Time/Space Tradeoffs for Polygon Mesh Rendering

Reuven Bar-Yehuda Craig Gotsman
Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000
Israel

{reuven|gotsman}@cs.technion.ac.il

Abstract

We investigate architectural schemes, generalizing that of existing graphics engines, supporting fast rendering of triangle meshes. A mesh defined on n vertices is rendered by sending vertices down a graphics pipeline, after which they are pushed on a stack, to be popped when no longer needed. Only individual triangles whose vertices are present in the stack may be rendered. The storage cost of the mesh rendering is the size of the stack required to store mesh vertices during the rendering process. This may be significantly less than n. The time cost of the mesh rendering is the number of vertices sent down the graphics pipeline. If a large enough stack is available, it suffices to send each vertex once. If only a small stack is available, some vertices may have to be sent more than once, so a time/space tradeoff exists.

With our architecture, a stack of size $O(\sqrt{n})$ is sufficient to render any triangle mesh defined on n vertices, such that each vertex is sent only once through the graphics pipeline (time cost = n). We provide an algorithm that generates an appropriate "rendering sequence" of commands for any given mesh. Moreover, we show that no algorithm can do better, i.e. $\Omega(\sqrt{n})$ is a lower bound.

Some n-vertex meshes may be rendered using a stack whose size is significantly less than $O(\sqrt{n})$. An algorithm generating a minimum-time rendering sequence requiring the minimum stack size is an open question. We provide an approximation: If it is theoretically possible to render a triangle mesh in minimum time with a stack of size S, our algorithm generates a minimum-time rendering sequence requiring a stack of size no larger than $2S\log_{3/2} n$

If only a stack of size k is available, we provide an algorithm generating a rendering sequence requiring a stack of size no larger than k, such that at most $n(1+c/k)$ vertices must be sent through the pipeline, for some constant c.

Keywords: Polygons, Rendering, Graphics Pipeline.

1 Introduction

In computer graphics and geometric modeling applications, it is common to approximate freeform smooth objects as polyhedra. The polyhedra may then be manipulated and rendered (drawn) efficiently using hardware-implemented routines, incorporating visible surface determination, shading, etc. The set of triangles defining a polyhedron is commonly known as a triangle mesh, having the topology of a planar graph.

Polygon meshes, and in particular, triangle meshes, are so common that leading graphics software libraries (e.g. GL [9], OpenGL [12], IGL [3]) provide function calls dedicated to mesh rendering, and some of these are supported in the hardware of some machines (e.g. SGI, Intel i860). For example, the SGI hardware dedicates three registers to vertex storage. A triangle mesh is rendered using the GL graphics library by sending a sequence of vertices through the graphics pipeline, and a triangle is drawn automatically between every three consecutive vertices of the sequence, the three registers being used as a queue. Thus, a sequence of m vertices (perhaps with repetitions) specifies $m-2$ triangles. Since at least one vertex must be supplied to render each triangle, ideally, only one vertex should be sent through the graphics pipeline for each triangle. By Euler's theorem ([14], p. 19), the number of triangles in a mesh may reach up to twice the number of vertices, hence the GL scheme requires sending each vertex twice on the average, assuming the mesh can be specified in one sequence.

The time cost of the rendering operation is the number of vertices sent down the graphics pipeline, as these vertices require expensive geometric projection and clipping operations. However, the class of triangle meshes that may be specified in just one vertex sequence (also called "rendering sequence") without repetitions is extremely limited. These are "strip" like meshes (see Fig. 1(a)), also called *sequential* triangulations. To relax this constraint, the GL library allows the programmer to swap the contents of the two inner hardware registers at negligible cost (relative to sending a vertex through the pipeline). Interleaving `swap` commands among the vertices sent allows a larger class of *Hamiltonian* triangulations to be rendered at the cost of $m+2$ vertices for m triangles. This is precisely the family of triangle meshes in which the triangles can be covered by a single path, along which each triangle appears only once, and only passages between triangles with common edges are allowed. These sequences are called a *Hamiltonian cover* of the mesh. (see Fig. 1(b)). Unfortunately, not all triangle meshes are Hamiltonian, so an arbitrary mesh must be specified as a list of k vertex sequences, each beginning with the `begin` command and ending with the `end` command (see Fig 1(c)). The rendering time cost in this case is $m+2k$ vertices for m triangles. Minimizing k is NP-hard (by reduction to the Hamiltonian path problem), and approximating the minimum is an open question. Arkin et. al [2] show how to construct Hamiltonian triangulations on any planar point set, how to determine if a polygon has a Hamiltonian triangulation, and how to produce *sequential* covers of a given triangle mesh, within a constant factor of the optimum. Akeley et al [1] provide a greedy algorithm, based on heuristics, for the computation of a rendering sequence for a given triangle mesh. It is not clear how close to optimum the cost of the generated rendering sequence is.

In this paper we consider extending the existing hardware architectures by increasing the amount of storage dedicated to the rendering process. This has been advocated by Deering

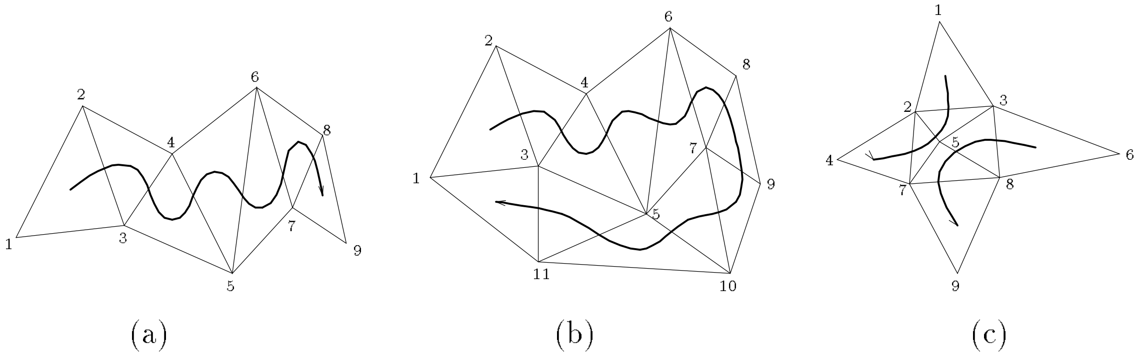


Figure 1: Rendering with 3 registers: (a) Sequential triangle mesh. The equivalent GL rendering sequence is `begin`,1,2,3,4,5,6,7,8,9,`end`. (b) Hamiltonian triangulation. The equivalent GL rendering sequence is `begin`, 1,2,3,4,5,6,7,8,`swap`,9,`swap`,10, 5,11,3,1,`end`. (c) Non-Hamiltonian triangulation requires $k = 2$ vertex sequences to cover it. The equivalent GL rendering sequence is `begin`,4,2,7,`swap`,5,`swap`,3,1,`end`,`begin` 9,7,8,`swap`,5,`swap`,3,6,`end`.

[5], whom is concerned with the "compression" of scene geometry. The extension allows one to "remember" more of the vertices which have already travelled down the graphics pipeline, reducing the number of vertices sent in order to render a triangle mesh. In particular we show that a stack of size $12.72\sqrt{n}$ suffices to render any triangle mesh defined on n vertices in minimum time n (as opposed to Hamiltonian triangle meshes, which may require up to $2n$ time), and provide an algorithm that generates the appropriate rendering sequence for a given mesh. Moreover, this bound is tight up to a constant factor, in the sense that there exist triangle meshes on n vertices that a machine with a stack of size less than $1.649\sqrt{n}$ cannot render by sending each vertex only once.

Some triangle meshes possess a topology which might allow minimum cost rendering with stack size considerably less than $O(\sqrt{n})$. We provide an algorithm that generates a rendering sequence, such that, if a stack of size S is theoretically sufficient for that mesh, the rendering sequence we generate requires a stack of size no larger than $2S \log_{3/2} n$. In this sense we approximate the best possible up to a $O(\log n)$ factor.

If we are willing to trade off stack size in favor of rendering time, we show that any triangle mesh with n vertices can be rendered in $n(1 + c/k)$ time, for some constant c, if the stack size is k, and provide an algorithm generating the rendering sequence. This is an approximation scheme enabling as close to minimum time rendering as desired, at the expense of extra storage.

2 Some Planar Graph Theory

3D triangle meshes are embedable in the plane, hence are planar graphs, in which every edge participates in a face (triangle) of the graph (mesh). When dealing with planar graphs, we can make use of the celebrated *planar separator theorem* and its variants. We say that a class of graphs has a $(g(n), \beta)$-separator for $1/2 \leq \beta < 1$, if for any graph $G(V, E)$ of n

vertices in the class, V can be partitioned into three sets U, S, W such that no edge in E joins a vertex in U with a vertex in W, $|U| \leq \beta n, |W| \leq \beta n$ and $|S| < g(n)$. In this case, we call S a $\beta - separator$ of G. For large n, this effectively means that both U and W will contain between βn and $(1-\beta)n$ vertices.

Many problems concerning graphs in the class can be solved efficiently using the divide-and-conquer method if the vertex set S is small enough and can be computed efficiently. We use the following separator theorems:

Theorem 1 *([8]) The class of planar graphs has a $(\frac{7}{3}\sqrt{n}, \frac{2}{3})$-separator that can be computed in $O(n)$ time.* ▮

The original separator theorem of Lipton and Tarjan [11] obtained $g(n) = \sqrt{8n}$, which was later improved by Djidjev [6] to $g(n) = \sqrt{6n}$, and subsequently by Gazit [8] to $g(n) = \frac{7}{3}\sqrt{n}$. These functions are the best possible up to a constant factor, as the following theorem asserts:

Theorem 2 *There exist planar graphs which do not have a $(1.649\sqrt{n}, 1/2)$-separator.* [1] ▮

Theorem 2 is obtained by considering a triangulation of the surface of a sphere. Any 1/2-separator contains the vertices along the circumference of the sphere. An identical argument was used by Djidjev [6] to obtain a lower bound of $1.55\sqrt{n}$ on the size of a 2/3-separator. The first lower bound of $\sqrt{n}$ on the 2/3-separator size was obtained in the original paper by Lipton and Tarjan [11].

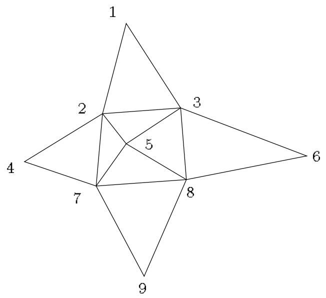
Theorem 1 implies the existence of a $O(n)$ procedure `separate`(G) which, given a planar graph G on n vertices, computes a triple $< U, S, W >$ such that S 2/3-separates G to U and W and $|S| \leq \frac{7}{3}\sqrt{n}$.

3 Minimum Time Triangle Mesh Rendering

We consider an architecture for rendering triangle meshes, where the machine is equipped with registers for storing mesh vertices in a stack mechanism. Three operations are possible: `push`(v) - send a vertex v down the graphics pipeline into the vertex stack, `draw`(i_1, i_2, i_3) - draw a triangle incident on the vertices in entries i_1, i_2, i_3 of the vertex stack, `pop` - remove the top vertex from the stack. `push`(V) for a vertex set V is shorthand for the sequence (`push`$(v) : v \in V$), and `pop`(k) shorthand for a sequence of k `pop` operations. The dominant cost is incurred by the `push` operation.

Just as the `swap` operation of the existing GL scheme is cheap (it involves sending only one bit down the pipeline), we make a similar assumption for the `pop` operation. The `draw` operation is slightly more expensive, as it would involve transmitting three integers, but no computation.

A *rendering sequence* for a mesh M is a sequence of `push`, `draw` or `pop` operations, such that after performing these operations, all triangles of M have been drawn. The *cost* of `push`(V) is $|V|$. The *time cost* of a rendering sequence is the total cost of the `push` operations in the sequence. The *storage cost* of a rendering sequence is the size of the stack needed to



`push`(3) `push`(5) `push`(7) `push`(2) `draw`(2,3,5) `draw`(2,5,7) `push`(1) `draw`(1,2,3) `pop` `push`(4) `draw`(2,4,7) `pop` `pop` `push`(8) `draw`(3,5,8) `draw`(5,8,7) `push`(6) `draw`(3,6,8) pop `push`(9) `draw`(7,8,9) `pop` `pop` `pop` `pop` `pop`

Figure 2: A minimum-time rendering sequence for a triangle mesh on 9 vertices with stack size $k = 5$ produced by `minimum_time_render`. Each vertex is pushed only once. The syntax of the `draw` operation has been slightly abused, as its arguments here are vertex labels, instead of stack indices.

implement it. The minimum time cost of rendering a triangle mesh defined on n vertices is n. See Fig. 2 for an example. We now show that with a vertex stack of size at least $12.72\sqrt{n}$, it is possible to render any triangle mesh in minimum time. The minimum time cost rendering sequence is generated by the recursive procedure `minimum_time_render`, taking advantage of the planar separator theorem (Theorem 1).

Note that `minimal_time_render` is run once offline to generate the rendering sequence, which is then stored with the mesh, and used every time the mesh is to be rendered. This justifies the $O(n \log n)$ run time of the generation algorithm, even though the rendering sequence achieves only a $O(n)$ savings in actual rendering time.

[1]The exact constant is $\sqrt{\frac{\sqrt{3}\pi}{2}}$.

```
minimum_time_render(M(V))
/* Generate minimum-time rendering sequence for mesh M on vertex set V */

{                         /* Global Stack is initialized to be empty.   */
   if (|V|!=0) {
       <U,S,W> := separate(M);
       output "push(" S ")";
       push(Stack,S);
       for all triangles t of M having one vertex in S, and the other two in Stack,
           output "draw(" index(v1(t)) index(v2(t)) index(v3(t)) ")";
       minimum_time_render(M(U));  /* M(U) is the subgraph of M induced by U */
       minimum_time_render(M(W));
       output "pop(" |S| ")";
       pop(Stack,|S|);
   }
}
```

Theorem 3 *Algorithm* `minimum_time_render` *generates a rendering sequence for $M(V)$ requiring a stack of size no larger than $12.72\sqrt{n}$. No vertex of V is pushed more than once. The algorithm runs in $O(n \log n)$ time and $O(\sqrt{n})$ space.*

Proof: The height of the stack used by the rendering sequence generated by the algorithm is the sum of the sizes of the separators computed along the worst path of the recursion tree. By Theorem 1, it is bounded by

$$\begin{aligned}
Height &\leq \frac{7}{3}\sqrt{n} + \frac{7}{3}\sqrt{2n/3} + \frac{7}{3}\sqrt{4n/9} + \cdots \\
&< \frac{7}{3}\sqrt{n}\sum_{i=1}^{\infty}(2/3)^{i/2} \\
&= \frac{7/3}{1-\sqrt{2/3}}\sqrt{n} \\
&< 12.72\sqrt{n}
\end{aligned}$$

At each recursive stage, the algorithm partitions V into three disjoint sets U, S, W. The vertices in S are pushed on the stack, and then the procedure applied on U and W, implying that no vertex of V is pushed more than once.

The computation time consumed by `minimum_time_render` results from the recursive calls to `separate` and the determination of which triangles are to be drawn at each recursive stage. The total time of the latter, over the entire algorithm, is $O(n)$, as each vertex enters the stack only once, and then all triangles incident on it are checked. Since `separate`$(M(V))$ runs in $O(|V|)$ time, the following inequality holds for the complexity $C(n)$ of the calls to `separate` in `minimum_time_render`:

$$C(n) = O(n) + C(\beta_n n) + C(\gamma_n n)$$

where $1/2 \leq \beta_n \leq 2/3, 1/2 \leq \gamma_n \leq 2/3$ for all n. This implies $C(n) = O(n \log n)$. The space consumed by `minimum_time_render` is that required for `Stack`, whose size does not exceed $O(\sqrt{n})$. ∎

We now prove that this upper bound on the memory required for minimum-time rendering is tight up to a constant factor.

Theorem 4 *Any machine with the capability of rendering all triangle meshes on n vertices in minimum time requires a stack of size at least $1.649\sqrt{n}$.*

Proof: Assume a machine capable of rendering any triangle mesh in minimum time. Given *any* triangle mesh M, the state of the machine at any stage during rendering of M may be described by the three disjoint vertex sets: U = vertices already popped from the stack, S=vertices in the stack, W = vertices not yet pushed onto the stack. Since the vertices of U will never be pushed onto the stack again (otherwise the rendering will not have minimum time), all edges incident on vertices of U must have been rendered already. On the other hand, since the vertices of W have not yet been seen by the machine, no edge incident on vertices of W has been rendered yet. This means that G does not contain any edge between a vertex of U and a vertex of W, implying that S separates G.

Stopping the rendering process after the first `push` operation where $|W| < n/2$ implies that certainly $|U| < n/2$ before that operation ($|W|$ was previously $\geq n/2$). This still holds after the `push` operation, as these vertices are then in S, therefore S 1/2-separates G at that moment. By Theorem 2, there exist triangle meshes such that the size of S must be at least $1.649\sqrt{n}$. ∎

4 Minimal Storage Polygon Mesh Rendering

In the previous section, we proved that *any* triangle mesh with n vertices may be rendered in minimum time n using a stack of size $O(\sqrt{n})$. This was due largely to the fact that a well-balanced separator of size $O(\sqrt{n})$ is guaranteed to exist for any planar graph with n vertices, and may be computed efficiently.

However, for many planar graphs, much smaller separators may exist, implying that a significantly smaller stack might suffice for *minimum* time rendering. Hence, it is advantageous to apply algorithms for computing minimum separators. A minimum β-separator of a graph G is a β-separator of G of minimum size. Denote by $Sep_{min}(G(V), \beta)$ the size of the minimum β-separator of $G(V)$. Since separator computations will typically be done offline, in a preprocessing step, the complexity of the separator-computing algorithm is not critical, as long as it is polynomial. Computing the minimum β-separator of a n-vertex planar graph for any constant $1/2 \leq \beta < 1$ is believed to be NP-hard [13], so a polynomial solution currently is only approximate, i.e. given a planar graph $G(V)$ and $1/2 \leq \beta < 1$, a separator of size no larger than $K(n)Sep_{min}(G(V), \beta)$ is computed, where $K(n)$ is an approximation factor. In a series of works, approximations to the optimum separator problem have been proposed for $K(n) = O(\log n)$ [15], $K(n) = O(1)$ [4], and $K(n) = 2$ [7]. We call the algorithm of [7] `approximate_optimum_separate`.

Denote by $Stack_{min}(G(V))$ the size of the minimum stack theoretically required to render $G(V)$ in minimum time. We are not able to present an algorithm that produces a minimum-time rendering sequence for a planar graph using a stack of size $Stack_{min}(G(V))$, but we are able to provide an approximation:

Theorem 5 *Applying algorithm `minimum_time_render` to a n-vertex mesh M with calls to `approximate_optimum_separate` instead of `separate` generates a rendering sequence requiring a stack of size no larger than $2Stack_{min}(M)\log_{3/2} n$.*

First, the following Lemma establishes a relation between the quantities Sep_{min} and $Stack_{min}$.

Lemma 1 *Let $G(V)$ be a planar graph. For any subgraph $G(V')$ of G induced by $V' \subset V$, and for any $1/2 \leq \beta < 1$, $Sep_{min}(G(V'), \beta) \leq Stack_{min}(G(V))$.*

Proof of Lemma: As already observed in the proof of Theorem 4, the contents of the stack of the rendering machine at any time during the rendering process defines a separator of $G(V)$, and, therefore, also of $G(V')$.

Consider a minimum-stack rendering sequence of $G(V)$ (which requires a stack of size $Stack_{min}(G(V))$). For any $1/2 \leq \beta < 1$, it is possible to find a point in time during the rendering, in which S', the subset of V' in the stack, β-separates $G(V')$. By definition, the size of the entire stack at that moment is no larger than $Stack_{min}(G(V))$. S' is a subset of the vertices in the stack, hence $|S'| \leq Stack_{min}(G(V))$. The fact that S' is a β-separator of $G(V')$ implies that $Sep_{min}(G(V'), \beta) \leq |S'|$, hence $Sep_{min}(G(V'), \beta) \leq Stack_{min}(G(V))$. ∎

Proof of Theorem: The proof of this theorem proceeds by an argument similar to that applied by Leighton and Rao [10] for their approximation scheme to the minimum cut linear arrangement problem on general graphs.

The stack size required by the generated rendering sequence is the sum of the separator sizes computed along the worst path of the recursion tree of algorithm `minimum_time_render`. Calling `approximate_optimum_separate` guarantees that the size of each such separator is no larger than $2Sep_{min}(G(V_i), \beta_i)$, for the appropriate $V_i \subset V$ and β_i, therefore, by Lemma 1, not more than $2Stack_{min}(G(V))$. The depth of the recursion tree is no more than $\log_{3/2} n$, so the total stack size is no larger than $2Stack_{min} \log_{3/2} n$. ∎

We note that $Sep_{min}(G(V), \beta)$ for many triangle meshes on n vertices is close to the upper bound $O(\sqrt{n})$, so computing a minimum β-separator might not be a major improvement on the β-separator computed by the algorithm of Gazit [8].

5 Time/Space Tradeoffs

It is possible to use the same basic technique we used to generate a minimum-time rendering sequence for the entire mesh, for much smaller pieces. This facilitates a much smaller vertex stack size, but increases the rendering time cost, as some vertices must be pushed more than once. To establish this, we need a refined version of the basic separator Theorem 1, dealing with separations to less balanced parts:

Theorem 6 *For any $f(n) = O(n)$ and any planar graph $G(V)$ with n vertices, it is possible to compute in $O(n)$ time a subset $S \subset V$, such that S separates V into U and W, $|S| \leq 11.85\sqrt{f(n)}$ and $f(n)/3 \leq |U| \leq 2f(n)/3$.*

Proof: Denote $\alpha = 11.85$. Given a planar graph $G(V)$, apply Theorem 1 recursively. At each recursion level, a vertex set is separated into two sets, of which one is chosen for further separation. This continues until a set of size less than $2f(n)/3$, but larger than $f(n)/3$, is reached. We prove inductively that the theorem holds at each recursion level.

The induction basis, for which $f(n) = n$, is implied by Theorem 1. Now assume that the induction hypothesis holds for a m-vertex planar graph $G(V)$, i.e. the size of the separator S which yielded this set is less than $\alpha\sqrt{m}$. Now separate $G(V)$, containing m vertices, with a separator S' of size $\frac{7}{3}\sqrt{m}$, to two vertex sets V_1 and V_2 of size m_1 and m_2, respectively. Assume $|S \cap V_1| = \alpha_1\sqrt{m}$ and $|S \cap V_2| = (\alpha - \alpha_1)\sqrt{m}$.

Now $S_1 = S' \cup (S \cap V_1)$ is a separator of $G(V_1)$, and $S_2 = S' \cup (S \cap V_2)$ is a separator of $G(V_2)$. If, contrapositively, both $|S_1| > \alpha\sqrt{m_1}$ and $|S_2| > \alpha\sqrt{m_2}$, then $(\alpha_1 + 7/3)\sqrt{m} > |S_1| > \alpha\sqrt{m_1}$, and similarly $(\alpha - \alpha_1 + 7/3)\sqrt{m} > \alpha\sqrt{m_2}$, so $(\alpha + 14/3)\sqrt{m} > \alpha(\sqrt{m_1} + \sqrt{m_2})$. Now since $m_1 + m_2 = m$ and $m/3 \leq m_1, m_2 \leq 2m/3$, this implies $\sqrt{m_1} + \sqrt{m_2} > \frac{1+\sqrt{2}}{\sqrt{3}}\sqrt{m}$, in turn implying $\alpha + 14/3 > \frac{1+\sqrt{2}}{\sqrt{3}}\alpha$, contradicting the fact that $\alpha = 11.85$.

Therefore, without loss of generality, we may assume that $|S_1| < \alpha\sqrt{m_1}$ and proceed recursively to separate V_1, until a component of the required size is obtained.

$C(n)$, the complexity of this procedure, satisfies

$$C(n) \leq O(n) + C(2n/3)$$

implying $C(n) = O(n)$. ∎

Theorem 6 implies the existence of a $O(n)$ procedure `separate`(G, k), which, given a planar graph $G(V)$ on n vertices, computes a triple $< U, S, W >$ such that S separates G to U and W, $(W = V - U - S)$, $k/3 \leq |U| \leq 2k/3$ and $|S| < 11.85\sqrt{k}$.

Given a triangle mesh M, using Theorem 6, we separate a small submesh M' from M, use `minimum_time_render` to generate a rendering sequence for the triangles defined on M' (including the separator), and then discard M'. The process is continued with the remainder of the mesh (again including the separator).

Assume the vertex stack size is k. The following algorithm generates a rendering sequence for a triangle mesh under this constraint. d is a sufficiently large constant, whose exact value may be determined later.

```
render(M(V))    /* generate rendering sequence for mesh M on vertex set V */

{
   while (|V|!=0) {
      <U,S,W> := separate(M(V),d*k*k)
      minimum_time_render(M(U+S));     /* U+S is the union of U and S */
      V := W+S;
   }
}
```

Theorem 7 *Algorithm* `render` *generates a rendering sequence requiring a stack of size no larger than k. The rendering sequence pushes no more than $n(1 + c/k)$ vertices, for some constant c. The algorithm runs in $O(n^2/k^2)$ time.*

Proof: At each iteration of the loop, `minimum_time_render` pushes the vertices of `S` once in order to render $M(U \cup S)$, but these are pushed again in the rendering of $M(W \cup S)$. The number of iterations is $O(n/k^2)$. By Theorem 6, at each iteration $|S| = O(k)$, so the number of vertices pushed at that iteration is $|U \cup S| \leq |U| + |S| = |U| + O(k)$. The total number of vertices pushed during `render` is therefore $= n + O(n/k^2)O(k) \leq n(1 + c/k)$, for some constant c. `render` does not require a stack of size more than k since this holds for `minimum_time_render`. Each iteration runs in $O(n)$ time, so the total run time is $O(n^2/k^2)$. ∎

6 Conclusion and Discussion

We have explored the advantages of extending the architecture of contemporary graphics engines to larger vertex stores, in order to render triangle meshes more efficiently. We have shown that any n-vertex mesh may be rendered in minimum time with storage cost $\theta(\sqrt{n})$. We have also optimized this, and shown how to gracefully trade off memory for time.

The bounds and algorithms presented here are valid for triangle meshes, by far the most common in computer graphics applications. For meshes containing polygons with more than three vertices, our rendering model requires that all the vertices be present in the stack when the polygon is to be rendered. This is not guaranteed by our algorithms in their current form. However, they may be modified easily so that entire polygons are pushed or popped from the stack (instead of individual vertices), at the expense of some more time and space complexity.

Our rendering model assumes that the dominant time cost of rendering a triangle is incurred at the geometric stage of the rendering pipeline. For some machines, this is not the case. The bottleneck of the rendering pipeline might be elsewhere, e.g. in the triangle scan conversion. In this case the projected area of the triangles is important, and larger triangles are to be considered more complex than smaller ones.

Acknowledgements

We thank Noga Alon, Estie Arkin, Guy Even, Joe Mitchell, Seffi Naor, Satish Rao and an anonymous referee for helpful discussions and comments on the topic of this paper.

This research was supported by the Fund for the Promotion of Research at the Technion, and the Technion V.P.R. Fund - Promotion of Sponsored Research.

References

[1] K. Akeley, P. Haeberli, and D. Burns. C program on SGI machines in /usr/people/4Dgifts/iristools/libgutil/tomesh.c. 1990.

[2] E.M. Arkin, M. Held, J.S.B. Mitchell, and S.S. Skiena. Hamiltonian triangulations for fast rendering. In J. van Leeuwen, editor, *Algorithms – ESA '94*, LNCS 855, pages 36–47, Utrecht, NL, September 1994.

[3] R. Cassidy, E. Gregg, R. Reeves, and J. Turmelle. *IGL: The Graphics Library for the i860.* 1991.

[4] F.R.K. Chung and S.T. Yau. A near optimal algorithm for edge separators. In *26th Annual ACM Symposium on the Theory of Computing.* ACM, 1994.

[5] M. Deering. Geometry compression. *Computer Graphics (Proceedings of SIGGRAPH)*, 29:13–20, 1995.

[6] H.N. Djidjev. On the problem of partitioning planar graphs. *SIAM J. Alg. Disc. Methods*, 3(2):229–240, 1982.

[7] N. Garg, H. Saran, and V.V. Vazirani. Finding separator cuts in planar graphs within twice the optimal. In *35th Annual Symposium on Foundations of Computer Science*, pages 14–23. IEEE, 1994.

[8] H. Gazit. An algorithm for finding a $\frac{7}{3}\sqrt{n}$ separator in planar graphs. *Preprint*, 1994.

[9] Silicon Graphics Inc. *GL Programming Guide.* 1991.

[10] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 256–269. IEEE, 1988.

[11] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36(2):177–189, 1979.

[12] J. Neider, T. Davis, and W. Mason. *OpenGL Programming Guide.* Addison-Wesley, 1994.

[13] J.K. Park and C.A. Phillips. Finding minimum-quotient cuts in planar graphs. In *25th Annual ACM Symposium on the Theory of Computing*, pages 766–775. ACM, 1993.

[14] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[15] S.B. Rao. Faster algorithms for finding small edge cuts in planar graphs. In *24th Annual ACM Symposium on the Theory of Computing*, pages 225–237. ACM, 1992.