Budget Sampling of Parametric Surface Patches

Jatin Chhugani
jatinch@cs.jhu.edu
Johns Hopkins University

Subodh Kumar
subodh@cs.jhu.edu
Johns Hopkins University

Abstract

We investigate choosing point samples on a model comprising parametric patches to meet a user specified budget. These samples may then be triangulated, rendered as points or ray-traced. The main idea is to pre-compute a set of samples on the surface and at the rendering time, use a subset that meets the total budget while reducing the screen-space error across the model. We have used this algorithm for interactive display of large spline models on low-end graphics workstations. This is done by distributing the points on the surface to minimize surface error. These points are then drawn as screen-space squares to fill the gaps between them. Our algorithm works well in practice and has a low memory footprint.

Keywords: Point sampling, Spline surfaces, Adaptive tessellation

1 Introduction

Curved surface models are used in applications ranging from computer aided design and medical visualization to entertainment. For example submarines, airplanes, automobiles, etc. are commonly modeled as splines. Our overall goal is to render complex models consisting of tens of thousands of spline patches, and be able to explore and interact with them in real time on a graphics workstation with limited resources. Although the algorithm described in this paper is broadly applicable to any parameterizable surface, we focus on spline models.

1.1 Related research

Ray tracing [Whitted 1979; Kajiya 1982; Nishita et al. 1990], pixel level subdivision [Catmull 1974; Shantz and Chang 1988], and scan-line based algorithms [Whitted 1978; Blinn 1978; Lane et al. 1980] have been used for displaying spline models. These techniques, although not as efficient as triangulation based methods, generate more accurate images because they compute the position and color of each pixel. In contrast, a triangle based tessellation has accurate position and color only at the vertices of the triangles. One compromise is to decompose the surface into nominally curved 'elements', which follow the surface more closely than a triangle may [Szeliski and Tonnesen 1992; Witkin and Heckbert 1994; Kalaiah and Varshney 2002]. Such elements are not easily made C^0 continuous as triangular approximations are. Nonetheless, if they are small enough, they can be shaded well to produce accurate images [Kalaiah and Varshney 2002; Adamson and Alexa 2003].

Recent research [Zwicker et al. 2001; Kalaiah and Varshney 2002; Adamson and Alexa 2003] shows how to perform this shading well. One aspect that has not received sufficient attention is: how to select the locations of these elements on the surface, specially when the total number of samples is bounded. This is precisely what we address in this paper.

Computer graphics systems traditionally use triangles to approximate surfaces since triangle rendering may be hardware-accelerated. View-dependent uniform [Abi-Ezzi and Shirman 1991; Kumar et al. 1996; Kumar et al. 1997] and adaptive triangulation [Filip 1986; Vlassopoulos 1990] (as well as a combination of the two [Chhugani and Kumar 2001]) have been employed to render spline models. Alternatively, one may generate a large number of triangles and later perform view-dependent simplification of the triangles [Hoppe 1996; Rossignac and Borrel 1993; Xia et al. 1997; Schroeder 1997]. Recently, the two methods have been combined into a single algorithm [Chhugani and Kumar 2001]. However, experience shows that in order to guarantee a small screen-space error, we are forced to use many triangles that are small on the screen.

For triangles smaller than a pixel, point-based rendering has been gaining popularity. The idea of using points as display primitives for continuous surfaces was introduced in 1985 by [Levoy and Whitted 1985], and recently has been explored further in [Grossman and Dally 1998; Rusinkiewicz and Levoy 2000; Pfister et al. 2000; Stamminger and Drettakis 2001]. These techniques use hierarchical data structures and forward warping to store and render the point data efficiently. Wand et al. [Wand et al. 2001] present an output sensitive rendering algorithm that renders a dynamically chosen sample of sub-pixel triangles. Algorithms that combine triangle and point rendering have also been proposed [Cohen et al. 2001; Chen and Nguyen 2001].

The focus of this paper is on allocating a budget of total number of samples among a number of surface patches of a model. This is similar in spirit to the problem solved by [Funkhouser and Séquin 1993]. However, the granularity of primitive selection in our application is much finer and hence the knapsack formulation is too slow. We assume as input: $\{\mathbf{F}_i\}$, a set of surface patches, and C, the total number of elements that can be displayed (or otherwise processed) in a single frame. We first allocate C_i to patch $\mathbf{F}_i$ so that $\sum C_i = C$, ensuring fairness. Fairness implies that all patches have approximately the same maximum *screen-space* error. (We measure error in terms of geometric distance between surfaces.) Further, for each patch $\mathbf{F}_i$ we determine which C_i points on the domain (samples) to choose so that the deviation of these samples from the actual surface is minimized. We also produce a size parameter for these samples: the elements must be drawn large enough so that no holes are left between neighboring elements.

Our goal is to limit the overhead of the sample selection algorithm at the rendering time, as the CPU is often required to process the elements themselves. Hence, we perform most of the computation before the rendering starts. This pre-computation phase chooses a list of samples on the surface, sorted by their 'importance' in reducing deviation. This results in more samples chosen in the areas of high curvature. At the rendering time, we first compute C_i for patch $\mathbf{F}_i$ and then simply select its C_i most important samples.

We demonstrate our algorithm by employing it in a surface rendering system. In our system we have used points (OpenGL GL_POINT) as screen-primitives for simplicity. (Hence we will use screen-primitive and point-primitive interchangeably.) We also choose to let the primitives overlap in object space (in the spirit of point-based rendering) and thus do not need to maintain or manage any topological information. Since these points are distributed uniformly in the screen space, they may also be used to speed up surface ray intersecting for efficient ray-tracing. One might also similarly choose to, instead, triangulate the samples and obtain uniform screen triangles.

1.2 Main contributions

Our algorithm has two parts. It pre-computes a set of sample points on each surface patch, computing more samples in highly curved areas. At the rendering time, a view-dependent subset of these samples is selected and associated primitives are sent to the graphics pipeline. Its main novel components are:

- An algorithm to pre-compute samples locations in the decreasing order of importance
- An algorithm to fairly allocate the overall budget to individual patches, which distribute screen-space error uniformly across the model
- A system implementing the above algorithms demonstrating efficient spline model rendering

In this paper we present a novel error diffusing budget allocation of samples. It is important to note that we consider the curvature of the underlying surface to determine which samples to pre-compute and which subset to use at run time; other recent point-based rendering algorithms [Zwicker et al. 2001; Stamminger and Drettakis 2001; Fleishman et al. 2003] do not. As a result of these strategically placed samples, we are able to generate fewer points for similar error bounds. We do not need to maintain any expensive data structure of selected samples either. We also guarantee a hole free tiling of surface patches; some [Stamminger and Drettakis 2001] do not.

In a system where the rendering speed is important, a point-based scheme needs to be combined with triangle based tessellation [Cohen et al. 2001; Chen and Nguyen 2001] for largely flat areas on the screen. Furthermore, richer primitives instead of point-primitives may be used as in [Kalaiah and Varshney 2002] to improve image quality. We do not address point filtering or anti-aliasing here but recent techniques [Zwicker et al. 2001] could be applied at the cost of hardware rendering performance. Our method is well suited for smooth shading. In fact, applications requiring textures seem to necessarily require richer primitives (or much smaller points). Another limitation of our technique is the need to pre-compute a fixed set of samples. However, the data stored per sample is small and hence many may be stored. Further, on the rare occasion when even more samples are needed, extra samples may be computed dynamically. This may be acceptable if only few patches need extra samples on average.

1.3 Organization

In this presentation, we assume familiarity with NURBS and Bézier surfaces and Delaunay triangulation. We elaborate the basic idea behind our approach in Section 2. This is followed by the details of the algorithm in Sections 3 (pre-sampling) and 4 (render time sample selection). Section 5 describes our implementation and reports the results obtained. Finally, conclusions are drawn and future directions listed in Section 6.

2 Background

In order to ensure that the elements at the chosen samples cover the surface, we must draw them large enough to fill the gaps between them. We have chosen spheres to bound the size of the elements in the object space. We generate the minimum radius of these spheres so that they cover the patch. Thus, if the elements drawn at each sample cover the sphere, we eliminate holes. In our system we display the samples as GL_POINTs. We describe how to form covering spheres next.

2.1 Spheres as elements

Imagine spheres at the sampled points on the surface such that every point on the surface lies inside at least one sphere. In other words, when these spheres are projected on the image plane, the ellipses thus formed would have no gaps between them (Figure 1). As we show later, the spheres centered on our sampled points have the property that the actual surface does not deviate by more than r from the surface of the spheres, where r is the radius of the largest sphere. We choose the sample points so that they locally reduce the deviation of the surface from the approximating surface (i.e. the surface of the sphere) and hence reduce the radius of the sphere.

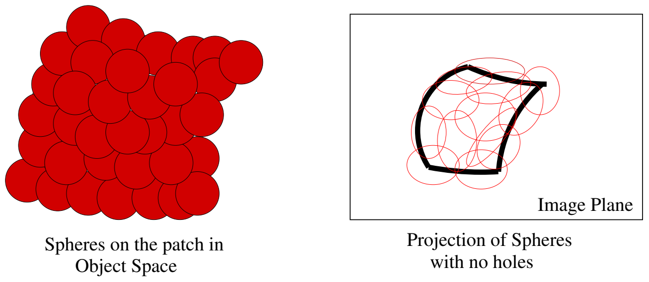
Image Plane
Spheres on the patch in Object Space
Projection of Spheres with no holes

Figure 1: Projection of surface samples on image plane

To render an individual sphere, we compute the center, Q, of the projected ellipse (by projecting the sample point). We next compute the maximum deviation (say d) of the elliptical surface from Q. Now if we render a square splat of dimension $2d$ centered at Q, the surface of the projected ellipsoid is covered (Figure 2). At the rendering time, we need to compute Q and d. Assigning the same value of d to all spheres on a spline patch greatly simplifies the problem without increasing by much the total number of spheres needed for a given error bound. We will prove that the screen-primitive's size, d, that we assign to each patch is optimal up to an integer. In other words, the value of d we compute is less than one pixel away from the optimal value. Thus for each patch, we choose samples that when projected as points on the screen, do not leave any holes, and also obey the deviation bounds. To compute the samples and their point sizes for a given patch, we just need $\log M$ table lookups for a patch, where M is the total number of pre-computed samples.

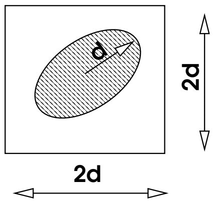

Figure 2: Square splat covering the projection of a sphere on the image plane

2.2 Overall algorithm

Our algorithm has two main steps:

Pre-sampling: We progressively compute an ordered list of sample points on the domain of each spline patch. These samples are associated with spheres centered at them. Each new sample minimizes the deviation of the resulting spheres from the original surface (across all points on the domain). The point also stores this deviation value. Note that these values are non-increasing. As samples are added, old spheres' radii change.

View-dependent point selection: At the rendering time, we start with the list of pre-computed domain samples for each patch. We first compute the screen-space error $\delta(\mathbf{F})$ that must be incurred for each patch to remain under the overall budget. For a patch, $\mathbf{F}$, we first compute $\Delta(\mathbf{F})$ (the required object-space error) $= \gamma\delta(\mathbf{F})$, where γ is the scaling of the longest projection (detailed in section 2.3). We now search for $\Delta(\mathbf{F})$ in the sorted list of error values stored for patch $\mathbf{F}$. By construction, adding the corresponding sample and all its predecessors guarantees $\Delta(\mathbf{F})$.

2.3 Scale factor computation

The allocation across the patches is entirely view-dependent and performed online. The allocation within a patch uses surface derivatives and is slower. In order to perform this allocation fast, we have decided to choose samples from a list of candidates. This list must be pre-computed. This means the error at the samples must be pre-computed in the object space. However, we measure the final error as the distance between the rendered primitive and the actual surface in the screen space. We use the scale factor to transform errors between the object and screen spaces and show how to compute it in this section.

We define the scale factor of a point p in object space as the length of the smallest vector anchored at p that projects to a unit vector in the screen space (Figure 3). As shown below, this minimum value equals $\frac{(f+z)^2}{fL}$ where L is the length of the vector from the eye to the point p, and f is the focal length, and $(f+z)$ is the length of the projection of the vector from the eye to the point p, along the principle viewing direction.

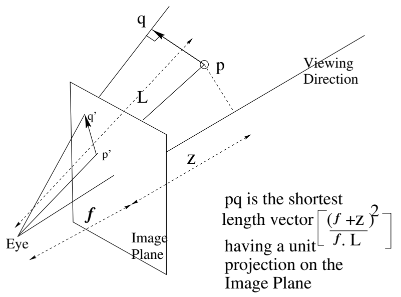
q
p
Viewing Direction
L
q'
p'
z
f
Eye
Image Plane
pq is the shortest length vector $\left[\frac{(f+z)^2}{f.L}\right]$ having a unit projection on the Image Plane

Figure 3: Scaling of a unit vector in screen space

We first compute the maximum deviation of the perspective projection of the surface of a sphere from the projection of its center. Let $p = (x,y,z)$ be the center of the sphere and let (r,θ,ϕ) be the local spherical coordinates of the vector rooted at the center of the sphere that has the projection of maximum length. So the Euclidean coordinates of the tip of the vector (say $q(r,\theta,\phi)$ are $(x + r\sin(\theta)\cos(\phi), y + r\sin(\theta)\sin(\phi), z + r\cos(\theta))$. Let XY plane be the image plane and let $(0,0,-f)$ be the camera center. Projection of p on the image plane = $p' = (\frac{fx}{f+z}, \frac{fy}{f+z}, 0)$ and projection of q = $q'(r,\theta,\phi) = (\frac{f(x+r\sin(\theta)\cos(\phi))}{f+z+r\cos(\theta)}, \frac{f(y+r\sin(\theta)\sin(\phi))}{f+z+r\cos(\theta)}, 0)$. So maximizing the projection is equivalent to maximizing the length of $p'q'$ over all θ and ϕ, which yields $|p'q'| = \frac{fL^2}{(f+z)\left[(f+z)\sqrt{\left(\frac{L^2}{r^2}-1\right)}-\sqrt{x^2+y^2}\right]}$ where $L = \sqrt{x^2+y^2+(f+z)^2}$

This gives the ratio (γ) of length of the vector to its projection length = $\frac{r}{|p'q'|}$. In particular, the scaling at a point is given by an infinitesimally small vector rooted at the point, i.e., $r \to 0$:

$$\gamma = \lim_{r\to 0} \frac{r}{|p'q'|} = \frac{(f+z)^2}{fL} \qquad (1)$$

3 Pre-Computation

Recall that we precompute a list of 'important' samples for each patch, as well as the radii of covering sphere at those locations.

3.1 Pre-sampling

We would like to produce the best approximating set S_n of n samples for each value of n. Let us say, we cover samples S_i with the set O_i of object primitives (sphere in our example). Since we do not want to store a different set for each n, we mandate $S_n \subset S_{n+1}$. Given n centers and radii, we need to find the $n+1^{st}$ center and the new $n+1$ radii that minimize the deviation of the resulting O_{n+1} from the surface. To save time and space we have chosen to compute a single radius r_{n+1} that may be used for all objects in O_{n+1}. The following algorithm, though, is able to generate radii incrementally so that only a small number of samples in S_n need to change their corresponding radii in S_{n+1}. The location of samples in patch domains are chosen as follows:

1. Start with a minimal sample set (e.g. the four corners) in the domain.

2. Generate the (2D) Delaunay triangulation of this minimal set. (Other good quality triangulation may be used as well.) Now compute the center and the radius of the circumscribing spheres for each of the triangles obtained (see Section 3.2).

3. While the sphere with the maximum radius has a radius greater than a user specified error Δ_u:
   Append to S, (Q, r_{max}), the center and the radius of the largest circumsphere of all triangles (in domain space). Also add Q to the Delaunay triangulation.

At the end of the process, we have an ordered list, S, of domain samples for each patch and their radii. We only store the largest radius for each S_n, thus overdrawing some samples.

Claim 1: *Maximum deviation of a surface patch from the approximating sphere is equal to the radius of the sphere that encloses that patch.*

Justification: We guarantee this by making sure that the sphere bounds the surface element (more in section 3.2). Clearly, every point inside a sphere of radius r is at a distance less than or equal to r from the surface. ∎

Claim 2: *At any instant of the domain triangulation, let r_{max} be the maximum radius of the circumspheres. If we draw spheres with radius = r_{max} on all the sampled points on the surface, no holes are left on the surface.*

Justification: Suppose the radius of the circumcircle of a triangle, t, is r. If we draw three circles with radius r and centers at the three vertices of t, no point on the triangle would be left uncovered (Figure 4). Hence drawing spheres with a radius r_{max}, $r_{max} > r$, at each vertex would cover all triangles. ∎

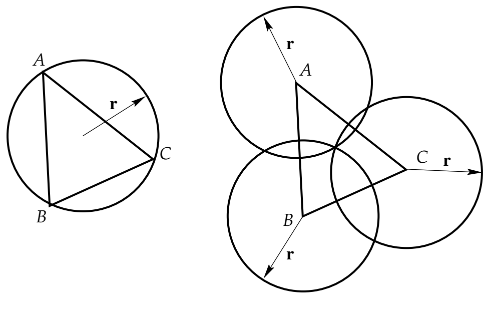

Figure 4: Covering surface by drawing spheres at the vertices

Moreover, the maximum deviation of the surface from these spheres still remains the same, because for each circumscribing sphere of a triangle, each surface point is at most at a distance of *radius* from the center and each of the *three* substituting spheres pass through the center of the original sphere. Hence the deviation criteria and the criterion for no holes are simultaneously satisfied by selecting points in the above fashion. Also with each sample point, we get the maximum object space deviation Δ_i (for the i^{th} selected point), between the approximation and the surface if all the points S_j, $j \leq i$ are considered on the surface for point rendering. Note that $\Delta_{i+1} \leq \Delta_i$ and $\Delta_{|S|} \leq \Delta_u$, where $|S|$ is the total number of samples pre-computed and Δ_u is the user specified value as described in section 3.1. (Technically, deviation could increase on adding a new point for some degenerate patches as shown in Figure 5. Even in these rare cases, adding a sequence of samples always leads to a lower deviation. We add or delete the sequence of samples together and assign a single index to them.) Thus given a deviation bound δ, we can find the prefix of S that generates an approximation with deviation less than or equal to δ in the screen space (or $\gamma\delta$ in the object space).

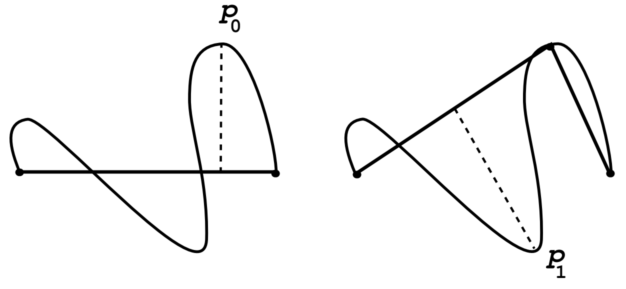

Figure 5: The maximum deviation occurs at p_0 when the curve is approximated by a straight line as shown on the left. The deviation increases when we use three samples as shown on the right.

3.2 Computation of sphere parameters

For each triangle in the object space, we need to find the center and the radius of the sphere that encloses every point on the surface corresponding to the region inside the triangle on the domain space. Let the three domain points (of Bezier patch $\mathbf{F}$) be represented as $t = (p_1, p_2, p_3)$, $p_i = (u_i, v_i)$. Let $P_1 = \mathbf{F}(p_1), P_2 = \mathbf{F}(p_2)$ and $P_3 = \mathbf{F}(p_3)$ be the corresponding object space points. Now compute the circumcenter, Q, and the circumradius, r_1, for the triangle $P_1P_2P_3$, unless:

1. All the three points are coincident. Set $Q = P_1$ and $r_1 = 0$.
2. The three points lie on a straight line. Let P_1 and P_2 be the extreme points. Set $Q = \frac{P_1+P_2}{2}$ and $r_1 = \frac{|P_1P_2|}{2}$.
3. The three points form an obtuse angled triangle (circumcenter is outside). Let P_1 and P_2 be the end points of the longest edge. Set $Q = \frac{P_1+P_2}{2}$ and $r_1 = \frac{|P_1P_2|}{2}$.

Having evaluated Q and r_1, shoot a ray QR perpendicular to the plane of the triangle, intersecting the surface at point R (Figure 6). We use Powell's method [Chhugani and Kumar 2001; Press et al. 1993] to find the point of intersection. Let $|QR| = r_2$. Consider the sphere with center R and radius $r = \sqrt{r_1^2 + r_2^2}$. We find the point, P, on the sub-patch furthest from R (again using Powell's method), and if $|PR| > r$, set $r = |PR|$.

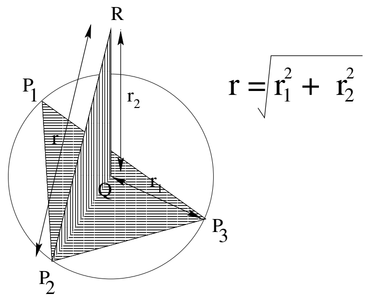

Figure 6: Computation of the radius and center of the circumscribing sphere

4 Online Sampling

In this section we describe the point-based tessellation algorithm that meets a user-specified budget. This meets a target frame-rate also even with a GL_POINT based renderer as we have used. Since we use small point sizes in practice, we have found that the fill rate is usually not the bottleneck. Thus bounding the number of points drawn is sufficient to meet a target frame-rate in practice.

Recall that a fair allocation of the budget implies a uniform screen-space error across all patches. In other words, we should allocate the *same* screen-space error to all patches: the smallest such error that still allows us to meet the overall budget. Our algorithm, instead, allocates error such that the screen-space error of all patches are within a pixel of each other.

4.1 Sample selection for a patch

To meet the required screen-space deviation, $\delta(\mathbf{F})$ for patch $\mathbf{F}$, the object-space deviation required for the approximation is $\Delta(\mathbf{F}) = \gamma(\mathbf{F})\delta(\mathbf{F})$, where $\gamma(\mathbf{F})$ is the minimum of the scale factors of all points comprised by $\mathbf{F}$. (Recall that the minimum value of γ corresponds to the maximum projected length of the error vector.)

Unfortunately, the scale factor of a set of points can vary arbitrarily. Hence, even though we could use a single scale factor per patch, we employ a more spatially coherent scheme. We use an octree based spatial partitioning of space. For all patches contained in a sufficiently small partition, we use the same scale factor. Typically partitions close to the view-point are more refined than those

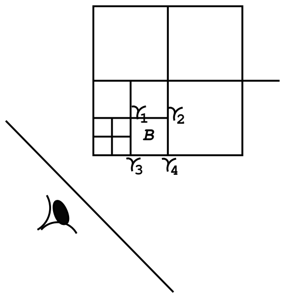

Figure 7: Grouping patches based on scaling factor

further away, as the scale factor close to the view point varies faster. The sample selection proceeds as follows:

1. Start with the octree cubes used in the previous frame. We call a cube *terminal* if the scale factor of the eight corners of the cube differ by less than γ_u, a user-specified tolerance. For example, the terminal nodes are shown in Figure 7. The example node B is terminal because $\max_{1 \le i,j \le 4} |\gamma_i - \gamma_j| < \gamma_u$. If a leaf node from the previous frame ceases to be terminal, we subdivide it. Otherwise, if a leaf node's parent becomes terminal, we recursively delete the nodes at the current level. If γ_u is chosen to be $\frac{1}{\delta}$, the approximation does not under-deviate by more than a unit pixel, where δ is the desired upper bound on the screen space deviation.

2. The scale factor of a cube is the smallest of the scale factors of its corners. For each patch completely contained in a terminal cube, B, with scale factor $\gamma(B)$, we choose all samples $S_i, i \le j$, such that the associated deviation $\Delta_j < \delta\gamma(B)$ and $\Delta_{j-1} > \delta\gamma(B)$.

3. If a patch lies in more than one terminal cubes that are all adjacent to each other, we assign to the patch, the minimum scale factor of those cubes.

4. For larger patches, however, we do need to use different scale factors in different regions of the patch. We subdivide the domain of patches that span terminal cubes that are not adjacent to each other. We apply the scaling algorithm described above to each sub-domain K to compute the required object-space deviation Δ_K. For each sub-domain, we find the subset, $S_K \subset S$, of samples in S that belong to domain K.

4.2 Budget allocation per patch

At the rendering time, to guarantee a given frame rate, we can render only a certain number of points per frame on a given graphics platform. Let us say the maximum number of allowable points per frame is C, a user specified constant.

Formally, we need to compute C_i, the budget for patch i, such that $\sum_i(C_i) = C$. Let the total number of patches be N. For distribution purpose, we would want to minimize the screen space error for every patch. Since we use the size of the screen-primitive to bound the maximum deviation, by choosing the same screen-primitive size for every patch we fairly distribute error. In particular, a point size of d bounds the maximum screen space deviation of the actual surface from the approximation surface by $\delta = \frac{d}{2}$.

We compute the patch budget C_i from the overall budget C incrementally from the previous frame's solution. Consider frame j. Assume the scale factor for the i^{th} patch is $\gamma_i(j)$. Hence, if a point size of $\delta_i(j)$ is chosen, we can compute the maximum allowable object space deviation for the patch ($\Delta_i(j) = \gamma_i(j)\frac{\delta_i(j)}{2}$). This object space deviation equals the maximum allowable radii of spheres on the surface. So we need to search for $\Delta_i(j)$ in the list of pre-computed values of errors and use the corresponding prefix of samples. This requires at most $log(M_i)$ lookups, where M_i is the number of pre-computed samples for the i^{th} patch. So we can define a function $C_i^j : \mathscr{R} \rightarrow \mathscr{N}$ that takes the point-size of the rendered points for that patch, and returns the number of points required. We can obtain such $C_i^j, \forall i \in [1..N]$.

Hence the optimization problem can be stated as follows: Minimize ($Max_i \delta_i(j)$, such that $\Sigma_i C_i^j(\delta_i(j)) \le C$ and $\forall i\ \delta_i(j) \ge 0$

C_i^j resembles a step function, (see Figure 8) and we need to formulate this function for every patch for each frame. The optimal

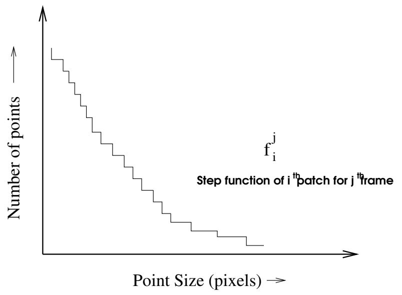

Figure 8: Step function for a particular position of some patch

solution would assign the same point size to each patch. Solving the above equation analytically, to get the optimum solution, might require in the worst case $O(N(M_{max}))$ steps, where M_{max} is the maximum number of pre-computed points for any patch. This is clearly an expensive solution. In practice, though, we need only integer point sizes. So we propose the following algorithm, which does not require computing the function C_i^j for the whole point size range, but instead just for a few discrete values.

1. Assign a point size of $d = 1$ for each patch.

2. Compute the number of total points required, and let $C' = \Sigma_i C_i$, where C_i is the number of points required for the i^{th} patch.

3. If $C' \le C$, then render all the patches with this point size.

4. If $C' > C$, increase the point size d by 1, and go back to step 2.

The above algorithm does a linear search to arrive at a point size that satisfies the budget. Clearly any integer point size less than d would overshoot the budget. However, this algorithm is linear in the maximum allowed point size. In order to improve it, we exploit the temporal coherence of the movement of the navigator to obtain a 2*n-time bounded* approximation algorithm.

1. Allocate to each patch, the point size it had in the previous frame.

2. Compute the number of total points required, and let $C' = \Sigma_i C_i'$ where C_i' is the number of points required for the i^{th} patch.

3. If $C' < C$, decrease the point sizes of each of the patches by one, and recompute C' after processing each patch, and terminate when $C' \ge C$.

4. If $C' > C$, increase the point sizes of each of the patches by one, and recompute C' after processing each patch, and terminate when $C' \leq C$.

In practice, because of the temporal coherence of the eye point, the solution does not change much between frames, and hence usually in 1–2 passes, we obtain the optimal solution up to integral point sizes. This implies that our solution generates errors less than one pixel larger than the optimal. For a better bound on the point size, we could perform a binary search, using the same methodology. However, the cost of computing a tighter approximation far exceeds the benefit of using such a solution.

Claim 3: *Let d_i and d_j be the point sizes allocated to patches i and j respectively. Then $|d_i - d_j| \leq 1$.*

Justification: The proof follows from our technique of changing point sizes for each patch. Each patch starts with a point size of 1 (before the first frame). The sizes are sequentially increased or decreased in round-robin order. Hence the difference in point sizes between any two patches would at most be one. ∎

Note that constant screen-space point sizes only imply that the screen space error is well distributed across the model. We still have fewer, bigger object-space splats for flat regions and smaller, denser ones for more curved regions.

5 Implementation and Results

We have implemented our algorithm and tested it on a variety of models. All timings reported in this paper are from an Onyx2 with a 400 MHz R12000 and an InfiniteReality graphics card. Our experiments consisted of viewing a variety of models from various view points.

Our method of selecting points does not lead to a large overlap between neighboring points. Let us say we compute the size $d(>1)$ to render m points in any frame. If we render the same patch with a point size of $d-1$, we can see some holes on the screen for the corresponding patch (see Figure 9). Hence given the criteria of reducing the deviation, our algorithm performs well in practice.

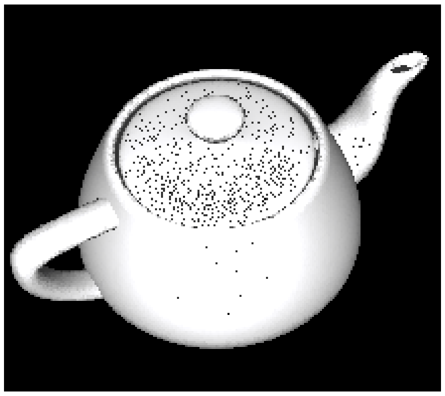

Figure 9: Holes on the Teapot model when the points are rendered with a point size just one less than the computed one

At the pre-processing stage, for each sampled point, we store the position (u,v), the deviation of the triangulation and the radius (or deviation) of the sphere required at that point. The actual domain position of each sample need not be very precise. Note that one byte can accommodate more than enough tessellation (255×255). In fact, using one byte each for u and v, and two bytes for storing the deviation (appropriately scaled) is usually enough for good quality rendering. Thus we may represent a spline patch with its original control points, and $6n$ additional bytes, for selected n samples on a patch. The number of points may be reduced to satisfy storage bounds. This value of n determines the extent to which point rendering can be used. Additionally, we can also pre-compute and store the three spatial coordinates of the sampled points and up to three normal coordinates, although they can be efficiently computed and cached at the rendering time [Kumar et al. 1995].

In Table 1, we report the pre-processing time for different models. This pre-processing algorithm takes time that is proportional to the total surface area of the model, and not to the number of patches. Hence some of the models with large number of patches can be pre-processed comparatively faster. The pre-processing for any patch is independent from that of any other patch of the model. Hence pre-computation can also be carried out in parallel. However, all the times reported are for a single processor. Also, we compute a large number of samples per patch (by giving a small deviation threshold), to carry out extensive tests. Hence the pre-processing times are very high for some of the models.

In Table 2, we report the time spent by the algorithm to compute the appropriate point samples that need to be sent to the graphics pipeline. Note that the overall frame rate for the large models (e.g., the garden) is low even with few points. This is because our sample selection method takes time proportional to the number of patches. The hierarchical version of the algorithm would reduce this time further. Also, we report the average screen space error (defined as the arithmetic mean of the screen space error used for displaying the model every frame) for the simulated browsing of the models. In fact, even the variance in error across frames is low and thus acceptable quality is maintained in all frames. It can be seen that a very small fraction of the total rendering time ($\sim 10\%$) is spent in software to figure out the correct samples. We achieve real time frame rates for most of the models on a hardware customized for triangle rendering. Another interesting observation was that when the screen space area of a patch was considerably large, screen space errors of even $3-4$ pixels did not produce noticeable artifacts (see color plate). This can be explained by the small percentage of error in the projection. Hence, one might use the metric of relative screen-space error (obtained by dividing the screen-space error with some normalized area of the screen space projection).

Model	*Number of Patches*	*Num Samples Pre-computed*	*Pre-process time in minutes*
Teapot	32	129,273	09
Spoon	66	234,290	17
Goblet	72	123,396	15
Dart	100	141,150	09
Coke	330	475,674	32
Scissors	505	141,243	14
Pencil	570	1,051,624	70
Dragon	5354	1,473,961	96
Garden	38646	1,231,200	82

Table 1: Pre-sampling performance

We also note some aliasing artifacts for a low budget of points. These are noticeable across the boundary of patches that do not have any neighboring patches. See Figure 10 for an example. (The artifacts are enhanced by a two times image scaling.) Methods like [Zwicker et al. 2001] help alleviate it. However, they don't work

Model	*Points per frame*	*Average error*	*%-time spent in software*	*Frame rate*
Teapot	90,000	1.8	0.3	31
Spoon	90,000	1.1	0.6	34
Goblet	100,000	1.45	0.5	34
Dart	90,000	0.7	0.6	36
Coke	90,000	2.09	0.9	25
Scissors	90,000	1.7	0.9	24
Pencil	70,000	3.00	2.44	23
Dragon	50,000	3.12	11.9	20
Garden	50,000	7.5	19.1	7

Table 2: Run-time behavior of our algorithm

well for large point sizes. Reducing the point size on the boundary or silhouettes will reduce this problem.

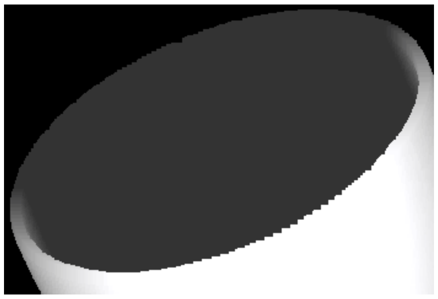
Figure 10: Aliasing effects across the boundary of a patch

Visual artifacts can also be seen across boundaries of patches having appreciable discontinuity in normal values from the boundaries of neighboring patches (see Figure 11). To reduce this problem, we smooth normals near the boundary. These normals are averaged with the nearby boundary points on the adjacent patch. For example, normal, $N_{\mathbf{A}}(u,v)$, for patch **A** with adjacent patch **F** is replaced by $\frac{\varepsilon+(1-v)}{2\varepsilon}N_{\mathbf{A}}(u,v)+\frac{\varepsilon-(1-v)}{2\varepsilon}N_{\mathbf{F}}(u,0)$, for all samples with v in $[1-\varepsilon,1]$, with a small ε. In practice, ε around 0.005 works well. In Figure 12 we show the improvement in rendering of the base of a goblet model using the modified algorithm.

6 Conclusion

We have presented a view-dependent algorithm for distributing samples on a parametric patches. We have demonstrated a display system using points as primitives proxying for each sample. The algorithm does most of its work off-line. At the rendering time, it performs minimal computation to select the set of samples that need to be rendered. This may be used to provide a guaranteed frame-rate visualization. We are able to obtain real time rendering rates with small errors for most models. Our current scheme pre-computes a list of samples for each patch. If more samples are sometimes needed, one could generate them online as in [Chhugani and Kumar 2001]. For patches large on screen, however, it is faster to use triangle primitives. Current hardware is often not well optimized for rendering points. We expect the gains of using point-based tessellation to increase when such optimizations becomes routine. Preventing artifacts at the boundary of point-based elements and traditional triangles remains an open problem when the budget is limited. In our scheme we have chosen to allocate the same screen-space size to all elements of a patch. At the cost of more space, we could compute and store per element sizes. Another possibility of improvement lies in reducing (or eliminating) the online sampling density for invisible areas of the surface. The pre-sampling can also be made more rigorous by using disk-like, instead of sphere, object primitives and also by considering surface normals and other application dependent features.

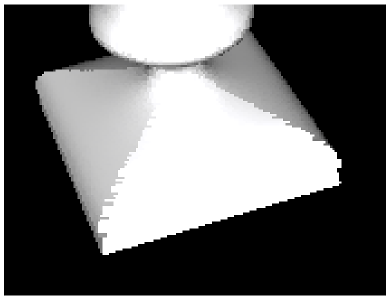
Figure 11: Visual artifact due to a large discontinuity in the normal values

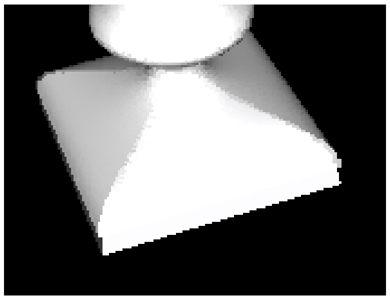
Figure 12: Reduction in artifact for $\varepsilon\ = 0.004$

Acknowledgments

This research was supported in part by an NSF CAREER award (CCR-9733827), an NSF infrastructure grant (CDA-9303080) and a Link Foundation Fellowship. We thank Shankar Krishnan for his maxima finding code and Budirijanto Purnomo and the anonymous reviewers for suggestions to improve the presentation. Models were courtesy of Lifeng Wang, the modeling group at University of British Columbia and XingXing Graphics Co. (Garden), David Forsey (Dragon), and Alpha 1 system (Coke and Goblet).

References

Abi-Ezzi, A., and Shirman, L. 1991. Tesselation of curved surfaces under highly varying transformations. In *Eurographics 1991 Proceedings*, 385–397.

Adamson, A., and Alexa, M. 2003. Ray tracing point set surfaces. In *Shape Modeling International*. (To appear).

Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., and Silva, C. 2001. Point set surfaces. In *IEEE Visualization*, 21–28.

BLINN, J. 1978. *Computer Display of Curved Surfaces*. PhD thesis, University of Utah.

CATMULL, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah.

CHEN, B., AND NGUYEN, M. 2001. Pop: A hybrid point and polygon rendering system for large data. In *IEEE Visualization*, 45–52.

CHHUGANI, J., AND KUMAR, S. 2001. View-dependent adaptive tesselation of spline surfaces. In *Symposium on Interactive 3D Graphics*, 59–62.

COHEN, J., ALIAGA, D., AND ZHANG, W. 2001. Hybrid simplification: Combining multi-resolution polygon and point rendering. In *IEEE Visualization 2001 Proceedings*, 37–44.

FILIP, D. 1986. Adaptive subdivision algorithms for a set of bézier triangles. *Computer-Aided Design 18*, 2, 74–78.

FLEISHMAN, S., COHEN-OR, D., ALEXA, M., AND SILVA, C. 2003. Progressive point set surfaces. *ACM Transactions on Graphics*. (To appear).

FUNKHOUSER, T., AND SÉQUIN, C. 1993. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proc. ACM SIGGRAPH*, 247–254.

GROSSMAN, J. P., AND DALLY, W. J. 1998. Point sample rendering. In *9th Eurographics Workshop on Rendering*, 181–192.

HOPPE, H. 1996. Progressive meshes. In *In Proc. SIGGRAPH 1996*, 99–108. http://research.microsoft.com/ hoppe/.

KAJIYA, J. 1982. Ray tracking parametric patches. *Computer Graphics 6*, 3, 245–254. (Proceedings Siggraph '82).

KALAIAH, A., AND VARSHNEY, A. 2002. Differential point rendering. *Rendering Techniques 2002*, 139–150.

KUMAR, S., MANOCHA, D., AND LASTRA, A. 1995. Interactive display of large scale nurbs models. In *Symposium on Interactive 3D Graphics*, 51–58.

KUMAR, S., MANOCHA, D., AND LASTRA, A. 1996. Interactive display of large nurbs models. *IEEE Transactions on Visualization and Computer Graphics 2*, 4, 323–336.

KUMAR, S., MANOCHA, D., ZHANG, H., AND HOFF, K. 1997. An accelerated walkthrough of large spline models. In *Symposium on Interactive 3D Graphics*, 91–101.

LANE, J., CARPENTER, L., WHITTED, J., AND BLINN, J. 1980. Scan line methods for displaying parametrically defined surfaces. *Communications of ACM 23*, 1.

LEVOY, M., AND WHITTED, T. 1985. The use of points as a display primitive. Technical Report TR-85-022, University of North Carolina, Chapel Hill.

NISHITA, T., SEDERBERG, T., AND KAKIMOTO, M. 1990. Ray tracking trimmed rational surface patches. *Computer Graphics 24*, 4, 337–345. (Proceedings Siggraph '90).

PFISTER, H., ZWICKER, M., VANBAAR, J., AND GROSS, M. 2000. Surfels: Surface elements as rendering primitives. In *In Proc. SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 335–342.

PRESS, W., TEUKOLSKY, S., VETTERLING, W., AND FLANNERY, B. 1993. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press.

ROSSIGNAC, J., AND BORREL, P. 1993. Multi-resolution 3d approximations for rendering complex scenes. *Geometric Modeling in Computer Graphics* (june), 455–465. Eds. B. Falcidieno and T.L. Kunii, Genova, Italy.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A multiresolution point rendering system for large meshes. In *In Proc. SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 343–352.

SCHROEDER, W. J. 1997. A topology modifying progressive decimation algorithm. In *IEEE Visualization 1997*.

SHANTZ, M., AND CHANG, S. 1988. Rendering trimmed nurbs with adaptive forward differencing. *Computer Graphics 22*, 4, 189–198.

STAMMINGER, M., AND DRETTAKIS, G. 2001. Interactive sampling and rendering for complex and procedural geometry. *Rendering Techniques 2001*, 151–162.

SZELISKI, R., AND TONNESEN, D. 1992. Surface modeling with oriented particle systems. *Computer Graphics 26*, 2. (Proceedings Siggraph '92).

VLASSOPOULOS, V. 1990. Adaptive polygonization of parametric surfaces. *Visual Computer 6*, 291–298.

WAND, M., FISCHER, M., PETER, I., AUF DER HEIDE, F. M., AND STRASSER, W. 2001. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *In Proc. SIGGRAPH 2001*.

WHITTED, J. 1978. A scan line algorithm for computer display of curved surfaces. *Computer Graphics 12*, 3, 8–13.

WHITTED, J. 1979. An improved illumination model for shaded display. *Computer Graphics 13*, 3, 1–14. (Proceedings Siggraph '79).

WITKIN, A., AND HECKBERT, P. 1994. Using particles to sample and control implicit surfaces. *Computer Graphics 28*, 3. (Proceedings Siggraph '94).

XIA, J. C., EL-SANA, J., AND VARSHNEY, A. 1997. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics 3*, 2, 171–183.

ZWICKER, M., PFISTER, H., VANBAAR, J., AND GROSS, M. 2001. Surface splatting. In *In Proc. SIGGRAPH 2001*, 371–378.