Load Balancing for Interactive Display of Surfaces*

Subodh Kumar　　Chun-Fa Chang　　Dinesh Manocha

University of North Carolina
Chapel Hill, NC 27599-3175, USA
Ph: (919) 962-1943. Fax: (919) 962-1799.
Email: {kumar,chang,manocha}@cs.unc.edu

Abstract

We present efficient parallel algorithms for interactive display of higher order surfaces on current graphics systems. At each frame, these algorithms approximate the surface by polygons and rasterize them over the graphics pipeline. The time for polygon generation for each primitive varies between successive frames and we address a number of issues related to balancing the load across processors. This includes algorithms to statically distribute the primitives, reduce dynamic load imbalance as well as *distributed wait-free* algorithms for machines on which re-distribution is efficient, e.g. shared memory machine. These algorithms have been implemented on different graphics systems and applied to interactive display of trimmed spline models. In practice, we are able to obtain almost linear speed-ups (as a function of number of processors). Moreover, the distributed wait-free algorithm is faster by $25-30\%$ as compared to static and dynamic schemes.

Keywords: Surface tessellation, Load balancing, Real-time rendering, Virtual Reality, NURBS.

1 Introduction

Higher order surfaces are ubiquitously used in computer graphics and geometric modeling. This includes splines, NURBS, algebraic surfaces and generalized implicit models. A number of techniques based on polygonization, ray-tracing, scan-line conversion and pixel-level subdivision have been proposed for rendering them on current graphics systems. However, in practice, only the algorithms based on polygonization are able to render large models at *interactive frame* rates. At each frame, these algorithms approximate the surface using triangles and render the resulting triangles over the graphics pipeline (using Gouraud or Phong shading). In terms of performance, a standard graphics primitive like a triangle, takes almost the same time to render per frame. On most current graphics systems, the time to render a primitive *varies* with its on-screen size [Fea89, Ake93], but that variation is relatively small as compared to the total rendering time. As a result, the overall time to render higher order surfaces is mainly determined by time for polygon generation and the number of polygons generated. In this paper, we mainly deal with one such class of surfaces: *trimmed NURBS surfaces*, and demonstrate our algorithm on the model of a notional submarine storage and handling system (shown in Fig. 1, 38,000 trimmed Bézier patches).

*Supported in part by DARPA ISTO Order No. A410, NSF Grant No. MIP-9306208, an Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, NSF Grant CCR-9319957, ONR Contract N00014-94-1-07 38, ARPA Contract DABT63-93-C-0048 and NSF/ARPA Center for Computer Graphics and Scientific Visualization.

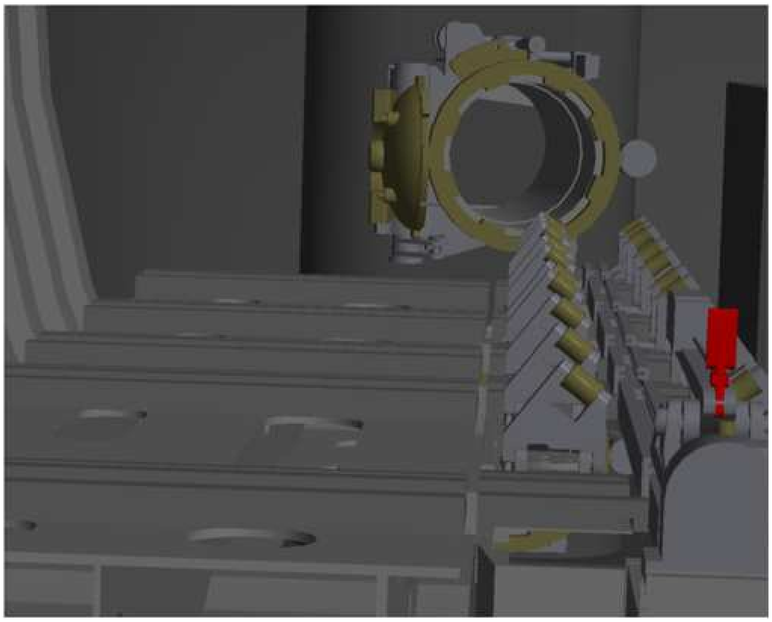


Figure 1: Submarine Storage and Handling System

Recently, a number of polygonization based trimmed spline renderers have been proposed in the literature [AES91, AES94, RHD89, LC93, KML95, KM95, KS95]. These algorithms use the host CPU's to approximate the surfaces with polygons, and employ standard graphics hardware to render these polygons. As we deal with large models composed of tens of thousands of surfaces, single CPU graphics systems are currently not fast enough to polygonize large models at interactive frame rates. As a result, interactive algorithms for rendering utilize multi-processor configurations. The algorithms for tessellating spline surfaces are relatively simple to parallelize. However, the distribution of surfaces along different processors is important for the overall performance. The time to polygonize each surface varies as a function of the viewpoint. Furthermore, for many of these spline rendering algorithms, polygon generation is often a bottleneck on systems with high end graphics engines [Fea89, Ake93]. This means the triangle pipeline is often idle.

In this paper, we present static and dynamic *load balancing* algorithms for interactive display of large models defined using higher order surfaces. The algorithms presented are general purpose, but we specialize them for interactive display of trimmed spline surfaces. The *main contributions* are:

1. **Balancing static and dynamic load:** We present a static allocation scheme for distributing the model across multiple processors and reduce dynamic load imbalance. The resulting algorithm takes into visibility computations (like view-frustum culling) and highly varying transformations. It works very well on graphics systems, where re-distributing primitives between processors can be prohibitively slow, as this essentially involves data movement between processors. The resulting algorithm exploits *frame-to-frame coherence.* In particular, it is based on the observation that such load exhibits spatial coherence. Primitives on same part of the screen tend to have similar statistical load. We also extend it to configurations, where re-allocation is efficient, e.g. shared memory machines. We present a *scalable, wait-free* algorithm to re-distribute primitives, whenever the load becomes imbalanced. This algorithm is *distributive*, in the sense that there exists no 'master' processor spending resources on

load-balancing.

2. **Greedy rendering:** We present a *greedy rendering* technique to update the scene as a function of viewpoint. The resulting algorithm lowers the system latency for rendering higher order surfaces and is very important for *head-mounted displays* and *walkthrough applications*, where a lagging image can induce motion-sickness. The resulting algorithm uses the concept of greedy rendering, which is like *progressive refinement* to improve the quality of the image whenever the user motion stops.

The resulting algorithms have been implemented on different graphics systems and applied to a number of models. In practice, we are able to obtain 80-90% of the ideal speed-up, using the best algorithm. The static allocation scheme improves the average frame rate by $15-20\%$ as compared to round robin allocation schemes. The distributed algorithm improves it further by $25-30\%$ on shared-memory graphics systems.

1.1 Related Work

Load-imbalance is an old and well studied problem in parallel and distributed computing. [Gea95] offers an excellent survey on load-balancing techniques. If the load is known à priori, it can be optimally allocated to processors in an off-line process, spending little time at run-time to manage load. For dynamic loads, a much more dynamic algorithm is warranted. In the graphics literature a number of algorithms have been proposed for polygonal models and for parallel ray-tracing. In [Rob88, Whi94, Mue95] algorithms dividing the primitives in terms of screen-regions are presented. Moreover, [PB89, EGT90, ZKN92] balance the load in object space. However, these techniques cannot be applied to rendering of higher-order surfaces, as the rendering-cost of a NURBS surface *varies* significantly across frames. For dynamic load-balancing, in the presence of shared-memory, distributed computing literature presents a number of algorithms to arbitrate shared accesses with consistency [Lam87, YA93, Her93]. Indeed, for each sequential data structure there exists a shared implementation that requires no locks [Her93]. For example, [Her93] present a hierarchy of shared objects, with wait-free accesses. However, the objects presented in these papers are more general and do not result in significant performance improvement. Moreover, many of these implementations rely on the existence of atomic 'test and set' like instructions, which may not be available on all graphics systems.

The rest of the paper is organized in the following manner. We briefly review the NURBS pipeline in Section 2. The static load-balancing technique is presented in Section 3. In Section 4 we present the primitive re-distribution algorithm. We consider the problem of real-time display and present the *greedy rendering* algorithm in Section 5. We discuss its implementation and performance in Section 6.

2 NURBS Rendering

The NURBS surfaces are rendered based on the algorithm presented in [KM95]. Given a trimmed NURBS model, the algorithm represent them as trimmed Bézier patches using knot-insertion algorithm. At run-time it tessellates each patch into an appropriate number of triangles at each frame (see Fig. 2). A brief overview of this algorithm is given below.

1. Perform view-frustum and back-facing patch visibility to eliminate hidden patches.

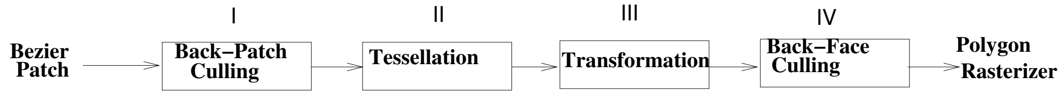


Figure 2: NURBS rendering Pipeline

2. For each frame, given the viewing matrix, compute the required tessellation step sizes n_{u_p}, n_{v_p} for each patch p, and n_{t_c} for each trimming curve c.
3. Tessellate patch p into quads, choosing tessellants $\frac{1}{n_{u_p}}$ and $\frac{1}{n_{v_p}}$ apart along the u and v parametric axes respectively. Tessellate trimming curve c into n_{t_p} piecewise linear segments.
4. Generate triangles for the patch by triangulating the region enclosed by the trimming curve, using the tessellants generated in step 3.

Any triangulation-based surface rendering algorithm first needs to allocate resources to generate these triangles. The desirable tessellation is computed and the vertices and normals are evaluated. The total time is, therefore a function of the number of triangles generated. The performance of triangle rendering is system dependent and typically a function of the number of triangles and the size and distribution of these triangles on the screen.

It is possible to compute a very fine triangulation à priori and to render all the triangles for each frame. In this case, almost no time is spent in triangle generation and all of the time is spent on rendering. However the number of triangles needed for close-up (zoomed) views of some surfaces can be extremely high (a few thousand) and for models consisting of thousands of surfaces, this requires hundreds of megabytes of storage, and the capability to render hundreds of millions of triangles per second. We can reduce the demand on triangles rendering capability by computing different *levels of detail* of each surface and at each phase choosing one of the approximations as a function of the viewing parameters. But the memory requirements only get worse.

On the other hand, we can compute, on-line, the minimum number of triangles required for a smooth image as a function of the viewing parameters (for each frame). The resulting algorithm is based on adaptive subdivision and takes considerable time in the triangle generation for each frame. As a result, it can be too slow for interactive performance on large-scale models.

A *major goal* of this paper is to present a hybrid between the two solutions, combining the benefits of the two techniques.

3 Static Load Balancing

When the user zooms in to a small part of the model, that part occupies a significant part of the screen, and hence a significant part of the total tessellation cost. If that part of the cost is not fairly distributed between processors, some of them become bottleneck. If we randomly distribute primitives, say in a round robin manner, to processors, load imbalances of more than 1:50 is not uncommon.

A number of load balancing algorithms reduce the problem to graph partitioning [HL95, KK95]. All these algorithms assume the existence of a load-graph. To construct such a graph, we must first know the processing cost of each primitive. However, the rendering cost of a NURBS primitive is a function of parameters such as:

- the degree of surface

- the complexity of the trimming curves
- the screen size of the primitive

Though, the dependence of this cost on the viewing parameters makes it difficult to find an assignment which is optimal in all frames, primitives on same part of the screen tend to have similar load. This observation of spatial coherence suggests the distribution of 'nearby' primitives to different processors.

To model our problem as a graph partitioning problem, each primitive is represented by a vertex of the graph. The vertices and edges are assigned weights so that sub-graph weights estimate rendering cost well. Initially, a complete graph with n vertices is constructed where n is the number of primitives. The weights are assigned as follows:

- **Vertex weight**: the sum of the estimated rendering cost of the patches in the primitive. This is a function of patch degrees, and the degrees of the trimming curves.
- **Edge weight**: the inverse of the geometric distance between the primitives of two vertices it connects.
- **Subgraph cost function**: For each edge, we calculate $W_e(W_{v1} + W_{v2})$, where W_e is the edge weight and W_{v1}, W_{v2} are the vertex weight of the vertices the edge connects. The cost function is the sum of the above function for every edge in the subgraph.

Our goal is to partition the graph into p disjoint subgraphs of almost equal weight, where p is the number of processors and the cost function in each subgraph is minimized,

Approximation algorithms such as simulated annealing can then be used for optimization. The heuristic is: when two nearby primitives are in the same processors, the cost function of the subgraph for that processor is increased because the the weight of the edge connecting them is high. Thus one of the primitive is very likely to be moved to the other processor during the optimization process.

4 Primitive Re-distribution

As mentioned earlier, in any given frame the processor load may not be balanced. This means that some processors may finish the tessellation of their work-load while others are still tessellating the surfaces. We refer to the first set as *idle* and the second one as *busy*. This work-load is dynamic, in the sense that the cost to render the same set of patches changes with time. It is precisely due to this reason that a static primitive distribution cannot achieve optimal speedup. For systems with efficient inter-processor communication, primitive re-distribution results in much more balanced better load, and with little overhead.

In this section we present two simple re-distribution algorithms. The first algorithm maintains a global queue of patches, arbitrating access to the queue using *locks*. The second algorithm improves this scheme by eliminating processor waits. Furthermore, it is much more scalable as the number of available processors increases.

4.1 Global Queue (with locking)

Each element of the queue (Fig. 3 corresponds to N_p patches. Each processor deletes the element in the front of the queue and computes the tessellation for the corresponding patches and renders the triangles. This step is repeated until the queue becomes empty. The granularity N_p of a queue element affects the processor utilization. In our experience, N_p should be a small fraction of N, the total number of patches to be rendered. Since, a 1 : 100 load imbalance can sometimes occur for static schemes, we use a value of $\frac{N}{100P}$, where P is the number of processors.

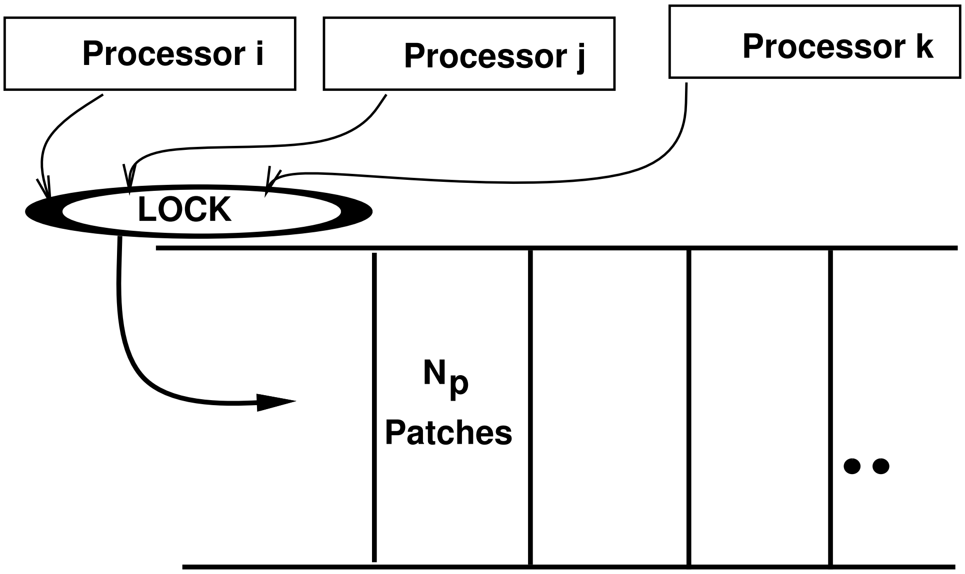


Figure 3: Global Queue, with locking as a potential bottleneck

A problem with this method is the fixed choice of granularity. Different values of N_p are required to achieve optimal speed-up for different environments and models. In addition, although, this algorithm performs better than the static distribution algorithm, the overhead of lock-maintenance and mutual-exclusion prevents us from making full use of dynamic load-balancing. Next, we propose an algorithm that reduces this overhead (based on *load stealing*). At each frame, each processor starts with an approximate work-load. As processors get idle, they steal patches from busy processors, computing the tessellation of those additional patches.

4.2 Load Stealing

Each processor p maintains its current work-load in a global structure labeled as $ActivityList[p]$. It is the list of patches, that are allocated to p for the current frame. At the beginning of simulation, this allocation is same as the static allocation presented in Section 3. At each frame, each processor runs its tessellation loop and updates its allocation.

4.2.1 Tessellation Loop

Due to temporal coherence, the rendering cost of a primitive does not vary significantly between successive frames. Hence the *ActivityList* for a processor at the previous frame is a good starting guess for a balanced *ActivityList* for the current frame. The basic idea of our algorithm is to allow no additional overhead for busy processors. All the extra computation is done by the idle processors. Here is the basic tessellation loop:[1]

While (*Next* = *Front* of *ActivityList* is not End of list){ :1

[1]To simplify the notation, we don't index variables by the processor id, when it is clear from context.

```
        Update Front;                                              :2
        tessellate (PATCH Next)                                    :3
        update triangles for Next                                  :4
    }                                                              :5
    Find a busy processor p_b, Share its load.                     :6
```

In this algorithm, possible concurrent access to shared variables occur at line 6. The first contention at line 6 is between idle processors. Multiple processors may 'find' the same p_b. To reduce such contention, each processor maintains a list of other processors in a random order, and checks processors' *ActivityList*s in that order. Of course, this does not guarantee mutual exclusion. This access is arbitrated by locks. Each processor has a lock, *LockList*, associated with it. Before an idle processor p_i checks the *ActivityList* of p_b ($i \neq b$), it secures a *non-blocking lock* on *LockList*[p_b]. Non-blocking lock ensures that if p_i cannot acquire *LockList*[p_b], there must exist another idle processor p_j currently sharing *ActivityList*[p_b]. A processor p_i, when idle, executes the following loop until all tessellators become idle:

```
for p_b in RandomList {                                            :61
    lock LockList[p_b]                                             :62
    if lock not acquired, go on to next in RandomList              :63
    if (Front of ActivityList[p_b] is not End of list)             :64
        Share load with processor p_b                              :65
    unlock LockList[p_b]                                           :66
    Perform Tessellation loop                                      :67
}                                                                  :68
```

4.2.2 Lock-free implementation

Once the idle processor used for sharing a busy processor's load is fixed, the only possible concurrent shared variable access can occur at line 65. This contention is between the idle processor p_i and the selected busy processor p_b. Since it involves a busy processor, exclusion by locking is not an option, since we seek to introduce no overhead at the busy processors. That contention is resolved by letting the p_i update the *ActivityList* of p_b, asynchronously. It is possible that p_b reads the old *ActivityList*, and tessellates some patches that are taken off its *ActivityList* by p_i. This case is handled by letting p_i re-read the current position of p_b after updating its *ActivityList*, and not tessellate any patches of new *ActivityList*[p_i], that p_b may already have tessellated. The resulting algorithm is:

```
Delete the second half of unprocessed ActivityList[p_b]            :651
Add it to ActivityList[p_i]                                        :652
Read new Front[p_b] (p_b's copy)                                   :653
Mark the Tessellation loop to start after new Front.               :654
```

There still exists a race condition, in that p_b could have read its *Front* (line 1) just before p_i updates its *ActivityList* (at line 651), and not yet written the new *Front* (line 2) when p_i reads it back (at line 653). If the cost of tessellating each primitive is not high, we can let the two processors duplicate the effort of tessellating one primitive, when this race condition occurs, since it is quite rare. Note that line 653 appears after line 652 by design, since it reduces the probability of race condition, assuming all processors are equally fast. However, the race condition can be eliminated using the following modification.

Busy processors write an additional binary shared variable, $ListTransient$. Each busy processor p sets its $ListTransient$ to 1 before reading its $ActivityList$, and resets it to 0 after having updated its $Front$. Now, we let the idle processor busy-wait[2] for $ListTransient[p_b]$ to be 0 before reading the new $Front$ of p_b.

In practice, $ActivityList$ is not implemented as a list of patch ids but as a list of id ranges. By design, the algorithm tends to keep adjacent patches on same processor. This means, a processor's work load does not consist of just a random list of patches, but, rather, a number of contiguously stored groups of patches.

This algorithm works well because work-load consistency is always maintained. No patches are lost, since p_i does not tessellate a patch, j, only if it knows that p_b must render j. At the same time, p_i does not tessellate any duplicate patches, since p_b's queue is updated before p_i reads the new $Front$. We omit the rigorous details due to space limitations.

5 Greedy Rendering

A fundamental component of real-time graphics is to have the image appear on screen in time. The quality of such image may not be of primary concern. In its most general sense, such applications are referred to time-critical rendering. We present a method to allocate the time spent in each frame in which all triangles must be sent down the rendering pipeline. This is an essential component of in-time rendering.

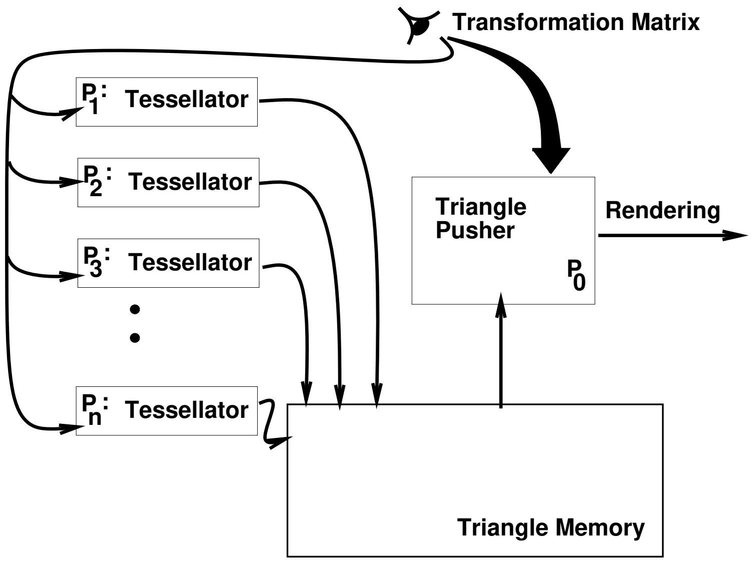


Figure 4: Greedy Rendering Technique

5.1 System Architecture

Fig. 4 shows our overall system architecture. A triangle pushing processor is allocated for the sole purpose of sending triangles down the rendering pipeline. This processor runs asynchronously with the tessellators. It has a 'view' of the current set of triangles to be rendered.

[2]For multiple processors, busy-waiting is the right solution, because it does not induce context-switch.

This triangle pushing processor executes the following loop:

```
Get Current Viewer Position                          :P1
Compute and push transformation matrix               :P2
For each triangle in view                            :P3
     write triangle to the pipeline.                 :P4
```

The triangle pusher never waits for any tessellators to finish. The tessellators, asynchronously generate triangles and update the pusher's view. The tessellators, however, themselves execute in synchrony: a tessellator goes on to frame number $i+1$ only if all patches have been tessellated for frame i. The tessellators follow the algorithm presented in Section 4. Once a frame is complete, all tessellators read the then current view-matrix from a shared global variable, and use this matrix to generate new triangles. Clearly, if tessellation is slow, as is commonly the case, we almost always render sub-optimal number of triangle, as the tessellator *lag* behind the renderer by a few frames. The tessellators mostly generate triangles with an old view-matrix. This, in our, experience is not a major problem, due to coherence. The tessellators mostly lag by no more than $3-4$ frames. And the view-matrix does not change significantly in $3-4$ frames, meaning the 'old' triangles lead to a reasonably smooth image.

5.2 View Update

It is important to update the view in a fashion that the pusher never mixes old and new triangles. In addition, patches are not independent entities. The triangles for adjacent patches must match up at the patch boundaries to prevent *cracks*. We partition the adjacency graph of patches into connected components. The update of all members of a connected components occurs simultaneously. For each component c there exists a shared variable $ComponentTris(c)$. $ComponentTris(c)$ is a pointer to a list of triangle addresses. Each address corresponds to the triangles of a patch, which are stored contiguously in memory. The tessellator generating triangles for patch i, of component c, stores the address of new triangles in $Newlist[i]$. Once the new triangles have been generated, and the pointer list set up, the tessellator that generates the last triangle of the group, writes the address of the new list in $ComponentTris(c)$.

```
Generate Triangles at address a
Newlist[i] = a
if for all patches in the component NewList[i] is non NULL
   ComponentTris(c) = NewList
```

To ensure that the pusher, does not read triangles from different list. It reads ComponentTris(c) just once per frame, and saves it locally. Thus it either renders all new, or all old triangles. Modifying line P3 to reflect this:

```
List = ComponentTris(c)
for each patch i in component c
Render all triangles at List(i)
```

5.3 General Environments

This method works well for environments consisting of a number of small solid models, resulting in a number of small components. In environments where this is not the case, we must use a more general technique:

1. Tessellate patch boundaries separately from the patch interior
2. Generate boundary strips between the boundary and patch interior
3. For each boundary with a different patch, generate boundary strips N_i with new interior tessellation and new boundary tessellation, and strips O_i with new interior tessellation and old boundary tessellation.
4. Maintain adjacency graph.
5. If an adjacent boundary has not been tessellated for the current frame, use O_i. If the adjacent boundary has been tessellated, use N_i. Again, the processor to tessellate a boundary second updates the address on behalf of both processors.

In practice, we have found it much more efficient, and hardly distracting, to render the images with cracks, and let the cracks 'fill up' when the user stops, and the tessellation catches up.

As a final note, let us emphasize that this method is not necessarily bound to multiple processor environments. For a given application, even single processors can allocate triangle pushing time per frame. For example, suppose it takes t_p seconds to process each triangle. For each frame the triangle pushing thread has an account of the number of triangles available for that frame, call it m. Suppose further that the desired frame rate is R frames per second. In each frame, then, the processor must allocate no more than $\frac{1}{R} - mt_p$ seconds per frame for triangle generation.

6 Implementation and Performance

On Pixel Plane 5, a heterogeneous message passing multicomputer, we implemented our static load balancing scheme presented in section 3 and compared its performance with random distribution. With a configuration of 25 graphics processors, our scheme shows an average speedup of 15-20%. We evaluated our algorithms on a number of models, including various parts of the submarine storage and handling system. Table 1 shows the increase in average processor utilization using the graph partitioning algorithm, and the actual speedup gained.

		Round-Robin Distribution		Graph Partioning	
Model	# Patches	Frames/Sec	Utilization	Frames/Sec	Utilization
Torpedo	1201	10.80	72.6%	11.74	80.1%
Pivot	4101	15.41	61.9%	15.95	81%
Ship	3392	19.05	65.3%	19.86	78%

Table 1: Performance of Static Load-distribution Algorithm

On Silicon Graphics Onyx/Reality Engine II, we implemented the 'static', 'global queue', and 'load stealing' load distribution schemes, and compared their performances. Fig. 5 shows three graphs with models of 5302, 10,604, and 15,906 trimmed NURBS patches respectively. Each graph shows the change of the tessellation rate[3] with different number of processors. The graphs show that the global queue scheme provides 5% to 10% improvement over the static scheme, while the

[3] Since greedy rendering proceeds asynchronously, the frame-update rate is hardly affected. The lag changes with the tessellation rate.

FRAMES/Sec

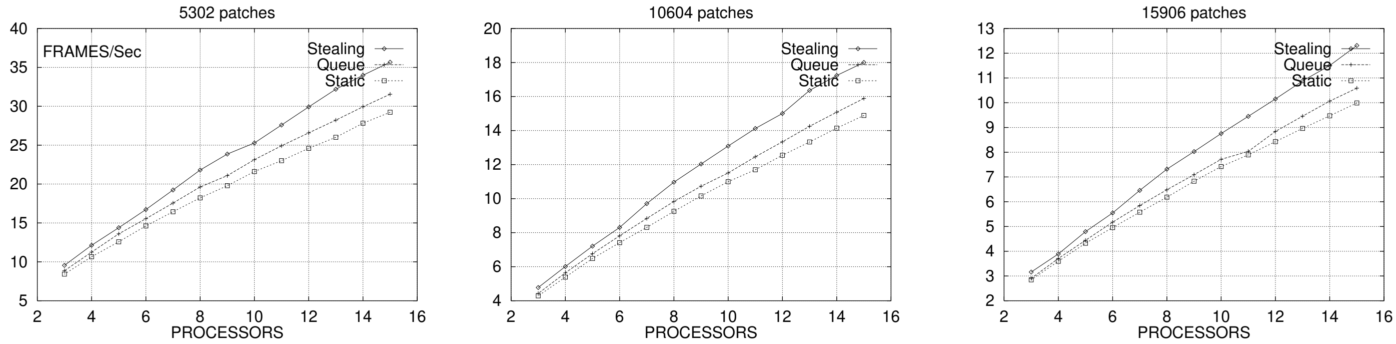


Figure 5: Performance Comparison of three load-balancing techniques

load stealing scheme provides another 10% to 20% improvement over the global queue scheme. In addition, as the number of processors increases, the load stealing scheme shows the best scaling behavior.

If a large number of processors are not available, or if the size of the model is large, the tessellation rate drops to under 5 frames per second. With greedy rendering, we are consistently able to render in the speed of about 4 times of the tessellation rate. Table 2 shows the performance of the greedy algorithm on a number of models. Column 5 shows the average number of frames the tessellators lag behind the triangle pusher, and represents the speed-up gained by the greedy algorithm. This speed-up comes at a minimal loss in image quality. The color plate shows the images corresponding to the worst lag in a user sequence for three parts of the submarine storage and handling system. As can be seen, even in the worst case, the lagging images are of reasonable quality.

Model	# Patches	Tessellations/Sec	Updates/Sec	Average Lag
Dragon	5354	4.20	19.96	15.76
Car 1	5053	4.84	19.97	15.13
Car 2	8693	2.20	8.55	6.35

Table 2: Performance of the Greedy Rendering Algorithm

7 Conclusion

We have presented two techniques to distribute work across processors to speed up the rendering of dynamic objects. The greedy rendering technique can reduce image-jumps for most applications, thus reducing motion-sickness. The load-balancing technique can increase processor utilization to 70-90%. If load-redistribution is not a feasible option, in our experience any significant load-balancing is difficult, and quite application specific. On the other hand, load-redistribution can have significant impact on update-rates.

8 Acknowledgements

We wish to thank Anselmo Lastra, and Lars Nyland for their valuable suggestions to optimize our load-balancing effort. Thanks to Lars for making his simulated annealing implementation available to us, and also to John Keyser for all his help with data collection and organization. Our gratitude to Mike Muuss, Paul Stay and others and at the Army research laboratory for making time on their parallel SGI machine available for our experiments. Thanks also to Greg Angelini and Jim Boudroux and Electric Boat for the model of the storage and handling system.

References

[AES91] S.S. Abi-Ezzi and L.A. Shirman. Tessellation of curved surfaces under highly varying transformations. *Proceedings of Eurographics'91*, pages 385–397, 1991.

[AES94] S.S. Abi-Ezzi and S. Subramaniam. Fast dynamic tessellation of trimmed nurbs surfaces. *Computer Graphics Forum*, 13(3):107–26, 1994. Proc. of Eurographics'94.

[Ake93] K. Akeley. Reality engine graphics. In *Proceedings of ACM Siggraph*, pages 109–116, 1993.

[EGT90] D. Ellsworth, H. Good, and B. Tebbs. Distributing display lists on a multicomputer. In *Symposium on Interactive 3D Graphics*, Snowbird, UT, 1990.

[Fea89] H. Fuchs and J. Poulton et al. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of ACM Siggraph*, pages 79–88, 1989.

[Gea95] G. Georgiannakis and C. Houstis et. al. Description of the adaptive resource management problem, cost functions and performance objectives. Technical Report TR 130, The Institute of Computer Science, Foundation for Research and Technology - Hellas, 1995.

[Her93] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Proc. Supercomputing '95*, 1995.

[KK95] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Technical Report TR95-064, Department of Computer Science, University of Minnesota*, 1995.

[KM95] S. Kumar and D. Manocha. Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design*, 27(7):509–521, July 1995.

[KML95] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large scale NURBS models. In *Symposium on Interactive 3D Graphics*, pages 51–58, Monterey, CA, 1995.

[KS95] R. Klein and W. Straber. Large mesh generation from boundary models with parametric face representation. In *Proc. of ACM/Siggraph Symposium on Solid Modeling*, pages 431–440, 1995.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[LC93] W.L. Luken and Fuhua Cheng. Rendering trimmed NURB surfaces. Computer science research report 18669(81711), IBM Research Division, 1993.

[Mue95] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Symposium on Interactive 3D Graphics*, pages 75–84, Monterey, CA, 1995.

[PB89] Thierry Priol and Kadi Bouatouch. Static load balancing for A parallel ray tracing on a MIMD hypercube. *The Visual Computer*, 5(1/2):109–119, March 1989.

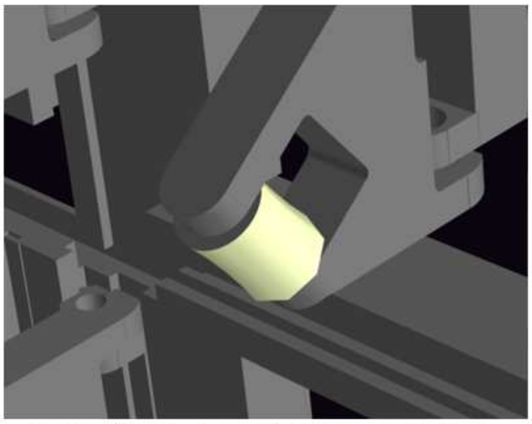
[RHD89] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. In *Proceedings of ACM Siggraph*, pages 107–117, 1989.

[Rob88] D. Roble. A load balanced parallel scan-line z-buffer algorithm for the ipsc hypercube. In *Pixim*, pages 177–192, Paris, France, 1988.

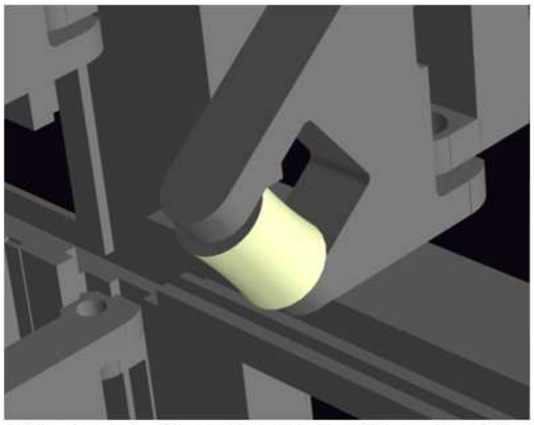
[Whi94] S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48, 1994.

[YA93] J.H. Yang and J. Anderson. Fast, scalable synchronization with minimal hardware support. In *ACM symposium on Principles of Distributed Computing*, pages 171–182, 1993.

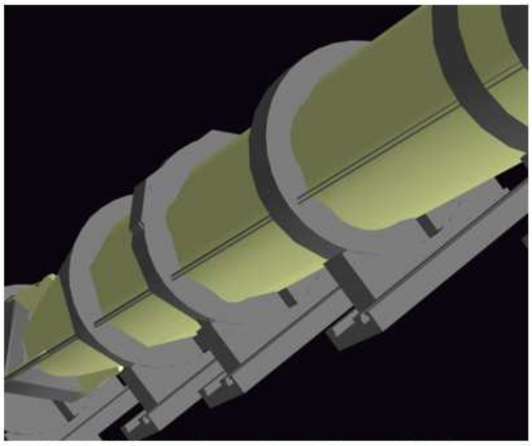
[ZKN92] Y. Zheng, D. Kerbyson, and G. Nudd. 'efficient load balancing techniques for image analysis on an m-simd machine. Technical Report CS-RR-214, Dept. of CS, University of Warwick, 1992.



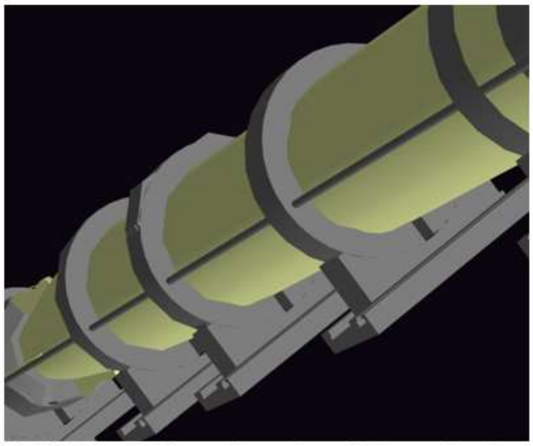
(a1) Pivot: Lagging Tessellation



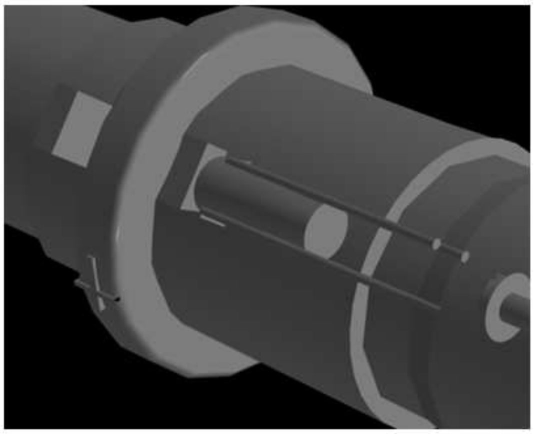
(a2) Pivot: Synchronized Tessellation



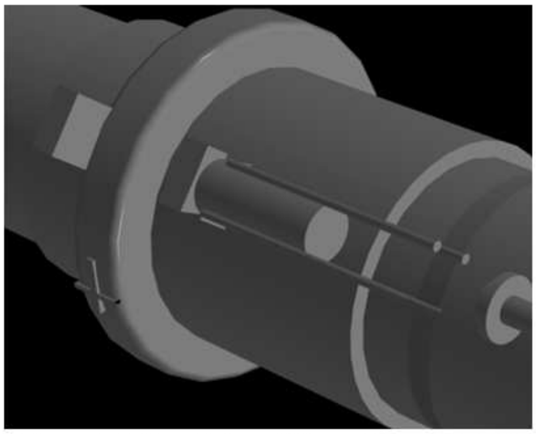
(b1) Line: Lagging Tessellation



(b2) Line: Synchronized Tessellation



(c1) Tube: Lagging Tessellation



(c2) Tube: Synchronized Tessellation

Color Plate I: Greedy Rendering