithm is in the class called *sort-last*.

These algorithm classes are not comprehensive. Algorithms exist that perform more than one redistribution (e.g., [Orte93]). However, these algorithms do not appear to be feasible for multicomputer algorithms. On current multicomputers, the redistribution step takes a significant amount of time, making algorithms that perform it more than once have a disadvantage compared to algorithms that do only one redistribution. Additional redistributions might give better load-balancing. However, this seems unlikely, as the algorithms presented in later chapters achieve high efficiency using one redistribution.

Even so, the classification method allows high level analysis of algorithms, which will be done in sections 3.2 and 3.3. The next sections describe the classes in more detail.

3.1.1 Sort-First

In sort-first, the redistribution step is performed as early as possible. Most of the work is done using image parallelism, where each processor is responsible for one or more regions on the screen. However, as mentioned above, the start of the transformation process is done with the triangles distributed across the processors by some method. Each triangle is transformed to screen space, and then sent to the processors handling the regions the triangle overlaps. Only the vertices of the triangle are transformed; other calculations, like transforming normal vectors and shading, are done on the receiving processors. The transformed points may be sent, along with the triangle, if it is faster to do so instead of retransforming the triangle.

The algorithm works well with primitives that are more complex than triangles because the cost to transform the primitive is small compared to the total cost. This is true for complex primitives like curved surfaces. The algorithm also works well when the frame-to-frame coherence of each triangle's location on the screen is used. If the display list is not changed between frames, the receiving processor can save the raw triangles for one frame in the hope that it will be need them for the next frame. The processor can reuse a triangle without further redistribution if the processor is assigned the same region(s) in the next frame, and the triangle is still in the processor's region. In some cases the redistribution traffic can be nearly reduced to zero.

The screen-space subdivision methods described in Chapter 2 can be used to parallelize the rasterization part of the pipeline. However, Mueller's simulations [Muel95] show that better load-balancing comes with the cost of decreased coherence. The better load-balancing methods break up the screen so that each processor could have several regions. However, the best coherence comes when each processor has only

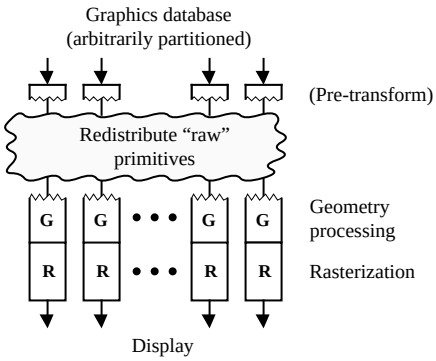


Figure 3.2 Sort-first. Redistributes raw triangles during geometry processing (from [Moln94]).

one region, as each region is larger, increasing the probability that a triangle will remain on the same processor.

Sort-first algorithms that exploit coherence must partition the triangles during geometry processing the same way as they are partitioned during rasterization: the triangles are divided following the chosen screen-space subdivision method. Triangles that are outside the viewing frustum must also be partitioned among the processors so that they can be transformed each frame to see if they become visible. One way to partition the off-screen triangles is to randomly assign them to processors when they leave the screen. Other methods are possible.

Of the three classes of algorithms, sort-first has been studied the least. One of Cox's parallel RenderMan implementations [Cox95] uses a sort-first algorithm. Because it redistributes complex primitives such as curved surfaces, and because it only renders one frame, it does not make use of coherence. I have heard that Robert Stein of the Parallel Software Group is in the process of building a sort-first system, but little information is available about the work. Mueller's paper [Muel95] is the only one published to date that considers sort-first algorithms that use coherence.

No sort-first rendering algorithm has been developed that supports editing a hierarchical display list, such as used in PHIGS [ANSI88]. Most retained-mode libraries have hierarchical display lists. When coherence is used, the triangles that are distributed across the processors must be from a flattened display list, without hierarchy. Some correspondence between the hierarchical and flattened display lists must be maintained so that editing is possible. When the display list is edited the flattened display list must also be updated. This can be difficult as a single polygon may be instanced more than once. Also, hierarchical display lists have attributes that affect triangles below them in the list. An attribute can affect hundreds or thousands of triangles. When the attribute is edited, the new attribute must somehow be associated with each affected triangle in the flattened display list.

Editing a flat display list is somewhat easier. One method for allowing editing would keep a separate version of the display list that is statically partitioned among the processors. When a triangle is updated, the display list is queried, and the old triangle returned. The new triangle is sent to the processors that overlap the old triangle, which is determined by transforming the old triangle and calculating the regions it overlaps. A second method would also keep a static display list, but with a list of the processors that have a copy of each triangle. Each processor would update the display list whenever it starts or finishes keeping a copy of a triangle. The first method does not work if there are transformation matrices internal to the display list. However, it does not need the additional communication used by the second method to keep the triangle-processor directory up to date.

3.1.2 Sort-Middle

In sort-middle algorithms, display triangles are redistributed. The display triangles are first generated by transforming the raw triangles, and are then redistributed among the processors. The display triangles are rasterized after the redistribution. As with the other algorithm classes, the tasks before and after the redistribution step must both be partitioned among the processors so that the loads are balanced. Both of these partitionings are fairly simple to implement. Most parallel polygon implementations to date have been in the sort-middle category.

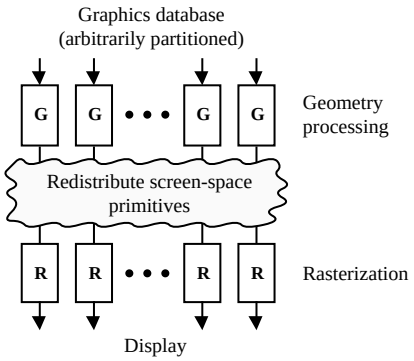


Figure 3.3 Sort-middle. Redistributes screen-space triangles between geometry processing and rasterization (from [Moln94]).

The geometry processing can be partitioned arbitrarily as long as the load is balanced. For flat, all-triangle, models (ones that don't have hierarchy), the simple method of assigning polygons to processors in round robin fashion has worked well [Whit92, Croc93]. Here, a polygon p gets assigned to processor p mod N, where N is the number of processors. This works even though the cost to transform a triangle can vary by a factor of up to 50. If the triangle is off-screen, it can be rejected quickly. It can also be rejected quickly when the back face is visible, and back-face culling is enabled. On the other hand, clipping a triangle when it crosses a viewing frustum boundary can take ten times a normal triangle's processing time. Also, large triangles (large in screen area) are more expensive than small triangles because, on average, they need to be redistributed to more processors. The simple method works because it places triangles that are near to each other in the display list, which are usually also near to each other in world space, on different processors. Even though the cost of transforming the triangles changes, it changes somewhat evenly across the processors because the imbalances are usually averaged out across the entire display list.

Attributes (transformation matrices, color changes) can possibly affect all the triangles found later in the display list, making them harder to distribute. One can simply send attribute changes to all of the processors. However, a processor may not have any of the triangles that the attribute affects, so having that processor process the attribute is inefficient. An algorithm that removes the extra attribute changes is described in [Ells90a]. This paper also describes a method for distributing hierarchical display lists. More details about distributing display lists appear in Chapter 5.

Sort-middle also supports an immediate-mode interface. However, load-balancing the transformation process is more difficult with an immediate mode interface compared to a retained-mode interface. One reason is that the user performs the data partitioning, usually without regard to the transformation process. It is more likely that a single processor's triangles are located close together in world space, and thus it is more likely that they will all go off screen, or increase in screen size due to perspective. Removing this load imbalance is fairly difficult. When the geometry processing is done on the same processor where the triangles are generated, there is no redistribution step beforehand that can be used to repartition the triangles. Compare this with the rasterization task, which has a redistribution step before it runs. Overloaded processors can send untransformed triangles to other processors, but the necessary communication takes a large fraction of the transformation time. This means that correcting the load imbalances will only help when there are gross imbalances. When there are small imbalances, the additional communication cost will overwhelm the savings from the higher efficiency. Also, the cost of determining the heavily- and lightly-loaded processors must be considered, which is a global operation. Load-balancing a parallel immediate-mode system has not yet been fully studied. It is currently being studied for the PixelFlow system at UNC [Moln92].

Many of the image parallelism methods described in Chapter 2 can be used to partition the rasterization step. Sort-middle algorithms do not need to constrain the partitioning algorithm to improve coherence as sort-first algorithms do. Image parallelism methods for sort-middle are discussed further in Chapters 4 and 6.

Sort-middle has been studied extensively. Two parallel software systems, by Whitman [Whit92, Whit94] and Crockett and Orloff [Croc93], are sort-middle algorithms. Roble [Robl88] has a sort-middle-like

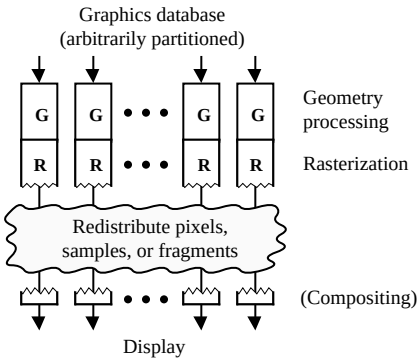


Figure 3.4 Sort-last. Redistributes pixels, samples, or pixel fragments during rasterization (from [Moln94]).

software implementation, but he did not perform the transformation in parallel. Several hardware systems have been proposed that use image parallelism suitable for sort-middle [Fuch79, Park80, Whel85, Hu85, Ghar88], and at least one full hardware system has been built [Fuch89]. Many of the commercial graphics systems are best considered sort-middle systems [Akel88, SGI90, Akel93] even though they not fully parallel (they send all primitives through a single processor at some point in the pipeline).

3.1.3 Sort-Last

In sort-last, the redistribution step is done after rasterization is started. However, the redistribution step must be done before all the hidden surfaces have been removed, since any processor may determine the final pixel value. This means that pixels, spans, or samples are redistributed along with depth information. Nearly all the rendering is performed before the redistribution; only hidden surface removal and possibly deferred shading are done after redistribution.

There are two main variants of sort-last. One method, called *sort-last-sparse*, has each processor transmit only the pixels within its triangles. The other method, called *sort-last-full*, has each processor transmit the entire screen. In sort-last-sparse, hidden surface elimination among a processor's triangles is usually not performed because the cost usually outweighs the cost of sending more than one sample for a few pixels [Cox92]. This especially true when using more than a few tens of processors. Local hidden surface elimination is always done in sort-last-full.

The triangles are partitioned among processors using object parallelism methods, methods similar to the ones described for sort-middle algorithms. The main difference is due to more work being done on a processor before redistribution: in sort-middle, only the transformation is done, while in sort-last, much of the rasterization is done. This means that the difference in work to process different triangles can vary by a larger factor, up to a factor of a million (compare the cost of a one-pixel triangle with a million-pixel triangle). The large difference in work will make load-balancing more difficult. For example, statically dividing the triangles among the processors will not work as well as with sort-first and sort-middle, since the larger variance in the time to rasterize each triangle will increase the variance in the time taken by each processor.

Cox's sort-last-sparse implementation, which uses a static assignment of regions, did not effectively balance the loads [Cox95]. Molnar studied load-balancing effects for systems with specialized SIMD rasterization hardware [Moln91b]. The SIMD hardware's rasterization time is less sensitive to the size of the triangles, which puts less stress on the load-balancing algorithm.

Load-balancing the tasks after redistribution is fairly simple with sort-last. It is trivial for sort-last-full: each processor can handle an equal portion of the screen. It is also simple to do for sort-last-sparse. The screen can be divided in a single-pixel interleaved fashion. As discussed in section 2.4.1, this means each processor will process nearly the same number of pixels for composition.

Sort-last has been fairly well investigated. The primitive-per-processor systems [Deme80, Wein81, Scha83, Deer88, Schn88] are all sort-last-full systems. Two large-grain image composition systems [Shaw88, Moln92] are also sort-last-full. Sort-last-sparse has been implemented on the CM-5 [Cox95], and is currently available in commercial workstations [Kubo93, E&S92].

3.1.4 Other Algorithms

As mentioned before, algorithms do not necessarily fit into one of the three classes. For example, an algorithm can redistribute the data more than once. A data parallel (i.e., SIMD) algorithm implemented by Ortega et. al. [Orte93] redistributes both spans and potentially visible samples. It also redistributes the data more than once to improve the load-balancing. However, since good algorithms exist that have only one redistribution step, it appears that the additional redistribution is unnecessary.

Other hybrid algorithms are more promising. These algorithms have one redistribution step but vary the place where the redistribution is performed. Cox's hybrid algorithm [Cox95] is mainly a sort-first implementation, but uses a sort-last-sparse-like algorithm to move work from overloaded processors to others that finish early. Or, a graphics library that has both immediate-mode and retained-mode interfaces could use a sort-middle algorithm to process triangles given to the library via the immediate mode interface, and use a sort-first algorithm to process the triangles retained between frames. The sort-first algorithm would be able to reduce the required communication by migrating the triangles to the processors that render them. The latter hybrid is a topic for future research.

3.2 Models of the Algorithm Classes

Now that the algorithms have been described, along with the characteristics of typical load-balancing algorithms, we can compare the suitability of the algorithm classes for use on general-purpose multicomputers running the typical application, the visualization of scientific applications. Three characteristics of these applications bear on the analysis: (1) it is typical for the application to regenerate large portions of the geometric model each frame; (2) the models are fairly large, having at least 50,000 triangles; and (3) the triangles are fairly small, usually containing fewer than 10 pixels. I will compare the algorithms by developing equations for the overhead costs of algorithms in each class, and then use the equations to compare the algorithms.

The analytic performance models of the algorithm describe most of the computation and communication done by the algorithm. They do not include load-balancing, as detailed load-balancing information has not yet been published for most of the classes. The models also do not include provisions for editing the user's geometric models. Even so, high-level analytic models enable one to calculate some bounds on the performance of the various classes.

The models presented here have been developed starting with the models published in [Moln94]. However, the models presented here are somewhat more detailed than those. They are also specialized to include costs incurred when running on general-purpose systems, and to the characteristics of interactive applications. (The consequence of the latter specialization is the dropping of provisions for tessellation, as most interactive applications do not use curved surfaces.) The main difference is that the costs (times and sizes) are quantified, instead of being left unbound. The timing values for the models in this chapter are derived from measurements of a sort-middle implementation running on the Caltech Touchstone Delta system. The implementation is described further in later chapters, and the measurement methods are described in Appendix B. Because the values are from a single implementation, they are only directly representative of one class of algorithms. Using the values for other classes is an approximation. For example, a sort-last implementation does not need to perform bucketization. The sort-middle code has provisions for bucketization, and thus takes additional time. On the other hand, each method requires that additional code be inserted, making them take more time than the sort-middle measurements indicate. Given these caveats, the sort-middle measurements are useful as estimates of the actual times.

In the equations, times are indicated using a subscripted t; the subscript indicates the category. Redistribution costs are not shown as times since they consume both communication bandwidth and processor time. Instead, the redistribution costs are shown as the total number of bytes to be transferred each frame. The sizes of different data types are shown using subscripted s. The processing time required by the redistribution can be calculated by multiplying the total number of bytes by the per-byte time communication time. The symbols used in the analysis are listed in table 3.1, and the times and data sizes are in tables 3.2 and 3.3.

Symbol	Description
N	number of processors
n	number of triangles
$\bar{h}$	average number of scanlines that each triangle crosses
$\bar{a}$	average number of pixels in each triangle
S	number of samples per pixel (for anti-aliasing)
O	overlap factor
A	screen area (number of pixels)

Table 3.1 Symbols used in the analysis of the algorithm classes.

Cost	Description	Value (μsec)
t_{pre_xform}	time to transform a triangle to screen space	3.2
t_{geom}	time to transform a triangle, and calculate per-vertex lighting	8.3
t_{bucket}	time to put a triangle into one bucket	2.7
t_{tri}	time required for per-triangle rasterization setup	18.8
t_{line}	time for per-scanline setup	2.74
t_{scan}	time to scan-convert a pixel	0.03
t_{comp}	time to composite a pixel	0.2
t_{extra_comp}	extra time to composite a sort-last-sparse pixel	0.28

Table 3.2 Costs used in the analysis of the algorithm classes. The values are from an Intel Touchstone Delta implementation.

Name	Description	Size (bytes)
s_{raw_tri}	size of raw triangle	88
s_{disp_tri}	size of display triangle	56
s_{sparse_samp}	size of sort-last-sparse sample	12
s_{full_samp}	size of sort-last-full sample	8

Table 3.3 Data sizes used in the analysis of the algorithm classes.

3.2.1 Serial Algorithm Cost

As a starting point, let us derive a model for the serial rendering algorithm, as listed in table 3.4. The first pipeline stage is the geometry processing, which takes t_{geom} time for each of the n triangles. The rendering stage first does per-triangle setup (sorting vertices, calculating slopes), then does per-scanline calculations, and finally per-pixel calculations. The per-pixel calculations are broken into two parts, scan conversion and visibility. While shading is not explicitly mentioned, the time to perform it is included in the timing values for the geometry processing and the per-pixel calculations. The times in table 3.2 are for Gouraud shading.

Pipeline Stage	Serial Cost
Geometry processing	nt_{geom}
Scan conversion	$nt_{tri} + n\bar{h}t_{line} + n\bar{a}St_{scan}$
Visibility	$n\bar{a}St_{comp}$

Table 3.4 Serial algorithm cost.

3.2.2 Sort-First Overhead

As mentioned previously, sort-first algorithms can use the coherence of a polygon's position on the screen to reduce redistribution. If there is no coherence, each of the n triangles will have to be redistributed. As mentioned in section 3.1.2, when a triangle is distributed to processors subdividing the screen it may need be sent to more than one processor. The average number of regions that the triangles overlap is denoted O, the *overlap factor*. This factor can be approximated when the screen is divided evenly using a formula

Pipeline Stage	Overhead Cost
Pre-transform	cnt_{pre_xform}
Geometry processing	$n(O-1)t_{geom}$
Bucketization	nOt_{bucket}
Redistribution	$cnOs_{raw_tri}$
Scan conversion	$n(O-1)t_{tri} + n(\sqrt{O}-1)\bar{h}t_{line}$
Visibility	—

Table 3.5 Overhead costs in sort-first algorithms. The communication cost is shaded to indicate that it is the amount of communication, not the time required.

developed by John Eyles. For a region with width W and height H, and a triangle with width w and height h, the overlap factor is approximately

$$\left(\frac{W+w}{W}\right)\left(\frac{H+h}{H}\right). \tag{3.1}$$

Derivations of this formula appear in [Moln91b] and [Moln94].

So, when there is no coherence, the redistribution cost is nOs_{raw_tri}. To model coherence, we use c, the fraction of triangles that must be redistributed in each frame. If there is no coherence, c is 1. With coherence, the redistribution cost is $cnOs_{raw_tri}$.

There are several components of the computation overheads. The first is the pre-transformation. A processor may transform a triangle to screen space only to find out that the triangle does not overlap the processor's portion of the screen. The processor then sends the raw triangle to a processor that will handle it, discarding the transformed triangle. This assumes that is cheaper to retransform the triangle instead of sending the transformed triangle, which is currently the case. The cost of this lost work is cnt_{pre_xform}, the number of triangles multiplied by the fraction of triangles needed to be redistributed and the time to transform a raw triangle.

Each processor that stores (or receives) a triangle transforms it each frame. This duplicated geometry processing results in an overhead of $n(O-1)t_{geom}$ since triangles that cross region boundaries are transformed more than once. In addition, all the regions that a triangle overlaps must be examined to make sure that the corresponding processor has a copy of the triangle. If the processor does not have a copy, then one is sent to it. The cost of this checking is nOt_{bucket}

Finally, there is additional cost due to screen space subdivision, in that the per-polygon setup and scanline processing must be repeated for each region that the polygon overlaps. The polygon setup portion gives a cost of $n(O-1)t_{tri}$. The repeated scanline processing is proportional to a one-dimensional version of the overlap factor. This is approximated by using the square root of the factor. Thus, each scanline must be processed $\sqrt{O}-1$ times, giving a cost of $n(\sqrt{O}-1)\bar{h}t_{line}$. The costs for sort-first are summarized in table 3.5.

3.2.3 Sort-Middle Overhead

Conceptually, the overhead for sort-middle is simpler than for sort-first. Because each triangle is on one processor, the geometry processing needs to be done only once. At the end of the geometry processing, the triangle must be put into the correct buckets. On average, each triangle is placed into O buckets, giving a cost of nOt_{bucket}. The redistribution size is proportionate to the bucketization cost, since all triangles are sent each frame. The size of the triangle given is the size of a Gouraud shaded triangle, allowing 16 bytes per vertex (floats for x, y, z plus bytes for r, g, b), plus 8 bytes of region identifier, primitive identifier, and primitive size (this allows for primitives other than triangles). The scan conversion and visibility costs are the same as sort-first, since they are also done using screen-space subdivision. The model of the overhead is summarized in table 3.6.

Pipeline Stage	Overhead Cost
Geometry processing	—
Bucketization	nOt_{bucket}
Redistribution	nOs_{disp_tri}
Scan conversion	$n(O-1)t_{tri} + n(\sqrt{O}-1)\bar{h}t_{line}$
Visibility	—

Table 3.6 Overhead costs in sort-middle algorithms. The communication cost is shaded to indicate that it is the amount of communication, not the time required.

3.2.4 Sort-Last-Sparse Overhead

In sort-last-sparse, there is no additional cost for geometry processing or scan conversion, as the work is done on a single processor (see table 3.7). During scan conversion, each pixel is placed in a bucket for redistribution to the correct processor, taking additional time proportional to the number of samples, $n\bar{a}S$. Since it is much easier to bucketize a single pixel compared to bucketizing a triangle, it should take much less time; thus, I omit this factor. The redistribution is also proportional to the number of samples, so the redistribution cost is $n\bar{a}Ss_{sparse_samp}$. The sample size s_{sparse_samp} is larger than the sample for sort-last-full, as each sample must contain the screen location of the sample as well as color and depth values. There is additional cost for visibility, even though the number of comparisons is the same as the serial algorithm. Each pixel takes longer to composite because the incoming pixels are in near-random order instead of being contiguous. The additional time is taken in calculating the address of the pixel in the processor's frame buffer, and by having to wait more often for values to be fetched from memory instead of cache.

Pipeline Stage	Overhead Cost
Geometry processing	—
Scan conversion	—
Redistribution	$n\bar{a}Ss_{sparse_samp}$
Visibility	$n\bar{a}St_{extra_comp}$

Table 3.7 Overhead costs in sort-last-sparse algorithms. The communication cost is shaded to indicate that it is the amount of communication, not the time required.

3.2.5 Sort-Last-Full Overhead

Like sort-last-sparse, sort-last-full has no additional cost for geometry processing or scan conversion. Since each of the N processors outputs a full screen of samples (AS samples), the redistribution cost is $NASs_{full_samp}$. Each processor does visibility tests for each sample during scan conversion, which is the same number of tests as the serial algorithm. So, each of the visibility tests of the redistributed samples is an additional cost, giving (N-1)AS or NAS additional compositing operations, depending on whether the first processor composites with an empty frame or not. We will use NAS for simplicity.

Pipeline Stage	Overhead Cost
Geometry processing	—
Scan conversion	—
Redistribution	$NASs_{full_samp}$
Visibility	$NASt_{comp}$

Table 3.8 Overhead costs in sort-last-full algorithms. The communication cost is shaded to indicate that it is the amount of communication, not the time required.

3.3 Comparisons of the Classes

The next three sections compare the various classes of algorithms. The comparisons will be made using the timing values and sizes shown earlier to calculate the relative overhead or time taken by the different algorithm classes. Only some of the variables will be bound because the equations are compared analytically. For simplicity, the average triangle height $\bar{h}$ will be set to $\sqrt{2\bar{a}}$ which assumes that each triangle has a square bounding box. The overhead costs are summarized in table 3.9.

Pipeline Stage	Overhead Costs			
	Sort-First	Sort-Middle	Sort-Last-Sparse	Sort-Last-Full
Pre-transform	cnt_{pre_xform}	—	—	—
Geometry processing	$n(O-1)t_{geom}$	—	—	—
Bucketization	nOt_{bucket}	nOt_{bucket}	—	—
Redistribution	$cnOs_{raw_tri}$	nOs_{disp_tri}	$n\bar{a}Ss_{sparse_samp}$	$NASs_{full_samp}$
Scan conversion	$n(O-1)t_{tri} + n(\sqrt{O}-1)\bar{h}t_{line}$	$n(O-1)t_{tri} + n(\sqrt{O}-1)\bar{h}t_{line}$	—	—
Visibility	—	—	$n\bar{a}St_{extra_comp}$	$NASt_{comp}$

Table 3.9 Overhead costs summary.

3.3.1 Sort-First vs. Sort-Middle

The main attraction of sort-first is its reduced communications cost. Of the four classes, it has the lowest communication cost if the amount of coherence is high (c is low). But, it has higher computation costs than sort-middle, because more of the pipeline is processed using screen-space subdivision, and thus the overhead factor applies to a larger part of the pipeline.

We can calculate the relative amount of computation overhead by using typical values for the overlap factor and average triangle size. I will use two values for the overlap factor, 1 and 2, which are the range of values seen later in the dissertation. I will also use two values for the average triangle size: 5 and 50 pixels. When the four combinations of the two pairs of values are placed in the above equations and compared, the amount of extra computation done by sort-first is either 1 + 1.2c, 1.3 + 0.1c, or 1.2 + 0.1c times the amount of work done by sort-middle. When there is no coherence to be exploited, for example the geometry is being generated for each frame, sort-first has between 1.3 and 2.2 times as much repeated computation as sort-middle.

This does not include the time for communication. It can be estimated by computing a per-byte communication cost. The Delta has a per-message overhead of 504 µs, and a per-byte time of 0.24 µs. If we assume 4096 bytes per message, this gives a cost of 0.36 µs per byte. When there no coherence, and using the same range of values for O and $\bar{a}$ as before, sort-first algorithms have between 45 to 64% more overhead than sort-middle ones.

Factors not expressed in the model will also have an impact. Sort-first load-balancing algorithms must trade off load-balancing efficiency against coherence. Because sort-middle load-balancing algorithms do not have this constraint, they should be able to balance the loads more effectively.

A second factor not in the model is the cost to support selective editing. Because sort-first migrates triangles between processors, there must be some mechanism to find the triangle being updated. If the display list is flat with no modeling transformations, the processor storing a triangle can be determined by transforming it to screen space. The actual location of the triangle within the processor could then be determined by searching a directory.

However, most retained display lists support hierarchy with modeling transformations, where a single triangle could be instanced multiple times, each with a different location on the screen. Each triangle instance needs to be associated with the attribute state in effect at each instance in the hierarchy. The attribute state includes a modeling matrix and triangle color. The modeling matrix is composed of one or more concatenated transformation matrices.

One way to associate a triangle instance with its attribute state would be to maintain a bi-directional link between each triangle and each instance's modeling matrix. Each modeling matrix also needs bi-directional links to all the transformation matrices that it is composed of. When a transformation matrix in the hierarchy is updated, all the associated modeling matrices must be updated, along with all the triangle instances associated with the updated modeling matrices. Then, the new screen-space values of each triangle are calculated, and the actual triangles updated using the same methods for a flat display list. The colors in the attribute states can be kept updated using a similar method that uses bi-directional links between each triangle and the color element that affects it.

Maintaining all the links will be fairly expensive compared to the cost of rendering the display list. It will take as much time to update the attribute state for a triangle as it would to do all of a triangle's geometry processing. This estimate is for a serial algorithm. The model must be distributed across multiple processors. Parallel algorithms to maintain the required data structures have not yet been developed, and will have higher costs. The costs will only be a small fraction of the geometry processing time when only a small fraction of the triangles is updated. The costs will be quite high if the model is regenerated in each frame.

The better algorithm class depends on the application space. Visualization requires support for editing; support for hierarchical display lists is quite desirable. Given the required bookkeeping costs, it is currently hard to tell whether sort-first will be faster than sort-middle when no changes are made to the model. If there are substantial changes, it appears that sort-middle algorithms will be faster. Since substantial or complete model changes are common in and important to visualization applications (as discussed in section 1.1.2), sort-middle algorithms appear to be the better choice.

3.3.2 Sort-Middle vs. Sort-Last-Sparse

The overhead in sort-middle algorithms is roughly proportional to the number of triangles multiplied by the overlap factor, while the overhead in sort-last-sparse algorithms is proportional to the number of triangles multiplied by the number of samples per triangle. The overhead can be estimated by setting the overlap factor to 1 and calculating the size of the triangle that makes the overhead the same for both sort-middle and sort-last-sparse. This approximation for O is reasonable because the triangle sizes turn out to be small. If one sample per pixel is used, and with the same redistribution costs as before, the overhead is the same with 4.8 pixels per triangle. With 16 samples per pixel, the overhead time is the same with 0.3 pixels per triangle.

Load-balancing is not factored in. The simple methods used by sort-middle are not sufficient for sort-last-sparse. I expect that the more complex methods that are necessary will be more expensive. However, this is hard to quantify since the papers that describe sort-last-sparse implementations with active load-balancing do not include load-balancing information.

As processor speeds and the number of processors in parallel systems both increase, systems will be able to process models containing more triangles. Since at least some of the additional triangles will be visible, the depth complexity will rise more slowly than the increase in triangles, and thus the average triangle size will decrease. This implies that sort-last-sparse will become faster than sort-middle at some point in the future. For the models in appendix A and a 640x512 resolution screen, the average triangle sizes range from 2.9 to 6.7 pixels, straddling the crossover point of 4.8 pixels. When using a screen resolution of 1280x1024, the triangle sizes are all above the 1.2 pixel crossover point. When performing anti-aliasing, sort-last-sparse must send each sample instead of each pixel (assuming brute-force supersampling is used), which can be viewed as increasing the screen resolution. With 16 sample (4x4 subpixel) anti-aliasing the effective triangle sizes are well above the crossover point.

3.3.3 Sort-Middle vs. Sort-Last-Full

The main characteristic of sort-last-full is that the amount of data each processor redistributes does not depend on the number of triangles, the size of the triangles, or the overlap factor. Instead, the amount depends on the number of samples used per frame. This means that the amount of redistribution and computation overhead is fixed for a given screen size and number of samples.

Unfortunately, the fixed amounts are quite large. With A = 640x512, S = 1, and O = 1.2, the amount of redistribution per processor for sort-last-full is the same as in sort-middle if each processor has 40,000 triangles. This is many more triangles than can be rendered interactively on current systems: individual processors in current parallel systems can render 15,000–25,000 triangles per second.

Sort-last-full has one useful characteristic, that the redistribution can be structured so that all communication is between adjacent processors. This can be done by arranging the processors in a pipeline, and having the end processor send its frame buffer and depth map to the next processor. That processor composites its frame buffer with the incoming one, and passes the data on to the next processor. The advantage of this is that the bisection bandwidth (the bandwidth across the center of the communication network, where one would cut to split the system into two equal portions) of the system does not need to increase as the performance increases. This characteristic is unique among the algorithm classes. The other algorithm classes all

have global redistribution steps, which means that the bisection bandwidth will increase as the performance increases.

The bisection bandwidth is simple to model for a 2D mesh given two assumptions:

1. Each processor sends the same amount of data to each other processor.

2. The processors are arranged in a square.

Without loss of generality, we can also assume that the messages are routed first horizontally and then vertically. The peak bandwidth is used in the middle of the processor array, where there are $\sqrt{N}$ links. On each row of the array (which has $\sqrt{N}$ processors), each processor on the left half of the array will send one half of all of its data over the link dividing the left and right halves of the array. So, if each processor sends b bytes per frame, the link bandwidth in each direction is $b\sqrt{N}/4$. At 10 frames per second, A, S, and O set as above, and a sort-middle rendering rate of 25K triangles per second per processor, the two systems will have the same per-link bandwidth when N = 3895, which is roughly a 62x62 processor array. With A = 1280x1024, N = 62330. Even with a small screen, N is larger than current systems; with the large screen, it is nearly 50 times the size of current large systems. Thus, the scalability advantage of sort-last-full algorithms does not apply to systems seen today, or in the near future.

3.4 Summary

In conclusion, sort-first algorithms are not suitable for scientific visualization applications because these applications require extensive editing. Provisions for editing will be expensive and will make sort-first algorithms significantly slower than sort-middle algorithms when the model is regenerated each frame (when there is no coherence to exploit). Sort-last-full algorithms have a very high, fixed amount of neighbor-to-neighbor communication. While this allows those algorithms to scale to large numbers of processors, the large fixed amount is not practical for current systems.

Sort-middle and sort-last-sparse algorithms have applications on current systems. The latter works well with very small triangles, ones with less than about 10 pixels per polygon. When rendering NTSC (640x512) resolution images, many complex models have triangles that fall into that range. However, when using workstation resolution (1280x1024) or performing anti-aliasing, the triangles are larger, and sort-middle algorithms will generally be faster. Because of this, I expect sort-middle algorithms to be more suitable for scientific visualization applications, and thus I will investigate sort-middle further in the next chapters.

In the future, the number of triangles in models will increase with future increases in system performance. However, the depth complexity will not increase at the same speed, making the average polygon size decrease. So, at some point, sort-last-sparse will have the same redistribution costs as sort-middle, and it will become the preferred algorithm.

CHAPTER FOUR

SORT-MIDDLE ALGORITHMS

This chapter begins the analysis of the choices that a designer must make when devising a sort-middle algorithm. Here, we will look into the choices that influence the overall structure of the algorithm. We will also investigate different methods for redistributing triangles between processors. These two topics will give a foundation for Chapters 5 and 6, which investigate load-balancing methods. The various choices made here will be done using arguments and mathematical models of the time required instead of using more detailed methods such as simulation. A taxonomy of the choices covered in this chapter appears in figure 4.1.

The algorithm's overall structure impacts how well it can be scaled. Different styles of assigning regions to processors have vastly different costs. The total costs can be $O(1)$, $O(N \log N)$, or even $O(N^2)$ (N is the number of processors). The styles with higher costs can be impractical when using hundreds of processors. Different redistribution methods also have varying amounts of overhead. Some methods have overhead costs of $O(N^{3/2})$ while other have $O(N^2)$ costs. With many processors, the methods with higher overhead can also be impractical.

The design choices are not independent; a choice in one area can limit the options for another. This makes it impossible to discuss all the choices independently. Each section in this chapter describes one design choice, and how it interacts with choices covered earlier. The next three sections discuss the following:

- Section 4.1: whether all processors work on all tasks.
- Section 4.2: when the various tasks in the algorithm are performed.
- Section 4.3: when regions are assigned to processors.

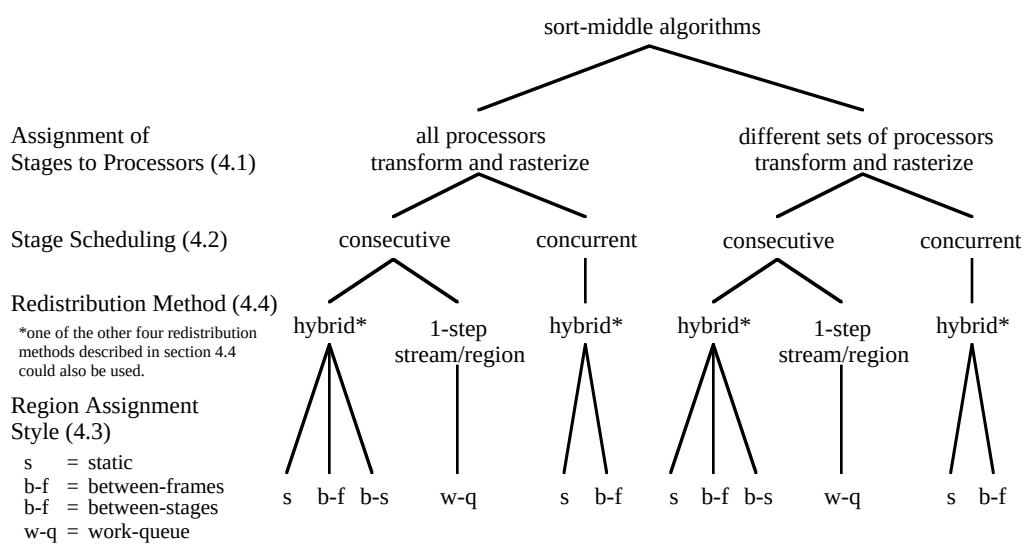


Figure 4.1 Taxonomy of the sort-middle algorithms encompassed by the design choices considered in this chapter. The numbers indicate the section where each choice is described.

Section 4.4 will describe different methods for redistributing the triangles between processors, and the last section will summarize the best choices for the chosen system and application. Since, for our application, we would like interactive performance when using many processors, the best choices will be the ones with a small amount of overhead. Choices with a large amount of overhead would take too long since the frame time is limited. With other applications, the best choices could be different. For example, applications with lower frame rate requirements or smaller target systems might trade higher overhead for other desirable features.

4.1 Assigning Stages to Processors

One issue independent from the design of the graphics algorithm is whether the graphics library software runs on the same set of processors as the application; the alternative is to run the application and graphics tasks on separate sets of processors. Running the graphics software on separate processors adds additional communication. The additional communication would be costly if an immediate mode interface to the graphics library is used, since all of the primitives would be sent each frame. The communication would also be costly when a retained-mode interface is used and substantial changes are made to the geometry.

Additionally, running the graphics software on a set of processors makes it impossible for memory and processing time to be shared between the application and graphics software. Running the application and graphics separately would make it likely that one set of processors would be idle at times. However, using two sets of processors does decrease the amount of overhead since the respective tasks would be partitioned among fewer processors. Because the reasons for using one set of processors outweigh the single reason for two sets of processors, I will assume that the application and graphics software share one set of processors.

A related issue is how the stages of the sort-middle graphics pipeline should be assigned to processors. As described in the previous chapter, sort-middle algorithms have three stages: geometry processing, redistribution, and rasterization (see figure 3.3). The structure of sort-middle algorithms dictates that all of the processors participate in the redistribution. Two options exist for scheduling the other two stages. A sort-middle algorithm could have all processors perform both the geometry processing and rasterization. Or, it could divide the processors into two groups, with one group running the application and performing the geometry processing, and the second group performing the rasterization.

Keeping the processors all in one group means that changes in the amount of geometry processing relative to the amount of rasterization will not cause a load imbalance, which would be the case if the tasks were handled by separate groups of processors.

On the other hand, splitting the processors into two groups can reduce overhead in the algorithm. The load-balancing job is simpler because each task is divided among fewer processors. The redistribution cost is also reduced. When using a simple redistribution method, each geometry processor sends a message to each of the rasterization processors. If the N processors are split evenly into two groups, each processor must send $N/2$ messages. However, if the processors are not split into groups, each processor must send N messages.

In most cases, the improved load-balancing when using one set of processors outweighs the larger amount of overhead. The ratio between rasterization and geometry processing workloads can easily vary by a factor of 3 across different datasets and different viewpoints. This means that one set of processors may by idle half or more of the time. The increase in overhead only appears to outweigh the improved processor utilization in extreme cases: when using hundreds of processors on relatively small (less than 100,000 triangles) data sets. Thus, using all of the processors for both the geometry processing and rasterization appears to be the better choice.

Several other parallel renderers for general purpose parallel systems have all processors perform all the tasks [Robl88, Croc93, Orte93, Whit94, Cox95]. A specialized graphics multicomputer [Fuch89] assigns different tasks to different sets of processors because the system has two different processor types, each specialized to its assigned task. Chapter 8 revisits the topic of assigning different tasks to sets of specialized processors.

4.2 Scheduling the Stages

Given that all processors will perform all three stages of the rendering algorithm, we must also decide how to schedule the execution of the stages. Of the stages, the redistribution stage is the most straightforward to schedule. The redistribution stage performs communication while the other two stages of the pipeline do calculations. In most multicomputers, the communication network is decoupled from the processors, so it is possible for calculations and communication to be done at the same time. Many multicomputer operating systems support this with asynchronous message passing calls. Instead of immediately performing the actual operation, these calls queue the send and receive operations.

Because the redistribution stage uses a separate resource, it should be overlapped with the other stages so as to minimize the overall frame time. Increasing the amount of overlap will, up to some point, decrease the overall frame time since it reduces the amount of time the processors spend waiting for triangles to rasterize. The maximum overlap of the computation and calculation occurs when each triangle is redistributed as soon as it is transformed, sending each in its own message. This is extremely expensive in most systems, where the cost per byte of small messages is much larger than the cost for large messages (as shown in Appendix B, [Bokh90], and [Kwan93]). Instead, several triangles should be buffered and then sent in a single message. The optimum size of the messages is discussed in section 4.4.7. Two previous multicomputer implementations that overlap the redistribution are Pixel-Planes 5 [Fuch89] and Crockett and Orloff's iPSC/860 implementation [Croc93]. Most of the sort-middle-like hardware implementations also overlap the redistribution [Fuch79, Park80, Akel88, Akel93]. Roble's multicomputer [Robl88] and Whitman's shared-memory [Whit94] implementations do not overlap the redistribution.

The remaining two stages, geometry processing and rasterization, can either be run concurrently or consecutively. If they are run concurrently, a processor switches processing between the stages fairly frequently, perhaps every few triangles. This will reduce the memory requirements by reducing the number of redistributed but not yet rasterized triangles. Because the geometry processing finishes near the end of each frame, the buffers containing unsent triangles must also be flushed near the end of each frame. If the processors' workloads are nearly balanced, all the flushes will be done in a short time period, congesting the network. This congestion will often make some processors wait until the final triangles arrive. The extent of this congestion depends on the number of processors. The bisection bandwidth does not scale with the number of processors, making the congestion worse with many processors. Increasing the number of processors also increases the percentage of triangles that are sent when the buffers are flushed if a constant model size and message size are assumed. With hundreds of processors, it is possible that triangles are only redistributed when the buffers are flushed, causing the stages to be effectively run consecutively.

If the stages are run consecutively, the geometry processing stage completes near the middle of each frame, and the buffers are flushed at the stage's completion. This allows time for the network congestion to clear before the end of the frame. In many cases, a processor will be able to rasterize polygons received earlier in the frame while the later polygons are arriving. However, this comes at a cost of additional memory. An implementation must be able to store all the transformed triangles before starting the rasterization. The other option, running the stages concurrently, only needs a limited amount of buffering.

There are three other differences. Running the stages concurrently requires that the region-to-processor assignment be fixed for the entire frame. Also, running the stages consecutively will improve the cache coherency as compared to running the stages concurrently. When only one stage is run at a time, only its data needs to be in the processor's cache. This will affect the performance when running on processors with small caches. Finally, running the stages concurrently requires that the processors instantiate all of their per-region frame buffers during the rasterization. Depending on the other design choices, a processor may be able to render one region at a time when the stages are run consecutively. Rendering one region at a time will save a significant amount of memory if the regions are large.

The differences between the two scheduling options are summarized in table 4.1. The table also has references to other work that uses each scheduling option.

4.3 Region Assignment

The third design choice is the time when regions are assigned to processors, or *style* of region assignment. The regions can be assigned once, at compilation time; statically, once per frame; or dynamically, before the rasterization of each region. There are two once-per-frame styles, one that does the region assignments

Characteristic	Run stages concurrently	Run stages consecutively
Buffer flushes	At end of frame	In middle of frame
Memory use for buffering triangles	Limited	Enough to hold all transformed triangles
Choice of region assignment style	Limited	More options
Requires instantiating all region frame buffers at once	Yes	No
Cache coherency	Fair	Good
Examples	Fuch79, Park80, Akel88, Akel93, Cox95, Croc93, "MEGA"	Robl88, Fuch89, Whit94

Table 4.1 Summary of geometry processing and rasterization stage scheduling characteristics.

between the geometry processing and rasterization stages, and another that performs the assignments between frames. The four assignment styles are discussed in the next four subsections. Later subsections describe hybrid assignment styles, compare the costs of the four styles, and summarize the discussion.

4.3.1 Work-Queue Region Assignments

The first region assignment style makes the assignments between the rasterization of each region, using a work-queue approach. During each frame, the geometry processing is first performed. Then, each processor is assigned a region to rasterize. When a processor completes a region, it is assigned another region. This continues until all regions have been rasterized.

When a region is assigned to a processor, that association must at some point be sent to all the other processors so that they can send their triangles for rasterization. When a processor is finished with a region, it sends a message to the processor assigning the regions so that another region can be assigned.

This could be implemented using three tasks. One task running on one processor would handle the region assignments. It would initially assign a region to each processor and assign the remainder as the processors finish their assignments. A second task running on every processor would accept incoming region assignment messages and send triangles for the specified region to the designated processor. The third task also runs on every processor. It accepts messages that tell the processor it has been assigned a new region and then rasterizes the region.

Algorithms using work-queue assignments can balance the workloads more evenly by assigning regions from the most time-consuming to the least. If this is not done, the last region assigned could be especially time-consuming, which means that it will be finished after the other processors have finished. This will decrease the processor utilization if the early-to-finish processors wait for the entire frame to be completed before proceeding to the next frame. Assigning regions in sorted order comes at the cost of collecting per-region processing time estimates (perhaps the number of triangles per region) from all the processors, and sorting them before rasterization can start.

The main advantages of this style are that the load-balancing is automatic, and that it uses the actual rendering times instead of estimates of how long each region will take to render. The other styles do not have these advantages. The style has one minor disadvantage, that it only performs redistribution during rasterization, instead of during the entire frame (unless one frame is rasterized while another is being transformed, e.g., [Fuch89]). The main disadvantages are (1) gathering the region costs and sorting the regions places a global synchronization and some serial computation in the middle of the frame, and (2) large systems will need use many messages to inform the processors of the region assignments, which will be expensive. The number of messages received by each processor will be large because at least one region is needed for each processor.

A variant of this assignment style modifies the strategy towards the end of a frame. Once all the regions have been assigned to a processor, when a processor finishes a region the work for an already assigned region is split, "stealing" work from another processor. If a scanline rendering algorithm is used, the work is split by splitting the region that is being rendered and classifies the remaining triangles according to the new boundaries. If a *z*-buffer algorithm is used, the work can be split by giving the new processor some of the triangles, having processor render them, and then return the samples to the original processor. Whitman has implemented the first algorithm [Whit94], and Cox the second [Cox95]. However, this work-stealing

strategy will have higher region assignment costs compared to the normal work-queue assignments. Since the work-queue assignment method already has rather high assignment costs (as will be seen in a later section), this variant will not be further discussed.

4.3.2 Assignments of Regions Between Geometry and Rasterization Stages

When regions are assigned between the geometry and rasterization stages, estimates of how long each region will take to rasterize are summed across all the processors after the geometry processing has finished. These estimates are used to assign regions to processors so that the load is balanced. Then, the region assignments are broadcast to all the processors, which send their triangles to the appropriate processors and start the rasterization.

The regions can be assigned using a greedy multiple-bin-packing algorithm. The region time estimates are first sorted. Then, the regions are assigned from the most complex to the simplest. At each step the region is assigned to the processor with the lightest load. The result is similar to using work-queue assignments except that estimated processing times are used instead of the actual times. The greedy assignment algorithm is fast, but does not find the assignments that optimally balance the loads between processors. Finding the optimal assignments is NP-complete, and thus would take too long when there are more than a few processors. I expect that heuristics could be applied to improve the greedy algorithm but have not investigated them.

This style uses fewer messages than the previous style. Its main disadvantage is that the load-balancing step places a global synchronization and some serial computation in the middle of the frame. Like the previous style, it also does not allow the entire frame time to be used for redistribution.

4.3.3 Assignments of Regions Between Frames

As in the previous style, the between-stages assignment style collects per-region time estimates and makes the region assignments after the geometry processing has completed. However, the assignments are not used for the current frame; instead, they are used for the next frame. This style depends on having the assignments made using one frame's time estimates be reasonably correct for the next frame. This is generally true for interactive applications, as the distribution of polygons on the screen generally has frame-to-frame coherence.

This style has the advantage that redistribution can be done during the entire frame, and that the load-balancing operation can be overlapped with other processing. The data collection and load-balancing calculations can be done by one processor while the others are rasterizing. Its main disadvantage is that region assignments computed from the previous frame's time estimates will result in a larger load imbalance as compared to assignments made using the current frame's estimates. In section 6.3.5, comparisons of simulations that use the same load-balancing methods except that one assigns regions between stages and the other assigns regions between frames shows that between-stages assignment has up to 38% higher processor utilization. The rasterization overhead is the same for the pairs of load-balancing algorithms because the algorithms use the same region boundaries.

4.3.4 Static Region Assignment

The last style considered uses static assignment of regions to processors. Its main advantage is that no time is taken to compute or communicate the region assignments. A second advantage is that it allows the entire frame time to be used for redistribution. However, the lack of load-balancing will reduce the processor utilization during rasterization. When the region boundaries are held constant, assigning regions between frames can have twice the processor utilization compared to static region assignments; the average increase is 28%. Again, the rasterization overhead is the same for the pairs of load-balancing algorithms because they share region boundaries.

4.3.5 Hybrids

The four assignment styles discussed are not the only ones possible. Hybrids between the styles are possible. For example, some regions could be assigned using one of the once-per-frame assignment styles, and the rest assigned using the work-queue assignment style. The hybrid styles have hybrid tradeoffs, with some of the assets and liabilities of each. For simplicity, the remainder of the dissertation only discusses the four primary styles.

Cost	Work Queue	Between Stages	Between Frames	Static
Cost gathering (latency)	0 or $O(r \log N)$	$O(r \log N)$	$O(r \log N)$	0
Calculation (minimum)	0 or $O(r \log r)$	$O(r \log N + r \log r)$	$O(r \log N + r \log r)$	0
Communication messages	$Nr + N$	$2N$	$2N$	0
Communication bytes	$O(Nr)$	$O(Nr)$	$O(Nr)$	0

Table 4.2 Costs of different rasterization region assignment styles for r regions and N processors. The first two costs for the "work queue" are non-zero when the regions are assigned in sorted order.

4.3.6 Region Assignment Cost Comparison

Each of the assignment styles has different costs; they are summarized in table 4.2. The cost gathering row gives the amount of time taken to gather per-region cost estimates. The first three entries the time needed to gather the per-region costs using a tree with log N levels. The next row gives the amount of calculation. The $O(r \log r)$ component is for sorting the regions, and the $O(r \log N)$ component is for assigning the regions using a heap to find the most lightly loaded processor at each step. The last two rows give the amount of communication needed for both gathering the estimates and broadcasting the assignments. The work-queue assignment style sends a message to each processor for each region plus each processor sends a per-region cost message. The once-per-frame styles send a cost message and receive a message with the region assignments.

Work-queue region assignment is the most expensive due to the increased communications. The two once-per-frame styles have the same costs, and the static assignment has no region assignment costs.

By providing values for the constants implied in the order notation used in table 4.2 we can get a better feel for the different styles' costs. I will only calculate the costs related to processing time because figuring the amount of time the processors spend waiting for incoming messages depends on the structure of the overall algorithm, which has not yet been fixed. However, the costs shown can be used as a lower bound of the actual costs. The time taken by load-balancing will be revisited in Chapter 6 without these assumptions.

The time required for calculations, for sorting the regions and assigning them to processors, are based on times measured on the Pixel-Planes 5 i860 processors. The region sorting time is taken from the equation $t_{sort_1} + t_{sort_r} r + t_{sort_r\log r} r \log_2 r$, and the assignment time from the equation $t_{assign_1} + t_{assign_N} N + t_{assign_r\log N}(r-N)\log_2 N$. The constants were computed computing by fitting the equations to measured times using least-squares minimization. The $\log_2 r$ portion of the assignment time equation is multiplied by $r-N$ instead of r because the first N regions can be assigned in constant time. The work-queue assignment style only sorts the regions while the once-per-frame styles perform both calculations.

The time required for communication assumes that the once-per-frame and work-queue styles have every processor transmit a message containing the per-region time estimates, with four bytes for each estimate. The work-queue assignment style uses a four-byte message to inform every processor when a region has been assigned. The once-per-frame styles send a single large message containing all the assignments (using four bytes per region) to all of the processors. The communication costs have a per-message cost, t_{msg}, and a per-byte cost, t_{byte}. Appendix B describes how these two costs were measured on the Intel Touchstone Delta.

The assignment styles assume that the number of regions is a constant multiple (4) of the number of processors; this is what is chosen in Chapter 6.

Symbol	Value	Description
t_{sort_1}	31.5 µs	sorting time that is independent of r
t_{sort_r}	1.25	sorting time that scales with r
$t_{sort_r\log r}$	0.473	sorting time that scales with $r\log_2 r$
t_{assign_1}	103	assignment time that is independent of N
t_{assign_N}	0.956	assignment time that scales with N
$t_{assign_r\log N}$	0.671	assignment time that scales with $(r–N)\log_2 N$
t_{msg}	504	per-message communication time
t_{byte}	0.24	per-byte communication time

Table 4.3 Symbols used for illustrating the region assignment costs.

The formula for the cost to make work-queue region assignments is

$$t_{lb_wq} = t_{sort_1} + rt_{sort_r} + t_{sort_r\log r} r\log_2 r + (Nr + N)t_{msg} + (2\cdot 4)Nrt_{byte},$$

and the formula for once-per-frame assignments is

$$\begin{aligned} t_{lb_of} &= t_{sort_1} + rt_{sort_r} + t_{sort_r\log r} r\log_2 r + t_{assign_1} + Nt_{assign_N} + t_{assign_N\log N}(r-N)\log_2 N \\ &+ 2Nt_{msg} + 4Nrt_{byte}. \end{aligned}$$

These formulas follow from the formulas in the appropriate columns of table 4.2.

Even though only the processing cost is considered, the formulas used show an estimate of the overall cost. The formulas assume that the costs are equally borne by the processors. This is not true, as the region sorting and region assignment are usually done on one processor. Also, when collecting the per-region time estimates in a summing tree, some processors will receive multiple messages while others will not receive any. Assuming that the costs are equally born underestimates the true cost. However, the costs are also overestimated: messages that are sent to every processor could use the operating system's broadcast triangle, which would have lower cost.

The costs are illustrated in figure 4.2, which plots the average per-processor load-balancing cost for a range of numbers of processors. The chart has one curve for the work-queue style and a second for both once-per-frame styles. The difference between the curves shows that work-queue assignment is more expensive than once-per-frame assignment. With small systems, neither style takes much time; the once-per-frame style is roughly ten times faster. With 512 processors, the once-per-frame style is more than two orders of magnitude faster.

The costs of the once-per-frame assignment styles could be reduced by taking advantage of coherence. By assuming that the triangles' screen-space distribution is uniform throughout the geometry processing, once-per-frame region assignment could be done during the geometry processing. This would be done by collect-

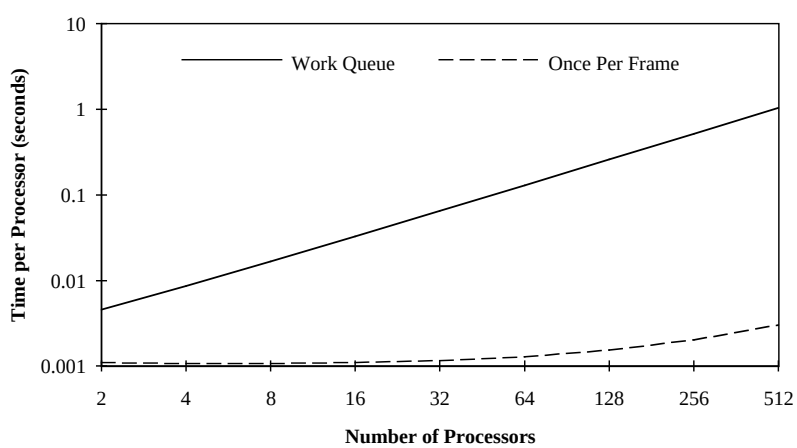


Figure 4.2 Total processing cost for load-balancing.

ing the per-region costs after the geometry processing has processed only a fraction of the triangles, and then using those costs to make the region assignments and broadcast them. Once a processor receives the region assignment message it can start the triangle redistribution and continue like the between-frames assignment style. Coherence can also be used by only collecting per-region costs from some of the processors. This strategy assumes that the distribution of the triangles' screen space locations is the same for all the processors.

The coherence in each case is generally not true when an immediate mode interface is used. Most immediate-mode applications give each processor one portion of the overall data, a portion that occupies a small part of world space. The triangles generated by each processor are thus not correlated. Also, applications usually progress sequentially through the data, so the distribution of triangles on the screen at the start of the geometry processing is different from the distribution at the end. However, if a retained-mode interface is used, the triangles can be randomly distributed across the processors to increase the coherence.

The decreased load-balancing cost due to using coherence comes with larger load imbalances. The extent of the load imbalance decrease must be measured or calculated before using either strategy. The second strategy is explored further in section 6.5 and 7.2.3.

4.3.7 Summary of the Region Assignment Styles

The properties of the assignment styles (other than their costs) are summarized in table 4.4. For completeness, the third property specifies whether a certain redistribution method is required even though redistribution has not yet been discussed.

The style of choice depends on several factors. It depends on how well the styles can balance the loads in addition to the cost of the region assignment style. The style depends on the number of processors since the assignment cost depends on the number of processors. The best style also depends on the frame rate, since with a high frame rate there is not as much time to perform the load-balancing before it affects the performance. Finally, the best style depends on the amount of frame-to-frame coherence. If there is sufficient coherence, the advantages gained with an algorithm that assigns regions between frames will outweigh the poorer load balance. None of these issues have yet been discussed.

However, given the factor of 10 to 500 difference in costs between once-per-frame and work-queue assignments, it is clear that the less costly styles will be better with large systems or high frame rates. The crossover point between the different styles will be determined using the simulations described in Chapter 6; the actual discussion is in section 6.5.

4.4 Redistribution

The final, and most complex, algorithm design choice is the structure of the triangle redistribution. The three redistribution options for sort-middle algorithms are:

Characteristic	Work Queue	Between Stages	Between Frames	Static
Allows overlapping computation stages for same frame	No	No	Yes	Yes
Allows redistribution during the entire frame	No	No	Yes	Yes
Requires stream-per-region one-step redistribution	Yes	No	No	No
Stalls during load-balancing	If sorted assignment used	Yes	No	No
Uses actual rendering times	Yes	No	No	No
Depends on coherence	No	No	Yes	No
Examples	Fuch89, Whit94	Robl88	-	Croc93, Fuch79, Park80, Cox95, "MEGA"

Table 4.4 Characteristics of the region assignment styles.

- the minimum number of messages sent and received per processor,
- whether the messages are sent hierarchically, and
- the maximum message size.

The first two options can make a large difference in the overall speed when running with many processors. When running the PLB Head model (see Appendix A) using 256 processors on the Intel Touchstone Delta, a run using the best values for the first two options was 40 times faster than a run with the worst parameters. Although the two runs cannot be exactly compared because other optimizations were made, I believe a factor of 15 difference is due to the redistribution parameters. Section 7.3.1 shows more experimental results. The third option, message size, is less critical as long as the messages are reasonably large. Experiments shown in section 7.3.3 show at most a 18% difference in performance between the best and worst parameters tried.

The next two sections discuss the first two options, the number of streams sent per processor and whether to use hierarchical sending. Then, two sections calculate and then illustrate the time required by the four possible combinations of the first two redistribution options. The next section develops a better hierarchical redistribution method, and the final section discusses choosing the best message size.

4.4.1 Number of Message Streams Sent Per Processor

In sort-middle algorithms, each processor needs to send data to every other processor. The data could be sent as one large message, but that would complicate memory management, and would not overlap the communication with the geometry processing. Instead, each processor sends a stream of messages to the other processors. Two options are possible: the processors can either send a stream of messages to each of the other processors, or they can send a stream of messages for each screen region. With the first option, during geometry processing each processor maintains one buffer per processor, sending it when full. With the second option, each processor maintains one buffer per region. The options affect the algorithm's overall memory usage and redistribution costs.

When sending a stream of messages for each processor (called *stream-per-processor* redistribution), each message generally will hold triangles that overlap several regions, the regions assigned to the destination processor for rasterization. During geometry processing, a straightforward stream-per-processor implementation would place triangles into the appropriate processor's buffer without considering which of the processor's regions the triangle overlaps. When a processor rasterizes the triangles in the buffer, it could either instantiate a color and *z* buffer for all the regions assigned to it, and render all the triangles in a given message. Or, a processor could make multiple passes over the message. During each pass, it would instantiate one or more of the per-region color and *z* buffers, and rasterize the triangles that overlap the instantiated buffers. The second method would take more time since making multiple passes causes the cache to be less effective.

Note that triangles could be sent with the triangles in each message sorted by region. This would simplify rasterizing one region at a time. However, this would either require that the triangles be copied into the message buffer, or require that the operating system have a function that does the assembly during the transmission. Since the Delta architecture does not have such a function, and because copying all the triangles or making multiple passes over the triangles would be fairly expensive, I only consider stream-per-processor redistribution with messages containing unsorted triangles.

When a stream of messages is sent for each region (called *stream-per-region* redistribution), each message contains triangles that overlap a single region. This makes it straightforward to rasterize one region at a time.

To summarize, in theory both options allow regions to be rendered one at a time, but in practice only the second option, stream-per-region redistribution, allows it. Rendering one region at a time requires less memory than rendering the regions all at once. When using only a few processors, the regions are large, so instantiating all the regions could exceed the amount of memory for a processor. This is especially true when calculating a high resolution image, or when anti-aliasing. Also, rendering one region at a time would make the cache more effective, decreasing the overall time by a small percentage.

However, communication costs are lower with stream-per-processor redistribution. Since there are more regions than processors, stream-per-processor redistribution would have less per-message overhead. The amount of savings is minimal when using only a few processors, since nearly all the messages will be of maximum size. But, when using many processors, the messages are very small, making the per-message cost dominate the communication cost. Because of this, the reduction in the number of messages can result in substantial savings.

Thus, when using a few processors, it is best to use stream-per-region redistribution, because it will save memory used for frame buffers and improve cache coherence. Once there is enough memory to instantiate all the per-region color and *z* buffers, it is best to use stream-per-processor redistribution. However, when using work-queue region assignments, only stream-per-region redistribution can be used, since each processor only works on one region at a time.

4.4.2 Two-Step Redistribution

In the last section, the minimum number of messages is either rN or N^2. This means that, if the number of processors is increased, the minimum number of messages sent by each processor will increase, thus increasing the per-frame overhead. With a sufficiently large number of processors the per-message costs will dominate the total communication cost.

A solution has been known in the parallel processing community [Kuma94]. The redistribution step in sort-middle algorithms is a type of personalized all-to-all communication, where every processor sends a single, different message to every other processor. The solution only covers the case where the message size is fixed, and for only selected system sizes. That is, for a mesh network, the algorithm only works if, for some k, there are k^2 processors; for a cube there are k^3 processors, etc. The work here extends the algorithm to allow each processor to send more than one message (e.g., with stream-per-region redistribution), to allow messages of varying sizes, and to allow an arbitrary number of processors.

The standard solution is to first route the data in one dimension in the network. The data is then copied into a new set of messages, with one message per processor in the next dimension, and then sent to the processors. This continues for each dimension in the network. For a 2D mesh, this can be viewed as splitting the processors into groups, with all the processors in a row or column in the same group. In the first step, each processor exchanges data with processors in other groups. Each processor sends data only to one processor in each other group. In the second step, each processor exchanges data within its own group. Figure 4.3a gives an example of a 4 by 4 mesh of processors split with each column forming a group. In the figure, the groups are separated by thick lines and the processors that exchange messages in the first step are shaded similarly.

In the more general case, the grouping is not necessarily done by cutting in only one dimension. Instead, for a 2D mesh, the processors are grouped by cutting the processor mesh from top to bottom and from right to left, giving a number of rectangular groups. This method gives more choices for the number of groups. However, it also allows the groups to contain different numbers of processors, which causes load imbalances that will be addressed in a later section. I call this method *two-step redistribution*, and the simpler method described in section 4.4.1 *one-step redistribution*. One- and two-step redistribution can be com-

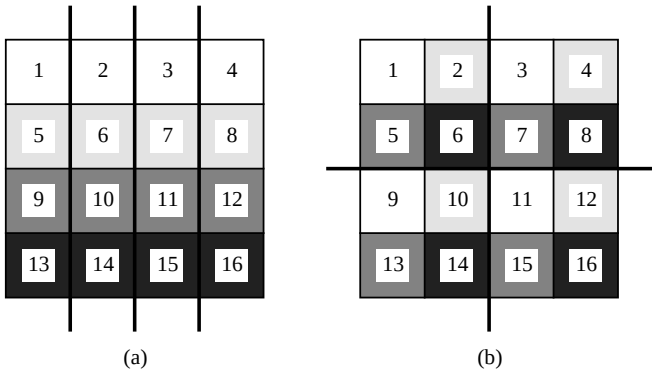


Figure 4.3 Two examples of grouping for two-step redistribution. The thick lines separate groups of processors. Processors that exchange messages in the first step are shaded the same.

bined with either stream-per-processor or stream-per-region redistribution.

Two step redistribution should not be used in all cases. While it does reduce the minimum number of messages, it increases the per-byte communication costs since each triangle must be sent twice, and it also requires that the triangles be copied. It only should be used when it reduces the overall costs. The decision whether to use two-step redistribution will be made by using a performance model of the communications, which will be developed in the next sections. Note that systems with relatively low per-message communication costs may not see any speedup when using two-step redistribution. Also, two-step redistribution cannot be used with work-queue region assignment because the processors are only assigned one region at a time.

4.4.3 A Simple Method for Choosing the Number of Groups

We would like to show the costs of the redistribution methods, but to do this we must know how many groups to use. In this section, we will make a simplifying assumption to get a first cut at the number of groups. We will revisit calculating the number of groups in a later section, without using the simplifying assumption. Here, we assume that messages have unlimited length, that a processor only sends one message per region or processor per frame. In this case, in the first step each processor sends one message to each of the other g groups, giving a total of $N(g-1)$. The second step depends on whether stream-per-processor or stream-per-region redistribution is used.

With stream-per-processor communication, in the second step each processor sends one message to each of the other $N/g-1$ processors in its group. Thus, the total number of messages m sent by all the processors is

$$m = N(g-1) + N(N/g-1).$$

To find the minimum number of messages, differentiate with respect to g, giving

$$\frac{\partial m}{\partial g} = N - \frac{N^2}{g^2}.$$

Setting $\partial m/\partial g$ to 0 and solving for g gives $g = \sqrt{N}$.

When using stream-per-region redistribution, in the second step of the redistribution each processor sends a message for each of the regions assigned to the other processors in the group. On average, each group is assigned r/g regions, for a total of rN/g messages. However, no processor sends triangles for its own regions. Putting these together, the total number of messages is $m = N(g-1) + r\,N/g - r$. Thus, with two-step stream-per-region redistribution, the minimum number of messages occurs when $g = \sqrt{r}$.

4.4.4 Costs of Different Redistribution Methods

A simple performance model illustrates the savings offered by two-step redistribution. Let us start with one-step stream-per-processor redistribution. I will assume that each of the processors is sent the same number of triangles, which would tend to be true if the rasterization loads were balanced. The n/N triangles assigned to each processor results in nO/N triangles per processor after the geometry processing (recall O is the overlap factor), which is then split into N streams of messages, one for each processor. If the maximum number of triangles per message is n_{msg} then each processor will send $\lceil nO/(n_{msg}N^2) \rceil$ messages to each of the other processors, for a total of $N\lceil nO/(n_{msg}N^2) \rceil$ messages per processor. To get the total cost, multiply the number of messages by the number of processors N and the per-message cost t_{msg}, and add in the per-byte costs. The latter is the number of triangles, n, multiplied by the overlap factor, the size of the triangle s_{tri}, and the per-byte cost t_{byte}. The total cost is

$$N^2 \left\lceil \frac{nO}{n_{msg}N^2} \right\rceil t_{msg} + nOs_{tri}t_{byte}.$$

The formula when using stream-per-region redistribution is similar if we make the simplifying assumption that each processor sends the same amount of data per region. (This assumption is usually invalid, and will tend to underestimate the total costs. However, the underestimate will not affect the results in the next section.) So, the cost for this case is

$$rN\left\lceil \frac{nO}{n_{msg}rN} \right\rceil t_{msg} + nOs_{tri}t_{byte}.$$

To derive the costs when using two-step redistribution, observe that the optimum values for g in the previous section have each processor sending the same number of messages in each step. In the first step, each processor divides its nO/N triangles into streams of messages, one stream for each group of processors, giving $Ng\left\lceil nO/(n_{msg}Ng) \right\rceil$ messages from all N processors ($g = \sqrt{N}$ or $\sqrt{r}$, depending on the redistribution method). To get the final cost, multiply the number of messages in the first step by two to get the total number of messages, double the per-byte costs in the formulas above (since everything is sent twice), and add in the cost of reformatting the messages from one step to the next. The per-triangle cost of the reformatting is given by t_{refmt}. The costs for two-step stream-per-processor and stream-per-region redistribution are respectively given by

$$2N\sqrt{N}\left\lceil \frac{nO}{n_{msg}N\sqrt{N}} \right\rceil t_{msg} + 2nOs_{tri}t_{byte} + nOt_{refmt},$$

and

$$2N\sqrt{r}\left\lceil \frac{nO}{n_{msg}N\sqrt{r}} \right\rceil t_{msg} + 2nOs_{tri}t_{byte} + nOt_{refmt}.$$

4.4.5 Redistribution Cost Examples

Figures 4.4, 4.5, and 4.6 plot these equations, each with a different model size. The plots use $G = 6$, $n_{msg} = 73$ triangles, $t_{msg} = 504$ µs, $s_{tri} = 56$ bytes, $t_{byte} = 0.96$ µs, and $t_{refmt} = 0.05$ µs. The value for O is computed using equation 3.1 and assumes that the regions are square, that $w = h = 3.5$, and that the screen resolution is 640 by 512. All the timing values except t_{refmt} were measured for the Intel Touchstone Delta; t_{refmt} is estimated. Appendix B describes how the values were measured.

The charts confirm that the choices which have a large minimum number of messages have a higher cost with large number of processors. As is expected, the two-step redistribution is more expensive with few processors. The crossover point where two-step redistribution becomes more efficient changes with the model size and the redistribution method. Stream-per-region redistribution benefits more from two-step redistribution because the messages are smaller. With larger models, two-step redistribution is not as advantageous because the one-step messages may already be large enough so that the larger messages with two-step redistribution may not give any benefit.

4.4.6 A Better Method for Choosing the Number of Groups

The equations in the previous sections give a first approximation to the number of groups, and illustrate the costs of the different redistribution methods. The equations for the number of groups minimize the overall number of messages. However, they assume that all the groups are the same size, which cannot be true in all cases. When the number of groups is not a divisor of the number of processors, some groups will be larger than others. The difference in group sizes can be large due to the constraint that the group boundaries cut the grid of processors from top to bottom and left to right. For example, with 128 processors and 12 groups, eight groups will have 12 processors and four groups will have 8 processors. The different group sizes can cause a load imbalance in three ways:

1. A group smaller than others will need to receive more first-step messages per processor than normal since the N incoming messages will be sent to fewer processors.

2. A group larger than others will need to exchange more second-step messages between processors in the group.

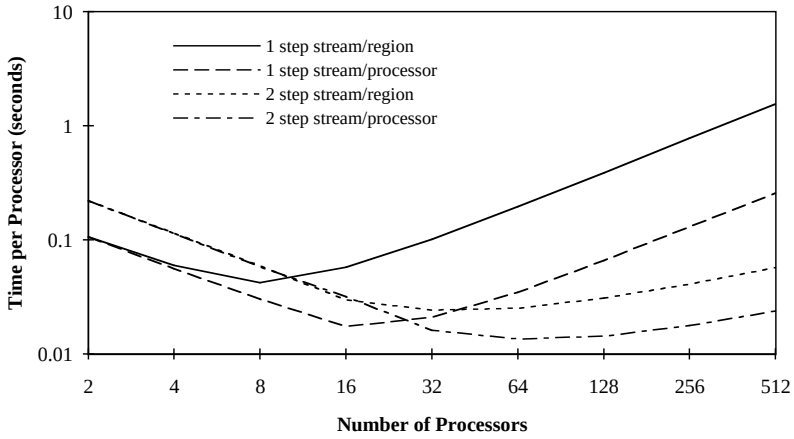


Figure 4.4 Comparison of communication costs as a function of number of processors for a 10,000 triangle scene for the four redistribution methods.

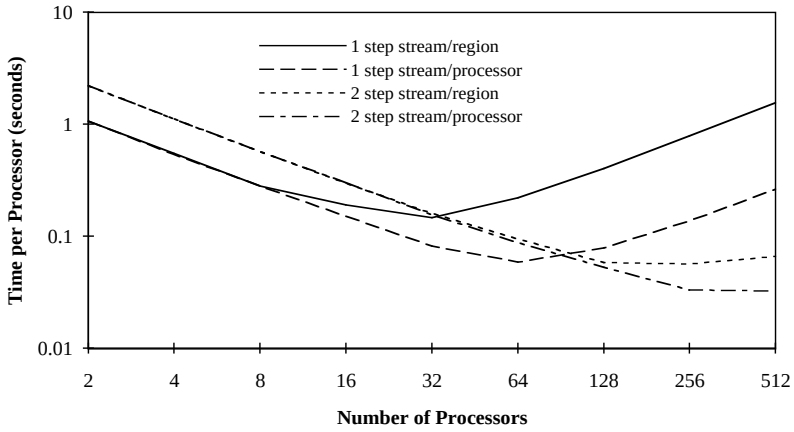


Figure 4.5 Comparison of communication costs as a function of number of processors for a 100,000 triangle scene for the four redistribution methods.

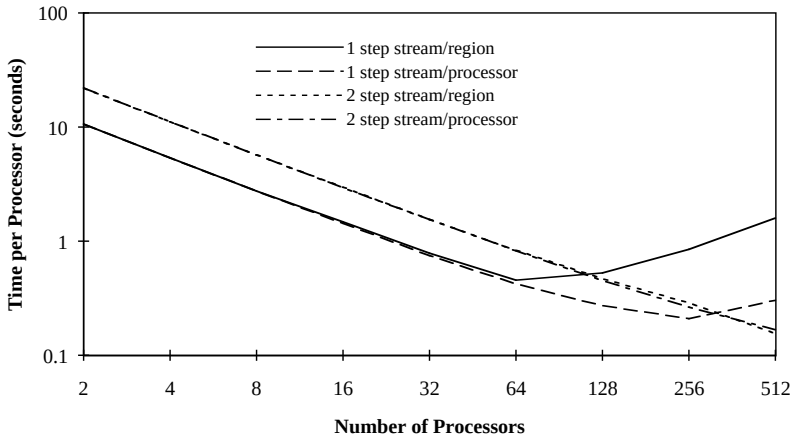


Figure 4.6 Comparison of communication costs as a function of number of processors for a 1,000,000 triangle scene for the four redistribution methods.

3. Some processors inside the group will get more messages than others. If the groups are not the same size, the number of processors outside a group may not be divided evenly by the number of processors within the group. The consequence is that some processors will receive a larger-than-average number of messages and triangles from processors outside the group.

The above load imbalances can be reduced by choosing the optimal group size using a more precise performance model of the redistribution process. The model will be used to pick the number of groups by evaluating the model for all the possible numbers of groups, and picking the number of groups that gives the best performance. To simplify the argument, the model will only be developed for stream-per-processor redistribution, since two-step redistribution is never used with stream-per-region redistribution. Two-step redistribution is only used with many processors, while the stream-per-region redistribution is only necessary when using a few processors. The performance model determines the number of messages and triangles sent and received by a processor in each step, and calculates the time required for that amount of communication.

Calculating the amount of communication required for a given number of groups is somewhat involved. There can be many different ways to cut a rectangular array of processors that result in the correct number of groups. For example, a 4 by 8 array can be cut into 6 groups in three different ways: (1) by making five cuts horizontally, (2) by making two cuts horizontally and one vertically, and (3) by making one cut horizontally and two vertically. Each method of partitioning the processors into groups will have a set of group sizes, and each distinct group size can have different communication times.

An algorithm can calculate the communication time for a given number of groups g by looping over the possible processor array partitionings. In each iteration, the algorithm computes a pair (g_v, g_h) giving the size of the array of groups horizontally and vertically ($g = g_v g_h$). Then, it iterates over the set of group sizes for the current (g_v, g_h), computing the time required for each group size N_g using the performance model function $c(N_g)$. The communication time for a given processor array partitioning is the maximum time taken by any group. However, the communication time for a given number of groups is the minimum time taken for any of the possible partitionings since one can choose which partitioning to use. Put into a formula, the communication time $C(g)$ required for g groups is

$$C(g) = \min_{\text{all } (g_v, g_h) \text{ for } g} \left(\max_{\text{all } N_g \text{ for } (g_v, g_h)} c(N_g) \right). \tag{4.1}$$

4.4.6.1 Detailed Communication Performance Model

To evaluate the above equation, we first need to derive $c(N_g)$, the performance model function. We start the derivation of the performance model with the first step of the redistribution. Here, the nO/N triangles redistributed per processor are divided into g streams of triangles being sent between pairs of processors. By dividing the number of triangles per stream by n_{msg} and taking the ceiling we get the number of messages per stream. Each processor sends that many messages to each of the $g - 1$ other groups, so the number of messages sent per processor in the first step is

$$m_{1s} = (g-1)\left\lceil \frac{nO}{n_{msg} Ng} \right\rceil. \tag{4.2}$$

Note that $^1/g$ of the triangles are not sent by each processor, since they are bound for other processors within the same group. The number of triangles sent in the first step by each processor is similar to the equation for the number of messages. The division by the message size, n_{msg}, is removed, as well as the ceiling function. The ceiling function is not needed since the time to send a triangle is small. The equation is

$$n_{1s} = (g-1)\left\lceil \frac{nO}{Ng} \right\rceil \cong (g-1)\frac{nO}{Ng}. \tag{4.3}$$

The number of messages received in the first step is similar. In a group with N_g processors, the group as a whole receives $N - N_g$ streams of messages. By dividing by the group size and taking the ceiling, and then

multiplying by the number of messages per stream calculated above, we get the worst-case number of messages received by a processor in the first step

$$m_{1r} = \left\lceil \frac{N - N_g}{N_g} \right\rceil \left\lceil \frac{nO}{n_{msg} Ng} \right\rceil. \tag{4.4}$$

The worst-case number of triangles received is

$$n_{1r} = \left\lceil \frac{N - N_g}{N_g} \right\rceil \frac{nO}{Ng}. \tag{4.5}$$

This is the number of triangles per stream multiplied by the worst-case number of streams.

To compute the number of messages sent per processor in the second step, we first compute the worst-case number of triangles that a processor could have to reformat. This is the sum of the worst-case number of incoming triangles and the nO/Ng triangles that were not sent in the first step. To get the number of messages per stream, we divide the number of triangles by the product of the number of streams of messages sent out in the second step, N_g, and the maximum message size, and then take the ceiling. Then, we multiply by the number of streams sent in the second step, $N_g - 1$, which is the number of other processors in the group, giving

$$m_{2s} = (N_g - 1) \left\lceil \frac{1}{n_{msg} N_g} \left(\left\lceil \frac{N - N_g}{N_g} \right\rceil \frac{nO}{Ng} + \frac{nO}{Ng} \right) \right\rceil,$$

or after simplification

$$m_{2s} = (N_g - 1) \left\lceil \frac{nO}{n_{msg} Ng N_g} \left(\left\lceil \frac{N - N_g}{N_g} \right\rceil + 1 \right) \right\rceil. \tag{4.6}$$

Again, the equation for the number of triangles is similar:

$$n_{2s} = \frac{nO(N_g - 1)}{Ng N_g} \left(\left\lceil \frac{N - N_g}{N_g} \right\rceil + 1 \right). \tag{4.7}$$

The number of messages received in the second step is

$$m_{2r} = (N_g - 1) \left\lceil \frac{nO}{n_{msg} N N_g} \right\rceil, \tag{4.8}$$

which is the average second-step stream size multiplied by the number of other processors in the group. For simplicity, this ignores that some processors send more triangles than others in the group, which means that those could also have to send more messages than the minimum. The number of triangles received in the second step is

$$n_{2r} = \frac{nO(N_g - 1)}{N N_g}. \tag{4.9}$$

This formula does not have the approximation used in equation 4.8. Finally, the number of triangles to be reformatted between messages for the first and second steps is

$$n_{refmt} = \frac{nO}{Ng} \left(\left\lceil \frac{N - N_g}{N_g} \right\rceil + 1 \right). \tag{4.10}$$

The formula is similar to the ones for m_{2s} and n_{2s}. It is the worst-case number of incoming triangles from the first step plus the triangles that were not sent in the first step.

Given the equations 4.2-4.10, the actual communication cost (in seconds) for processors within a group of size N_g is

$$\begin{aligned} c(N_g) = {} & (m_{1s} + m_{2s})t_{msg_s} + (m_{1r} + m_{2r})t_{msg_r} + (n_{1s} + n_{2s})s_{tri}t_{byte_s} \\ & + (n_{1r} + n_{2r})s_{tri}t_{byte_r} + n_{refmt}t_{refmt}, \end{aligned} \tag{4.11}$$

where t_{msg_s} and t_{msg_r} are the per-message costs of sending and receiving messages, respectively, and t_{byte_s} and t_{byte_r} are the respective per-byte costs. The time to reformat a triangle is t_{refmt}.

To find the optimum group size, an algorithm evaluates the performance model that calculates the time for a given group size (equation 4.1) for all possible group sizes, and picks the number of groups that gives the minimum time.

4.4.6.2 Improvements due to the Better Method

Figure 4.7 shows the model evaluated for different model sizes and different numbers of processors. The timing values and data sizes are from measurements of an implementation running on the Intel Touchstone Delta; Appendix B gives the values and describes how they were obtained.

In the figure, the cases with fewer than 64 processors are not shown, since one-step redistribution should be used. With few triangles and more than 32 processors, two-step redistribution is the optimum. The number of groups is $\sqrt{N}$, if that evenly divides the number of processors. Otherwise, a value near $\sqrt{N}$ is used that produces a smaller spread of group sizes. A smaller spread of group sizes will reduce the load imbalances that are caused both by small groups having to accept more second-step messages than the average, and by large groups having to exchange an above-average number of messages during the second step. With more triangles, the optimum number of groups changes to one that better divides the number of processors, i.e., one that better equalizes the group sizes. Having more triangles makes it less important to optimize the number of messages; instead, the overall communication loads are optimized. With even more triangles, one-step redistribution is better.

Figure 4.8 compares the estimated communication time with the optimized group sizes as compared to using $\lfloor \sqrt{N} \rfloor$. The optimized group sizes save up to 25% of the communication time. With 128-512 processors, the usual saving is between 5 to 15%. One caveat: the figures are from the performance model, which assumes that every processor generates the same amount of data for each other processor. This assumption is not true, and I expect that the actual savings will be somewhat less than predicted.

The optimum number of groups often changes by a surprising large amount in response to small changes in the model size, which results in the optimum number of groups jumping around in figure 4.7. This can be explained by looking at the simple formula derived earlier for the total number of messages m sent by all the processors,

$$m = N(g-1) + N(N/g-1),$$

where one can see that the total number of messages is the same whether the number of groups is i or is N/i. The message counts for the two cases are not the same with the more complex formula, but the counts are similar. If the optimum number of groups i is relatively far away from $\sqrt{N}$, a second local minimum exists at N/i. Small performance changes resulting from slightly different model sizes can cause the optimum number of groups to switch between the two minima. For example, with 256 processors the optimum number of groups changes from 24 to 10 when the model size is increased from 350,000 to 355,000 triangles, but the difference in speed is less than one percent.

4.4.7 Message Size

The last redistribution design choice is the maximum message size, the size of the message buffers for sending triangles. It is not as important as the redistribution choices discussed earlier as long as a message size is picked that amortizes the per-message cost over a few dozen triangles. Once that threshold is crossed, the processing time is somewhat insensitive to the size. Experiments described in section 7.3.3. show that the total frame time changes by at most 18% when the message buffer is varied by a factor of 4.

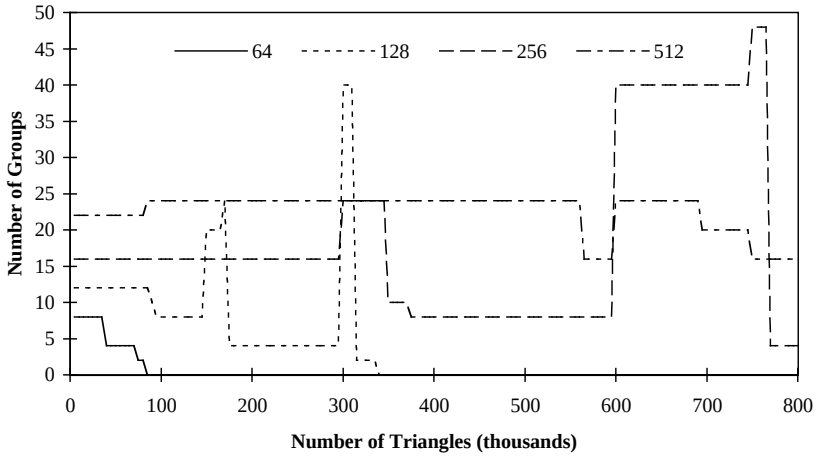


Figure 4.7 Predicted optimum group size for different model sizes and numbers of processors. Each curve corresponds to a different number of processors. Zero groups indicates that one-step redistribution is favored. Points are computed and plotted for every multiple of 5,000 triangles.

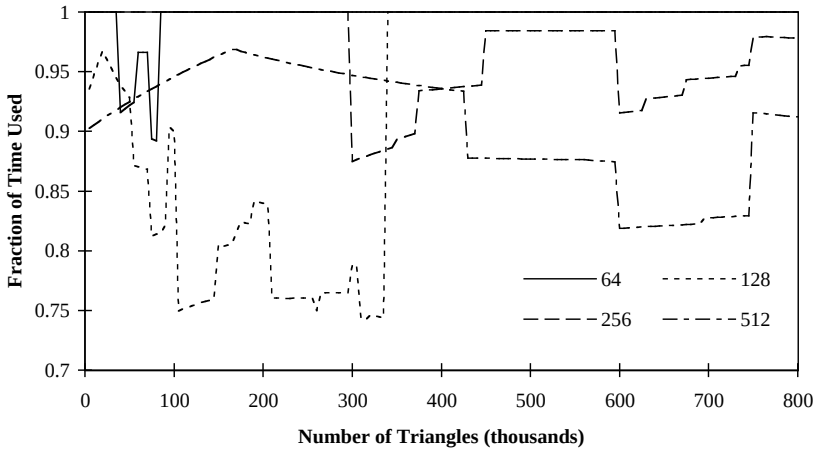


Figure 4.8 Fraction of time required for redistribution using the number of groups calculated using the performance model compared to the time required when using $\sqrt{N}$ groups. Points are computed and plotted for every multiple of 5,000 triangles.

The message size choice involves a tradeoff between the amount of message overhead and the amount of communication that is overlapped with other processing. With large messages, the per-message overhead is amortized over more triangles. However, it will also take longer until message buffers are filled, increasing the time during a frame when the network is not used. Furthermore, the amount of buffer flushing at the end of the geometry processing is larger; it could be the entire frame's triangles. When the geometry processing loads are balanced, all the processors flush their buffers at the same time and overload the network. If the network is too congested, some processors may have to wait for triangles to arrive instead of rasterizing them.

In large systems, the best buffer size is smaller than in small systems. First, the number of triangles transferred between processors is larger in small systems, so larger buffers would not significantly change the amount of data flushed at the end of geometry processing. Second, the fraction of the bisection bandwidth available to a processor is larger in small systems. This means that a small system can handle a larger amount of simultaneous buffer flushing as compared to a large system.

However, with very large systems, the average number of triangles sent between pairs of processors can be quite small, so small that the per-message cost is a substantial fraction of the overall redistribution cost. If sufficient communication bandwidth is available, the best buffer size would be large enough to allow the

triangles to be sent in a single message for most processors. With some systems it might be better to use large messages even though there is not enough bandwidth. The large messages would be better if the savings from using fewer messages outweigh the time spent waiting for messages to arrive.

The best message size could be determined by modeling the effects of the message size. Crockett and Orloff [Croc93] did this for their implementation running on an Intel iPSC/860 system, which uses a hypercube network. Their model assumes that the message transfer delay is the same for each processor. While this is true for hypercubes and torus networks, it is not true for mesh networks. These networks are more congested in the center, which means that messages sent to processors in the center have a higher probability of being delayed as compared to processors on the periphery of the network.

A second method to determine the message size would be to simulate the algorithm. This would require a detailed description of the network and the software protocols. I am unable to get this information for the Intel Delta, as this information is proprietary. Even if the characteristics were available, the simulation would take days or weeks with large models, and would have to be repeated for a range of system sizes. Also, given the small expected time savings, even if the network's characteristics were available I believe that the large simulation cost would not be worth the expected gain.

A third method for determining the best message size remains: performance measurement of an actual implementation. This is the method that will be used in this dissertation. The results will appear in section 7.3.3, which describes an implementation and performance results.

4.5 Recommendations and Summary

At this point, I can make recommendations for some of the design choices discussed in this chapter, and must defer the others for later chapters. The assignment and scheduling of the stages and the redistribution method can be fixed now. The choice of region assignment style must be deferred until load-balancing has been investigated, and the message size will be determined experimentally. These choices are the first three levels of the taxonomy shown in figure 4.1. The choices are:

- Assignment of stages to processors: all processors perform the transformation and rasterization stages. This maximizes the overlap of the calculation and communication.
- Stage scheduling: the geometry processing and rasterization stages should be run consecutively. This increases the overlap of communication and calculation, and is compatible with one desirable region assignment style, between-frames region assignment.
- Redistribution method: The redistribution method should be a hybrid between the different options discussed, since several methods are feasible under different conditions. The optimum method should be chosen at runtime by evaluating a performance model and choosing the one that will give the best performance. With most systems, stream-per-processor redistribution should be used, with one-step or two-step redistribution chosen based on the performance prediction. But, if the amount of frame buffer memory needed for stream-per-processor redistribution is not available, one-step stream-per-region should be used.

While supporting all three redistribution methods adds complexity, it is a reasonable approach. The geometry processing inner loop can support the three methods by using an array to map regions into message buffers. Different initializations of the mapping array will change the inner loop to support the different methods.

Figure 4.9 sketches the basic rendering algorithm according to the recommendations listed above. It does not include region assignments, load-balancing, or the details of the redistribution method; they will be added in the coming chapters. The triangle copying performed on line 7 is done using an array to map region numbers to buffers. The array is different depending on the redistribution method being used. If one-step stream-per-region redistribution is used, the array is the identity map, since one buffer is used for each region. With one-step stream-per-processor redistribution, the array holds the region-to-processor assignment, so that all regions for a processor are mapped to the same buffer. Finally, with two-step stream-per-processor redistribution, the array holds the composition of the region-to-processor and processor-to-group mappings. This maps the region to the processor that will render it, and also maps the processor to the group that the processor has been assigned. In the three cases, the number of buffers allocated on line 3 is respectively r, N, and g. More redistribution details appear in Chapter 7.

```
synchronize /* ensure previous frame is finished */

allocate output buffers (one per region, processor, or group)
for each raw triangle {
      transform to screen space, clip, light vertices
      calculate the region(s) it overlaps
      copy display triangle into corresponding buffer(s)
      send buffer if it is full
}
flush output buffers

if (stream per region) {
      allocate one region-sized frame buffer (z and color)
      for (each region) {
            rasterize the region’s display triangles
            send to frame buffer and/or disk
      }
}
else {
      allocate frame buffer for each of my regions
      rasterize triangles into appropriate frame buffer
      send region-sized frame buffers to frame buffer
            and/or disk
}
```

Figure 4.9 Pseudocode for the basic algorithm.

The choices described in this chapter are for the Intel Touchstone Delta; other systems could force different choices. For example, the Pixel-Planes 5 system uses one-step stream-per-region redistribution instead of varying the choice at runtime. The choice is forced because the specialized rasterization processors only have enough memory to render one region at a time. Even though this is the most expensive redistribution option, on Pixel-Planes 5 it is still reasonably efficient because the system has a much lower per-message communication cost than the Delta, and because the system can only be scaled to about 85 processors. A comparison of the two machines’ different redistribution characteristics appears in Chapter 8. However, before discussing systems with specialized hardware, we need to fill in the details of the above algorithm, starting with the discussion of load-balancing.

CHAPTER FIVE

LOAD-BALANCING GEOMETRY PROCESSING

This chapter starts the discussion of load-balancing methods for the geometry processing task, that is, ways to evenly partition the geometry processing task among the processors. One could argue that the chosen application space of visualizing scientific calculations makes the discussion very simple: since most of those applications use an immediate mode interface, the partitioning follows the partitioning of the user's data. As mentioned in section 3.1.2, using partitions other than the user's appears to be too expensive to be practical. However, some visualization applications will use retained models, so discussing how to partition them is pertinent. The discussion should also be useful for other applications that use retained models. The chapter will focus on how to distribute hierarchical display lists since they are commonly used in graphics packages.

The chapter extends the earlier discussion of geometry processing load-balancing (in section 3.1.2) by removing the restriction of only having flat models and a single type of primitive. Much of the work presented here appeared previously [Ells90a]. However, this chapter adds some improvements and characterizes the performance for systems with many more processors.

The chapter starts with a short description of hierarchical display lists, followed by a discussion of two distribution strategies. One strategy distributes structures among the processors, and the second distributes primitives within each structure among the processors. The second strategy is chosen because it does not require the primitives to be redistributed among the processors as the viewpoint changes. However, the strategy duplicates much work, so section 5.2 describes optimizations that reduce this work. Section 5.3 adds another optimization, randomizing the distribution, that avoids some pathological cases. The last sections evaluate the different algorithms and summarize the results.

5.1 Algorithms for Distributing Hierarchical Display Lists

Retained display lists usually support hierarchy, as seen in PHIGS [ANSI88]. The basic unit of organization in hierarchical display lists is the structure, which is a grouping of structure elements. Structure elements consist of:

- primitives, such as lines and polygons.
- attribute elements, which hold changes in attributes such as modeling transformations, color changes, and line widths.
- execute elements, which cause an instance of another structure.

When the display list is traversed, an *attribute state* is used to hold the current values of all the attributes. When a primitive is encountered, the current modeling transformation is used to compute the primitive's screen space location. Other attributes determine the primitive's color and other features. When an attribute element is encountered, the attribute state is updated with the new attribute value. Modeling transformations can either be concatenated with the current transformation or replace it.

Execute elements produce an instance of the named structure; they are analogous to procedure calls in programs. The called structure inherits the current attribute state. For example, this allows a structure to be instanced in two different locations by calling the structure twice, each time with a different modeling transformation in effect. When a structure is called, the attribute state is saved beforehand and then restored after the call returns. This means that an instanced structure cannot affect its caller's traversal state.

Display list traversal starts with the posted, or root, structures, which are those marked to be called by the graphics library. For more details about hierarchical display lists, see Chapter 7 of [Fole90].

One complicating requirement of display list distribution algorithms is that they must preserve the semantics of the changing attribute state. A primitive's color could be determined by an attribute element located much earlier in the same structure, or in a different structure. The semantics must also be kept correct when the structure is edited. Inserting, deleting, or modifying a modeling transformation can potentially change the screen position of thousands of primitives in many different structures. The next two sections describe two strategies for distributing a display list among different processors, including how the attribute state semantics are preserved.

5.1.1 Distribute-by-Structure

The first distribution strategy is to place entire structures on different processors so that the load is balanced. Some structures may need to be divided if there are too few structures to balance the loads evenly. Structures instanced more than once could be replicated on different processors.

A processor must have the inherited attribute state for a structure before traversing it. Each processor can compute its structures' attribute states if a partial skeleton structure hierarchy is maintained on each processor. The skeleton hierarchy only exists for the portion of the hierarchy above each local structure, and only contains attribute and execute elements. During traversal, the skeleton hierarchy is traversed to compute the inherited attributes for each local structure. Attributes in the skeleton structures will be traversed by multiple processors, lowering the parallel efficiency. Such attributes will be replicated on different processors, which will increase the cost of changing them.

A second method to compute each structure's inherited state would maintain a skeleton of nearly the entire structure hierarchy on a single processor. This processor would traverse the skeleton hierarchy during image generation. During traversal, when a processor finds an execute structure it would send the current attribute state to the processor assigned the corresponding instance, and then continue the traversal. Compared to the first method, this method reduces the amount of processing done on more than one processor. However, having a single processor traverse the skeleton hierarchy introduces a possible serial bottleneck. Also, if a processor only has structures that are encountered late in the traversal, it may have to wait for nearly the entire frame, also causing a bottleneck. Having the skeleton traversal be done one frame ahead of the remainder of the geometry processing removes the last bottleneck but adds latency.

Since distribute-by-structure has not been thoroughly investigated, it is likely that other methods for computing the inherited attribute states exist that have smaller amounts of repeated work, no traversal bottlenecks, or both. But, the distribute-by-structure strategy has another disadvantage: because objects are often placed in a structure, when an object goes off screen the processor workloads will become unbalanced. The loads can also become unbalanced as the display list is edited, especially when entire structures are added or removed from the active hierarchy. The loads can be rebalanced by redistributing the structures. However, this would require additional processing time. The amount of processing would be significant, since it is a global operation, and because the method for computing each structure's traversal state must also be accomodated.

5.1.2 Distribute-by-Primitive

The second strategy for distributing display lists distributes each structure independently. The distribution is done primitive by primitive such that successive primitives are placed on different processors. Attribute elements are given to all the processors so that the attribute state can be updated. Execute elements are also given to every processor so that each can follow the entire hierarchy. One way to look at this method is that each processor has the entire structure hierarchy but with each structure smaller than the original. Figure 5.1 shows an example.

An implementation of this strategy would keep track of each processor's load for each structure, that is, an estimate of how much time each processor should take to process the structure. Each primitive is given to the processor that currently has the smallest load. When an element is given to a processor, the load value associated with that processor is updated by adding an estimate of the time to process that element. There is no overall processor load since each structure is balanced independently. Figure 5.2 shows a distribution example with a more complex structure.

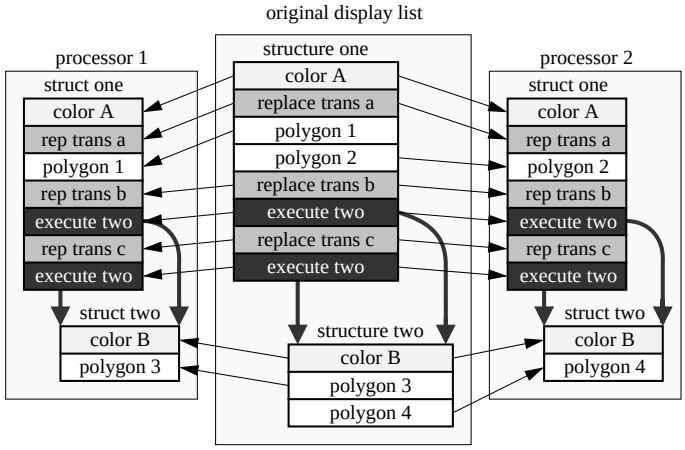


Figure 5.1 Simple distribute-by-primitive example, where two structures are distributed across two processors. The thick lines indicate hierarchy links, and the thin lines show how the structure elements are distributed.

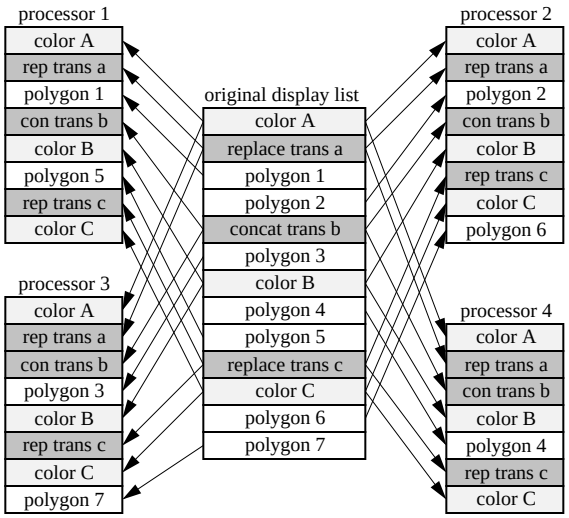


Figure 5.2 Simple method for distributing a single structure across four processors.

Obviously, this strategy can have considerable overhead. Every processor must traverse the entire structure hierarchy and process every attribute change. The advantage of this method is that editing the display list is simple and fast. The only requirement is that the memory location of every copy of each structure element must be recorded in a data structure. For primitives, the data structure records the processor that has the primitive, and the location of the primitive within the processor's memory. For attributes, the data structure records the location in each processor's memory where the attribute copies are stored. Per-processor loads for each structure are also saved so they can be consulted and updated when structure elements are added to the structure. The loads are also updated when structure elements are deleted.

Because each structure is independently load-balanced, the loads are balanced independently of the number of times the structure is instanced. Also, the loads should be balanced when part of the model is outside the viewing frustum. Primitives outside the frustum are generally concentrated in a few parts of the display list since successive primitives in a structure are often near each other in modeling space. Those successive primitives have generally been sent to different processors, and thus the easier-to-process primitives, those outside the viewing frustum, are distributed across all the processors.

However, this strategy does not work well with very small structures. The structure cannot be evenly divided if there is not at least one primitive for each processor. Also, the per-structure overhead is significant when processing small structures.

5.1.3 Summary of Strategies

The first strategy appears to require considerable duplicated processing and needs active load-balancing so that the loads remain balanced during an application. The second strategy requires more duplicated processing, but should work with static load-balancing. The second strategy is superior, especially since the amount of duplicated processing can be reduced by optimizing it.

5.2 Optimizing the Distribute-by-Primitive Algorithm

This section describes three optimizations of the distribute-by-primitive display list distribution algorithm. While the optimizations are independent, they all increase the complexity and cost of editing the display list. In practice, the algorithm with the smallest amount of duplicated work makes supporting editing too difficult, so it is only used with static models. The Delta implementation described in Chapter 7 uses the first two optimizations, while the Pixel-Planes 5's PPHIGS graphics library [Fuch89, Ells90b] (described in Chapter 8) only uses the third optimization since it allows arbitrary editing.

5.2.1 Removing Superfluous Attribute Changes

The first optimization reduces the overhead by not giving processors superfluous attribute changes. This is done by distributing a structure's elements in traversal order. When an attribute is encountered, it is not sent to any processor. Instead, an attribute state used during the distribution is updated with the new attribute. Also, additional state is modified to indicate that no processor has the new attribute value. When a primitive is distributed to a processor, the algorithm checks the attribute state to see whether the processor that will receive the new primitive has the associated attributes. If the processor has not yet been given any associated attribute, it is given those values as well as the actual primitive.

When an execute structure element is encountered, it is sent to all the processors. This must be done because all processors will traverse the entire structure hierarchy during the geometry processing. Before distributing the execute element, the attribute state must be updated on all processors. The entire state must be updated since the executed structure can potentially contain any primitive type.

Modeling transformations that are concatenated with the current transformations require slightly different processing. A concatenation transformation will affect all following primitives in the structure until a replace transformation or the end of structure is encountered. This could be dealt with by giving all processors a copy of the transformation, as described in [Ells90a].

Removing the superfluous attribute changes makes editing the display list more difficult. Overwriting a structure element with the same type of element is straightforward, since the old one is replaced. Deleting a primitive can make one or more attribute elements superfluous, which should be deleted. Deleting an attribute can increase the number of primitives affected by an earlier attribute, possibly requiring that earlier attribute to be sent to more processors. Inserting a primitive can require that the attributes be sent to the processor where the new primitive is sent. Finally, inserting an attribute may make other attributes on various processors superfluous. Inserting or deleting will usually require that the display list be analyzed. This can be quite costly if no single processor has the entire display list, as stepping through the list will require communication at each step. More details of the editing operations appear in the earlier paper [Ells90a].

The amount of analysis is bounded, although it can be large. At most cN primitives before and after the modification will have to be traversed, where c is the largest number of primitives that can be distributed before all processors have received a copy of all attribute changes. The value is the sum of the cost of the most expensive primitive and the cost of updating all the associated attributes, together divided by the cost of the least expensive primitive. This bound results from the policy of always sending the next primitive to the most lightly loaded processor. This policy may be violated when primitives are inserted or deleted, increasing the amount of traversal required.

5.2.2 Performing the Transformation Concatenation During Distribution

A further optimization performs the transformation concatenation during the distribution, turning the transformation into a replace transformation. This optimization reduces the amount of work during traversal since the new transformation may not have to be sent to all the processors. However, it also makes editing extremely difficult since changing one transformation may require modifying many successive transformations.

```
distribute_struct()
{
     curr_trans = identity matrix
     curr_color = default_color
     have_trans[1..N] = have_color[1..N] = TRUE
     loads[1..N] = 0

     for (each element in structure) {
          switch (element.type) {
          case POLYGON:
               i = proc_with_lightest_load(loads)
               update_state(i)
               send polygon to processor i
               loads[i] += POLYGON_LOAD
               break
          case COLOR:
               curr_color = new_color
               have_color[1..N] = FALSE
               break
          case REPLACE_TRANS:
               curr_trans = new_trans
               have_trans[1..N] = FALSE
               break
          case CONCAT_TRANS:
               curr_trans = curr_trans * new_trans
               have_trans[1..N] = FALSE
               break
          case EXECUTE_STRUCT:
               for (i = 1 to N) {
                    update_state(i)
                    send execute_struct to processor i
               }
          }
     }
}
update_state(i)
{
     if ( ! have_trans[i]) {
          send curr_trans to processor i
          have_trans[i] = TRUE
          loads[i] += TRANS_LOAD
     }
     if ( ! have_color[i]) {
          send curr_color to processor i
          have_color[i] = TRUE
          loads[i] += COLOR_LOAD
     }
}
```

Figure 5.3 Pseudocode for the optimized distribution algorithm.

Pseudocode for an algorithm that uses the first two optimizations, removing superfluous attributes and performing concatenation during distribution, is shown in figure 5.3. Figure 5.4 gives an example of the algorithm working on the same structure as the one used in figure 5.2.

5.2.3 Primitive Structures

A third optimization reduces the overhead of small structures and divides the workload more evenly. If a structure does not call any other structures and is instanced many times, that structure can be replicated on

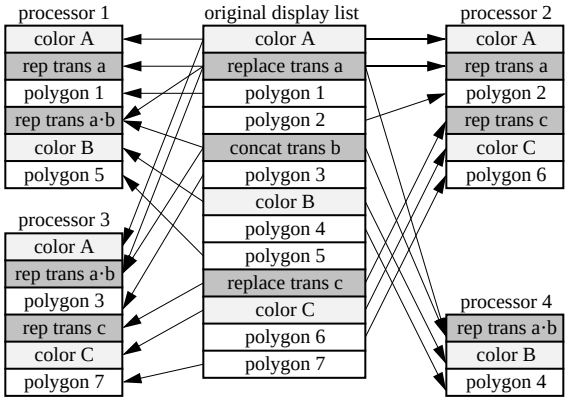


Figure 5.4 Distributing a single structure across four processors with superfluous attributes omitted and concatenation transformations folded into replace transformations.

all processors, and instances distributed among the different processors. Each execute element is distributed as if it were a primitive with a load equal to the time to process the entire structure. However, this method only works if the structure is static, and is created before the instances are created. Adding or deleting elements in the structure will unbalance the loads since the distribution will not be changed. The structure must be created in advance so that the processing time can be calculated before distributing the execute elements. This optimization is implemented on Pixel-Planes 5. The user must indicate which structures should be treated specially (as *primitive structures*) so that the system can do the necessary error checking.

5.3 Randomization to Further Remove Coherence

The algorithm presented above often keeps the loads balanced as parts of the model go off screen, as will be shown in the next section. However, assigning successive primitives to successive processors does not always remove correlations of the primitives' locations and orientations. The correlations occur most often with algorithmically generated models, as they are very regular. For example, if triangle mesh has a dimension that is a multiple of the number of processors, triangles will be assigned to processors in a regular pattern.

To combat this, I add randomization to the algorithm. I do this by quantizing the loads on each processor by dividing each load value by a small constant, and then using the integral value of the result to classify the processors. This scheme places processors that have nearly the same loads in the same group. Then, the distribution algorithm sends primitives to a randomly chosen processor from the most lightly loaded group. Adding randomization does not change the overall structure of the algorithm shown in figure 5.3; only the `proc_with_lightest_load` function is modified. To keep the loads balanced, the quantization amount should be fairly small, such as the cost of a triangle.

The function to select the processor for the next primitive can be efficiently implemented using a modified heap, where each node can contain multiple entries. Each node contains all the processors with loads quantized to a single value. The maximum number of nodes is often fairly small: it is the load of the primitive with the largest processing time divided by the quantization value.

5.4 Evaluating the Algorithms

I used a simulator to evaluate the two optimized load-balancing methods, with and without randomization, and to get an estimate of the processor utilization during geometry processing. Using a simulator is easier than instrumenting an actual implementation because it is difficult to measure the time required for communication. Measuring the communication time is difficult because much of it is performed in response to interrupts. The simulator should be fairly accurate because the geometry processing is deterministic, and because the associated performance model has a timing value for each of the major paths through the code. The next sections describe the simulator, the performance model used in the simulator, and its results.

5.4.1 Simulator Description

The simulator works on a static model for a series of frames. I used static models to simplify the implementation. Using a series of frames allows a variety of configurations to be tried, with the model filling a small part of the screen or the entire screen, or with only a portion of the model on screen. The models used in the simulation and a sample of the frames in each model's viewing sequence are shown in Appendix A.

At startup, the simulator reads models in an ASCII version of the PPHIGS (Pixel-Planes 5 Hierarchical Interactive Graphics System) [Ells90b] archive format. Only a subset of the format is supported: only simple polygons, colors, and transformations. The simulator reads in a viewing frustum (camera transformation matrix, field of view, and hither plane) for each frame.

The simulator uses the chosen algorithm to partition the model across the processors. Then, it transforms each partition for each of the recorded frames, calculating the time taken by each processor using a performance model. The overall processor utilization is then calculated by dividing the average of the processor times by the maximum.

The simulator does not update the processor loads when color changes or transformations are sent to a processor. However, this algorithm distributes the sample models the same way as the actual distribution algorithm because the sample models are extremely regular. Either color changes affect so many triangles that they are sent to all processors, or, if there are more than a few processors, they affect so few primitives that every distributed triangle has a color change before it. The models contain at most one transformation, which must be sent to all processors. So, attributes are either already balanced because they are on all processors, or because they are balanced as triangles are balanced.

The amount of repeated work due to changing attributes is not measured by the simulator. The worst case is having to change the color and transformation once for every primitive, which would have quite a high amount of repeated work. Of the four models, the Terrain model has the most repeated work since there are only 2.3 primitives for each color change. Each distributed primitive is preceded by a color change so each color change is executed on average 2.3 times. Note that if the unoptimized distribution method were used with the terrain model running with 512 processors, each color change would be executed an average of 512/2.3 or 225 times! This would dominate the geometry processing cost.

5.4.2 Performance Model Used in the Simulator

The simulator's performance model is fairly simple. It uses a single value for the three cases of on-screen triangles, back-facing triangles, and off-screen triangles. The model is based on processing times of an implementation on the Intel Touchstone Delta. Descriptions of the measurements as well as values for processing times not mentioned in the text below appear in Appendix B.

The processing time for on-screen triangles is 20.2 µs, which is given by the equation $t_{geom} + O\left(t_{bucket} + t_{msg_s} s_{prim} + t_{msg_s} \Big/ \left\lceil s_{prim} / s_{msg} \right\rceil\right)$. This is the geometry processing time plus the overlap factor times the work that must be done for each region that the triangle overlaps: the bucketization time plus the time to send a triangle. The send time has two components: the per-byte communication time multiplied by the size of a triangle in bytes, and the amortized per-message time, which is the per-message time divided by the number of triangles per message (the message size divided by the triangle size). The overlap factor is computed for each triangle using 6 regions per processor, with each region being the same size and approximately square. These are similar to the region parameters used later in the dissertation. Including the overlap factor in the simulations decreases the processor utilization noticeably compared to not modeling it, especially for the cases without back-face culling. Finally, the last two cases are much simpler: the off-screen value t_{geom_off} is 6.6 µs, and the back-face value t_{geom_back} is 6.2 µs. Triangles that are both off-screen and back-facing use the back-facing value since back-face culling is performed first.

The performance model does not model the case where the geometry processing and rasterization of a given triangle are done on the same processor, where the triangle does not need to be redistributed. On average, this happens for $1/N$ of the triangles. This case is difficult to model since the load-balancing method for the rasterization would also have to be simulated. However, that fraction of undistributed primitives is only significant when using a few processors. The decrease in processor utilization due to non-redistributed triangles should be smaller than the effect of enabling back-face culling for two reasons: (1) the fraction of non-redistributed triangles is smaller than the fraction of culled triangles when more than 2 processors are

used, and (2) the difference in processing time between distributed and non-redistributed triangles is less that the difference between front-facing and back-facing triangles. As seen below, enabling back-face culling with only a few processors only makes an appreciable change when not using randomization with the Polio data set.

5.4.3 Simulation Results

The results from the simulation appear in figures 5.5 and 5.6. The first figure shows the utilization with back-face culling disabled while the second figure shows the utilization with culling enabled. Only the Terrain model usually has back-face culling disabled; the other three models usually have it enabled. Both are shown since the use of back-face culling changes the results.

When back-face culling is disabled, all four models give reasonably good performance. The Terrain and Polio perform much better with the randomized assignment. Both models have coherence that reduces the processor utilization. The Polio model has 16 triangles in each sphere. Without randomization and at least 16 processors, each processor has a set of triangles that face the same direction since the tessellation factor divides the number of processors evenly. Triangles at the front and rear of the sphere will have a larger screen area than the triangles at the sides, increasing the average overlap factor and thus increasing those

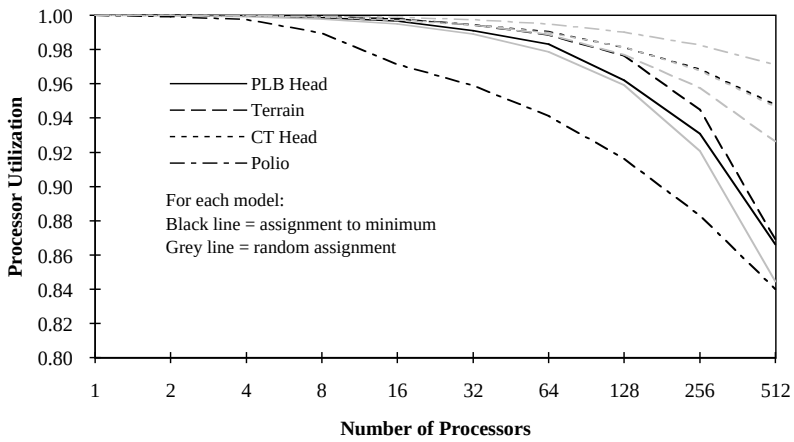


Figure 5.5 Processor utilization as predicted by simulation for the four models and two distribution methods: assignment to the processor with the minimum load and random assignment to the processor with the minimum load. These values are with back-face culling disabled. Note that the results for the CT Head model cases are nearly the same, so the corresponding curves are on top of each other.

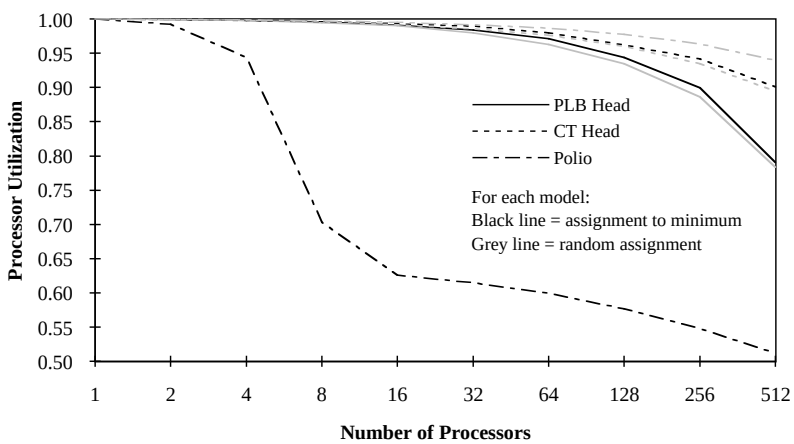


Figure 5.6 Processor utilization as predicted by simulation for three models and two distribution methods: assignment to the processor with the minimum load and random assignment to the processor with the minimum load. These values are with back-face culling enabled.

processors' load. Without randomization, the processor utilization with 512 processors is 84%; with randomization, it is 97%.

The distributed Terrain model has coherence because it has strips of 510 triangles, which means that the non-randomized version assigns diagonal strips of triangles to processors. The non-randomized algorithm performs well when the entire model is on screen, giving 97% utilization with 512 processors. The performance drops when much of the model is off-screen: at the end of the viewpoint series, 15% of the model is on screen, with a processor utilization of 59% when using 512 processors.

The PLB Head has slightly decreased processor utilization when using the randomized version, although the utilization is still reasonable. Randomization has no effect with the CT Head model.

The performance with back-face culling enabled is quite different. In general, the utilization is lower, which is to be expected. The Polio model gives especially poor utilization when using the non-randomized version, since some processors are assigned triangles that are all front-facing while others have only back-facing triangles. The randomization greatly increases the utilization with the Polio model, and only slightly decreases the utilization of the other two models.

5.5 Summary

The more complex optimized distribution methods increase the probability of having high processor utilization. The simple, unoptimized distribution method can have low utilization if there are many attribute changes or many small structures. The more complex methods have a higher utilization, but still only work well when there are either many attribute changes (like one per primitive) or very few. Cases between these extremes will not give performance near the optimum. One possible solution would be to "batch" a few primitives together, where a small number of successive primitives are given to each processor instead of switching processors whenever the most lightly loaded processor changes. Batching the primitives would decrease the amount of attribute replication required. Additionally, randomizing the assignment increases the probability that the assignment will give high utilization, but with some models it reduces the utilization slightly. The increase in robustness is worth the slight possible decrease in utilization.

However, the more complex methods greatly increase the work when the structures are edited. An earlier paper [Ells90a] describes the work necessary to allow inserting or deleting primitives in the middle of a structure; the algorithm is quite complex. The implementation described later in the dissertation does not allow editing, so it uses the complex method with randomization.

Finally, the distribution method breaks down when there are many small structures. It is very difficult to break up the structures so that each processor has the same amount of work for each structure. The use of primitive structures reduces this problem, but the strategy is not a general solution and requires user action. Instead, a distribute-by-structure algorithm will have to be used, so that at most only a few processors work on small structures. Since these algorithms have not been investigated, the methods for calculating the inherited state for each structure and for rebalancing the loads will have to be developed.

The distribute-by-primitive methods do give satisfactory results in many cases. The simple distribution method is used in Pixel-Planes 5. The amount of overhead is usually quite reasonable because the normal configuration only divides the display list among about 20 processors. The complex method can be used effectively with hundreds of processors if the models have a simple hierarchy.

CHAPTER SIX

LOAD-BALANCING RASTERIZATION

In this chapter, I complete the investigation of the sort-middle algorithm design decisions. I look into different methods for load-balancing the rasterization stage of the pipeline. A load-balancing method consists of a technique for dividing the screen into regions, and an algorithm to assign those regions to processors. While the earlier analysis in Chapter 4 showed that the work-queue assignment style was more expensive than the other styles, the analysis did not rule out any of the assignment styles. Thus, I will consider all four region assignment styles. First, I will describe load-balancing algorithms for each assignment style, and then I will identify the best algorithm for each assignment style. Then, I will compare the different styles, with each using its best load-balancing algorithm. I choose a single load-balancing method for each assignment style because the only practical way to compare the methods uses a simulator. The latter fact makes it impractical to switch between different load-balancing methods at run time.

Both the assignment style and load-balancing method are important for efficient rasterization, as poor choices can easily add 30% to the rasterization time. When using many processors, poor choices can multiply the overall frame time by four or more.

I will consider four design choices that specify the load-balancing method; they are shown as a taxonomy of algorithms in figure 6.1 (in section 6.2). The first two choices specify the load-balancing algorithm, which encompasses both how the screen is divided into regions, and how the regions are assigned to processors. The third choice is the granularity ratio, the number of regions per processor, which determines the number of regions. Larger ratios balance the loads better but also increase the overlap factor.

The last choice is the function for estimating the time to rasterize a given triangle. This choice only applies to load-balancing methods for once-per-frame assignment, since only those methods use estimates of the per-region rasterization times to make the region assignments. I will evaluate whether using a more exact estimate is worth the extra computation cost.

The first section of this chapter describes how the different load-balancing methods will be evaluated, which will be done using a combination of a simulator and a performance model. The next section introduces the different load-balancing methods. The following section explores the results from the simulation and performance model, showing results other than which methods give the best performance. Section 6.4 shows which load-balancing method gives the best performance for each assignment style, and section 6.5 evaluates the different assignment styles. The last section summarizes the design decisions.

6.1 Evaluation Methodology

The different load-balancing methods and region assignment styles will be evaluated by their performance: how long they take to render a frame. Two types of calculations are necessary to predict the overall frame time. The first type of calculations is suitable for a performance model since the calculations are relatively simple. Some parts of the overall algorithm are suitable because they are simple loops: one for each raw (untransformed) triangle, another for each display triangle, another nested loop for every scanline in each triangle, and a fourth for each pixel. A performance model can also account for the communication time by totaling the number of messages and bytes sent and received by each processor.

The second type of calculations is beyond what a performance model can reasonably calculate. The distribution of triangles on the screen determines the how well the loads are balanced, expressed as the processor utilization percentage, and also determines the number of times work must be repeated because a triangle

overlaps multiple regions, expressed as the overlap factor. The two values will be calculated by the simulator and then used by the performance model to calculate the overall frame time. (Others [Cox95] have argued that these values could be modeled analytically, so only a performance model would be needed, but such work is beyond the scope of this dissertation.)

This evaluation methodology has the potential to make it easier to select a load-balancing method for systems other than the one considered (the Touchstone Delta). The simulation results depend more on the distribution of triangles on the screen and less on a particular system's rasterization speed. If most of the new system's rasterization cost results from the per-primitive costs instead of the per-pixel, per-line, or per-region costs, the earlier simulation runs do not have to be repeated. If so, the main task would then be to write a parallel renderer on the target system, without using load-balancing, and to measure the time spent in its inner loops. The system's communication costs must also be measured. Then, the earlier simulation results and the new performance model can be used to predict the performance of the various load-balancing methods, and the best method can be selected.

This evaluation methodology cannot take into account all possible factors, since it is difficult for the simulator and performance model to capture all the details. The only way to consider all the factors would be to measure the performance of an actual implementation. The network performance is the weakest part of the overall methodology, since the latency and capacity of the network are not modeled. Those factors affect the performance when using hundreds of processors, as will be seen in Chapter 7 where the actual implementation takes more time than predicted. However, modeling those factors would be difficult, as it would require more details about the interconnection network than are available for commercial systems. Also, a much more detailed simulation would be necessary, one that would take days to weeks of computation for large models.

The simulator and performance model have some advantages over measuring an actual implementation. When measuring an implementation, it is difficult to evaluate the performance of specific parts of an algorithm, such as the rasterization load-balancing being considered here. In an implementation, the boundaries between different parts of the computation can be blurred by overlapping different operations. For example, on many systems much of the communication processing is done using interrupts. Additionally, it is easier to try different load-balancing methods on a simulator, as it does not need to be optimized. One can make a quick implementation of the algorithm but give the simulator or performance model timing values for an optimized version, and the results would then correspond to those expected for the optimized version. Using a simulator and performance model is easier than measuring an implementation if access to the target system is difficult or impossible, such as when the system has not yet been built.

The following section gives details about the simulator. Later sections derive the performance model. First the base algorithm is derived, without considering load-balancing, and then other subsections add the costs for each of the region assignment styles.

6.1.1 Simulator Description

The simulator used to evaluate the load-balancing methods shares code with the simulator used in Chapter 5 to evaluate different model partitioning methods. It uses the same models and series of frames shown in Appendix A. Using a series of frames is necessary when evaluating the rasterization load-balancing algorithms because many algorithms depend on various types of frame-to-frame coherence; the effectiveness of the coherence cannot be measured using a single frame.

Because the simulator only computes the processor utilization and overlap factor values, it only needs to execute the load-balancing algorithm and the earlier portions of the rendering pipeline. The simulation of that portion of the overall pipeline is exact: it uses the same inputs (the same triangles and viewpoints) that an implementation would use, and executes the same load-balancing algorithm. The main source of inaccuracy is that the simulator uses a slightly simplified performance model of the rasterization workload to calculate the amount of rasterization work on each processor.

During the simulation of a frame, the simulator transforms the model to screen space using the recorded viewing frustum. It then bucketizes the triangles according to the load-balancing method's region boundaries. The region assignments are computed according to the load-balancing method, and, for some methods, new region boundaries are also computed. For the non-static assignment methods, computing region assignments and boundaries involves estimating the time required to rasterize each region, which is done using the transformed triangles and the method's estimation function. Some of the load-balancing methods

Description	Value
t_{tri}, time to set up triangle for rasterization	18.8 µs
t_{line}, time to do scanline calculations	2.74
t_{pixel}, time to rasterize a pixel	0.23
$t_{tri_receive}$, time to receive a triangle	8.52
t_{reg_ovhd}, time to process each region (estimated)	100
t_{pix_ovhd}, time to process each pixel to clear color and z	0.078

Table 6.1 Timing values used in the simulator.

will use the region-processor assignments calculated during the current frame, while others will use the assignments from the previous frame. The last step is to calculate the amount of work done by each processor using the method's region-processor assignments.

The simulator calculates the amount of work using a performance model of the rasterization algorithm. The rasterization algorithm is the standard z-buffer algorithm, the one that is also used in the implementation described in a later chapter. The z-buffer algorithm performs some calculations for each triangle, another set of calculations for each scan line that the triangle covers, and a third set of calculations for each pixel rendered. The performance model follows the z-buffer algorithm's structure: the model is $t_{tri} + t_{line}h + t_{pixel}a$, where h is the number of scan lines crossed by the triangle, a is the number of pixels in the triangle, t_{tri} is triangle setup time, t_{line} is the scanline interpolation time, and t_{pixel} is the pixel rasterization time. These times were determined by a least-squares fit of the model to the sizes and rasterization times measured on the Touchstone Delta; see Appendix B for details. When calculating the time to rasterize a polygon, the simulator first clips it to the region that it is being rasterized into, counts the number of scanlines and pixels that the triangle covers, then applies the performance model.

Two other values capture overhead costs. The first, t_{reg_ovhd}, gives the per-region overhead, and the second, t_{pix_ovhd}, gives the per-pixel overhead. The first time is estimated, and the second was measured on the Delta. The final value gives the time to receive the triangle: $t_{tri_receive}$. It assumes that the 42-byte triangle record is sent in 4096 byte messages, and was also measured on the Delta. The values appear in table 6.1.

The simulator can use other performance models when evaluating systems that use different algorithms. For example, Pixel-Planes 5 would use a time based on the number of vertices in the polygon.

The simulator outputs the maximum and average of the time each that processor requires to rasterize each frame. The average time divided by the maximum gives the expected processor utilization. The simulator also calculates and outputs the overlap ratio for each frame by summing the number of triangles overlapping each region and dividing by the number of triangles in the viewing frustum.

6.1.2 Performance Model

The simulator returns one value, the processor utilization during rasterization, that may appear to allow one to compare different load-balancing methods. However, that value must be balanced with the other costs associated with the method: the time needed to perform load-balancing plus the time spent on duplicated processing. The performance model gives the overall frame time, taking the utilization and overlap factor calculated by the simulation, statistics about the model, and measured processing times, and returns the expected rendering time.

The performance model will first be developed for the basic algorithm, without considering load-balancing or the region assignment style. The two values from the simulations, the rasterization processor utilization U_R and overlap factor O, depend on the load-balancing method, and thus distinguish the performance resulting from the different methods. The performance model will be developed in pipeline order: first the geometry processing, then the redistribution and rasterization. Later sections will modify the basic performance model to include the load-balancing time. A different performance model will be developed for each of the four region assignment styles.

6.1.2.1 Performance Model for the Basic Algorithm

The first part of the model calculates the time for geometry processing. The time to process each on-screen triangle is the basic per-triangle time t_{geom}, plus the time to copy the triangle into other message buffers when the triangle overlaps more than one region. The latter is modeled is Ot_{bucket}. Multiplying the total per-triangle time by the number of on-screen triangles n_{on} gives total time, $n_{on}(t_{geom} + Ot_{bucket})$. Similarly, the time to process off-screen triangles is $n_{off}t_{geom_off}$, and the time for back-face-rejected triangles is $n_{back}t_{geom_back}$. The per processor time is sum of the three times, divided by the number of processors (assuming perfect load-balancing). Then, we divide that result by U_G, the processor utilization during geometry processing, to get the time given unbalanced loads, giving

$$[n_{on}(t_{geom} + Ot_{bucket}) + n_{off}t_{geom_off} + n_{back}t_{geom_back}]/NU_G .$$

The U_G value comes from the simulations in Chapter 5, and depends on the data set and number of processors.

The time for redistribution depends on whether one- or two-step redistribution is used. The algorithm chooses the faster one automatically using the assumption that the rendering loads are balanced since it is hard to measure how well the loads are balanced. Also, if two-step redistribution is used, the number of groups is calculated assuming balanced rendering loads. However, these assumptions are not used in the performance model to calculate the total frame time. A different performance model must be used for each redistribution type because the redistribution portion of the models depends on the type of redistribution used. The model for one-step redistribution will be developed first; the two-step model follows.

To model one-step redistribution, we need equations for the number of messages and triangles sent per processor. The equations are similar to the ones for two-step redistribution derived in Chapter 4 (see equations 4.2 and 4.3 in section 4.4.6.1). The number of messages sent and received per processor is expected to be the same if the loads are balanced, and is:

$$m_1 = (N-1)\left\lceil \frac{nO}{n_{msg}N^2} \right\rceil. \tag{6.1}$$

The number of triangles sent per processor is:

$$n_1 = \frac{nO(N-1)}{N^2}. \tag{6.2}$$

However, when making work-queue region assignments only one-step stream-per-region redistribution can be used. The other redistribution methods would not get the data to the processor with the newly assigned region in time (see section 4.4 for more information). Here, each processor sends its nO/N triangles in r streams of messages. However, it does not send messages to itself, which removes an average of G message streams per processor. Thus, the number of messages sent (and received) per processor is $m_{1reg} = (r-G)\lceil nO/rNn_{msg} \rceil$. The per-processor number of triangles is unchanged.

The numbers of messages and triangles are converted into times by multiplying by t_{msg_s} and $s_{tri}t_{byte_s}$ when sending messages, and by t_{msg_r} and $s_{tri}t_{byte_r}$ when receiving messages. The time for sending messages is divided by U_G to account for imperfect geometry processing load-balancing, and the time for receiving messages is divided by U_R for rasterization load imbalances.

The amount of work per processor for the rasterization is calculated using a model similar to the one developed in section 6.1.1 for the simulator, but using the average number of scanlines and pixels covered by the triangles ($\bar{h}$ and $\bar{a}$, respectively). The per-triangle cost is multiplied by the overlap factor O, as it is incurred more than once. The per-scan-line cost is multiplied by the square root of the overlap factor, which turns the two-dimensional overlap factor into a one-dimensional value. This is an approximation because the regions are not always square. The overlap factor does not apply to the per-pixel cost since each pixel is rendered only once. Added to this are the overhead costs: the per-region cost t_{reg_ovhd}, and the per-pixel cost t_{pix_ovhd}, respectively multiplied by the number of regions r and the number of pixels A. The total rasterization cost is divided by N to get the per-processor time. That value and the per-processor message receive costs are divided by U_R to correct for the unbalanced rasterization loads.

Putting all the terms together, we get the following equation giving the per-processor time when performing one-step redistribution:

$$
\begin{aligned}
t_1 = & \left[n_{on}(t_{geom} + Ot_{bucket}) + n_{off} t_{geom_off} + n_{back} t_{geom_back} \right] \Big/ NU_G \\
& + (m_1 t_{msg_s} + n_1 s_{tri} t_{byte_s}) / U_G \\
& + \left[n_{on}(Ot_{tri} + \bar{h}\sqrt{O} t_{line} + \bar{a} t_{pixel}) + r t_{reg_ovhd} + A t_{pixel_ovhd} \right] \; NU_R \\
& + (m_1 t_{msg_r} + n_1 s_{tri} t_{byte_r}) \big/ U_R \, .
\end{aligned}
\tag{6.3}
$$

To get the time when performing two-step redistribution, we use the number of messages and triangles sent and received in each step that were developed in Chapter 4, equations 4.2–4.10. Only outgoing messages in the first step and incoming messages in the second step need to be compensated for load imbalances since the formulas developed earlier include the times for the other load imbalances caused when the groups are not all the same size. The compensation is done by dividing times for outgoing first step and incoming second step messages respectively by U_G and U_R. The per-processor time when performing two-step redistribution is:

$$
\begin{aligned}
t_2 = & \left[n_{on}(t_{geom} + Ot_{bucket}) + n_{off} t_{geom_off} + n_{back} t_{geom_back} \right] \Big/ NU_G \\
& + \left[m_{1s} t_{msg_s} + n_{1s} s_{tri} t_{byte_s} \right] \Big/ U_G \\
& + m_{1r} t_{msg_r} + n_{1r} s_{tri} t_{byte_r} + n_{refmt} t_{refmt} + m_{2s} t_{msg_s} + n_{2s} s_{tri} t_{byte_s} \\
& + \left[n_{on}(Ot_{tri} + \bar{h}\sqrt{O} t_{line} + \bar{a} t_{pixel}) + r t_{reg_ovhd} + A t_{pixel_ovhd} \right] \; NU_R \\
& + (m_{2r} t_{msg_r} + n_{2r} s_{tri} t_{byte_r}) \big/ U_R \, .
\end{aligned}
\tag{6.4}
$$

The values for the times and sizes appear in table 6.2. Nearly all of the times are measured values from the Touchstone Delta; the measurement methods are described in Appendix B. The estimated times are marked with "(est.)." The estimation methods are also described in Appendix B.

6.1.2.2 Performance Models Including Load-Balancing

Now that we have developed the performance models for the basic algorithm, we will modify them so that they include the time for performing the load-balancing operations. These models will be used later to evaluate each region assignment style. Four sub-sections follow, each giving the model for one assignment style.

Symbol	Value	Description
t_{geom}	7.9 µs	time to transform, clip test, and light a triangle
t_{bucket}	2.7	time to determine buffer and copy triangle to it
t_{geom_off}	6.6	time to transform and clip test an off-screen triangle
t_{geom_back}	6.1	time to transform and clip test a back-facing triangle
t_{msg_s}	208	time to send a message
t_{msg_r}	296	time to receive a message (using a message handler)
t_{byte_s}	0.11	time to send a byte
t_{byte_r}	0.13	time to receive a byte
t_{refmt}	4.1	time to reformat a triangle between messages (est.)
t_{tri}	18.8	time to set up triangle for rasterization
t_{line}	2.74	time to do scanline interpolations and calculations
t_{pixel}	0.23	time to rasterize a pixel
t_{reg_ovhd}	100	time to process each region (overhead) (est.)
t_{pix_ovhd}	0.078	time to process each pixel to clear color and *z*
s_{tri}	42 bytes	size of a triangle
s_{msg}	4096 bytes	size of a message

Table 6.2 Constants used in the performance model. Estimated times are marked with "(est.)."

6.1.2.2.1 Time When Using Static Region Assignments

The frame time when using static region assignments is simply the time for the basic algorithm since it has no load-balancing work. Of course, the overlap factor and processor utilization would be from a simulation that uses static region assignments. (This does not account for the synchronization between processors needed to insure that all have finished the frame, but this does not affect the argument since all the methods would need to include it.)

6.1.2.2.2 Time When Using Between-Frames Region Assignments

The performance model is much more complex when the assignments are made between frames. The assignments for the next frame are made during the current frame's rasterization so that the load-balancing latency can be at least partially hidden. The latency is only hidden when the communication and calculations can be performed more quickly than the rasterization. The complexity comes because we must calculate the time within a frame where load-balancing can be overlapped, which requires redeveloping most of the performance model. First, we develop the time taken by the load-balancing calculations, and then calculate the partial frame time that can hide the calculations.

The components of the load-balancing performance model can be divided into three categories. Some components are part of the serial load-balancing, where the latency of the operation determines whether processors have to wait for the result; for example, the time required to gather the per-region cost estimates. The second category of components involve processing on every processor, so the amount of processing is totaled and added to the overall frame time. An example of this category is the processing time incurred by each processor when broadcasting the region-processor assignments. The third set of components contributes to the load-balancing latency and also requires processing, so those components are part of both the serial load-balancing time and the overall frame time. However, the processor performing these load-balancing tasks can be relieved of other tasks, so the processing time is effectively partitioned among all the processors. The assignment of regions to processors falls into the latter category.

The time needed to perform the region assignments includes the time to gather the per-region cost estimates, the time to make the assignments, and the time to broadcast the assignments. We will first compute the times that are part of the serial load-balancing, where the latency is computed, and then the per-frame processing times.

The time to make the region-processor assignments was measured on the Touchstone Delta and was described earlier in section 4.3.6. Making the assignments involves sorting the gathered per-region costs and then using the greedy bin-packing algorithm. The two components are denoted t_{sort} and t_{assign}. These processing times are part of both the load-balancing latency and the overall frame time.

The costs are gathered by organizing the processors in a binary tree, with each processor receiving up to two messages from other processors. The number of levels in the tree is $\lceil \log_2 N + 1 \rceil - 1$. At each level, a processor must receive two messages and transmit one. The per-message send and receive times are both in the critical path since the per-message processing must be done before the first byte is transferred. However, only the per-byte cost of receiving messages contributes to the latency since the message sending is overlapped with the receiving message, and the per-byte receive cost is higher. Since four bytes are sent per region, the time per level is $(t_{msg_s} + 2t_{msg_r} + 8rt_{byte_r})$, giving a total time to gather the region costs of

$$l_{gather} = \left(\lceil \log_2 N + 1 \rceil - 1\right)\left(t_{msg_s} + 2t_{msg_r} + 8rt_{byte_r}\right). \tag{6.5}$$

The time is represented by an l since it represents the latency required for gathering the costs instead of processing time.

The time to broadcast the assignments is similar to gathering the estimates, but a processor has to send two messages instead of receiving two messages, and only two bytes are sent per region. Since a processor sends two messages and receives only one, and $2t_{byte_s}$ is greater than t_{byte_r}, the per-byte sending time is part of the critical path. So, the broadcast time is

$$l_{broadcast} = \left(\lceil \log_2 N + 1 \rceil - 1\right)\left(2t_{msg_s} + t_{msg_r} + 4rt_{byte_s}\right). \tag{6.6}$$

Besides the latency required, the processing time must also be modeled, since it directly increases frame time. When gathering the region costs, in the worst case a processor must receive two messages and send one message, so the total processing time is

$$t_{gather} = t_{msg_s} + 2t_{msg_r} + 4r(t_{byte_s} + 2t_{byte_r}). \tag{6.7}$$

The time to broadcast the region assignments follows from the last two equations:

$$t_{broadcast} = 2t_{msg_s} + t_{msg_r} + 2r(2t_{byte_s} + t_{byte_r}). \tag{6.8}$$

The last part needed to compute the overall frame time is the time available to compute the new region assignments, the time *after geometry processing*, or AGP. The performance model developed in section 6.1.2.1 can be adapted to give this value. The time includes both the rasterization time and a portion of the communication time, since the cost estimates can be sent by the leaf processors in the gathering tree (those processors that do not receive messages) as soon as they have finished processing all their triangles, before they flush their buffers.

To compute the portion of the communication time, we must compute the amount of the communication that remains AGP, that is, the number of messages and bytes sent per processor. When using one-step stream-per-processor redistribution, each processor will send one buffer to each of the other processors AGP, so the number of messages sent AGP is $m_{1_agp} = N - 1$ (compare this with equation 6.1). Any buffer sent earlier will have been completely filled, so the number of triangles in the remaining buffer is the total number of triangles sent by the processor to each other processor modulo the message size:

$$n_{1_agp} = (N-1)\left(\frac{nO}{N^2} \bmod n_{msg}\right).$$

When using two-step sending, the amount of communication after the end of geometry processing has five components. Using the same arguments used for the one step case, we can modify equations 4.2 and 4.3 to get $m_{1s_agp} = (g-1)$, the number of first step messages sent AGP, and $n_{1s} = (g-1)(nO/Ng \bmod n_{msg})$, the number of triangles sent AGP. Triangles received in the first redistribution step suffer from a load imbalance, as described in section 4.4.6.1. In the worst case, a processor receives $m_{1r_agp} = \lceil (N - N_g)/N_g \rceil$ messages instead of the $g - 1$ sent. The number of triangles received follows the same pattern:

$$n_{1r_agp} = \left\lceil \frac{N - N_g}{N_g} \right\rceil \left(\frac{nO}{Ng} \bmod n_{msg}\right).$$

The number of triangles reformatted by each processor follows from equation 4.10 and the one for n_{1r_agp}:

$$n_{refmt_agp} = \left(\left\lceil \frac{N - N_g}{N_g} \right\rceil + 1\right)\left(\frac{nO}{Ng} \bmod n_{msg}\right).$$

The number of triangles sent in the second step AGP is equal to the sum of the number of triangles received AGP in the first step, and the number of triangles that had arrived earlier but are still waiting in partially filled buffers. For simplicity, I ignore the load imbalances due to some processors receiving more than the average number of triangles. In the first step, each pair of processors sends an average of nO/Ng triangles, so each of the g first-step buffers will have $nO/Ng \bmod n_{msg}$ triangles at the end of geometry processing, which means that $nO/Ng - (nO/Ng \bmod n_{msg})$ triangles will have been sent beforehand. Since, on average, each processor receives messages from each of the other $g - 1$ groups plus its own messages for the group, it will have received a total of

$$g\left[\frac{nO}{Ng} - \left(\frac{nO}{Ng} \bmod n_{msg}\right)\right]$$

triangles. Those messages will have been divided into buffers for each of the N_g processors within the group and any full buffers sent, so each processor will have retained

$$n_{2s_buf} = \left(\frac{g}{N_g} \left[\frac{nO}{Ng} - \left(\frac{nO}{Ng} \bmod n_{msg} \right) \right] \right) \bmod n_{msg}$$

triangles in each buffer. Each processor must also send the unsent triangles from the first step. Each processor will receive triangles from g groups, which are then divided among the N_g processors in the group. Thus, each processor will send the following number of triangles in the second step:

$$n_{2s_agp} = (N_g - 1) \left[n_{2s_buf} + \frac{g}{N_g} \left(\frac{nO}{Ng} \bmod n_{msg} \right) \right].$$

The number of messages sent by a processor to each of the other processors in the group is computed by dividing the number of triangles sent per processor pair by the number of triangles in a message and taking the ceiling. Each processor sends that number of messages to each of the other N_g processors in the group, so each processor sends the following number of messages in the second step:

$$m_{2s_agp} = (N_g - 1) \left\lceil \frac{1}{n_{msg}} \left[n_{2s_buf} + \frac{g}{N_g} \left(\frac{nO}{Ng} \bmod n_{msg} \right) \right] \right\rceil.$$

Since we are not modeling load imbalances, in the second step the number of messages and triangles sent and received per processor AGP is the same, i.e., $n_{2s_agp} = n_{2r_agp}$ and $m_{2s_agp} = m_{2r_agp}$.

Given these formulas for the amount of communication after the end of geometry processing, we can modify the formulas for the entire frame time (equations 6.3 and 6.4) to get the total AGP time. The modification is simple: replace the communication symbols with the corresponding AGP ones, and delete the work associated with geometry processing. When using one-step redistribution, the time is

$$\begin{aligned} t_{1_agp} = {} & (m_{1_agp} t_{msg_s} + n_{1_agp} s_{tri} t_{byte_s}) / U_G \\ & + \left[n_{on} (Ot_{tri} + \bar{h} \sqrt{O} t_{line} + \bar{a} t_{pixel}) + r t_{reg_ovhd} + A t_{pixel_ovhd} \right] NU_R \\ & + (m_{1_agp} t_{msg_r} + n_{1_agp} s_{tri} t_{byte_r}) / U_R , \end{aligned}$$

and when using two-step redistribution the time is

$$\begin{aligned} t_{2_agp} = {} & (m_{1s_agp} t_{msg_s} + n_{1s_agp} s_{tri} t_{byte_s}) / U_G \\ & + m_{1r_agp} t_{msg_r} + n_{1r_agp} s_{tri} t_{byte_r} + n_{refmt_agp} t_{refmt} + m_{2s_agp} t_{msg_s} + n_{2s_agp} s_{tri} t_{byte_s} \\ & + \left[n_{on} (Ot_{tri} + \bar{h} \sqrt{O} t_{line} + \bar{a} t_{pixel}) + r t_{reg_ovhd} + A t_{pixel_ovhd} \right] NU_R \\ & + (m_{2r_agp} t_{msg_r} + n_{2r_agp} s_{tri} t_{byte_r}) / U_R . \end{aligned}$$

With those formulas, the overall frame time is

$$\begin{aligned} t_{betw_fr} = {} & t_{rend_betw_fr} + t_{gather} + t_{broadcast} + (t_{sort} + t_{assign}) / N \\ & + \min(0, l_{gather} + l_{broadcast} + t_{sort} + t_{assign} - t_{agp}), \end{aligned}$$

where $t_{rend_betw_fr}$ is the overall frame time (t_1 or t_2 depending on the redistribution method) using the best between-frames load-balancing method, and t_{agp} is similarly t_{1_agp} or t_{2_agp}. The times for t_{sort} and t_{assign} are amortized over all the processors because they are only done once. However, this assumes that the processor performing the load-balancing calculations will be given less rasterization work; that is, we include the load-balancing calculations in the overall load-balancing operations. Section 7.1.4 will give details on how this is done.

6.1.2.2.3 Time When Using Between-Stages Region Assignments

Because the between-frames and between-stages region assignment styles perform the same operations but at different times, the frame-time formula for the between-stages style uses several terms from the between-frames formula. If the geometry processing is perfectly load balanced, the region assignments are made when the processors are idle. Thus, the time to gather the costs and then compute and broadcast the assign-

ments is simply added to the overall rendering time. This time is the same as the latency values for between-frames assignments. The sorting and assignment times are also the same. Thus, the total time is

$$t_{betw_stages} = t_{rend_betw_stages} + l_{gather} + l_{broadcast} + t_{sort} + t_{assign}, \tag{6.9}$$

where $t_{rend_betw_stages}$ is the rendering time using the best between-stages load-balancing method.

6.1.2.2.4 Time When Using Work-Queue Assignments

When making work-queue region assignments, an algorithm must first do the geometry processing and then perform the rasterization. The rasterization should be done from the most complex region to the simplest to insure that complex regions are not started towards the end of the frame. This is done by gathering per-region cost estimates as done with the once-per-frame assignment methods and sorting them. Then, when a region is assigned it is broadcast to all the processors, taking a total processing time of

$$r[t_{msg_s} + t_{msg_r} + 4(2t_{byte_s} + t_{byte_r})].$$

This assumes that the operating system broadcast function is used, which only incurs the per-message sending cost once. However, we include the per-byte sending cost twice since it is harder to optimize. The message size is 4 bytes (2 bytes for region id and 2 bytes for processor id). So, the overall time is

$$t_{work_queue} = t_{rend_work_queue} + l_{gather} + t_{sort} + r[t_{msg_s} + t_{msg_r} + 4(2t_{byte_s} + t_{byte_r})]. \tag{6.10}$$

The formula uses the rendering time for the best work-queue load-balancing method. The performance model also uses the stream-per-region formula for the number of messages sent per processor.

6.1.2.2.5 Comments on the Performance Models

The formulas for the load-balancing time suffer from four types of inaccuracies. First, they do not capture effects due to network congestion. For example, the time to gather the per-region cost estimates will be higher if the network is near saturation. Also, the first two assignment styles allow redistribution for the entire frame, whereas the last two methods only allow redistribution during the rasterization step. However, the second effect is not so pronounced when using more than 64 to 128 processors, depending on the size of the model. With enough processors for a given model, there is enough buffering to hold all the triangles during geometry processing, which means redistribution is only performed after geometry processing has been completed.

Some of the load-balancing algorithms (to be described) change the region boundaries. The time to calculate the changes is not included in the current performance models. However, the time should be a small fraction of the current time. The boundary adjustment algorithm takes time proportional to the number of regions, with only a few operations per region, instead of the $O(r \log r)$ time required for the other load-balancing calculations.

The formulas also assume that load-balancing calculations are done on a single processor. Both the sorting and region assignments can be done in parallel. The sorting could be done using a Quicksort-type method, with one portion of the region estimates sorted on another processor after the first pivoting. The region assignments could be done, for example, by two processors. Each processor would assign regions to half of the processors, with the first processor assigning the even-numbered regions after sorting, and the second assigning the odd-numbered regions. Since the odd- and even-numbered sorted elements are not guaranteed to have the same sum, one set of processors could be given more work to perform than the other set, which would result in a slightly lower processor utilization. Parallelizing the load-balancing calculations will not result in a large improvement because they take at most one third of the overall load-balancing latency.

Finally, the geometry simulation is done separately from the rasterization, which means that U_G only depends on the model and the number of processors, and not on G or the position of the region boundaries. A larger value of G increases the number of region boundaries, which increases the probability that a triangle crosses a boundary and would have to be sent more than once, which in turn increases the standard deviation of the per-triangle geometry processing time. This means that the standard deviation of the per-processor geometry processing time will also increase, resulting in lower processor utilization. However, because the difference in processing cost between a triangle overlapping one region compared to a one

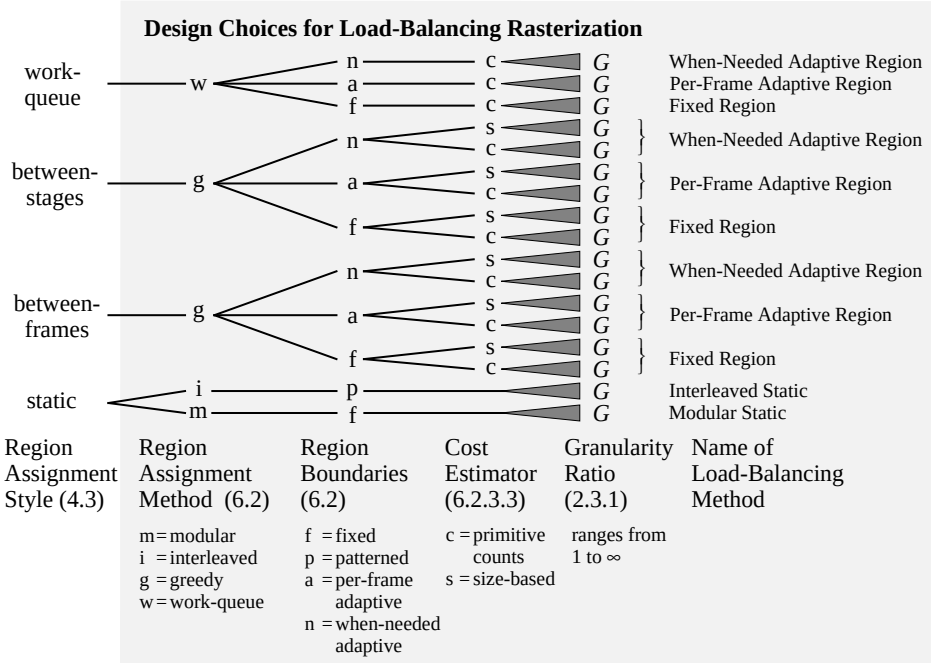


Figure 6.1 Taxonomy of the design choices described in this section with the names of the load-balancing methods (see figure 1.2 for the entire taxonomy tree).

overlapping two regions is smaller than the difference in cost between a back-face rejected triangle and a normal triangle, this decrease in utilization should be smaller than the decrease seen when back-face culling is enabled. The maximum decrease due to enabling back-face culling is only 7% (see figure 5.5 and 5.6).

6.2 Load-Balancing Methods

Now that the evaluation methodology has been introduced, we need to describe the load-balancing methods that will be considered. A load-balancing method specifies both how the screen is divided into regions as well as how the regions are assigned to processors. Since the style of region assignments has not yet been fixed, load-balancing methods for each style will be considered. The region assignment style and load-balancing method are important algorithm parameters since poor choices can halve the performance.

I consider six different design choices for load-balancing methods. A taxonomy of possible load-balancing methods resulting from the four most important design choices is shown in figure 6.1. The first three design choices determine how the screen is broken into regions:

- The first choice, the granularity ratio, determines the number of regions created per processor. I only consider methods that fix the granularity ratio instead of varying it at run time. Large ratios more evenly balance the processors' loads but also increase the overhead costs associated with triangles that overlap multiple regions.

- The second region boundary choice is the technique used for computing the region boundaries. Some techniques fix the region boundaries at system initialization, and others vary it during run time. I only consider region boundaries that form a grid, where the regions are formed by cutting the screen from top to bottom and from left to right. This constraint results in lower bucketization costs compared to bucketization with more general region boundaries.

- The last region-boundary design choice is the region aspect ratio. I only briefly consider this choice because others have shown that the best choice is to use square regions; thus, it is not shown in figure 6.1.

The last three design choices determine how regions are assigned to processors. The first is the region assignment method, the algorithm for making the assignments. The assignment method must work with the region assignment style, since the style determines the time when the assignments are made, and thus the

information available to the assignment method. I only consider one assignment method for most of the assignment styles, but many others are possible. The last two design choices specify the method for estimating the time to rasterize each region. One choice is the method for estimating the time of an individual primitive. The second design choice is not shown in figure 6.1 because it is not part of the main investigation. It specifies the number of processors from which the timing estimates are gathered. Using a subset of the processors is faster but results in less accurate estimates.

Because not all combinations of load-balancing methods and region assignment styles make sense, the load-balancing methods for each style of region assignments are described in separate sections. However, the two once-per-frame assignment styles, the between-frames and between-stages styles, are described in the same section since they use quite similar load-balancing methods.

The first subsection considers the region aspect-ratio design choice. Section 6.2.2 describes two load-balancing methods for static region assignment, when the assignments are made during system initialization. The following section describes methods for the two region assignment styles that make the assignments once for each frame; one makes the assignments between the geometry processing and rasterization stages, and the second makes them between frames. The last section, section 6.2.4, describes methods when regions are assigned during rasterization, using a work-queue approach. These methods are quite similar to the methods used with once-per-frame region assignments.

6.2.1 Region Aspect Ratio

One parameter of all the load-balancing methods is the region aspect ratio. Work by Whelan [Whel85] and Whitman [Whit92] have shown that square regions work better than rectangular regions for most scenes. This makes intuitive sense: on average, square regions would have a smaller overlap factor than long, thin regions. More evidence comes from the formula for estimating the overlap factor

$$O = \left(\frac{W+w}{W}\right)\left(\frac{H+h}{H}\right),$$

which was discussed in section 3.2.2.2. The symbols W and H are the width and height of the region, and w and h are the width and height of the triangle's bounding box. For a fixed region area of A_r pixels, and a region aspect ratio of $\alpha_r = W/H$ with square primitives ($w = h$), substituting $W = \alpha_r\sqrt{A_r}$ and $H = \alpha_r^{-1}\sqrt{A_r}$ into the equation above results in

$$O = \left(\frac{\alpha_r\sqrt{A_r}+w}{\alpha_r\sqrt{A_r}}\right)\left(\frac{\alpha_r^{-1}\sqrt{A_r}+w}{\alpha_r^{-1}\sqrt{A_r}}\right).$$

Taking the derivative of O with respect to α_r and solving for 0, we find the minimum overlap occurs when $\alpha_r = 1$, i.e., when the regions are square. Because the benefit of using square regions makes intuitive sense, and because it has been studied before, the region aspect ratio will not be explored with the simulation runs: only square regions will be used whenever possible.

6.2.2 Load-Balancing Methods for Static Region Assignments

I consider two methods that use static region assignments. The main advantage of these methods compared to the others is that there are no per-frame load-balancing costs. If load-balancing is not the bottleneck, then these methods will give poorer performance than ones that use other region assignment styles. One method assigns the processors arbitrarily, and the second method interleaves the processors in a regular pattern. While the interleaved method does not work as well as the first method with certain numbers of processors, it usually gives better results. The first method is primarily used in the first frame when between-frames region assignment is used because there is no previous frame to make the assignments. Each method uses a different algorithm to divide the screen into regions. The region boundaries are calculated during system initialization.

6.2.2.1 Modular Static Load-Balancing

The first static load-balancing method supports non-integral granularity ratios. The number of regions horizontally uses the aspect ratio of the screen α (width/height) to make the regions approximately square. The number of regions vertically and horizontally respectively are $r_v = \left\lfloor \sqrt{GN/\alpha} \right\rfloor$ and $r_h = \left\lfloor GN / r_v \right\rfloor$,

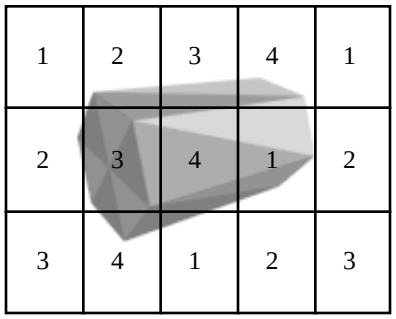


Figure 6.2 Example of modular static load-balancing with $N = 4$, $G = 4$, and $\alpha = {}^{5}/_{4}$. The lines show the region boundaries and the numbers show processor assignments.

where G is the desired granularity ratio. The actual number of regions is close to, but not necessarily equal to, the desired number of regions GN. The difference is generally quite small, and allows for the regions to be more square. If there were instead always exactly GN regions (assuming GN is an integer), the regions might be forced to have poor aspect ratios.

In this method, a region numbered r is assigned to processor r mod N. Some processors will usually have one more region to render than other processors. Figure 6.2 gives an example using a cylinder model with $N = 4$, $G = 4$, and $\alpha = {}^{5}/_{4}$.

6.2.2.2 Interleaved Static Load-Balancing

The second static load-balancing method, interleaved static load-balancing, only supports integral values of G. The integral values are required because the regions are assigned to processors in an interleaved fashion. The processors are placed into a footprint of i by j processors ($ij = N$). Every i^{th} region horizontally and j^{th} region vertically is assigned to the same processor. Different footprints are possible. The best footprint first has square regions, and second forms a square block of regions. Having a square footprint means that a processor is assigned a set of non-contiguous regions. Since much of the rasterization work is often concentrated in a few small parts of the screen, assigning non-contiguous regions to a processor decreases the probability that a processor is assigned regions that all have a large amount of rasterization work. To choose the best footprint, all the possible footprints should be checked, and the one with the most square regions selected. In the usual case, where the aspect ratio of the screen is near one, this will also choose a near-square footprint. However, in some cases the regions and/or the footprint cannot be made square. In those cases, the other static load-balancing method (modular static) should give better results.

Figure 6.3 gives an example with $N = 4$ and $G = 4$. Several footprints are possible. The 2x2 footprint is used as it makes both the regions and the footprint somewhat square.

Static interleaved region assignment has been investigated previously by Whelan [Whel85]. Other hardware architectures use similar methods, but interleaving scanlines or single pixels instead of regions [Hu85, Fuch79, Park80, SGI90]. The latter systems have large overlap factors so they are not suitable for use on general purpose multicomputers.

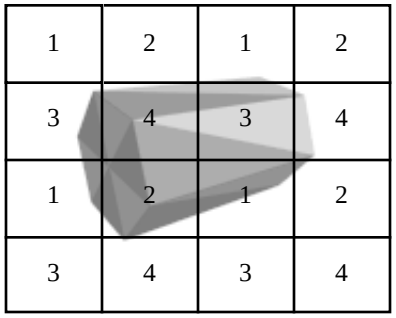


Figure 6.3 Example of interleaved static load-balancing with $N = 4$, $G = 4$, and $\alpha = {}^{5}/_{4}$. The lines show the region boundaries and the numbers show processor assignments.

6.2.3 Load-Balancing Methods for Once-Per-Frame Assignments

The two once-per-frame region assignment styles either perform the assignments between the geometry processing and rasterization stages, or between successive frames. The load-balancing computations are the same for both styles; the only difference is that the assignments are immediately used with the between-stages style, and are used in the next frame with the between-frames style. Because they are so similar, load-balancing methods for the two assignment styles will be described at the same time, although they will have to be evaluated separately.

The load-balancing methods considered involve three independent choices: one for how the screen is partitioned, and two for what information is used to assign regions to processors. I will consider three different screen partitionings. One uses a fixed number of regions, all the same size. Another adjusts the region boundaries each frame, attempting to make each region have the same amount of work. The third method only adjusts the region boundaries when one region takes too long to rasterize, when the region is the bottleneck. By reducing the region size around screen hot spots, the adaptive-region methods may be able to use a smaller granularity ratio than ones using fixed region boundaries.

The second choice is the method for estimating the rasterization time of a single triangle. I consider two methods, one that simply counts triangles in each region, and another that considers the size of each triangle. The second method requires more calculations than the first, but should also result in a more even assignment of tasks to processors. A later section will show that using the more expensive method balances the loads only slightly more evenly because the models considered mainly have triangles that contain only a few pixels.

The third choice is whether to collect cost estimates from all the processors or from a subset. Gathering the costs from fewer processors saves time but, since the costs are not as representative of the actual times, it does not balance the loads as well.

The once-per-frame load-balancing methods share the algorithm for assigning regions to processors. The regions are assigned to processors using the greedy multiple bin packing described in section 4.3.2. The regions are first sorted according to the estimated rasterization time. The regions are then assigned to processors from the most time consuming to the least, at each step giving the region to the processor currently having the lightest load. The lightly loaded regions are assigned last so they can “fill in” any unevenness created when assigning the heavily loaded regions. As mentioned earlier in section 4.3.2, the greedy region assignment algorithm does not make the optimal assignments. An optimal algorithm would be NP-complete, and would take too long to compute. The simulation results (in section 6.5) show that the greedy algorithm does give good results. Note that the inefficiency from using the greedy algorithm is in addition to any inefficiency resulting from estimated rasterization times differing from the actual times.

The next sections first describe the methods for partitioning the screen into regions, and then describe and develop the two different per-region cost estimates.

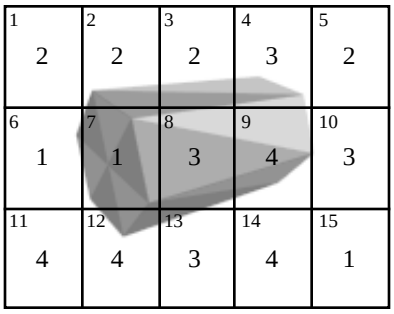
6.2.3.1 Fixed Region Load-Balancing

The first once-per-frame load-balancing method assigns fixed regions to processors. It uses the same same-size region boundaries as used with the modular static load-balancing method (described in section 6.2.2.1), so any granularity ratio can be used. The first step of the assignment calculates an estimate of the time needed to rasterize each region. The estimate can use any of the methods that will be described in section 6.2.3.3. The overhead for each region must also be added: the time to clear the region and send it to the frame buffer. The overhead estimate is generally a function of the number of pixels in the region.

An example of the algorithm is shown in figure 6.4. The assignment algorithm for the figure uses a value of one for the triangle time estimate, and three for the region overhead estimate. Back-facing triangles were removed. The figure uses $N = 4$ and $G = 4$. Whelan [Whel85] and Whitman [Whit92,Whit94] have used similar methods for dividing the screen.

6.2.3.2 Adaptive-Region Load-Balancing

When using adaptive regions, the goal is to split the screen so that each region will take the same time to rasterize. This can allow either one region to be assigned per processor, with a low overlap factor, or it can more finely divide the more time-consuming portions of the screen to avoid having a single region be a bot-



Region	Triangles	Region	Triangles
1	1	9	5
2	5	10	0
3	3	11	0
4	3	12	3
5	0	13	1
6	2	14	1
7	11	15	0
8	5		

Figure 6.4 Example of assigned-region load-balancing with $N = 4$, $G = 4$, and $\alpha = {}^{5}/_{4}$. The lines show the region boundaries, the large numbers in the region centers show the processor assignments, and the small number in each region's corner shows the region number. The table at the right shows the number of triangles in each region (recall that a triangle's bounding box is used to determine whether it overlaps a region). The regions are assigned in the following order: 7, 2, 8, 9, 3, 4, 12, 6, 1, 13, 14, 5, 10, 11, and 15.

tleneck. Whelan [Whel85], Whitman [Whit92], and Roble [Robl88] have each investigated different adaptive-region algorithms. All found the method somewhat lacking: either the time required to divide the screen was too large, the rasterization loads were not evenly divided, or a fixed region method was faster. Those earlier algorithms only determined the region boundaries after collecting information about the distribution of triangles over the screen. For two of the earlier methods, this requires that each triangle be processed twice, once to collect the triangle distribution statistics and a second time for normal bucketization. Two of the earlier algorithms, by Whelan and Roble, also use a single region per processor.

Instead, I consider adaptive methods that do not use a separate pass over the model to collect statistics for determining the region boundaries. I use the per-region costs gathered to assign regions to processors to also adjust the region boundaries for the next frame. Doing so reduces the cost of adjusting the region boundaries. The earlier works could not use this method because they were designed to render one frame per run. Additionally, I use more than one region per processor, as was done by Whitman. Finally, I constrain the region boundaries to form a regular grid, as described in the introduction to the load-balancing section (section 6.2).

Given these constraints and goals, when using adaptive region boundaries the load balancing for each frame progresses as follows. At the start of the frame, the region boundaries are set for the frame. After geometry processing stage is run, the per-region cost estimates are gathered. The estimates are then used to compute the new region boundaries and the region-processor assignments. The new region boundaries cannot be used for the current frame's rasterization because the triangles have already been bucketized using the old region boundaries. Instead, the region boundaries are used in the next frame.

The algorithm for computing the region-processor assignments depends on the region assignment style in use. With the between-stages assignment style, the assignments are made using the per-region cost estimates just gathered, and are then immediately used for the same frame's rasterization. When using between-frames region assignment, the assignments will be used in the next frame, along with the new region boundaries. This means that the per-region cost estimates should be adjusted to reflect the new boundaries before making the region-processor assignments.

The new region boundaries are computed by summing the per-region costs for each row and column of regions. The new region boundaries are adjusted as if all of the rows of pixels within a row of regions have the same cost, and every column of pixels with a column of regions similarly has the same cost. This approximation must be made because we only have cost estimates for entire regions. Given this approximation, it is simple to compute the new region boundaries: the algorithm just finds the new region boundaries that place the same amount of work in each row and column of regions.

When using between-frames region assignment, the per-region cost estimates are adjusted whenever new region boundaries are computed. This is done using the approximation that the time to rasterize a pixel is uniform across each region. Conceptually, the cost to render each pixel is set to the cost of the region to which it belongs divided by the number of pixels in the region (this is done using the old region boundaries). Since each pixel has a cost associated with it, each new region's cost is the sum of the pixel costs

for each pixel in the new region. The approximation that rasterization cost is the same for each pixel in a region will decrease the accuracy of the adjusted region cost estimates.

Computing the new region boundaries can be efficiently implemented using a summed-area table [Crow84], but with each entry in the table corresponding to a region cost instead of a texture pixel. Briefly, the method works as follows. By using a summed-area table of the old region costs, each new region cost can be calculated with the same formulas used to calculate a pixel's color when texturing using summed area tables. A further optimization is to create the summed-area table, resample it according to the new region boundaries (so the entries sum according to the new entries), and then "unsum" the summed-area table, turning it into the new array of region cost estimates.

The region adjustment method depends on frame-to-frame coherence. If there is not sufficient coherence, such as when the user specifies a completely new viewpoint, the new region boundaries may not divide the workload evenly. The new regions will not have the desired nearly equal amount of work. Instead, one of the regions could have so much work that it is the bottleneck for rasterizing the frame. However, this use of coherence is much less sensitive than the use of coherence for region processor assignments, where the assignments are made in one frame and used in the next.

6.2.3.2.1 Adjusting the Boundaries Every Frame

I looked at two techniques for adjusting the boundaries. The first technique adjusts the boundaries every frame, using the methods just described. Figure 6.5 shows an example of the adjusted boundaries for $N = 6$ and $G = 4$. Here, the estimated costs are calculated using an estimate that considers the size of each triangle (back-facing triangles were included in the estimates). Each frame shows the region boundaries used to rasterize the current frame. The boundaries for the next frame are calculated from the cost estimates gathered from the current frame's region boundaries. The region boundaries for the first frame are calculated using the algorithm described in section 6.2.2.1.

6.2.3.2.2 Adjusting the Boundaries When Needed

The second region-boundary adjustment technique usually does not adjust the boundaries in each frame. Instead, the boundaries are only adjusted when one region is the bottleneck, when it represents more than $1/N$ of the entire workload. Because the region cost estimates are only adjusted for some of the frames, the cost estimates for the other frames do not have their accuracy reduced by adjusting them to the new region boundaries. Thus, this technique does a better job at balancing the loads. Note that the determination of whether a region is the bottleneck uses the estimated region costs, not the actual rasterization times.

This technique could be generalized to a family of load-balancing methods where the decision to adjust the boundaries depends whether a region has more than k/N of the entire workload, where k is a constant in the range of 0 to 1. If k is less than 1 then the boundaries would be adjusted before a region is found to be a bottleneck. I only use the case where $k = 1$.

Figure 6.6 shows an example of adjusting the region boundaries when needed, for $N = 6$ and $G = 2$. The regions boundaries are only changed between frames 1 and 2 and between frames 6 and 7.

6.2.3.3 Cost Estimates

The once-per-frame load-balancing methods make the region-processor assignment costs using an estimate of the time, or cost, to rasterize each region. The last two algorithm choices concern the estimates of the per-region costs. One choice is the function used to estimate a triangle's rasterization cost. The other choice is whether to use estimates from either all the processors or a subset of them. The first choice trades geometry processing time against the load-balancing quality, and the second choice trades load-balancing time against load-balancing quality. Simulations later in the chapter evaluate how the choices affect the load-balancing quality.

6.2.3.3.1 Cost Function

Many different functions could be used to estimate the time to rasterize a primitive. Three possibilities are listed below, from the simplest to most complex:

- for each primitive type, a count of the number of primitives of that type multiplied by a fixed cost for that type.

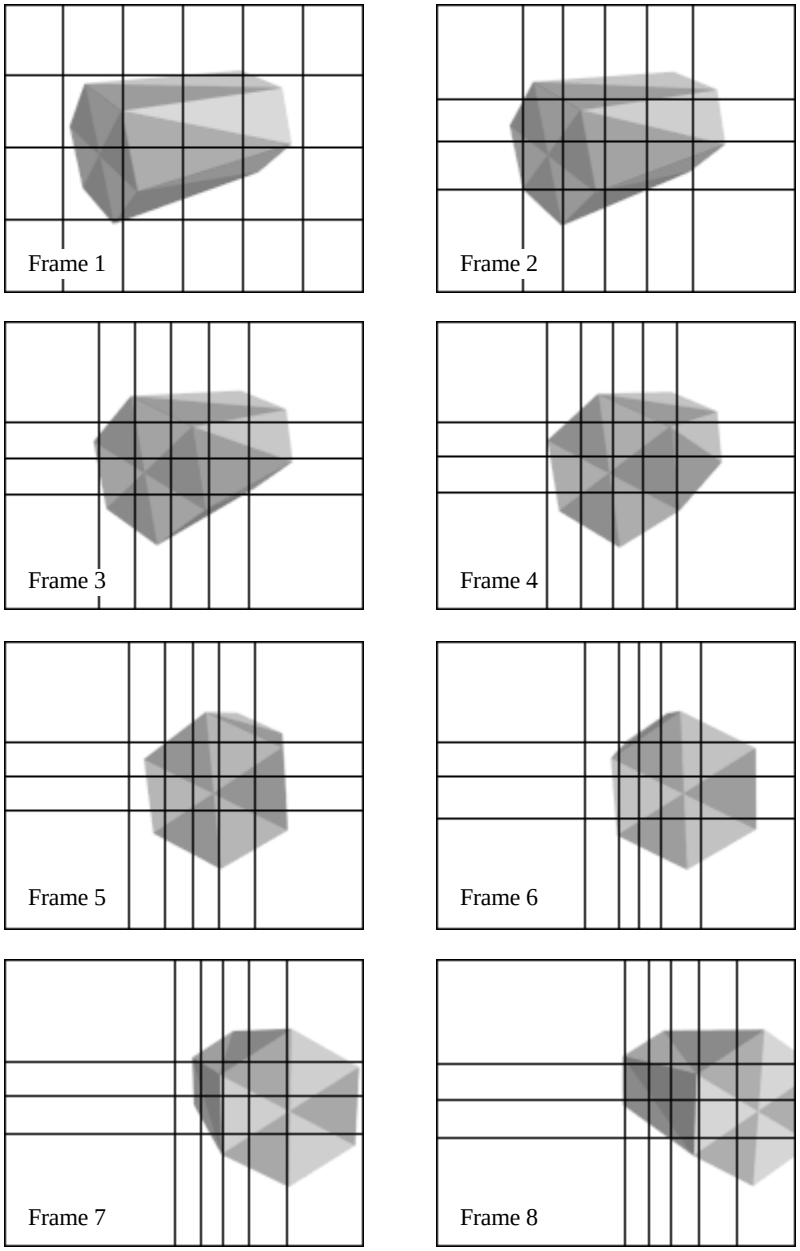


Figure 6.5 Example of adjusting the region boundaries for eight frames.

- a simple function of cost for each of the primitives, based on an intrinsic measure of the primitive. For example, the measure could be the v, number of vertices in the polygon, and the cost function $c_1 + c_2 v$, where c_1 and c_2 are constants. Or, for Bézier patches, the measure could be the degree of the patch.

- a function based on the screen-space size of the primitives.

The first cost estimate has a very low implementation cost, and is the simplest. It has been used by Whelan [Whel85] and Roble [Robl88]. The estimate can be fairly accurate if the primitives are all small or are roughly the same size. The second estimate is a better approximation. For some systems, like Pixel-Planes 5, this function closely models the actual time for on-screen primitives. For systems without specialized rendering processors, this function is still a poor estimate of the overall cost.

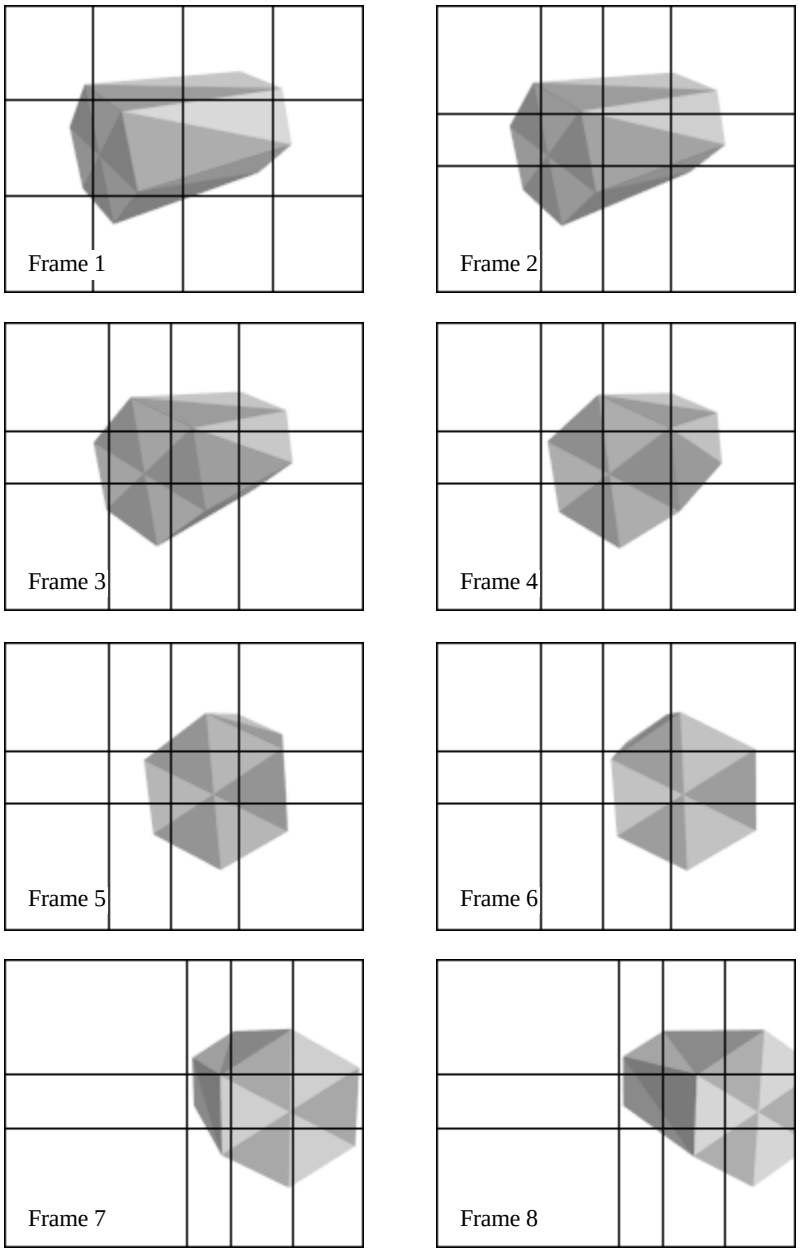
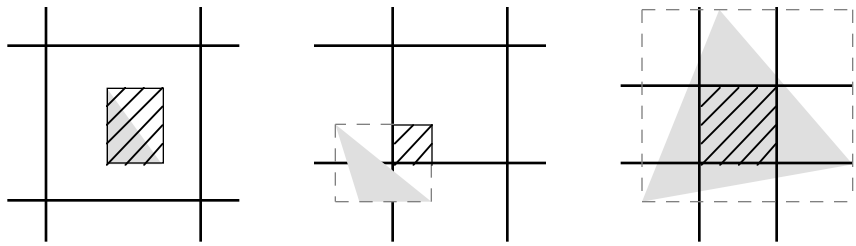


Figure 6.6 Example of when-needed adjustment of region boundaries. The boundaries are only adjusted between frames 1 and 2 and between frames 6 and 7.

The last cost estimate has the highest implementation cost. One possible function for triangles uses the intersection of the triangle's bounding box and the region in question. This function is $c_1 + c_2 h_{bb} + c_3 a_{bb}$, where c_1–c_3 are constants, h_{bb} is the number of scan lines within the triangle's bounding box clipped to the region, and a_{bb} is the area of the triangle's bounding box that is within the region. The value of the constants should be related to the constants from a performance model of the actual rasterization code that has the form of $t_{tri} + t_{line}h + t_{pixel}a$, where h and a are the actual number of scan lines and pixels in the triangle, and t_{tri}, t_{line}, and t_{pixel} are constants determined using a least-squares fit to the actual rasterization times. The value of c_1 should be t_{tri} since the triangle overhead is always incurred. When using scissor clipping, the value of c_2 should be t_{line}, as the number of lines within the bounding box is equal to the number of times the per-scanline loop is executed. This is obviously true if the part of the triangle is within the region. If the triangle is totally outside the region, the rasterization algorithm will still step over all the lines within



(a) 50% of bounding box (b) 0% of bounding box (c) 100% of bounding box

Figure 6.7 Cases where 50, 0, and 100% of the pixels in bounding box are rendered. The thick lines show the region boundaries, the gray shows the triangle in question, the dashed thin lines show the bounding box of the triangle, and the thin diagonal lines and box show the bounding box of the triangle that is within the region (the "region-clipped bounding box").

the clipped bounding box. If another clipping method is used, for example if the triangle is clipped to the bounding box before rasterization, then c_2 should be smaller than t_{line}.

Finding the value for c_3 is more difficult. For a triangle entirely within the region, between 0 and 50% of the pixels within the bounding box are rasterized (see figure 6.7a). If the triangle is entirely outside the region, none of the pixels inside the bounding box are rasterized (figure 6.7b). Finally, if the triangle totally overlaps the region, all the pixels inside the bounding box are rasterized (figure 6.7c). The relative probability of the cases depends on the size of the regions and the size of the triangles. If the triangles are small compared to the regions, the first two cases will be the most common. If the triangles are large compared to the regions, the last case will predominate.

Least-squares fitting of the triangle size to the bounding box size was not successful, because the spread in the values was too large (see figures 6.8 and 6.9). I set c_3 to $t_{pixel}/2$, which corresponds to the case where half of the bounding box's pixels are rasterized. This is a reasonable value for large polygons, and is an overestimate for small polygons. It is more important to have a better estimate for larger triangles because the per-pixel time dominates the total rasterization time.

Figures 6.8 and 6.9 show a scatter plot showing the number of pixels within the region-clipped bounding box plotted against the actual number of pixels inside both the triangle and the region. The plots also show a line indicating the 0.5 estimate. Figure 6.8 has triangles sampled from a series of views of the Terrain model (see Appendix A). The average triangle area was 2.8, compared to the average region area of 325 pixels. Figure 6.9 has triangles sampled from views of the Well model, with an average triangle area of 2210 pixels.

I will evaluate the performance of the one-per-frame load-balancing algorithms using both the first and third types of cost functions, respectively called *triangle-count* and *size-based* cost estimates. When using triangle counts, the simulator uses the time needed to receive and rasterize a 20-pixel triangle that overlaps 10 scanlines. The size-based cost estimates are based on the simulator's cost function shown earlier in table 6.1.

Other cost functions could be investigated. A better size-based function would include the size of the region in the function. This would allow the function to include a large fraction of the pixels inside the bounding box to be also inside the triangle when the bounding box is large compared to the region (as in figure 6.7c), and to consider a smaller fraction of the pixels to be inside the triangle when the bounding box is small compared to the region (as in figure 6.7a).

6.2.3.3.2 Number of Processors Sampled for Region Cost Estimation

The second cost estimate choice is the number of processors that participate in the gathering of the per-region cost estimates. Limiting the number of processors will reduce the time to gather the estimates. This may increase the performance when using hundreds of processors, since the time required to gather cost estimates may be high. However, limiting the number of processors will also reduce the correlation between the estimates and the actual rasterization time, thus reducing the processor utilization. When an immediate mode interface is used, I expect that using a subset of processors will often give fairly poor estimates since each processor often has triangles that are close together in world space and thus also are

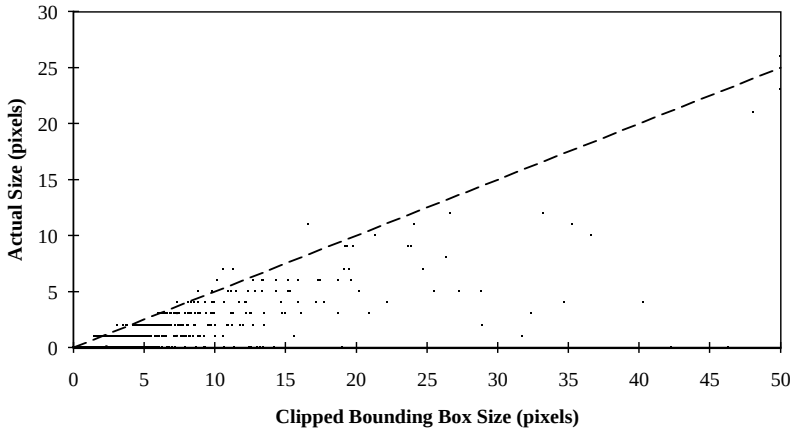


Figure 6.8 Comparison of the predictor of the number of pixels within the triangle against the actual number of pixels in the triangle, for triangles sampled from the Terrain model. The horizontal axis is the number of pixels within both the triangle's bounding box and the region, and the vertical axis is the number of pixels within both the triangle and the region (see figure 6.7). The dashed line indicates 0.5 actual pixels for every pixel in the bounding box.

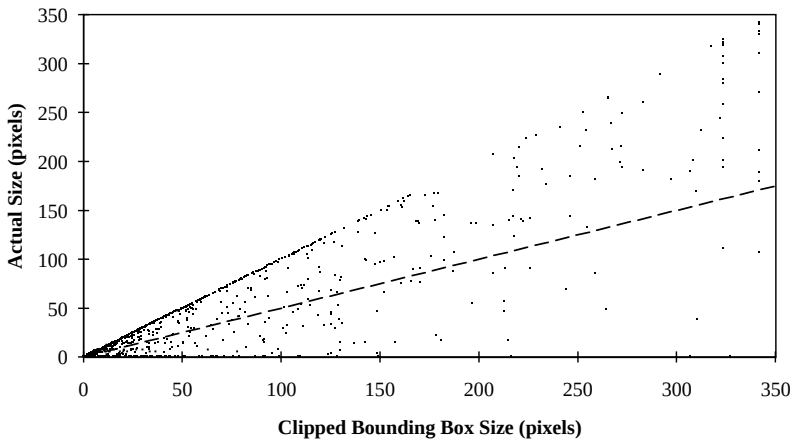


Figure 6.9 Comparison of the predictor of the number of pixels within the triangle against the actual number of pixels in the triangle, for triangles sampled from the Well model. The axes are the same as for figure 6.8. The dashed line indicates 0.5 actual pixels for every pixel in the bounding box.

close together in screen space. Using only some of the processors' per-region costs would not be representative of the overall costs. When using a retained-mode interface, each processor can be given triangles that are far apart in world space using the algorithms described in Chapter 5, which means sampling the cost estimates from a few processors should give reasonable results.

I did not make an exhaustive simulation of load-balancing methods that use sampled cost estimates because I investigated them after I completed the other investigations. These few results are given in section 6.5, which compares the different region assignment methods. In that section, I give results where cost estimates are gathered from at most 32 processors. I chose this value because load-balancing becomes a bottleneck when using 64 or more processors and models having less than about 30,000 triangles. Also, using at most 32 processors means that at least one-eighth of the processors contribute cost estimates (assuming a maximum of 512 processors).

6.2.4 Load-Balancing Methods for Work-Queue Assignments

I consider three load-balancing algorithms suitable for assigning regions between the rasterization of each region, where the regions are assigned using a work-queue approach. The algorithms use the same three screen-partitioning methods used by the once-per-frame load-balancing methods (sections 6.2.3.1-2). The per-region cost estimates play a greatly reduced role when making work-queue assignments compared to once-per-frame assignments. They are used to determine the order of assigning the regions so that they can be assigned from the most time consuming to the least. The estimates are not used to determine the actual assignments; instead, the next region is assigned to a processor when it finishes an earlier one. Since the estimates are not as important, I will only consider using triangle counts as the estimates, the cheapest estimate to compute.

The work-queue assignment style potentially gives higher processor utilization than the once-per-frame assignment styles because it does not rely on estimates of the time to assign regions. The utilization is higher for two reasons. First, the once-per-frame cost estimates are intentionally somewhat inaccurate in order to reduce the time needed to calculate them. An accurate model would require counting the pixels in each triangle, which is about as expensive as the transformation process itself. Second, even the most accurate estimates of the time to rasterize each primitive would not be enough to calculate the total frame time. A cycle-by-cycle simulation of the hardware is the only way to calculate the true frame time. Even actual system measurements will show run-to-run variations.

The current simulator can only duplicate the first advantage that the work-queue method has over the once-per-frame method. It assigns regions to processors by first computing the per-region times using a detailed performance model, and then uses the greedy bin-packing region-assignment method. The processor utilization is then computed using the same performance model, giving the effect of making the assignments based on the actual rasterization times.

6.3 A Tour Through the Simulation and Performance Model Results

This section explores the results of the simulator and the performance model. The goal is to give the reader an intuitive feel for the results, to see what happens when different parameters are varied. First, we will explore the frame-by-frame simulation output, seeing how values such as processor utilization, overlap factor, and percentage of triangles on screen vary from frame to frame, and as the number of processors and granularity ratio changes. Next, values averaged over all of the frames are shown. These averaged values are used later in the chapter and are the ones reported in Appendix C. Section 6.3.2 describes the output of the performance model, including how the predicted times are normalized for easier comparison.

Later sections explore different algorithm options. The first section describes the effect of using size-based cost estimates instead of ones based on triangle counts. The next section shows the differences between the partitioning methods, comparing the adaptive methods with the fixed-region method. The following section compares the processor utilization for load-balancing methods for the between-stages and between-frames assignment styles. The last section, section 6.3.6, shows the different components of the performance model including the load-balancing costs.

6.3.1 Typical Simulation Results

The simulator reports the minimum, maximum, and average times taken by the processors to rasterize each frame in the sequence. The average time represents the amount of work required to rasterize the frame, and the maximum time is the actual time to render the frame. Dividing the average time by the maximum time gives the average processor utilization. The difference between the minimum and maximum times gives a little information about the distribution of processing times. The simulator also calculates the per-frame overlap factor and, optionally, statistics about the on-screen triangles.

Figure 6.10 gives the minimum, maximum, and average times for the series of frames recorded for the Terrain model, using between-stages assignment of fixed regions using triangle counts, 128 processors, and a granularity ratio of four (some representative frames are shown in Appendix A). The times are normalized by dividing them by the maximum frame time of 0.187 seconds. The processor utilization, overlap factor, and the fraction of the model displayed are also plotted. The average rasterization time varies from frame to frame since the number and size of the triangles changes. The number of triangles varies because the triangles can go off screen and can be back-face culled (although back-face culling is not used for this model). The maximum time changes because the amount of work changes, and because the distribution of

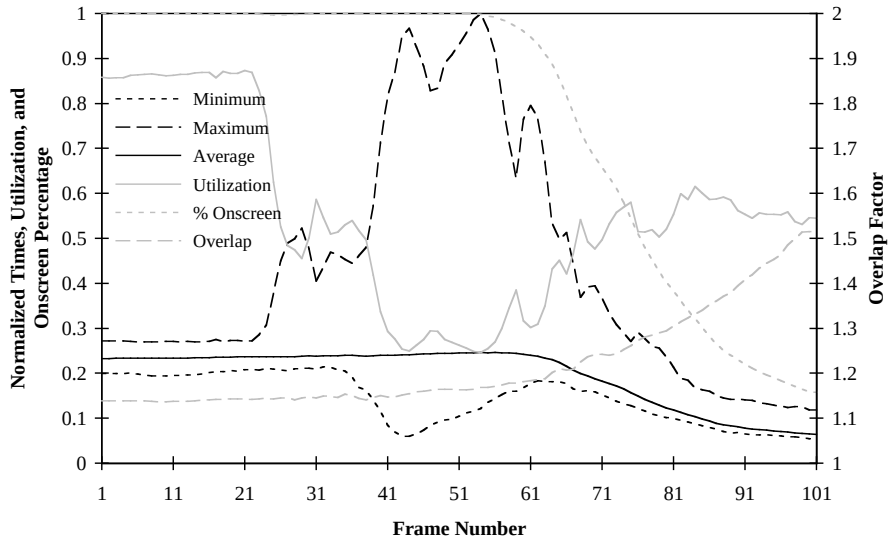


Figure 6.10 Per-frame values from a simulation run using the Terrain model with N = 128, G = 4, and between-stages assignment of fixed regions using triangle counts. Different curves plot the minimum, average, and maximum processor frame times; the processor utilization; the percentage of triangles that are on screen; and the overlap factor. The frame times are normalized by dividing them by the maximum frame time.

triangles on the screen changes. In frames 23 to 101, the maximum time results from a bottleneck region, the region with the largest amount of work.

The next two charts show the result when, first, the granularity ratio and, second, the number of processors is varied. They only show the processor utilization for the range of frames. The next two charts and the previous chart have one curve in common, which shows the processor utilization for N = 128 and G = 4. The curve is shaded similarly in the three charts.

Figure 6.11 shows the processor utilization as the granularity ratio is varied from 1 to 16. With a granularity ratio of 1, the load-balancing algorithm turns into static load-balancing with each processor working on one equal-sized region. The processor utilization improves with larger granularity ratios. For frames 1-23 and 67-101, the utilization does not improve very much when the granularity ratio is increased beyond eight. For frames 24-66, the increased G improves the processor utilization. This effect occurs because the model covers at least half of the screen in frames 1-23 and 67-101, while it only covers a smaller fraction in frames 24-66 (see figure A.2 in Appendix A). The increased granularity ratio is needed to break up the screen finely enough so that the most complex region is not the rasterization bottleneck.

Figure 6.12 shows the results as the number of processors is varied from 2 to 512 while holding G at 4. When only some of the screen is filled (around frame 50), the larger numbers of processors perform more poorly than the smaller numbers. This happens because the variance in the amount of work in each region increases as the number of regions increases. When the number of regions is halved by combining regions, the variance between regions will usually decrease because some complex and simple regions will be combined to make regions of average complexity.

When the screen is filled evenly, all numbers of processors have high levels of utilization. This is different from the earlier figure where the granularity ratio is varied, where there seems to be a minimum value that works well. This is most likely caused by the greedy load-balancing algorithm, which assigns the regions from largest to the most lightly loaded processor at each step. The screen must be divided into enough pieces so that there are enough smaller regions at the end to even out the loads once the large regions have been assigned.

All the different combinations of variables were simulated. The number of processors was varied from 2 to 512 in powers of two. The static assignment methods were simulated using granularity ratios of 1, 2, 4, 6, 8, 12, 16 and 32. The three per-frame assignment methods were simulated using granularity ratios of 1, 2,

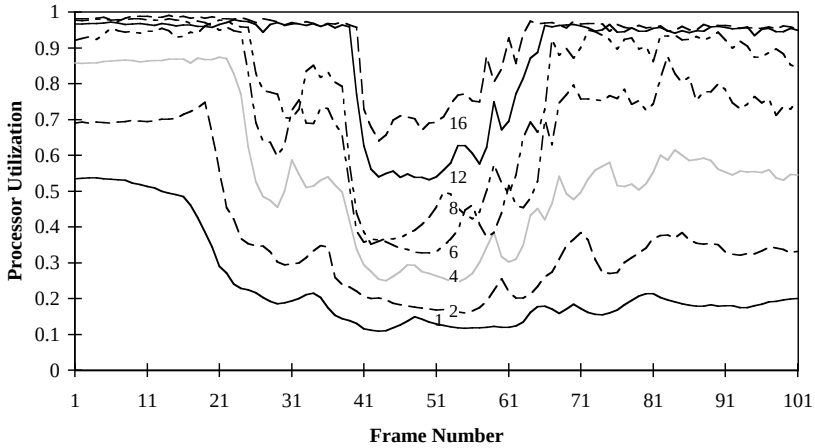


Figure 6.11 Simulated processor utilization for a series of frames and a range of granularity ratios, all using the Terrain model, N = 128, and between-stages assignment of fixed regions using triangle counts.

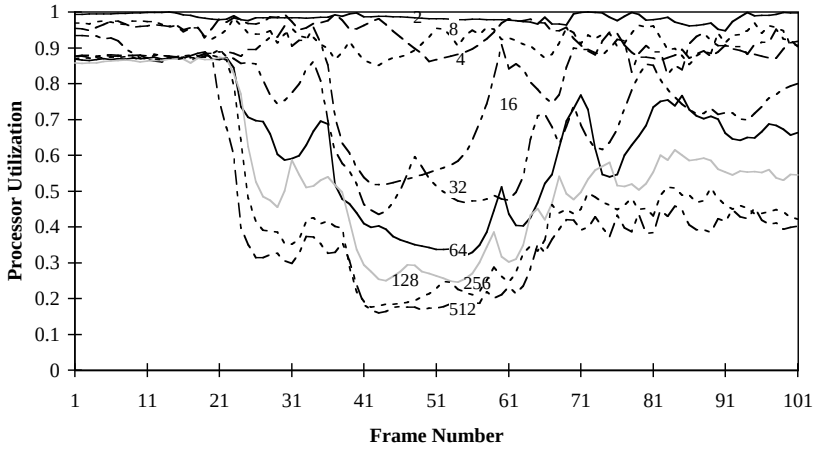


Figure 6.12 Simulated processor utilization for a series of frames and a range of number of processors, all using the Terrain model, G = 4, and current frame assignment of fixed regions.

4, 6, 8, 12, and 16, for both between-frames and between-stages region assignments, and for both triangle counts and size-based cost estimates. The three work-queue load-balancing methods were also simulated with G = 1, 2, 4, 6, 8, 12, and 16. Each combination was simulated for the PLB head, Terrain, CT head, and Polio models pictured in Appendix A, for a total of 4,356 runs. Charts of the data appear in Appendix C.

The charts in Appendix C and later charts in this chapter do not show the utilization for individual frames. Instead, the utilization is averaged across all the frames, collapsing each of the curves in figures 6.11 and 6.12 to a single point. This equally weights the processor utilization of each frame for the later comparisons. The overlap factor and the number of triangles will also be averaged across the frames when used in the performance model. Figure 6.13 shows a representative chart with this averaged utilization data, for the Terrain model using between-stages assignment of fixed regions using triangle counts.

The simulator also computes a per-frame overlap factor. This value does not have as much frame-to-frame variation as the processor utilization (see figure 6.10) so there is no need to show the per-frame values. Figure 6.14 shows the average overlap factor values for the Terrain model for a range of partition sizes and granularity ratios. The runs used the fixed region boundaries, and between-stages-assigned regions using triangle counts. Since the overlap factor increases with the number of regions, increasing either the granularity ratio or the number of processors increases the overlap ratio. Because the overlap factor depends on the model, each of the four models has a corresponding series of charts (see Appendix C).

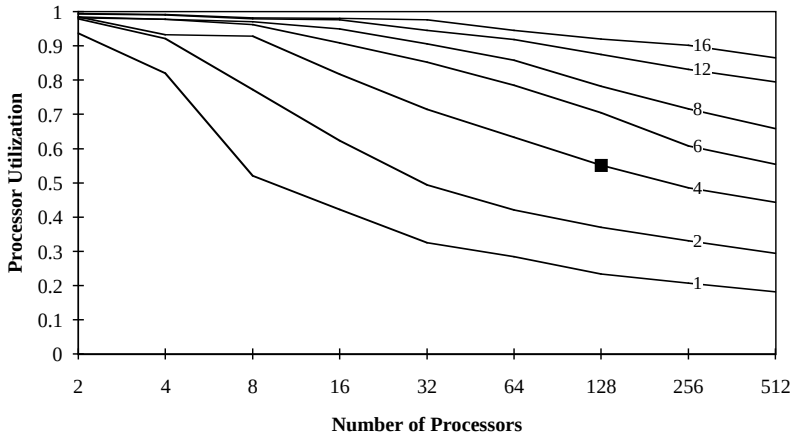


Figure 6.13 Average processor utilization expected for the Terrain model, for between-stages assignment of fixed regions using triangle counts, and for a range of granularity ratios and numbers of processors. Each curve shows the utilization for the marked granularity ratio. The point corresponding to the utilization curve in figure 6.10 is marked with a square.

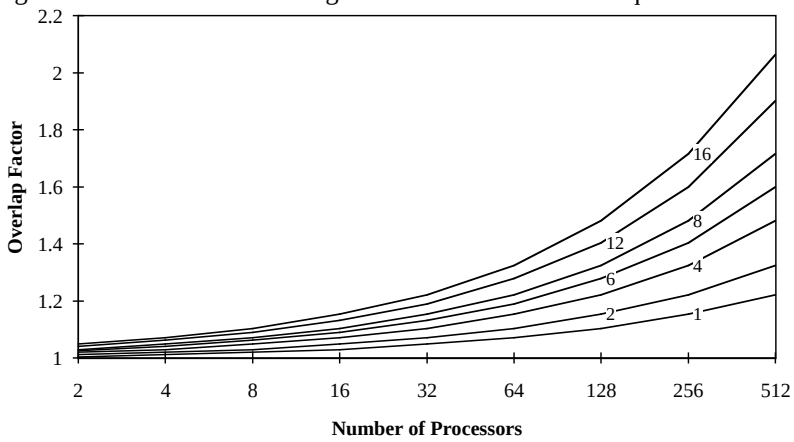


Figure 6.14 Average overlap factor for the Terrain model, for between-stages assignment of fixed regions using triangle counts, and for a range of granularity ratios and numbers of processors. Each curve shows the utilization for the marked granularity ratio.

6.3.2 Typical Performance Model Results

This section shows some results from the performance model, and describes how the values are normalized to show the performance differences between the different methods. The section describes the results of the performance model that exclude load-balancing costs. Because the load-balancing costs depend on the load-balancing style chosen, and because only a single representative example of the different results is shown, load-balancing costs are excluded from the timing values shown in this section through section 6.3.5. Including load-balancing costs would make the examples apply even more to a single assignment style.

One example of the speeds predicted by the performance model is shown in figure 6.15. The values in this figure are for the same case as those in figure 6.12: Terrain model, between-stages assignment using fixed regions, and triangle counts. Note that the best time predicted for 512 processors is nearly ten times slower than linear speedup. The causes of the less-than-linear speedup will be explored in Chapter 7.

The remaining comparisons will use normalized, or relative, times to emphasize the difference between the different methods, and to remove the bias introduced by varying the number of processors and the size of the model. The normalization is done by dividing each predicted time by the average of all the times with the same number of processors and the same model. Figure 6.16 shows this relative time for the same values as shown in figure 6.15. Similar charts for all the cases are in section C.2 of Appendix C. The load-

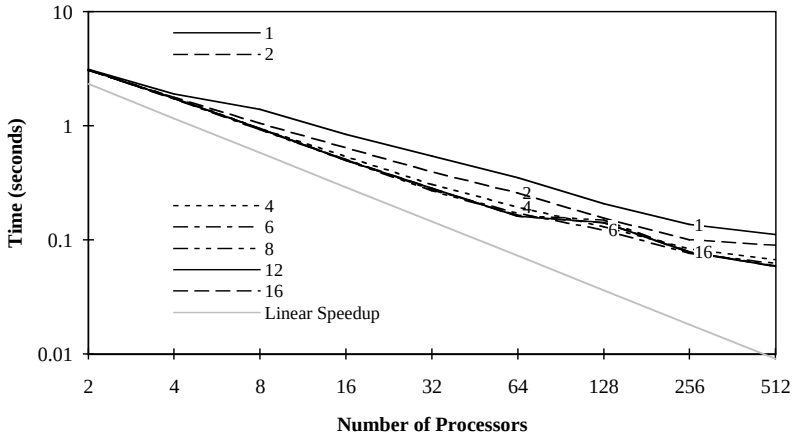


Figure 6.15 Time predicted by the performance model for the Terrain data set, for between-stages assignment of fixed regions using triangle counts for a range of granularity sizes. The gray line indicates linear speedup based on the algorithm running on a single processor.

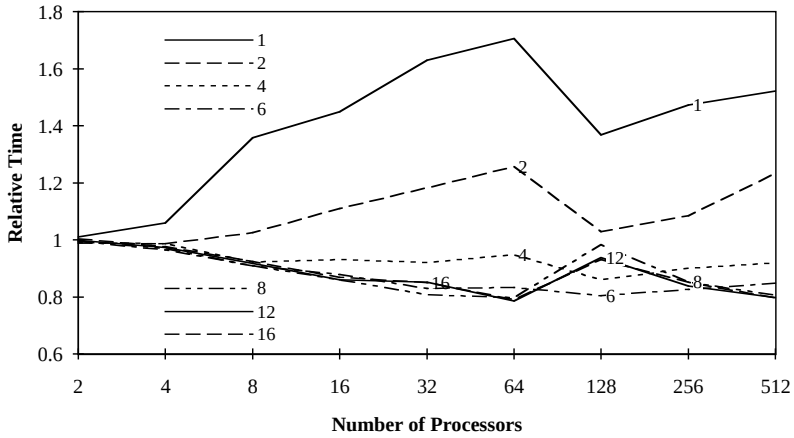


Figure 6.16 Relative times from the performance model for the Terrain data set, for between-stages assignment of fixed regions using the triangle counts for a range of granularity sizes.

balancing methods for the work-queue assignment style are normalized separately since they have very different results.

Many of the performance charts, including figure 6.16, have peculiar kinks in the curves near 64, 128, or 256 processors. The kinks are caused by load-balancing methods with different granularity ratios switching between one-step and two-step redistribution at different partition sizes. For example, in figure 6.16 two-step redistribution is done with at least 128 processors for granularity ratios from 1 to 6, and with at least 256 processors for ratios from 8 to 16. The redistribution switch occurs with different number of processors because higher granularity ratios cause higher overlap factors, which in turn increases the number of triangles to be redistributed. Having more triangles means that one-step redistribution is more efficient with more processors, as was shown in figures 4.3–4.5 in section 4.4.4.

A second cause of the kinks is that the per-processor redistribution time may be higher just before or after switching from one-step to two-step redistribution. This was seen in figure 4.6, where the per-processor cost with stream-per-processor redistribution with approximately 350 processors is higher than when using 256 or 512 processors (the figure only has data for numbers of processors that are powers of two, but this effect is also seen in charts with more data points). The rise in redistribution time can also be seen in figure 6.15 for the curves with granularity ratios of 8, 12, and 16. The differing switching times, the rise in redistribution time, and the normalization of the processing times combine to put kinks in all the curves in figure 6.16. The curves for granularity ratios of 1 and 2 have very noticeable kinks because they are more

Model	Number of Processors								
	2	4	8	16	32	64	128	256	512
PLB Head	0.20	0.15	0.27	0.78	1.51	1.61	1.90	2.63	3.36
Terrain	0.05	0.04	0.02	-0.03	-0.08	-0.31	-0.04	0.11	0.25
CT Head	0.14	0.21	0.36	0.73	1.19	1.78	1.67	1.92	2.33
Polio	0.21	0.62	0.91	1.20	1.73	1.97	1.88	1.52	1.35

Table 6.3 Percentage of average difference of expected performance between methods using triangle counts and size-based cost estimates. The maximum difference is only 3.4%.

affected by the rise in redistribution time as compared to the curves for ratios of 4 and 6. Also, since they are farthest away from the average time, the normalization process affects them the most.

The kinks are much less prominent when two-step redistribution starts with the same number of processors for all the granularity ratios. This occurs with the PLB Head model. An example of the smaller kinks is shown in figure C.18a in Appendix C, which shows the corresponding results for the PLB Head model for the cases shown in figure 6.16 for the Terrain model.

6.3.3 Size-Based Cost Estimates

The increase in performance given by using the better size-based cost estimates is surprisingly small. Table 6.3 shows the average increase in relative performance for each model and number of processors combination. The largest average increase is only 3.4%. The amount of improvement increases with the number of processors. Increasing the number of processors increases the number of regions, and thus decreases the average number of triangles per region. Errors in estimating the time to rasterize a triangle will have a larger effect when there are fewer triangles in a region, thus increasing the benefit of size-based timing estimates.

These figures do not include the additional time to compute the sized-based cost estimates. I estimate that the bucketization time t_{bucket} will be increased by 1 μs, which translates into a 0.2% increase in the overall time when using 512 processors, and a 2.6% increase when using 2 processors The additional computation cost outweighs the time savings when using less than about 64 processors.

I believe that the main reason for the small increase in performance when using the better cost estimates is that the chosen models have evenly sized triangles that, for most viewpoints, only cover a few pixels. The model with the largest average triangle size is the Polio model, which has an average of 6.7 pixels per triangle and an average of 2.5 scan-lines crossed per triangle. With triangles of that size, the per-triangle portion of the rasterization cost t_{tri} is 76% of the overall rasterization cost (this includes the time to receive the triangle). The low performance increase could also be caused by errors caused by the function that turns the number of pixels in a triangle's bounding box into an estimate of the number of pixels in a triangle; the discussion in section 6.2 showed that the size of the bounding box and the triangle did not have a high correlation. The use of size-based timing estimates should result in a larger performance increase when using larger triangles.

6.3.4 Adaptive Region Boundary Methods

The adaptive methods were, on average, slightly slower than the fixed region methods. For granularity ratios between 1 and 4, the adaptive methods were better than the fixed region methods. They had both higher processor utilization and overlap factor values. The change in the overall frame time due to the increased processor utilization outweighed the effect due to the higher overlap factor. The method decreases the region size around the screen's hot spots, which reduces the likelihood that the region is a bottle neck, but also puts the region boundaries near many triangles, increasing the overlap factor. The overall performance advantage does not exist with larger values of G. With G = 6 or 8, the utilization values were approximately equal, which results in slower performance when the higher overlap factor is considered.

Figures 6.17–19 illustrate these effects for the Terrain model using between-stages assignments using triangle-counts. The first figure plots the relative processor utilization for the adaptive method, that is, the utilization when using adaptive region boundaries divided by the utilization when using fixed boundaries. The top chart gives the values for the per-frame adaptive region method, and the bottom chart gives the

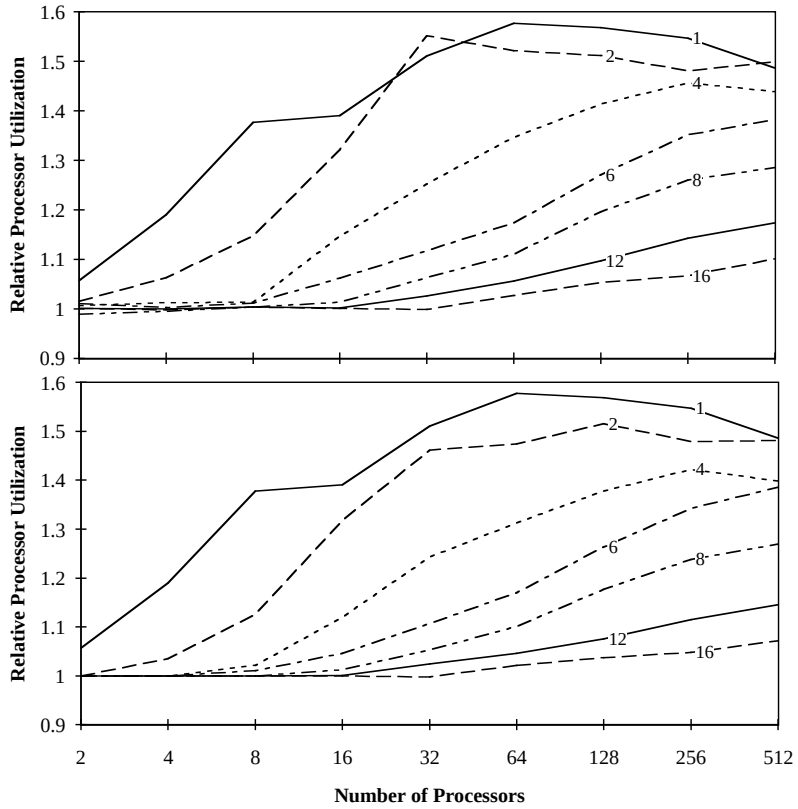


Figure 6.17 Simulated processor utilization for the per-frame adaptive-region (top) and when-needed adaptive-region (bottom) load-balancing methods relative to the fixed-region method. The values are for the Terrain model for between-stages assignment using triangle counts. Each curve shows the results for the indicated granularity ratio.

values for the when-needed method. Note that the when-needed method sometimes never adjusts the boundaries, making the performance exactly the same as the fixed region method. This happens with large values of *G* and few processors.

Figure 6.18 shows the relative overlap factor values. The adaptive methods always have higher overlap factors than the fixed-region method except when the when-needed method never adjusts the boundaries. The relative frame times are shown in figure 6.19. The curves with the lower granularity ratios are at the bottom of the chart, which indicates that the adaptive region boundary method is as much as 30% faster in those cases.

6.3.5 Load-Balancing Differences Between the Between-Stages and Between-Frames Assignment Styles

The last characterization shows how the processor utilization differs between the between-stages and between-frames assignment styles. The between-frames style will have lower processor utilization because it uses region assignments made during the previous frame using that frame's timing estimates. The overlap factor is not affected. The between-stages assignment style forces the load-balancing calculations to be done in the middle of the frame, while the processors wait, while using the between-frames style allows the calculations to be done during the rasterization.

Figure 6.20 shows a typical example of relative processor utilization, which is the between-stages utilization divided by the between-frames utilization. The chart shows the values for the Terrain model using triangle counts as the cost estimator. Because the performance model divides the raw rasterization time by the processor utilization to get the true frame time, a value of 1.1 in the chart means that rasterization using the between-frames style takes 1.1 times as long as rasterization with the between-stages style. Note that figure 6.13 shows one of the sets of curves that is being compared in this chart.

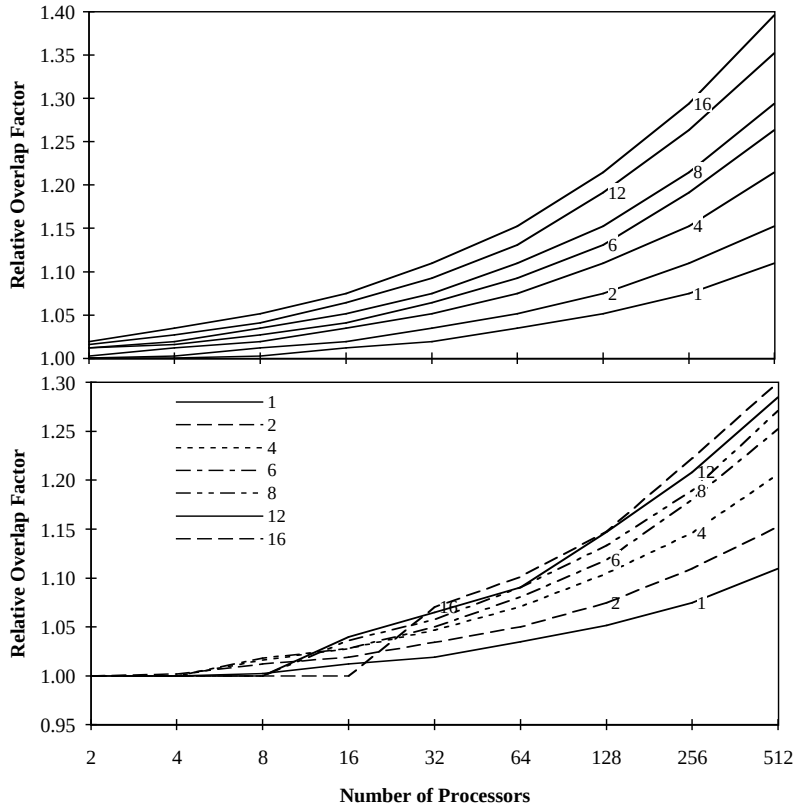


Figure 6.18 Simulated overlap factor values for the per-frame adaptive-region (top) and when-needed adaptive region (bottom) load-balancing methods relative to the fixed-region method. The values are for the Terrain model between-stages assignment using triangle counts. Each curve shows the results for the indicated granularity ratio.

The difference in processor utilization increases when either the number of processors or the granularity ratio increases. This is not surprising, since the product of the two numbers is the number of regions. Increasing the number of regions increases the probability that a clump of triangles changes regions.

The relative processor utilization between using the previous and current frame's cost estimates is summarized in table 6.4. This has the maximum relative processor utilization for each model and number of processors. Each entry in table is a maximum of 42 values (7 granularity ratios for each of 6 load-balancing methods).

The table shows that the difference in utilization depends on the model and its associated series of views. It is difficult to know whether the simulated models and viewpoints are representative of actual scientific visualization applications. One source of bias is that I recorded the frames for three models, and may have constrained the change in viewpoint. The viewpoints of the PLB Head were taken from the associated benchmark. Since that benchmark merely has the head rotate around its vertical axis, the amount of coherence in the distribution of triangles across the screen is probably fairly high.

The change in utilization is fairly low for the smaller granularity ratios. The difference is quite small for small number of processors, and is not unreasonable for 512 processors. The impact of the difference in utilization will be quantified by examining the difference in performance predicted by the performance model versus the difference in load-balancing costs. This will be done later, in section 6.5, when the different region assignment styles are compared.

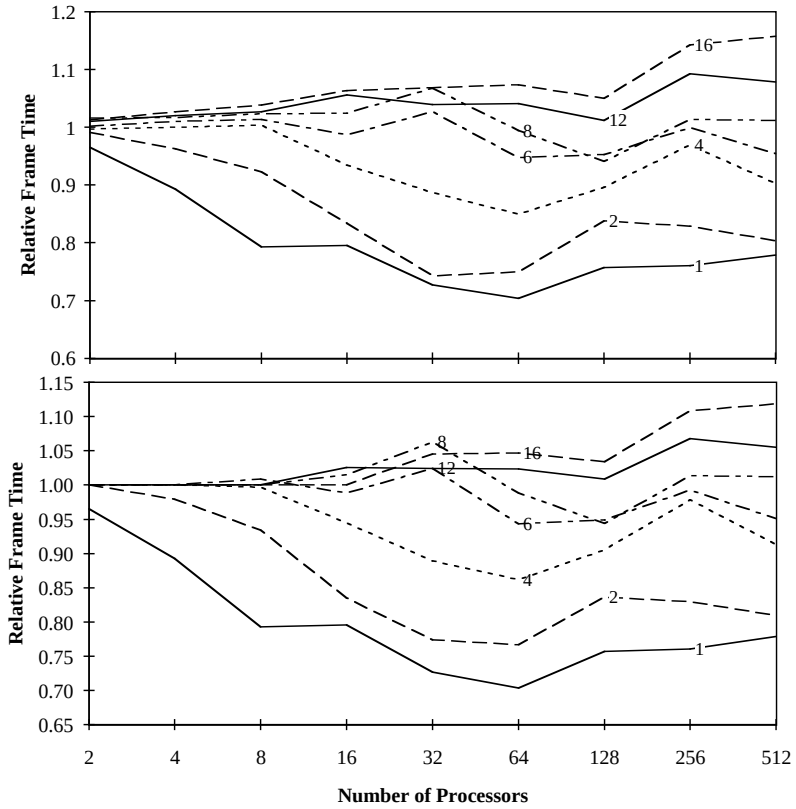


Figure 6.19 Predicted frame time for the per-frame adaptive-region (top) and when-needed adaptive-region (bottom) load-balancing methods relative to the fixed-region method. The values are for the Terrain model between-stages assignment using triangle counts. Each curve shows the results for the indicated granularity ratio.

6.3.6 Components of the Performance Model

A later section will show plots of the overall times for each method. While examining the overall times lets one choose the fastest method, it does not make it easy to understand the performance model's behavior. This section shows the components to allow the reader to better understand the behavior. The two charts in figure 6.21 show some of the components for the largest and smallest models, PLB Head and Polio. Each chart shows the predicted time, without load-balancing costs, for the chosen methods for each of the four region assignment styles. The curves for the between-frames and between-stages assignment styles are very close together in the PLB Head chart; they appear the same in the Polio chart. An additional solid black curve shows the portion of the frame time after the end of geometry processing, the time that the between-frames load-balancing method has to compute the next frame's region assignments.

Three other curves show the load-balancing costs; all are gray. Two curves show different methods' processing time, the time used per processor for load-balancing operations. One curve shows the time for the work-queue method and a second shows the time for the two once-per-frame methods. The third curve shows the latency required for the once-per-frame methods to make the region assignments. These times assume a granularity ratio of 6.

By comparing the curves one can get a feel for which load-balancing method gives the best performance. The between-frames method is faster than the between-stages method if the once-per-frame load-balancing latency (the same for both methods) is larger than the difference between the two method's processing times with load balancing excluded. This is only true when using few processors. It is harder to see the exact point when static load-balancing is faster than between-frames load-balancing, but the switch occurs just before the point where the load-balancing latency is larger than the time after geometry processing (AGP).

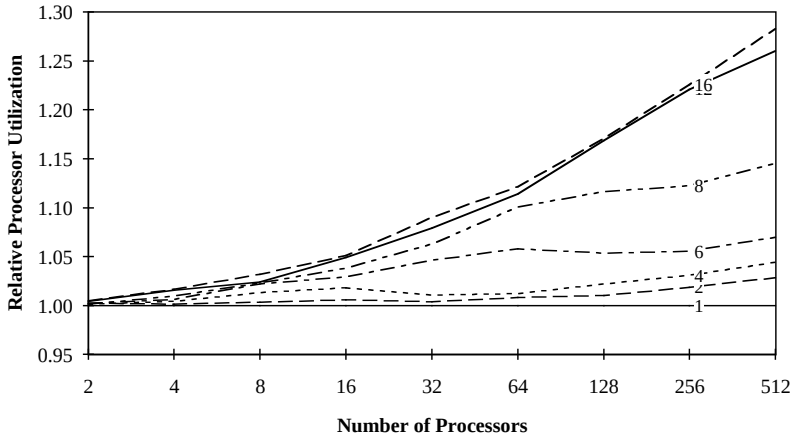


Figure 6.20 Relative simulated processor utilization for the Terrain model between the between-stages and between-frames assignment styles using triangle counts and fixed regions, for a range of granularity ratios and numbers of processors.

	Number of Processors									
Model	2	4	8	16	32	64	128	256	512	Max.
PLB Head	1.042	1.081	1.102	1.139	1.196	1.220	1.276	1.325	1.380	1.380
Terrain	1.025	1.060	1.088	1.096	1.131	1.179	1.248	1.303	1.360	1.360
CT Head	1.012	1.020	1.024	1.047	1.068	1.113	1.166	1.222	1.295	1.295
Polio	1.009	1.007	1.010	1.012	1.015	1.024	1.034	1.051	1.078	1.078

Table 6.4 Maximum relative processor utilization between corresponding load-balancing methods for the between-stages and between-frames assignment styles. All assignment methods and cost estimates are considered.

6.4 Best Load-Balancing Method for Each Assignment Style

This section describes the best load-balancing method for each of the five region assignment styles. The methods are used in section 6.5 to evaluate the different region assignment styles. The following subsections describe the best load-balancing method for one of the four assignment styles.

Each subsection shows the best load-balancing choices for the four different models and 9 processor partition sizes from 2 to 512. These choices show the extent of how the best choice depends on the application and system size. Each subsection also gives the best overall choice. The best choice is identified by computing the relative time of each of the 36 configurations shown for each load-balancing method; this is done by dividing a specific method's time by the average time over all the methods in the category. This normalization removes the influences of different model sizes and different number of processors from the predicted times. The best method has the smallest average relative time for the 36 configurations. A chart shows these values for each load-balancing method and granularity ratio.

Note that the load-balancing time is not included in these comparisons, even though the time is significant. The load-balancing time directly affects the frame time when the work-queue and between-stages region assignment styles are used. The load-balancing time also affects the frame time with the between-stages assignment style when using many processors. However, since we will select a single load-balancing method for each style across all the models and numbers of processors, including the load-balancing time could choose a method that is optimized for use over a large range of partition sizes when that region assignment style is only viable for a small range of partition sizes.

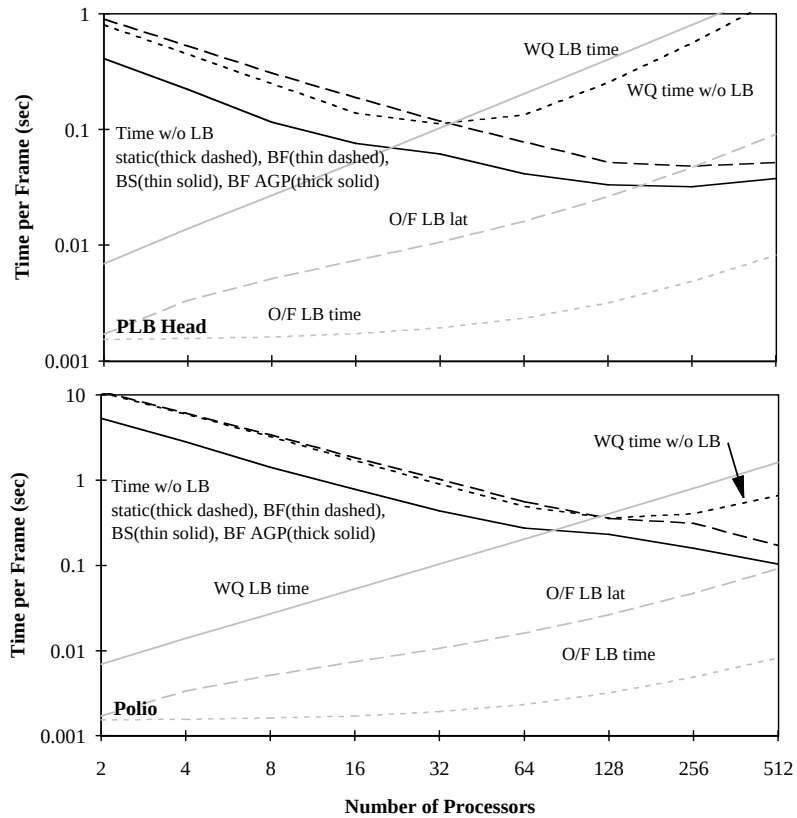


Figure 6.21 Performance model components for the PLB Head and Polio models.

Key:

Time w/o LB	Time without load-balancing for the static (thick black with long dashes), between-frames (BF, thin dashed), and between-stages (BS, thin solid black) assignment styles, plus the time after geometry processing for the between-frames method (thick solid black).
WQ time w/o LB	Time without load-balancing for the work-queue method (thick black with short dashes).
WQ LB time	Time per frame for work-queue load-balancing (solid gray).
O/F LB latency	Time needed to make once-per-frame region assignments (gray with short dashes).
O/F LB time	Processing time (per processor) for once-per-frame load-balancing (gray with long dashes).

6.4.1 Static Methods

The best static load-balancing method for various configurations is shown in table 6.5. While no single method is the best in all cases, the best compromise is the method using interleaved regions with 12 regions per processor. Figure 6.22 shows the average relative time for each load-balancing method and granularity ratio combination. The differences between the compromise choice and each case's best choice are shown in table 6.6. The interlaced load-balancing method gives more even performance (see the charts in Appendix C).

6.4.2 Once-Per-Frame Methods

The best load-balancing method for each of the two once-per-frame region assignment styles, for each model, and for a range of numbers of processors is shown in table 6.7. Each entry encodes the assignment method, type of cost estimate, and granularity ratio. Some entries show that more than one method was the

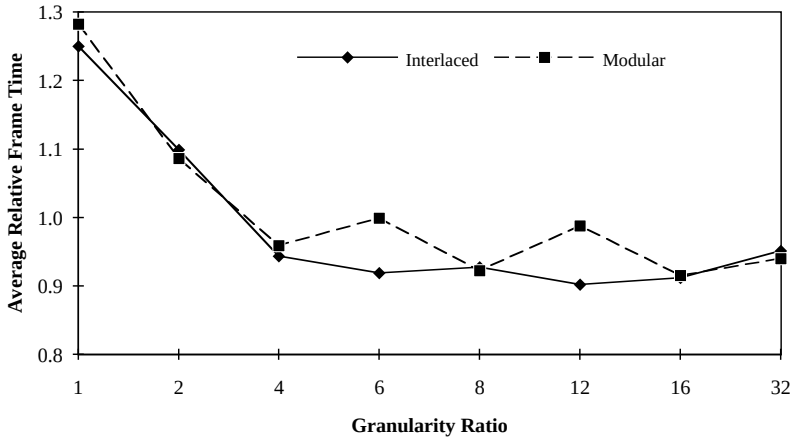


Figure 6.22 Average relative predicted frame times when using static region assignments. Each curve corresponds to the marked load-balancing method.

	Number of Processors								
Model	2	4	8	16	32	64	128	256	512
PLB Head	IM 1	I 1	M 4	M 32	M 32	M 16	I 12	I 32	M 32
Terrain	M 2	M 8	M 6	M 32	M 6	M 12	M 6	M 6	M 16
CT Head	M 8	M 8	M 16	M 32	I 12	I 12	I 8	I 4	I 6
Polio	IM 12	M 6	I 6	I 12	I 6	I 12	I 6	I 4	I 4

Table 6.5 Best load-balancing method for different models and number of processors between the modular and interleaved static load-balancing method. Key: First one or two letters specify load-balancing method: M = modular static, I = interleaved static. Two letters indicate the two methods have the same estimated time. The following number specifies the granularity ratio.

	Number of Processors								
Model	2	4	8	16	32	64	128	256	512
PLB Head	14.3	11.8	7.3	15.1	9.6	4.8	-	8.6	10.4
Terrain	1.7	6.4	4.4	11.1	13.4	13.4	35	9.6	9.8
CT Head	1.0	0.4	0.5	0.6	-	-	3.6	1.7	1.8
Polio	-	1.9	1.9	-	2.8	-	1.9	34.4	16.4

Table 6.6 Percentage difference between the best static load-balancing method (I 12) and the best ones for each model and partition size as shown in table 6.5. A dash indicates that the I 12 method is the best one for that configuration.

best. In many cases, several methods had times within 5% of the best method. Figures 6.23 and 6.24 show the relative average frame time for each load-balancing method and granularity ratio.

No one method or granularity ratio appears to be the best. It is not surprising that different models would need different granularity ratios, since different models can have different amounts of triangle concentrations on the screen.

The method corresponding to Ns4 (when-needed-adjusted adaptive regions, size-based estimates, granularity ratio of 4) gives the best average performance for the between-frames assignment style, and the Ns6 (when-needed-adjusted adaptive regions, size-based estimates, granularity ratio of 6) gives the best performance for the between-stages style. Each assignment method is the best for a number of cases, and the granularity ratio is in the middle of the all the ones in the table. However, this choice is not perfect: in a few cases the method gives non-negligibly less performance. Table 6.8 shows how much longer the Fs6 and Ns6 methods take over the best method for each configuration shown in table 6.7.

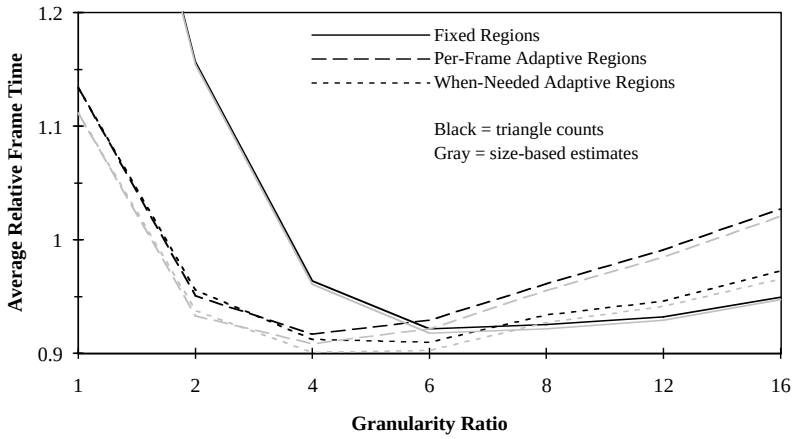


Figure 6.23 Average relative predicted frame times when using the between-frames region assignment style. Each curve corresponds to the marked load-balancing method. The black curves show the methods that use triangle counts, and the gray curves show the ones using size-based cost estimates. The value for both fixed-regions methods with *G*=1 is 1.56.

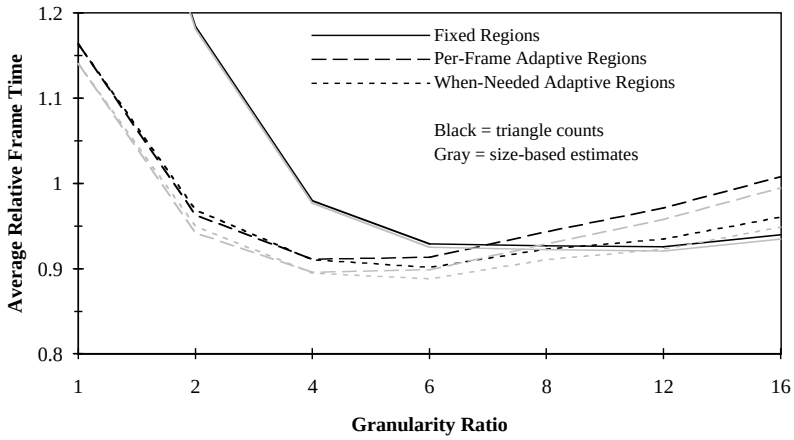


Figure 6.24 Average relative predicted frame times when using the between-stages region assignment style. Each curve corresponds to the marked load-balancing method. The black curves show the methods that use triangle counts, and the gray curves show the ones using size-based cost estimates. The value for both fixed-regions methods with *G*=1 is 1.6.

	Number of Processors								
Model	2	4	8	16	32	64	128	256	512
PLB Head BS	Fs2	FNs8	Fs6	Ns4	As4	Ns4	As6	Ns8	Ns8
PLB Head BF	Fst1	FNs8	Fs6	Ns4	Ft16	Ns4	Ns4	Fs16	Fs16
Terrain BS	ANs1	ANs1	FNs8	Fs8	Fs8	Ns6	Ns6	Ns6	Fs12
Terrain BF	ANs1	ANs1	Fs6	Fs8	Fs8	Ft12	At4	Fs6	Fs8
CT Head BS	As2	As2	FNs6	As2	Fs6	As4	Fs8	Fs6	Fs8
CT Head BF	As2	ANs1	FNs8	As2	Fs6	FNs8	Fs8	Fs6	Fs6
Polio BS	As2	As2	ANs1	As2	As2	As2	As2	As2	As2
Polio BF	As2	ANs1	ANs1	As2	As2	As2	As2	As2	As2

Table 6.7 Best load-balancing method for different models, number of processors, and between-stages (BS) and between-frames (BF) assignment styles. Key: The first one or two upper-case letters specify the load-balancing method: F = fixed region, A = per-frame adaptive regions, N = when-needed adaptive regions. Two letters indicate that the two methods have exactly the same estimated time. The next one or two lower-case letters specify the cost estimate: t = triangle-count based, s = size based. The last value is the granularity ratio.

Model	Number of Processors								
	2	4	8	16	32	64	128	256	512
PLB Head BS	0.7	1.4	1.5	1.3	0.5	1.8	0.1	0.2	3.6
PLB Head BF	0.2	1.5	0.8	-	0.9	-	-	3.3	6.3
Terrain BS	2.4	2.2	0.9	0.7	4.8	-	-	-	0.0
Terrain BF	1.8	4.9	2.5	4.6	3.8	3.6	1.7	**9.4**	2.2
CT Head BS	0.3	0.9	-	0.8	2.1	2.4	**6.0**	3.3	4.2
CT Head BF	0.1	1.3	0.8	2.3	1.7	0.9	1.7	2.4	4.7
Polio BS	1.0	1.2	1.9	0.5	1.2	2.4	4.4	**10.1**	**17.5**
Polio BF	0.2	3.5	2.0	2.0	1.8	1.6	3.2	**9.7**	**18.3**

Table 6.8 For both between-stages (BS) and between-frames (BF) assignment styles, percentage difference in estimated time between the best load-balancing method shown in table 6.7 and fixed region load-balancing with size-based cost estimates and $G = 6$. A dash indicates that the method is the optimal one. Values larger than 5% are in bold.

6.4.3 Load-Balancing for Work-Queue Assignments

The load-balancing methods for work-queue assignments performed well with a few processors but poorly when using many processors. With up to about 16 processors, the overall performance was comparable to or faster than the once-per-frame assignment methods. With more processors, the performance was quite low because one-step stream-per-region redistribution must be used when making the work-queue region assignments. The lower granularity ratios worked best when using many processors because they reduced the number of regions, and thus the number of messages required. As seen earlier, the adaptive methods worked better than the fixed region method with the smaller granularity ratios, which made a difference when using many processors. Figure 6.25 gives an example of the predicted frame times for the Terrain model using the fixed region boundaries. The corresponding relative predicted times are shown in figure C.22a in Appendix C.

Table 6.9 shows the best load-balancing method for different models and partition sizes. In many cases the when-needed adaptive method either rarely adjusted the regions, when G was large, or adjusted the regions every frame, when G was low. The result is that the when-needed method often gives equivalent performance to one of the other two methods.

The assignment methods shown are faster than the best once-per-frame cases when using 2 processors, and when using 4 processors with the CT Head and Terrain models. The best overall method is the when-needed per-frame-adaptive method using $G = 2$. Since these methods are only faster than the once-per-

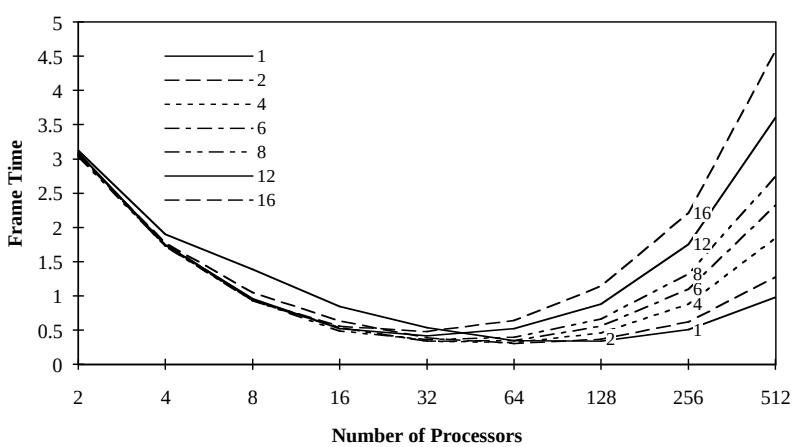


Figure 6.25 Typical predicted frame times when using work-queue assignment. Each curve corresponds to the marked granularity ratio. The times are for the Terrain model using fixed region assignments.

Model	Number of Processors								
	2	4	8	16	32	64	128	256	512
PLB Head	F 2	A 2	N 4	N 2	AN 2	AN 1	AN 1	AN 1	AN 1
Terrain	AN 1	AN 1	F 6	F 8	A 2	A 2	AN 1	AN 1	AN 1
CT Head	FN 8	A 2	FN 6	FN 8	AN 2	AN 2	AN 1	AN 1	AN 1
Polio	A 2	FN 6	FN 6	FN 6	FN 6	N 2	A 2	AN 1	AN 1

Table 6.9 Best load-balancing method for different models and number of processors for the three work-queue load-balancing methods. Key: The first one or two letters specify the load-balancing method: F = fixed region boundaries, A = per-frame adaptive region boundaries, N = when-needed adaptive region boundaries. Two letters indicate that the two methods have the same estimated time. The following number specifies the granularity ratio.

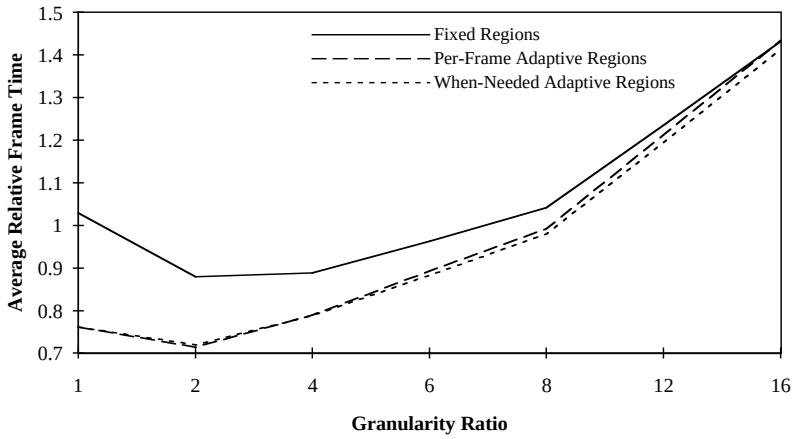


Figure 6.26 Average relative predicted frame times when using work-queue region assignments. Each curve corresponds to the marked load-balancing method.

frame cases with 2 or 4 processors, it is useful to note that the N 2 method also gives the best performance when only those two partition sizes are considered. Figure 6.26 shows the average relative performance for the three load-balancing methods for the simulated granularity ratios.

6.5 Comparison of Region Assignment Styles

Now that we have identified the best load-balancing method for each region assignment style, we can compare the expected performance of each style when load-balancing is also considered. This was done by evaluating the full performance models described at the beginning of the chapter to get the estimated time for each method.

The timing values appear at the end of this section in figure 6.27, with one set of charts for each of the four models. Two charts are shown for each model. The top charts give the overall estimated times, and the bottom charts give times relative to the fastest method for the same model and number of processors. That is, the top charts show the overall performance while the bottom charts shows the additional time required for the slower choices. One additional curve is shown: fixed-region load balancing using six regions per processor, assigned between-frames using triangle counts from at most 32 processors. This method is only included because it is used in the next chapter.

The frame times for the assignment styles that do active load-balancing decrease as more processors are used, but only up to a certain partition size. The transition size depends on the assignment style and the model: the size decreases with the expense of the load-balancing method, and increases with the size of the model. The second effect occurs because you have more time to perform the load-balancing with the larger models.

The highest performance assignment style for each configuration is shown in table 6.10. Using work-queue load-balancing is fastest with 2 or 4 processors, and is unreasonably slow with 32 or more processors. The between-frames assignment style is only slightly slower than the work-queue style with 2 or 4 processors: it takes at most 3.7% more time. If the load-balancing time is omitted, the between-stages style is faster than the between-frames style when more than two processors are used. However, the between-stages style is slower overall when the load-balancing time is considered. The between-frames style is the fastest for a number of configurations using between 32 and 256 processors. As shown in table 6.11, it is not much slower than the best assignment style for any case using less than 128 processors.

The static load-balancing method is the fastest with the smaller models when using 512 processors. In the cases where it is not the best method, static load-balancing takes between 2% and 60% more time; the average is 18% more time, or 17% more time if the configurations where it is the best choice are included (see table 6.12).

The best overall method would be a hybrid, switching between the best assignment style based on the current model and number of processors. A much simpler method would switch between the static and between-frames styles. This would only increase the frame time by at most 3.7%. The switch could be made automatically by evaluating the performance model at run time, and picking the faster method.

If a single assignment style were to be picked, it would be either between-frames assignment or static assignment. The static style would be the best choice if high speed when using all 512 processors was the most important, and the between-frames style is the best choice if performance over the entire range of partition sizes is important. The sampled between-frames method is a compromise between the two. The additional time required by each of the three styles compared to the best style for each configuration is shown in tables 6.11–13.

	Number of Processors								
Model	2	4	8	16	32	64	128	256	512
PLB Head	Frame	Queue	Frame	Frame	Frame	Frame	Frame	Frame	Static
Terrain	Queue	Queue	Stage	Stage	Frame	Stage	Frame	Frame	Frame
CT Head	Queue	Queue	Stage	Stage	Frame	Frame	Frame	Frame	Static
Polio	Queue	Stage	Stage	Stage	Frame	Frame	Frame	Frame	Frame

Table 6.10 Best region assignment style for different models and number of processors. Key: Queue = work-queue, Stage = between-stages, Frame = between-frames, and Static = static assignments.

	Number of Processors								
Model	2	4	8	16	32	64	128	256	512
PLB Head	-	1.33	-	-	-	-	-	-	43.53
Terrain	1.18	3.72	2.57	5.32	-	2.97	-	-	-
CT Head	0.72	1.69	1.58	1.51	-	-	-	-	25.11
Polio	0.03	2.23	0.07	1.39	-	-	-	-	-

Table 6.11 Percentage of additional time taken when using between-frames assignment of regions compared to the best assignment style for each configuration. A dash indicates that between-frames region assignment is the best option.

	Number of Processors								
Model	2	4	8	16	32	64	128	256	512
PLB Head	13.80	16.58	23.74	37.15	35.75	31.23	20.68	11.92	-
Terrain	3.61	13.44	17.61	26.82	31.36	29.94	59.65	14.50	2.37
CT Head	3.62	6.42	11.93	15.42	16.15	17.36	16.13	20.41	-
Polio	1.60	2.32	7.23	6.49	14.14	13.38	19.19	49.09	26.19

Table 6.12 Percentage of additional time taken when using static assignment of regions to the best assignment style for each configuration. A dash indicates that static region assignment is the best option.

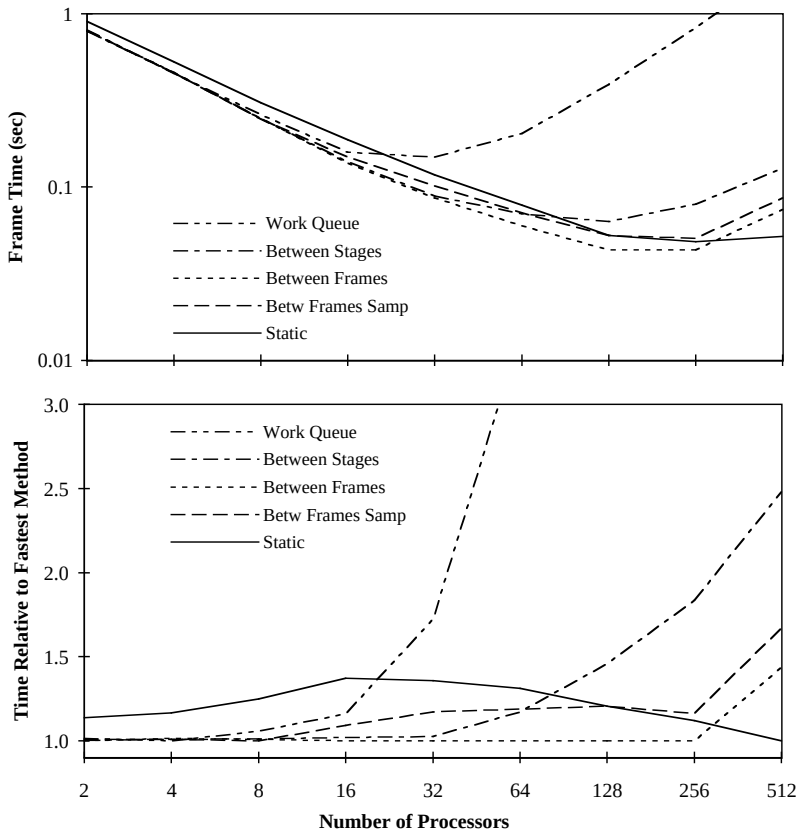


Figure 6.27a Comparison of five region assignment styles and load-balancing methods for the PLB Head model. The top chart gives the predicted frame times for each method. The bottom chart shows normalized times, with each value divided by the time for the fastest method for the same number of processors. Different curves plot the predicted performance when assigning regions statically, assigning regions between frames with sampled cost estimates, assigning regions between frames, assigning regions between stages, and assigning regions using a work-queue.

	Number of Processors								
Model	2	4	8	16	32	64	128	256	512
PLB Head	0.78	0.96	-	9.07	17.15	18.70	20.68	16.34	66.88
Terrain	2.00	1.63	0.05	2.21	-	1.68	0.61	-	13.72
CT Head	0.63	1.21	0.82	0.73	-	-	-	-	41.08
Polio	0.79	0.17	0.14	-	-	0.59	0.83	0.71	2.14

Table 6.13 Percentage of additional time taken when using between-frames assignment of regions, with six fixed regions per processor and using at most costs from 32 processors compared to the best assignment style for each configuration. A dash indicates that this type of region assignment is the fastest option.

Table 6.13 gives the performance of the sampled between-frames assignment style using the fixed-region load balancing method with six regions per processor. The values from this method are only included in the computations for figure 6.27 and table 6.13; they are not included in the computations for tables 6.10–6.12.

However, as mentioned earlier, the performance model does not account for any latency in the communication; thus, this time is a lower bound. Measurements of an actual implementation show that the time can be considerably longer when using many processors. More details follow in the next chapter.

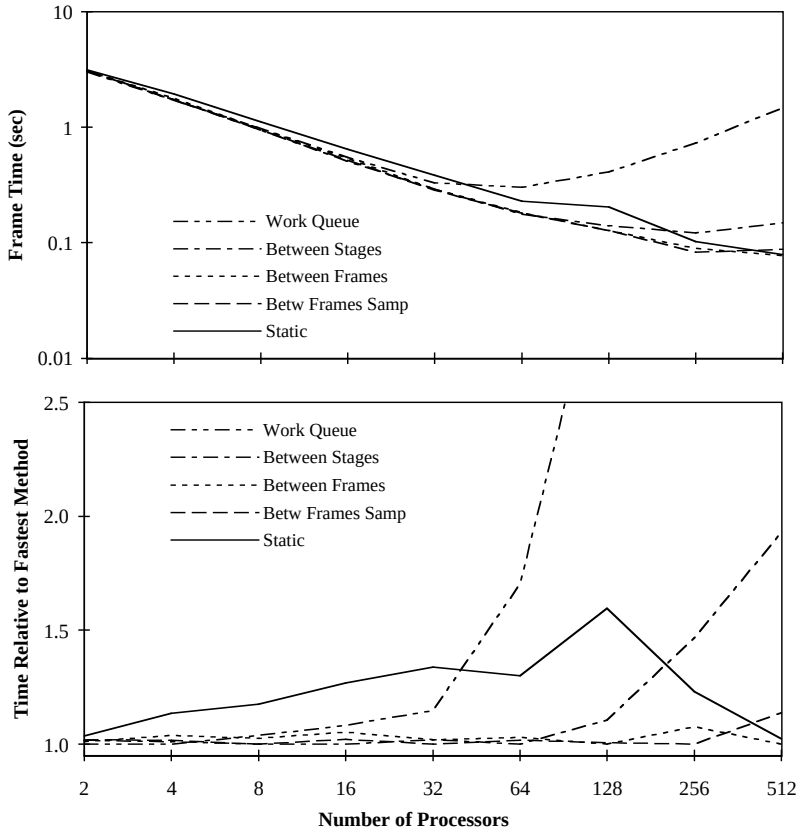


Figure 6.27b Comparison of five region assignment styles and load-balancing methods for the Terrain model.

6.6 Summary and Conclusions

In this chapter, we have examined several methods for load-balancing the rasterization. We have considered load-balancing methods for the four region assignment styles, and have evaluated them using a simulator and performance model. The simulator takes a geometric model and a series of viewpoints, simulates the load-balancing algorithm and the pertinent parts of the rendering algorithm, and computes the estimated processor utilization and the overlap factor. The performance model uses those two values to calculate the estimated overall frame time, including time for rendering as well as the load-balancing operations. We first selected the best load-balancing method for each type of region assignments, and then selected the best region assignment method.

We have seen two load-balancing methods for static region assignments. The interleaved method assigns regions to processors in a regular pattern, and the modular method assigns regions sequentially to processors. The simulator and performance model indicate that the interleaved method using 12 regions per processor gives the best performance.

Also, we have seen three different load-balancing methods for the two once-per-frame assignment methods. One load-balancing method uses fixed region boundaries. Two methods adjust the region boundaries with the goal of having each region perform the same amount of work, but with the constraint that the region boundaries form a (possibly irregularly spaced) grid. One of these methods adjusts the region boundaries every frame while the second only adjusts the boundaries when a region becomes a bottleneck, when it has more than $1/N$ of the rasterization workload. The simulations showed that the methods that adjusted the region boundaries balanced the loads better than the fixed region boundary method, especially when there are only a few regions per processor. The increased processor utilization is accompanied by higher overhead: the fixed region boundary method causes fewer triangles to cross the region boundaries, which must be processed more than once. The additional overhead does not overcome the additional processor

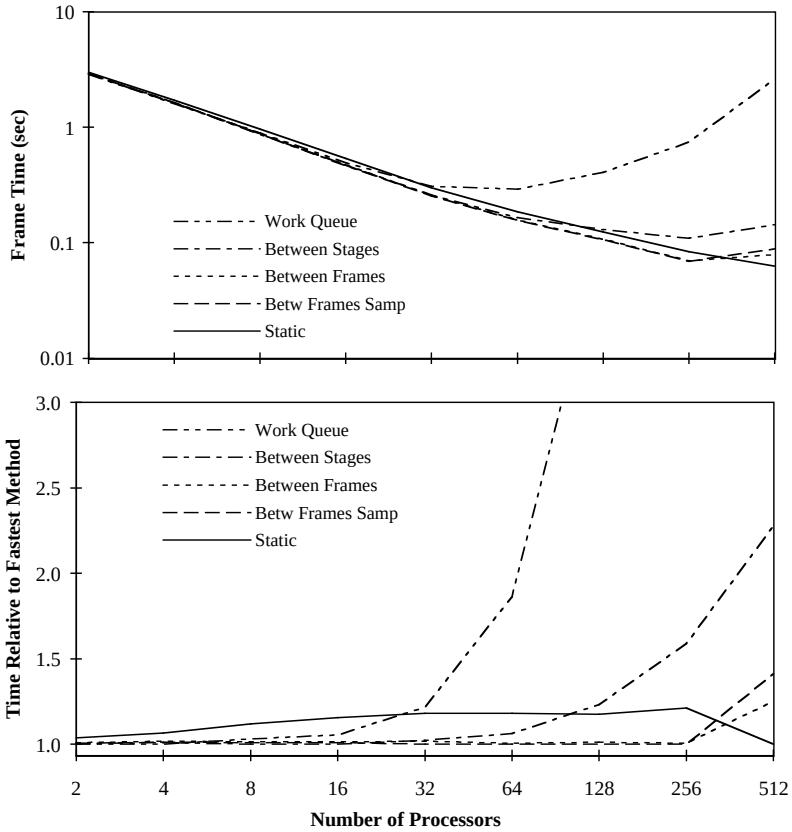


Figure 6.27c Comparison of five region assignment styles and load-balancing methods for the CT Head model.

utilization for the best granularity ratios, making the when-needed adaptive method the best one. The best granularity ratio for the between-stages assignment style is six, and the best ratio for the between-frames style is four.

The once-per-frame load-balancing methods use estimates of the time to rasterize each region when assigning regions to processors. Two per-region time estimates were evaluated: one that uses counts of the triangles in each region, and one that uses the bounding box of each triangle. Using the bounding box requires more calculation but gives better estimates, which allows the loads to be more evenly balanced. However, because the sample geometric models have triangles that only cover a few pixels, the improvement in performance was at most 3.4%. This improvement must be reduced by the additional processing time required, which negates the increase in rasterization efficiency.

Finally, we evaluated three load-balancing methods for use when assigning regions during rasterization using a work-queue. The methods use the same region boundary adjustment techniques as the once-per-frame methods. Because work-queue region assignment requires an expensive redistribution method, all the methods performed poorly when using more than 16 processors. When using a few processors, the best method used two regions per processor and per-frame adaptive region boundaries.

After selecting the best load-balancing method for each of the region assignment styles, we selected the best assignment style. The work-queue region assignment method works best with 2 or 4 processors. The static assignment method is the best with 512 processors, depending on the complexity of the model being displayed, because the other region assignment method's load-balancing operations cannot be completed in time. Assigning regions between frames works best in many of the remaining cases. It is at most 4% slower than the work-queue assignment style, and 5% slower than the between-stages style. The conclusion is that the best algorithm is a hybrid, one that switches between static assignments and assigning regions between frames, based on the size of the model and the number of processors. The switch can be made automatically by evaluating the performance model for each frame and selecting the faster method.

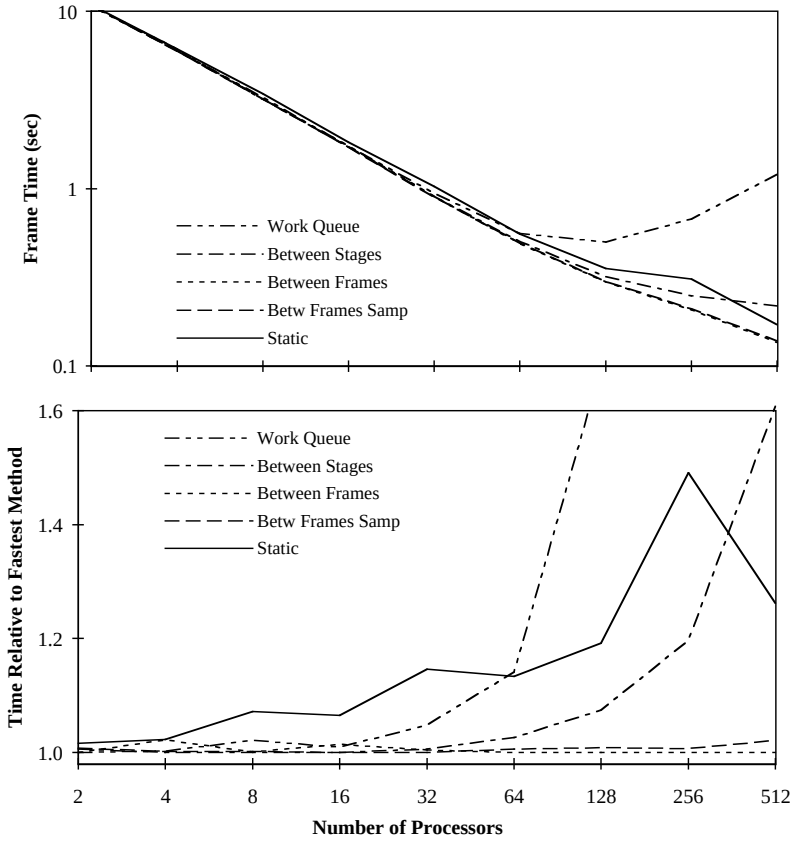


Figure 6.27d Comparison of five region assignment styles and load-balancing methods for the Polio model.

CHAPTER SEVEN

SORT-MIDDLE IMPLEMENTATION ON THE TOUCHSTONE DELTA

This chapter describes an implementation of the parallel rendering algorithm chosen in Chapters 4 and 6 for the Intel Touchstone Delta, a multicomputer that was described at the end of Chapter 1. The implementation allows us to evaluate the design choices made in the earlier chapters, and to check the accuracy of the performance model used to make some of the choices. The evaluation also validates the design methodology since it was used to make the design choices. Additionally, this chapter also allows us to explore one design parameter, the message size, that could not be evaluated with the tools used in the earlier chapters. Finally, the implementation description fills in the details of the algorithm.

As is detailed in the following sections, the implementation verifies the design choices and shows that the performance model is reasonably accurate for systems using up to 128 to 512 processors. The performance when rendering the smaller models matches the performance model better because the performance with larger models run into network limitations when running on large systems, which causes the triangle redistribution to be the bottleneck. Even with the bottlenecks, the rendering speed is over 3 million triangles per second when using the largest model and 512 processors.

Two recent implementations have shown higher performance than this implementation (both are described in Chapter 2). The first, the Cray Animation Theater package for the Cray T3D renders 4.5 million triangles per second [Kili94] using a 153-million triangle model on 256 processors. However, this not an interactive application, and the T3D is a more powerful machine. As a comparison, the successor to the Delta, the Intel Paragon, with 512 processors gives between 25 and 109% of the performance of a 256-processor T3D when running different NAS Parallel Benchmarks [Bail94, Sain95]. The geometric mean of the relative rates is 56%.

A second recent implementation is the one by Lee et. al. [Lee95]. While it also runs on the Touchstone Delta, the models that runs at rates higher than three million triangles per second have very small triangles—ones smaller than a pixel. This type of model is not typical of visualization applications.

The current implementation on the Delta does not allow fully interactive viewing of the rendered images because the attached frame buffer has limited bandwidth. Because of this, the algorithm is measured without sending the computed image to the frame buffer. More recent systems do not have this bottleneck, as detailed in section 7.2.1.

The first part of the chapter (section 7.1) describes the implementation in detail. Later sections (7.2–7.3) characterize the performance of the algorithm, and measure how well the performance model described in Chapter 6 predicts the performance. The final section describes possible improvements to the algorithm.

7.1 Implementation Description

The implemented algorithm is similar to the one that was chosen in the previous chapters. The algorithm is different because work on the design methodology continued after the work on the implementation was halted. The overall performance of the implementation is still fairly close to the performance of the selected algorithm. The differences are that the implementation uses fixed region boundaries instead of ones adjusted as necessary, it uses triangle counts instead of size-based estimates, and it uses a granularity ratio of 6 instead of 4. Also, a single load-balancing method is used that limits the collection of cost esti-

mates to 32 processors, instead of using a hybrid load-balancing method that switches between active and static load-balancing based on the model size and number of processors.

The implementation has three main characteristics:

1. Fast. The implementation has optimized inner loops and uses a simple shading model, Gouraud shading.

2. Simple. The implementation only supports convex lighted polygons and a few other functions, and uses static models for testing the performance.

3. Allows experiments. Switches can enable different algorithmic variants for testing, and instrumentation can show the speed of different portions of the algorithm.

The implementation is described in five sections. The first section describes the interface seen by a user, and the next describes the algorithm. The last three sections give implementation details about the redistribution of triangles, the load-balancing, and the graphics routines.

7.1.1 Interface Features

The implementation is organized into two parts: an immediate-mode graphics library, and a program that exercises the library. The immediate-mode interface has the following features:

- Shading Model. The implementation supports full color (24 bit) Gouraud shading using one directional light source (a point light source at infinity). The amount of ambient light can be set. Anti-aliasing is supported; when enabled, it samples each pixel in a 4x4 regular grid and applies a box filter to obtain the final color. Anti-aliasing is done in one pass by rendering an image that is expanded by 4 in each dimension and then filtering 16 pixels into 1.

- Viewing Model. Only perspective projection is supported. The viewing matrix, field of view, and hither plane can be controlled by the user. A right-handed coordinate system is used.

- Hidden Surface Elimination. Hidden surfaces are removed using a 32-bit *z*-buffer. The inverse of *z* ($1/z$) is interpolated so the *z* values are correct when interpolated in screen space.

- Primitives. Only convex polygons are supported. The interface requires that each vertex have a normal vector. A separate utility routine can be used to calculate the normal vectors for a polygon. The tested datasets only contain triangles with pre-computed normal vectors.

- Attributes. Colors and modeling matrices are supported. Separate colors for the front and back of polygons can be specified. The current modeling matrix can be replaced, or a new matrix can be pre- or post-concatenated with it. Back-face culling is also available.

- Image destination. The computed image can be sent to the Delta frame buffer, to disk, or to both. Image sizes up to 1280x1024 are supported.

Some of the settings in the interface have an effect for the entire frame, such as the window size, anti-aliasing, image destination, and hither distance. These must be set the same on all the processors; otherwise, the implementation will fail or produce a faulty image. The primitive generation for each frame is bracketed by calls that start and end the generation. The two calls are described in more detail in the next section.

7.1.2 Algorithm Description

Pseudocode for the per-frame loop is shown in figure 7.1. The details of the algorithm are described in later sections. The pseudocode is shown as if the high-level algorithm was written in a single routine. However, the middle portion of the per-frame loop is part of the application, and thus in the actual implementation the code is split into sections. The application executes the overall per-frame loop and calls subroutines to perform the graphics operations.

The first part of the per-frame loop performs the initialization, which is shown on lines 1 to 9. The middle part generates the triangles: the application executes an iteration of the per-triangle loop (lines 10–15) for each raw triangle. Once a triangle is generated, the application calls a subroutine that performs the body of

```
 1  if (processor 0) {
 2      compute processor assignments
 3      broadcast processor assignments
 4  }
 5  else
 6      read processor assignments
 7  synchronize   /* ensure previous frame finished */

 9  allocate output buffers (one per region/processor/group)
10  for each raw triangle {
11      transform to screen space, clip, light vertices
12      calculate region(s) it overlaps
13      copy display triangle into corresponding buffer(s)
14      send buffer if is full
15  }
16  flush output buffers
17  send per-region work estimates in summing tree to processor 0

19  if (stream-per-region) {
20      while (not all regions done) {
21          if (memory available)
22              allocate a region-sized local frame buffer
23          for (each frame buffer allocated) {
24              rasterize region's available display triangle
25              if (region is done)
26                  send region to frame buffer and/or disk
27          }
28      }
29  }
30  else {
31      allocate local frame buffer for each of my regions
32      rasterize triangles into appropriate frame buffer
33      send regions to frame buffers and/or disk
34  }
```

Figure 7.1 Pseudocode of the implementation's rendering loop.

the loop for that triangle. After all the triangles have been generated, the user calls a final function that executes the remainder of the algorithm (lines 16–34). When that function returns, the user's program has control until it calls the initialization function for the next frame.

The implementation uses one of two rasterization loops depending on whether stream-per-region or stream-per-processor redistribution is used. As mentioned earlier, stream-per-region redistribution works best with a few processors since it uses less frame and *z*-buffer memory, while stream-per-processor redistribution works best with more processors, since it reduces the minimum number of messages required. Only the outer rasterization loop needs to be duplicated; each rasterization loop calls common routines, so the duplicated code spans only a couple of pages.

7.1.3 Redistribution

This section gives the details of how the implementation performs redistribution. As the section goes into considerable detail, the casual reader may wish to skim the earlier subsections, resuming with section 7.1.3.5, as the details are mainly of interest to specialists or implementers. The choice of redistribution method is quite important, as it can make a factor of 100 difference in the cost of redistribution, but most issues were discussed earlier in section 4.4.

The redistribution of triangles between processors is organized in a series of streams between processors. The number and structure of the streams depend on the communication method. The implementation supports the four different methods described in section 4.4: one-step stream-per-region, two-step stream-per-region, one-step stream-per-processor, and two-step stream-per-processor.

Each stream consists of a series of shaded screen-space triangle that are placed into messages. The messages have a maximum size to simplify memory allocation, and to overlap the communication with other calculations. When the triangles are placed into the messages, they have a header that allows multiple primitive types. The header consists of three short integers: the type of primitive, the size of the primitive, and the region where the primitive will be rendered. Primitives that overlap multiple regions are sent once for each region they overlap, even if more than one of the overlapped regions is rendered on one processor. The probability that a primitive will be sent more than once to a processor is very low for small primitives: it is approximately $(O-1)/N$. For the algorithm used, this probability ranges from 0.1 to 2%, and thus the additional complexity for removing the duplicate primitives does not appear to be justified.

Only polygons are currently implemented. Their data structure consists of a short integer indicating the number of vertices and an array of vertex data. Each vertex has a float for x, y, and z, followed by 10 bits of red, green, and blue packed into four bytes. Including the header, a triangle data structure is 56 bytes long.

The messages are sent asynchronously, using the NX/M [Inte91b] `isend` function. This function sends the message immediately when possible; if not, it queues the message for later delivery. The function returns a message identifier, which is saved to later test whether the message has actually been sent. The implementation defers the check for message departure until the storage for message identifiers is full, or when the supply of message buffers is exhausted. This strategy decreases the amount of time spent waiting for a message to leave. Arriving messages are handled using message handlers, which are interrupt-driven receive routines. In one-step redistribution, and in the second step of two-step redistribution, the handler saves the triangles for later rasterization, and gives the operating system a new buffer to fill. In the first step of two-step redistribution the handler copies incoming triangles into the new buffers, sending the new buffers when filled, and gives the operating system a new buffer by calling the `hrecv` function. While the per-message overhead is higher when using message handlers, this technique avoids having to poll for incoming messages, and should reduce the amount of copying required. (If a message arrives before the operating system is given a buffer, it is stored in system memory, and is later copied into the user's buffer. The message is placed directly into the user's buffer if the operating system has a pointer to the buffer before the message arrives.)

The number of processor groups is chosen using the optimization method described in section 4.4.6. The number of groups is only chosen during initialization using a user-supplied number of primitives plus an estimated overlap factor. The replay program used to measure the implementation's performance (described in section 7.2) uses the number of triangles in the static model as the number of primitives, dividing by 2 if back-face culling is enabled. The overlap-factor calculation assumes primitives have a height of 2% of the vertical screen dimension, and is calculated using equation 3.1. The number of groups could be varied from frame to frame, using the actual number of primitives rendered in the previous frame as a guide. However, the optimization method can take significant time to calculate the number of groups: with 512 processors the performance model is evaluated many thousands of times, taking a total of about 46 ms. This would be too expensive when running at a high frame rate.

Also, the decision whether to use two-step redistribution is made automatically at system initialization by comparing the amount of time to perform one- and two-step redistribution as predicted by their respective performance models. Note that these performance models are the ones developed in Chapter 4; they do not consider processor utilization during rasterization as the utilization value is unavailable.

The next four sections describe how the four redistribution methods are implemented. The methods are one-step stream-per-region, two-step stream-per-region, one-step stream-per-processor, and two-step stream-per-processor. Note that the two-step stream-per-region method is not normally used as it is less efficient than the other methods.

7.1.3.1 One-step Stream-per-Region Redistribution

The first method has each processor send a stream of messages for each region per frame. During transformation, each processor maintains one buffer per region. When a buffer is filled, or at the end of the geometry processing, it is sent to the processor that has been assigned that region to rasterize. Each processor uses one mailbox (or message id) for each of the regions it will be rasterizing. Incoming messages are placed in a linked list, one list per region. Figure 7.2 gives an example of this method's communication paths when using four processors and six regions. The regions are assigned to processors as follows: pro-

cessor 0 has regions 0 and 1, processor 1 has region 2, processor 2 has 3 and 4, and processor 3 has region 5. Each of the paths between processors is traversed by one or more messages. A field in each message indicates whether the message is the last message in the path for the current frame.

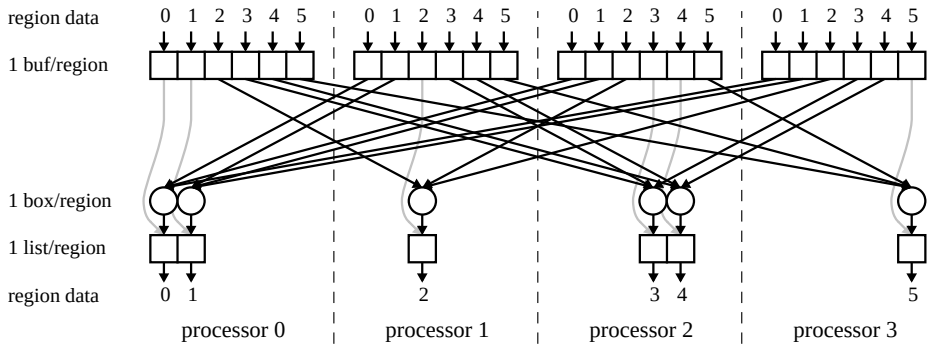


Figure 7.2 Communication paths for the one-step stream-per-region communication method. Each square represents a buffer or a buffer list, and each circle is a mailbox. Messages containing triangles that will rasterized locally are not sent to a mailbox; instead, they are immediately put in the appropriate list of messages (indicated by gray paths).

7.1.3.2 One-step Stream-per-Processor Redistribution

The second communication method sends a minimum of one message to each of the other processors, in one step. During rasterization, each processor maintains a buffer for every processor. Triangles for different regions are packed into one message. Like before, a buffer is sent to the rasterization processor when it is filled or at the end of the geometry processing. Each processor maintains a mailbox for each of the other processors, which usually increases the amount of buffering available compared to the buffering for the previous redistribution method. When a message arrives, it is put in the single list of triangles that will be rendered. Figure 7.3 shows the communication paths with the same number of processors and region assignments as the last example.

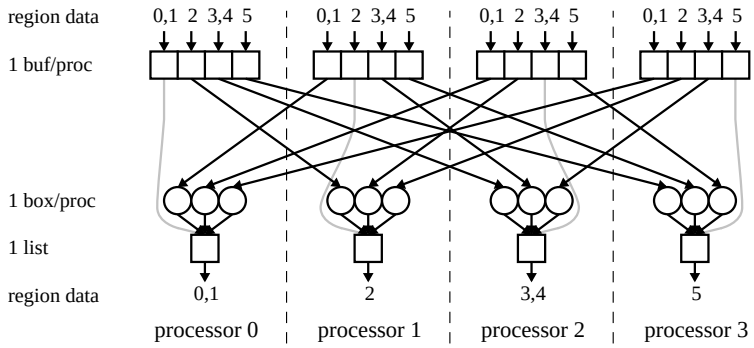


Figure 7.3 Communication paths for the one-step stream-per-processor communication method. Each square represents a buffer or a buffer list, and each circle is a mailbox. Messages containing triangles that will rasterized locally are not sent to a mailbox; instead, they are immediately put in the appropriate list of messages (indicated in gray).

7.1.3.3 Two-step Stream-per-Region Redistribution

The next communication method involves two-step redistribution, where the triangles are first sent to a forwarding processor, and then to the final processor. As discussed in Chapter 4, this reduces the minimum number of messages that must be sent between processors. Figure 7.4 shows the communication paths as before. The decrease in paths can be seen by comparing the number of streams that cross processor boundaries (shown as dashed lines) with the number shown in figure 7.2.

In this method, the processors are organized into groups; in the example, there are two groups. In the first step of the redistribution, each processor sends messages to one processor in each group; that processor is

called the *corresponding* processor. In the second redistribution step, each processor sends messages to the other processors in the same group. If the groups are all the same size, in the first step the ith processor in each group will exchange messages with the ith processor in each of the other groups. However, the groups often have different sizes. When that happens, every processor is assigned a corresponding processor for every other group. The assignments are made for each group by numbering the processors outside the given group modulo the number of processors within the group. The assigned number is the position of corresponding processor within the group. This numbering method has each processor receiving approximately the same amount of triangles.

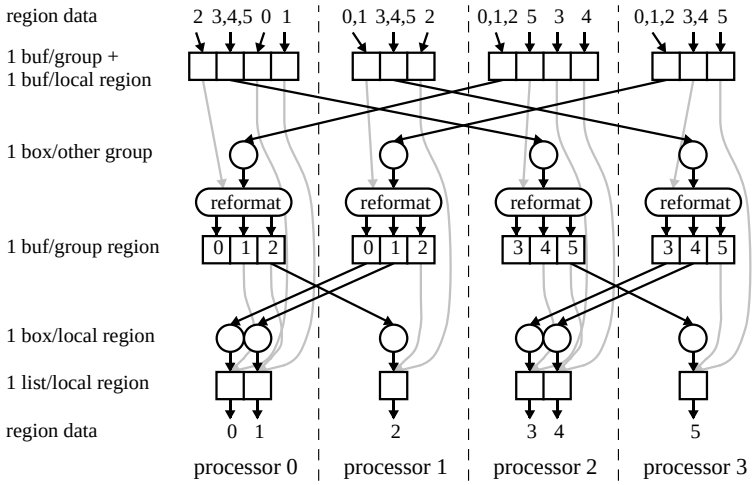


Figure 7.4 Communication paths for the two-step stream-per-region communication method. Each square represents a buffer or a buffer list, and each circle is a mailbox. Messages containing triangles that will be reformatted or rasterized locally are not sent to a mailbox. Buffers for regions assigned to the local processor are placed at the end of the first-step buffer array. The example is for four processors, six regions, and two groups.

During geometry processing, each processor maintains one message buffer for each of the other groups, plus one buffer for each of the regions that it will render. The latter is an optimization that saves having to reformat messages that do not need to be sent. The messages are sent to the forwarding processors, where the triangles in the messages are copied into new message buffers. The forwarding processors maintain one message buffer for every region assigned to a processor within the group. Those buffers are then sent to the final processor. Each processor maintains a mailbox for each of the other groups to handle incoming messages to be forwarded. It also maintains a mailbox for each region it will rasterize to handle messages containing the triangles for rasterization.

7.1.3.4 Two-step Stream-per-Processor Redistribution

The fourth method is similar to the previous method. Sending a stream of messages per processor instead of a stream per region means that the buffers for all the regions that a processor handles are replaced by a single buffer. This change is made for the transformation buffers, the reformatting buffers, and the list of triangles for rasterization. Figure 7.5 gives an example of this method using the same configuration and region assignments as the previous figure.

7.1.3.5 Other Redistribution Issues

Supporting four redistribution methods does complicate the implementation. The transformation inner loop can support the different methods by changing the contents of a table used to map a region to the buffer where the triangle should be placed. The only additional cost occurs when using one-step stream-per-region redistribution since the table lookup is not necessary. However, the additional cost is minimal. The routines that handle the message buffers are more complex, and thus slower. However, this additional time is amortized over all of the triangles in the message so it creates minimal impact.

One desirable redistribution feature has not yet been implemented. Since the load-balancing algorithm is predictive, it will not balance the loads very well if the next frame's distribution of triangles across the

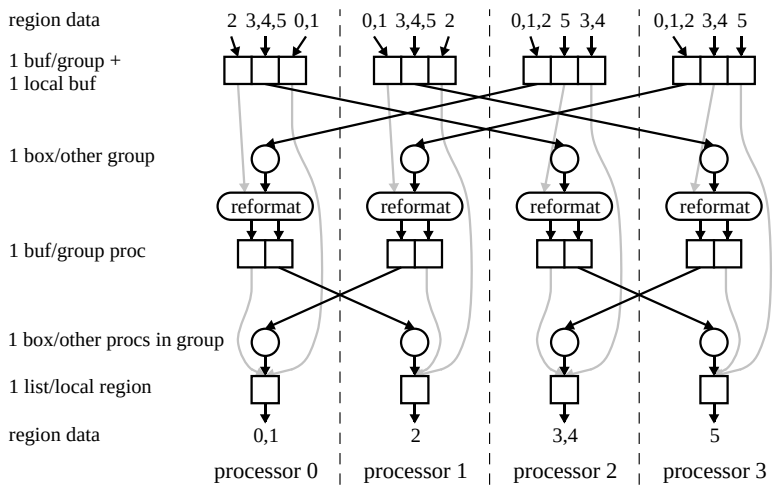


Figure 7.5 Communication paths for the two-step stream-per-processor communication method. Each square represents a buffer or a buffer list, and each circle is a mailbox. Messages containing triangles that will be reformatted or rasterized locally are not sent to a mailbox. The buffer for the regions assigned to the local processor is placed at the end of the first-step buffer array. The example is for four processors, six regions, and two groups.

screen is quite different from the previous frame's distribution. When this happens, the overloaded processors will receive a disproportionately large fraction of the triangles, and can run out of buffer space. It would be desirable for the implementation to detect running out of buffer space and recover, instead of causing a fatal error. The implementation could recover by not giving the operating system new receive buffers when it runs low on buffer space, eventually blocking incoming messages (some messages will be stored in the system's buffers). Once an overloaded processor has rasterized enough triangles to free up some buffer space, it can give the operating system receive buffers for more messages.

7.1.4 Load-Balancing Implementation

The load-balancing algorithm uses once-per-frame assignment of fixed, approximately same-sized square regions, as described previously. The assignment algorithm uses the previous frame's per-region triangle counts to assign regions to processors using a greedy bin packing algorithm (described in section 4.3.2). The triangle counts are collected on each processor during the geometry processing. A polygon with more than three vertices is counted as v–2 triangles, where v is the number of vertices, since the polygons are rendered one triangle at a time.

After the geometry processing is completed, the counts are collected across all the processors using a summing tree (see figure 7.6). Leaf processors immediately send their counts to the next higher node in the tree. Processors in the interior of the tree wait for the incoming counts, then sum their counts with the incoming ones and send the result to the next higher node.

The message fan-in, the maximum number of messages each processor receives, is a compromise between latency and the load-balancing of the data collection process. Larger fan-in values have the interior nodes do more work than leaf nodes, but has smaller latency when the network is congested. When the network is not congested, fan-in values larger than 2 can slightly increase or decrease the overall latency because the larger fan-in values decrease the depth of the gathering tree but increase the time at each level (see equations 6.5–6.6 with the expression for the number of levels replaced by $\lceil \log_f(N(f-1)+1)-1 \rceil$, where f is

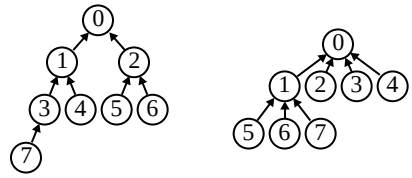


Figure 7.6 Summing trees using eight processors with fan-ins of two (left) and four (right).

the desired fan-in). Since the latency is more important with more processors, the implementation uses a fan-in of two when using under 32 processors, and a fan-in of 4 otherwise. The value of 32 was chosen because the load balancing bottleneck was seen when using more than 32 processors.

The implementation can be configured to gather triangle counts from all processors if sampling the counts is not desirable. This should be done whenever it is likely that each processor's triangles' screen locations would not be representative of the overall distribution of locations; this would often occur when the model is generated by the application for each frame.

The load-balancing algorithm uses an estimate of the time to rasterize each region instead of using the raw triangle counts. This allows the algorithm to account for the per-region overhead: clearing the buffers and sending the pixels to the frame buffer. It also allows the algorithm to account for the time taken by processor 0 to calculate the region assignments. The per-region rasterization estimate is calculated by multiplying the number of triangles by the sum of the time it takes to receive a triangle (assuming a 4096 byte message) and the time to rasterize a 5 pixel triangle that crosses 4 scanlines. The pixels and scanline values are representative of the four example models. Then, the region processing times are added, with a per-region fixed cost plus a time per pixel. The times used are the same ones used in the performance model in section 6.1.2. When anti-aliasing is enabled (4x4 regular grid sub-sampling), the triangle time is the rasterization time of an 80 sample triangle that crosses 16 scanlines, and the per-pixel region time is increased by 0.21 μs per pixel, the time required to filter a pixel.

The algorithm includes the time taken by processor 0 to perform the region assignments. This is done using a function to predict the time for running the algorithm; the function was determined using a least-squares fit to measured data, and is described in section 4.3.6. The constant cost of performing load-balancing is increased by 15 ms to account for any variations in the algorithm. This was done because all the other processors will have to wait if the load-balancing operation takes longer than predicted, while only processor 0 idles if the load-balancing operation finishes early. For the same reason, processor 0 is relieved of having to forward messages for two-step redistribution if it is assigned no regions to rasterize. The time taken for geometry processing and two-step redistribution is not included because those operations are assumed to be independently load balanced.

When gathering counts from a subset of the processors, the total counts should be divided by the fraction of processors contributing to the counts (e.g., if gathering from half the processors the counts should be doubled). The scaling will make the counts have the same scale as the per-region times and the region assignment times also used in the total per-region estimates. The current implementation has a minor bug in the cost gathering code: the counts are not scaled. Simulations similar to those shown in Chapter 6 indicate that the bug results in a decrease in processor utilization of about 1 to 10%. The smaller models have the largest decrease because the triangle rasterization is a smaller part of the total rasterization cost.

The inner loops of the load-balancing operation (sorting and reordering the heap during region assignment) have been optimized so that they are less likely to be a bottleneck when using many processors.

7.1.5 Graphics Routines

The graphics routines are optimized for speed. The transformation code was adapted from the Pixel-Planes 4 system [Fuch85]. It does the vertex transformation with one matrix multiply, and does not transform the normal vectors. Instead, the light vector is transformed into modeling space, so it can be used directly with the untransformed normals. Because the region boundaries are regular (consisting of cutting the screen from top to bottom and left to right), the regions that a triangle overlaps can be determined using a table lookup of the coordinates of the triangle's bounding box.

There is a "fast path" for transforming triangles. It is written in Intel i860 assembly language using the dual operation and dual instruction modes. It only handles triangles, and only clip-tests them against the hither plane. Other polygons and triangles that straddle the hither plane are processed in a much slower C routine. A fast path could be added for other primitives such as quadrilaterals.

The rasterization routine renders polygons a triangle at a time, allowing orientation-independent shading [Fole90, section 16.2.6]. It is derived from a version from a source who wishes to remain anonymous. This routine is also written in i860 assembly language using the dual-operation and dual instruction modes as well as the graphics instructions. Because this routine rasterizes two pixels at a time, all regions must be an even number of pixels across. This restriction is currently visible in the window size setting: only win-

dows with an even width are allowed. This restriction could be removed with additional code that, during rasterization, adds a column of pixels to regions with an odd width, and afterwards copies the pixels to a correctly sized region. The routine to filter pixels for anti-aliasing is also written in assembly language.

7.2 Performance Results

To test the immediate-mode implementation, I wrote a "replay" program that reads a static model and displays it from a series of recorded viewpoints. During initialization, the program reads a binary version of the model and partitions it among the processors using the more complex method described in section 5.2. Each processor scans the entire model and saves the portion that it will process. The models and viewpoints are the same as those used in the previous chapters, and are described in Appendix A.

During a run, the processors read the recorded viewpoints, and traverse their portion of the model independently, calling the immediate-mode functions. The normal vectors are part of the model, so they are not calculated during the runs. The different runs sampled many different numbers of processors and algorithm variations. Only one run was made for each configuration. The speeds given below are averaged across all frames in the sequence.

The different runs were chosen to explore the implementation's characteristics and to test the predictions made by the simulator and performance model. The different runs were made for each of the four models, and for *partitions* (subsets of the total 512 processors) of 1 to 512 processors (powers of two only). Runs with the larger models were not possible with few processors due to memory limitations.

7.2.1 Avoiding the Frame Buffer Bottleneck

The computed frames were not sent to the frame buffer or to disk since doing so would limit the frame rate. The Delta frame buffer is limited to about four frames per second. Saving the pixels would be the bottleneck in many cases, which would make it difficult to evaluate the algorithm's performance. This means that the rendering speeds given here could not be achieved in real systems since additional time would be needed to send the computed pixels to the frame buffer and display them. However, the speeds can be compared with some other implementations, since at least two report rendering rates when the computed pixels are discarded [Croc93, Whit94, Lee95].

Future systems can avoid the frame buffer bottleneck. For example, a distributed frame buffer has been developed for the Intel Paragon that allows real-time frame updates (at least 30 Hz) [Stol95, Wei95].

7.2.2 Polygons per Second

The first measure is the raw triangles-per-second performance; it is shown in figure 7.7. These figures are for the default settings: stream-per-processor communication, automatic selection between one-step and two-step redistribution, a granularity ratio of 6, a maximum message length of 4096 bytes, and using triangle counts from at most 32 processors. The triangle-per-second values in the charts were computed by first calculating a per-frame speed by dividing the number of triangles in the viewing frustum by the frame time. The final value is the average of the per-frame values.

The reader should note that the above specifications mean that culled back-facing triangles are counted when computing the triangles-per-second values, which is done in some other implementations [Croc93, Whit94, Lee95]. Three of the four models have back-face culling enabled. The reader can convert the performance figures that include back-facing triangles into the number of rendered triangles per second by multiplying the figures by the fraction of non-culled triangles. This fraction can be computed from the values in Appendix A. For example, for the Polio model, on average 49.2% of the triangles in the viewing frustum are rendered. The reported triangle rate of 3.1 million triangles per second corresponds to 1.5 million rendered triangles per second.

The highest speed achieved is 3.1 million triangles per second with 512 processors rendering the Polio model, the largest model. This is much higher than could be achieved on commercial workstations at the time of the experiments (1994). In general, the larger models have higher triangle rendering rates, as the per-frame overhead is amortized over more triangles. The Terrain model does not appear to follow this trend because it is the only model rendered without back-face culling. It should fit this trend if all models were rendered with the same culling setting.

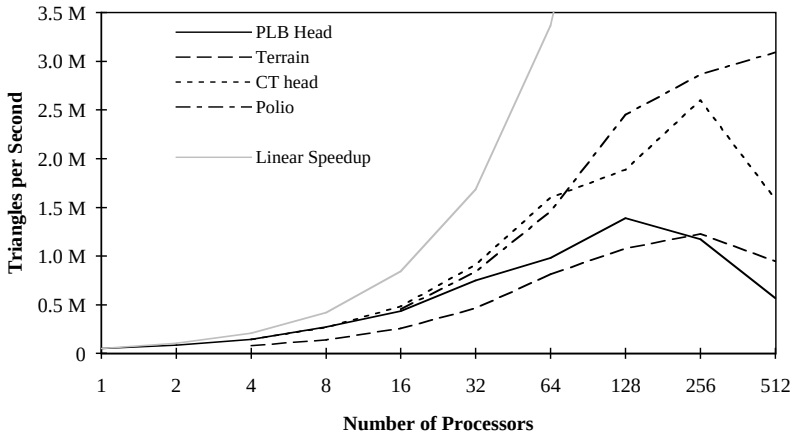


Figure 7.7 Speed of the implementation on the Intel Touchstone Delta for the four models compared with a linear speedup.

With the smaller models, the performance is at its maximum with 128 or 256 processors, and then declines. This is due to cost of the between-frames assignment style causing a load-balancing bottleneck, which was identified in Chapter 6. The performance increase is nowhere near linear speedup (the linear speedup is based upon the polygon rendering speed of the PLB Head model running on one processor). This is due to the increased message overhead when using more processors, and to network congestion. The increased message overhead is shown in section 7.2.5, which shows how time is spent on different parts of the algorithm.

The network congestion can be illustrated with the following simple calculations. At the highest rendering rate of 3.1 million triangles per second using 512 processors, each processor sends $3.1 \times 10^6 \times 1.88/512 =$ 11,400 triangles per second (1.88 is the overlap factor). With 512 processors, 16 processors will each send half their triangles across each of the links crossing the bisection point of the processor array, giving 91,100 triangles per second per link or 5.1 Mbytes per second per link. This is half of the maximum 10 Mbytes per second. The potential maximum cannot be achieved because the all-to-all communication has nearly random communication patterns. Furthermore, the links are not used during the entire frame generation time, and the two-step redistribution at times sends some data over the bisection links more than once.

7.2.3 Load-Balancing Bottleneck

I explored the load-balancing bottleneck by running the algorithm three different ways. The first method gathers the per-triangle counts from all processors and computes the region-to-processor assignments as before. The second method (the default) samples the per-triangle counts from at most 32 processors, and then makes the assignments. The third method does no load-balancing work at all. Instead, each processor uses region assignments saved in a file from a previous identical run. Comparing the last method with the others shows the extent of the load-balancing bottleneck.

Figure 7.8 illustrates the triangle rendering rates for each of the three methods and for each of the four models. The line pattern of each curve indicates the load-balancing method. The load-balancing method has a noticeable affect when using more than 32 processors. Using the previous run's assignments is the fastest, and using triangle counts from all the processors is the slowest. Unlike the results in Chapter 6, the current results show no advantage to using full load-balancing over sampled load-balancing when full load-balancing is not the bottleneck. When reading values from the file, the rendering rates increase as the number of processors increases. However, when performing load-balancing, in nearly all the cases the performance increases to a maximum value and then decreases. Sampled load balancing has higher performance than full load balancing, but its performance still decreases after a point because the load balancing costs still increase with the number of processors (see equations 6.5–6.8 in section 6.1.2.2.2).

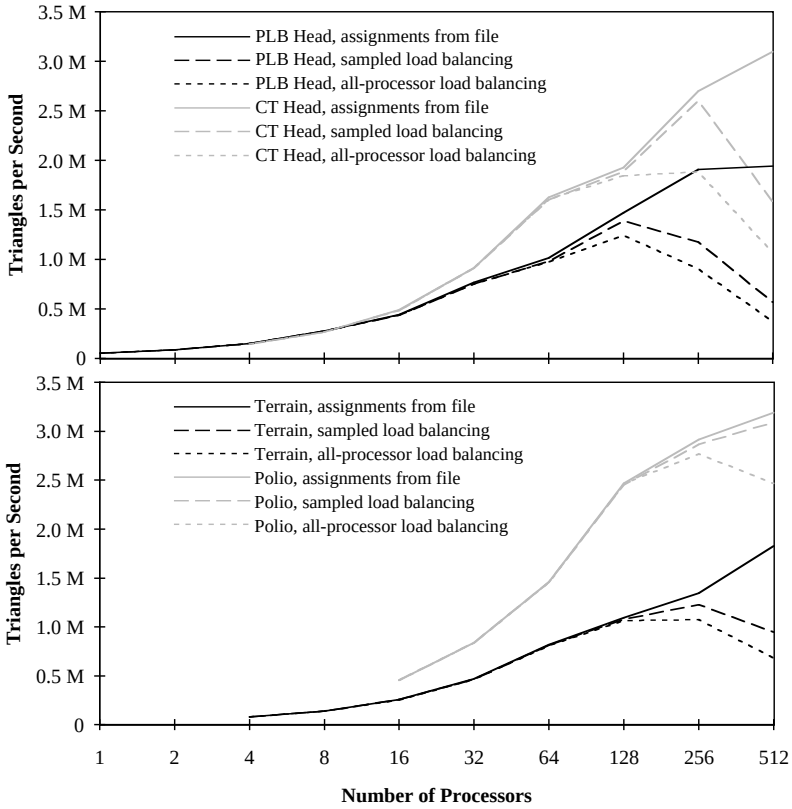


Figure 7.8 Comparison of triangle rendering rates for three load-balancing methods for the four models.

7.2.4 Comparison with the Performance Model

In Chapter 6, different load-balancing methods were evaluated using a simulator and a performance model. To give confidence in those results we need to verify that the predicted times from the performance model match the times from the actual system. The performance model was described earlier in Chapter 6 (see equations 6.3–6.10).

Figure 7.9 plots the predicted and actual total frame processing times for different numbers of processors and models. The total frame time is the same as the average frame time multiplied by the number of processors. The chart shows that the values are fairly close for up to 128 processors; the value diverge with more processors. With 2 to 128 processors, the predicted values are up to 26% higher than the actual times; the average difference is 15%. One explanation for the difference is that the measured per-message communication time is too high, because the time varies with different message traffic patterns, and because the traffic pattern during the measurement is different from the implementation's pattern (see Appendix B for more details). With 512 processors, the predicted frame time is between 58 and 89% of the actual time as the network bottlenecks mentioned earlier are not included in the performance model.

We can further evaluate the network bandwidth bottleneck by comparing the predicted times without load balancing with the actual times with any load balancing bottlenecks removed. The bottlenecks are removed as was done for figure 7.8 by reading the region-processor assignments from a file. These values are shown in figure 7.10. The curves for the smaller do not match as well as the ones that include load balancing. A likely explanation is that the message cost mismatch just mentioned occurs for all partition sizes, and is not hidden by the load-balancing bottleneck. For the larger models, the curves are similar to the those in figure 7.9, which gives more evidence for a bandwidth bottleneck.

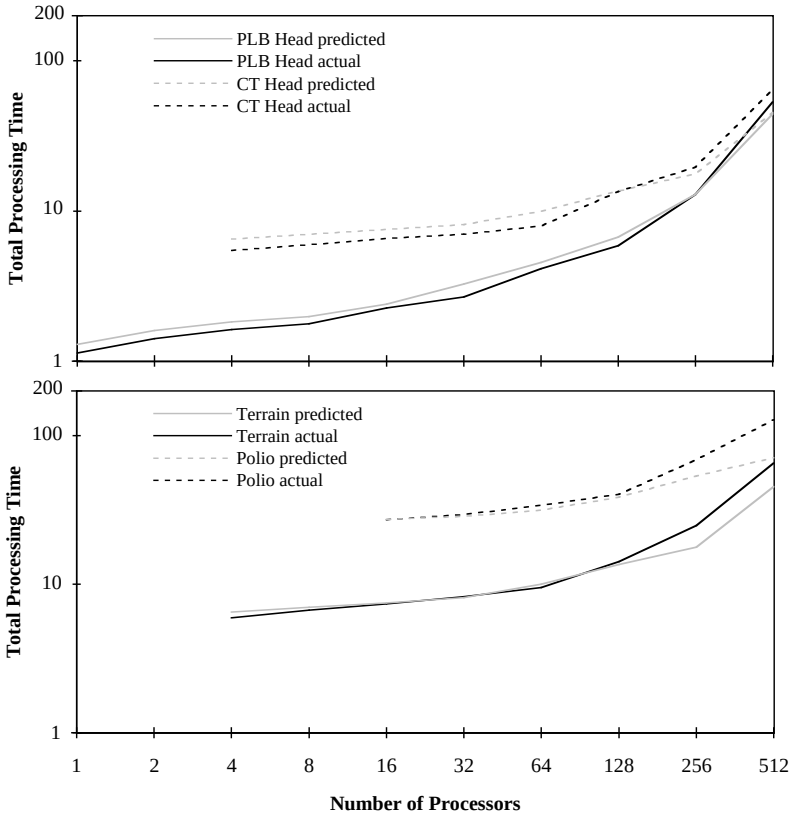


Figure 7.9 Predicted versus actual total frame processing times (average frame time multiplied by the number of processors) for the four models.

7.2.5 Components of the Processing Time

By dividing the overall processing time into its components, we can see why the algorithm fails to scale when using many processors. The various components are from the performance model developed in Chapter 6 instead of being measurements of the implementation because making accurate measurements of the different components is difficult. The measurements are difficult because the communication is done by the operating system at interrupt level. The performance model without load-balancing is used for simplicity. Load-balancing is the predicted bottleneck when using 256 or 512 processors and the PLB Head model, and when using 512 processors with the Terrain and CT Head models.

First, we need to define the components that will be plotted. Table 7.1 shows how the time from the overall performance model (equations 6.3 and 6.4) is divided into nine categories. The Geometry and Rasterization are the times that would be taken on a single processor. These times are similar to the times that would be measured on a uniprocessor implementation except that the code supports using multiple processors. Geometry and Rasterization Overhead are the amount of time spent repeating work that would be done once in a sequential algorithm. The communication time is divided into two parts: Message Time for the per-message cost, and Byte Time for the per byte cost. These and the other communication times have two cases, for each of one-step and two-step redistribution. The amount of time spent reformatting messages is given in the Forwarding component. Finally, the last four components show the time lost due to imperfect load-balancing. The components are for the geometry processing, the rasterization, and the communication associated with the two. Note that the equations in table 7.1 give the aggregate time instead of the per-processor time given by equations 6.3 and 6.4; the aggregate time is the per-processor time multiplied by the number of processors.

Figures 7.11 through 7.14 show how the time is spent rendering the PLB Head model and Polio models, respectively the smallest and largest models. The first two charts give the time spent with the PLB Head model, and the last two charts give the time for the Polio model. The first chart in each pair gives the average aggregate processing time for each frame, and the second chart shows the times as percentages.

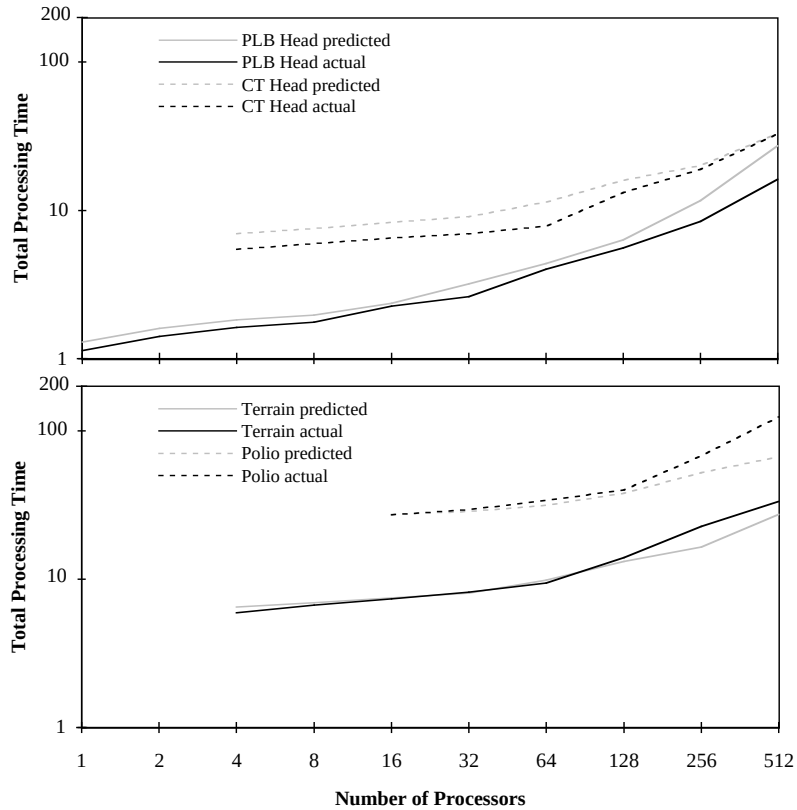


Figure 7.10 Predicted versus actual total frame processing times for the four models when load-balancing information is read from a file.

Category	Performance Model
Geometry	$[n_{on}(t_{geom} + t_{bucket}) + n_{back}t_{geom_back} + n_{off}t_{geom_off}]$
Geometry Overhead	$n_{on}(O-1)t_{bucket}$
Rasterization	$[n_{on}(t_{prim} + \bar{h}t_{line} + \bar{a}t_{pixel}) + t_{reg_ovhd} + At_{pixel_ovhd}]$
Rasterization Overhead	$[n_{on}(O-1)t_{prim} + n_{on}\bar{h}(\sqrt{O}-1)t_{line} + (r-1)t_{reg_ovhd}]$
Message Time	One step: $Nm_1(t_{msg_s} + t_{msg_r})$ Two step: $N(m_{1s} + m_{2s})t_{msg_s} + (m_{1r} + m_{2r})t_{msg_r}$
Byte Time	One step: $Nn_1s_{prim}(t_{byte_s} + t_{byte_r})$ Two step: $N[(n_{1s} + n_{2s})s_{prim}t_{msg_s} + (n_{1r} + n_{2r})s_{prim}t_{msg_r}]$
Forwarding	One step: 0 Two step: $Nn_{refmt}t_{refmt}$
Geometry Imbalance	$[n_{on}(t_{geom} + Ot_{bucket}) + n_{back}t_{geom_back} + n_{off}t_{geom_off}](1/U_G - 1)$
Rasterization Imbalance	$[n_{on}(Ot_{prim} + \bar{h}\sqrt{O}t_{line} + \bar{a}t_{pixel}) + rt_{reg_ovhd} + At_{pixel_ovhd}](1\ U_R - 1)$
Geometry Communication Imbalance	One step: $N(m_1t_{msg_s} + n_1s_{prim}t_{byte_s})(1/U_G - 1)$ Two step: $N(m_{1s}t_{msg_s} + n_{1s}s_{prim}t_{byte_s})(1/U_G - 1)$
Rasterization Communication Imbalance	One step: $N(m_1t_{msg_r} + n_1s_{prim}t_{byte_r})(1/U_R - 1)$ Two step: $N(m_{2r}t_{msg_r} + n_{2r}s_{prim}t_{byte_r})(1/U_R - 1)$

Table 7.1 Performance model divided into the different categories.

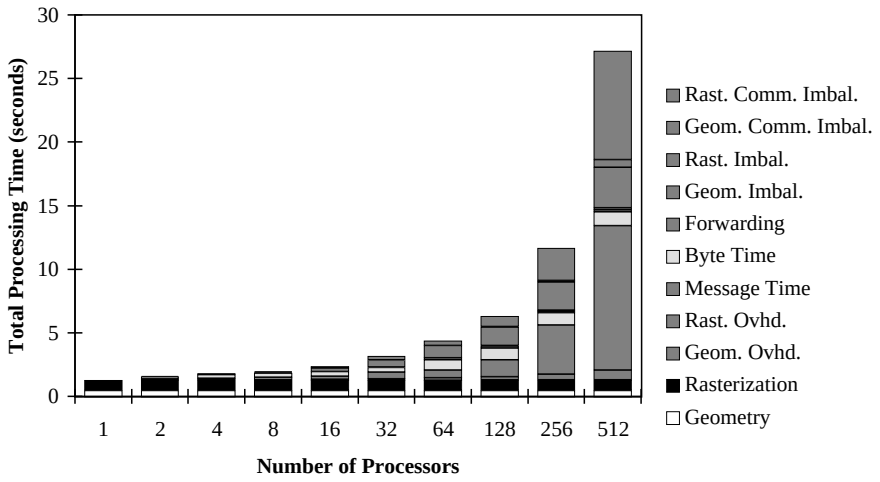


Figure 7.11 Aggregate time spent on different tasks for the PLB Head model.

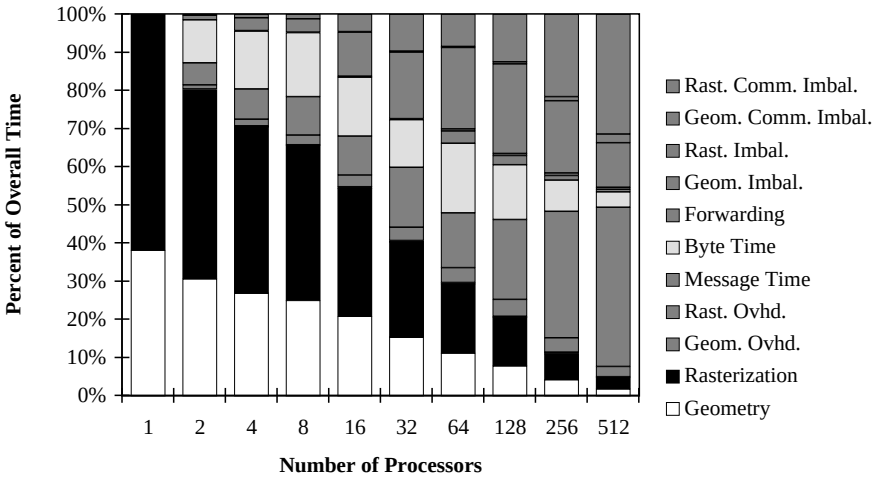


Figure 7.12 Time spent on different tasks for the PLB Head model, in percentages.

When using 512 processors, even with the largest model, Polio, the parallel efficiency is only 25%—only 25% of the overall time is spent performing the work that would be done when using a single processor. The efficiency is only 5% with the smallest model, the PLB Head. Of course, some of the parallel costs are unavoidable, such as the per-byte communication costs. Still, the portions of the algorithm that increase with *N* dominate the overall time of the algorithm when using many processors. Remember that these times do not include the load-balancing and communication bottlenecks described earlier as they are not reflected in the displayed performance model.

7.2.6 Anti-Aliasing Performance

The last component of the algorithm's performance is its speed when performing anti-aliasing. The triangle rendering rates are reduced when anti-aliasing is enabled, since it requires more work. The number of pixels to be rendered goes up by a factor of 16, and those samples must also be filtered down to the final value. Figure 7.15 shows the triangle rendering rates with anti-aliasing enabled, with all other settings at their defaults.

Figure 7.16 compares the point-sampled rendering rates and the anti-aliasing rates. The figure gives the anti-aliasing rendering rate as a the fraction of the point-sampled rate. The graph shows that the rendering rates converge as the number of processors increases. When using more processors, the amount of overhead increases while the serial rendering time remains constant, diluting the effect of the additional time

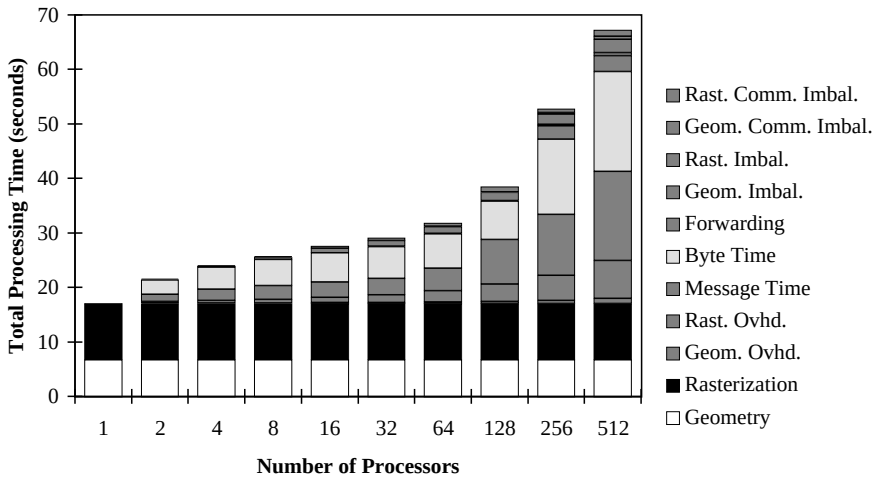


Figure 7.13 Aggregate time spent on different tasks for the Polio model.

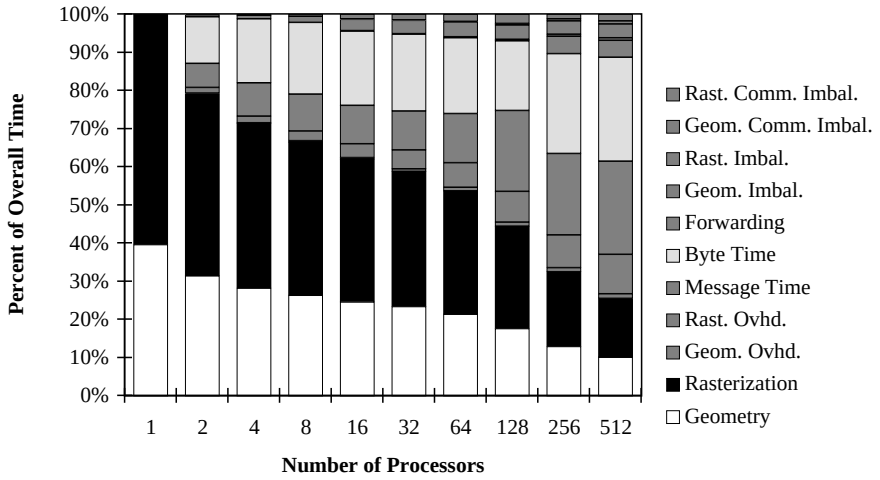


Figure 7.14 Time spent on different tasks for the Polio model, in percentages.

needed to perform anti-aliasing. When using 512 processors and the three smaller models, the load-balancing and communications bottlenecks hide the entire additional anti-aliasing cost.

7.3 Evaluation of the Design Choices

This section evaluates two design choices made in Chapters 4 and 6, the redistribution method and the granularity ratio. It also evaluates the maximum message size since that choice was deferred due to lack of evidence. The choices will be evaluated by varying the associated parameter during different runs of the implementation. The different runs also show the impact of each choice on the overall performance. The redistribution method is quite important; a poor choice makes the algorithm impractical on large systems. The granularity ratio has a smaller effect (20%) with small systems, but has larger effect with large systems because it affects the load-balancing bottleneck. The last choice, message size, makes at most a 15% difference in performance.

7.3.1 Redistribution Methods

The recommendations in Chapter 4 indicate that one-step stream-per-region redistribution should be used with a few processors, as it saves memory. With more processors, stream-per-processor redistribution should be used, with automatic selection between one-and two-step redistribution. The selection is made by evaluating the performance model for each method, and choosing the faster one.

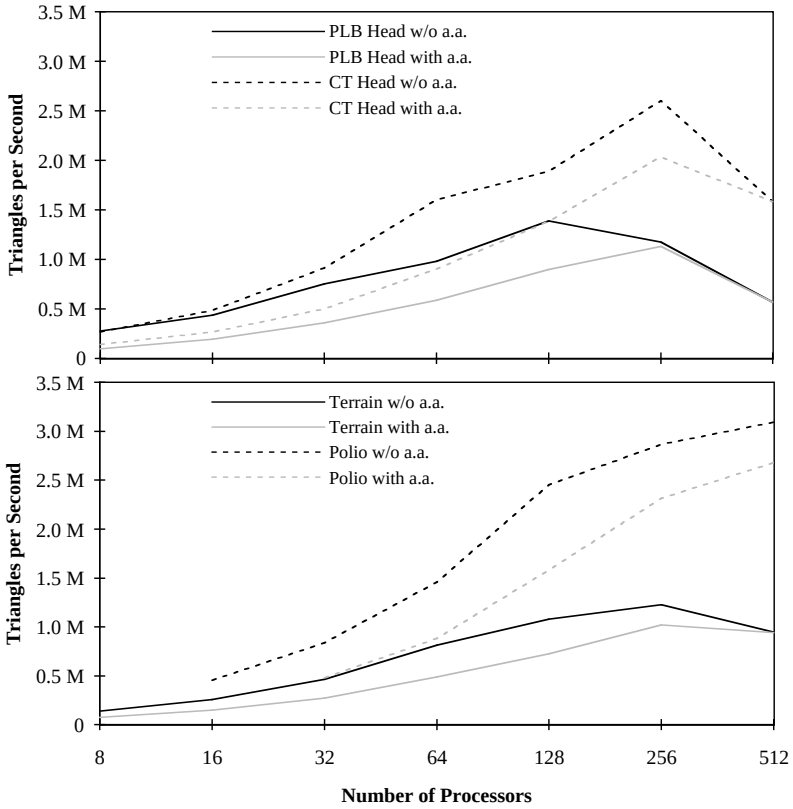


Figure 7.15 Speed when performing anti-aliasing ("with a.a.") compared with the non-anti-aliasing speed ("w/o a.a.").

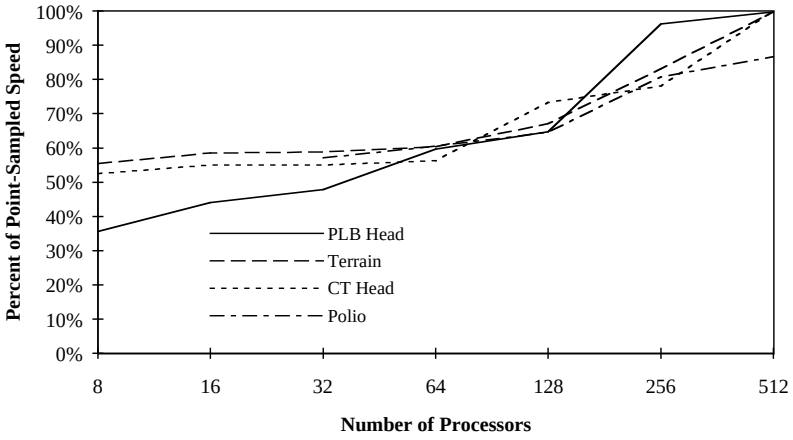


Figure 7.16 Rendering rate with anti-aliasing enabled as a fraction of the point-sampled rendering rate.

I ran the implementation using all four redistribution methods (with the automatic selection disabled) to confirm these choices. I made runs with two-step stream-per-region redistribution for completeness. Figures 7.17 and 7.18 show the results of the runs. I omitted runs using one-step stream-per-region redistribution on 256 and 512 processor systems because they took excessive amounts of time.

Redistribution using stream-per-region is nearly always slower than stream-per-processor redistribution. When using the PLB Head model, one-step redistribution, and one or two processors, stream-per-processor redistribution takes, respectively, 0.71 and 0.41% more time than stream-per-region redistribution. In all other cases, stream-per-processor redistribution was faster. However, because stream-per-processor redis-

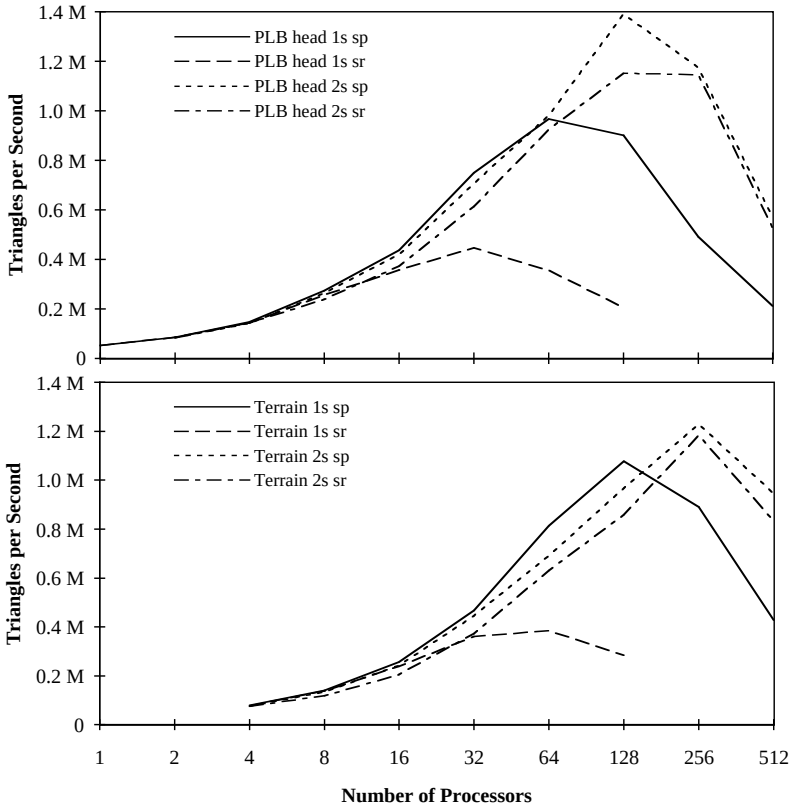


Figure 7.17 Speed with the PLB Head and Terrain models, for four different communication methods. Key: 1s = 1-step, 2s = 2-step, sr = stream-per-region, sp = stream-per-processor.

tribution does not efficiently allow rendering one region at a time (as discussed in section 4.4.1), copying to the frame buffer can only be done after all the regions have been rasterized. This means stream-per-processor redistribution can have more latency in a real implementation.

Using two-step redistribution improves the performance when using many processors. For the stream-per-processor runs, the maximum speedup of two-step versus one-step redistribution is 2.7. The automatic selection between the two methods is usually correct, but the selection does miss the crossover point. With the PLB Head model and 64 processors, two-step redistribution is automatically selected although one-step redistribution would be 0.17% faster. Two-step redistribution is used in two other cases even though it is slower: two-step redistribution with the CT Head model and 128 processors is 4.7% slower, and the Polio model with 256 processors is 13% slower. In the latter two cases, the likely explanation for the misprediction is that the performance model does not account for the increased network contention caused by sending the triangles a second time. The model would be closer if the network congestion (see section 7.2.2) was not reducing the implementation's performance. Overall, the automatic selection improves the algorithm performance over simply using one-step redistribution.

Table 7.2 shows the faster redistribution method for the different models and partition sizes, and also shows when the automatic selection fails.

7.3.2 Granularity Ratio

The next design choice to be evaluated is the granularity ratio, the average number of regions per processor. In Chapter 6 the granularity ratio was fixed at 6, which was a compromise (see tables 6.5–6.10). The optimum granularity ratio for the different models and number of processors ranged from 1 to 12 (the best load-balancing methods also varied).

I ran the program using the chosen granularity ratio of 6, plus nearby ratios of 4 and 8, with the four models and a range of partition sizes. The results are shown in figures 7.19 and 7.20. The first chart shows the raw

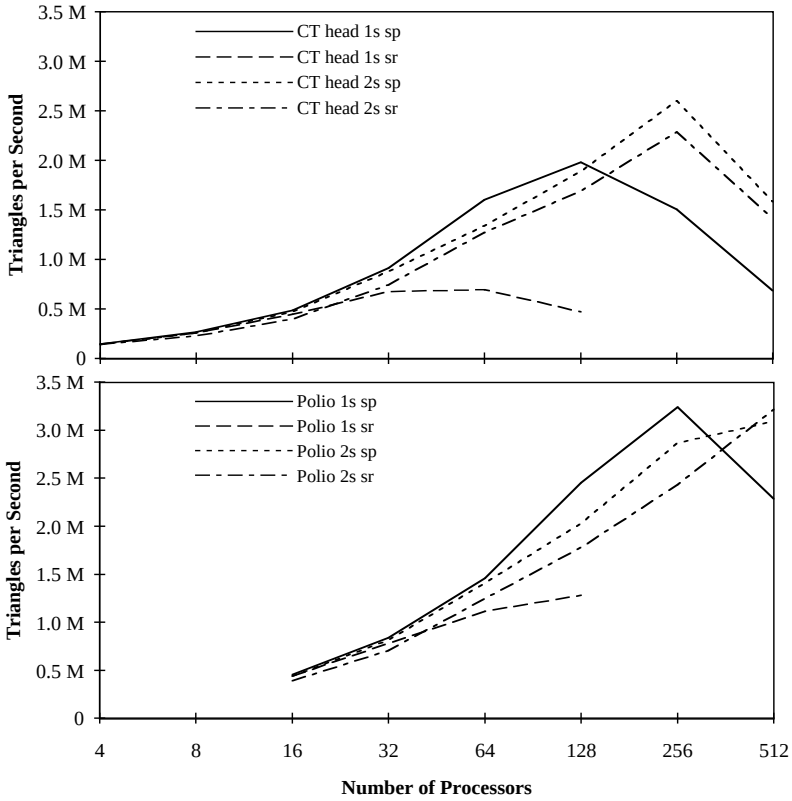


Figure 7.18 Speed with the CT Head and Polio models, for four different communication methods. Key: 1s = 1-step, 2s = 2-step, sr = stream-per-region, sp = stream-per-processor.

Model	Number of Processors									
	1	2	4	8	16	32	64	128	256	512
PLB Head	1 step	1 step	1 step	1 step	1 step	1 step	*2 step*	2 step	2 step	2 step
Terrain	1 step	1 step	1 step	1 step	1 step	1 step	1 step	1 step	2 step	2 step
CT Head	1 step	1 step	1 step	1 step	1 step	1 step	1 step	*2 step*	2 step	2 step
Polio	1 step	1 step	1 step	1 step	1 step	1 step	1 step	1 step	*2 step*	2 step

Table 7.2 The faster redistribution method used for the different models, for a range of partition sizes. Incorrect selections by the automatic selection algorithm are italicized.

rendering rates, and the second chart shows the normalized rendering rates: each data point is divided by the average of the three data points that correspond to the same model and number of processors.

For smaller numbers of processors, the best granularity ratio was 6 or 8. However, when using many processors, a granularity ratio of 4 worked the best—even though simulations predicted better performance with a higher granularity ratio. This is a result of the load-balancing bottleneck, which is reduced by a smaller granularity ratio. A smaller ratio results in fewer regions, which reduces the amount of data to be transferred when collecting per-region triangle counts, and reduces the number of regions to be assigned to processors. The results confirm that using $G = 6$ is still a reasonable compromise value.

We can also see how well the actual best granularity ratio matches the one predicted by the performance model, as shown in table 7.3. The predicted value is the best granularity ratio for the implemented algorithm, and includes time for load-balancing. The "actual" values are constrained because only granularity ratios of 4, 6, and 8 were run, while values of 1, 2, 4, 6, 8, 12, and 16 were simulated. Given the constraints, the granularity ratios match reasonably well. There are only small differences (less than 1%) in performance between using the best predicted ratio instead of the actual best ratio when using 4 to 16 pro-

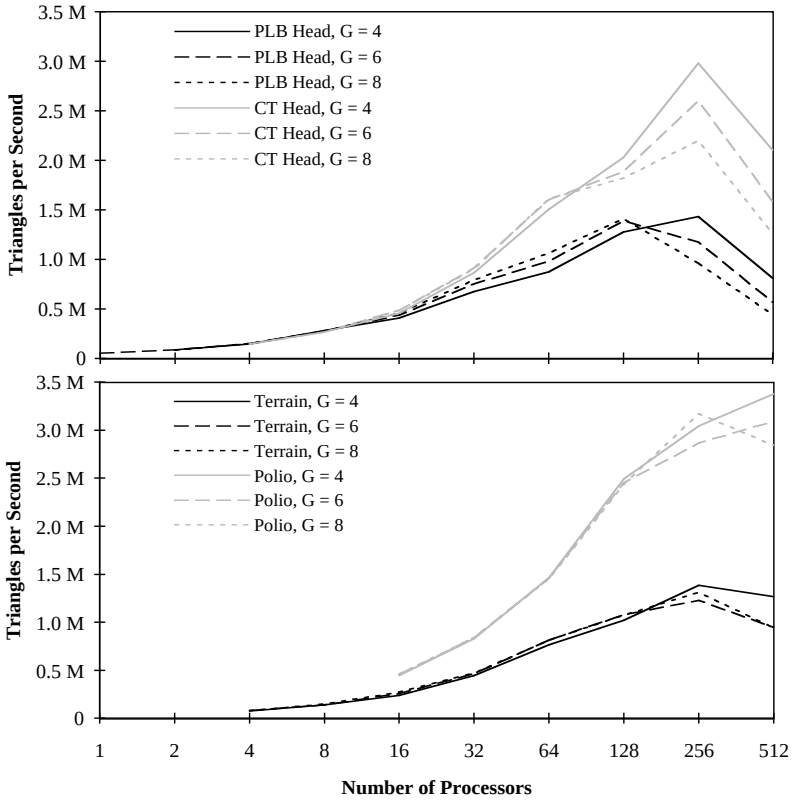


Figure 7.19 Average number of triangles per second for granularity ratios (G) of 4, 6, and 8.

cessors. The errors are larger when using more processors; the largest difference is 18% with the PLB Head model running with 256 processors.

7.3.3 Maximum Message Size

Unlike the earlier design choices, the analysis in the earlier chapters was not able to predict the impact of the maximum message size on the algorithm's performance. I ran the implementation using three different message sizes, 2 Kbytes, 4 Kbytes, and 8 Kbytes, to measure this effect. Each message size was tried using the four models and a range of partition sizes. The results are shown in figure 7.21; table 7.4 gives the fastest size in each case.

With a smaller number of processors, the performance increases with increased message size. When using few processors, the amount of data transferred between processors is larger than the 4 Kbyte message size used, so increasing the message size reduces the number of messages, and in turn reduces the overall per-message overhead. The speedup for using 8 Kbyte messages instead of 4 Kbyte messages is up to 15%, with an average of about 5%.

When using more than about 64 processors, it appears that other factors remove this advantage. One factor is that the number of groups used with two-step redistribution changes with the maximum message size, which changes the communication patterns, and can increase the network congestion. When using 512 processors, the Polio model, and 4 Kbyte messages, 48 groups are used, while 16 groups are used with 2 Kbyte and 8 Kbyte messages. Because 48 does not divide 512 evenly, intra-group traffic will cross the partition bisection, increasing the bisection bandwidth and reducing the overall speed. Also, using a smaller message size increases the portion of the frame time used for redistribution, and thus smaller message sizes work better with the larger models.

One somewhat surprising result is that the optimum message size is much larger than the average amount of data exchanged between processors. Table 7.5 gives the average amount of data sent between pairs of processors. The values were computed using a simplified analytic model compared to the formulas developed in Chapter 4: it assumes perfect load-balancing and that there are $\sqrt{N}$ groups. One possible cause of the

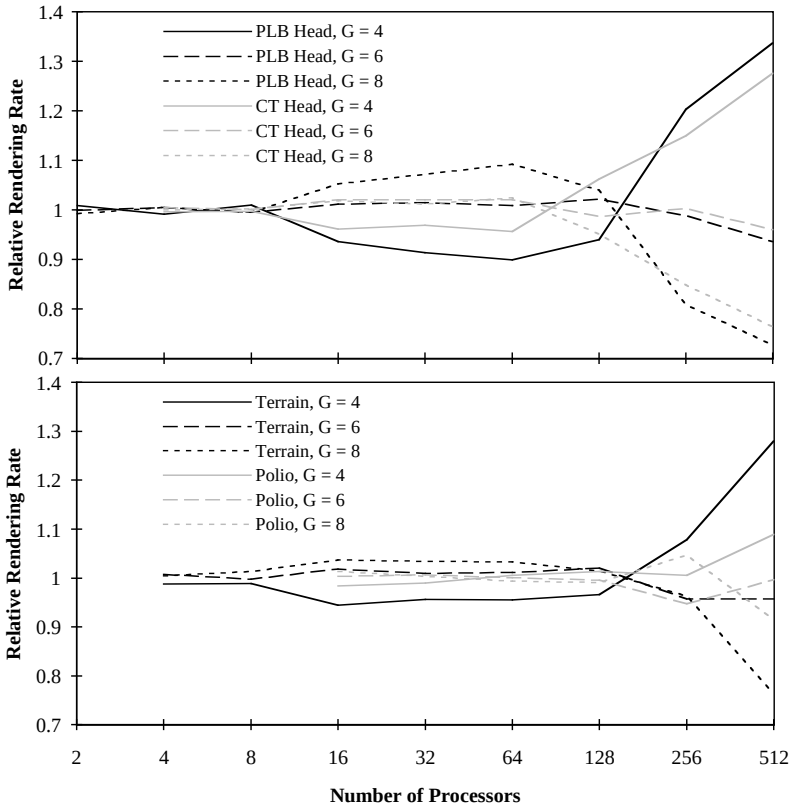


Figure 7.20 Normalized triangle rendering rates for granularity ratios of 4, 6, and 8.

Model	Number of Processors																	
	2		4		8		16		32		64		128		256		512	
	P	A	P	A	P	A	P	A	P	A	P	A	P	A	P	A	P	A
PLB Head	*1*	*4*	**8**	**6**	**6**	**4**	*12*	*8*	*16*	*8*	*12*	*8*	*12*	*8*	**6**	**4**	4	4
Terrain	2	-	**8**	**6**	**6**	**8**	8	8	8	8	*12*	*8*	6	6	**6**	**4**	4	4
CT Head	6	-	6	6	**6**	**8**	6	6	6	6	**6**	**8**	4	4	4	4	*2*	*4*
Polio	4	-	6	-	6	-	**8**	**6**	6	6	4	4	4	4	**4**	**8**	*2*	*4*

Table 7.3 Predicted (P) and actual (A) granularity ratios with the highest average triangle rate for the four models and a range of partition sizes. Differences are highlighted in bold if the predicted value was run, and in italic if it was not.

disparity is the random processor-to-processor variation in the number of triangles in each region. This variation causes the number of triangles exchanged between processors to vary. A second possible reason is that some processors are often assigned several nearly or completely empty regions. Since those processors have more per-region overhead, they are given fewer triangles than the average. Other processors will have more than the average number.

Using large messages with many processors will also increase the amount of time at the beginning of the frame when communication is not being done. This reduces the available communication bandwidth, which can be the bottleneck when rendering at high triangle rates. This is what seems to happen with the smaller data sets when running with more than 64 processors.

No single message size appears to be the best. Increasing the message size will increase the performance when using less than 64 processors, but not by a significant amount. So, I will keep the message size at 4Kbytes.

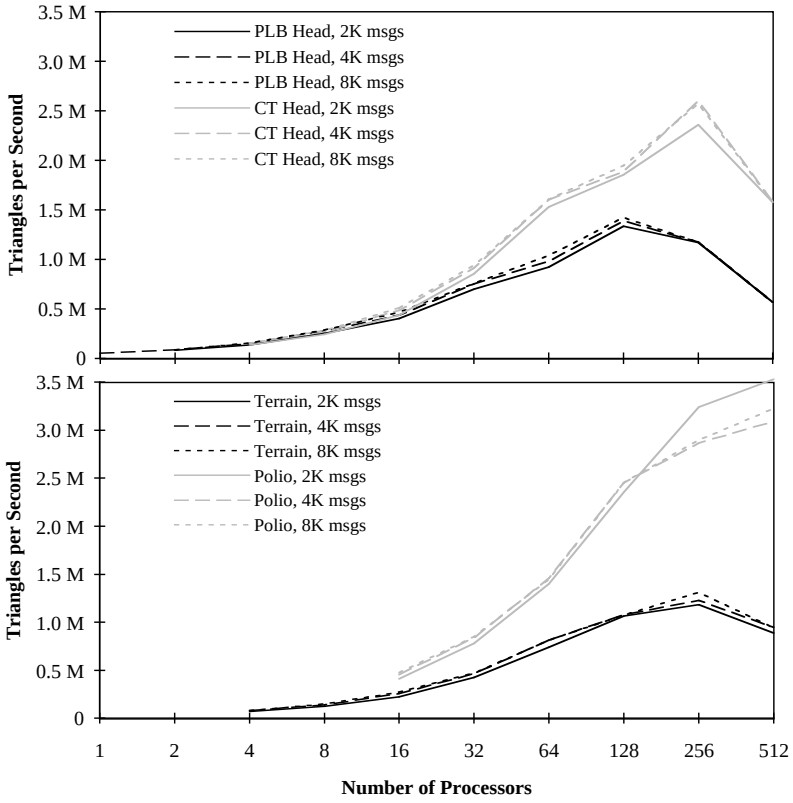


Figure 7.21 Implementation performance as the maximum message size is varied.

Model	Number of Processors								
	2	4	8	16	32	64	128	256	512
PLB Head	8	8	8	8	8	8	8	4	4
Terrain	-	8	8	8	8	8	8	8	4
CT Head	-	8	8	8	8	8	8	4	4
Polio	-	-	-	8	8	4	8	2	2

Table 7.4 Maximum message size, in kilobytes, which produces the highest average triangle rate.

Model	Number of Processors								
	2	4	8	16	32	64	128	256	512
PLB Head	375 K	96,041	24,682	6,367	1,677	3,576	1,308	551	227
Terrain	1795 K	456 K	116 K	29,825	7,734	2,032	547	2,398	950
CT Head	1424 K	362 K	92,583	23,906	6,232	1,650	4,787	1,993	810
Polio	5506 K	1401 K	360 K	93,228	24,514	6,561	1,809	8,178	3,382

Table 7.5 Number of bytes exchanged between processors (averages). The values do not decrease monotonically because both one- and two-step redistribution are used.

7.4 Summary and Discussion

The implementation delivers high performance, up to 3.1 million triangles per second when using 512 processors and the Polio model. However, it does not scale beyond 256 processors with smaller models. Part of the scaling problem is due to the increase in overhead as the number of processors increases, which is inherent in the algorithm. Other parts of the scaling problem could be viewed as limitations of the underly-

ing system. If the network was faster, the network congestion would be reduced. More recent systems have much more communication bandwidth than the Touchstone Delta. For example, the Intel Paragon [Inte92] has a configuration similar to the Delta's, but has a quoted peak link bandwidth of 200 Mbytes/sec. This rate is 20 times the measured link speed on the Delta. The increased link speed would not appreciably change the main problem, the load-balancing bottleneck. It can be dealt with by using the static load-balancing method instead of the fixed-region between-frames method when evaluating a performance model indicates that it would work better.

The scaling problem is emphasized because the serial part of the algorithm is highly optimized. If the rendering code was written in C instead of assembly language it would take 50 to 100% longer. With the lower rendering rates, the load-balancing bottleneck and the network congestion would not exist.

The performance measurements presented in this chapter match well with the actual performance except when the network congestion limits the performance. The performance model also matches the performance when the communication method and granularity ratio are varied, which confirms the predications made with the performance model.

The best maximum message size could be improved. The single 4 Kbyte value is reasonable; however, varying the message size as a function of the model size and the number of processors would give a slight performance boost. The function to determine the message size could be determined experimentally. One possible function would be to have the message size be a fraction (say, $^1/_4$ or $^1/_3$) of the total expected amount of data that a processor will send to another processor during a frame. A minimum message size (perhaps 2 K or 4 Kbytes) would be needed to make sure that the per-message overhead remains a reasonable fraction of the overall communication cost. Or, if the performance model was enhanced to include limitations in network bandwidth, it could be used to choose the best message size.

While the Touchstone Delta implementation could be improved in several ways, it does substantiate the results of the earlier chapters. Since the measured performance is close to the predicted performance, it shows that the performance model is realistic. Also, the implementation's high measured performance shows that the design methodology described in the earlier chapters can be used to choose a high performance polygon rendering algorithm.

CHAPTER EIGHT

USING SPECIALIZED RASTERIZATION HARDWARE

Up to this point, we have discussed polygon rendering running entirely on general-purpose processors. We will change direction in this chapter and consider the use of specialized rasterization hardware for polygon rendering on multicomputers, where a portion of a multicomputer's general-purpose processors is replaced with specialized rasterization processors. The chapter will focus on the design of parallel rendering algorithms for such systems, but the topics also affect hardware design. The discussion will not consider all the algorithm options as was done for general-purpose systems in the earlier chapters. Instead, the first section will discuss how using specialized hardware changes the earlier algorithm design decisions. The different design decisions will then be illustrated by a case study of a system that uses specialized hardware, Pixel-Planes 5 [Fuch89]. Two different algorithms for the system will be described, one that makes work-queue region assignments and another that uses static assignments. The first algorithm will be analyzed by characterizing its performance and comparing it to the Delta implementation. A later section will give evidence for the main motivation for using specialized rasterization processors: they give higher performance compared to general-purpose processors with the same cost. That section will examine the Pixel-Planes 5 processors to show that specialized processors can give over three times the rasterization performance for the same cost. The last sections close the chapter with future work and a summary.

While we will show that specialized hardware is more cost-effective than general-purpose processors for rasterization, general-purpose processors are still cost-effective for geometry processing. Geometry processing is similar to scientific calculations since both are floating-point intensive. Several commercial microprocessors are designed to work well on scientific calculations and thus have excellent floating point performance, such as the older Intel i860 as well as the recent implementations of the Hewlett-Packard Precision Architecture, PowerPC, and Digital Alpha architectures. Digital signal processing microprocessors are another option, as they also have good floating point performance. Since general-purpose processors are cost-effective, there is little reason to consider specialized hardware for geometry processing.

Systems that use specialized rasterization hardware are heterogeneous systems: general-purpose processors run the user's application and perform the geometry processing, and specialized processors perform the rasterization. We will call the general-purpose processors *geometry processors* and the specialized rasterization processors *rasterizers*.

8.1 Parallel Rendering using Hardware Rasterization

The earlier chapters analyzed different design options for parallel rendering algorithms to be used with general-purpose processors. This section considers how those analyses change when specialized rasterization hardware is used. The first subsection describes how heterogeneous systems constrain rendering algorithms and can introduce some inefficiencies. The next subsection explains why specialized rasterizers can require work-queue region assignment even though that load-balancing method can become very expensive (as was discussed in Chapter 4). The following subsection describes how hardware rasterizers may need to have fixed region sizes, and how this can limit the size of a system. The last subsection describes how the use of faster processors reduces the amount of overhead and how that affects the best parallel algorithm.

8.1.1 Heterogeneous Processing

Systems with specialized rasterizers have an additional goal over ones with a single processor type: tasks should be assigned to both the geometry processors and rasterizers so as to minimize the frame time. This involves properly scheduling the tasks within a frame so that the rasterizers don't have to wait for data from

the geometry processors, and it also requires balancing the loads between the two processor sets. The tasks can be scheduled by having the sets of processors work either on the same frame or on successive frames. If the processors work on the same frame, the region assignments for rasterization must be fixed during the frame. Furthermore, having the rasterizers finish one region before starting on the next is ruled out, as a region can only be finished after all the triangles have been transformed. Because multiple display triangles are usually collected and sent in one message, it is desirable to have the front-end processors start the next frame before the rasterization is complete; otherwise only one set of processors will be working at the start and end of frames.

Having the sets of processors work on successive frames removes the above restrictions, but at the cost of additional latency. The latency can be reduced by partially overlapping the frames. In this method, the geometry processors split the geometry processing into two parts: the processing up through bucketization, and any later processing. Initially, the processors only do the first part of the geometry processing for all the triangles. The processors then do the second part of the geometry processing in the same order as the regions are assigned, keeping just ahead of the rasterizers.

It is impossible to always balance the work perfectly between the general-purpose and specialized processors. The sizes of the sets of processor are fixed while the work for each set will vary as the model and the point of view changes. On the other hand, when using only general-purpose processors you can change the partitioning of processors between those doing geometry processing and those doing rasterization. Another option is to have all the processors perform both tasks, as has been described in earlier chapters. However, using all the processors for both tasks increases the amount of overhead since the tasks must be partitioned among more processors.

8.1.2 Work-Queue Region Assignments

The capabilities of a system's specialized rasterizers can constrain how regions can be assigned to the rasterizers. Only some systems allow the static, between-frames, and between-stages region assignment styles. Regions must be assigned to rasterizers using a work queue if a system's rasterizers can only accept triangles for one region at a time. As discussed in section 4.3, this is much more expensive than assigning regions once per frame, and will limit the maximum size of the system.

Two types of rasterizer designs will allow them to accept triangles for any of the regions assigned to them, avoiding work-queue region assignments. In one type of design, the rasterizers instantiate all the region buffers during the entire rasterization, and then process triangles without regard to their screen locations. In the second type of design, the rasterizers have buffering to hold an entire frame's triangles. They process their regions one at a time while triangles for the following frame arrive and are placed in separate buffers.

If a rasterizer must handle one region at a time and does not have buffering for triangles, work-queue region assignments are necessary. When a region assignment is made, it must be sent to all the geometry processors so each can send triangles overlapping the region to the appropriate rasterizer. The region assignment can be sent to a single geometry processor at a time, to all processors at once, or to k processors at a time.

In the first method, the region assignment notification is sent to a single geometry processor, which sends its triangles to the specified rasterizer and then sends the notification message to another geometry processor. This continues until all the processors have sent their triangles, after which the rasterizer is assigned a new region. If there are multiple rasterizers, then there will be multiple assignment-notification messages circulating at once. At times a single geometry processor will have more than one assignment notification. This means that it is responsible for sending enough triangles to keep multiple rasterizers busy. A geometry processor could have too many notification messages, causing some rasterizers to be starved for triangles. This problem can be ameliorated by adding buffering to the rasterizers. A second problem occurs when running with many geometry processors and a high frame rate. In this case, each processor will only send a small number of triangles when it receives a notification message. The rasterizers will be starved for data if the time to process a notification message takes too long, as the rasterizers will exhaust the buffered triangles before new ones arrive.

In the second method, the region assignments are broadcast to all the processors. This method does not suffer from the problems with message latency and with utilization messages piling up on a single processor, but it has problems when the communication network is not fair. During rasterization, the N_G geometry processors are sending to the N_R rasterizers. Hot spots in an unfair network could starve some rasteriz-

ers for triangles. Buffering on the rasterizers can reduce this problem. However, the buffering will not help when a frame is started, since all the rasterizers are then waiting for data, unless the rasterization for a new frame is started before the previous frame completes.

The two methods can be generalized into sending k assignment messages per rasterization processor. The first method corresponds to $k = 1$, and the second to $k = N_G$. Intermediate values of k balance the tradeoffs between the two extremes. Larger values of k may have a higher cost. If the rasterizers cannot keep count of the geometry processors that have sent all of their triangles, then the geometry processors will have to send additional messages to a designated processor to indicate that they have sent all of their triangles to the rasterizer. The number of messages is equal to k, as each of the k assignment messages must be sent to the designated processor. When $k = N_G$, the factor of two increase in the number of assignment messages will make a noticeable difference if the cost to send and receive assignment messages is already high. For example, in the Pixel-Planes 5 system described later in the chapter, when generating 1280 by 1024 pixel images at 24 frames per second each geometry processor spends 20% of its time sending and receiving assignment messages with $k = 1$. Doubling this cost would be unacceptable. Whether or not a factor of two increase is important depends on the number of regions, the cost of sending and receiving messages, and the desired frame rate.

The final cost of work-queue region assignments is the cost of performing the assignments. Having one processor make the assignments is the simplest method, but will be a bottleneck with large systems, and will also create a load imbalance for that processor. The assignments could be parallelized by taking the previous frames per-region workload estimates and making the assignments during the previous frame's rasterization. These "pre-assignments" for each rasterizer are then sent to different geometry processors, which make the real-time assignment of regions for that rasterizer, one region at time. The last regions to be rendered in a frame must be assigned by a single geometry processor to allow errors in the region pre-assignment to be corrected.

In summary, the cost of work-queue region assignment should be carefully considered before choosing to use it. Current multicomputers have fairly high per-message costs, which means that work-queue region assignments are impractical when using more than about 100 processors at high frame rates. The system size where work-queue region assignments become too expensive depends on the cost to send and receive messages and the desired frame rate.

8.1.3 Fixed Region Sizes

Design considerations can constrain the size of the regions. Systems that use arrays of processors, such as processor-per-pixel or processor-per-column systems, have a natural region size: the size of the array. Other systems use *virtual buffers* [Ghar89], where each processor has a small amount of fast memory. The faster memory gives the processors higher performance than conventional memory (such as dynamic memory). The small size of the fast memory can constrain the maximum region size. Also, the interface between the virtual buffer and the processor as well as the interface between the virtual buffer and the frame buffer may be simplified by using fixed region sizes. Pixel-Planes 5, discussed in the next section, uses fixed region sizes.

Small versions of systems with fixed region sizes work well. In small systems, the fixed region sizes can result in a larger overlap factor than ones that use dynamically-sized regions, but this cost can be allowed for. The large granularity ratio (ratio of regions to processors) in small systems keeps the processor utilization high.

The granularity ratio decreases when more rasterizers are added, decreasing the processor utilization, and thus limiting the achievable speedup. Figure 8.1 shows the result of scaling a system with fixed region sizes. The chart shows the results of the simulator described in Chapter 6 configured to simulate Pixel-Planes 5. The simulations used 128 x 128 regions and screen sizes of both 1280x1024 and 640x512. The simulator used a time of 6.8 μs to rasterize each triangle, and 600 μs for per-region overhead (no communication time is required for because it is overlapped with the triangle rasterization). Both values are from published sources [Fuch89, Tebb90]. The simulator does not model the non-deterministic effects due to network or other delays, but the work-queue region-assignment style would compensate for those effects in a real system. The overlap factor is explicitly modeled.

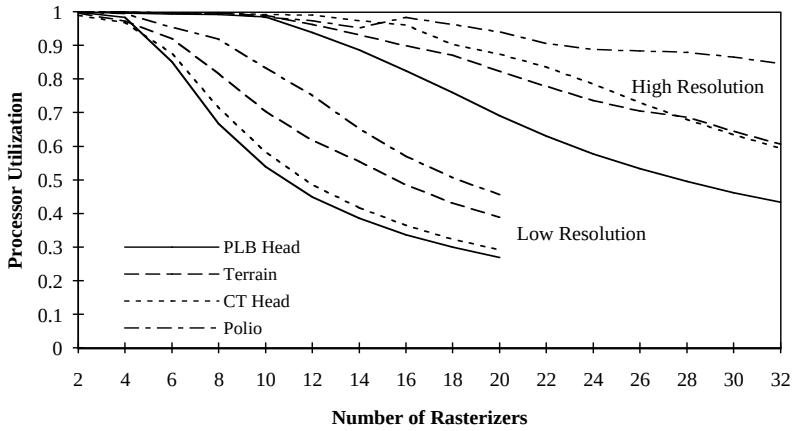


Figure 8.1 Rasterization processor utilization predicted by simulation for 128 by 128 pixel regions for a range of number of processors, and for the four models. The top four curves show the utilization for a high resolution image (1280 by 1024, or 80 regions). The bottom four curves show the resolution for a low resolution image (640 by 512, or 20 regions).

Figure 8.1 shows that the PLB Head processor utilization is below 80% when using more than 6 processors when computing a low resolution image, and when using more than 16 processors when computing high-resolution images. The main reason for the low utilization is that one region is a bottleneck: it takes so long that it determines the overall rasterization time. Three different methods could be used to increase the processor utilization. They are briefly described below, and are described in more detail in section 8.5.

One method uses image composition, like sort-last. Two or more processors are assigned to the bottleneck regions, each rasterizing a fraction of the triangles. After the processors have rasterized their triangles, they send the computed pixels and depth values to a common processor where they are composited to form the final region values. The additional step may increase the latency of the algorithm, and incurs additional communication and composition costs. However, the method will usually only be used for a small fraction of the regions, and thus the costs may be reasonable.

The second method is to rasterize more than one frame at a time. Rasterizers are first assigned regions from one frame, and then are assigned regions from the next frame without waiting for the earlier frame to be completely rasterized. This method does nothing to decrease each frame's rasterization time but does increase the processor utilization. The increased utilization will increase the frame rate and will decrease the overall latency if the rasterization bottleneck had caused results from earlier stages in the pipeline to wait. Compared to the other two methods, this method has higher latency, which may make user interaction difficult.

A third method is to subdivide the bottleneck regions, disregarding the processor's natural region size. Here, each processor is sent triangles that cover only the subdivided region. The processors still compute the pixels for the natural region size, but the pixels outside the actual region are discarded. This method requires a more complex bucketization method, and that the system have the flexibility to discard the extra pixels.

The best method depends on the characteristics of the actual implementation. Each method should be considered when the region size is determined. In general, the largest system desired constrains the design the most since it requires the largest amount of parallelism.

8.1.4 Effect of Faster Processors

The higher speed of specialized rasterizers has a side benefit. For a given performance level, fewer specialized processors are needed when compared to general-purpose processors. The benefit is that the smaller number of processors will be used more efficiently: they will be kept busy a larger fraction of the time, and fewer triangles will need to be processed more than once. This effect can be seen in the charts in section 6.3 and Appendix C.

8.1.5 Summary

This section has touched on the major differences between parallel algorithms suited for general-purpose processors and specialized rasterization processors. It is hard to make specific recommendations since there are many different specialized rasterization architectures, and because technology improvements can change the costs involved. Designers considering specialized rasterization processors can use the data in the appendices and performance models of their proposed systems to choose their best algorithm options.

8.2 PPHIGS Implementation on Pixel-Planes 5

This section illustrates the consequences of using specialized rasterization hardware by describing one system as a case study. The system is Pixel-Planes 5, a heterogeneous multicomputer with specialized rasterization processors [Fuch89]. It is a high-performance system: when it was introduced, its value for the Graphics Performance Characterization Group's "head" benchmark was nearly four times higher than the value for the fastest commercial system (faster systems have since been introduced). This benchmark uses the same model as the PLB Head model used in earlier chapters but computes a different size image and has multiple light sources.

The system was developed by a fairly large team, which was headed by Henry Fuchs and John Poulton, and included John Austin, Mike Bajura, Andrew Bell, Jeff Butterworth, John Eyles, Howard Good, Trey Greer, Edward Hill, Victoria Interrante, Anselmo Lastra, Jonathan Leech, Steven Molnar, Carl Mueller, Ulrich Neumann, Marc Olano, Mark Parris, John Rhoades, Andrei State, Brice Tebbs, Greg Turk, Laura Weaver, and myself. The next few paragraphs describe the hardware configuration, and are followed by a description of the polygon rendering algorithm in day to day use, which uses work-queue region assignments. The algorithm is part of a graphics library called PPHIGS, for Pixel-Planes Hierarchical Interactive Graphics System [Ells90b]. (A later section, section 8.3, describes a second polygon rendering algorithm, called MEGA, that uses static region assignment.) The system hardware is first described in section 8.2.1, which is followed by the rendering algorithm in section 8.2.2. An analysis of the system's performance follows in section 8.2.3. This includes characterizing the system's performance over a wide variety of system sizes and investigating the best mix of geometry processors and rasterizers.

8.2.1 Hardware Configuration

Pixel-Planes 5 consists of a series of boards plugged into an interconnection network. There are 5 main board types: the Intel i860 Graphics Processors; the specialized rasterizers, called renderers; two types of frame buffers; and the host interface. The ring network will be described first, and followed by the different board types.

Ring Network. The ring network has an aggregate bandwidth of 640 Mbytes/sec. It is similar to a token-ring local area network except it has higher performance, and the bandwidth is split into 8 independent channels of 80 Mbytes/sec. The bandwidth is split into independent channels to match the send and receive capabilities of the individual processors. Access to the network is round-robin allocation of channels to processors, giving each processor equal access to bandwidth. However, access among all processors to a given destination is not fair. The first requesting processor downstream of the destination is always given access. This protocol can cause starvation if some destinations are very popular.

Graphics Processors. The general-purpose processors (called GPs) are built around 40 MHz Intel i860XR processors. Each processor has 8 Mbytes of local memory and one port to the ring network. A processor can sustain sending at the full 80 Mbytes/sec bandwidth, and receiving at approximately 50 Mbytes/sec.

Renderers. The renderers are 128x128 SIMD arrays of 1-bit processors. Each processor runs at 40 MHz, and has 208 bits of local memory and 4096 bits of off-chip "backing store" memory. Access to backing store memory requires a 102 μs transfer cycle, which transfers a 32-bit word for every processor. The peak processing speed of each board is quite high compared to conventional processors: each processor can perform a 32-bit integer addition in about 100 clock cycles, which gives a per-processor rate of 400,000 integer additions per second and a per-board rate of 6.6 billion additions per second. Each renderer has two ports on the ring. One port accepts commands for the SIMD array, and the other port sends and receives pixel data.

Unlike most SIMD processors, there is no communication between processors, since it is not necessary for the rasterization process. Also, the renderer array controller is much simpler than ones used with conven-

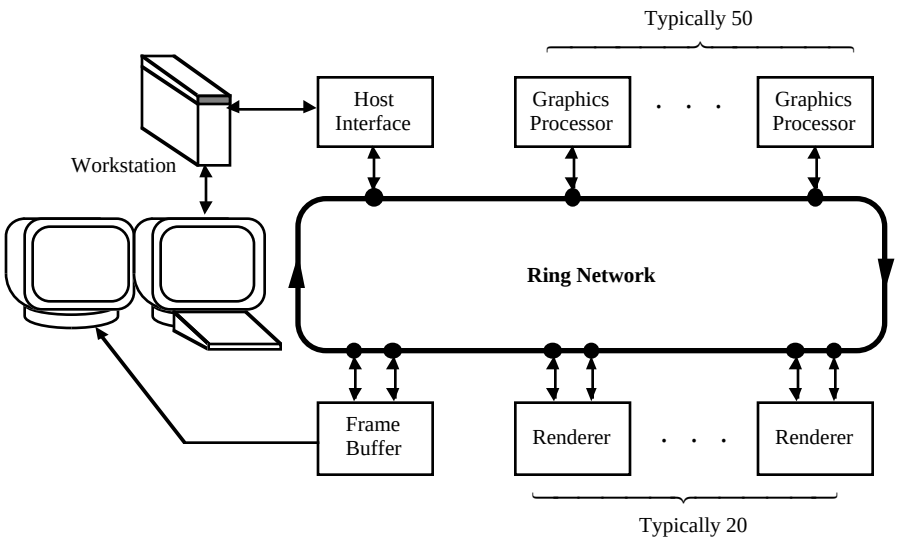


Figure 8.2 Block diagram of the Pixel-Planes 5 system. The number of graphics processors and renderers vary according to the system size. The figure shows typical processor counts for a three-rack system, the largest possible.

tional SIMD arrays: the controller only processes straight-line instructions. This was done because majority of a renderer's work is rasterizing primitives, which do not use any control structures. Any necessary control structures can be synthesized by the GPs.

The renderers' processors have a key feature for accelerating rendering, introduced with Pixel-Planes 1 in 1981 [Fuch81]. The feature allows them to quickly calculate linear expressions in screen space, expressions having the form $Ax + By + C$, where x and y are screen coordinates, and A, B, and C are fixed-point constants. The low bits of x and y are set by a processor's location within the array while the high order bits are set to position the renderer at different screen locations. Screen-space linear expressions are very common in graphics: they can be used for calculating the edges of polygons, and interpolating values such as z, color components, and normal vectors. The hardware for linear expressions speeds up the evaluation substantially. For an 8-bit value, using the linear expression evaluator is over 60 times faster than performing the multiplications and additions with the processors' ALUs.

Frame Buffers. The Pixel-Planes system has three frame-buffer board types. The high-resolution frame-buffer board can display a 1280 by 1024 pixel image, and can be updated at 24 Hz. It can also operate in a lower resolution mode, displaying a 640 by 512 image; when in this mode, the image can be updated at 72 Hz. The other two frame-buffer board types only support 640 by 512 images; they can be updated at 60 Hz, and have two buffers on a single board. One of these boards outputs NTSC-compatible signals, and is primarily used to drive head-mounted displays. The other board outputs a special signal to drive a head-mounted display that uses field-sequential color CRTs. The typical system configuration has several frame-buffer boards of different types.

Host Interface. The system also has a host workstation, a Sun 4/280. It has an interface allowing it to send messages to and receive messages from the ring.

8.2.2 Pixel-Planes 5 Rendering Algorithm

The rendering tasks are divided among the processors as follows. The host runs the application and manages the retained display list using a PHIGS-like API [Ells90b]. One GP, called the master graphics processor (MGP), is reserved to manage the algorithm. The other GPs handle the front-end calculations: transforming vertices, clipping, transforming normals, and calculating the linear expression constants. These are called slave graphics processors (SGPs). The renderers rasterize the primitives and perform the shading. They use deferred shading [Tebb90] to efficiently calculate procedural textures and Phong shading.

The standard rendering algorithm takes the following steps to generate a frame:

1. The host runs the application, calculating changes in the display list and viewpoint for the next frame. Changes in the display list are distributed across the SGPs using the simple distribution algorithm described in section 5.1 (in the first frame, the entire model is distributed). The host sends changes to the SGPs, where they are buffered.

2. When the host receives a command to display a new frame, it relays the command to the MGP, which informs the SGPs. The SGPs first process their display list changes, and then traverse their part of the display list. During traversal, primitives are transformed to screen space, clipped, and their normals transformed to eye space. Then, they generate the commands that the renderers will execute to rasterize the primitives. The commands are bucketized according to the regions that the associated primitive overlaps and are stored into 4 Kbyte message buffers.

3. After a SGP has traversed its display list, it sends the sizes of its buckets to the master graphics processor (MGP). The total size of a bucket is used as an estimate of the time a region will take to rasterize. The MGP sums the per-region bucket sizes and then sorts them.

4. The MGP assigns the region with the largest bucket size to a renderer, and sends commands to the render to configure its linear expression evaluation tree for the new screen location. The information about the assignment is placed into a message called a *rendering token*. Because a SGP can only send data fast enough to keep up with two or three renderers, the tokens are sent to different SGPs. Each token is sent to the SGP numbered $R(N_G - 1)/N_R$, where R is the number of the renderer being assigned. The MGP assigns regions to other free renderers, assigning them from most to least time consuming, and sends a token for each assignment.

5. The first SGP to receive a rendering token for a region computes and then sends commands to initialize the renderer. The commands clear out the color, *z*, and other memory locations. Next, the SGP sends its bucket of rendering commands to the renderer, which causes the renderer to rasterize that SGP's primitives that are inside the region. Once the commands have been sent, the SGP updates the token to indicate that it has sent its commands, and sends the token to the next SGP (the one with the next higher number, wrapping around to SGP 0 if necessary).

6. The next SGP to receive a token sends its bucket of commands to the renderer, updates the token to indicate that it has sent its commands, and then forwards the token on to the following SGP. This continues until the last SGP gets the token.

7. The last SGP to get a token for a region does additional processing to finish the region's rasterization. Like the other SGPs, it first sends its bucket of commands to the renderer. Then, it computes commands to shade all the pixels in the region and sends these commands to the renderer (remember that Pixel-Planes 5's standard rendering library uses deferred shading). The commands compute texture values (when textures are used), re-normalize the interpolated normal vectors, and perform Phong shading. Then, the SGP sends commands to the current renderer to send the values to the frame buffer via the backing store, and then to send a message to the MGP. This message indicates that the current region's pixels have been sent to the frame buffer. Finally, the SGP sends the rendering token back to the MGP, which indicates that the renderer has received all the commands for the region and can be assigned to the next one. The transfer of pixels to the frame buffer can proceed in parallel with the next region's rasterization, since the pixel transfer is done by the renderer backing store port.

8. When the MGP receives a token from a SGP, it reassigns the renderer as described in step 4 if there are more regions to be rendered. The MGP increments a counter when it receives a message from a renderer indicating that a region has been copied to the frame buffer.

9. When all the regions have been copied to the frame buffer, the MGP sends a command to the frame buffer instructing it to swap display buffers. The frame buffer will send a message back to the MGP after it swaps buffers.

8.2.2.1 Pipelining the Algorithm

The system could wait for step 9 to complete before starting again with step 1. However, that would leave much of the system idle during any single step. During editing (step 1), the host would be busy, the SGPs

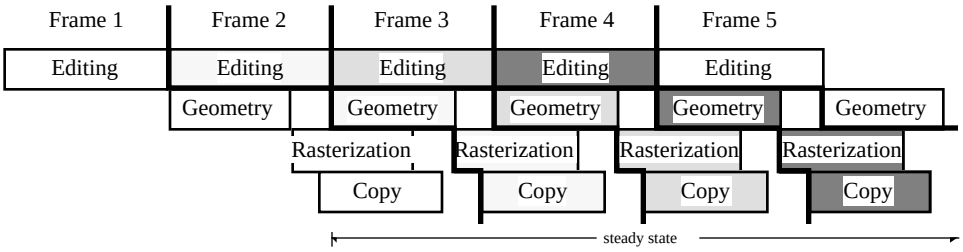


Figure 8.3 Pixel-Planes 5 pipelining with an editing stage bottleneck. The stages are shaded to indicate the frame number.

nearly idle, and the renderers idle. During geometry processing (steps 2 and 3), only the SGPs would be busy. The renderers would be busy and the SGPs nearly idle during rasterization (steps 4 to 8) Finally, only the renderer backing-store ports and the frame buffer would be busy while the last pixels are copied to the frame buffer and the buffers swapped (step 9).

Only one part of the system is busy during each of the four major pipeline stages: editing, geometry processing, rasterization, and pixel copy. In the actual software implementation, the four stages are overlapped, with each stage processing a different frame. Each of the four stages could be the bottleneck. Each causes a different pipeline organization, which are shown in figures 8.3 to 8.7.

Figure 8.3 shows one way the four stages can overlap. Here, the editing stage takes the longest, and is the bottleneck. The other stages each take the same, smaller amount of time. Because the editing stage is the bottleneck and does not need to wait for an earlier stage to complete, the host can start the next frame as soon as it completes the previous one. A new frame's geometry and rasterization stages can be started as soon as the earlier stage finishes. The copy stage can be started when the first region has been completed.

When the geometry stage is the bottleneck, the steady-state pipeline has the next frame's editing stage start after current frame's geometry stage starts (see figure 8.4). At the start, the stages are like the previous case. The pipeline reaches the steady state at the start of frame four.

Note that the next editing stage for frame three could start earlier, but doing so would only increase the latency, not the throughput (assuming the time to complete each stage is constant). The latency would be reduced if the next editing stage started a short time after the current geometry stage starts. The time would be chosen so that the editing stage completes at the same time as the geometry. However, this would require predicting how long the editing and geometry stages will take. Instead, to keep the throughput high, the next editing stage starts as soon as the geometry stage starts. The next geometry stage starts as soon as the earlier one completes, since it is assumed to be the bottleneck. The rasterization and copy stages follow the geometry stage as described in the previous case.

The third pipeline configuration has the third stage, rasterization, as the bottleneck. For the first three frames, the editing and geometry stages run at full speed. On the fourth frame the rasterization bottleneck causes the editing stage to wait. It is waiting for the third frame's geometry stage to start, which in turn is waiting for the second frame's rasterization stage to start. As with the previous bottleneck, the editing and

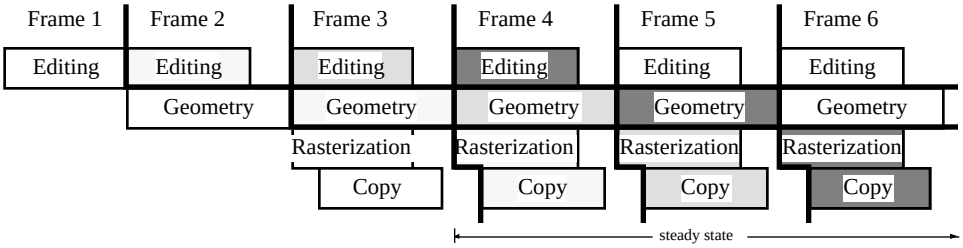


Figure 8.4 Pixel-Planes 5 pipelining with a geometry stage bottleneck.

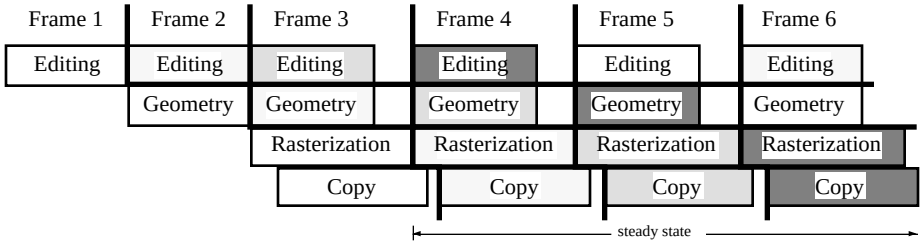


Figure 8.5 Pixel-Planes 5 pipelining with a rasterization stage bottleneck.

geometry stages could start earlier if sufficient buffering existed, but this would usually just add latency. The copy stage follows the rasterization as before. It takes roughly as long as the rasterization stage because it starts when the first region is rasterized and ends after the last region has been rasterized. Figure 8.5 shows the pipeline in this case.

The fourth pipeline configuration occurs when the copy stage is the bottleneck, when the system is limited by the frame buffer's maximum update rate. In this case, shown in figure 8.6, it takes several frames for the earlier stages to fill the buffering in the system and reach the steady state. In the figure, this happens at the start of the fifth frame, where the three earlier stages are waiting for the copy to finish. Unlike the other stages, the pixel copying is not started as soon as the first region is finished, but instead it is done when the previous frame's copy has finished. This means that the SGPs cannot tell the renderers to send the computed pixels to the frame buffer immediately after rasterization, since they would be sent to the buffer being displayed. Instead, the MGP tells the renderers to send the waiting pixels after the frame buffer has switched buffers.

The pipelining in Pixel-Planes 5 is handled by the software. Having as many as four frames in progress at once makes the software more complex than software for rendering one frame at a time. The software must keep track of multiple copies of the parameters. It also keeps track of two sets of buckets shared by the frames in progress. One set of buckets is filled by the geometry processing stage, and the second is emptied by the rasterization stage.

The implementation has one optimization that has not been discussed: it can rasterize two frames at once. After the rasterization has been started for all of a frame's regions, any free renderers can be assigned to the next frame as long as the geometry processing for the later frame has been completed. The optimization increases the utilization of the renderers when some regions are bottlenecks, when some regions have that more than $1/N_R$ of the total workload. The main cost is additional software complexity: many of the messages and data structures must be tagged with a frame identifier. Figure 8.7 shows the pipelining when two frames are rasterized at the same time. In fact, it would be unusual to see the case shown in figure 8.5, where there is a rasterization bottleneck and only one frame is rasterized at a time. It can only happen when all the renderers finish a frame's rasterization nearly simultaneously.

Having a single, separate processor (the MGP) perform the renderer assignments and the synchronization is less than optimal. Because work to be done is proportional to the frame rate, the MGP is nearly idle at low frame rates. However, when running with many regions and at high frame rates, the MGP can be the bot-

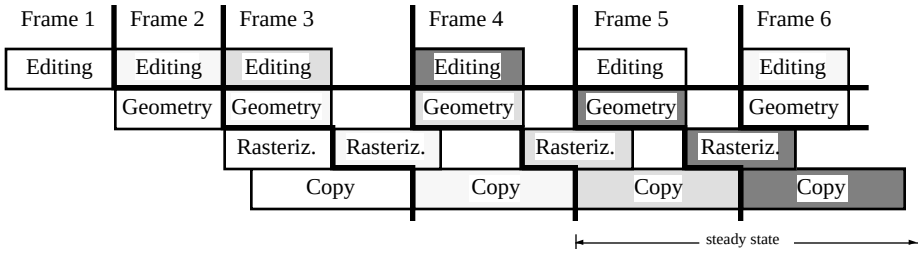


Figure 8.6 Pixel-Planes 5 pipeline example with a copy stage bottleneck.

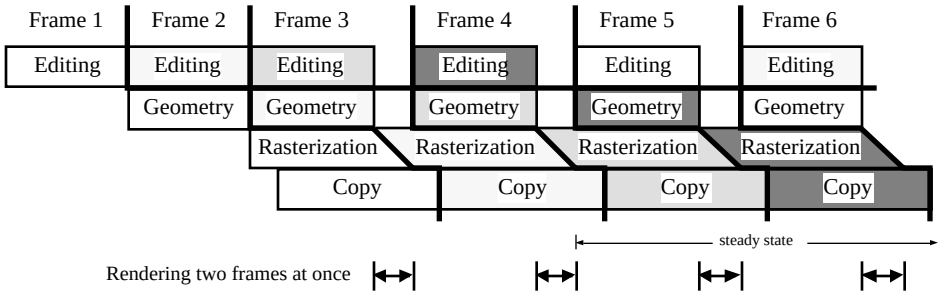


Figure 8.7 Pixel-Planes 5 pipelining with a rasterization stage bottleneck, rasterizing two frames at once. The times when two frames are rasterized at once are indicated.

tleneck. Measurements show that the bottleneck primarily occurs when the simpler regions are quickly rendered at the end of each frame.

8.2.3.2 Advanced Rendering Features

Pixel-Planes 5 has other rendering features beyond the Phong shading and texturing supported in the basic software algorithm. These features require a more complex rendering algorithm. The first two features run part of the pipeline more than once to generate a single frame.

Anti-aliasing. Pixel-Planes 5 supports anti-aliasing by rendering multiple frames and blending them together. The successive frames are offset from each other by a fraction of a pixel, sampling different locations within each pixel with each frame. The multiple offset frames are generated by running the last three pipeline stages (geometry, rasterization, and copy) for each frame. This method was introduced with Pixel-Planes 4 [Fuch85] and was also done using an accumulation buffer on Silicon Graphics systems [Haeb90]. The system can either accumulate all the samples for an anti-aliased image before displaying it, or it can display the current result at the end of each pass. The latter method is called *successive refinement* because successive frames increase the image quality. When using successive refinement, the anti-aliasing passes are stopped when the user requests a new frame.

When performing anti-aliasing, the rendering algorithm assigns the same regions to each renderer for each of the offset frames. This is needed to avoid having to copy the earlier frame's computed pixels between renderers, which would be fairly time consuming (0.82 ms per region), and would consume ring bandwidth.

Stereo. Stereo is supported by rendering two frames with slightly different viewing transformations to generate the left and right eye views. The user specifies both views during the single editing pipeline stage. The geometry and rasterization stages are run twice. The copy stages for the two frames run concurrently since each frame's pixels are sent to different memory locations in one or more frame buffers. When anti-aliasing and stereo are both used, frames for the left and right eye are done for each step in the anti-aliasing loop.

Stereo frames can be displayed either using two NTSC or field-sequential-color frame buffers, one for each eye (generally used with head mounted displays). Or, a single frame buffer can be used. If a single NTSC frame buffer is used, the image for each eye is sent to separate video fields. Then, external hardware blocks light to the left and right eye as the fields are displayed. The hardware could be LCD glasses that allow the left and right lenses to be selectively darkened. Or, the hardware could be a selective polarizer in front of the display plus polarizing glasses worn by the viewer. The polarizer displays each field with different polarizations. The glasses' two lenses each have different polarizing materials such that each eye only receives light from one field.

A similar setup is used when using the high-resolution frame buffer. One method uses two banks of memory to display the left and right eye images that are alternately displayed in successive fields. A second method places the left and right eye images in the top and bottom halves of a single buffer, and doubles the monitor's vertical scanning rate. In each method either LCD glasses or selective polarization with polarizing glasses routes the different fields to the correct eye. More information about stereo displays can be found in section 18.11.5 of [Fole90].

Transparency. Transparent objects are rendered using a multipass technique developed concurrently by the Pixel-Planes group and by Stellar [Apga88] (see figures 8.8 and 8.9). In the first pass all opaque polygons are rendered and shaded. In later passes the transparent polygons are rendered, compositing the transparent polygons from back to front. In each pass, all the transparent objects are rendered using a modified *z*-buffer algorithm. Instead of finding the polygon closest to the viewer, each pixel processor finds the transparent polygon that is the farthest from the image plane but has not yet been composited. At the end of the pass, each pixel has the polygon that is the farthest from the viewer. That polygon is shaded and composited with the colors from earlier passes, and then the "far-buffer" is updated. (In the first pass, the far buffer and color value are set to the values that were generated while rendering the opaque polygons.)

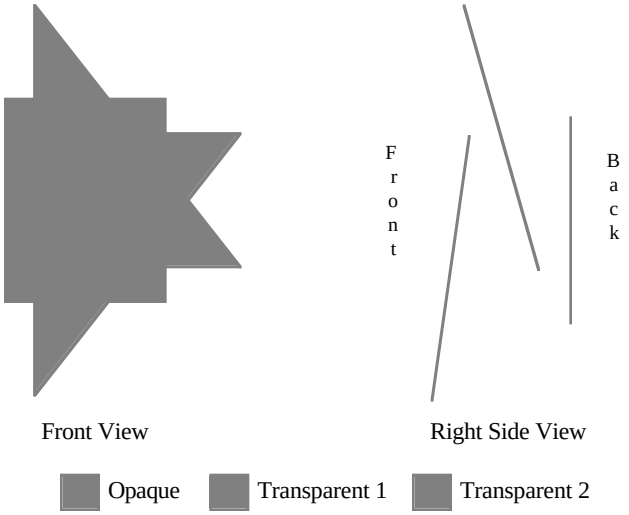


Figure 8.8 Model used in the multipass transparency example in figure 8.9. The model has two semi-transparent triangles (shaded with diagonal lines) overlapping each other. An opaque rectangle (shaded with vertical lines) is behind the triangles. The model is shown from the front and from the right.

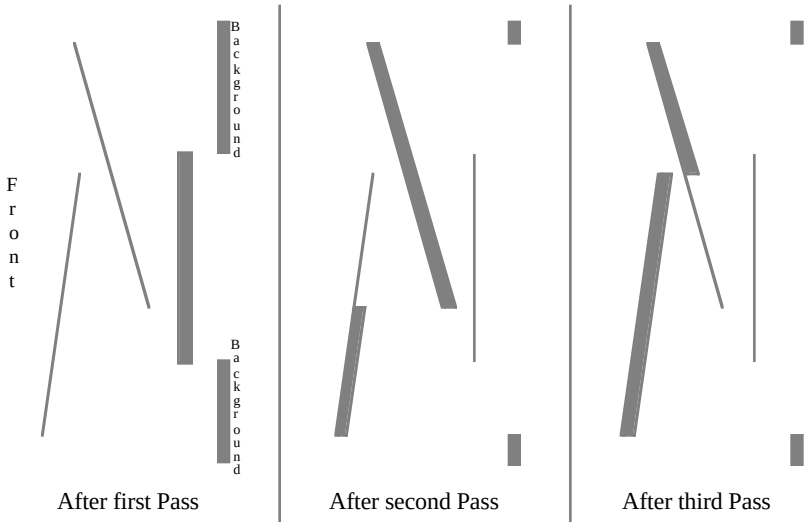


Figure 8.9 Multipass transparency example showing a single column of pixels from the model in figure 8.8. The shading indicates the position of the far *z*-buffer at the end of the first three passes.

When performing multipass transparency, the GPs puts renderer commands into two sets of buckets: one for opaque polygons, and one for transparent ones. The renderers first rasterize the opaque polygons, as before. The same algorithm is used for each pass for the transparent polygons, sending the rendering tokens through the SGPs and calculating the shading at the end. At the end of each pass, each pixel processor calculates whether it has added a new layer. These values are OR'd across all the processors and the

result sent to the MGP. No more passes are necessary if no pixel processor has added a layer. This method allows the number of passes to be adjusted to the model's requirements on a region-by-region basis. The passes are also halted when a user-specified number of them have been done. The system can also calculate transparency from front to back using a similar algorithm (selected at compile time).

8.2.3 Performance

Section 8.1 described many of the differences in algorithms between general-purpose systems and those using specialized rasterization hardware. We can evaluate the effect of the specialized hardware by analyzing the performance of Pixel-Planes 5. I did this using the models and viewpoints described in Appendix A. I only used the three smaller models for most runs because the Polio model was too large to run with many configurations—when in PPHIGS format, the Polio model requires over 100 Mbytes, which means at least 40 SGPs are needed for reliable operation. The frames were computed using Phong shading and no anti-aliasing, and were sent to the high-resolution frame buffer. Different runs sampled different system sizes, sampling combinations of up to 54 SGPs and up to 20 renderers. Only even numbers of processors were used to reduce the number of runs. Also, different runs used the high-resolution frame buffer in both high-resolution (1280x1024) and low resolution (640x512) modes. The sequence of frames was run between 2 and 10 times, depending on the average frame rate, so that delays caused by the host computer would not skew the results. Even though the runs were made with the host giving the runs high priority (in NOPAUSE mode), the host occasionally slowed the system down.

8.2.3.1 Representative Runs

Figure 8.10 shows some representative frame times for four runs, each replaying the frames only once, for 19 GPs (18 SGPs) and 8 renderers. One curve shows the frame times when computing a high-resolution image. The frame times are fairly even. The times decrease at the end of the series as more of the model is off the screen (see Appendix A for representative frames). Three other curves show the frame times from runs that computed low resolution images. At the beginning and end of these runs, the frame times are lower than those from the high resolution run since the overlap factor is lower. In the middle of the runs, all the triangles are concentrated on a small portion of the screen, causing a few regions to be a rendering bottleneck. Most of the time a ringing effect is seen, where the frame times alternately undershoot and overshoot the normal frame time. The second and third low resolution runs show the effect while the first run shows that the ringing is occasionally minimized. Two other runs were very similar to the third run and are not shown.

The ringing effect occurs when a few regions are the bottleneck. When this happens, the geometry processing finishes before the completion of the previous frame's rasterization. Because only a few of the regions are a bottleneck, some of the renderers are idle when the geometry processing completes. Those idle renderers are put to work rasterizing the frame that had just completed its geometry processing stage, thus rasterizing two frames at once. This case is illustrated in figure 8.11. Rasterization of two frames at the same time is shown by the slanted division between the different rasterization blocks, showing gradual shift of renderers from one frame to the next.

The completion times of the last regions in the second frame will be grouped more closely together than in the first frame because the rasterization of the most complex regions started earlier in the second frame. The figure shows this by the different slopes between the different rasterization blocks. The rasterization of the third frame is similar to the first frame, the fourth frame is similar to the second, etc., leading to the ringing effect.

I was surprised to see that the ringing effect does not always occur. Sometimes the stages start and end so that the frame times are nearly the same, as shown by the low resolution run in the top portion of figure 8.10. I believe that this happens when (1) the geometry processing of a frame is delayed by the host so that it finishes after several renderers have completed the rasterization of an earlier frame, and (2) when only a few renderers are finishing up the bottleneck regions. This will remove the asymmetry of having, for example, the rasterization of odd frames having spread-apart region completion times while even frames have region completion times that are closer together. Instead, the distribution of region completion times will be nearly the same. Since this delay only occurs a fraction of the time, the ringing effect is seen most of the time.

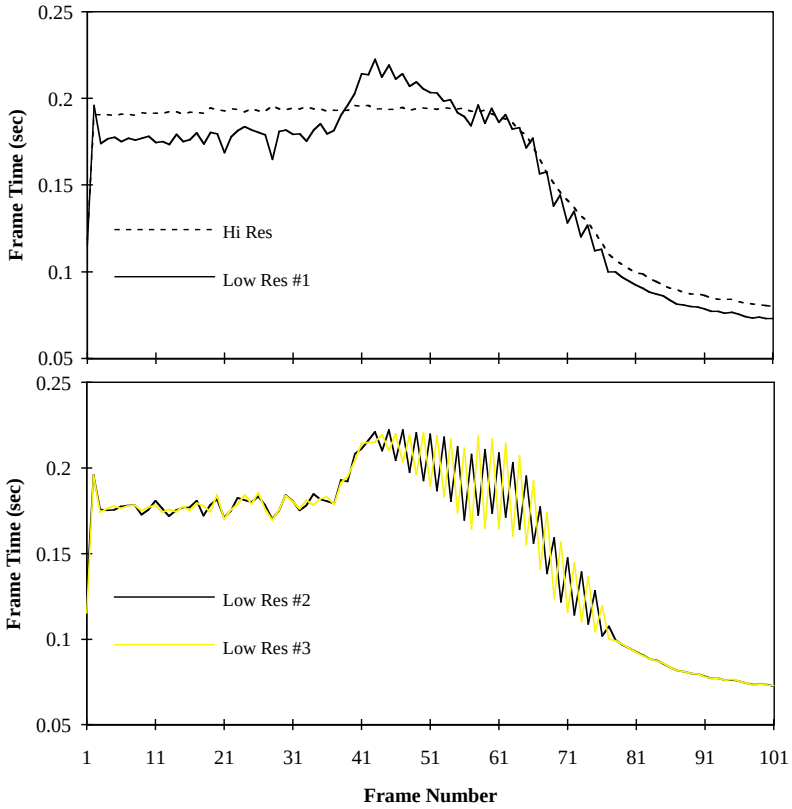


Figure 8.10 Sample Pixel-Planes 5 frame times for the Terrain model using 18 SGPs (19 GPs) and 8 renderers for one high resolution run and three low resolution runs. The low resolution runs shows a frame time ringing effect.

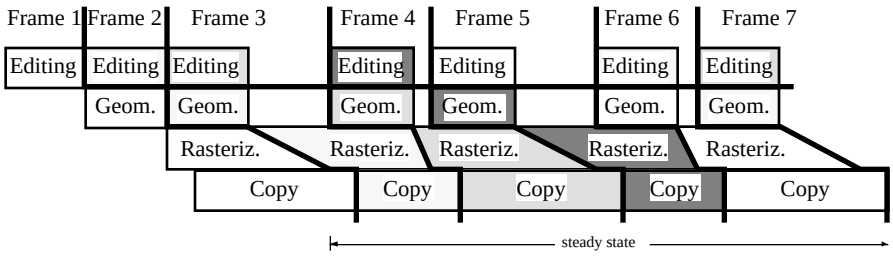


Figure 8.11 Pipelining example when rasterization of individual regions is the bottleneck. Note the uneven starts of frames 3 through 7.

8.2.3.2 3D Performance Charts

The results of the performance runs are shown in the following charts. Figures 8.12–17 show the performance of the three models running in high resolution and low resolution mode. The high resolution mode charts are on the top of the pages; the corresponding low resolution charts are on the bottom. The charts show the average triangles-per-second value for each configuration. Like the Delta runs, these values were calculated by first calculating a per-frame triangles-per-second value (by dividing the number of triangles in the viewing frustum by the frame time), and then averaging the per-frame values over the series of frames. Note that this figure is not necessarily the average number of rendered triangles per second since two of the models were run with back-face culling enabled.

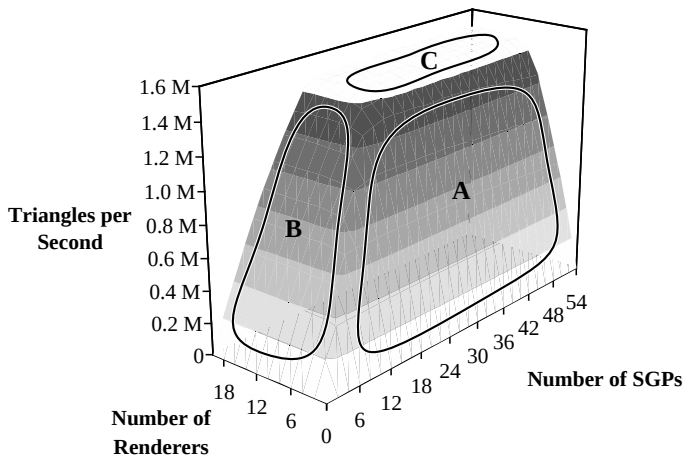


Figure 8.12 Pixel-Planes 5 performance when displaying the PLB Head model (containing 59,592 triangles) at high resolution for different numbers of SGPs and renderers. The performance values count back-facing triangles inside the viewing frustum. The three different faces referred to in the text are circled and marked “A”, “B”, and “C”.

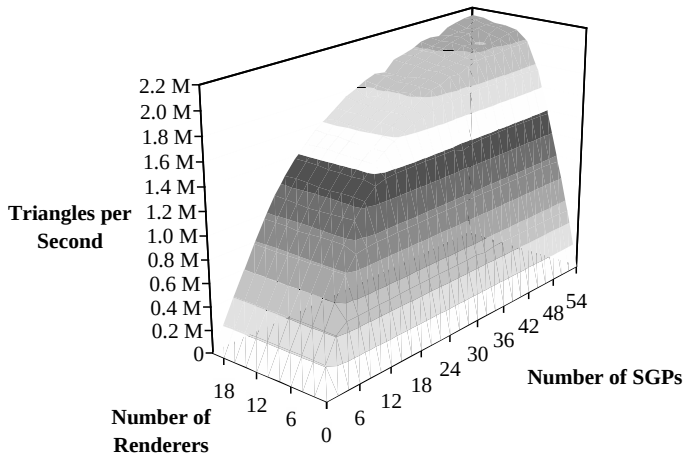


Figure 8.13 Pixel-Planes 5 performance when displaying the PLB Head model (containing 59,592 triangles) at low resolution for different numbers of SGPs and renderers. The performance values count back-facing triangles inside the viewing frustum.

Each 3D chart has two semi-planar inclined “faces”. The face on the right of the chart is due to the system being renderer bound while the face on the left is due to the system being SGP bound. In the first chart (figure 8.13), the renderer-bound face is marked “A” and the SGP bound face is marked “B”. At the intersection of the two faces the mix of SGPs and renderers is the optimum, where neither resource is the bottleneck.

The PLB Head model’s high resolution mode chart has a third face at the top of the chart (marked “C”) showing the system configurations that are limited by the maximum frame rate of 24 Hz. The corners between the top face and the other faces are not very sharp because the configurations on those corners are frame rate limited for a fraction of the series of frames. Adding more processors to these configurations gives diminished returns because they cannot increase the speed of the frames that are already frame-rate limited. The top face of the chart is bumpy because the times are noisy: even with the high priority mode enabled, the host cannot always change the view matrix in $^{1}/_{24}$ second.

The SGP-bound face of the charts is not planar in the three low resolution charts and, to a lesser extent, in the high resolution chart for the Terrain model. This happens because only 6 or 8 renderers can be used to render a single frame before one of the 20 regions becomes a bottleneck. If the SGPs can transform a frame

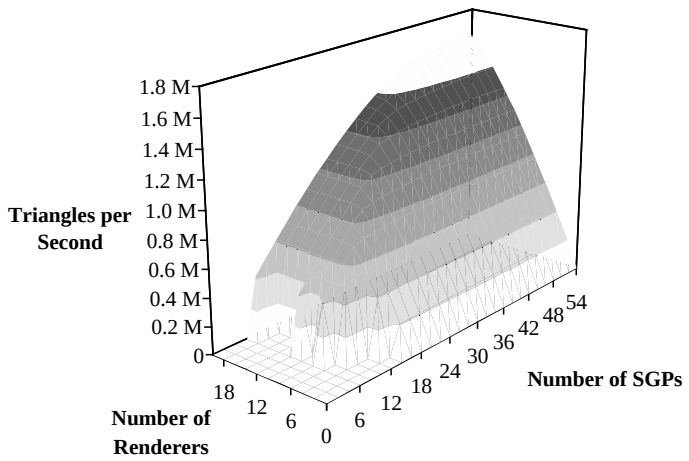


Figure 8.14 Pixel-Planes 5 performance when displaying the Terrain model (containing 162,690 triangles) at high resolution for different numbers of SGPs and renderers. The values for 44, 48, and 52 SGPs were not measured due to time constraints, and thus are interpolated between the values for the next higher and lower number of SGPs with the same number of renderers.

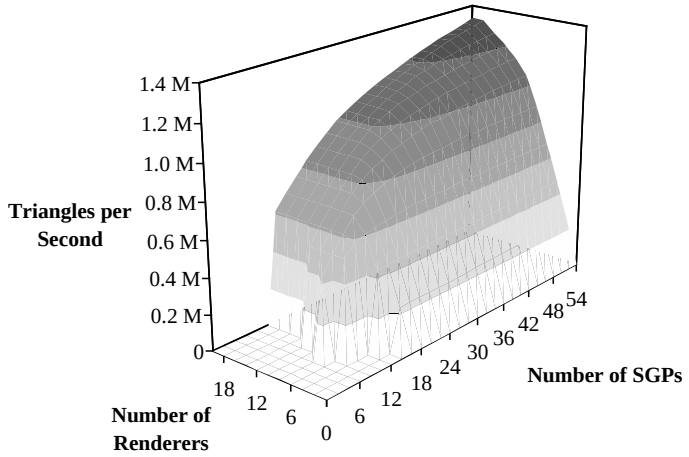


Figure 8.15 Pixel-Planes 5 performance when displaying the Terrain model (containing 162,690 triangles) at low resolution for different numbers of SGPs and renderers. The values for 44, 48, and 52 SGPs are interpolated as described above.

faster than the most complex region can be rasterized, then two frames can be rasterized at a time, putting more renderers to use. The performance increases when the number of SGPs is increased beyond what would otherwise be needed because it increases the amount of time when two frames can be rendered concurrently. This effect is less pronounced with the Terrain model running in high resolution mode, since the renderer bottleneck occurs in only a few of the frames: the ones where the model fills a small portion of the screen.

The high resolution PLB Head chart has another unique feature besides the frame rate limitation: when SGPs are added beyond the optimum, the performance decreases instead of staying constant. This decrease is hard to see since it is fairly small; there is only a 6% decrease when using 8 renderers. The performance decrease could happen because adding more SGPs decreases the amount of data that each SGP sends to each renderer. When more SGPs are used than is necessary, the amount of time available for the processors to send the renderer commands decreases. Because the time between the arrival of a token and the first renderer command message is sent is fairly independent of the number of SGPs, more SGPs increases the probability that a SGP cannot send its commands in time. A second possible explanation for the performance decrease is that having more SGPs increases the MGP's workload since it must process more synchronization messages. The increased workload would increase the fraction of time where the other processors wait on the MGP.

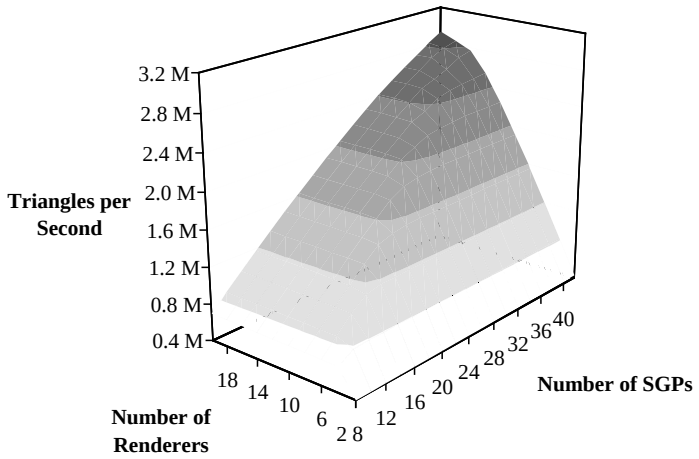


Figure 8.16 Pixel-Planes 5 performance when displaying the CT Head model (containing 229,208 triangles) at high resolution for different numbers of SGPs and renderers. The performance values count back-facing triangles inside the viewing frustum.

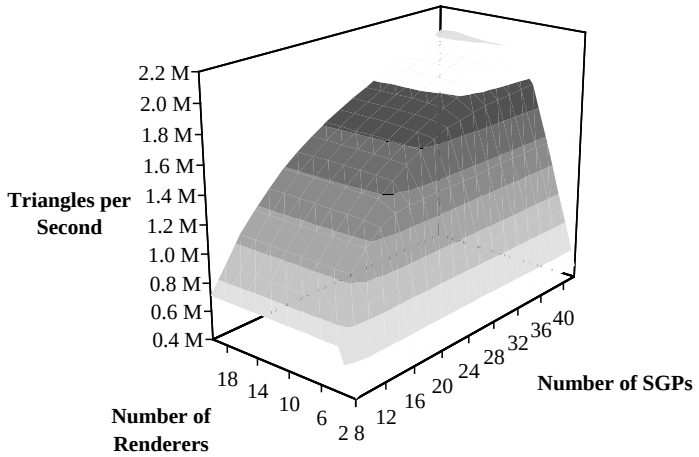


Figure 8.17 Pixel-Planes 5 performance when displaying the CT Head model (containing 229,208 triangles) at low resolution for different numbers of SGPs and renderers. The performance values count back-facing triangles inside the viewing frustum.

One surprising result is that the highest overall performance with the two larger models is less in low resolution mode than in high resolution mode. One might think that low resolution mode would be faster because less work is required: a low resolution image does not need as much computation because the overlap factor is smaller, and less time is needed to send rendering tokens between processors (the time is proportional to the number of regions). Instead, in low resolution mode performance is lower in large configurations because the renderer utilization is low. Since a low resolution image has fewer regions, the granularity ratio is lower, thus reducing the renderer utilization. The PLB Head model is the exception because its highest speed in high resolution mode is determined by the 24 Hz maximum update rate. Producing high resolution images is not always faster: it would be slower if the triangles are evenly distributed across the screen or if the number of renderers is small. For example, two renderers can render a high resolution image of the PLB Head model at 380,000 triangles per second and a low resolution image at 500,000 triangles per second.

8.2.3.3 Ratio of SGPs to Renderers

Section 8.1.1 described one disadvantage of systems that use specialized processors: it is impossible to always balance the workloads between the two sets of processors. This section gives an example of this

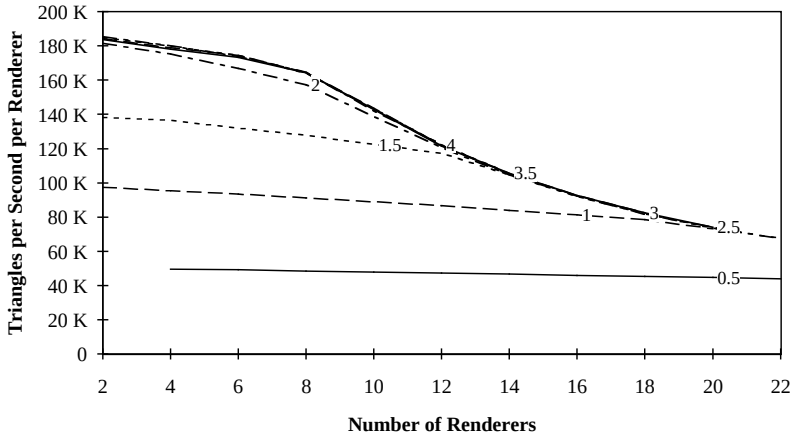


Figure 8.18 Pixel-Planes 5 performance per renderer when displaying the PLB Head model (containing 59,592 triangles) at high resolution for ratios of SGPs to renderers of 0.5 to 4.

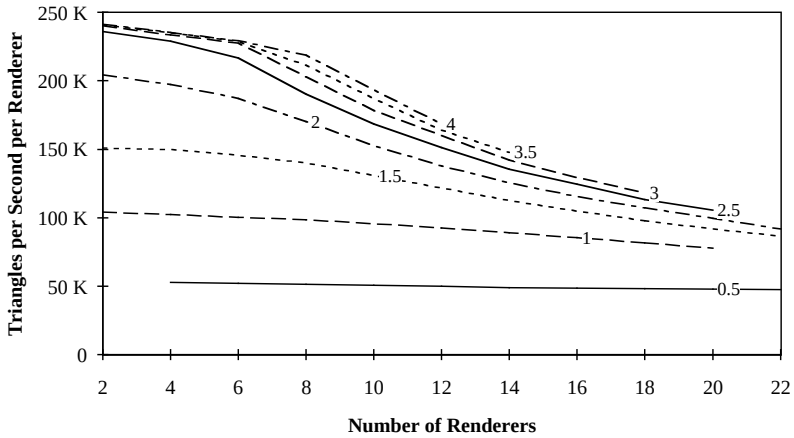


Figure 8.19 Pixel-Planes 5 performance per renderer when displaying the PLB Head model (containing 59,592 triangles) at low resolution for ratios of SGPs to renderers of 0.5 to 4.

difficulty by exploring how the Pixel-Planes 5 system should be configured to give good performance for a variety of models and image resolutions.

The exploration will use a second series of charts (figures 8.18–23). The charts plot different ratios of SGPs to renderers, and allow us to determine the most useful mix of processors. The top chart on each page shows the high resolution case and the bottom chart shows the low resolution case. Each chart has a series of curves giving the performance per renderer, which is the overall triangles per second figure divided by the number of renderers used. Each curve gives the performance for a constant ratio of SGPs to renderers, with ratios of 0.5, 1, 1.5, 2, 2.5, 3, 3.5, and 4 shown (if possible; the lower ratios were not possible with the larger models as there were too few SGPs).

We can determine the optimum ratio when running in high resolution mode simply by examining the charts. The optimum ratio is the minimum ratio that gives the maximum performance. With the PLB Head and Terrain models, the maximum performance is delivered with a 2:1 ratio of SGPs to renderers; increasing the ratio beyond 2 gives very little return. The CT Head model works best with a ratio between 2:1 and 2.5:1. The charts also show the system's efficiency decreasing with larger systems, which is due to the decreased renderer utilization resulting from the smaller granularity ratio as well as the increased overhead incurred when using more processors. Note that the first chart, figure 8.18, shows low renderer utilization due to the 24 Hz frame rate limitation.

When the SGP:renderer ratio is increased, the renderer utilization decreases much more quickly in low resolution mode compared to high resolution mode. This happens because the granularity ratio is much

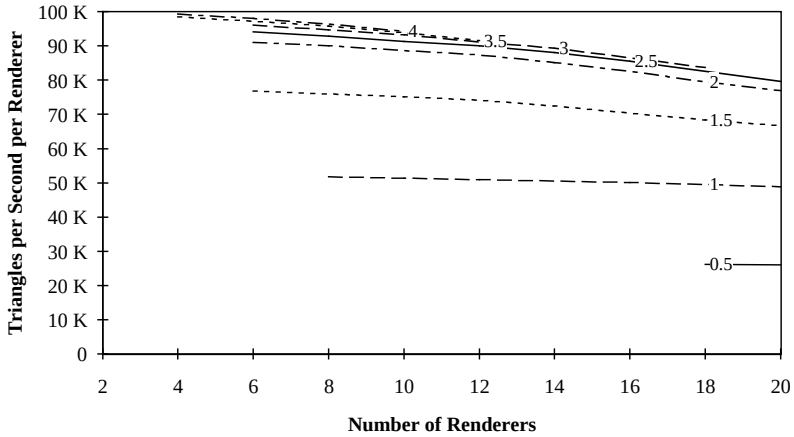


Figure 8.20 Pixel-Planes 5 performance per renderer when displaying the Terrain model (containing 162,690 triangles) at high resolution for ratios of SGPs to renderers of 0.5 to 4.

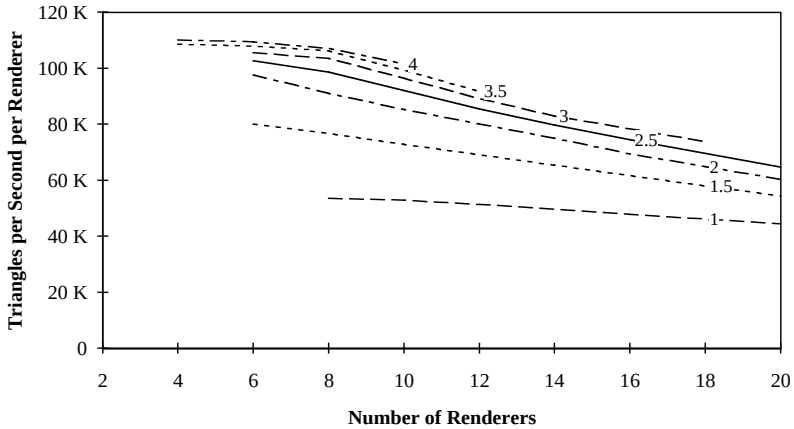


Figure 8.21 Pixel-Planes 5 performance per renderer when displaying the Terrain model (containing 162,690 triangles) at low resolution for ratios of SGPs to renderers of 1 to 4.

smaller. It is harder to determine the best SGP-renderer ratio because excess SGP power is used to increase the renderer utilization by rendering two frames at once. However, the point of diminishing returns is still fairly clear in the charts. With medium to large systems (at least 10 renderers), a ratio of 1.5:1 is best. Fewer SGPs are needed than in high resolution mode because the renderer utilization is lower. In smaller systems, more SGPs are needed because the renderer utilization is higher, so the PLB and CT Head models work best with a ratio of 2.5 to 1. The Terrain model requires too much memory to run on small systems.

It is unfortunate that different cases (different models and resolution modes) have different optimum processor ratios. The different models have different ratios because they have differing fractions of back-facing and off-screen triangles, both of which require more SGP power. Given this, a ratio of two SGPs for each renderer is a good compromise among the different cases.

8.2.3.4 Maximum Performance

The capability of Pixel-Planes 5 is shown in table 8.1, which gives some performance values from nearly the largest available system. That system has 55 GPs and 22 renderers (the maximum system would have 56 GPs). This configuration has room for only one frame buffer card, which is atypical; most Pixel-Planes 5 systems have several to allow driving both a workstation display and head-mounted displays. The table below gives the average rendering rate for the entire series of frames for each model as well as the peak rendering rate for the “best case” frames where the models’ screen positions were most favorable. The peak rate is useful for comparison with commercial systems because marketing considerations push towards reporting only the systems’ most favorable cases.

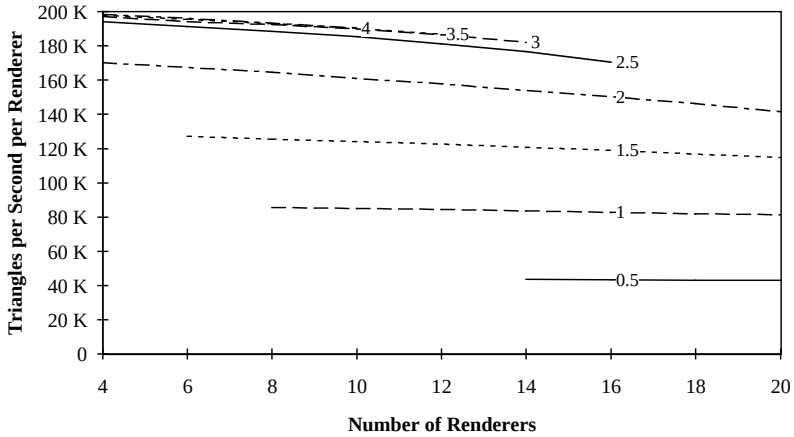


Figure 8.22 Pixel-Planes 5 performance per renderer when displaying the CT Head model (containing 229,208 triangles) at high resolution for ratios of SGPs to renderers of 0.5 to 4.

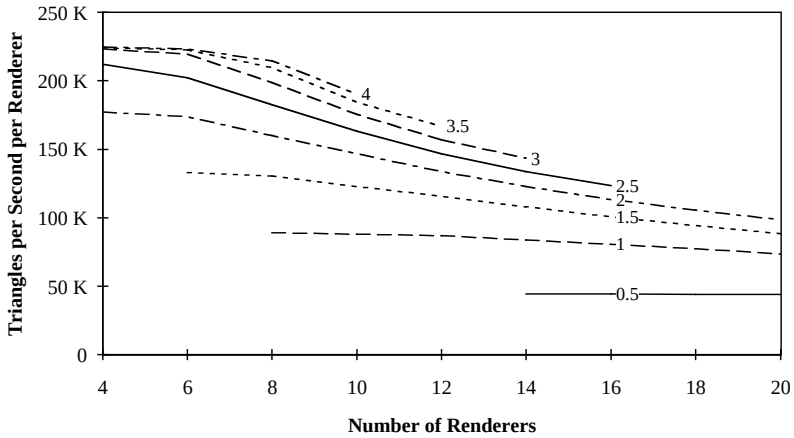


Figure 8.23 Pixel-Planes 5 performance per renderer when displaying the CT Head model (containing 229,208 triangles) at low resolution for ratios of SGPs to renderers of 0.5 to 4.

Model	Reso-lution	Average Rendering Rate		Peak Rendering Rate		
		On-screen	Rasterized	On-screen	Rasterized	Frames Used
PLB Head	High	1.49	0.65	1.46	0.71	7-20
	Low	2.15	0.92	2.16	1.05	7-20
Terrain	High	1.68	1.68	2.15	2.15	2-18
	Low	1.32	1.32	2.07	2.07	2-18
CT Head	High	3.22	1.59	3.46	1.72	5-20
	Low	2.23	1.10	2.36	1.17	5-20
Polio	High	4.24	2.07	4.72	2.12	40-60
	Low	3.09	1.52	3.38	1.67	80-100

Table 8.1 Average number of triangles per second, in millions, rendered on Pixel-Planes 5 using 54 SGPs and 22 renderers, for each of the four models. The numbers in the “On-screen” columns count all triangles in the viewing frustum while the “Rasterized” numbers only count rasterized triangles. The average rendering rate is the average for all the frames in the sequence while the peak rendering rate is the average rate for the frames listed in the last column.

8.3 MEGA Implementation on Pixel-Planes 5

A second rendering algorithm, called MEGA, has been implemented on Pixel-Planes 5. Implementation of the second algorithm was started in the middle of the PPHIGS development. At that point, the latter implementation's performance was disappointing as it had not been fully optimized, and one worry was that the token passing for the work-queue region assignments was too expensive. Hence, MEGA was developed to complement PPHIGS to insure that Pixel-Planes 5's full potential would be used. MEGA was designed and implemented by Anselmo Lastra, Ulrich Neumann, Marc Olano, and Mark Parris, although I can take credit for the original idea.

Instead of the expensive work-queue region assignments, MEGA uses static region assignments. It has a reduced feature set to allow easier optimization: it supports only triangles with precalculated Gouraud shading instead of several types of primitives with Phong shading and textures, and only supports flat display lists with viewpoint changes. Latency was also a concern, so MEGA's $1^1/_2$ frames of latency (after editing is completed) was attractive compared to PPHIGS' latency of up to three frames.

MEGA's static assignment of regions to renderers removes the cost of the work-queue region assignment style. The work-queue load-balancing method's worst-case cost is considerable. When running at 24 Hz with the high resolution frame buffer (with 80 regions), sending the rendering tokens among the GPs takes nearly 20% of the GP processing time. Running at 60 Hz with the NTSC frame buffer (20 regions) takes 14% of the time. This is only the most identifiable cost; there is additional price paid in renderer utilization. When rendering regions at such a high rate, the rendering tokens can pile up at a GP, which causes the associated renderers to wait for data. Renderers can also be forced to wait if the MGP takes too long to reassign the region to another processor, so the performance advantage may be larger than the 20% and 14% figures given above. The static assignment is less flexible than PPHIGS' because a renderer has only enough memory to hold 4 contexts at once: it only has 208 bits of memory, and each context requires 48 bits (24 bits each of color and z). This means that at least 6 renderers are needed when computing a low resolution image (5 is not sufficient due to an implementation restriction), and 20 renderers are needed when computing a high resolution image. If fewer renderers are available a smaller image is rendered. The renderers are assigned to regions in an interleaved fashion similar to the static interleaved load-balancing method described in section 6.2.2.2. This method usually has each renderer rasterizing non-contiguous regions.

The triangle rendering rate is improved by performing Gouraud interpolation at run time instead of calculating Phong shading. The GPs do not need to transform the normal vectors nor generate renderer commands to set the per-triangle values for Phong shading. The renderers can interpolate colors faster than normal vectors. Also, the surface properties do not need to be written into memory nor do the shading calculations need to be done. Having the system only support triangles allows further optimizations. Instead of writing the renderer instructions to the buckets during geometry processing, part of the instructions can be written during initialization. This can only be done when a single primitive type is used, since the instructions depend on the type of primitive. When multiple primitives are supported, they can be encountered in arbitrary numbers and different orders, making the early writing impossible. A second optimization polls outgoing message FIFO to see if there is room for another message. The PPHIGS rendering algorithm uses fairly expensive interrupts instead of polling. Polling is only reasonable in a simple system; in a complex system the number of places in the code where polling must be done would make it hard to manage. Finally, the reduced feature set means that it is more likely that the inner loops will remain in the cache, another concern of the PPHIGS development when the MEGA project began.

However, the above savings are traded against MEGA's lower renderer utilization. Static assignment gives lower renderer utilization when compared to equivalent active region assignments. Furthermore, MEGA uses four regions per renderer, lower than the best value selected in Chapter 6 for use with fixed region boundaries, which will also decrease the renderer utilization.

8.3.1 MEGA Rendering Algorithm

The MEGA rendering algorithm has the following steps:

0. During initialization, the host loads the model—a list of triangles—into the SGPs' memory. Triangle i is sent to GP $i \bmod N$. The host also sends the initial viewing parameters to the MGP, which also requests a new frame.

The MGP sends the initial viewing parameters to all the SGPs. They then perform the geometry processing: transforming their triangles to screen space, generating renderer instructions, and placing them into a queue to be sent. The instructions are sent as fast as the ring network can accept them. The renderers start the rasterization as soon as they receive instructions. The renderer instructions are sent one triangle at a time. Since multiple SGPs can be transmitting to a renderer, the ring hardware arbitrates access to the renderers so that each triangle's instructions are received atomically.

1. The MGP waits for new viewing parameters from the host.

2. When a SGP finishes its geometry processing and has sent all of its renderer instructions, it sends a completion message to the MGP. The MGP sends the SGP viewing parameters for the next frame, and the SGP starts the geometry processing for the next frame (as in step 0). However, the SGPs buffer the generated renderer commands instead of sending them.

3. When the MGP has (a) received completion messages from all the SGPs and (b) has received a message from the frame buffer indicating that it is now showing the previous frame, it tells the renderers to copy the computed pixels to the frame buffer.

4. The MGP waits for completion messages, one for each region, indicating that the region has been sent to the frame buffer. It then tells the frame buffer to switch buffers and send a message when the switch occurs (during vertical retrace).

5. The MGP tells the SGPs to begin sending commands to the renderers.

6. Finally, the MGP tells the host that the frame has been started. The host computes the new viewing parameters and sends them to the MGP. The MGP waits for the new parameters in step 1.

8.3.2 MEGA Performance Limitations

MEGA suffered performance limitations when running with real-world models. The renderers suffered from poor utilization, which was caused by the static region assignment, and from suboptimal use of the ring network. The SGPs had code that was not fully optimized. The next sections give details of the problems, and possible solutions.

8.3.2.1 Static Region Assignment

The simulation studies in Chapter 6 show that static region assignment results in lower rasterization processor utilization when compared to any dynamic region assignment method. When running in full-frame mode, MEGA uses 20 processors when generating a 1280 by 1024 image and either 6 or 10 processors for a 640 by 512 image. The expected processor utilization ranges from 33% to 86%, as shown in figure 8.24.

The renderer utilization could be improved by assigning regions once each frame using method similar to the one used in the Delta implementation. The assignment method is constrained because the maximum number of regions per processor is only four; the Delta implementation allows a maximum of 16. The greedy assignment algorithm will usually give slightly lower utilization with a small maximum number of processors. For example, a processor might be assigned two complex regions that make up all the work expected of that processor. However, it will have two other regions assigned to it after the other processors have been assigned the maximum 4 regions. A more complex algorithm could make a better assignment of regions to processors. However, figure 8.24 shows that the greedy assignment algorithm does a much better job than static assignment, and that using at most four regions per processor does not appear to significantly lower the performance since removing the four region-per-processor limitation results in only a small improvement.

8.3.2.2 Suboptimal Redistribution

The next most significant limitation is how MEGA performs the triangle redistribution between the geometry processing and rasterization: it sends each triangle in a separate message. The message-sending CPU overhead that troubles the Delta implementation is not the deciding factor in this case, since the sending routine has been optimized. Instead, it is the network's message overhead that limits the speed—when sending messages containing only one triangle, many fewer triangles can be redistributed than could be achieved with larger messages.

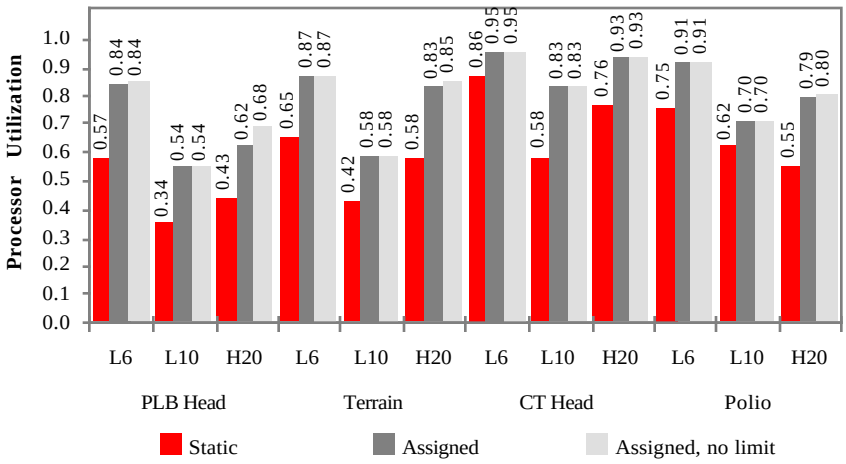


Figure 8.24 Renderer utilization for MEGA using the current static assignments, between-frame assignments with at most 4 regions per processor, and between-frame assignments without limiting the number of regions per processor. Each of the four models was run with 6 and 10 processors at low resolution (L6 and L10), and with 20 processors at high resolution (H20).

I measured this problem by writing a program that sends messages from SGPs to renderers using MEGA's redistribution method. It only does the redistribution; to avoid other bottlenecks, it does not do any geometry processing or rasterization (the messages are filled with no-ops). This program also has renderers send pixels to the high resolution (1280 by 1024) frame buffer at a rate of 24 Hz.

When sending one triangle per message (30 words), the maximum speed was 2.4 million triangles per second. This does not account for the overlap factor; with 50 pixel triangles, the overlap factor is 1.17, giving a maximum rendering rate of just over 2 million triangles per second—slower than the PPHIGS implementation. Since a renderer can process 165,000 triangles per second, the rate above means that at most 15 renderers can be kept busy. Using the largest message possible, containing 34 triangles, 4 million triangles can be redistributed per second, which corresponds to a rendering rate of 3.4 million triangles per second. Here, as many as 24 renderers can be used before the ring is saturated. These figures are an upper bound to MEGA's performance; a real application will not always be sending at full speed all the time. Sending more than one triangle per message would require a slight increase in code complexity.

The redistribution method has a secondary problem: the unconstrained access to renderers can, at times, cause some renderers to be idle. Since triangles are sent in the order that they are encountered in the (distributed) model, at times only a few renderers will be sent the bulk of the triangles, which idles the other renderers after they exhaust their buffered triangles. This will happen fairly often unless the order of the triangles in the model is randomized. Without randomization, the many models with successive triangles next to each other will cause this renderer starvation.

Since MEGA uses static models, the randomization was performed in a preprocessing step. Randomization is much harder if editing is supported. Instead, a second method to minimize starvation could be used. Here, each SGP maintains several queues of messages waiting to be sent, one for each renderer. The SGPs send successive messages from the head of successive queues, sending messages to different renderers in round-robin order. This should reduce the impact of temporary hot spots. The reader should note that this solution has not been tested.

8.3.2.3 Suboptimal Triangle Processing Code

The last MEGA limitation affects the SGPs, and is an implementation issue instead of an algorithmic issue. It concerns the high-level structure of the per-triangle processing code, and should apply to implementations other than MEGA. The limitation is a historical artifact: at the time when MEGA was implemented, the then-current PPHIGS implementation had low performance because its per-triangle loop did not fit into

the i860 cache so it had disappointing performance. To avoid this problem, the MEGA group structured the geometry processing code as a series of loops, each of which does a small amount of processing for a number (100) of triangles. For example, the first loop transforms the vertices, the second loop clips the vertices, the third loop does the perspective division and culls triangles that are too small. Using multiple loops also increases the performance of the i860's floating-point pipelined operations because the priming and flushing of the pipeline can be amortized over many triangles instead just over a single triangle's vertices.

However, PPHIGS' triangle loop was later optimized to fit into the processor's cache. Because MEGA's geometry processing has at most the same complexity as PPHIGS', the corresponding code could be rewritten so it was only one loop. This would avoid having to read and write intermediate results during each loop; instead, the intermediate results would be kept in registers. Since the i860 has limited memory bandwidth, it is quite likely that loading and storing the intermediate results limits the geometry processing speed. A single loop that did all the geometry processing should be faster.

8.3.2.4 MEGA Summary

To be fair to MEGA's authors, the limitations described earlier were only found after the system was implemented. The limitations of static load-balancing were only found during the research for this dissertation, which was done after the MEGA project.

The MEGA system still has promise for increasing Pixel-Planes 5's performance. I estimate that Pixel-Planes 5 could approach 3 million rendered triangles per second given a 60 GP, 20 renderer system and a favorable distribution of triangles across the screen.

8.4 The Value of Specialized Hardware

Now that we have discussed a case study of a system that uses specialized rasterization hardware, we can look into the value of using specialized hardware. The first measure will be to compute and compare the triangle rasterization price-performance ratios for specialized and general-purpose processors. The second measure compares the rendering performance of the Touchstone Delta with Pixel-Planes 5. Both comparisons show that the specialized hardware is more efficient at polygon rendering than the general-purpose hardware.

8.4.1 Rasterization Hardware Price-Performance Ratio

It is somewhat difficult to compare different architectures by using price-performance ratios. One can only compare implementations of the architectures, each of which involves a considerable amount of work. The different implementations will also vary in design quality and in their price sensitivity to cost (cost is not as important when few systems are built). One would have to evaluate a number of systems before arriving at a reasonably precise number.

Given these caveats, we can at least estimate the difference in price-performance difference by comparing the price-performance ratio of the two processors that are part of the Pixel-Planes 5 system. The cost of the parts required will be used as the system cost because the system was built instead of purchased. It has Intel i860 general-purpose processors as well as custom rasterization processors (the renderers). Appendix D gives details of the parts cost of the two processors as of the first half of 1992.

The discussion in the appendix concludes that a renderer has an estimated cost of $6823. It has a raw speed of 180,000 50 pixel 24-bit Gouraud shaded *z*-buffered triangles per second. However, it can only render a 128x128-pixel region at a time. If the triangles have a bounding box of 7x14 pixels, the expected overlap factor is 1.17, giving an actual speed of 150,000 triangles per second.

Two general-purpose processors will fit on each board, so the board and assembly costs are amortized over the two processors giving a total cost per processor of $3266. Using the same optimized rasterization code used in the Delta implementation, the Pixel-Planes 5 i860 processor takes 16.3 μs for triangle setup, 2.1 μs for scan line setup, and 0.20 μs for each pixel—slightly faster than the Delta. However, 10.3 μs is for calculating slopes, which the renderer does not do (it is done during geometry processing); instead 6 μs will be used for the setup value. These numbers give a raw speed of 26,300 triangles per second, when each triangle is rendered once. This assumes 50 pixel triangles and that the number of scanlines is 10.5, the average of 7 and 14. When using a 128x128 region size like the renderer, the setup is performed on average 1.17

times per triangle, and the scan line setup is done $\sqrt{1.17} = 1.08$ times per scanline. The actual speed of an i860 is thus 24,500 triangles per second.

The renderer board is 2.1 times as expensive as an i860 processor, but it can rasterize 6.1 times as many triangles per second. This means that the renderer has a price-performance ratio that is three times higher than the i860. Note that the i860 is using a small amount of special hardware to achieve the rendering speeds: it has a hardware interpolator that increases the speed of interpolating spans. Also, the overlap factor will generally be higher for the i860. More processors will be needed to match the speed of the renderer, decreasing the size of the regions and thus increasing the overlap factor.

However, the fraction of time that a system performs graphics must be considered. If a system only spends 1% of the time in rasterization it does not make sense to add the special hardware; general-purpose hardware would be more cost effective. Today, most parallel systems are only rarely used for graphics, so the general use of specialized hardware remains in the future. Only systems that will be used for graphics-intensive applications need the specialized hardware.

8.4.2 Comparison of Pixel-Planes 5 and the Touchstone Delta

We can study the advantages of specialized rasterization processors by comparing the performance of Pixel-Planes 5 with the Intel Touchstone Delta. The systems are fairly similar: they were both build during the same period; both use Intel i860 general-purpose processors; and both are prototype systems. Using the cost values for Pixel-Planes 5 just developed, we can calculate that a 55 GP and 22 renderer system has the equivalent cost of 101 GPs (this ignores the cost of the host interface, frame buffer, and external hardware, but the Delta has those as well). So, a maximally configured Pixel-Planes 5 system is roughly equal to a 128 processor partition of the Touchstone Delta.

Table 8.2 and figure 8.25 shows the average performance of Pixel-Planes 5 and the Delta for several different configurations. The table includes values for the four models and sequence of viewpoints used previously. Unlike the previous table, the values are the average performance for the entire series of frames. Pixel-Planes 5 is faster than a 128-processor Delta partition for all the models. Pixel-Planes is even faster than the normal Delta implementation using 512 processors, as the load-balancing bottleneck limits the performance of the Delta when using many processors. If the load-balancing bottleneck is removed (by first, in an earlier run, writing the load-balancing assignments to a file and then reading the assignments during the timing run), the Delta is faster than Pixel-Planes for three of the models in low resolution mode, and two of the models in high resolution mode. However, the PLB Head model in high resolution mode is frame-rate limited. Discarding the computed images, which was done with the Delta, would improve Pixel-Planes 5's performance.

The difference in performance is due to four main factors, listed below from most important to the least important:

1. **Reduced communication costs**. The processing time needed to send a triangle using 4 Kbyte messages is 1.6 μs on Pixel-Planes 5 and 9.0 μs on the Delta, even though Pixel-Planes sends 144 bytes per triangle while the Delta sends 56 bytes (Pixel-Planes sends normal vectors and much more shading information as well as the renderer command opcodes). The receive time is even more lopsided: the Delta must spend 11 μs in processing time for each received triangle. With Pixel-Planes, the receive operation incurs no processing time on the renderer since it is performed by dedicated hardware. Overall, the per-triangle communication time is 12 times as large on the Delta as on Pixel-Planes 5!

2. **Decreased parallel overhead**. Because the Pixel-Planes 5 rasterizers are more powerful than the Delta processors and spend less time for communication, many fewer rasterizers are needed to reach a given level of performance. This results in better processor utilization as the loads are better balanced. More importantly, the cost of the primitive redistribution step is reduced. If one-step redistribution is used, a 55 GP, 22 renderer Pixel-Planes 5 system must send a minimum of $55 \cdot 22 = 1{,}210$ messages per frame, while a 512-processor Delta system must send a minimum of $512 \cdot 512 = 262{,}144$ messages. Two-step redistribution will decrease the number of messages required on the Delta, but not by two orders of magnitude.

3. **Faster processors**. The Pixel-Planes i860 processors are somewhat faster due to a more aggressive design, even though they run at 40 MHz like the Delta (the memory system has more write bandwidth). Including setup but excluding geometry processing, a Pixel-Planes 5 i860 can rasterize 19,000 50-pixel

Model	Pixel-Planes 5		Intel Touchstone Delta					
	Low Res.	High Res.	Normal Load-balancing			Prerecorded Load-balancing		
			128	256	512	128	256	512
PLB Head	2.15	1.49	1.39	1.18	0.57	1.47	1.91	1.94
Terrain	1.32	1.68	1.08	1.23	0.95	1.10	1.34	1.83
CT Head	2.23	3.22	1.89	2.60	1.58	1.93	2.70	3.10
Polio	3.09	4.24	2.45	2.87	3.09	2.47	2.91	3.19

Table 8.2 Comparison of Pixel-Planes 5 and the Intel Touchstone Delta. The values are the average number of triangles per second, in millions; they are for the entire sequence of frames for each model, and count the triangles in the viewing frustum. The Pixel-Planes values are for 54 SGPs and 22 renderers; values for both high and low resolution images are given. The Delta values are for low resolution images, and include values when performing load-balancing and when reading load-balancing information from a file.

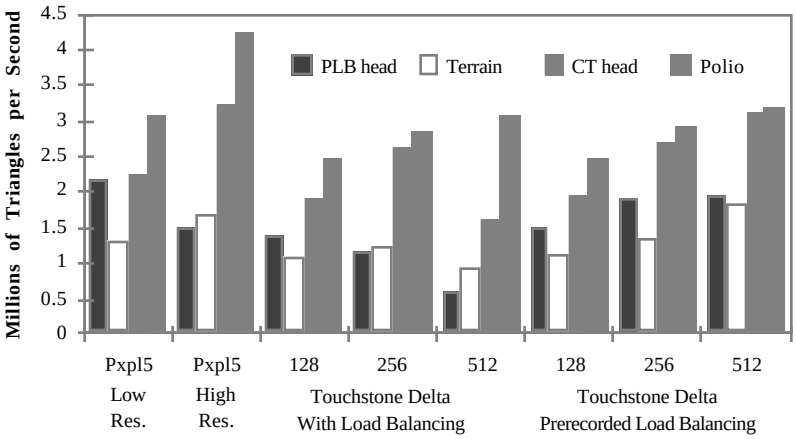


Figure 8.25 Performance comparison of Pixel-Planes 5 and the Intel Touchstone Delta. The values are the same as shown in table 8.2.

triangles per second while the Delta can render 16,000 triangles, which makes the Pixel-Planes processors 20% faster (all the performance figures assume 128x128 pixel regions). Also, the Pixel-Planes renderers can rasterize triangles much faster than the Delta i860s. Renderers can rasterize 50-pixel triangles at 150 K triangles per second while the Delta can rasterize 20 K triangles per second (excluding setup on the Delta), 7.5 times faster—before the difference in receive cost is factored in. This means that, for 50-pixel triangles, 22 Pixel-Planes renderers are the equivalent of 165 Delta processors. The speed difference is smaller with 5 pixel triangles: a renderer can rasterize 170 K triangles per second while the Delta i860 can rasterize 56 K triangles per second. Here, the 22 renderers would be the same speed as 67 Delta processors.

4. **Increased bisection bandwidth**. Each of the 32 Delta network links (16 in each direction) crossing the bisection of a 512-processor partition can carry 10 Mbytes/sec, for a total of 320 Mbytes/sec. Since half of the triangles must cross the bisection, the maximum bandwidth for redistribution is 640 Mbytes/sec. However, one cannot achieve this maximum bandwidth when using the Delta's 2D mesh for globally redistributing triangles; roughly half is available [Dall90]. The nearly random communication pattern during redistribution means that messages that do not cross the bisection can block messages that do, reducing the available bandwidth. On the other hand, the Pixel-Planes 5 ring has a maximum bandwidth of 640 Mbytes/sec, with all the bandwidth available. The benchmark program used in section 8.3.2.2 to determine MEGA's redistribution speed achieved 98% of the peak speed when sending 4080 byte messages.

One could say that it is unfair to compare the two systems, as one is comparing a general-purpose system running one particular application with a system that was designed to run that application. One view of the

comparison is that it gauges the efficiency of general-purpose architectures when performing polygon rendering versus the ideal of a specialized design.

8.5 Possible PPHIGS Future Work

Perhaps the largest weakness in the parallel algorithm for Pixel-Planes 5 PPHIGS is that clumps of primitives cause rasterization bottlenecks in large systems, or in medium-sized systems when producing NTSC-resolution images. As mentioned earlier, the rasterization bottlenecks could be reduced or eliminated by modifying the software. Three different methods follow:

1. Add additional sets of buckets so multiple frames can be in the rasterization stage. This method will allow the GPs and renderers to continue working as long as the workloads are balanced between the sets of processors. It increases the amount of memory required since one or more frames of renderer commands would be stored for the bottleneck regions. To implement this method the parallel rendering software (known as the Rendering Control System) will need to be changed. However, the software that converts primitives into renderer commands would not require modification.

2. Assign multiple renderers to the bottleneck regions by subdividing the regions in screen space. One renderer would be assigned to each smaller region. This would increase the performance even though this in theory wastes some processing power by not using all the pixel processors. Since it is more efficient to fix the region boundaries before the geometry processing (as opposed to a two-pass method), determining the regions to be split should be done using coherence much like the adaptive region boundary methods described in section 6.2.3.2. That is, the region boundaries for frame i are determined at the end of frame i–1. Primitives that overlap more than one of the smaller regions will have to be rendered by more than one renderer, reducing the speedup. This method would need new, more complex (and thus slower) bucketization software. For speed reasons, the current bucketization code is replicated in several different assembly language routines for different primitives, increasing the work to make the changes. The parallel rendering software would also require modification.

3. Assign multiple renderers to the bottleneck regions by subdividing the set of primitives. Each renderer would render primitives generated by a portion of the SGPs. One of the renderers assigned to each region would be designated to receive the final color and depth information from each of the other renderers. Once it has all the values, it composites them together and sends the result to the frame buffer. To reduce the communication time, the renderers shade their pixels before sending them to the designated renderer instead of sending the per-pixel state. The per-pixel state is much larger than a color and z value: it contains the surface parameters, a normal vector, and perhaps a texture coordinate. Shading the pixels beforehand reduces the communication time from 5.6 to 1.6 ms. This time is increased if more than two renderers are assigned to a region because the designated renderer can only receive one set of pixels at a time. Fortunately, the communication time can be overlapped with calculation by giving the secondary processors less work to do than the designated renderer. This will make them finish earlier than the designated renderer so they can send their pixels early enough so they arrive before they are needed. If other regions are waiting to be rendered, the secondary processors can work on them while the pixels are being sent. This third method only requires changes in the parallel rendering software. Its main disadvantage is that it increases the amount of communication, and that pixels will have to be shaded more than once. However, if the granularity ratio is reasonable (above 2) there can only be a few bottleneck regions, so the amount of extra communication and work should be limited.

The first method does not appreciably increase the processing costs but would produce an algorithm with more latency than the other methods. The second method is slightly more efficient than the third method if the primitives are fairly small, but would require much more work to complete the modifications. So, the third method is the one that would produce the best overall results while also minimizing the time required to make the changes. It will allow much better performance when calculating low resolution images with a large system.

The amount of extra work taken by the third method would generally be small. Shading the split regions makes up the bulk of the extra processing. Figure 8.26 shows the number of extra regions required to remove the region bottlenecks. It shows that more extra regions are needed when computing NTSC resolution images because the bottlenecks are worse. In many cases, a small number of regions should be split into more than two regions. With 24 renderers, the largest number of renderers available in systems having

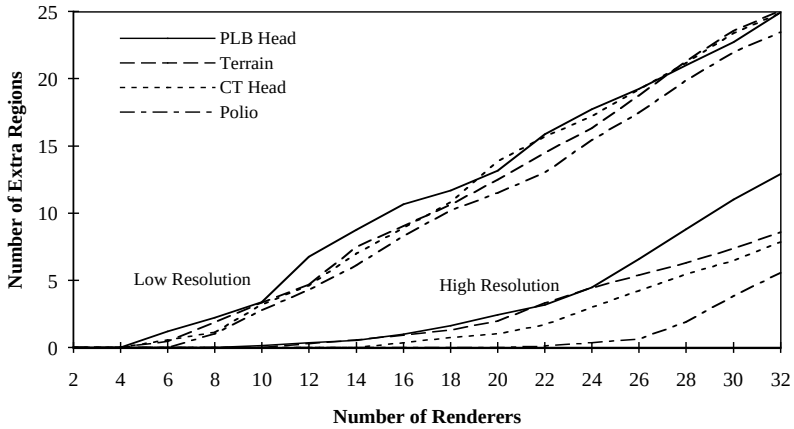


Figure 8.26 Number of extra regions rendered when region bottlenecks are removed by splitting the region's triangles over more than one renderer averaged over the sequence of frames for the four models, and for different numbers of renderers.

about a 2:1 ratio of GPs to renderers, the largest number of extra regions required to remove the bottleneck of a single region is 5; that is, the region had to be rendered by 6 renderers instead of 1.

We can calculate the worst-case percentage of extra work by assuming a high frame rate. With 24 renderers computing a high-resolution image, the model with the largest number of extra regions is the PLB Head model, with an average of 4.5 extra regions. Since shading takes 0.5 ms, the extra work takes a total of 2.25 ms. Because is amortized over the 24 renderers, it takes 94 μs of the frame time when running at the maximum 24 Hz. This is only 0.2% of the frame time. The extra work takes at most 2.2% of the frame time when generating NTSC-resolution frames at 60 Hz since the largest average number of extra regions is 17.7.

The above calculations assume that each renderer can work on a split region for the entire frame time. This is not true as some renderers will have to finish early so the computed pixels can be sent to the designated renderer early enough so that they arrive by the time that renderer finishes its region. In the worst case, with 5 additional regions to be sent to the designated processor, the first renderer will have to finish 8 ms before the designated renderer finishes so that there is time for all 5 regions to be sent. This would require the region be split even more to allow some renderers to finish early. In some cases transmitting the split regions will be the bottleneck, but this would only happen when running at very high frame rates.

Unfortunately, there are no current to implement any of these changes as Pixel-Planes 5 is now several years old, and the forthcoming PixelFlow system should give much higher performance.

8.6 Summary

This chapter has explored the use of specialized rasterization hardware in multicomputers. One example of specialized rasterization hardware was shown to have a price-performance advantage of three over general-purpose hardware. That advantage must be balanced with the expected utilization of the rasterization hardware. If the system only spends a small fraction of time doing rendering, the rasterization hardware will be largely unused, reducing or eliminating the price-performance advantage.

Systems designed for rendering, such as Pixel-Planes 5, are more efficient when scaled to large systems. By examining figures 8.18–23, we can see in high-resolution mode the performance per renderer drops about 15% between the smallest and largest systems, which is near a linear speedup. The corresponding SGP performance has not been measured but the speedup should not be as linear. Even so, it should be better than the Intel Touchstone Delta, as the largest model (Polio) had a parallel efficiency of only 26% when running on 512 processors. However, Pixel-Planes 5 is currently not as capable at dealing with per-region bottlenecks as the Delta since the latter allows an arbitrary number of regions. The analysis in this chapter showed the extent of these bottlenecks, which could be removed if sufficient resources were found. If those bottlenecks were removed, Pixel-Planes 5 would increase its price-performance and overall speed advantage compared to the Touchstone Delta. Even so, Pixel-Planes's 5 peak performance of 4.7 million triangles per second was the highest reported performance for several years after its introduction.

CHAPTER NINE

CONCLUSIONS AND FUTURE WORK

9.1 Summary and Conclusions

This dissertation has presented an investigation of algorithms for interactive visualization on multicomputers, with an emphasis on systems having tens or hundreds of processors. We have identified a new polygon rendering algorithm for multicomputers, one that uses the frame-to-frame coherence of the distribution of polygons across the screen to reduce the time required to perform load-balancing. An implementation of the algorithm gives high performance, 3.1 million triangles per second on the Intel Touchstone Delta using 512 processors. This performance is for a real-world model averaged over a series of frames, and includes back-facing triangles. This rendering rate was the highest known to the author for Intel parallel supercomputers when the implementation was first completed. Later implementations have reported higher rates, as described at the beginning of Chapter 7.

The dissertation also describes a brief investigation into rendering algorithms suitable for heterogeneous multicomputers that have specialized rasterization processors. An algorithm similar to the above also gives high performance on Pixel-Planes 5, giving an average speed of 4.2 million triangles per second and a peak speed of 4.7 million triangles per second. When the system was introduced, this performance was nearly four times the speed of the fastest available commercial workstation, as measured by the "head" benchmark that is part of SPEC Graphics Performance Characterization Group's Picture Level Benchmark series.

The investigation has also revealed that the new implementation does not always work well with hundreds of processors. The active load-balancing method is the bottleneck when rendering the smaller models with 128 to 512 processors. Even bypassing the bottleneck, the parallel efficiency (the percentage of the maximum theoretical speedup) is low: with 512 processors, the largest model runs at 25% parallel efficiency; the smallest model runs at 5% efficiency. The main reason for the poor efficiency is the high communication cost, which appears to be unavoidable when using the Touchstone Delta.

Specifics of the Chosen Algorithm. The polygon rendering algorithm was chosen by modeling the performance of different classes of algorithms. The performance models pointed toward sort-middle algorithms as the best current choice. Then, the specific algorithm was chosen by analyzing specific components of the algorithm. The chosen algorithm has the following characteristics:

- All processors participate in the two sequential stages of the graphics pipeline, geometry processing and rasterization, instead of having separate sets of processors perform each stage.
- Redistribution of triangles is performed throughout the frame to make full use of the available bandwidth.
- The redistribution method varies with the number of processors and model size. Small systems use one-step stream-per-region redistribution, as it minimizes memory use. Larger systems use one-step stream-per-processor redistribution since it reduces the minimum number of messages required, thus reducing the overhead. Systems with hundreds of processors use two-step stream-per-processor redistribution, which reduces the number of messages required from $O(N^2)$ to $O(N^{3/2})$, but requires that the triangles be sent twice. This fairly complex redistribution method gives dramatically faster results (up to 25 times faster) than naïve ones when using hundreds of processors.

The rasterization load-balancing method reduces the cost of performing the load-balancing calculations by performing them in parallel with the rasterization. This technique is successful at hiding the cost with smaller systems, up to about 100 processors, depending on the size of the model. Beyond that point, the $O(N \log N)$ cost of the load-balancing calculations increases enough to make them the overall bottleneck. The specific load-balancing method was chosen by comparing several different methods and parameters using a simulator combined with a performance model. The simulator transforms the chosen model and executes the load-balancing algorithm, and then it computes values for the amount of duplicated processing and percentage of time that the processors have useful work to perform. Those values are then placed in the performance model which is used to compute the overall frame time.

No one method or parameter set performed the best in all cases. The best compromise for most systems uses adaptively sized regions with four regions per processor. The regions are assigned by a single processor using a greedy bin-packing algorithm and are then used for the next frame. The region boundaries are only changed when necessary: when one region is a rasterization bottleneck. This method depends that there be frame-to-frame coherence of the distribution of triangles across the screen. Other methods make the assignments and use them in the same frame; thus, they do not depend on the coherence. These methods balance the loads among the processors more evenly but also have a higher load-balancing cost. For the chosen load-balancing method and the example geometric models, depending on coherence results in a small decrease in performance when using a few processors. When using many processors, the performance decrease due to using coherence is much less than the increase resulting from the reduced load-balancing costs.

When using 512 processors, the best performing algorithm uses a fixed, static assignment of regions to processors so that each processor has regions distributed across the screen. The break point between using active and static load-balancing depends on the frame rate and the system's communication speed. Lower frame rates and communication costs drive the trade-off point towards active load balancing.

Design Methodology. The methodology used to choose the overall algorithm can be used to choose an algorithm for other systems. With similar systems, the algorithm's overall structure remains the same, but the other choices made using the performance models should be verified by examining the results of the model using timing values taken from the new system. The simulations used to choose a load-balancing method may not have to be repeated if the following two conditions are true: (1) most of the rasterization cost comes from the per-triangle cost instead of per-pixel or per-region costs, and (2) the simulated data sets are representative of the new application. If the simulations are reused, only a small amount of work is required to measure the rasterization and communication speeds of the new system. Those values can then be used in the performance model to choose a load-balancing algorithm.

Load-Balancing Geometry Processing. Load-balancing methods for the geometry processing were improved over earlier work by adding randomization to an earlier algorithm used for partitioning the geometric model among the processors. The new algorithm avoids load imbalances when the model's mesh dimension or tessellation size is close to a multiple of the number of processors. When that happens, some processors are given a disproportionate share of the triangles that are off-screen or back-face culled. These cases seem to appear surprisingly often with algorithmically generated models.

Specialized Rasterization Processors. Systems that use specialized rasterization processors were also examined. The price-performance ratio of specialized rasterization processors versus general-purpose processors was estimated by computing this ratio for one system, Pixel-Planes 5. Its specialized rasterizers have three times the per-dollar performance of its general-purpose processors. The differences in parallel algorithms for systems with specialized processors and systems with general-purpose processors were briefly discussed. The main result is that care must be taken to avoid bottleneck regions during rasterization; these regions are caused by hot spots in the distribution of triangles across the screen. This problem is seen more often with large systems, as seen in the analysis of Pixel-Planes 5. That system and the sort-middle implementation on the Delta are compared. While the systems' largest configurations have comparable performance, the Pixel-Planes system requires much less hardware. The difference is due to using specialized rasterizers as well as a 12-times difference in the processing time to redistribute a triangle between processors.

9.2 Future Work

As the research progressed, some topics appeared promising for further research. The topics are grouped into six areas:

- non-sort-middle algorithms
- new methods for partitioning the model
- improvements to the chosen sort-middle algorithm
- new load-balancing algorithms
- improvements to the sort-middle implementation
- improvements to the performance model

Non-Sort-Middle Algorithms. The investigation of the classes of parallel algorithms showed that algorithms other than those in the sort-middle category have promise for different or future applications. Sort-first algorithms appear to offer significant communication savings compared to sort-middle, only requiring an average of 5 to 20% of the triangles to be redistributed. Since every processor can still potentially send data to every other processor, these savings do not reduce the per-message communication costs that dominate the overall costs when using hundreds of processors to render smaller models with a sort-middle algorithm. However, it does reduce the per-byte communications costs, which take up to 25% of the overall time. Mueller [Muel95] is currently investigating sort-first algorithms but the area is still largely unexplored.

Polygon sizes will continue to decrease as models get more complex. When each triangle covers less than five samples it will take less communication to send the samples instead of the triangle. At this point, sort-last-sparse algorithms will offer reduced communication and processing. This was recently demonstrated by Lee et. al. [Lee95].

Hybrid algorithms also hold promise. A visualization system could use a sort-first algorithm for the static portion of the model, using coherence to reduce the overall communication, but use a sort-middle algorithm for the portion of the model that changes from frame to frame. Such a system would have the benefits of each type of algorithm.

New Model Partitioning Methods. The distribute-by-primitive partitioning method described in Chapter 5 does not work well with small structures. Additionally, when using many processors the partitioning must minimize placing attribute changes on multiple processors to avoid large amounts of repeated work. However, when the number of attribute changes is reduced, it is much more complicated to make changes to the display list. The distribute-by-structure method has some potential for solving or reducing these problems because it does not require every processor to process every structure with the correct inherited attributes.

Given the difficulty of partitioning hierarchical display lists, perhaps a better solution would be to use a new API better suited for parallel traversal. One possibility would be to have a single hierarchy per processor. A second possibility would be to restrict the structure of the display list. For example, IRIS Performer [Rohl94] only allows primitives in leaf nodes, and only allows attribute changes in interior nodes. Such restrictions would make the distribute-by-structure method much simpler and efficient to implement.

Research is also needed for parallel immediate mode interfaces. Some global parameters, such as window size, image destination, hither distance, and anti-aliasing mode, must be the same on every processor. The Delta implementation does not check the values for consistency, but it should. It is not clear how to do the checking efficiently. The problem is much more difficult when deferred shading is used since the shading parameters must be coordinated across the processors. The research group developing the successor to Pixel-Planes 5, PixelFlow, is currently working on these and other problems.

Message Size Design Choice. Of the sort-middle options studied, the message size was investigated the least. The experiments showed that using fixed 4 Kbyte message buffers was not the best choice for smaller systems. Since larger messages delay the start of redistribution, using larger messages could force the redistribution to start so late that it becomes the bottleneck. A performance model could be developed to identify redistribution delays and thus be used to evaluate different message sizes, as done by Crockett and Orloff [Croc93]. The model could be based on the one developed to incorporate the load-balancing time for the between-frames load-balancing method (in section 6.1.2.2.2). The new model would calculate

the amount of redistribution done during rasterization and add additional time if redistribution is the bottleneck.

New Load-Balancing Algorithms. In Chapter 6, no single granularity ratio appeared to be the best; instead, each model had a range of optimum granularity ratios that depended upon the number of processors. This suggests adaptively varying the granularity ratio based on the current distribution of triangles on the screen. However, it is difficult to come up with a metric to specify the number of regions, since it must consider overhead due to polygon overlap along with balancing the loads. A simple adaptive scheme could work better, perhaps one that uses a fixed maximum ratio but decreases it when the distribution of triangles across the screen is fairly even.

Other load-balancing algorithms hold some promise. The algorithms that vary the region boundaries balance the loads better than the ones that used fixed region sizes, but also have higher overhead. The adaptive algorithms used in this dissertation require that the regions form a regular grid, which may force regions to be smaller than necessary around congested areas. Removing the region-boundary restriction might reduce the overhead by not placing extra region boundaries around congested areas. Finally, the algorithm that only adjusted the regions when necessary only did so after one of the regions had been discovered to be the bottleneck; it might be better to adjust the boundaries when one region is close to being the bottleneck, thus avoiding the problem before it occurs.

The size-based cost estimates could also use improvement. The number of pixels in each triangle that are also within a given region was not well correlated with the predicted value based on the size of the triangle's bounding box that is also within the region. When the triangle is large compared to the containing region, many of the pixels in the bounding box were also in the triangle. The fraction of pixels both in the bounding box and triangle is smaller with smaller triangles. This suggests using the fraction of the region included in the triangle's bounding box as another variable in the formula used to calculate the estimate of each triangle's size.

One caveat for developers of new load-balancing algorithms is that the current algorithm has an average processor utilization of 84% (the utilization ranges from 51 to 91%), making large improvements unlikely. Also, better size-based cost estimates are only necessary with larger polygons. The current cost estimates work well with small polygons.

Implementation Improvements. The sort-middle implementation on the Touchstone Delta currently does not implement the hybrid load-balancing algorithm identified in Chapter 6. That algorithm automatically switches between active and static load-balancing to avoid making the load-balancing calculations the bottleneck. The implementation also does not have enough features for extensive general-purpose use. More primitive types are needed: color-per-vertex polygons, lines, text, etc. The output capabilities need enhancement: saving images to disk is slow, and no provisions are made to view the images on remote workstations. The latter should compress the images to provide higher frame rates [Croc94]. Finally, higher quality shading, such as Phong shading and perhaps texture mapping, would be useful. Texture mapping would require research into how to store the texture maps without using too much memory. Other than features, the implementation could be ported to an up-to-date multicomputer, such as the Intel Paragon. The network bottleneck should be reduced on these systems since they have a higher communication capacity.

Performance Model Improvements. The current performance model could be improved by using a better model of the communication speed since the current model gives a pessimistic value when processors are exchanging messages. Finally, it would be useful to include bisection bandwidth limitations in the performance model. The changes required in the performance model would be similar to those for choosing a more efficient message size as mentioned above. Both of the improvements in the performance model would improve both the algorithm for choosing the number of groups to be used as well as the methodology for choosing a load-balancing method. Both the algorithm and methodology suffer from the current inaccuracies.

A second improvement in the performance model would take into account how the rasterization load-balancing algorithm influences how well the loads are balanced during geometry processing. The current performance model uses a value for the processor utilization during geometry processing. Only one value is used for each model configuration and number of processors. However, the actual processor utilization depends on how the screen is broken into regions, which in turn depends on the load-balancing method

used. Higher granularity ratios will increase the overlap factor, which increases the standard deviation of the time to process each triangle. The latter will increase the standard deviation of the processors' total geometry processing times, decreasing the utilization. The load-balancing methods that use adaptive region sizes also have higher overlap factors (see section 6.3.4), and will thus have a lower processor utilization during geometry processing. This improvement will allow more accurate comparisons of the load-balancing algorithms for both geometry processing and rasterization. However, I expect the improvement to be fairly small.

APPENDIX A

SAMPLE MODELS AND VIEWPOINTS

This appendix describes the sample models used in this dissertation for simulation and performance tests. Four of the models are used throughout Chapters 5 through 8. The four models are computer generated, which is a common characteristic of scientific visualization applications. Since the models were not adaptively generated according to surface curvature, the sizes of the triangles in each model are fairly uniform. The sizes of the models range by over a factor of 10, from 60,000 to over 800,000 triangles. The models are rendered as independent triangles even though two could easily be rendered as triangle strips.

Four of the models have a series of 80-100 views associated with it, each specifying the complete viewing frustum. I recorded the viewpoints for three of these models as I interacted with them on Pixel-Planes 5. The viewpoints of the PLB Head model were specified as part of the SPEC Graphics Performance Characterization Group's Picture Level Benchmark.

One model is only used in the comparison of cost functions in section 6.2.3.3.1. It was used because it contains polygons that vary considerably in area, which stresses the cost functions. The polygons also vary in number of vertices. Unlike the earlier models, it was made by hand. I recorded a series of 1,098 viewpoints while interactively viewing it on Pixel-Planes 5.

The sections that follow describe the models, giving some statistics and showing a selection of frames from the sequence of viewpoints. The statistics were recorded when displaying the models at a resolution of 640 by 512. Statistics for other resolutions can be computed by scaling the triangle sizes and heights appropriately. The first set of statistics is:

- Number of frames: number of frames in the series.
- Number of triangles: number of triangles in the model.
- Number of color changes: number of color changes in the model.
- Number of matrices: number of transformation matrix changes in the model.
- Back-face culling: whether back-face culling is enabled.

The Well model description also contains the average number of vertices per polygon; this value is three for the first four models since they only have triangles.

The remaining statistics were computed for each frame in the sequence. The mean, minimum, and maximum values of the statistic are given.

- Rasterized triangles: number of potentially visible triangles, i.e. the number of triangles that are in the viewing frustum and are not back-face culled.
- Percent triangles rasterized: percentage of triangles in the model that are potentially visible.
- Triangles outside frustum: number of triangles that are outside the viewing frustum but are not back-facing.
- Triangles back-face culled: number of triangles that are back-facing and in the viewing frustum.

- Triangles outside and culled: number of triangles that were both back-facing and outside the viewing frustum.
- Mean triangle area: mean number of pixels in the potentially visible triangles. The area is calculated using a function based on the vertex locations, not by counting the number of pixel centers that the triangle overlaps. Only the on-screen portion of the triangle is considered.
- Maximum triangle area: triangle with the largest area as calculated by the above formula.
- Mean triangle height: mean vertical difference between the topmost and bottommost vertices of the on-screen triangles. Only the on-screen height is considered.
- Maximum triangle height: triangle with the largest height, calculated as above.
- Depth complexity: average number of triangles overlapping each pixel, calculated by dividing the sum of the triangle areas by the number of pixels.

A.1 PLB Head

Parameter	Value
Number of frames	81
Number of triangles	59,592
Number of color changes	0
Number of matrices	1
Back-face culling	Enabled

Parameter	Mean	Minimum	Maximum
Rasterized triangles	25,941	19,508	31,123
Percent triangles rasterized	43.53%	32.74%	52.53%
Triangles outside frustum	194	0	651
Triangles back-face culled	33,445	28,469	39,682
Triangles outside and culled	12	0	80
Mean triangle area	4.37	2.78	6.65
Maximum triangle area	170.53	98.57	241.29
Mean triangle height	3.61	3.27	4.05
Maximum triangle height	33.95	7.60	49.61
Depth complexity	0.334	0.264	0.396

Table A.1 Statistics for the PLB Head model.

The PLB Head model is a cylindrical scan of a human head, and was made with a laser range finder. The model and viewpoints are from the Graphics Performance Characterization Committee's Picture Level Benchmark. However, the lighting and image resolutions used in the performance runs do not correspond to the benchmark. The benchmark uses Phong lighting with three light sources and a 900 by 720 resolution while the implementation uses Gouraud lighting with one light and 640 by 512 resolution. The model appears courtesy of IBM AWD Graphics Lab, Austin, Texas.

Figure A.1 Selected frames from the PLB Head model sequence. The image to the left is frame 1. The 72 small images below are spaced evenly among the 81 frames in the sequence.

A.2 Terrain

Parameter	Value
Number of frames	101
Number of triangles	162,690
Number of color changes	71,436
Number of matrices	1
Back-face culling	Disabled

Parameter	Mean	Minimum	Maximum
Rasterized triangles	126,319	25,754	162,690
Percent triangles rasterized	77.64%	15.83%	100%
Triangles outside frustum	36,371	0	136,936
Mean triangle area	2.85	1.06	10.05
Maximum triangle area	19.47	3.70	67.77
Mean triangle height	1.67	0.84	4.02
Maximum triangle height	12.26	2.58	27.29
Depth complexity	0.705	0.525	0.932

Table A.2 Statistics for the Terrain model.

The Terrain model shows Landsat data of a portion of the Sierra-Nevada mountain range. The data appears courtesy of H. Towles, Sun Microsystems. The original model had one color per triangle. During conversion to an internal format, superfluous color changes—where the new color was the same as the old color—were removed. The model consists of 319 strips of 510 triangles, which were broken into individual triangles (triangle strip primitives were not used).

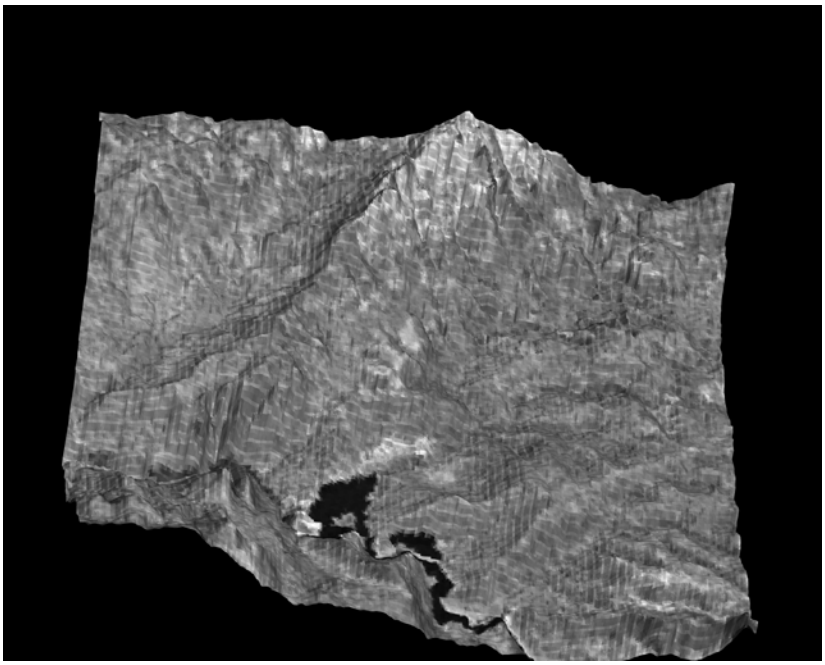
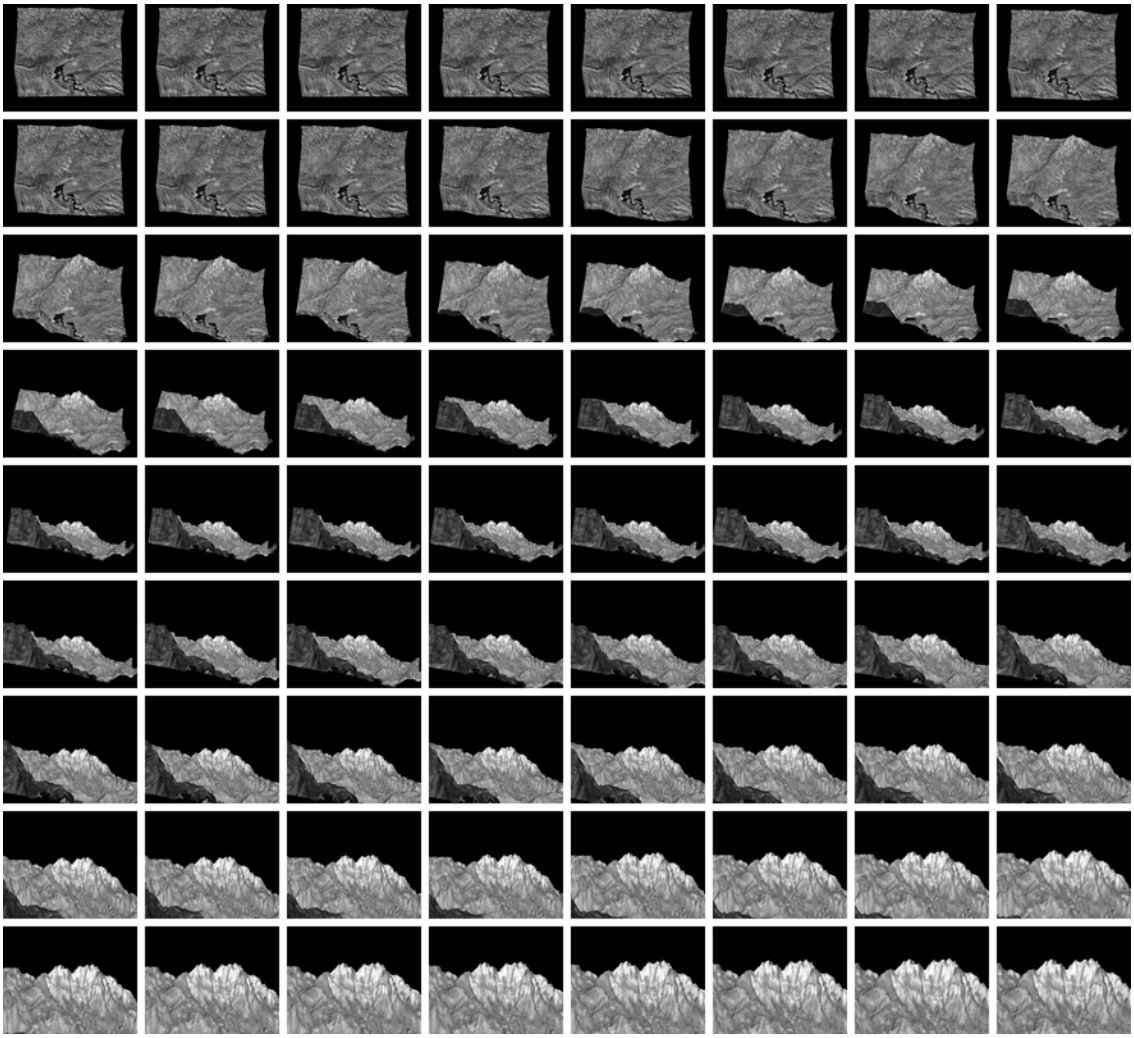


Figure A.2 Selected frames from the Terrain model sequence. The image to the left is frame 23. The 72 small images below are spaced evenly among the 101 frames in the sequence.



A.3 CT Head

Parameter	Value
Number of frames	92
Number of triangles	229,208
Number of color changes	1
Number of matrices	1
Back-face culling	Enabled

Parameter	Mean	Minimum	Maximum
Rasterized triangles	102,718	79,040	117,450
Percent triangles rasterized	44.81%	34.48%	51.24%
Triangles outside frustum	13,547	0	37,397
Triangles back-face culled	99,575	76,325	114,335
Triangles outside and culled	13,368	0	36,446
Mean triangle area	3.82	2.23	5.83
Maximum triangle area	16.38	9.95	23.24
Mean triangle height	2.24	1.89	2.80
Maximum triangle height	5.34	4.37	6.54
Depth complexity	1.131	0.794	1.494

Table A.3 Statistics for the CT Head model.

The CT head model shows an isosurface of a computed tomography (CT) scan of a child's head. The original volume data had a resolution of 128x128x124, and was converted into triangles using a marching-cubes program [Lore87]. The volume data is courtesy of Dr. Franz Zonneveld, Utrecht University Hospital, The Netherlands. The marching cubes program was written by Steve Lamont and Jim Chung.

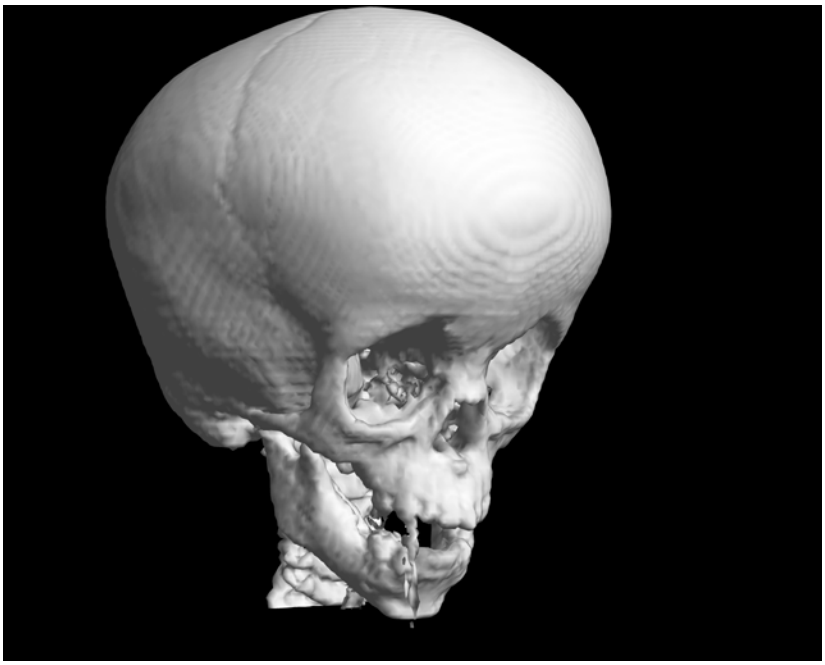


Figure A.3 Selected frames from the CT Head model sequence. The image to the left is frame 36. The 72 small images below are spaced evenly among the 92 frames in the sequence.

A.4 Polio

Parameter	Value
Number of frames	110
Number of triangles	806,400
Number of color changes	240
Number of matrices	0
Back-face culling	Enabled

Parameter	Mean	Minimum	Maximum
Rasterized triangles	377,899	284,348	449,183
Percent triangles rasterized	46.86%	35.26%	55.70%
Triangles outside frustum	18,496	0	115,201
Triangles back-face culled	390,935	285,532	446,407
Triangles outside and culled	19,070	0	121,319
Mean triangle area	6.67	3.18	12.43
Maximum triangle area	359.27	38.41	3038.09
Mean triangle height	2.54	1.77	3.13
Maximum triangle height	24.11	10.34	92.68
Depth complexity	7.37	4.36	11.68

Table A.4 Statistics for the Polio model.

The Polio model shows the outer shell of the polio virus. The 50,400 spheres of the space-filling molecular model were each tessellated into 16 triangles. The artifacts and the total number of triangles could have both been reduced by using adaptive tessellation of the spheres based on their screen size, but that would have increased the complexity of the simulations. The data appears courtesy of James Hogle, Marie Chow, and David Filman, Research Institute of Scripps Clinic.

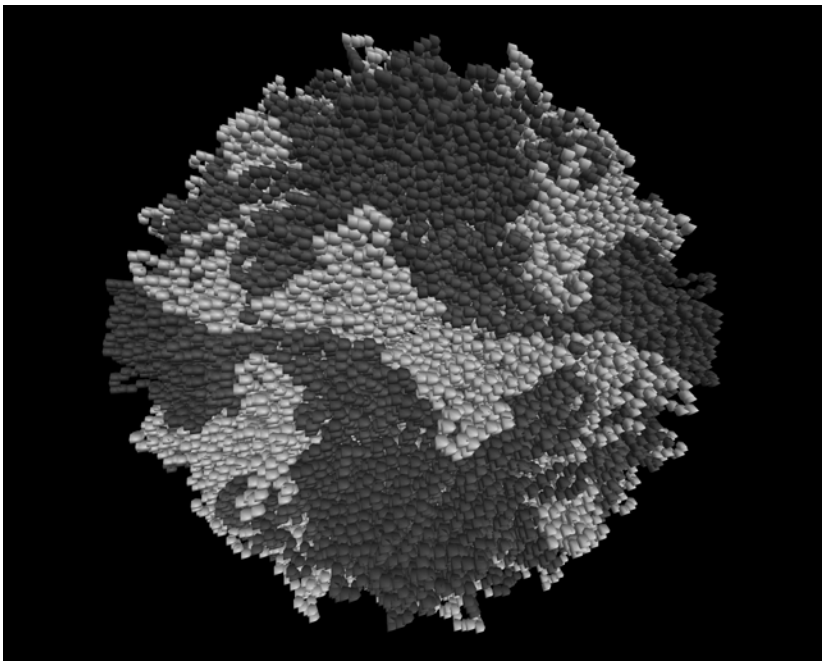
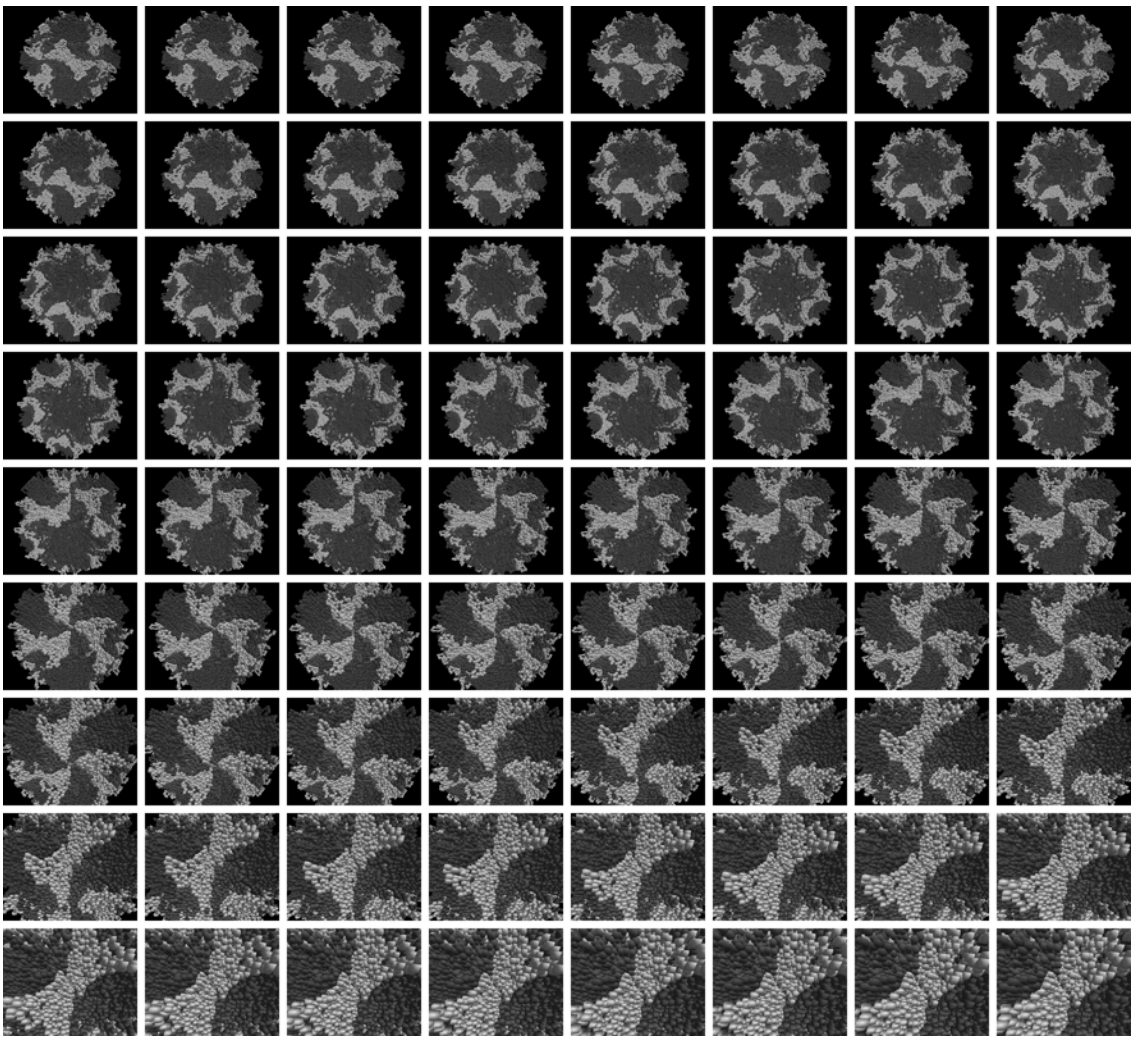


Figure A.4 Selected frames from the Polio model sequence. The image to the left is frame 1. The 72 small images below are spaced evenly among the 110 frames in the sequence.



A.5 Well

Parameter	Value
Number of frames	1,098
Number of polygons	356
Average number of vertices per triangle	4.48
Number of color changes	8
Number of matrices	0
Back-face culling	Disabled

Parameter	Mean	Minimum	Maximum
Rasterized polygons	282.57	0	356
Percent polygons rasterized	79.37%	0%	100%
Polygons outside frustum	73.42	0	356
Mean polygon area	2,140	0	25,014
Maximum polygon area	45,476	38.44	274,080
Mean polygon height	81.27	0	254.44
Maximum polygon height	344.17	0	512
Depth complexity	1.42	0	5.78

Table A.5 Statistics for the Well model.

The Well model shows the Old Well at the University of North Carolina at Chapel Hill. Unlike the other models, it contains many polygons with more than three vertices. This model is only used to evaluate cost estimates for large triangles in section 6.2.3.3.1. The model was made by Eric Grant.

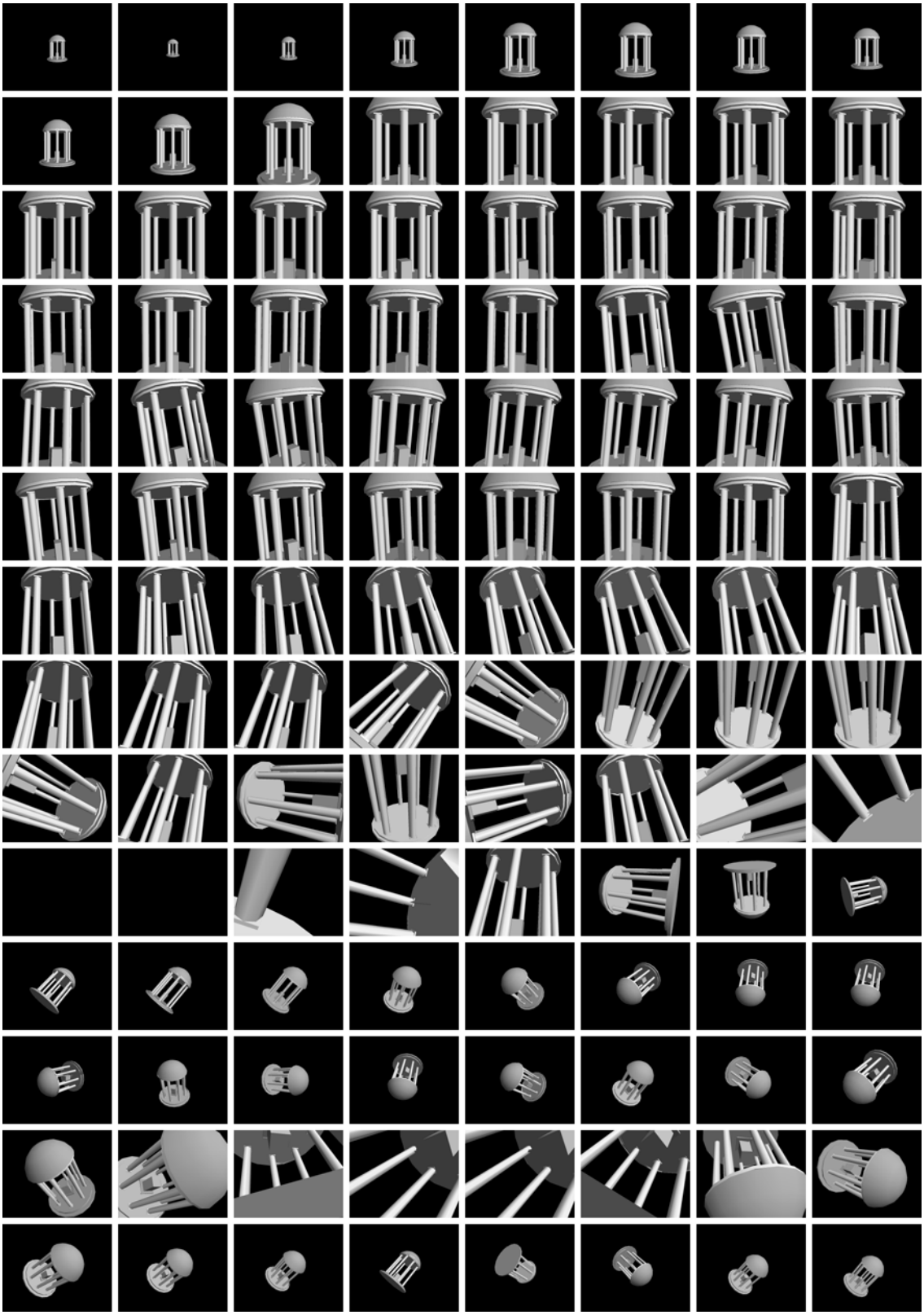


Figure A.5 Selected frames from the Well model sequence. The 112 frames shown are spaced evenly among the original 1,098 frames.

APPENDIX B

SOURCES OF DELTA TIMING VALUES AND SIZES

This dissertation uses many timing values and sizes in various performance models. This section describes the values and sizes, and how they were determined. Most of the timing values were measured, although some were estimated. All measurements were made on the Intel Touchstone Delta. The table in section B.1 describes the timing values, and is followed by more lengthy descriptions that do not fit in the table. Section B.2 describes the data structure sizes. The timing values for Pixel-Planes 5 are described in section 8.1.3.

B.1 Timing Values

Symbol	Value	Description	Source of Value
t_{bucket}	2.7 µs	time to determine the next buffer for a triangle and copy triangle to it	See section B.1.2.
t_{byte}	0.24	incremental per-byte communication time	Equal to $t_{byte_s} + t_{byte_r}$.
t_{byte_r}	0.13	incremental time to receive a byte in a message	See section B.1.1.
t_{byte_s}	0.11	incremental time to send a byte in a message	See section B.1.1.
t_{comp}	0.2 µs	time to composite a pixel	Estimated using cycle counts, and using the i860 graphics instructions at 40 MHz. Includes for every two pixels: 8 cycles for reading the old and new *z* values, 1 cycle to compare them, 6 cycles to (conditionally) write the results, and 1 cycle of overhead.
t_{extra_comp}	0.28	extra time needed to composite a sort-last-sparse pixel	Estimated using cycle counts, assuming two pixels composited at once using the i860 graphics instructions at 40MHz. Includes for every pixel: 1 cycle to read the new *z* value, 8 cycles to read the old *z* value (assumes page miss), 1 cycle for comparisons, 8 cycles to (conditionally) write the new colors, and 1 cycle of overhead. The value of t_{comp} is then subtracted from the total time.
t_{geom}	8.3	time to transform a triangle to screen space, clip test, and light it	See section B.1.2.
t_{geom_back}	6.1	time to transform a back-facing triangle (for both on- and off-screen triangles)	See section B.1.2.

Table B.1 Description of times, their values, and the source of the values (continued on next page).

Symbol	Value	Description	Source of Value
t_{geom_off}	6.6	time to transform a triangle off screen and in front of the near (hither) plane	See section B.1.2.
t_{line}	2.74	time to do scanline interpolations and calculations	See section B.1.3.
t_{msg}	504	message overhead	Equal to $t_{msg_s} + t_{msg_r}$.
t_{msg_r}	296 µs	time to receive an empty message	See section B.1.1.
t_{msg_s}	208	time to send an empty message	See section B.1.1.
t_{pixel}	0.23	time to rasterize a pixel	See section B.1.3.
t_{pix_ovhd}	0.078	time to initialize a pixel's color and *z* value	Measured time for initializing entire frame on implementation then divided by number of pixels in frame.
t_{pre_xform}	3.2	time to transform a triangle to screen space	See section B.1.2.
t_{tri}	18.8	time to set up a triangle for rasterization	See section B.1.3.
t_{refmt}	4.1	time to reformat a triangle between messages	Estimated as 1.5 times t_{bucket} as it uses approximately that number of instructions, and the operations are similar.
t_{reg_ovhd}	100	time needed for per-region overhead	Estimated.
t_{scan}	0.03	time needed to scan convert a pixel	Equal to $t_{pixel} - t_{comp}$. The likely reason for the small value is that the computations are overlapped with memory accesses.

Table B.1 Description of times, their values, and the source of the values (continued).

B.1.1 Communication Times

The communication times correspond to the CPU time necessary to send and receive messages. The send values are for asynchronous sends, and the receive values are for handler receives, which call a user-specified routine to handle the incoming message. Most of the send and receive operations in the implementation use these types of communication.

The times were measured using three processors. Each processor executed an identical loop consisting of first checking for message arrival, then checking whether a certain time has passed, and finally executing a linear series of no-ops to insure that user code is in the cache. The different processors perform different actions because they check for different message arrivals and time values. The values are set so that processor 0 sends messages at regular intervals to processor 1 until the end of the test, when it sends a different message to the other two processors. The loop values on processors 1 and 2 make them wait for the end of the test. The number of loop iterations performed by each processor varies, since they do different amounts of work. Processor 2 is the control processor: it establishes the number of iterations performed when not communicating. The measured loop counts on processors 0 and 1 are smaller. The total time used for communication by processors 1 and 2 is computed by subtracting the smaller loop counts from processor 2's count and dividing by the number of loops executed by processor 2 per second. The time per message is then the total time divided by the number of messages used in the test (5000).

This method was used to calculate the time required for messages having sizes of even powers of 2 from 1 to 16,384. The send and receive times were then each fitted to an equation having a constant term plus a per-byte term. The constant term corresponds to the per-message overhead. The resulting values for t_{msg_r}, t_{msg_s}, t_{byte_r}, and t_{byte_s} are shown in table B.1.

This method will overestimate the time required when processors are exchanging messages. The messaging model uses flow control to insure that a processor does not run out of memory for incoming

messages. When a user sends messages unidirectionally, as done in the benchmark procedure, the flow control messages must be sent as additional messages. However, when two processors exchange messages, the flow control information can be piggy-backed on the user's messages, thus reducing the per-message cost. I have not analyzed the amount or characteristics of the cost reduction.

B.1.2 Geometry Times

The geometry processing times were measured on a single processor. The normal geometry processing time t_{geom} and the bucketization time t_{bucket} were measured by first measuring the time needed to transform a triangle that falls into one region, t_1, and the time needed for a triangle that falls into 12 regions, t_{12}. Each measurement is the average processing time for a large number of identical triangles (16,384 and 2,000, respectively). The bucketization time t_{bucket} is $(t_{12}–t_1)/11$ and the geometry time is $t_1–t_{bucket}$. The values appear in table B.1. Unfortunately, identical triangles were used for each measurement, thus making the t_1 and t_2 times smaller than the normal case due to caching, and in turn making both t_{geom} and t_{bucket} smaller than the normal case.

The back-face culling and off-screen geometry processing times (t_{geom_back} and t_{geom_off}) were calculated by rendering the first frame of PLB Head model sequence using one processor and one region. The model was transformed with and without back-face culling enabled. The average displayed triangle rate was first computed by dividing the total time with culling disabled divided by the number of triangles. The per-triangle back-face time was computed by subtracting the product of the number of front-facing triangles and the average displayed triangle rate from the total time with back-face culling disabled, and then dividing by the number of back-facing triangles. The off-screen time was measured by transforming the model with all triangles off the side of the viewing frustum. The per-triangle time is the total time divided by the number of triangles.

The time needed to just transform a triangle, t_{pre_xform}, was estimated from time needed to perform all the geometry processing for a triangle falling into one region. The number of lines of assembler code was counted for all the geometry processing and for just performing the transformation. Assuming each line takes the same amount of time, the transformation time is the geometry time multiplied by the fraction of the lines of geometry code needed for to perform the transformation.

B.1.3 Rasterization Times

The rasterization times fit a performance model for triangle rasterization that has one value for the setup time plus another for each of the inner loops. The model is the equation $t_{tri} + t_{line}h + t_{pixel}a$, where h is the number of scan lines crossed by a triangle, a is the number of pixels in the triangle, t_{tri} is triangle setup time, t_{line} is the scanline interpolation time, and t_{pixel} is the pixel rasterization time. The values were determined by a least-squares fit of the model to the sizes and rasterization times (on the Touchstone Delta) for 3000 random triangles. The values of the constants are reported in table B.1. The average error between the actual and predicted rasterization times is 5%.

B.2 Data Structure Sizes

Symbol	Size	Description	Source of Size
s_{disp_tri}	56 bytes	size of a display (trans-formed) triangle; same as $s_{tri.}$ (only used in Chapter 3)	Uses 2 bytes for primitive type, 2 bytes for primitive size, 2 bytes for number of vertices, 2 bytes for region id, and, for each of the three vertices, 4 bytes for each of x, y, and *z* and 4 bytes for color.
s_{full_samp}	8	size of a sample used in sort-last full	Uses 4 bytes for color plus 4 bytes for *z*.
s_{msg}	4096	size of a message	See section 4.4.6.
s_{tri}	56	size of a triangle; same as s_{disp_tri}	See source of s_{disp_tri}. Used in Chapters 4-7 since sort-middle algorithms only redistribute display triangles.
s_{raw_tri}	88	size of a raw (untrans-formed) triangle	Uses for each triangle 2 bytes for primitive type, 2 bytes for primitive size, 2 bytes for number of vertices, 2 bytes for region id and 4 bytes for each of front and back color. For each of the 3 vertices, 4 bytes for each of *x*, *y*, *z*, *x* normal, *y* normal, and *z* normal.
s_{sparse_samp}	12	size of a sample used in sort-last sparse	Uses 4 bytes for color, 2 bytes for each of *x* and *y*, plus 4 bytes for *z*.

Table B.2 Description of data structure sizes and the source of the sizes.

APPENDIX C

SIMULATION AND MODELING RESULTS

This section contains the results of the simulation and performance modeling used to evaluate the load-balancing methods. The simulator and the performance model are described in section 6.1, and the different load-balancing methods are described in section 6.2. The results are presented as a series of charts.

The data is also available in electronic form. The University library has the results on a 3.5-inch high-density floppy disk formatted for PC compatibles. The data can also be obtained by contacting the author.

C.1 Simulation Results

The simulation runs evaluated load-balancing methods for each of the region assignment styles. Two load-balancing methods were evaluated for the static assignment style, and three methods were evaluated for the two once-per-frame assignment styles, between-stages and between-frames. Different simulation runs for the once-per-frame load-balancing methods evaluated per-region cost estimates based on the number of triangles in each region, as well as estimates based on the size of each triangle's bounding box. Finally, three different load-balancing methods were evaluated for the work-queue assignment style. Both the values for the processor utilization and the overlap factor are presented.

Each chart of processor utilization presents results for a range of numbers of processors and granularity ratios. The static and work-queue load-balancing methods are each presented using one chart per model. The other methods are each presented using two charts per model, one set for between-stages region assignment and another for between-frames assignment. Each curve in each chart gives the results for a constant granularity ratio, and is labeled with that number. In some charts it is hard to associate the labels with the curves because the curves are so close together. Many of the ambiguous curves can be identified by noting that increasing the granularity ratio usually increases the processor utilization, especially with 128 to 512 processors. The charts for the static load-balancing methods have curves for granularity ratios of 1, 2, 4, 6, 8, 12, 16, and 32, while the other charts have curves for ratios of 1, 2, 4, 6, 8, 12, and 16. In the charts with two sets of curves, the labels for the between-stages curves are attached at $N = 256$ while the labels for the between-frames curves are attached at $N = 128$.

The overlap factor charts are similar to the utilization charts but only one set of curves is shown per chart. Some of the load-balancing methods have the same overlap factors. A table before each series of charts shows the mapping from load-balancing method and region assignment style to overlap chart.

The charts are organized into four major sections, with one section for each model. Within each section, a subsection contains the charts for the static, once-per-frame, and work-queue assignment styles. The overlap factor charts for each model are in a separate subsection following the model's utilization charts.

C.1.1 PLB Head Model

C.1.1.1 Methods for Static Region Assignments

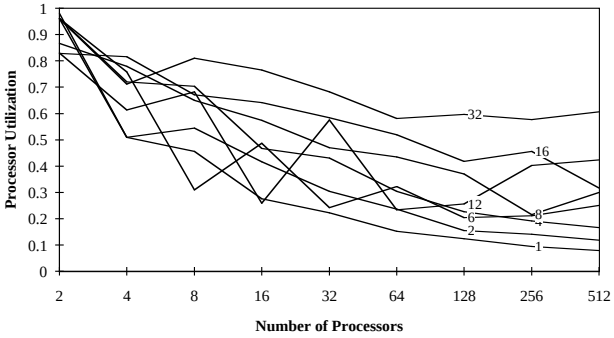


Figure C.1a Simulated processor utilization for the PLB Head model using modular static load-balancing.

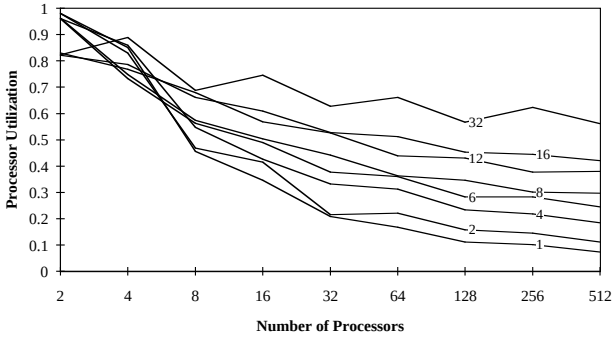


Figure C.1b Simulated processor utilization for the PLB Head model using interleaved static load-balancing.

C.1.1.2 Methods for Once-Per-Frame Region Assignments

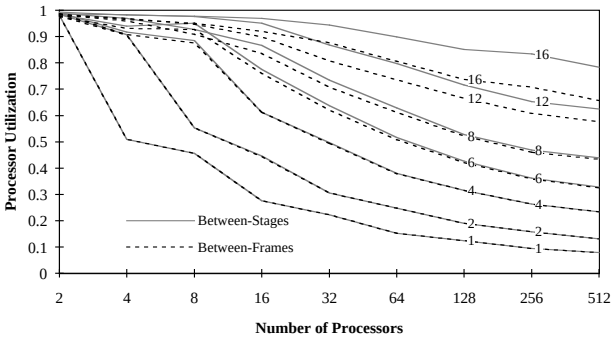


Figure C.2a Simulated processor utilization for the PLB Head model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

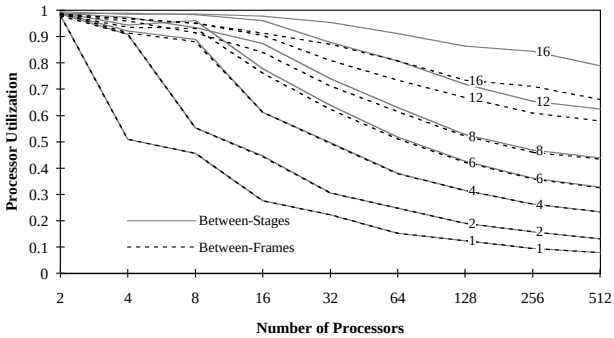


Figure C.2b Simulated processor utilization for the PLB Head model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates

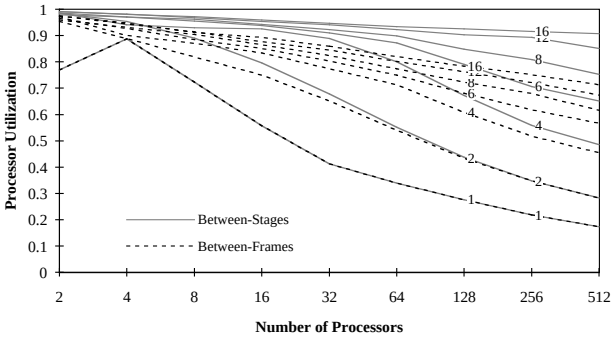


Figure C.2c Simulated processor utilization for the PLB Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

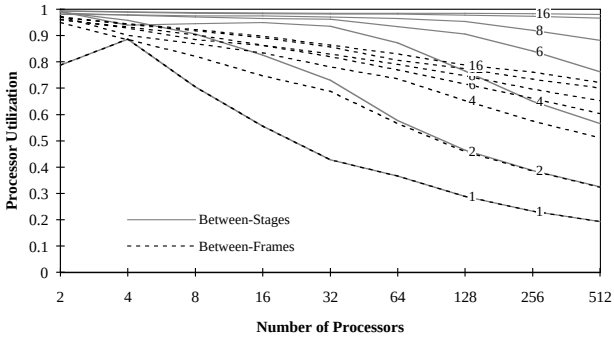


Figure C.2d Simulated processor utilization for the PLB Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

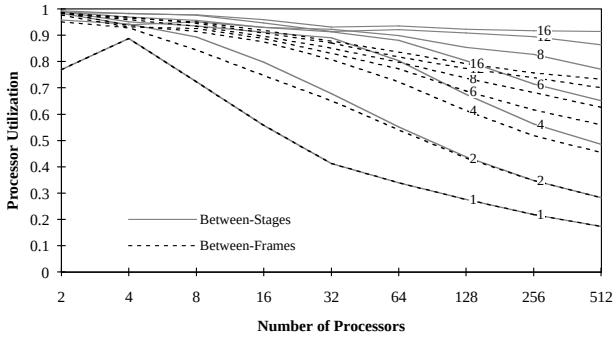


Figure C.2e Simulated processor utilization for the PLB Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

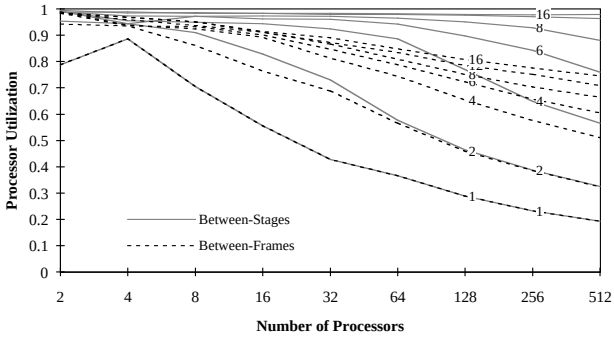


Figure C.2f Simulated processor utilization for the PLB Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.1.1.3 Methods for Work-Queue Region Assignments

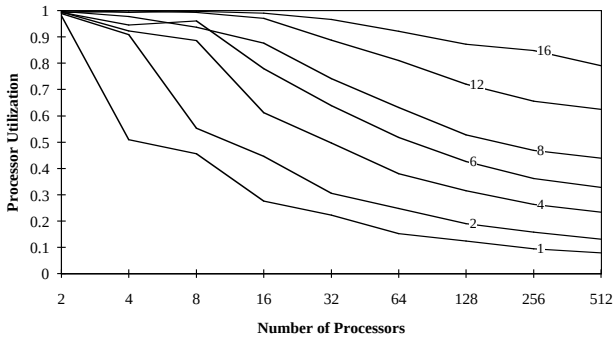


Figure C.3a Simulated processor utilization for the PLB Head model using work-queue-assigned fixed-region load-balancing.

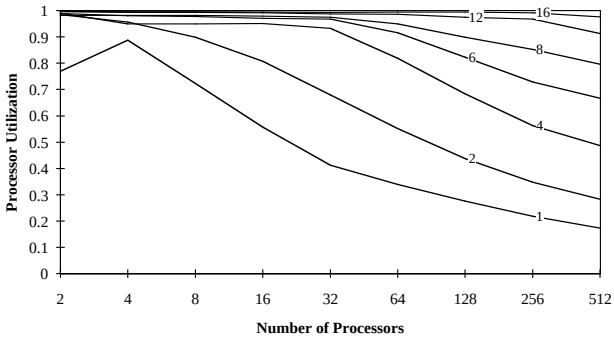


Figure C.3b Simulated processor utilization for the PLB Head model using work-queue-assigned per-frame adaptive-region load-balancing.

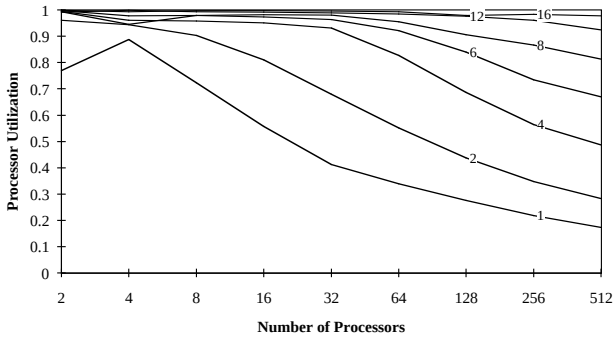


Figure C.3c Simulated processor utilization for the PLB Head model using work-queue-assigned when-needed adaptive-region load-balancing.

C.1.1.4 Overlap Factor Values

There are fewer overlap factor charts than processor utilization charts because many load-balancing methods share region boundaries. Nearly all the load-balancing methods that use fixed region boundaries have the same overlap factors. The static interleaved load-balancing method uses different region boundaries so it has its own chart. More charts are needed to give the values for the adaptive methods since the region boundaries depend on the method for changing the boundaries as well as the cost estimate function. However, the values are the same for both once-per-frame region assignment styles since all the methods determine the boundaries for the next frame in the current frame. Table C.1 summarizes the mapping from load-balancing method to overlap chart.

Region Assignment Style	Load-balancing Method	Cost Estimate Function	Overlap Figure
Static	Modular	—	C.4a
Static	Interleaved	—	C.4b
Once-Per-Frame	Fixed Region	Triangle Counts	C.4a
Once-Per-Frame	Fixed Region	Size-Based	C.4a
Once-Per-Frame	Per-Frame Adaptive-Region	Triangle Counts	C.4c
Once-Per-Frame	Per-Frame Adaptive-Region	Size-Based	C.4d
Once-Per-Frame	When-Needed Adaptive-Region	Triangle Counts	C.4e
Once-Per-Frame	When-Needed Adaptive-Region	Size-Based	C.4f
Work-Queue	Fixed Region	—	C.4a
Work-Queue	Per-Frame Adaptive-Region	—	C.4g
Work-Queue	When-Needed Adaptive-Region	—	C.4h

Table C.1 Overlap figures mapping table for the PLB Head model.

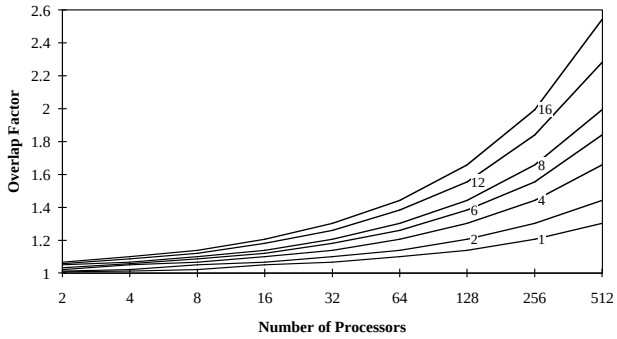


Figure C.4a Simulated overlap factor values for the PLB Head model for load-balancing methods using fixed region boundaries (see table C.1).

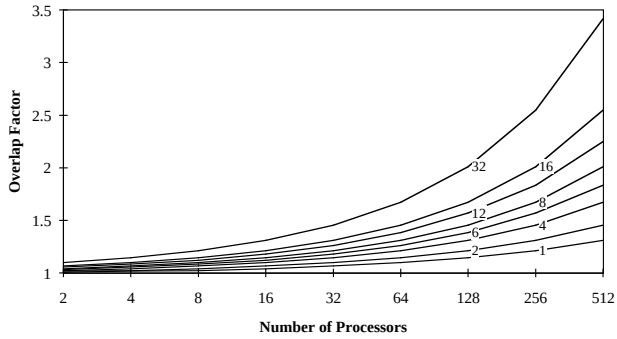


Figure C.4b Simulated overlap factor values for the PLB Head model for static interleaved load-balancing.

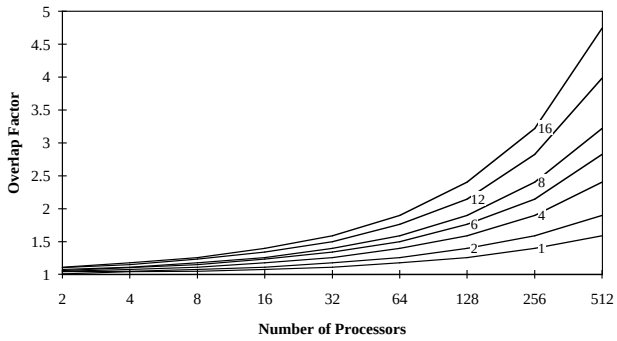


Figure C.4c Simulated overlap factor values for the PLB Head model for once-per-frame-assigned per-frame adaptive-region load-balancing using triangle-count cost estimates.

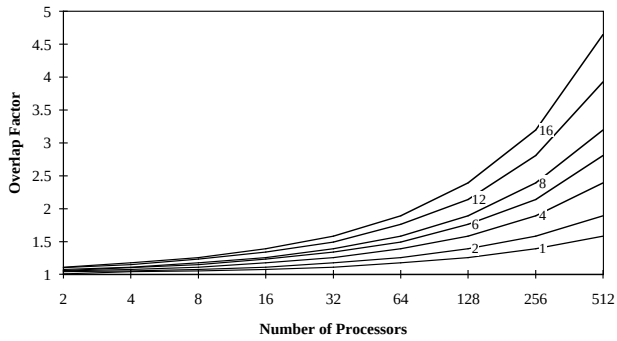


Figure C.4d Simulated overlap factor values for the PLB Head model for once-per-frame-assigned per-frame adaptive-region load-balancing using size-based cost estimates.

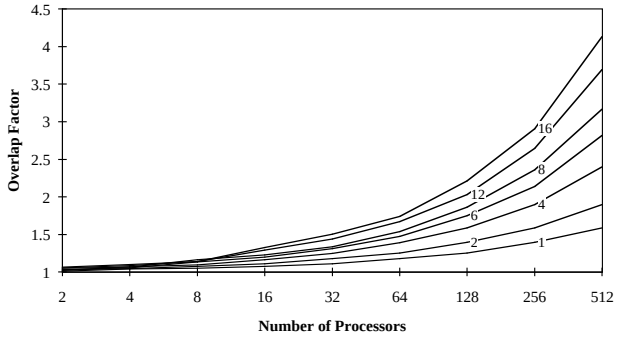


Figure C.4e Simulated overlap factor values for the PLB Head model for once-per-frame-assigned when-needed adaptive-region load-balancing using triangle-count cost estimates.

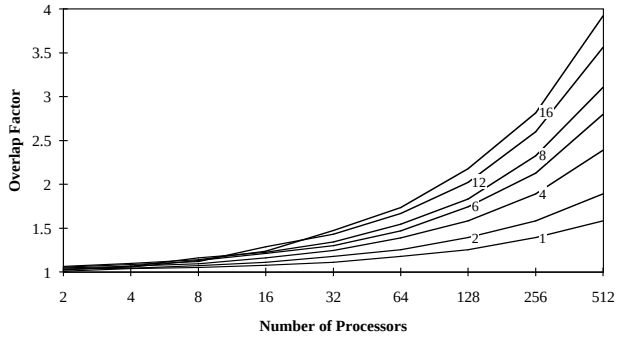


Figure C.4f Simulated overlap factor values for the PLB Head model for once-per-frame-assigned when-needed adaptive-region load-balancing using size-based cost estimates.

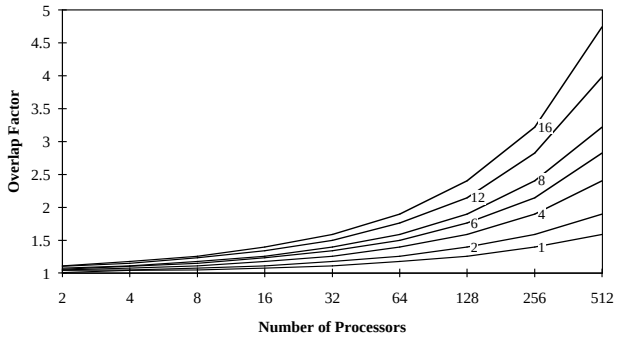


Figure C.4g Simulated overlap factor values for the PLB Head model for work-queue-assigned per-frame adaptive-region load-balancing.

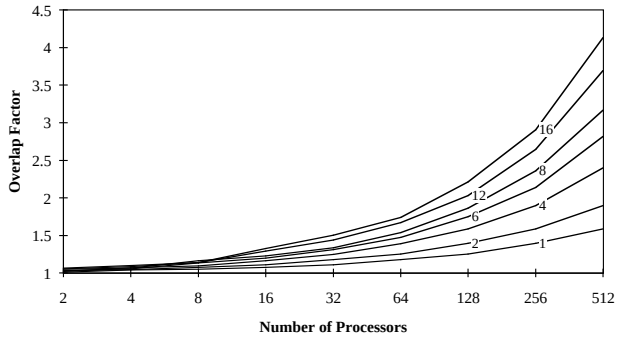


Figure C.4h Simulated overlap factor values for the PLB Head model for work-queue-assigned when-needed adaptive-region load-balancing.

C.1.2 Terrain Model

C.1.2.1 Methods for Static Region Assignments

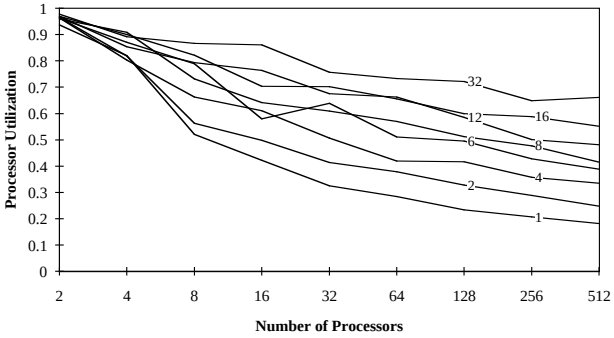


Figure C.5a Simulated processor utilization for the Terrain model using modular static load-balancing.

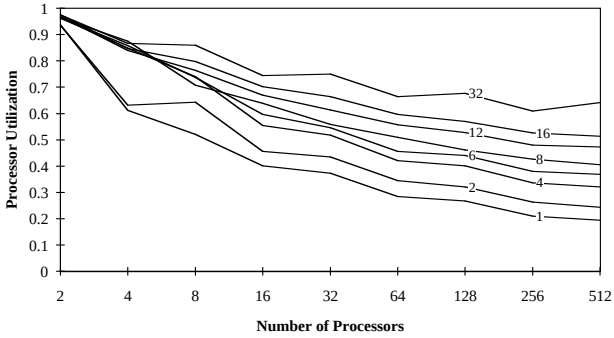


Figure C.5b Simulated processor utilization for the Terrain model using interleaved static load-balancing.

C.1.2.2 Methods for Once-Per-Frame Region Assignments

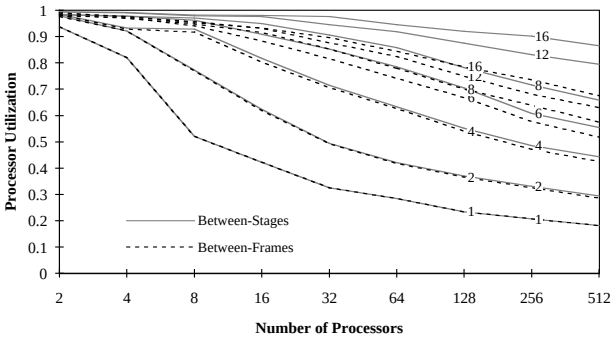


Figure C.6a Simulated processor utilization for the Terrain model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

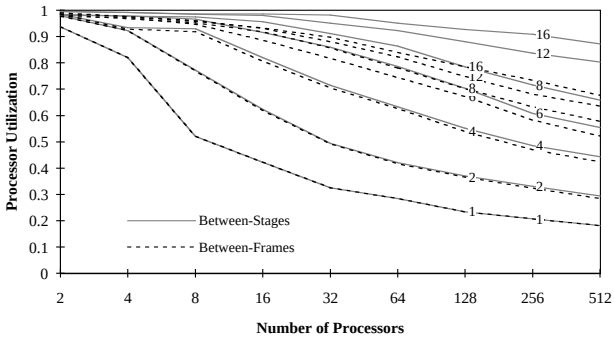


Figure C.6b Simulated processor utilization for the Terrain model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates.

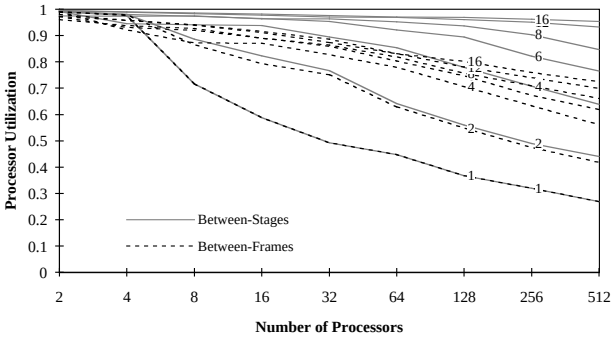


Figure C.6c Simulated processor utilization for the Terrain model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

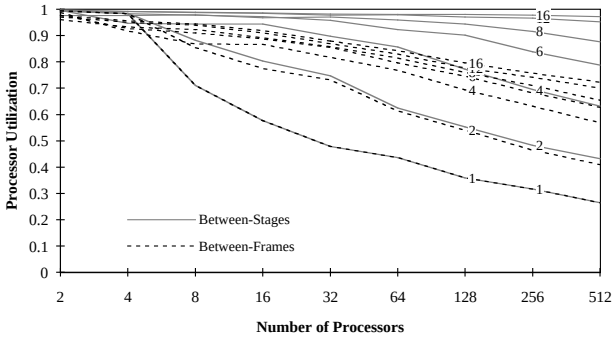


Figure C.6d Simulated processor utilization for the Terrain model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

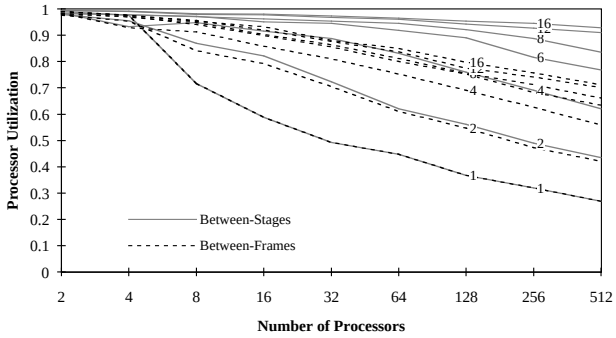


Figure C.6e Simulated processor utilization for the Terrain model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

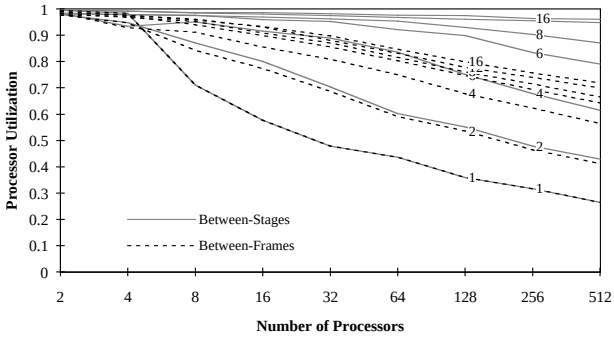


Figure C.6f Simulated processor utilization for the Terrain model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.1.2.3 Methods for Work-Queue Region Assignments

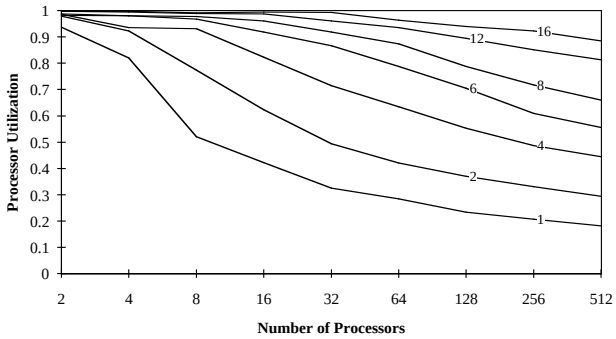


Figure C.7a Simulated processor utilization for the Terrain model using work-queue-assigned fixed region assigned load-balancing.

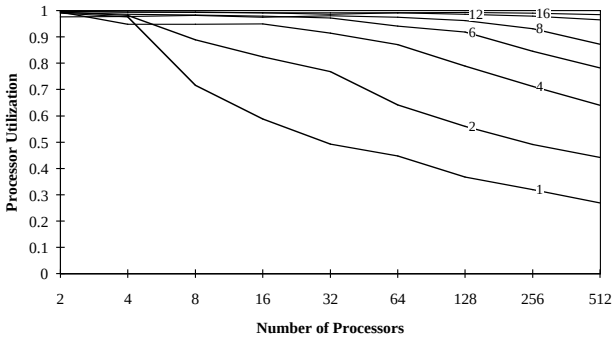


Figure C.7b Simulated processor utilization for the Terrain model using work-queue-assigned per-frame adaptive-region load-balancing.

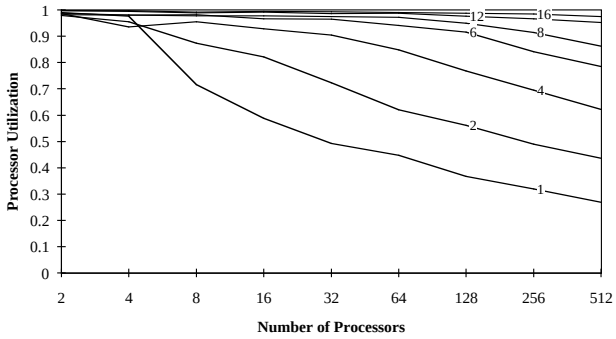


Figure C.7c Simulated processor utilization for the Terrain model using work-queue-assigned when-needed adaptive-region load-balancing.

C.1.2.4 Overlap Factor Values

There are fewer overlap factor charts than processor utilization charts because many load-balancing methods share region boundaries. Section C.1.1.4 gives the details. Table C.2 summarizes the mapping from load-balancing method to overlap chart.

Region Assignment Style	Load-balancing Method	Cost Estimate Function	Overlap Figure
Static	Modular	—	C.8a
Static	Interleaved	—	C.8b
Once-Per-Frame	Fixed Region	Triangle Counts	C.8a
Once-Per-Frame	Fixed Region	Size-Based	C.8a
Once-Per-Frame	Per-Frame Adaptive-Region	Triangle Counts	C.8c
Once-Per-Frame	Per-Frame Adaptive-Region	Size-Based	C.8d
Once-Per-Frame	When-Needed Adaptive-Region	Triangle Counts	C.8e
Once-Per-Frame	When-Needed Adaptive-Region	Size-Based	C.8f
Work-Queue	Fixed Region	—	C.8a
Work-Queue	Per-Frame Adaptive-Region	—	C.8g
Work-Queue	When-Needed Adaptive-Region	—	C.8h

Table C.2 Overlap figures mapping table for the Terrain model.

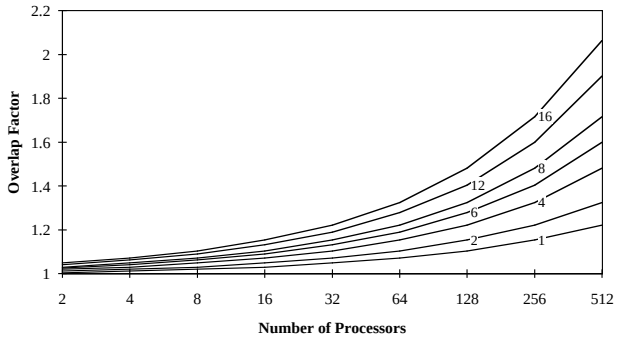


Figure C.8a Simulated overlap factor values for the Terrain model for load-balancing methods using fixed region boundaries (see table C.2).

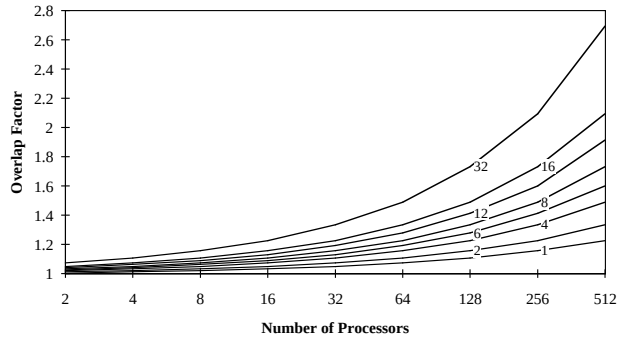


Figure C.8b Simulated overlap factor values for the Terrain model for static interleaved load-balancing.

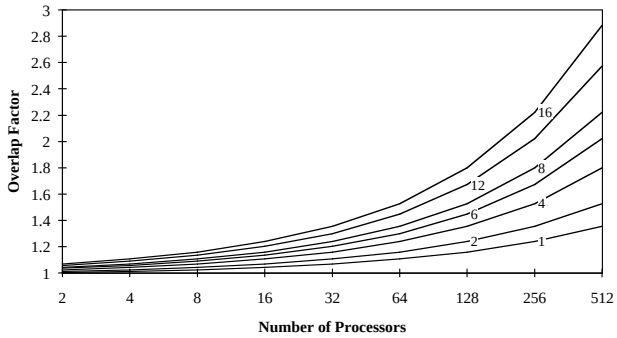


Figure C.8c Simulated overlap factor values for the Terrain model for once-per-frame-assigned per-frame adaptive-region load-balancing using triangle-count cost estimates.

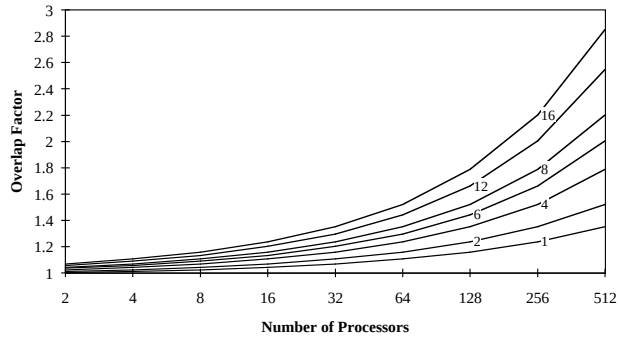


Figure C.8d Simulated overlap factor values for the Terrain model for once-per-frame-assigned per-frame adaptive-region load-balancing using size-based cost estimates.

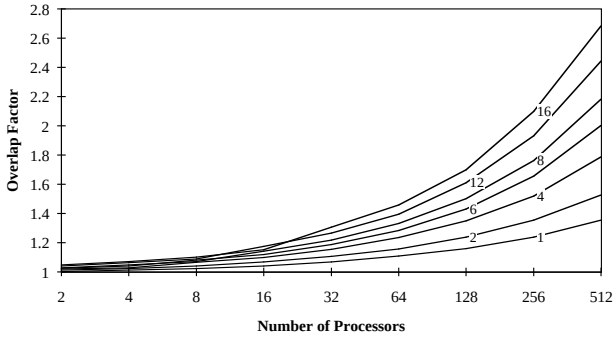


Figure C.8e Simulated overlap factor values for the Terrain model for once-per-frame-assigned when-needed adaptive-region load-balancing using triangle-count cost estimates.

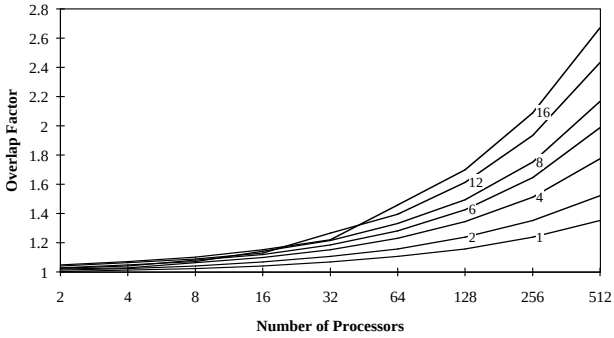


Figure C.8f Simulated overlap factor values for the Terrain model for once-per-frame-assigned when-needed adaptive-region load-balancing using size-based cost estimates.

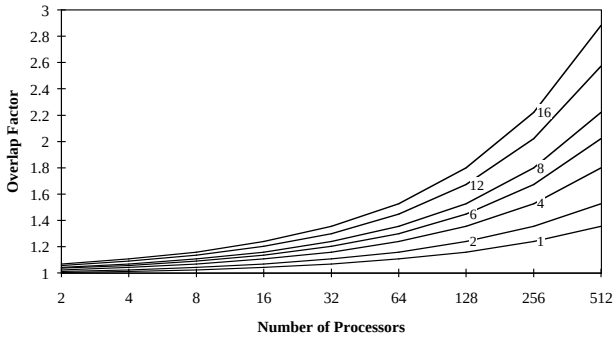


Figure C.8g Simulated overlap factor values for the Terrain model for work-queue-assigned per-frame adaptive-region load-balancing.

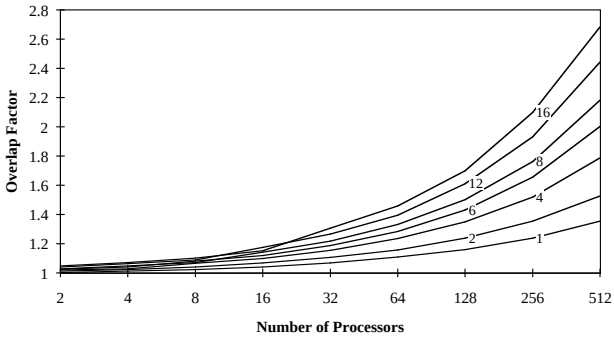


Figure C.8h Simulated overlap factor values for the Terrain model for work-queue-assigned when-needed adaptive-region load-balancing.

C.1.3 CT Head Model

C.1.3.1 Methods for Static Region Assignments

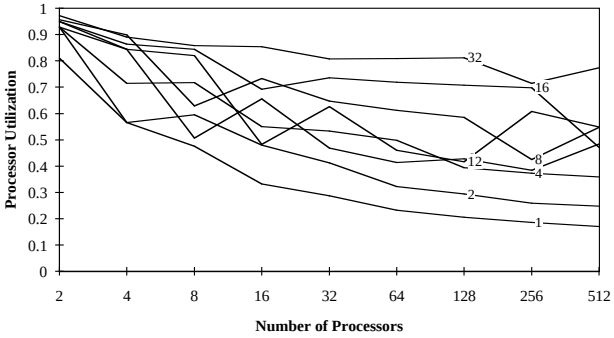


Figure C.9a Simulated processor utilization for the CT Head model using modular static load-balancing.

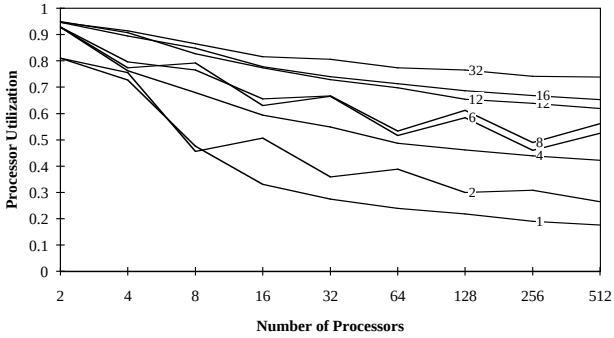


Figure C.9b Simulated processor utilization for the CT Head model using interleaved static load-balancing.

C.1.3.2 Methods for Once-Per-Frame Region Assignments

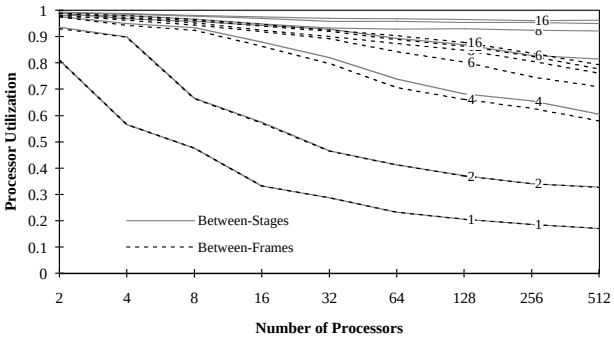


Figure C.10a Simulated processor utilization for the CT Head model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

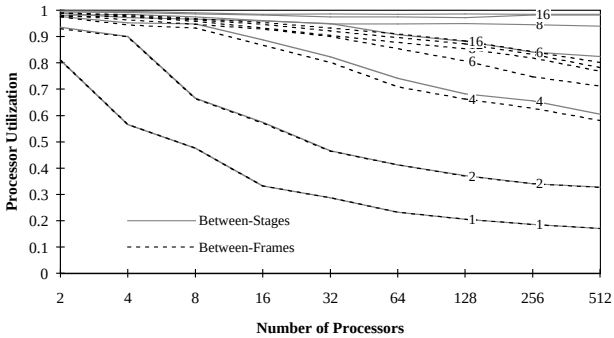


Figure C.10b Simulated processor utilization for the CT Head model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates.

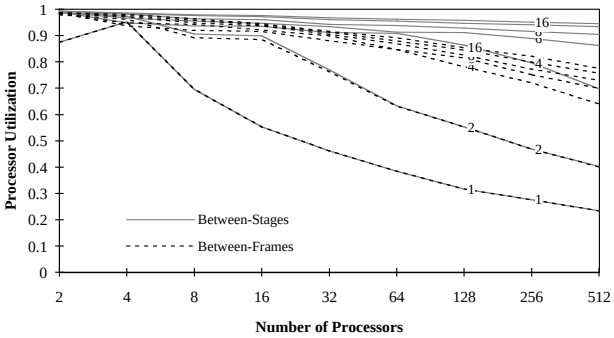


Figure C.10c Simulated processor utilization for the CT Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

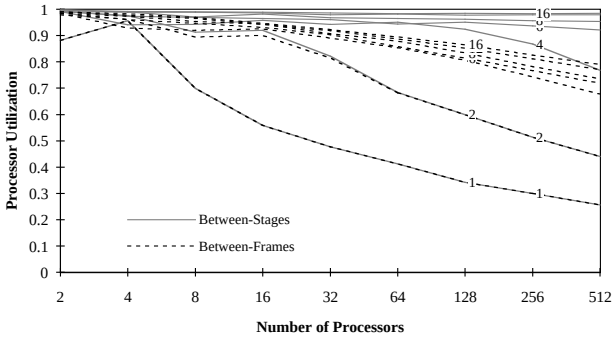


Figure C.10d Simulated processor utilization for the CT Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

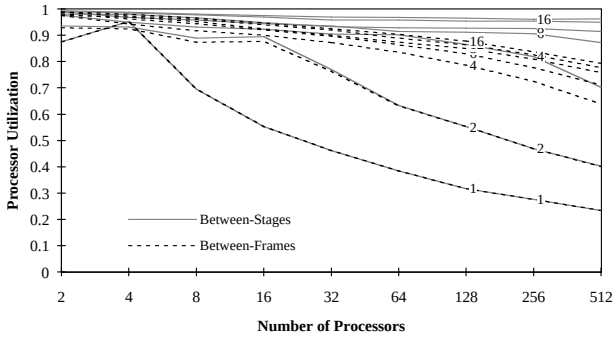


Figure C.10e Simulated processor utilization for the CT Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

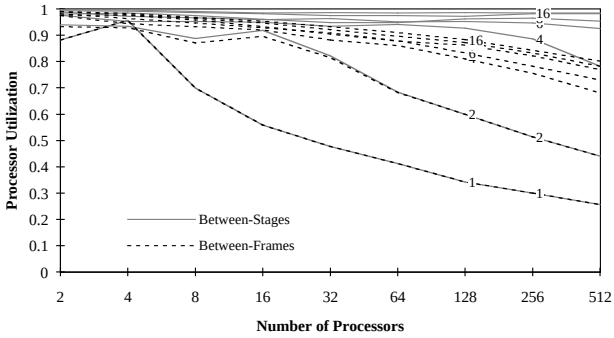


Figure C.10f Simulated processor utilization for the CT Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.1.3.3 Methods for Work-Queue Region Assignments

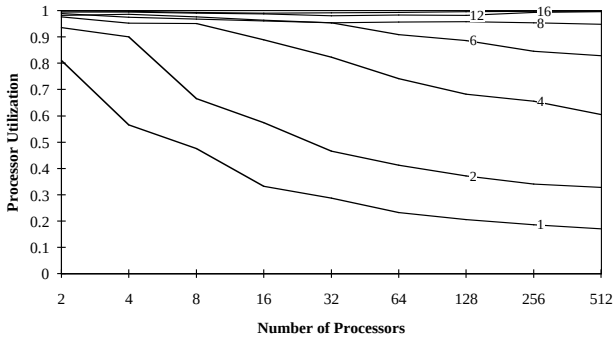


Figure C.11a Simulated processor utilization for the CT Head model using work-queue-assigned fixed region load-balancing.

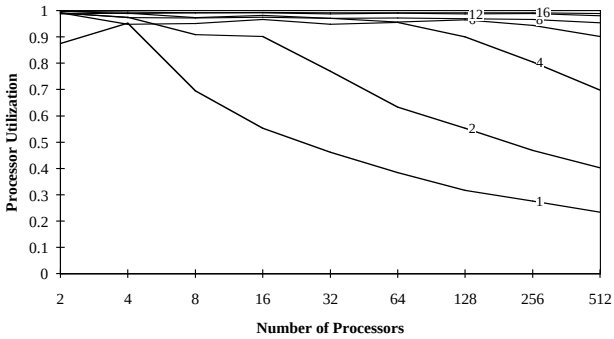


Figure C.11b Simulated processor utilization for the CT Head model using work-queue-assigned per-frame adaptive-region load-balancing.

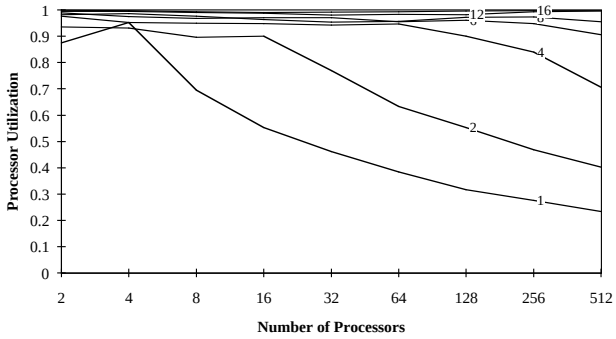


Figure C.11c Simulated processor utilization for the CT Head model using work-queue-assigned when-needed adaptive-region load-balancing.

C.1.3.4 Overlap Factor Values

There are fewer overlap factor charts than processor utilization charts because many load-balancing methods share region boundaries. Section C.1.1.4 gives the details. Table C.3 summarizes the mapping from load-balancing method to overlap chart.

Region Assignment Style	Load-balancing Method	Cost Estimate Function	Overlap Figure
Static	Modular	—	C.12a
Static	Interleaved	—	C.12b
Once-Per-Frame	Fixed Region	Triangle Counts	C.12a
Once-Per-Frame	Fixed Region	Size-Based	C.12a
Once-Per-Frame	Per-Frame Adaptive-Region	Triangle Counts	C.12c
Once-Per-Frame	Per-Frame Adaptive-Region	Size-Based	C.12d
Once-Per-Frame	When-Needed Adaptive-Region	Triangle Counts	C.12e
Once-Per-Frame	When-Needed Adaptive-Region	Size-Based	C.12f
Work-Queue	Fixed Region	—	C.12a
Work-Queue	Per-Frame Adaptive-Region	—	C.12g
Work-Queue	When-Needed Adaptive-Region	—	C.12h

Table C.3 Overlaps figure mapping table for the CT Head model.

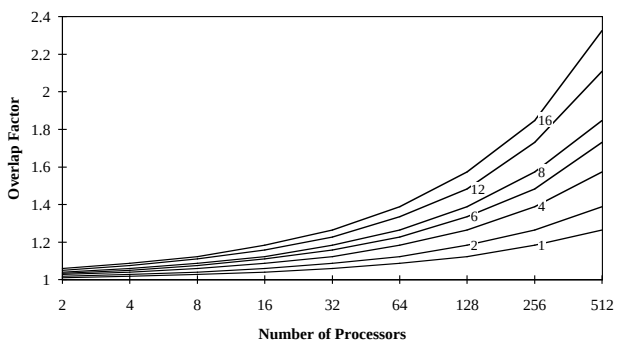


Figure C.12a Simulated overlap factor values for the CT Head model for load-balancing methods using fixed region boundaries (see table C.3).

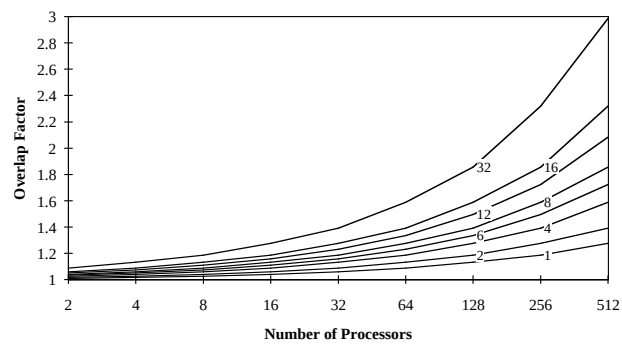


Figure C.12b Simulated overlap factor values for the CT Head model for static interleaved load-balancing.

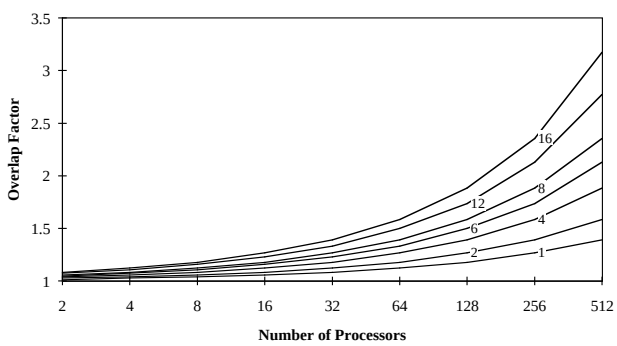


Figure C.12c Simulated overlap factor values for the CT Head model for once-per-frame-assigned per-frame adaptive-region load-balancing using triangle-count cost estimates.

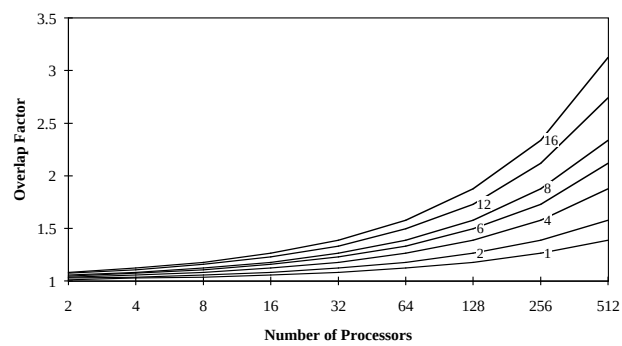


Figure C.12d Simulated overlap factor values for the CT Head model for once-per-frame-assigned per-frame adaptive-region load-balancing using size-based cost estimates.

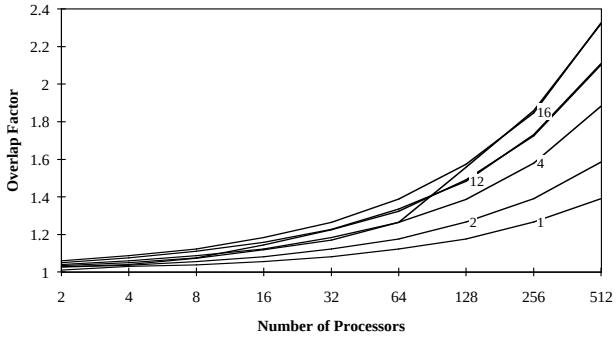


Figure C.12e Simulated overlap factor values for the CT Head model for once-per-frame-assigned when-needed adaptive-region load-balancing using triangle-count cost estimates.

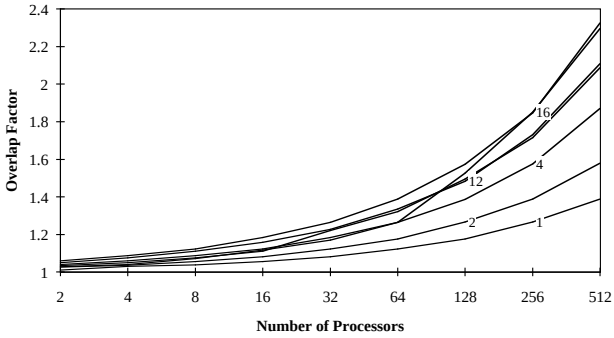


Figure C.12f Simulated overlap factor values for the CT Head model for once-per-frame-assigned when-needed adaptive-region load-balancing using size-based cost estimates.

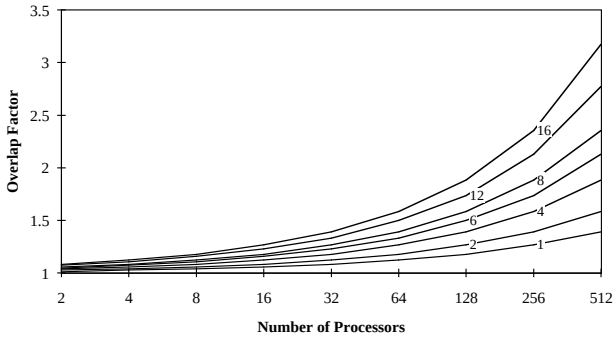


Figure C.12g Simulated overlap factor values for the CT Head model for work-queue-assigned per-frame adaptive-region load-balancing.

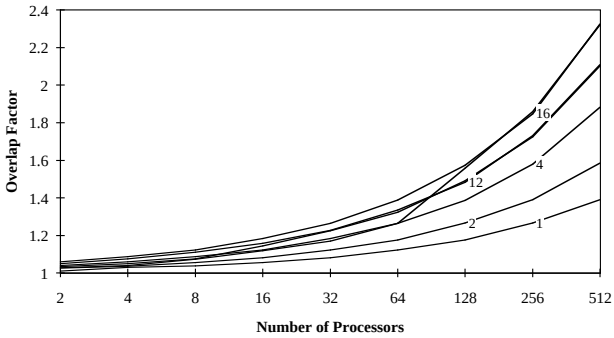


Figure C.12h Simulated overlap factor values for the CT Head model for work-queue-assigned when-needed adaptive-region load-balancing.

C.1.4 Polio Model

C.1.4.1 Methods for Static Region Assignments

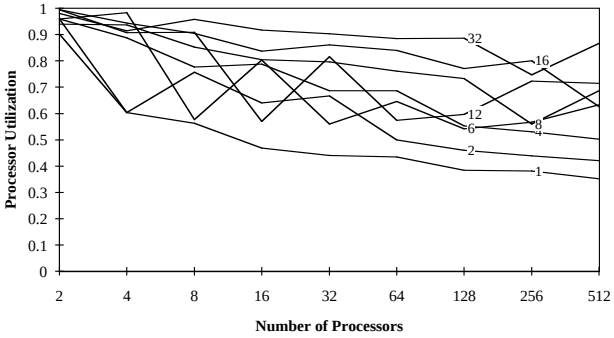


Figure C.13a Simulated processor utilization for the Polio model using modular static load-balancing.

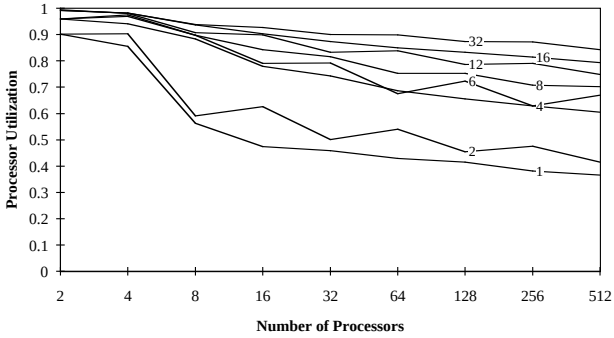


Figure C.13b Simulated processor utilization for the Polio model using interleaved static load-balancing.

C.1.4.2 Methods for Once-Per-Frame Region Assignments

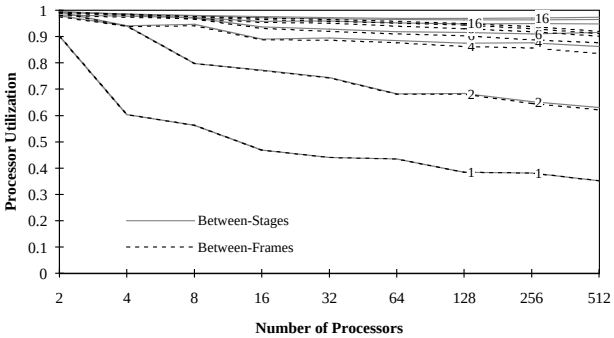


Figure C.14a Simulated processor utilization for the Polio model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

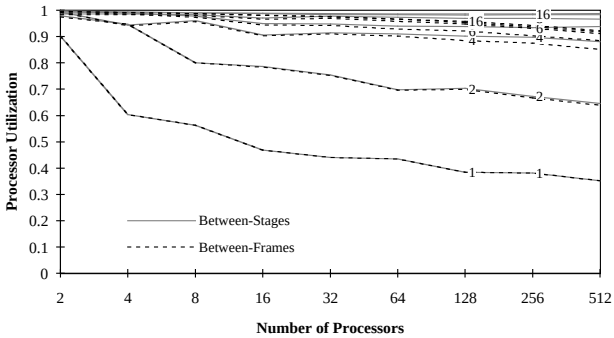


Figure C.14b Simulated processor utilization for the Polio model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates.

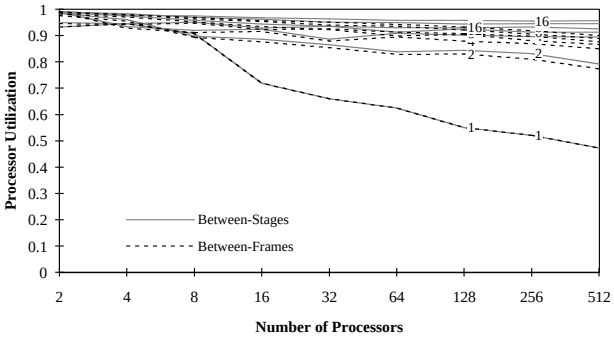


Figure C.14c Simulated processor utilization for the Polio model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

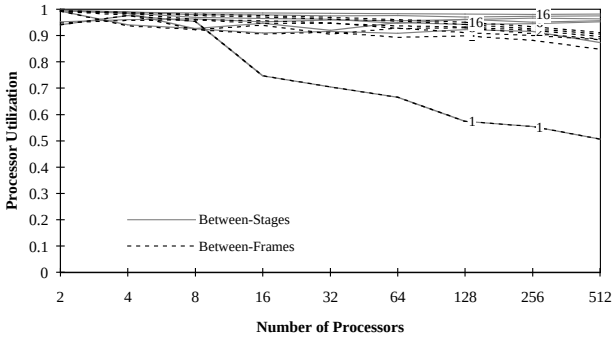


Figure C.14d Simulated processor utilization for the Polio model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

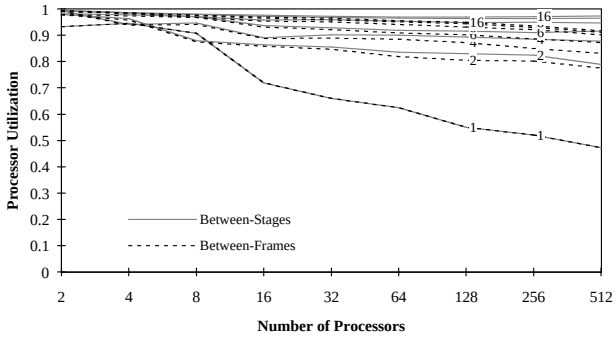


Figure C.14e Simulated processor utilization for the Polio model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

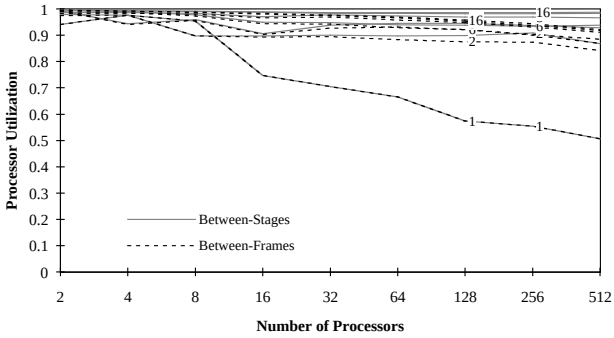


Figure C.14f Simulated processor utilization for the Polio model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.1.4.3 Methods for Work-Queue Region Assignments

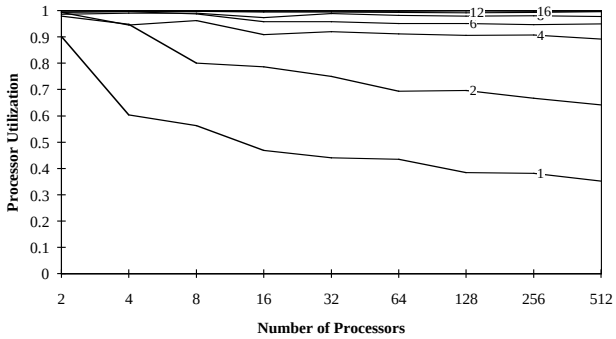


Figure C.15a Simulated processor utilization for the Polio model using work-queue-assigned fixed region load-balancing.

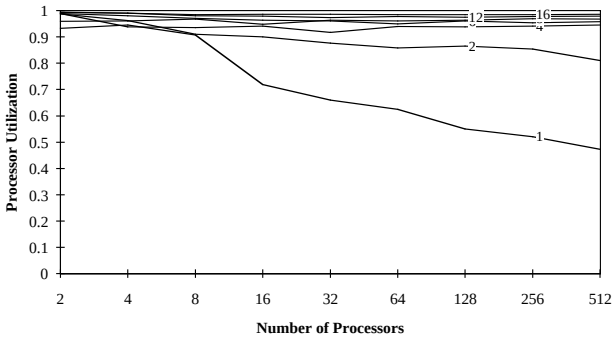


Figure C.15b Simulated processor utilization for the Polio model using work-queue-assigned per-frame adaptive-region load-balancing.

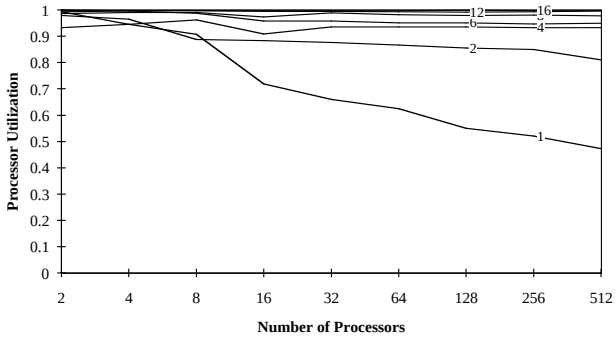


Figure C.15c Simulated processor utilization for the Polio model using work-queue-assigned when-needed adaptive-region load-balancing.

C.1.4.4 Overlap Factor Values

There are fewer overlap factor charts than processor utilization charts because many load-balancing methods share region boundaries. Section C.1.1.4 gives the details. Table C.4 summarizes the mapping from load-balancing method to overlap chart.

Region Assignment Style	Load-balancing Method	Cost Estimate Function	Overlap Figure
Static	Modular	—	C.16a
Static	Interleaved	—	C.16b
Once-Per-Frame	Fixed Region	Triangle Counts	C.16a
Once-Per-Frame	Fixed Region	Size-Based	C.16a
Once-Per-Frame	Per-Frame Adaptive-Region	Triangle Counts	C.16c
Once-Per-Frame	Per-Frame Adaptive-Region	Size-Based	C.16d
Once-Per-Frame	When-Needed Adaptive-Region	Triangle Counts	C.16e
Once-Per-Frame	When-Needed Adaptive-Region	Size-Based	C.16f
Work-Queue	Fixed Region	—	C.16a
Work-Queue	Per-Frame Adaptive-Region	—	C.16g
Work-Queue	When-Needed Adaptive-Region	—	C.16h

Table C.4 Overlap figures mapping table for the Polio model.

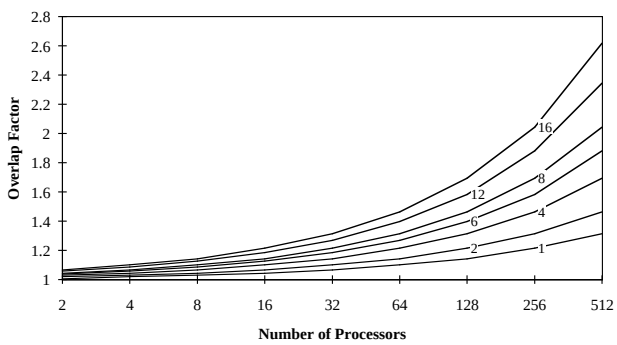


Figure C.16a Simulated overlap factor values for the Polio model for load-balancing methods using fixed region boundaries (see table C.4).

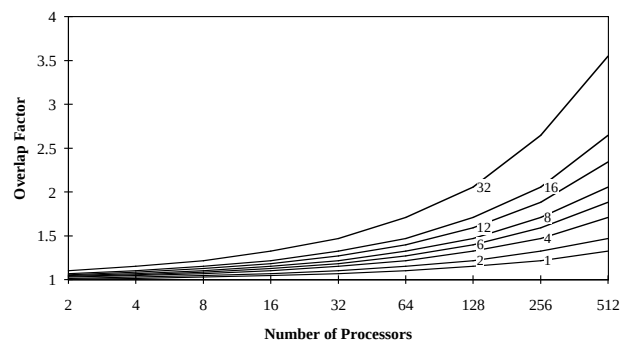


Figure C.16b Simulated overlap factor values for the Polio model for static interleaved load-balancing.

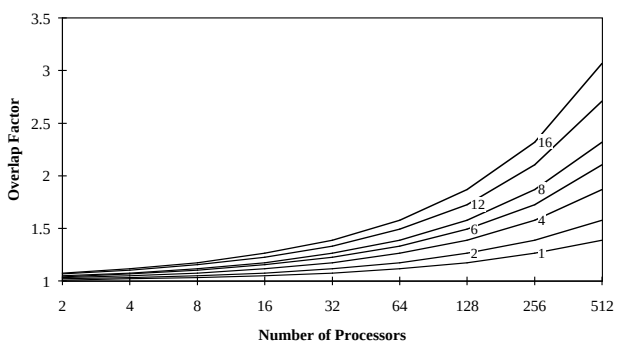


Figure C.16c Simulated overlap factor values for the Polio model for once-per-frame-assigned per-frame adaptive-region load-balancing using triangle-count cost estimates.

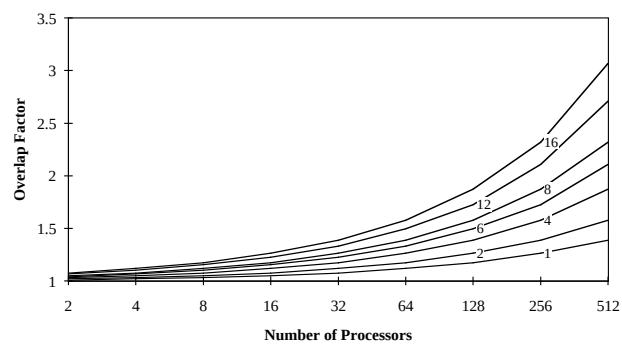


Figure C.16d Simulated overlap factor values for the Polio model for once-per-frame-assigned per-frame adaptive-region load-balancing using size-based cost estimates.

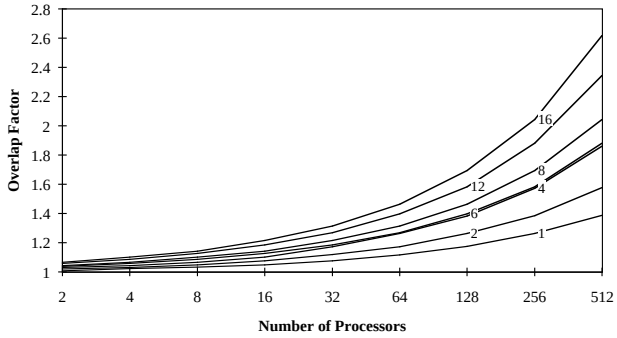


Figure C.16e Simulated overlap factor values for the Polio model for once-per-frame-assigned when-needed adaptive-region load-balancing using triangle-count cost estimates.

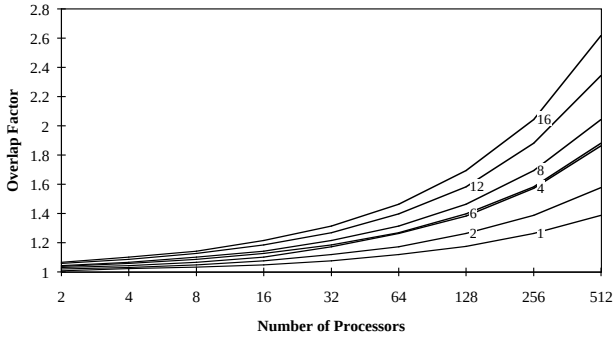


Figure C.16f Simulated overlap factor values for the Polio model for once-per-frame-assigned when-needed adaptive-region load-balancing using size-based cost estimates.

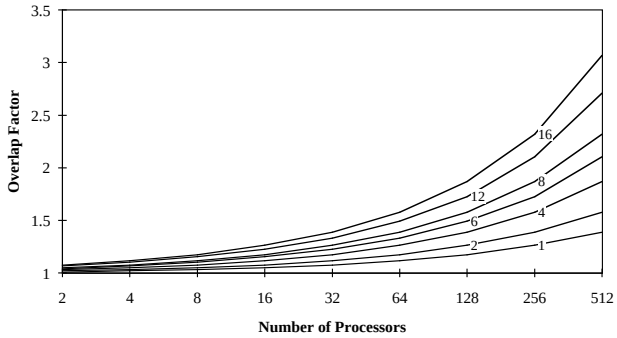


Figure C.16g Simulated overlap factor values for the Polio model for work-queue-assigned per-frame adaptive-region load-balancing.

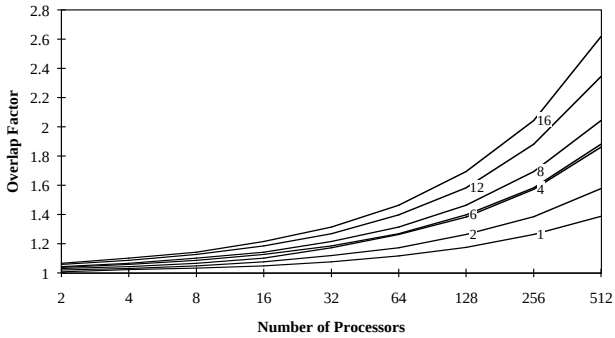


Figure C.16h Simulated overlap factor values for the Polio model for work-queue-assigned when-needed adaptive-region load-balancing.

C.2 Performance Model Results

This section gives the expected time of the different load-balancing methods as predicted by the performance model. The times plotted in the charts have been normalized by dividing each data point by the average time of all the data points with the same model and number of processors. The work-queue load-balancing methods were normalized separately from the others because they had very different performance characteristics. The charts have the same formats as the charts in the previous section. More details appear in section 6.3.2.

C.2.1 PLB Head Model

C.2.1.1 Methods for Static Region Assignments

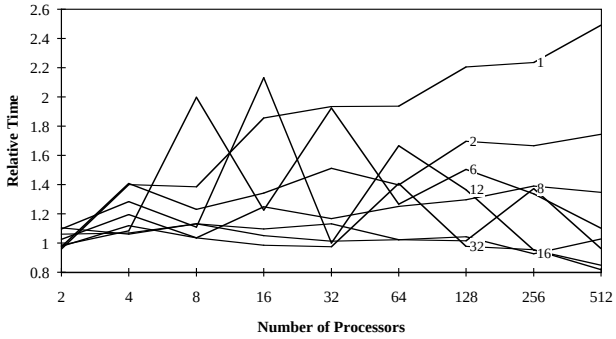


Figure C.17a Expected relative time for the PLB Head model using modular static load-balancing.

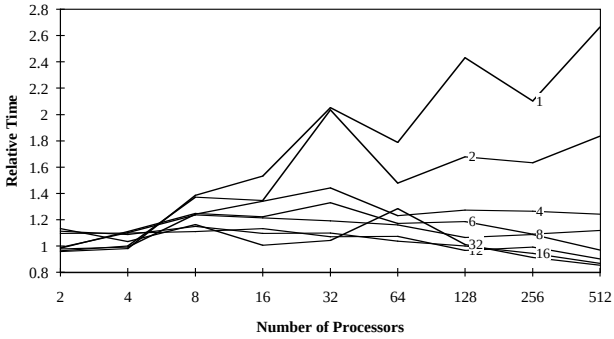


Figure C.17b Expected relative time for the PLB Head model using interleaved static load-balancing.

C.2.1.2 Methods for Once-Per-Frame Region Assignments

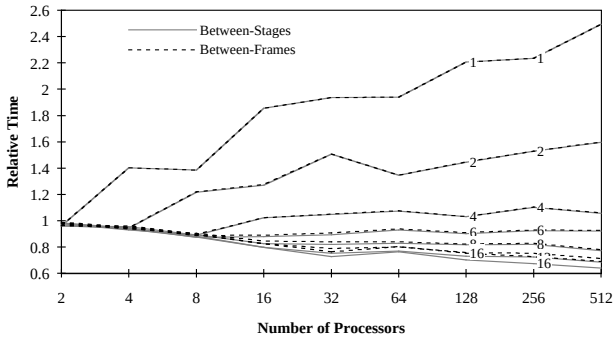


Figure C.18a Expected relative time for the PLB Head model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

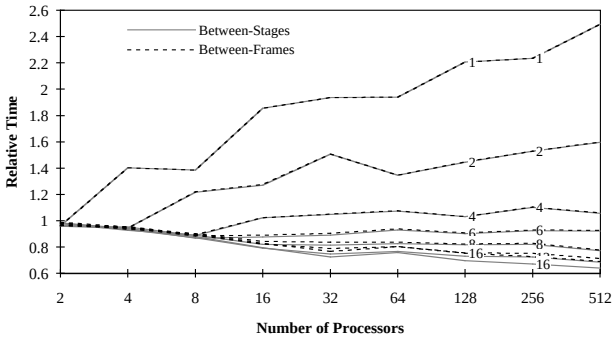


Figure C.18b Expected relative time for the PLB Head model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates.

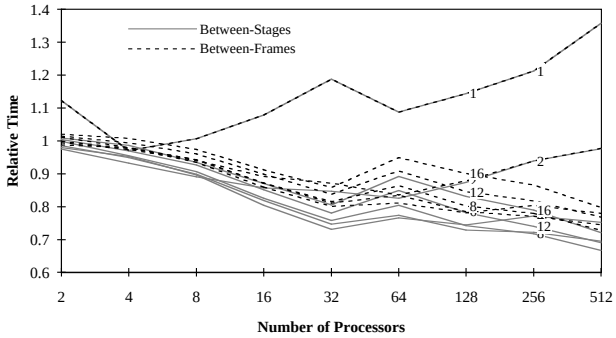


Figure C.18c Expected relative time for the PLB Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

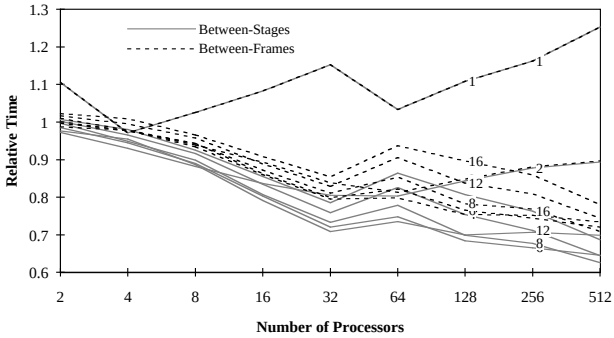


Figure C.18d Expected relative time for the PLB Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

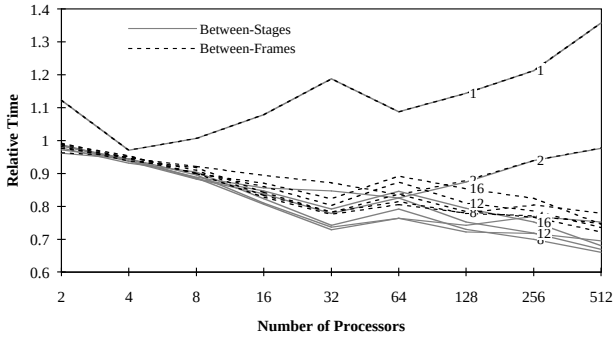


Figure C.18e Expected relative time for the PLB Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

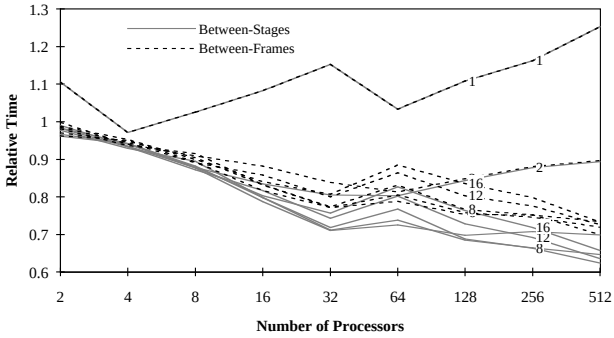


Figure C.18f Expected relative time for the PLB Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.2.1.3 Methods for Work-Queue Region Assignments

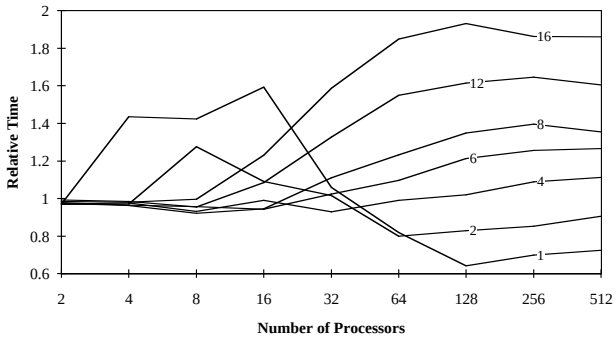


Figure C.19a Expected relative time for the PLB Head model using work-queue assigned fixed-region load-balancing.

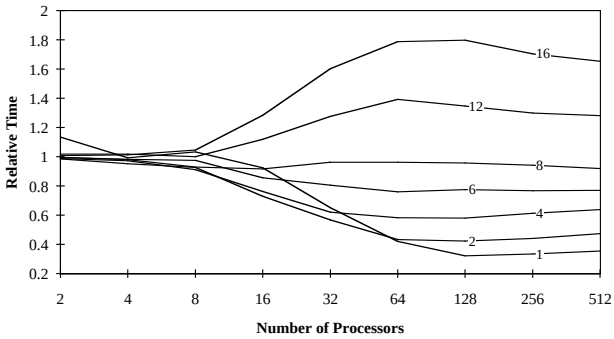


Figure C.19b Expected relative time for the PLB Head model using work-queue assigned per-frame adaptive-region load-balancing.

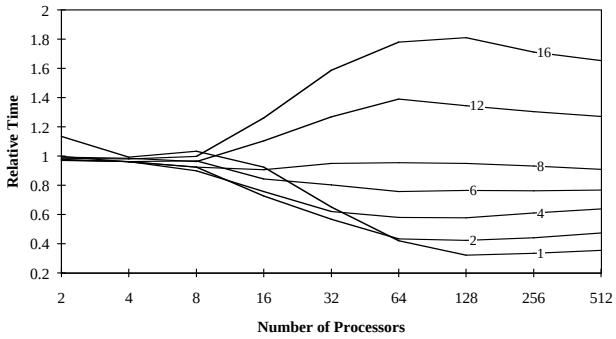


Figure C.19c Expected relative time for the PLB Head model using work-queue assigned when-needed adaptive-region load-balancing.

C.2.2 Terrain Model

C.2.2.1 Methods for Static Region Assignments

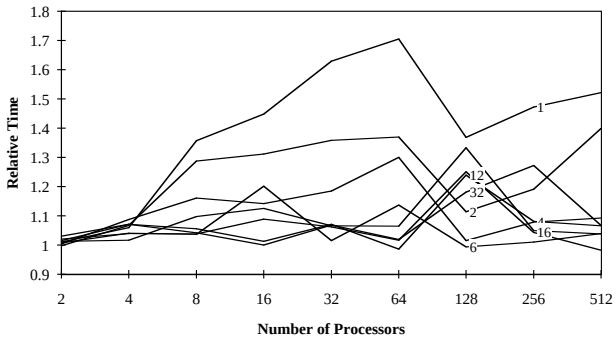


Figure C.20a Expected relative time for the Terrain model using modular static load-balancing.

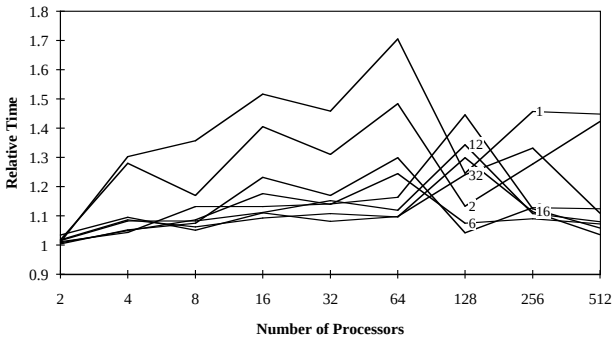


Figure C.20b Expected relative time for the Terrain model using interleaved static load-balancing.

C.2.2.2 Methods for Once-Per-Frame Region Assignments

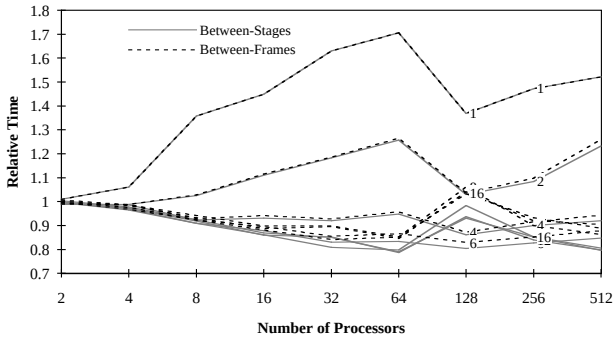


Figure C.21a Expected relative time for the Terrain model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

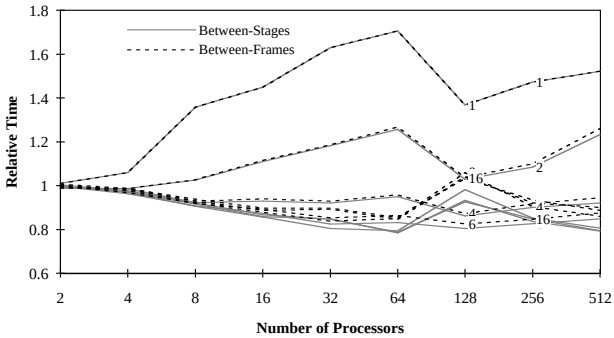


Figure C.21b Expected relative time for the Terrain model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates.

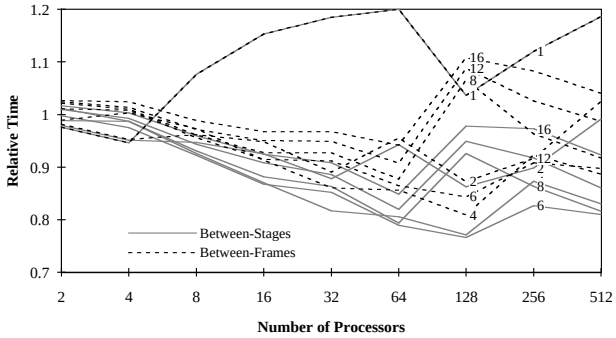


Figure C.21c Expected relative time for the Terrain model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

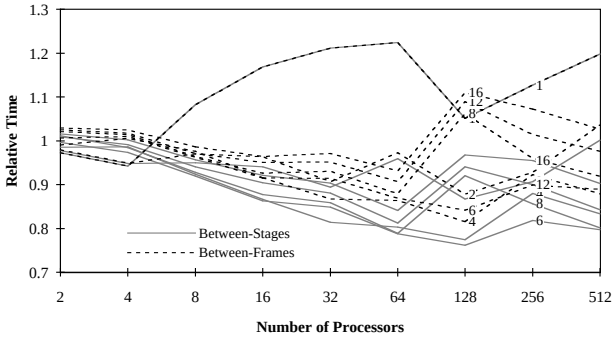


Figure C.21d Expected relative time for the Terrain model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

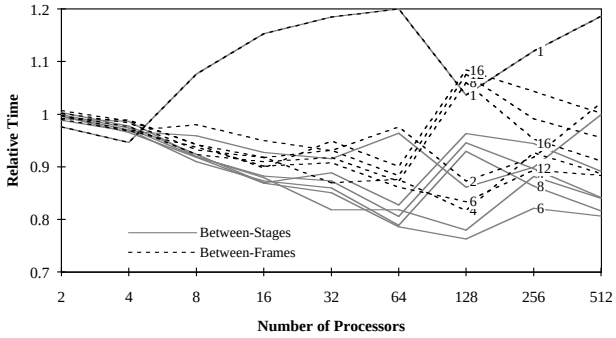


Figure C.21e Expected relative time for the Terrain model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

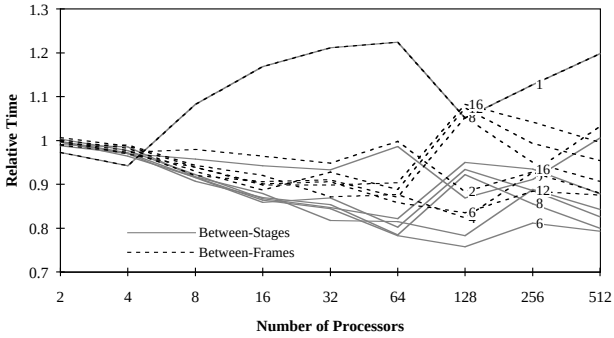


Figure C.21f Expected relative time for the Terrain model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.2.2.3 Methods for Work-Queue Region Assignments

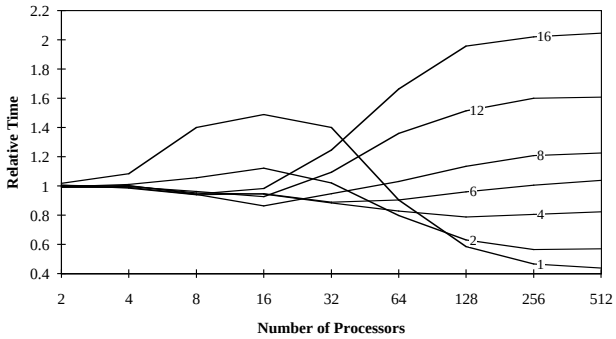


Figure C.22a Expected relative time for the Terrain model using work-queue assigned fixed-region load-balancing.

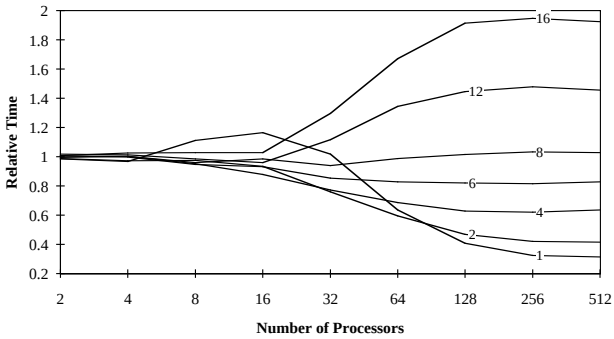


Figure C.22b Expected relative time for the Terrain model using work-queue assigned per-frame adaptive-region load-balancing.

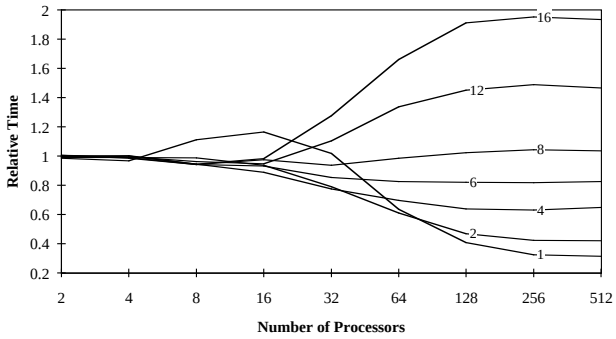


Figure C.22c Expected relative time for the Terrain model using work-queue assigned when-needed adaptive-region load-balancing.

C.2.3 CT Head Model

C.2.3.1 Methods for Static Region Assignments

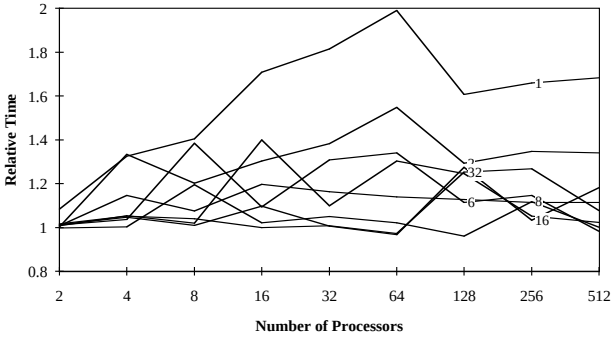


Figure C.23a Expected relative time for the CT Head model using modular static load-balancing.

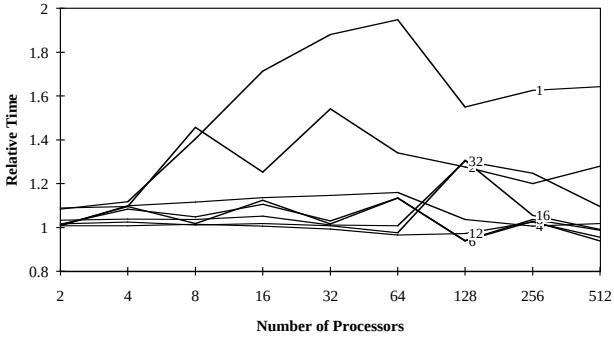


Figure C.23b Expected relative time for the CT Head model using interleaved static load-balancing.

C.2.3.2 Methods for Once-Per-Frame Region Assignments

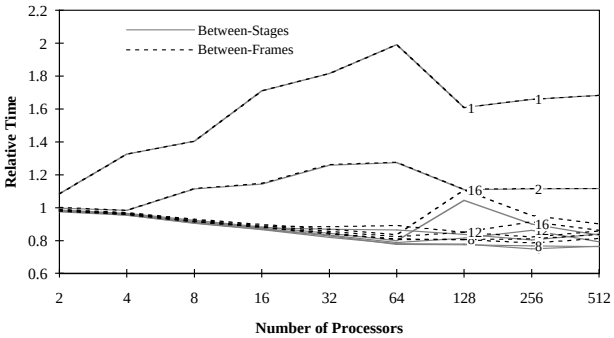


Figure C.24a Expected relative time for the CT Head model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

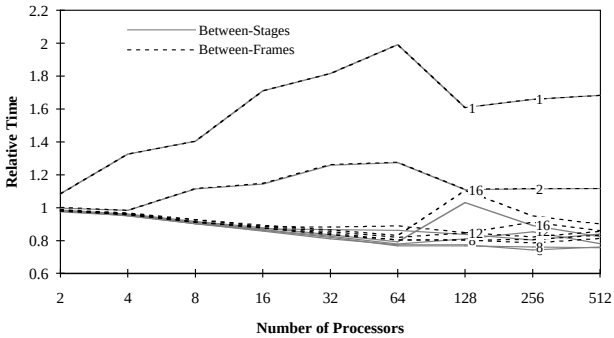


Figure C.24b Expected relative time for the CT Head model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates.

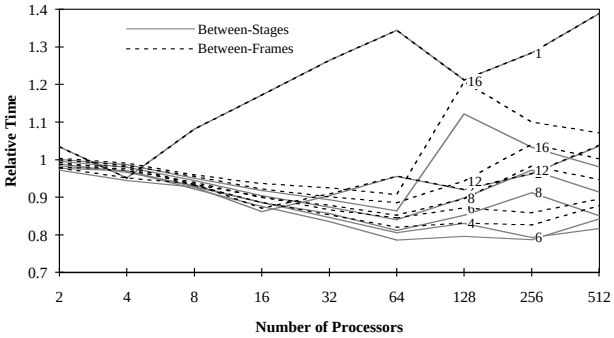


Figure C.24c Expected relative time for the CT Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

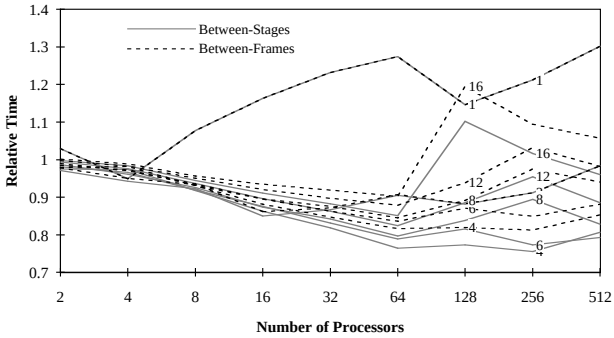


Figure C.24d Expected relative time for the CT Head model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

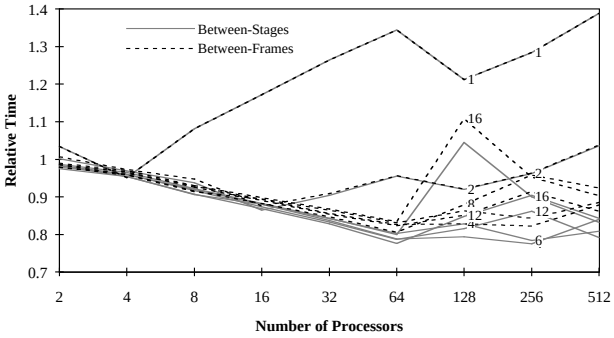


Figure C.24e Expected relative time for the CT Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

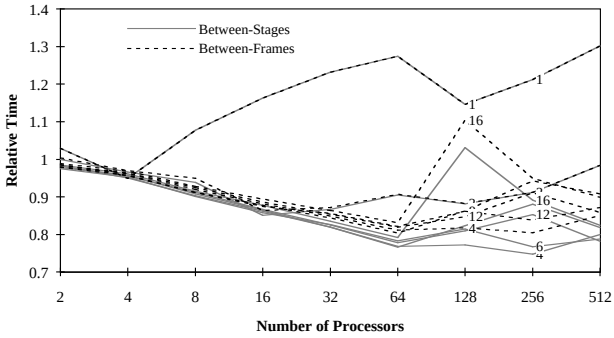


Figure C.24f Expected relative time for the CT Head model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.2.3.3 Methods for Work-Queue Region Assignments

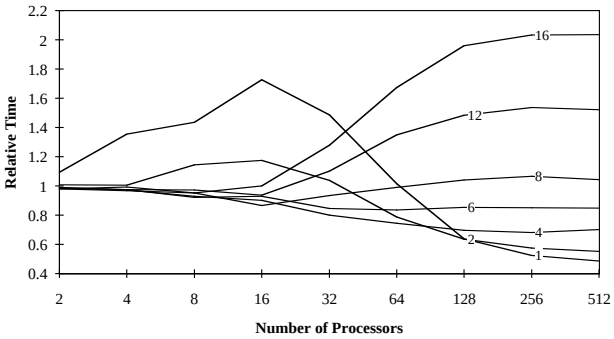


Figure C.25a Expected relative time for the CT Head model using work-queue assigned fixed-region load-balancing.

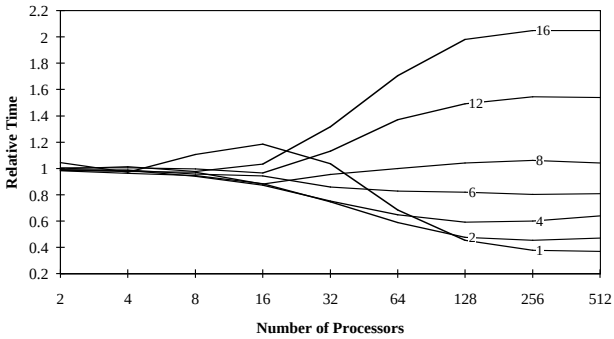


Figure C.25b Expected relative time for the CT Head model using work-queue assigned per-frame adaptive-region load-balancing.

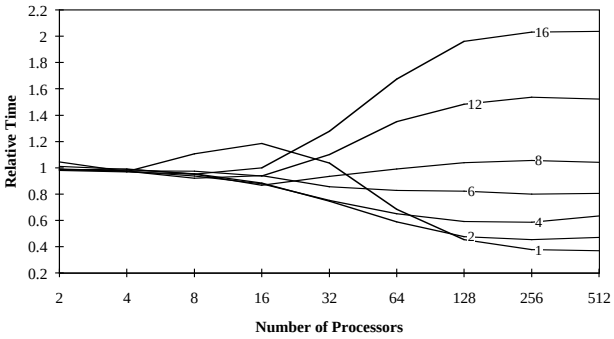


Figure C.25c Expected relative time for the CT Head model using work-queue assigned when-needed adaptive-region load-balancing.

C.2.4 Polio Model

C.2.4.1 Methods for Static Region Assignments

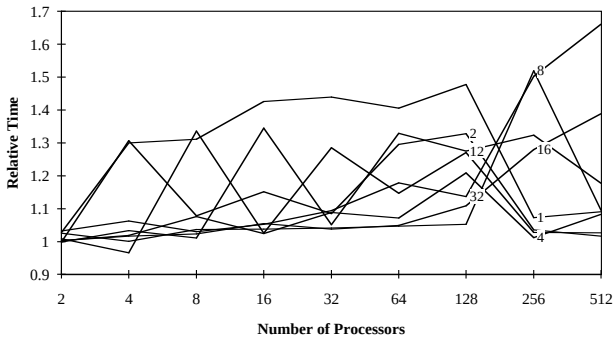


Figure C.26a Expected relative time for the Polio model using modular static load-balancing.

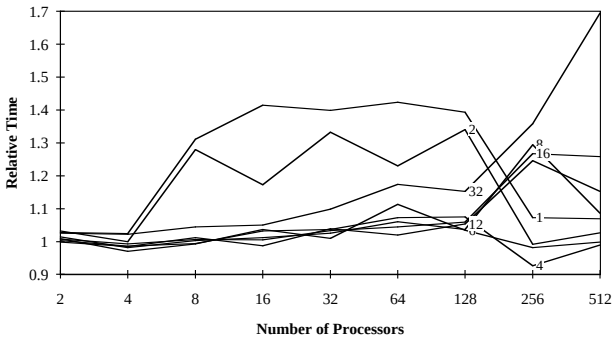


Figure C.26b Expected relative time for the Polio model using interleaved static load-balancing.

C.2.4.2 Methods for Once-Per-Frame Region Assignments

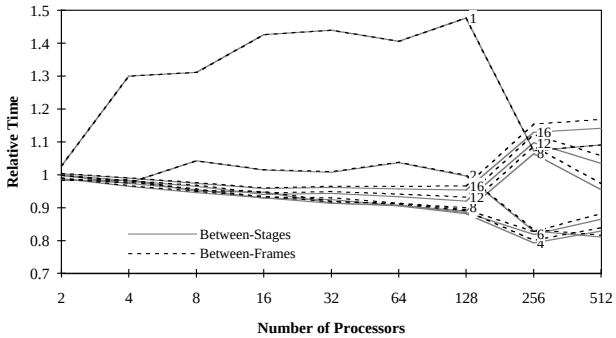


Figure C.27a Expected relative time for the Polio model using between-stages and between-frames assigned fixed-region load-balancing and triangle-count cost estimates.

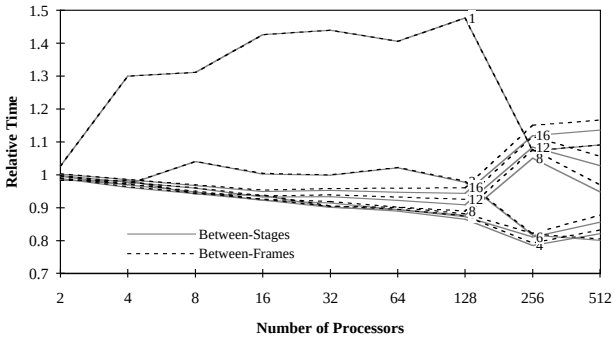


Figure C.27b Expected relative time for the Polio model using between-stages and between-frames assigned fixed-region load-balancing and size-based cost estimates.

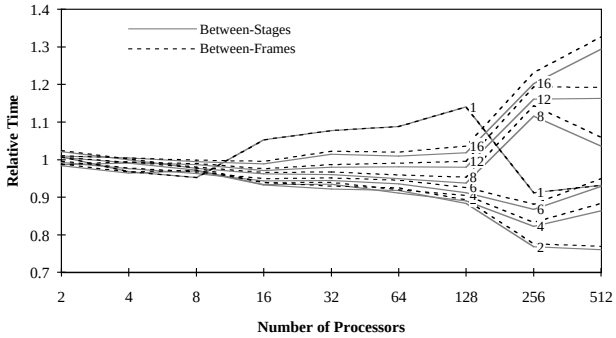


Figure C.27c Expected relative time for the Polio model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and triangle-count cost estimates.

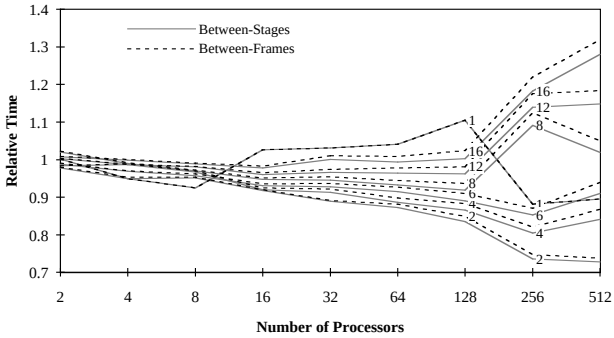


Figure C.27d Expected relative time for the Polio model using between-stages and between-frames assigned per-frame adaptive-region load-balancing and size-based cost estimates.

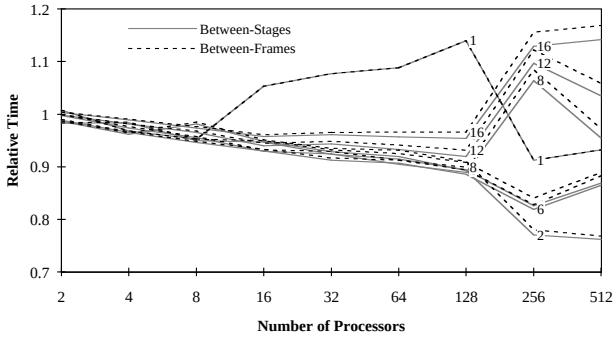


Figure C.27e Expected relative time for the Polio model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and triangle-count cost estimates.

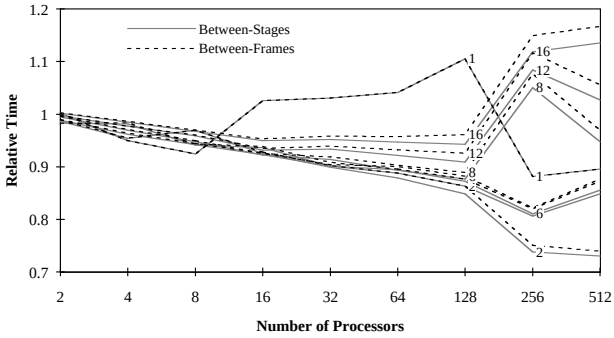


Figure C.27f Expected relative time for the Polio model using between-stages and between-frames assigned when-needed adaptive-region load-balancing and size-based cost estimates.

C.2.4.3 Methods for Work-Queue Region Assignments

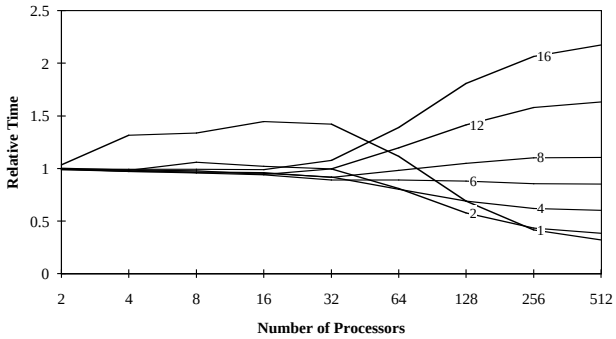


Figure C.28a Expected relative time for the Polio model using work-queue assigned fixed-region load-balancing.

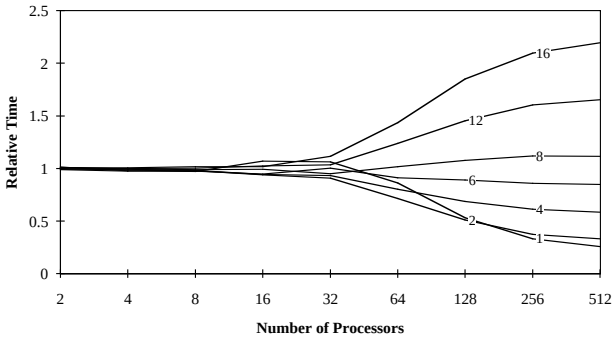


Figure C.28b Expected relative time for the Polio model using work-queue assigned per-frame adaptive-region load-balancing.

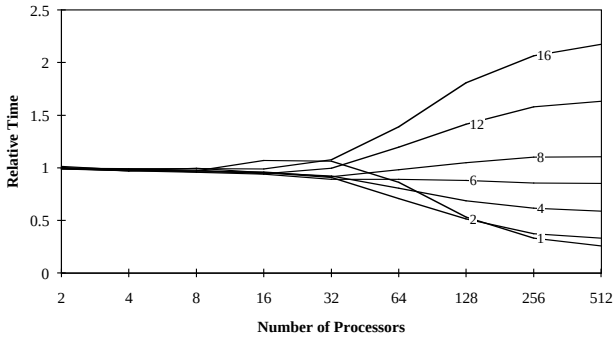


Figure C.28c Expected relative time for the Polio model using work-queue assigned when-needed adaptive-region load-balancing.

APPENDIX D

PIXEL-PLANES 5 BOARD COSTS

This appendix gives the details of how the costs of the Pixel-Planes 5 boards were calculated. The costs are used in Chapter 8 to compare the price-performance ratios of specialized processors and general-purpose processors rendering polygons.

Table D.1 gives the cost breakdown of the renderer board, using prices from the first half of 1992. The costs shown are the costs of the parts; the retail cost would be much higher. The prices of the custom chips (the first three items in the table) are hard to estimate, since the cost depends on the number of parts made in each fabrication order. The prices given here would be for a small production run. The cost of the network interface, $2500 ($1800 parts, $500 board, and $200 assembly, estimated by John Thomas [Thom93]), must be added, for a total cost of $6823 per board.

Table D.2 gives the cost of an i860 processor (a GP). Two processors will fit on each board, so the board and assembly costs are amortized over two processors. Adding half of the cost of a network interface board brings the cost per processor to $3266.

Part	Quantity	Price Each	Price
Image Generation Controller	1	$150.00	$150.00
Enhanced Memory Chip	64	25.00	1600.00
Corner Turner	4	50.00	200.00
1 Mbit VRAM	64	10.00	640.00
PAL	23	6.00	138.00
75C03A FIFO	8	18.00	144.00
Clock Driver	6	6.00	36.00
Clock Generator	2	30.00	60.00
MSI/LSI TTL	82	2.50	205.00
Passive parts	1	300.00	300.00
Circuit board	1	500.00	500.00
Board Assembly	1	350.00	350.00
		Total	$4323.00

Table D.1 Cost of a Pixel-Planes 5 renderer board. The total does not include the cost of the network interface board.

Part	Quantity	Price Each	Price
Intel i860XR 40 MHz	1	$500.00	$500.00
1 Mbit DRAM	64	5.00	320.00
72225 FIFO	8	50.00	400.00
EPROM	1	10.00	10.00
PAL	17	6.00	102.00
Clock Driver	2	6.00	12.00
Clock Generator	1	30.00	30.00
MSI/LSI TTL	27	2.50	67.50
Passive parts	1	150.00	150.00
Circuit board	0.5	500.00	250.00
Board Assembly	0.5	350.00	175.00
		Total	$2016.50

Table D.2 Cost of a Pixel-Planes 5 Intel i860 processor (one half of a board). The total does not include the cost of the network interface board.

APPENDIX E

GLOSSARY

This appendix has a glossary of the new and nonstandard words and phrases used in the dissertation. The number at the end of the definition gives the section where the word or phrase is defined in the text.

between-frames	region-assignment style that changes the region-processor assignments between frames (4.3.3).
between-stages	region-assignment style that assigns regions to processors between the geometry processing and rasterization stages of a frame (4.3.2).
bisection bandwidth	bandwidth across the center of a communication network, where one would cut to split the system into two equal portions (3.3.3).
bucketization	process of classifying triangles according to the regions they overlap and copying the triangles to a data structure, or bucket, associated with each region (2.3.1).
composite	method of combining two images with depth for each pixel so that the pixels closest to the viewer are retained. For each pixel, the depth value from each image is compared, and the pixel from the closer to the viewer is placed in the output image (2.4.2).
deferred shading	shading method that performs shading calculations after the per-triangle processing (2.1).
display triangle	transformed triangle, in screen coordinates (3.1).
fully parallel algorithm	parallel algorithm where the data in each step is divided among multiple processors (1.3).
geometry processing	portion of the graphics pipeline that performs geometric calculations. It consists of the transformation of triangles from modeling to eye space to screen space, clipping, and any per-vertex shading calculations (2.1).
granularity ratio	ratio between the number of processors and the number of regions (2.3.1).
immediate-mode	type of graphics library interface where the application retains the geometric data and gives the graphics library the geometric data each frame (1.4.3).
MEGA	Pixel-Planes 5 rendering algorithm that uses a fixed, interleaved assignment of renderers to processors (8.3).
one-step	redistribution method where processors send messages directly to the destination processor (4.4.2).
overlap factor	average number of regions that a triangle overlaps (3.2.2.1).
parallel efficiency	measured speedup of a parallel implementation has over a single-processor implementation divided by the number of processors used (1.2).

primitive structure	a structure distributed as if it were a single primitive (5.2.3).
rasterization	scan conversion, hidden surface elimination (*z*-buffering), and any per-pixel shading calculations (2.1).
raw triangle	untransformed triangle (3.1).
retained-mode	type of graphics library interface where the graphics library retains the geometric data. The application edits the data using procedure calls as necessary (1.4.3).
sort-first	a class of parallel rendering algorithms that redistributes raw triangles (3.1).
sort-last	a class of parallel rendering algorithms that redistributes pixels, samples, or triangle fragments (3.1).
sort-last-full	sort-last variant where every processor transmits a full screen image (3.1.3).
sort-last-sparse	sort-last variant where each processor only sends the pixels or samples that it rasterizes (3.1.3).
sort-middle	a class of parallel rendering algorithms that redistributes display triangles (3.1).
stage	a portion of the graphics pipeline; the stages considered here include the geometry processing (transformation, clipping, lighting), the rasterization, and the redistribution (2.1; also see 3.1).
static	region-assignment style that fixes the region-to-processor assignments during system initialization (4.3.4).
step	sometimes used as a synonym of stage.
stream-per-processor	type of sort-middle redistribution where processors send a stream of messages for each processor; can be one-step or two-step (4.4.1).
stream-per-region	type of sort-middle redistribution where processors send messages containing triangles for each region, implying that each processor sends a stream of messages for each region; can be one-step or two-step (4.4.1).
two-step	redistribution method where processors send messages to a forwarding processor so that each message is sent and received twice (4.4.2).
work-queue	region-assignment style that assigns regions to processors dynamically using a work-queue, making the assignments during the rasterization of a frame (4.3.1).

REFERENCES

[Akel88] Akeley, Kurt, and Tom Jermoluk, "High-Performance Polygon Rendering", *Computer Graphics*, Vol. 22, No. 4, August 1988, (*Proceedings of SIGGRAPH '88*, Atlanta, Georgia, August 1–5, 1988), pp. 239–246.

[Akel93] Akeley, Kurt, "RealityEngine Graphics," Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993). In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, pp. 109–116.

[Alma94] Almasi, George S., and Allan Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, California, 1994.

[ANSI88] ANSI (American National Standards Institute), *American National Standards for Information Processing Systems—Programmer's Hierarchical Interactive Graphics System (PHIGS) Functional Description, Archive File Format, Clear-Text Encoding of Archive File*, ANSI, X3.144-1988, ANSI, New York, 1988.

[Apga88] Apgar, Brian, Bret Bersack, and Abraham Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000", *Computer Graphics*, Vol. 22, No. 4, August 1988, (*Proceedings of SIGGRAPH '88*, Atlanta, Georgia, August 1–5, 1988), pp. 255–262.

[Atha88] Athas, William C., and Charles L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, Vol. 21, No. 8, August 1988, pp. 9–24.

[Bail94] Bailey, David H., Eric Barszcz, Leonardo Dagum, and Horst D. Simon, "NAS Parallel Benchmark Results 10-94," NAS Technical Report NAS-94-001, October 1994, NASA Ames Research Center, Moffett Field, California.

[Bell85] Bell, C. Gordon, "Multis: A New Class of Multiprocessor Computers," Science, Vol. 228, No. 4698, April 26, 1985, pp. 462–467.

[Blin77] Blinn, James F., "Models of Light Reflection for Computer Synthesized Pictures," *Computer Graphics*, Vol. 11, No. 2, Summer 1977, (Proceedings of SIGGRAPH '77 (San Jose, California, July 20–22, 1977), pp. 192–198.

[Bokh90] Bokhari, Shadid, "Communication Overhead on the Intel iPSC-860 Hypercube," ICASE Interim Report 10 (NASA CR 182055), Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia, May 1990.

[Braw89] Brawer, Steven, *Introduction to Parallel Programming*, Academic Press, San Diego, California, 1989.

[Buit75] Bui-Tong, Phong, "Illumination for Computer Generated Pictures," *Communications of the Association for Computing Machinery*, Vol. 18, No. 6, June 1975, pp. 311–317.

[Burk90] Burke, Andrew, and Wm Leler, "Parallelism and Graphics: An Introduction and Annotated Bibliography," in Scott Whitman, Ed., *Parallel Algorithms and Architectures for 3D Image Generation, Course 28 Notes for SIGGRAPH '90* (Dallas, Texas, August 6–10, 1990), pp. 111–140.

[Carp84] Carpenter, Loren, "The A-buffer, An Antialiased Hidden Surface Method", *Computer Graphics*, Vol. 18, No. 3, July 1984, (*Proceedings of SIGGRAPH '84*, Minneapolis, Minnesota, July 23–27, 1984), pp. 103–108.

[Clea83] Cleary, John. G., Brian M. Wyvill, Graham M. Birtwistle, and Reddy Vatti, "Multiprocessor Ray Tracing," Technical Report 83/128/17, Department of Computer Science, University of Calgary, October 1983. Also available *Computer Graphics Forum*, North-Holland Publishers, 1986, pp. 3–12.

[Cook82] Cook, Robert L, and Kenneth E. Torrance, "A Reflectance Model for Computer Graphics," *ACM Transactions on Graphics*, Vol. 1, No. 1, January 1982, pp. 7–24.

[Cox92] Cox, Michael, and Pat Hanrahan, "Depth Complexity in Object Parallel Graphics Architectures," *Proceedings of the Seventh Workshop on Graphics Hardware, Eurographics '92*, Cambridge, England, September 1992.

[Cox95] Cox, Michael B., *Algorithms for Parallel Rendering*, Ph.D. Dissertation, Department of Computer Science, Princeton University, Princeton, New Jersey, May 1995.

[Croc93] Crockett, Thomas W., and Tobias Orloff, "A MIMD Rendering Algorithm for Distributed Memory Architectures", in *Proceedings 1993 Parallel Rendering Symposium*, San Jose, California, October 25–26, 1993, ACM Press, New York, pp. 35–42.

[Croc94] Crockett, Thomas W., "Design Considerations for Parallel Graphics Libraries," *Proceedings of the Intel Supercomputer Users Group 1994 Annual North American Users Conference*, San Diego, CA, June 1994, pp. 3–14. Also available as Institute for Computer Applications in Science and Engineering Report No. 94-49 (NASA CR-194935), NASA Langley Research Center, Hampton, Virginia.

[Crow84] Crow, Franklin C., "Summed-Area Tables for Texture Mapping," *Computer Graphics*, Vol. 18, No. 3, July 1984, (*Proceedings of SIGGRAPH '84*, Minneapolis, Minnesota, July 23–27, 1984), pp. 207–212.

[Crow88] Crow, Franklin, G. Demos, J. Hardy, J. McLaughlin, and K. Sims, "3D Image Synthesis on the Connection Machine," in P. M. Dew, R. A. Earnshaw, and T.R. Heywood, Eds., *Parallel Processing for Computer Vision and Display*, (*Proceedings of Parallel Processing for Computer Vision and Display*, International Conference, University of Leeds, United Kingdom, January 12–15, 1988), Addison-Wesley, Wokingham, England, 1989, pp. 254–269.

[Dall87] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer Academic Publishers, Boston, Massachusetts, 1987, pp. 171–172.

[Dall90] Dally, William J., "Performance Analysis of *k*-ary *n*-cube Interconnection Networks," *IEEE Transactions on Computers*, Vol. 39, No. 6, June 1990, pp. 775–785.

[Deer88] Deering, Michael, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", *Computer Graphics*, Vol. 22, No. 4, August 1988, (*Proceedings of SIGGRAPH '88*, Atlanta, Georgia, August 1–5, 1988), pp. 21–30.

[Deme80] Demetrescu, Stefan G., *A VLSI Based Real-Time Hidden Surface Elimination Display System*, Master's Thesis, Computer Science Department, California Institute of Technology, Pasadena, California, 1980.

[Dipp84] Dippé, Mark, and John Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", *Computer Graphics*, Vol. 18, No. 3, July 1984, (*Proceedings of SIGGRAPH '84*, Minneapolis, Minnesota, July 23–27, 1984), pp. 149–158.

[Duff85] Duff, Tom, "Compositing 3-D Rendered Images", *Computer Graphics*, Vol. 19, No. 3, July 1985, (*Proceedings of SIGGRAPH '85*, San Francisco, California, July 22–26, 1985), pp. 41–44.

[E&S92] Evans and Sutherland Computer Corporation, *Freedom Series Technical Report*, Salt Lake City, Utah, March 1992.

[Ells90a] Ellsworth, David, Howard Good, and Brice Tebbs, "Distributing Display Lists on a Multicomputer," *Computer Graphics*, Vol. 24, No. 2, March 1990, (*Proceedings 1990 Symposium on Interactive 3D Graphics*, Snowbird, Utah, March 25–28, 1990), pp. 147–155.

[Ells90b] Ellsworth, David, and Howard Good, "Pixel-Planes 5 Hierarchical Graphics System Programmer's Manual," *Pixel-Planes 5 System Documentation*, Computer Science Department, University of North Carolina at Chapel Hill, Chapel Hill, NC, 1990.

[Ells93] Ellsworth, David, "A Multicomputer Polygon Rendering Algorithm for Interactive Applications", in *Proceedings 1993 Parallel Rendering Symposium*, San Jose, California, October 25–26, 1993, ACM Press, New York, pp. 43–48.

[Ells94] Ellsworth, David, "A New Algorithm for Interactive Graphics on Multicomputers", *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 33–40.

[Fole90] Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1990.

[Fole94] Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, *Introduction to Computer Graphics*, Addison-Wesley, Reading, Massachusetts, 1994.

[Fuch77] Fuchs, Henry, "Distributing a Visible Surface Algorithm Over Multiple Processors," in *Proceedings of the 1977 ACM Annual Conference*, Seattle, Washington, October 17-19, 1977, pp. 449-451.

[Fuch79] Fuchs, Henry, and Brian W. Johnson, "An Expandable Multiprocessor Architecture for Video Graphics," in *Proceedings of the 6th Annual Symposium on Computer Architecture*, April 23–25, 1979, IEEE Computer Society, pp. 58–67.

[Fuch81] Fuchs, Henry, and John Poulton, "PIXEL-PLANES: A VLSI-Oriented Design for a Raster Graphics Engine," *VLSI Design*, Vol. 2, No. 3, 3rd Quarter 1981, pp. 20–28.

[Fuch85] Fuchs, Henry, Jack Goldfeather, Jeff Hultquist, Susan Spach, John Austin, Frederick Brooks, Jr., John Eyles, and John Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-planes", *Computer Graphics*, Vol. 19, No. 3, July 1985, (*Proceedings of SIGGRAPH '85*, San Francisco, California, July 22–26, 1985), pp. 111–120.

[Fuch89] Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Computer Graphics*, Vol. 23, No. 3, July 1989, (*Proceedings of SIGGRAPH '89*, Boston, Massachusetts, July 31–August 4, 1989), pp. 79–88.

[Ghar88] Gharachorloo, Nader, Satish Gupta, Erdem Hokenek, Peruvemba Balasubramanian, Bill Bogholtz, Christian Mathieu, and Christos Zoulas, "Subnanosecond Pixel Rendering with Million Transistor Chips", *Computer Graphics*, Vol. 22, No. 4, August 1988, (*Proceedings of SIGGRAPH '88*, Atlanta, Georgia, August 1–5, 1988), pp. 41–49.

[Ghar89] Gharachorloo, Nadar, Satish Gupta, Robert Sproull, and Ivan Sutherland, "A Characterization of Ten Rasterization Techniques", *Computer Graphics*, Vol. 23, No. 3, July 1989, (*Proceedings of SIGGRAPH '89*, Boston, Massachusetts, July 31–August 4, 1989), pp. 355–368.

[Gour71] Gouraud, Henri, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, Vol. C-20, No. 6, June 1971, pp. 623–629.

[Gree89] Green, Stuart, and Derek Paddon, "Exploiting Coherence for Multiprocessor Ray Tracing," *IEEE Computer Graphics and Applications*, Vol. 9, No. 6, November 1989, pp. 12–26.

[Gree91] Green, Stuart, *Parallel Processing for Computer Graphics*, MIT Press, Cambridge, Massachusetts, 1991.

[Haeb90] Haeberli, Paul E., and Kurt Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering", *Computer Graphics*, Vol. 24, No. 4, August 1990, (*Proceedings of SIGGRAPH '90*, Dallas, Texas, August 6–10, 1990), pp. 309–318.

[Hans92] Hansen, Charles D, and Paul Hinker, "Massively Parallel Isosurface Extraction," *Proceedings of Visualization '92* (October 19–23, Boston Massachusetts), IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 77–83.

[Hill93] Hillis, W. Daniel and Lewis W. Tucker, "The CM-5 Connection Machine: A Scalable Supercomputer," *Communications of the ACM*, Vol. 36, No. 11 (November 1993), pp. 31–40.

[Hu85] Hu, Mei-Cheng, and James Foley, "Parallel Processing Approaches to Hidden Surface Removal," *Computers & Graphics*, Vol. 9, No. 3, 1985, pp. 303–317.

[Hult92] Hultquist, Jeff P., "Constructing Stream Surfaces in Steady 3D Vector Fields," *Proceedings of Visualization '92* (October 19–23, Boston Massachusetts), IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 171–178.

[Inte91a] *Touchstone Delta System Description*, Technical Report Intel Advanced Information, Intel Corporation, 1992.

[Inte91b] Intel Corporation, *Touchstone Delta System User's Guide*, Order Number 312125-001, Intel Supercomputer Systems Division, Beaverton, Oregon, 1991.

[Inte92] *Paragon XP/S System Description*, Technical Report Intel Advanced Information, Intel Corporation, 1992.

[Kenw92] Kenwrite, David N., and Gordon D. Mallinson, "A 3-D Streamline Tracking Algorithm Using Dual Stream Functions," *Proceedings of Visualization '92* (October 19–23, Boston Massachusetts), IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 62–68.

[Kess93] Kessler, R. E., J. L. Schwarzmeier, "Cray T3D: A New Dimension for Cray Research," in *Proceedings COMPCON Spring '93* (San Francisco, CA, February 22–26, 1993), IEEE Computer Society Press, Los Alamitos, California, 1993, pp. 176–182.

[Kili94] Kilian, Alan, Personal communication (electronic mail messages sent to the mp-render mailing list), 1994.

[Koba87] Kobayashi, Hiroaki, Tadao Nakamura, and Yoshiharu Shigei, "Parallel Processing of an Object Space for Image Synthesis Using Ray Tracing," *The Visual Computer*, Vol. 3, No. 1, February 1987, pp. 13–22.

[Kubo93] Kubota Pacific Computer, Inc., *Denali Technical Overview*, Version 1.0, Santa Clara, California, March 1993.

[Kuma94] Kumar, Vipin, Anath Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings, Redwood City, California, 1994, pp. 90–98.

[Kung82] Kung, H. T., "Why Systolic Arrays?," *IEEE Computer*, Vol. 15, No. 1, January 1982, pp. 37–46.

[Kwan93] Kwan, Thomas T., Brian K. Totty, and Daniel A. Reed, "Communication and Computation Performance of the CM-5," in *Proceedings of Supercomputing '93*, Portland, Oregon, November 15–17, 1993, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 192–201.

[Lawr75] Lawrie, D. H., "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, Vol. C-24, No. 12, December 1975, pp. 1145–1155.

[Lee95] Lee, Tong-Yee, C. S. Raghavendra, and J. N. Nicholas, "Image Composition Methods for Sort-Last Polygon Rendering on 2-D Mesh Architectures," in *Proceedings of the 1995 Parallel Rendering Symposium*, October 30–31, 1995, Atlanta, Georgia, ACM Press, New York, pp. 55–62.

[Litt92] Littlefield, Richard J., "Characterizing and Tuning Communication Performance on the Touchstone Delta and iPSC/860", *Proceedings of the 1992 Delta Applications Workshop*, January 21–22, 1992, Pasadena, California.

[Lore87] Lorensen, William, and Harvey Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", *Computer Graphics*, Vol. 21, No. 4, July 1987, (*Proceedings of SIGGRAPH '87*, Anaheim, California, July 27–31, 1987), pp. 163–169.

[McCo87] McCormick, Bruce H., Thomas A. DeFanti, and Maxine D. Brown, Eds., "Visualization in Scientific Computing," *Computer Graphics*, Vol. 21. No. 6, November 1987.

[Moln91a] Molnar, Steven E., "Efficient Supersampling Antialiasing for High-Performance Architectures," Technical Report TR-91-023, Department of Computer Science, University of North Carolina at Chapel Hill, 1991.

[Moln91b] Molnar, Steven E., *Image-Composition Architectures for Real-Time Image Generation*, Ph.D. Dissertation, Computer Science Department, University of North Carolina, Chapel Hill, North Carolina (Technical Report TR91-046), 1991.

[Moln92] Molnar, Steven, John Eyles, and John Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Computer Graphics*, Vol. 26, No. 2, July 1992, (*Proceedings of SIGGRAPH '92*, Chicago, Illinois, July 26–31, 1992), pp. 231–240 .

[Moln94] Molnar, Steven E., Michael Cox, David Ellsworth, and Henry Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, Vol. 14., No. 4, July 1994, pp. 23–32.

[Muel95] Mueller, Carl, "The Sort-First Rendering Architecture for High-Performance Graphics," *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, Monterey, California, April 9–12, 1995, ACM Press, New York, New York, 1995, pp. 75–84.

[Orte93] Ortega, Frank A., Charles D. Hansen, and James P. Ahrens, "Fast Data Parallel Polygon Rendering," in *Proceedings of Supercomputing '93*, Portland, Oregon, November 15–17, 1993, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 709–718.

[Pack87] Packer, Jamie, "Exploiting Concurrency; A Ray Tracing Examples," Inmos Technical Note 7, Inmos, Ltd., Bristol, United Kingdom, 1987.

[Park80] Parke, Frederic, "Simulation and Expected Performance Analysis of Multiple Processor Z-buffer Systems," *Computer Graphics*, Vol. 14, No. 3, July 1980, (*Proceedings of SIGGRAPH '80*, Seattle, Washington, July 14–18, 1980), pp. 48–56.

[Park82] Parks, Joseph K., *A Comparison of Two Multiprocessor Graphics Machine Designs*, Masters Thesis, Computer Science Department, University of North Carolina, Chapel Hill, North Carolina, 1982.

[Quin87] Quinn, Michael J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987.

[Robl88] Roble, Douglas R., "A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube," in *Proceedings of Pixim '88*, Paris, France, October 1988.

[Roge85] Rogers, David F., *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.

[Rohl94] Rohlf, John, and James Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics* , Annual Conference Series, 1994, ACM SIGGRAPH, New York, pp. 381–394.

[Sain95] Saini, Subhash and David H. Bailey, "NAS Parallel Benchmark Results 3-95," NAS Technical Report NAS-95-011, April 1995, NASA Ames Research Center, Moffett Field, California.

[Scha83] Schacter, Bruce J., *Computer Image Generation*, John Wiley and Sons, New York, 1983, pp. 60–66.

[Schn88] Schneider, Bengt-Olaf, and Ute Claussen, "PROOF: An Architecture for Rendering in Object Space," in A. A. M. Kuijk, Ed., *Advances in Computer Graphics Hardware III*, (*Proceedings Third Eurographics Workshop on Graphics Hardware*, Sophia-Antipolis, France, September 1988), Springer-Verlag, Berlin, 1991, pp. 121–140.

[Seth88] Sethian, J. A., James B. Salem, and A. F. Ghoniem, "Interactive Scientific Visualization and Parallel Display Techniques," Thinking Machines Corporation Technical Report VZ88-1, May 1988.

[SGI90] Silicon Graphics Computer Systems, *Power Series Technical Report*, Mountain View, California, 1990.

[SGI92] Silicon Graphics Computer Systems, *Reality Engine™ in Visual Simulation Technical Overview*, Document RE-VisSim-TR(8/92), Mountain View, California, 1992.

[SGI93] Silicon Graphics Computer Systems, *Graphics Library Programming Guide*, Vol. II, Document number 007-1702-020, Mountain View, California, August 1993.

[Shaw88] Shaw, Christopher, Mark Green, and Jonathan Shaeffer, "A VLSI Architecture for Image Composition," in A. A. M. Kuijk, Ed., *Advances in Computer Graphics Hardware III*, (*Proceedings Third Eurographics Workshop on Graphics Hardware*, Sophia-Antipolis, France, September 1988), Springer-Verlag, Berlin, 1991, pp. 183-199.

[Stol95] Stoll, Gordon, Bin Wei, Douglas Clark, Edward Felten, Kai Ki, and Patrick Hanrahan, "Evaluating Multi-Port Frame Buffer Designs for a Mesh Connected Multicomputer," *Proceedings of the 22nd International Symposium on Computer Architecture* , Santa Margherita Ligure, Italy, June 2224, 1995, pp. 96–105.

[Suth74] Sutherland, Ivan E., Robert F. Sproull, and Robert A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, Vol. 6, No. 1, March 1974, pp. 1–55.

[Tebb90] Tebbs, Brice, Ulrich Neumann, John Eyles, Greg Turk, and David Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading", International Workshop on Algorithms and Parallel VLSI Architectures, Pont-à-Mousson, France, June 10–16, 1990, pp. 152–156. Also published as Technical Report 92-034, Computer Science Department, University of North Carolina at Chapel Hill, 1992.

[Thom93] Thomas, John, Personal communication, 1993.

[Thom96] Thompson, Tom, "The World's Fastest Supercomputer (for Now)," *Byte*, Vol. 21, No. 1, January 1996, p. 62.

[Upst89] Upstill, Steve, *The RenderMan Companion*, Addison-Wesley, Reading, Massachusetts, 1989.

[Vis92] *Proceedings of Visualization '92* (October 19–23, 1992, Boston, Massachusetts), IEEE Computer Society Press, Los Alamitos, California, 1992.

[Vis93] *Proceedings of Visualization '93* (October 25–29, 1993, San Jose, California), IEEE Computer Society Press, Los Alamitos, California, 1993.

[Vis94] *Proceedings of Visualization '94* (October 17–21, 1994, Washington, DC), IEEE Computer Society Press, Los Alamitos, California, 1994.

[Vis95] *Proceedings of Visualization '95* (October 29–November 3, 1995, Atlanta, Georgia), IEEE Computer Society Press, Los Alamitos, California, 1995.

[Wei95] Wei, Bin, Gordon Stoll, Douglas Clark, Edward W. Felten, Kai Li, and Patrick Hanrahan, "Synchronization for a Multi-Port Frame Buffer on a Mesh-Connected Multicomputer," *Proceedings of the 1995 Parallel Rendering Symposium*, Atlanta, Georgia, October 30–31, 1995, ACM Press, New York, pp. 81–88.

[Wein81] Weinberg, Richard, "Parallel Processing Image Synthesis and Anti-Aliasing," *Computer Graphics*, Vol. 15, No. 3, August 1981, (*Proceedings of SIGGRAPH '81*, Dallas, Texas, August 3-7, 1981), pp. 55-61.

[Whel85] Whelan, Daniel, *A Multiprocessor Architecture for Real-Time Computer Animation*, Ph. D. Dissertation, Department of Computer Science, California Institute of Technology, Pasadena, California (Technical Report TR 5200), 1985.

[Whit92] Whitman, Scott, *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Ltd., Wellesley, Massachusetts, 1992.

[Whit94] Whitman, Scott, "Dynamic Load Balancing for Parallel Polygon Rendering", *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 41–48.