sLIB: A Comprehensive Renderer for Parametric Surface-Models

Subodh Kumar

Johns Hopkins University
subodh@cs.jhu.edu

Abstract

We present sLIB: a library for interactive rendering of models with trimmed NURBS surfaces. sLIB provides users with the ability to configure the rendering pipeline based on the available hardware, i.e, the number of processors and the quality of the graphics board. sLIB includes a suite of algorithms, some of which have been presented before. It is based on dynamic triangulation of groups of surface primitives. The five major components of sLIB are surface pre-processing, pipeline management, visibility computation, tessellation, and triangle rendering. The pipeline manager monitors the performance of the CPUs and the graphics board to tune the system for performance improvement. We have successfully used sLIB to obtain faster rendering rates than has been achieved before. sLIB scales well with both the hardware and the data size. In addition, sLIB provides a flexible framework for designing and testing surface rendering algorithms.

1 Introduction

Non-Uniform Rational B-Spline (NURBS) surfaces are routinely used to design geometric models like automobiles, submarines, airplanes, etc. Often these surfaces are approximated by triangles for efficient display on current graphics systems. A one-time *static* tessellation of many real-world models usually requires the generation of millions of triangles [Bro96]. Most Current graphics systems are not able to render these many triangles at interactive speeds, i.e., more than 10 times a second. A number of surface simplification schemes have recently been proposed [CVMe96, GH97, Hop97] that speed up polygonal rendering. However, they often require management of large amounts of data and are still unable to generate more detail than the initial tessellation. Moreover, it seems unnatural that in order to render surfaces, we first generate a large number of triangles and then find a way to reduce the count. On the other hand, efficient techniques for *dynamic* surface tessellation [RHD89, AES91, KML96, KMZH97] have made it possible to approximate surfaces with a small number of triangles in the first place. Furthermore, the surface representation provides much finer control on the update of detail as compared to purely polygonal approaches [XV96, Hop97]. The dynamic tessellation approach also promises to be more scalable and efficient, in general. While the model sizes will continue to grow, the representation for the model-subset that must be processed every frame need not grow linearly. In other words, more information may be compressed in a small number of higher-order primitives like surfaces and super-surfaces [KMZH97].

We present sLIB , a library for efficient rendering of parametric surfaces. We will often refer to sLIB as a system, since it incorporates all components of a surface rendering system except the user interface. It may be used to read a list of tensor-product or triangular NURBS or Bézier surfaces and render them. Our internal representation is based on the Bézier form. Some of the components used by sLIB have been presented before, others are introduced here for the first time. More importantly, we present a comprehensive system which seamlessly integrates the different algorithms to obtain rendering speeds not obtained before. In addition, sLIB performs run-time performance monitoring to dynamically re-arrange the surface rendering pipeline. This modular framework also allows us to experiment with a number of competing schemes by simply plugging them in or out. Each module, M_a, of sLIB is designed such that if it needs the results of another module, M_b, the appropriate data-structures of M_b are accessible through a generic field in the common parameter, `pglob`. Each module accepts `pglob` as its input parameter. When a new module registers itself to the system, it is allocated a pointer field within `pglob`. M_b can then set the pointer to the data it needs to export. If M_b is disabled (i.e., not registered), M_a can be configured to disable itself automatically, enable M_b via a call-back, or do without the results of M_b. Of course, it is the responsibility of M_a to produce meaningful results if it decides to proceed without the results of M_b. For example, the tessellation module may generate triangles for hidden patches, if it does not have the results of the visibility computation module. It is entirely possible that the rendering module is directly hooked to the visibility module and it discards all triangles of a hidden patch. sLIB also has inbuilt support for multi-processors systems and manages load distribution. We use the lock-free synchronization scheme of [KCM96] to reduce the number of system calls needed for mutual exclusion by appropriately ordering our read-write operations. sLIB consists of five main components:

Surface preprocessing: As a part of preprocessing, a hierarchical representation of the model is generated, adjacencies between surfaces are computed, and surface properties like maximum magnitudes of derivatives and bounds on Gauss map are generated. NURBS representation is also converted to Bézier at this stage. Complex Bézier surfaces are further subdivided into simpler parts. Adjacent Bézier patches are hierarchically combined into *super-surfaces* (further details in Section 2).

Pipeline management: The pipeline manager is like a job-scheduler, except jobs are not separate processes — they are just procedures that cooperatively relinquish control to the manager by returning or directly to other jobs by callbacks. The default initial order of jobs may be changed via calls to library routines. This order dynamically changes at rendering time when a system bottleneck is discovered. In order to keep the system stable, re-configurations are not performed unless a bottleneck persists. Of course, the application program may disable the dynamic re-configurations altogether.

Visibility computation: sLIB determines the visibility of *primitives* (i.e., Bézier patches or super-surfaces). A visibility primitive is the smallest entity which may be deemed by sLIB to be visible or invisible. A visibility primitive is a node in the hierarchical representation of the model. Not all primitives need be at the same level for a frame. For example, one primitive could be a single Bézier patch, while another is a super-surface (a collection of patches). sLIB includes a variety of view-frustum and back-patch [KM96] culling routines. Different types of bounding box hierarchies are available and one (or more) of them may be activated.

Additionally, the application program may plug in its own visibility processor.

Tessellation: The tessellator module triangulates one primitive at a time. sLIB tessellates a primitive based on a number of different quality constraints (see Section 3), one of more of which may be enabled. Using the current viewing parameters and the stored surface properties (generated by the preprocessor), sLIB computes the sampling size required for each primitive. These samples are subsequently triangulated in a crack-free manner. sLIB directly generates triangle strips, using an incremental algorithm that exploits coherence.

Triangle rendering: The Triangle rendering component is our interface to OpenGL. It passes triangle-strips to the graphics board. (Our current implementation assumes OpenGL, but it can easily be modified for other graphics standards.)

Any of these components may be turned on or off and the rest of the system still maintains the expected behavior. We found this feature to be quite useful for profiling. Our system renders about $200,000$ patches per second on an Onyx2 with 4 processors and InfiniteReality graphics. It out-performs simple triangle based rendering on a variety of Silicon Graphics O2, Extreme, Octane, and Onyx2 systems as well. See section 5 for more details on our experiments.

1.1 Mathematical Background

A rational tensor-product Bézier patch, $\mathbf{F}(u,v)$, of degree $m \times n$, defined for $(u,v) \in [0,1] \times [0,1]$, is specified by a mesh of control points, $\mathbf{p}_{ij}$, and their weights, w_{ij}, $0 \le i \le m, 0 \le j \le n$:

$$\mathbf{F}(u,v) = \frac{\displaystyle\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\mathbf{p}_{ij}\mathcal{B}_i^m(u)\mathcal{B}_j^n(v)}{\displaystyle\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\mathcal{B}_i^m(u)\mathcal{B}_j^n(v)}$$

where the Bernstein function $\mathcal{B}_i^n(t) = \begin{pmatrix} n \\ i \end{pmatrix} t^i(1-t)^{n-i}$ Additionally, a closed loop of trimming curves are specified on the domain of trimmed Bézier surfaces. For counter-clockwise loops we display only the part of the surface enclosed by the trimming curve and for clockwise loops we display only the part not enclosed (see Fig.1).

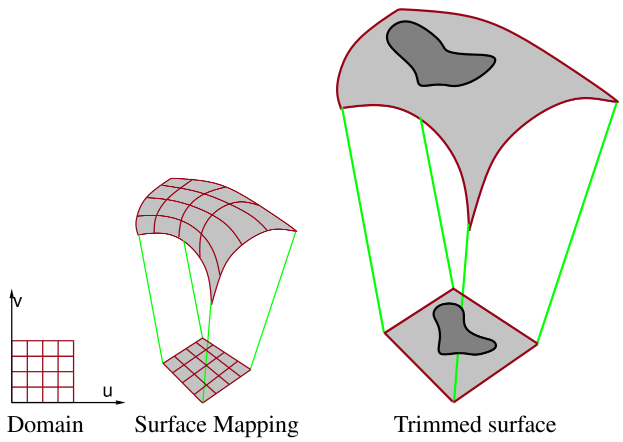


Figure 1: Tensor Product Parametric Surface

The normal direction is given by $\mathbf{N}(u,v) = \mathbf{F}_u \times \mathbf{F}_v$ where $\mathbf{F}_u$ and $\mathbf{F}_v$ are the partial derivatives.

1.2 Previous work

Curved surface rendering has been an active area of research for more than two decades. Ray tracing, pixel level subdivision and scan-line based algorithms have been used in the past to render surfaces. However, none of these are quite efficient on current graphics systems. In order to exploit fast triangle rendering hardware, recent research has focussed on generating polygonal approximations of surfaces [RHD89, AES91, LC93, PR95, KML96]. A combination of on-line tessellation (for high-detail objects) and off-line tessellation combined with multiple levels of detail (for low-detail objects) has also been used [KMZH97]. We borrow the terminology of *super-surfaces* from [KMZH97], but our notion of super-surface is different – it represents a dynamically chosen collection of patches, rather than a static cluster. In addition, we tessellate super-surfaces at rendering time and treat them as first class primitives, thus obtaining a finer control on the triangulation generated each frame. In addition to reducing the complexity of the crack-prevention algorithm, our scheme is based on the parametric domain and hence allows us to easily build a hierarchy of super-surfaces. As a result, we are able to perform simplification of a small number of surfaces directly, as opposed to that of a potentially large number of polygons.

1.3 Main contributions

In the experience of the Kumar et al. [KML96, KMZH97], their prototype implementation was not easily extensible. Separate copies of the code were made for testing specific algorithms, with each copy taking on its own path. The result was a tree of separate codes. For a useful testbed, one requires integration of all components into one system with the code configured to take advantage of all optimizations together. This motivated us to use a design that allows easy plug-in of new services. This design also allows multiple competing routines to co-exist. The system can be configured to use one of these via function pointers. Kumar et al. report that the most expensive operation in their system is testing if the tessellation of a surface patch needs to change from the a frame to the next. We improve the performance of this test by using what we call the *fractional condition* framework for visibility testing and tessellation testing. Instead of executing the loop:

```
for each patch:  compute new visibility
(tessellation step size, resp.),
```

we test

```
Has the view-point moved 'enough' for a
potential change in visibility (tessellation
step size, resp.)?
```

If the view point does change sufficiently, we recompute the new visibility or step size and recompute the new value of 'enough'. Using this framework, we reduce both the time for testing if a primitive is visible, and the time for testing if a primitive needs re-tessellation. Other important contributions of this paper include:

- Dynamic super-surface construction, testing and tessellation
- Construction of a hierarchical surface representation
- Curve simplification if trimming curves are specified in the piece-wise linear form

1.4 Organization

In the rest of this paper, a basic familiarity with NURBS and Bézier surfaces is assumed. We start with the details of super-surface construction and the hierarchical model representation in Section 2. We present a brief overview of the tessellation and visibility algorithms by [KML96, KM96] in Section 3 and describe our improvements. Section 4 describes the pipeline management. Section 5 discusses our system implementation and analyzes its performance. Finally, conclusions are presented in Section 6.

2 Model Representation

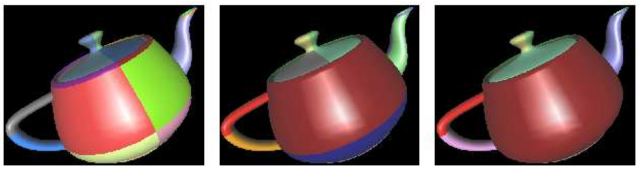
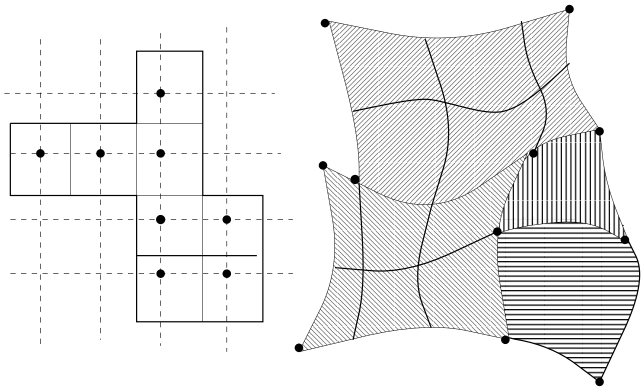


Figure 2: Super-surface hierarchy – three levels of the hierarchy of the Utah teapot are represented here. Each super-surface is colored in a single color.

We assume that the input is specified as an unordered list of triangular Bézier or NURBS surface patches (as in an iges [NCG91] file). Optionally, each surface patch may include trimming curves specified in Bézier, NURBS, or piece-wise linear forms. While representing smooth curves as polygonal chains is contrary to our design philosophy, many real-world models do maintain a dense sampling of the trimming curves. For example, constructive solid geometry modelers generate curves of intersection as the trimming curves. It is often not possible to find an exact rational parametrization of these curves in the domains of the corresponding intersecting surfaces. (Currently, sLIB does not include any facility for constructing approximate Bézier curves from the piece-wise linear representation.)

We subdivide NURBS curves and surfaces into Bézier form using knot insertion [Far93]. Additionally, each Bézier trimming curve is also subdivided at patch boundaries.



(a) (b)

Figure 3: Super-surface Construction. (a) shows a uniform tessellation (dashed lines) of a single super-surface in its domain. Domains of its constituent patches are marked with solid lines. (b) shows four adjacent super-surfaces and their constituent patches. Special points are marked in bold.

For each Bézier surface we pre-compute the bounding boxes of its Gauss maps and the maximum magnitudes of its partial derivatives. We refer to such object-space data merely as surface *properties*. We will later detail the specific properties we precompute when we describe how those properties are used. We refer to the volume of the bounding box (e.g., of a pseudo-Gauss map) as its *magnitude*. We subdivide [Far93] each *complex* Bézier surface. Complexity of a Bézier patch, $\mathbf{F}$, is defined as follows:

- Subdivide $\mathbf{F}$ into $\mathbf{F}_1$ and $\mathbf{F}_2$, at u (or v) $= \frac{1}{2}$.
- If $\frac{1}{2} < |\frac{\mathcal{P}(\mathbf{F})-\mathcal{P}(\mathbf{F}_i)}{\mathcal{P}(\mathbf{F})}| < 2$, output $\mathbf{F}$, otherwise $\mathbf{F}$ is complex; output $\mathbf{F}_1$ and $\mathbf{F}_2$. $\mathcal{P}$ denotes a property, $i \in \{1, 2\}$.

Since we perform uniform subdivision of domain, subdivision of complex patches ensures that the simpler half is not over-tessellated to satisfy quality constraints on the complicated half.

Before we construct the surface hierarchy we derive adjacencies between surface patches. We do this using an extension of the scheme proposed in [BDK98]. If boundary control polygons p_1 and p_2, respectively, of patches P_1 and P_2 match exactly, we draw an edge between P_1 and P_2 in the adjacency graph. If curves do not match exactly, we use the adjacency score [BDK98] (based on relative distances to neighboring boundary curves) to assign adjacencies.

The surface hierarchy is constructed in a bottom-up fashion. At each level, i, we use a technique similar to [KMZH97] to construct $\{\text{super-surface}(i)\}$: we grow super-surfaces$(i-1)$ into larger super-surfaces(i). We continue to merge adjacent super-surfaces of level -1 until the number of super-surfaces reduces approximately by a factor of log, i.e.,

$$|\{\text{super-surface}(i)\}| \leq \frac{|\{\text{super-surface}(i-1)\}|}{U_k \log |\{\text{super-surface}(i-1)\}|},$$

where U_k is a small constant like 2. Mergers between candidate pairs of super-surfaces proceed in the order of increasing relative difference in the magnitude of their properties. In practice, we construct 5-8 levels for the best performance. More levels only add extra overhead. At rendering time, this hierarchy is traversed and based on the viewing parameter a super-surface is either further expanded, or is output as a primitive and is subsequently tessellated and rendered.

The properties of a super-surface are obtained by a union of the properties of the component surfaces. Note that when super-surfaces are merged, their trimming curves with overlapping edges are also merged. Since not all super-surface are tessellated at the same level, it is important to maintain a valid representation of the boundary curves between super-surfaces. In particular, there are special points on the boundary curves of a super-surface that must be included in any sampling: points adjacent to more than two super-surfaces are special points. These are exactly the domain corners of super-surfaces as marked in Fig. 3(b).

3 Surface Processing

This section presents our algorithmic improvements to [KML96, KMZH97]. Our system is, in part, based on [KML96, KMZH97], hence we start with its brief overview:

3.1 Previous algorithm

[KMZH97] presents a technique to combine adjacent Bézier patches into super-surfaces. They triangulate each super-surface and generate various LODs [CVMe96] of the triangulation. At rendering time, the appropriate level is displayed for each super-surface based on its distance from the viewer. If the highest level of detail of a super-surface is insufficient, each Bézier patch of the super-surface is triangulated separately. Extra processing is required to prevent cracks between super-surfaces at different levels of detail.

Incremental tessellation and rendering of Bézier patches is discussed in [KML96]. In brief, they

1. Perform view-frustum and back-face culling. Using the bounding volume of a Bézier patch, **F**, [KML96] computes the back-space, an open cone S_B (see Fig. 4). If $\mathbf{N} = \mathbf{F}_u \times \mathbf{F}_v$ is contained in S_B, **F** is back-facing. Note that it uses bounding volumes for both **F** and **N**. Later in this section, we show how we reduce it to just one bound and improving the accuracy of the test.

2. Test the sampling required to satisfy one or more of the screen-space quality constraints: bounded triangle size, bounded normal deviation, bounded surface deviation, and bounded normal difference. The number of samples needed is computed as $s_F(B_F)\mathcal{P}(\mathbf{F})$, where s_F is the scaling factor for the bounding box of **F**and $\mathcal{P}(\mathbf{F})$ is a pre-computed property of the surface. The scaling factor is the maximum ratio between any unit vector in B_F, and its projection on-screen. [KMZH97] notes that a profile of the code shows that up to 30% of the CPU time is spent in computing s_F at rendering time. We show later, how to reduce this time.

3. If a sampling different from the previous frame is required, re-sample the interior of the patch and the boundary curves and incrementally re-triangulate the new set of samples.

3.2 Back-patch computation

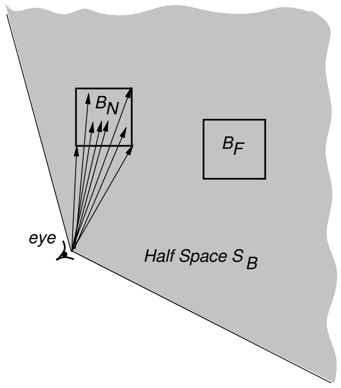


Figure 4: Back-Patch Computation: B_F the bounding volume of patch **F** defines the half-space S_B. If **N** lies entirely in S_B all points on **F**are back-facing.

Let **F**be a rational Bézier patch (we drop parameters (u, v), when they are implicit from context.) **F**is back-facing if $\mathbf{N} \cdot (\mathbf{F} - \mathbf{E}) > 0$ where $\mathbf{N} = \mathbf{F}_u \times \mathbf{F}_v$. for all $(u, v) \in [0, 1] \times [0, 1]$, where **E** is the eye point. The visibility test may also be written as $\mathbf{N} \cdot \mathbf{F} > \mathbf{N} \cdot \mathbf{E}$. m_F, the minimum value of $\mathbf{N}.\mathbf{F}$, is a pre-computed property of the surface.

If **F** is an $m \times n$ rational Bézier patch, **N** is a $3m \times 3n$ rational Bézier patch while $\mathbf{N} \cdot \mathbf{F}$ is a $3m - 1 \times 3n - 1$ rational Bézier patch. At rendering time, we check if $\mathbf{N} \cdot \mathbf{E}$ is greater than m_F. This test can be positive if $\mathbf{p}_{ij} \cdot \mathbf{E}$ is greater than m_F, where $\mathbf{p}_{ij}$ are the control points of **N**. In practice, we just test against the bounding box of the control points.

3.3 Super-surfaces

Note that both the tessellation and visibility tests, as devised for Bézier patches, are applicable directly to super-surfaces. In particular, m_P for any super-surface, P, is the minima of m_F of all its constituent patches. Similarly, the scale factor of a super-surface is the maxima of scale-factors of all constituent patches. The bounding volume of **N** is the union of the bounding volumes of constituent patches. Thus we do not need to compute an analytical representation for a super-surface.

To perform an $n_u \times n_v$ tessellation of a super-surface, P, we simply consider the union of the domains of all its constituent patches, arranged in in an arbitrary plane as shown in Fig. 3(a). Each sample point of P is contained in exactly one of the constituent domains. We use the representation of the corresponding patch to raise the sample to $\mathbb{R}^3$. Again, boundary curves are tessellated separately. Triangulation between the interior of a super-surface and the boundary curves continues to be a planar triangulation just as in the simple Bézier patch case. The rest of the algorithm remains the same.

3.4 Fractional condition

As mentioned earlier, two of the most important operations in surface rendering are tests:

1. WHETHER a primitive, P, is visible
2. WHETHER the current sampling of P is sufficient

We introduce a significant speed-up by changing the 'whether' to 'how much'. Instead of boolean answers, we use a fractional response. The idea is quite simple: we express *visibility-factor* and *sampling-factor* as real numbers. A high magnitude of the visibility-factor of a primitive implies it is highly invisible and need not be re-tested for small changes to the w-point. Similarly, a high magnitude of sampling-factor implies that primitive need not be re-tested for small changes in view-point. Visibility-factor, d_v, of a primitive, P, is defined as the minimum distance the eye-point must move until visibility of P needs to be re-computed. If the eye-point moves less than $d_v(P)$, P continues to be visible or hidden. Sampling-factor, d_s is defined similarly.

Here, we outline the processing of only fractional back-patch visibility. Other fractional conditions are computed analogously.

Suppose we compute $v_i = \mathbf{N} \cdot \mathbf{E}_i$ for frame i for a given patch, **F**. **F** is backfacing if $v_i < m_F$. Let $\mathbf{E}_{i+1}$ be the eye-point during frame $i + 1$.

$v_{i+1} = \mathbf{N}.\mathbf{E}_{i+1} < m_F \iff \mathbf{N} \cdot (\mathbf{E}_{i+1} - \mathbf{E}_i) < m_F - \mathbf{N} \cdot \mathbf{E}_i$

Thus, while the maximum value of $\mathbf{N} \cdot (\mathbf{E}_j - \mathbf{E}_i), j > i$, remains less than $m_F - \mathbf{N} \cdot \mathbf{E}_i$, the patch remains hidden and visibility need not be explicitly re-computed. The maximum value of $\mathbf{N} \cdot (\mathbf{E}_i - \mathbf{E}_{i+1})$ is $|\mathbf{N}||\mathbf{E}_{i+1} - \mathbf{E}_i|$. This implies that re-testing is not required as long as the movement of eye point (i.e. $|\mathbf{E}_{i+1} - \mathbf{E}_i|$) is less than $d_v = \frac{m_F - v_i}{|\mathbf{N}|}$. Note that m_F and the maximum value of $|\mathbf{N}|$ are pre-computed properties of primitives.

3.5 Trimming curve processing

[KMZH97] performs a sampling of all trimming curves to obtain a piecewise linear representation. However, if the trimming curves are already specified in a highly tessellated form, it is difficult to reduce the number of points on the trimming curves without introducing cracks in the triangulation. Since we compute adjacencies, we can simplify the common boundary curve separately and use the simplified version on both adjacent patches, thus avoiding cracks. We perform this simplification by deleting points on the curve as long as each resulting curve segment is less than Δ in length, where Δ [KML96] is a user controlled bound on the maximum triangle-edge length. A problem with this approach is that resulting trimming polygon may be self-intersecting. In order to avoid this, we select *special* points on the curve that cannot be deleted. The special points of a trimming curve on a patch domain are selected as follows:

1. Choose the point $\mathbf{p}_m$ with the minimum internal angle
2. Mark $\mathbf{p}_m$ as `special` and add it to `chain`
3. Add point $\mathbf{p}_{m+i}$ to `chain` (Addition is modulo n.)
4. If a segment of C intersects the Convex Hull of `chain`, $\mathbf{p}_{m+i-1}$ is special. Set $m := m + i$, and proceed recursively.

A point on the common boundary curve is marked as special if it is found to be special in either of the two adjacent surface domains. It can be proven that if special points are included, the resulting trimming polygon is guaranteed to be simple.

4 Pipeline management

The sLIB pipeline has four major stages. Since this is not a static sequential pipeline, we call them *threads*. The four threads are surface-tree construction ($\mathcal{C}$), visibility computation ($\mathcal{V}$), surface tessellation ($\mathcal{T}$), and rendering ($\mathcal{R}$). The threads can interact with each other through in-memory data structures, or through files. The file interface can easily be extended to use inter-process communication or unix sockets. The interactions may be synchronous or asynchronous. In the rest of this section, we primarily discuss the $\mathcal{V}$, $\mathcal{T}$, and $\mathcal{R}$ threads, since the $\mathcal{C}$ thread runs just once and it cannot be re-configured in our current scheme: it must be completed before other threads start. Other threads repeat every frame.

4.1 Threads

Each thread is designed around traversing an associated data-structure. During one execution we *instance* a thread once. Different instances of a thread may be used to compare competing schemes. Similarly the data structure traversed by a thread can have several instances. Each corresponding `traverse` method can output a `primitive` that the thread can decode using its `header`. The format for `primitive` is extensible. Each thread also provides call-back hooks: routines that are called just before or just after it processes a primitive. There also are hooks for two call-backs, respectively, as the first and the last step of each frame of the thread. Similarly, each data-structure provides a call-back hook at each primitive. We also define the concept of *autonomy*. The pipeline manager only schedules autonomous entities for execution. Other routines required for rendering must be called via call-backs by autonomous entities. Any thread may be marked as autonomous. The manager does not pre-empt any autonomous entity. We let the application program controls the preemption by allowing an entity to explicitly relinquish control by returning. Before returning, a thread saves its state and sets its `active` flag to request re-scheduling. The frame loop of pipe-manager is as follows:
Repeat for each frame

- Set all `autonomous` routine `active`
- While $\exists$ an `active autonomous` routine `R`, call `R`
- if `Tuning Enabled`, Fine Tune Pipeline

This manager allow us to re-arrange the pipeline in a variety of ways. For example, the surface tree may be declared autonomous. At each node it performs visibility computation, tessellation and rendering in that order via call-backs. Or each thread may be declared autonomous, with $\mathcal{V}$ producing structure L_1, a list of visible primitives, that $\mathcal{T}$ traverses to produce L_2, a list of triangle-strips which $\mathcal{R}$ traverses and finally sends to the OpenGL server. It is the 'responsibility' of each data structure to provide concurrency control for simultaneous reads and writes. To implement the asynchronous rendering scheme of [KMZH97], we merely disable the concurrency control and allow $\mathcal{T}$ and $\mathcal{R}$ to maintain consistency by their order of accesses to L_2. We demonstrate the initial configuration of the pipeline using an example: (This, in fact, is the default set-up.)

Create n processes. (n is user specified, and is typically equal to the number of processors on the system.)
Default initial 1-process configuration:

- Mark $\mathcal{V}$ as autonomous
- Set `Associated` data structure to `Surface Tree`
- Set `PrimitiveEnd` Callback to
- `Tessellate(Primitive);Render(Primitive)`
- Start PipeManager

Default initial 2-process configuration:
Process 0:

- Mark $\mathcal{R}$ as autonomous
- Set `associated` data structure to `Render List`
- Start PipeManager

Process 1:

- Mark $\mathcal{V}$ as autonomous
- Set `Associated` data structure to `Surface Tree`
- Set `PrimitiveEnd` Callback to
  { `Tessellate(Primitive)`; Add to `Render List` }
- Start PipeManager

All unspecified call-backs are `NULL`. If more than two processors are available, all the other processors start with $\mathcal{T}$ as the autonomous entity with `Surface Tree`. Note that in our current implementation, `Surface Tree` is actually a forest with a few tens of trees. Load balancing is easily achieved by starting with an equitable distribution of these trees, followed by work stealing as proposed in [KCM96].

4.2 Dynamic control

While our scheme has all the mechanism for dynamic configuration of the pipeline, currently only simple changes are made:

1. Scheduling priority of a bottlenecked autonomous thread may be increased
2. An autonomous instance of bottlenecked thread may be created (on a process where it is not autonomous).
3. Maximum traversal depth of `Surface Tree` may be changed, thus obtaining either less detailed or more detailed primitives.

If thread A spends more than 10% of its time waiting at a synchronization barrier with thread B, B is declared to be bottlenecked. In our current implementation, a thread must be bottlenecked for at least three frames in any block of 5 frames to cause an adjustment in pipeline. If the graphics board is found to be the bottleneck, we currently only change the maximum traversal depths of $\mathcal{V}$ and $\mathcal{T}$ to offset the bottleneck.

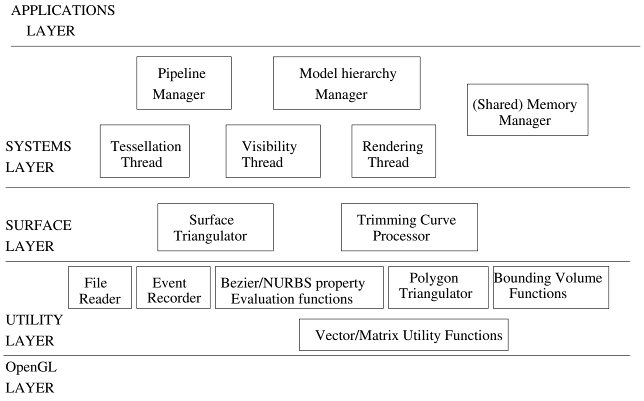


Figure 5: System Architecture

5 Implementation and Performance

We have implemented sLIB using C/C++. It has been tested on the IRIX (Silicon Graphics' version of UNIX) platform. The system design itself is independent of the platform. It currently assumes the availability of shared memory and of an OpenGL interface to the graphics board. It also assumes that all data fit in (virtual) memory. Even though the memory requirements of sLIB are modest compared to systems that directly store polygons, the main memory may be insufficient for large enough models. We observe significant slow-down in such cases. Currently sLIB has no module for disk management. It does have a memory management module for caching triangle-strips.

In our design, we chose to store the Bézier surface representation (control points) separately from the tessellated triangles, since the control points remain static, but the cache sizes for primitives may change between frames. Caches are allocated in multiples of 1 kilobytes to avoid fragmentation. We store the surface properties with the triangle cache: if re-tessellation is not required, no access to the control points' memory page is needed. Note that the surface properties are less than 50 bytes and it does not impose severe overhead to move them when the cache of a primitive requires relocation. We found the version that stores control points with the cache, slower to relocate as the representation of a trimmed Bézier patch may take a few hundred bytes. In addition, a primitive may be a super-surface and may comprise multiple sets of control points.

Our library can be divided into three separate layers as shown in Fig.5. A function in any layer has access to all functions in layers below. However, most calls are only to functions in the layer immediately below. Functions in the system layer typically call other functions of that layer via call-backs. Our current scheme provides a number of plug-in opportunities for exploring trade-offs via function pointers. To avoid the overhead associated with function pointers, we also allow hard-wiring of function calls using conditional compiling. Note that the responsibility still lies with the user to configure the pipeline consistently. Our original attempt allowed fewer user errors and was stage driven rather than procedure driven, i.e., instead of deciding where to place a given procedure in the pipeline, we tried to find the appropriate procedure to place at stage i of the pipeline. It turned out to be too rigid and did not provide enough freedom to configure the pipeline the way we sometimes needed.

We believe that our system is first of its kind for rendering large surface models. Good systems [Per97, 3di97, VIS97] for efficient rendering management are available, but none are designed for dynamic surface tessellation, nor are they flexible enough to allow the types of experiments we are interested in. With the exception of [KMZH97], most other surface rendering systems are known to be more than a hundred times slower than sLIB . In addition, these systems [Ake93, DN93, Pea97] are little more than a collection of functions for evaluating properties of NURBS surfaces and for triangulating them; they do not form a comprehensive rendering system. Compared to the performance of [KMZH97], sLIB achieves a speed-up of about 1.5-2.5. Approximately 40% of this improvement may be attributed to the fractional condition scheme and 20% to our improved visibility algorithm. Most of the rest of the improvement comes from using parametric super-surfaces. Note that the tessellations produced by using our parametrically constructed super-surfaces also satisfy quality constraints not satisfied by the static LODs of [KMZH97].

To test the performance of our system, we rendered a variety of Bézier surface models on several SGI platforms, as shown in Table 1. Fortunately a polygonal form of the submarine model was also available to compare our approach to direct polygonal rendering. We obtained an speed-ups of 5 to 15 times over a simple polygonal rendering. In this comparison, we did not use multiple LODs for the polygonal rendering. we report our best performance on each system (modulo pipe-line changes). All numbers are for synchronous rendering. Asynchronous rendering is 2-3 times faster, with a degradation in image quality.

We also tested our performance on an O2 with 128 M memory. While most models listed in Table 1 did not fit in memory, we were still able to render about 15,000 Bézier patches per second. Due to a set up problem with our IRIX installation, we were unable to use approximately 50-100M of memory on our machines.

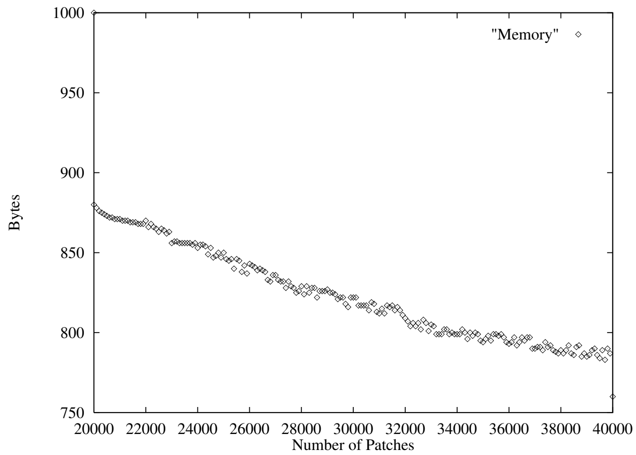


Figure 6: Effect of growing model size on memory requirement. Y axis shows average number of bytes per patch.

On average, the space consumption was observed to be 800-900 bytes per Bézier patch (including cached triangles). On some executions, however, the average space per patch exceeded 10 kilobytes for some frames. These were highly closed-up views and a number of non-visible patches still had their caches active and were thus contributing to the total count. Furthermore, the average space required per patch went down as the size of models went up (see Fig. 6), since larger models typically have smaller patches on screen. Also note from Fig. 7 that the rendering slow-down curve is well below linear with increasing model size. As model sizes continue to grow, primitives processed by our system per frame consists of more Bézier patches. These observations lead us to believe that the approach of dynamic tessellation scales well.

One of the surprising discoveries was the effect of interrupt overhead on the rendering thread $\mathcal{R}$, specially when it is the only autonomous thread on its process. The rendering processor is often blocked waiting for the OpenGL FIFO buffer to be emptied, signifying a rendering bottleneck. This block can cause multiple in-

Platform	Garden (Fig.8) (39,000 patches)	Car (Fig.9) (9,000 patches)	Submarine (Fig.10) (24,000 patches)	Speed-up over polygonal sub
Octane, 195MHz R10000 , 256M Mem, MXI graphics	0.9	7.9	1.2	5.6
Octane, 2 195MHz R10000, 256M Mem, MXI graphics	1.5	12.3	2.8	8.5
Onyx2, 1 195MHz R10000, 1G Mem, InfiniteReality	1.9	11	2.8	7.9
Onyx2, 4 195MHz R10000, 1G Mem, InfiniteReality	5.2	17.5	6.3	16.8

Table 1: Performance of sLIB : The numbers are average frames per second rendered. Each experiment for a given model was performed using the same sequence of 5000 user view-points. The reported frame rate is 5000/observed wall clock time. The full sub was not loadable on the the Octanes. Comparison is based on one part of it.

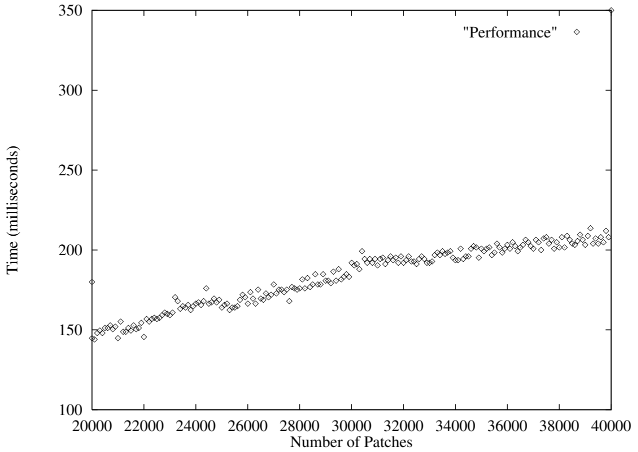


Figure 7: Effect of growing model size on rendering speed. Y axis shows average time to render.

terrupts on the IRIX system. We found an $\mathcal{R}$ processor waiting for OpenGL up to 50% of the time on a dual processor Octane. A significant part of this was attributed to processing interrupts associated a full FIFO. The total idle time actually went down by about 15-20% when we artificially introduced sleeps in $\mathcal{R}$ (sleep times were counted as idle). This occurred due to a decrease in the number of interrupts. This result suggests always running a second autonomous on the process running $\mathcal{R}$, although at reduced priority.

Our dynamic pipeline re-configuration was not very successful. While the overhead for turning it on was statistically insignificant, its effect caused an improvement of less than 5%, with occasional slow-downs of up to 2-3%. We believe that while our mechanism for pipe-line re-configuration is quite promising, a more intelligent algorithm is needed to obtain significant speed-ups.

6 Conclusion

We have designed and implemented sLIB : a configurable system useful for interactive rendering of trimmed tensor product or triangular NURBS surfaces. It also forms a general test-bed for quickly prototyping new algorithms. In addition, various system trade-offs can be systematically explored using sLIB , by changing just the part being profiled and leaving all else constant. As a case in point, we have recently been exploring using a network of workstations to enable interactive triangulation and rendering of large scale models. sLIB framework has been quite useful in quick testing of our preliminary ideas.

A novel feature of sLIB is its ability to re-configure the rendering pipeline to best suite the relative processing powers of the CPU and graphics board available. sLIB also includes improvement over the visibility and tessellation algorithms in the literature. In addition, the fractional condition framework of estimating when a primitive needs to be re-evaluated for visibility or for triangulation completely eliminates the need to process about 20-25% of the model. In summary, sLIB achieves a speed-up of about 2-3 over the previously best known algorithms. More importantly, sLIB is a comprehensive system and represents an integration of a variety of techniques in the literature. We also believe, sLIB scales almost linearly with increasing processing power and deteriorates only sub-linearly with increasing model size. A copy of sLIB is now available on our website (withheld for blind review). We plan to make a formal public release in the near future.

In future, we plan to perform an extensive analysis and comparison of processing and memory requirements of our approach with the best known surface simplification schemes (coupled with static-triangulation). To this end, we hope to collect the implementations by the corresponding authors for a useful comparison. In extensions of our work, we plan to make the on-line statistic collection and pipeline management more intelligent, and even predict bottlenecks. Currently the system is too specific to the threads we have chosen. Expanding the framework to add more threads would be useful. Such expansion would also facilitate fine-tuning of the rendering system at a finer level than just threads. We are currently working on algorithms for efficient farming of visibility omputation and tessellation over a network of workstations. We also plan to add the capability of importing a variety of standard surface file formats into sLIB .

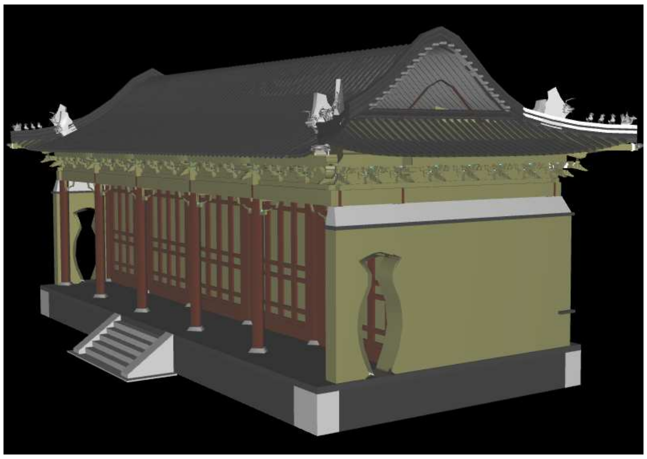


Figure 8: Garden Model

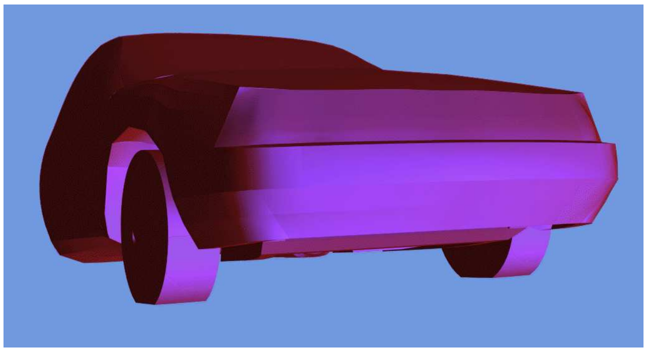


Figure 9: Car Model

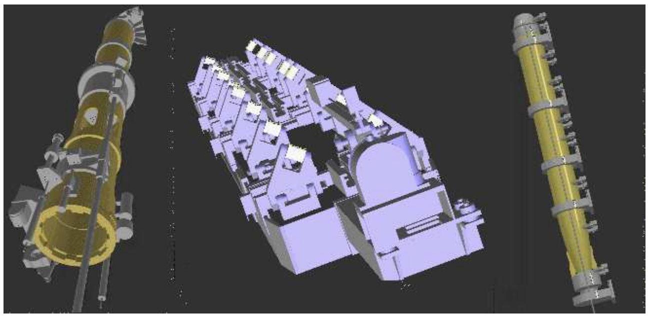


Figure 10: Submarine Model

References

[3di97] Ibm 3d interaction accelerator, 1997. http://www.research.ibm.com/3dix/description00.html.

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

[Ake93] K. Akeley. Reality Engine Graphics. In *Proc. ACM SIGGRAPH*, pages 109–116, 1993.

[BDK98] G. Barequet, C. Duncan, and S. Kumar. RSVP: A geometric toolkit for controlled repair of solid models. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):162–177, 1998.

[Bro96] F. Brooks. Newport news shipbuilding company interview re ship design process. Technical Report TR96-002, Department of Computer Science, University of North Carolina, 1996. Restricted Circulation.

[CVMe96] J. Cohen, A. Varshney, D. Manocha, and G. Turk et al. Simplification envelopes. In *Proc. ACM SIGGRAPH*, pages 119–128, 1996.

[DGHM93] T. DeRose, R. Goldman, H. Hagen, and S. Mann. Functional composition algorithm via blossoming. *ACM Transactions on Graphics*, 12(2):113–135, 1993.

[DN93] M.F. Deering and S.R. Nelson. Leo: A system for cost effective 3d shaded graphics. In *Proc. ACM SIGGRAPH*, pages 101–108, 1993.

[Far93] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1993.

[GH97] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proc. ACM SIGGRAPH*, pages 209–216, 1997.

[Hop97] H. Hoppe. View dependent refinement of prograssive meshes. In *Proc. ACM SIGGRAPH*, 1997.

[KCM96] S. Kumar, C. Chang, and D. Manocha. Scalable algorithms for interactive visualization of curved surfaces. In *Supercomputing*, Pittsburgh, PA, 1996.

[KM96] S. Kumar and D. Manocha. Hierarchical visibility culling for spline models. In *Proc. Graphics Interface*, pages 142–150, Toronto, Canada, 1996.

[KML96] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large NURBS models. *IEEE Transactions on Visualization and Computer Graphics*, 2(4):323–336, Dec 1996.

[KMZH97] S. Kumar, D. Manocha, H. Zhang, and K. Hoff. Accelerated walkthrough of large spline models. In *Proc. Symposium on Interactive 3D Graphics*, pages 91–101, Providence, RI, 1997.

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

[NCG91] Initial graphics exchange specification (iges) version 5.1. National Computer Graphics Association, 1991.

[Pea97] L. Piegl and W. Tiller et al. Nlib, an object-based user-oriented nurbs library (a companion to the nurbs book), 1997. (Also see, NuGraphs system at http://www.okino.com/nrs/nrs.htm).

[Per97] Iris performer, 1997. http://www.sgi.com/Technology/Performer/technical.html.

[PR95] L. Piegl and A. Richard. Tessellating trimmed NURBS surfaces. *Computer Aided Geometric Design*, 27(1):16–26, 1995.

[RHD89] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *ACM Computer Graphics*, 23(3):107–117, 1989. (SIGGRAPH Proceedings).

[VIS97] Hewlett-packard visualize graphics, 1997. http://www.hp.com/wsg/products/grfx/grfxhome.html.

[XV96] J. Xia and A. Varshney. A dynamic view-dependent simplification for polygonal models. In *Proc. IEEE Visualization*, pages 327–334, San Fransisco, CA, 1996.