Eurographics Symposium on Geometry Processing (2003)
L. Kobbelt, P. Schröder, H. Hoppe (Editors)

Smooth Geometry Images

F. Losasso[1], H. Hoppe[2], S. Schaefer[3], and J. Warren[3]

[1] Stanford University, [2] Microsoft Research, [3] Rice University

Abstract

Previous parametric representations of smooth genus-zero surfaces require a collection of abutting patches (e.g. splines, NURBS, recursively subdivided polygons). We introduce a simple construction for these surfaces using a single uniform bi-cubic B-spline. Due to its tensor-product structure, the spline control points are conveniently stored as a geometry image with simple boundary symmetries. The bicubic surface is evaluated using subdivision, and the regular structure of the geometry image makes this computation ideally suited for graphics hardware. Specifically, we let the fragment shader pipeline perform subdivision by applying a sequence of masks (splitting, averaging, limit, and tangent) uniformly to the geometry image. We then extend this scheme to provide smooth level-of-detail transitions from a subsampled base octahedron all the way to a finely subdivided, smooth model. Finally, we show how the framework easily supports scalar displacement mapping.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Boundary Representations

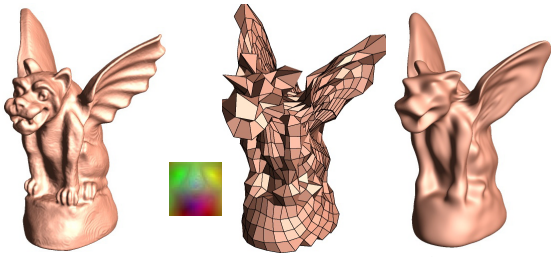
1. Introduction

Smooth surface representations are pervasive in off-line rendering systems [e.g. DeRose et al. 1998], because they avoid the rendering artifacts of faceted geometry, and reduce memory requirements by allowing a more compact description of shape. In this paper, we present a smooth surface scheme that can be efficiently evaluated using the fragment shaders in currently available graphics hardware.

Graphics Hardware. GPUs are evolving from fixed function pipelines to flexible programmable processors. Fragment shaders now support longer programs and a larger instruction set [Lindholm et al. 2001]. Whereas vertex shaders operate on each vertex independently, fragment shaders gather data using texture read operations. This flexibility can be applied to non-traditional uses, such as image-processing and non-photorealistic effects [Mitchell 2002].

Recently, the rasterization pipeline has begun to support floating-point types, thereby allowing geometry itself to flow through the GPU as a texture signal. For instance, even ray tracing computations can be implemented in the fragment shaders [Purcell et al. 2002]. Because GPU's contain several fragment shaders operating in parallel, there is significant opportunity for speed-up over a sequential CPU computation.

These fragment shaders gather data through texture accesses, where texture elements (texels) are organized into regular grids (1D, 2D, 3D). Signals over surfaces are thus typically resampled into 2D images parametrized onto the surface – so called texture atlases. In contrast, the surface geometry is traditionally represented using irregular (triangle) meshes.



original geometry image (33x33) C^1 subdivision

Figure 1: *Overview of our smooth surface representation.*

Geometry images. Managing the irregular meshes that arise in most smooth surface representations is a daunting task for a fragment shader program. To allow geometry to flow more naturally as a signal through the current GPU architecture, surfaces should be represented as regular grids if possible.

The geometry images introduced by Gu et al. [2002] represent an arbitrary surface using a regular 2D grid sampling. To permit this sampling, the surface is parametrized onto a square by first cutting the mesh along a network of edge paths. The generality of the cutting procedure permits treatment of surfaces with boundaries and of arbitrary genus. However, the drawback in our smooth surface setting is that the arbitrary topology of the cut can result in complicated continuity constraints along the boundary of the square domain.

For our purposes, we use a different parametrization for geometry images, introduced by Praun and Hoppe [2003] and

summarized in Section 4. Although the construction is limited to closed, genus-zero meshes, it uses a much simpler, topologically symmetric cut curve. Exploiting the symmetries of this cut, we develop a scheme to represent an *entire* closed surface as a *single uniform bicubic B-spline patch* (Figure 1). This patch is automatically C^2 on its interior. More importantly, the closed patch is also C^2 along the cut curve except at four extraordinary vertices. For these four vertices, we develop a simple linear constraint on the neighboring control points that ensures the closed patch is C^1 at these vertices.

Subdivision. Because of the tensor-product structure, the patch control points can be stored as a geometry image. To render the smooth patch, we subdivide this image using the GPU. All subdivision computations involve regular 3x3 masks. These masks are applied uniformly to the geometry image using simple fragment shader programs. After subdivision, we use a prototype RENDER_TO_VERTEX OpenGL extension to write the subdivided image to vertex buffers for final rendering. This scheme is easily generalized to include continuous level-of-detail varying from a base octahedron (generated by subsampling) to a highly subdivided, smooth model. We also present a simple extension of the scheme to support scalar displacement mapping.

To conclude, we compare the performances of CPU and GPU implementations of our scheme. In practice, the GPU implementation exhibits an order of magnitude increase in performance over the CPU version.

2. Previous work

Subdivision surfaces. Bolz and Schröder [2002] accelerate the rendering of subdivision surfaces by exploiting SIMD instructions on the CPU. Pulli and Segal [1996] evaluate Loop surfaces on geometry engines by pairing up adjacent triangles in the base mesh. The subdivided neighborhoods then have mostly a regular 2D structure, except for auxiliary arrays to handle arbitrary vertex valences. Bischoff et al. [2000] process triangles individually, and use fast forward differencing to reduce memory needs.

Polynomial surfaces. OpenGL has interface functions to evaluate polynomial curves and surfaces. These have been implemented within geometry engines on some SGI computers, but more commonly these operations are performed on the CPU by the driver. A similar DrawRectPatch interface is exposed in DirectX.

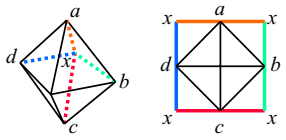
Curved PN-triangles. Vlachos et al. [2001] describe a hardware scheme that constructs a smooth patch for each triangle in a mesh. The patch is defined using only the 3 vertex positions and normals, and therefore requires no additional memory bandwidth. It significantly improves the silhouettes of coarse models, but unfortunately does not provide C^1 surface continuity.

Surface splines. Several schemes represent surfaces of arbitrary topology using a network of spline patches, e.g. [Loop 1994, Peters 2000]. Forsey and Bartels [1988] represent intricate planar and toroidal shapes using a single B-spline patch, but use adaptive refinement that requires a patch network for rendering. Note that our fragment-shader-based subdivision scheme could be used to efficiently evaluate each individual patch in such networks.

Unlike these previous techniques, our method exploits the powerful fragment shading pipeline of modern GPU's to efficiently evaluate the smooth surface. Also, we represent an entire spherical model conveniently as a single patch.

3. Geometry image representation

Overview. Our geometry images are built by parametrizing a given surface over an octahedral domain, and then unfolding the domain into a square image by introducing an "X-cut" centered at a vertex *x*.



The resulting boundary topology is extremely simple – each side of the square has reflection symmetry about its midpoint.

This parametrized square is then sampled using a regular 2D quadrilateral grid to create a geometry image. The entries in the geometry image *g* are indexed as

$$\begin{pmatrix} x & \cdots & a & \cdots & x \\ \vdots & & \vdots & & \vdots \\ d & \cdots & & \cdots & b \\ \vdots & & \vdots & & \vdots \\ x & \cdots & c & \cdots & x \end{pmatrix} = \begin{pmatrix} g_{1,1} & \cdots & \cdots & \cdots & g_{m-1,1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ g_{1,m-1} & \cdots & \cdots & \cdots & g_{m-1,m-1} \end{pmatrix} = g\,,$$

where the cut symmetry imposes the image boundary conditions $g_{i,1} = g_{m-i,1}$, $g_{1,i} = g_{1,m-i}$, $g_{m-1,i} = g_{m-1,m-i}$, and $g_{i,m-1} = g_{m-i,m-1}$.

We refer to the quadrilateral mesh formed by the grid *g* as the *opened mesh*. When its boundaries are fused back together, we refer to it as the *closed mesh*.

Note that the 4 corners of the opened mesh fuse into the same vertex *x*, and thus this vertex *x* has valence 4 in the closed mesh. Indeed, the only extraordinary (non-valence-4) vertices in the closed mesh are the 4 boundary midpoints (*a*, *b*, *c*, *d*), which have valence 2.

Bicubic surface. To define a bicubic surface from the geometry image, we begin by padding the image with a one-sample-wide border. These border samples are defined by traversing across the X-cut on the closed mesh to find neighboring interior samples. More precisely, the samples are obtained by transitively applying the rules:

$$\begin{aligned} g_{0,i} &= g_{2,m-i} & g_{m,i} &= g_{m-2,m-i} \\ g_{i,0} &= g_{m-i,2} & g_{i,m} &= g_{m-i,m-2} \end{aligned} .$$

The bicubic surface defined by this padded geometry image is

$$P(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{m} g_{i,j}\, N(s-i)\, N(t-j)\,,$$
$$s,t \in [1, m-1] \subset R\,,$$

where $N(s)$ is the uniform cubic B-spline basis function.

Due to the symmetric structure of the padding rules, the boundaries of this surface patch join along the X-cut to form a closed, genus-zero surface. Note that the parametrization for this surface patch is C^2 everywhere (including its boundaries). As a result, the corresponding surface patch is C^2 everywhere except for those points where the derivatives of the parametrization vanish identically.

Unfortunately, at the four cut vertices a, b, c, and d (which have valence 2 in the closed mesh), the parametrization has derivatives that are zero for any choice of g. For example, the cut vertex $a = g_{h,1}$ (with $h = m/2$) has a padded neighborhood of the form

$$\begin{pmatrix} \cdots & g_{h+1,2} & g_{h,2} & g_{h-1,2} & \cdots \\ \cdots & g_{h-1,1} & g_{h,1} & g_{h-1,1} & \cdots \\ \cdots & g_{h-1,2} & g_{h,2} & g_{h+1,2} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix}, \qquad (1)$$

Applying the tangent masks shown in Table 1 at a yield zero derivatives independent of the values of the control points. As a consequence, the resulting surface patch usually has a cusp at a (see Figure 2).

However, if the position of the cut vertex $g_{h,1}$ is constrained to lie at the centroid of its four neighbors in the mesh,

$$g_{h,1} = \tfrac{1}{4}\left(g_{h-1,1} + g_{h,2} + g_{h-1,2} + g_{h+1,2}\right), \qquad (2)$$

the cusp is suppressed. In particular, the perturbed surface is C^1. Figure 2 shows the smoothing effect of this perturbation. The appendix contains a short analysis of the smoothness of the perturbed scheme.

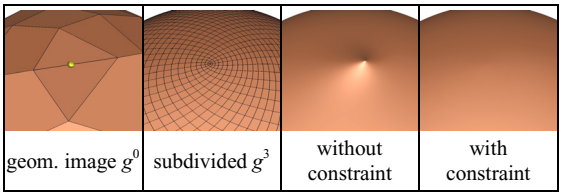


Figure 2: *Subdivision behavior near valence-2 vertex.*

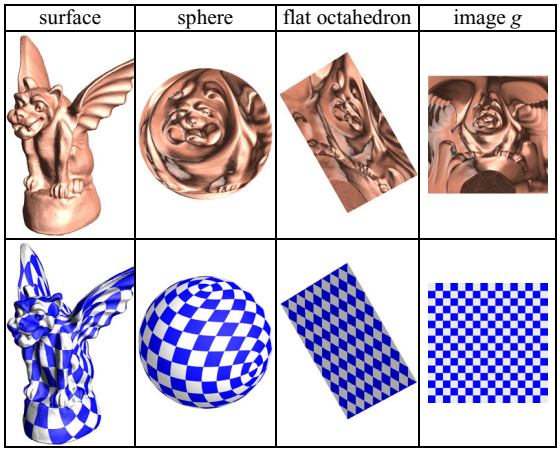


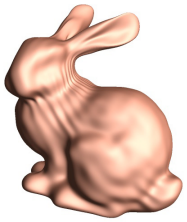
Figure 3: *Geometry image parametrization.*

4. Geometry image creation

Parametrization. As described in [Praun and Hoppe 2003], surface parametrization proceeds in 3 steps (Figure 3). The mesh is first parametrized over a sphere. This parametrization minimizes a stretch metric to reduce undersampling of complicated geometric shapes. Second, the sphere is mapped onto a flattened octahedron, again using a stretch-minimizing map. Finally, the octahedron is unfolded on a square geometry image using the X-cut. The benefit of using a flattened octahedron as the domain is that it unfolds isometrically.

Fitting. Next, we use this parametrization to construct a geometry image whose associated surface patch $P(s,t)$ accurately fits the original surface. Of course, we cannot simply use point samples of the original surface as the control points for the bicubic patch $P(s,t)$ since the B-spline patches are approximating in nature. Instead, we sample the original surface on a fine grid using the parametrization and then compute the bicubic surface patch $P(s,t)$ that best fits the data in a least-squares sense. This optimization problem is a linear one that we solve using sparse conjugate-gradients. During optimization, we effectively eliminate the 4 linear constraints at the cut vertices $\{a, b, c, d\}$ by removing these vertices from the set of free variables – they are always defined as the centroids of their neighbors.

The bicubic surface that results from this fitting optimization sometimes suffers from ripples (undulations), as shown on the right. To obtain a smoother surface (as in Figure 7), we add a simple fairness functional that measures the squared distance between each control point and the centroid of its neighbors. (Alternatively, we could integrate fairness of the limit surface as in [Halstead et al. 1993].)



The automatic preprocess to generate the geometry images takes 10-27 minutes for the models (70K-200K faces) in this paper on a 3GHz Pentium4 PC. The bottleneck is the spherical parametrization step.

5. Subdivision using Graphics Hardware

Bicubic subdivision. Given the closed mesh g^0 associated with geometry image g, the B-spline surface patch $P(s,t)$ is the limit g^∞ of a subdivision process $g^0, g^1, g^2, g^3, \ldots$ applied to the initial mesh g^0 (see Figure 4). Each subdivision step $g^{k+1}=S[g^k]$ splits all quadrilaterals into 4 children, inserting new vertices on each edge and face. These new vertices in g^{k+1} are positioned as local affine combinations of vertices g^k.

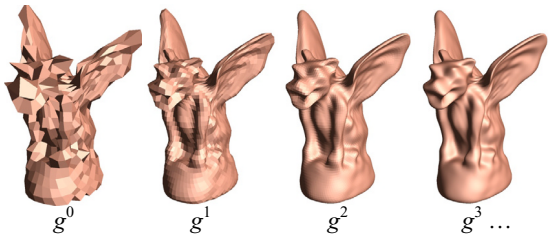


Figure 4: *Three steps of bicubic subdivision.*

$\frac{1}{2}\begin{pmatrix}1\\1\end{pmatrix}\quad \frac{1}{2}(1\ 1)\quad \frac{1}{4}\begin{pmatrix}1&1\\1&1\end{pmatrix}$	$\frac{1}{16}\begin{pmatrix}1&2&1\\2&4&2\\1&2&1\end{pmatrix}$	$\frac{1}{36}\begin{pmatrix}1&4&1\\4&16&4\\1&4&1\end{pmatrix}$	$\frac{1}{6}\begin{pmatrix}-1&0&+1\\-4&0&+4\\-1&0&+1\end{pmatrix}$
splitting L	averaging A	limit E	tangent T_s

Table 1: *Masks for bicubic subdivision. (T_t is T_s transposed.)*

The subdivision operator S can be decomposed into two simpler steps, a *splitting* step L, and an *averaging* step A [e.g. Lane and Riesenfeld 1980]. For B-spline surfaces, the splitting operator L performs bilinear quadrisection of the faces, positioning new vertices in g^{k+1} at the centroids of their neighboring 2 or 4 vertices in g^k. The averaging operator A applies a linear filter kernel to all vertices. These linear combinations are denoted by the masks shown in Table 1.

After a finite number of subdivision steps, the vertices of the subdivided surface g^k can be sent to their limit positions p^k on $P(s,t)$ by evaluating a *limit* mask $E[g^k]$. Perturbing the vertices to their limit position lets the mesh more closely approximate the smooth surface $P(s,t)$. The normals n^k to $P(s,t)$ at p are the cross product of two tangent vectors computed by applying the tangent masks T_s and T_t to g^k. (For shading, n^k should be unit length, so the cross product is normalized via the function *unit*.)

As a concrete example, to render the bicubic surface subdivided 3 times, we compute its limit positions p^3 and their normals n^3 as:

$$g^1 = A[L[g^0]],\quad g^2 = A[L[g^1]],\quad g^3 = A[L[g^2]],$$
$$p^3 = E[g^3],\quad n^3 = unit[T_s[g^3]\times T_t[g^3]]\ .$$

GPU-based implementation. We store geometry images in off-screen pixel buffers containing three 24-bit floating-point channels. We have found that 24 bits is sufficient precision for all the necessary arithmetic. The pixel buffers are configured to support both read and write operations.

All the masks in Table 1 are implemented using fragment shader programs. As an example, let us consider the limit operation $g' = E[g]$. The buffer g is bound as the source texture, and the buffer g' is assigned as the rendering destination. A square is drawn that covers all the destination pixels. The rasterizer thus generates fragments to be "shaded" by the fragment processor for every pixel. At each pixel, the fragment program reads the appropriate texels from the source texture (i.e. vertices in g) and applies the appropriate mask. The resulting floating-point coordinates are written into the destination buffer g'.

Given an image g^k of size $[0..m]^2$, the splitting operation $L[g^k]$ produces a new image of size $[0..2m]^2$ by applying the masks in Table 1. Note that this process is equivalent to bilinearly filtering g^k on the grid of half integers ranging over $[0..m]^2$. Since current graphics cards do not have dedicated hardware to perform this type of filtering on floating-point textures, we must perform the operation in a fragment program.

Given integer texture coordinates $\{i, j\}$ in the range $[0..2m]^2$, our goal is to bilinearly sample the image g^k at the half-integer texture coordinates $\{i/2, j/2\}$. At first glance, this sampling process might appear to be non-uniform (and thus difficult to implement in a fragment shader) since the expressions used in computing the new texture depends on whether i and j are odd or even. However, the bilinear interpolation can be treated as a uniform operation by using the *floor* operation in the fragment shader. If $\{I, J\}$ is the integer part of $\{i/2, j/2\}$ and $\{\alpha, \beta\}$ is the fractional part, the $\{i, j\}$th entry of $L[g^k]$ is exactly

$$(1-\alpha)(1-\beta)g^k_{I,J} + \alpha(1-\beta)g^k_{I+1,J} + (1-\alpha)\beta g^k_{I,J+1} + \alpha\beta g^k_{I+1,J+1}$$

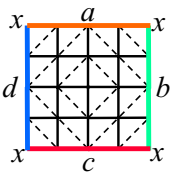
After applying the averaging mask A to $L[g^k]$, we crop the resulting image on all sides by one pixel to create the subdivided image g^{k+1} with dimensions $[0..2m\text{-}2]^2$. The cropped samples are no longer needed since border padding maintains a constant size of one throughout subdivision.

While the expression for computing the unit normals n^k may appear complicated, both tangent masks refer to the same 8 neighbor samples, and both the cross product and normalization require only a few instructions.

Having obtained limit points p^k and normals n^k, we re-cast these off-screen buffers as a vertex stream. This ability to interpret floating-point images as vertices is made possible by a prototype RENDER_TO_VERTEX extension to OpenGL. The alternative, reading the buffer back into host memory and sending the vertices back to the graphics card, would be prohibitively expensive. Finally, we feed the vertex stream through the GPU to render the subdivided surface.

For efficiency, we pre-allocate image buffers of the appropriate sizes to store $\{g^1, g^2, \ldots, g^k, p^k, n^k\}$ where k is the desired number of subdivision levels. Several models can share the same set of subdivision buffers since these are only used to store temporary information.

Continuous level-of-detail. To maintain real-time performance, the number of subdivision levels k can be adapted for each model based on factors such as viewing distance, model importance, and overall scene complexity [Funkhouser 1993]. Even though we send vertices to their limit positions, instantaneously changing the subdivision level creates visual "pops." Our solution is to smoothly transition between levels using linear interpolation.



Given a *continuous* level of subdivision $k+\alpha$ with $0<\alpha\le 1$, we wish to compute a mesh $p^{k+\alpha}$ that varies continuously as a function of α between p^k and p^{k+1}. As a preliminary to blending, we first triangulate p^k and p^{k+1} such that all dotted diagonals of the resulting triangulation are oriented as shown to the right. (This choice of triangulation avoids the possibility of having the diagonals of quads flip during subdivision.) Next, we linearly subdivide the triangles of p^k. Note that this subdivision yields a mesh with the same connectivity as p^{k+1} while retaining the geometric shape of p^k. Finally, we linearly interpolate between these two meshes as a function of α. To save texture memory, rather than computing p^k=E[g^k] as an image, we instead gather the samples of p^k by subsampling p^{k+1}. This also reduces the number of texture reads in the linear interpolation step.

We employ a similar blending process to construct a set of normals $n^{k+\alpha}$ for the mesh $p^{k+\alpha}$. Given the normals n^k for p^k, we first linearly subdivide these normals. To obtain blended normals, we linearly interpolate between these linearly subdivided normals and n^{k+1}. In a purely diffuse shading model, applying Gouraud shading to these linearly subdivided normals over the linear subdivision of p^k reproduces the same pixel intensities generated by Gouraud shading p^k using the normals n^k. As a result, the diffuse shading of the model exhibits no "popping" of intensity.

Unfortunately, specular vertex shading is nonlinear due to the use of exponentiation in the shading model. To obtain smooth transitions while performing specular shading, we instead perform Phong shading and linearly interpolate unit normals along edges of the mesh. Fortunately, the latest graphics hardware supports performing the Phong calculation in a fragment shader.

Subsampling images. When the rendered model generates only a few pixels (such as when the model is distant), fewer polygons are required to accurately render the model. In particular, the mesh g^0 may already have more polygons than necessary. The simple grid structure of the geometry image allows for convenient subsampling, which can be viewed as a form of "negative" subdivision. If the unpadded geometry image g^0 has size $(2^{k+1}+1)^2$, repeated subsampling of the interpolating mesh p^0 yields a sequence of increasingly simplified interpolating meshes p^{-1}, p^{-2}, …, that terminates in a base octahedron p^{-k}. Note that the use of subsampling allows the previous level-of-detail blending to work for these meshes without modification. Figure 5 shows an example of this subsampling for the gargoyle. We can thus smoothly transition between these approximations in real-time.

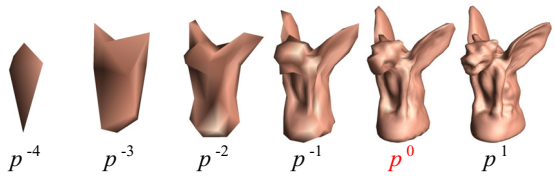


Figure 5: *Subsampling of limit mesh for "negative" subdivision.*

Displacement mapping. Detail at level k can be added to the smooth surface using a scalar displacement map d^k. The image d^k typically has higher resolution than the control mesh g^0. Because it only requires one channel, and that one channel can be quantized to fewer than 24 bits, it requires much less space than storing detail in the geometry image itself. With the geometry image representation, the parametrization of d^k over g^0 becomes implicit, thus removing the need for texture coordinates. Applying a displacement map d^k in the fragment shader consists of computing $p^k + d^k n^k$. An example is shown in Figure 6.

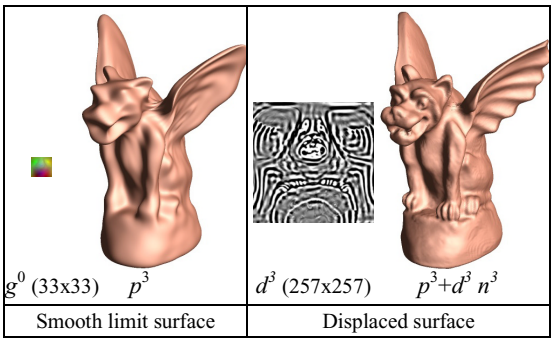


Figure 6: *Scalar displacement mapping applied to limit surface.*

Implementation details. The precise breakdown of the evaluation algorithm into fragment shading passes depends on the capabilities of the GPU. On the currently available ATI Radeon 9700 and 9800, we have separate shading passes for linear subdivision, averaging, exact limit position, exact limit normals, and the transitional blending based on subsampling. If a larger number of texture reads were available in future GPU's, the full subdivision step (linear subdivision plus averaging) could be performed in one operation, and perhaps the limit operation could also be included.

6. Results

To gather performance statistics, we implemented two separate programs that generate subdivided surfaces. The first is an efficient CPU implementation of subdivision (not using SSE or other SIMD instructions) running on a high-end personal computer (Intel P4 2.5GHz). The second program is a GPU implementation that subdivides the model using the technique described in the paper. The GPU implementation is tested on both the ATI Radeon 9700 and the ATI Radeon 9800. The Radeon 9800 runs at a higher clock speed, and has slightly more bandwidth than the Radeon 9700, but is otherwise identical for our purposes. Our GPU implementation forces the graphics card to flush the rendering pipeline using the OpenGL command glFinish() both before and after the subdivision process. We record the elapsed time between the two pipeline flushes, thus ensuring that the GPU is not performing other tasks when the timing starts, and that the GPU is done with the subdivision calculation when the timing ends. Because glFinish() has a large overhead and the drivers used in our implementation are preliminary, we expect better performance in the future.

Since each new level of subdivision generates a model with approximately four times as many vertices as the previous level, one would expect subdivision times to scale similarly. The CPU implementation behaves as expected. The GPU implementation, however, has a non-trivial setup overhead for the source and destination buffers, so it does not take four times longer to subdivide to one more level. The actual subdivision computation dominates the setup overhead only for large geometry images or with many subdivision levels.

Performance analysis of the GPU-based implementation is difficult due to the fact that the inner workings of the graphics card are not published. Using the information available to us, we estimate the utilization rates of GPU bandwidth and computation (see Table 2), and conclude that our method is currently compute-limited rather than bandwidth-limited. This conclusion is supported by three observations:

- The estimated consumed bandwidth is less than 15% of the theoretical maximum bandwidth available.
- The estimated number of fragment shader cycles used for subdivision coincides with the maximum possible, given the GPU clock speeds and number of shader units.
- The increase of 17% in clock speed and 10% in bandwidth between the two cards yields an increase of almost 15% in performance for our application.

We want to stress that these resource utilization numbers (for both bandwidth and computation) should be treated as very rough estimates. Indeed, bandwidth utilization is complicated by factors such as unknown caching schemes and bandwidth use by other buffers or by other GPU tasks. Similarly, computation utilization is complicated by issues such as unknown optimizations to the fragment programs by the graphics driver and unknown cycle counts for individual instructions.

Overall, the GPU implementation is up to one order of magnitude faster than our CPU implementation (Table 2).

As another CPU comparison number, Bolz and Schröder [2002] report a rate of almost 20 million triangles per second on a Pentium 4 when subdividing a 384-quad mesh to 6 levels. They exploit SIMD instructions and maximize memory cache coherency. It is important to note that their method only calculates the tangents to the surface, and not the actual surface normal as in our implementation. Computing the surface normal requires an additional vector cross-product and normalization, which incur a significant cost on the CPU.

Rendering is currently implemented using triangle strips that index into the computed vertex array. The current API requires us to send these indices every frame, even though they are constant. The indices follow a simple grid pattern, so they could easily be computed automatically in future GPU's.

		Subdivision Level (*k*)			
		1	**2**	**3**	**4**
GPU (R 9700)	Time (ms)	1.03	1.38	3.23	10.4
	Triangle rate (MΔ/s)	7.2	21.2	36.0	44.6
	Bandwidth utiliz. (%)	2.2	6.2	10.5	13.1
	Computation utiliz. (%)	14.3	46.7	81.1	101
GPU (R 9800)	Time (ms)	0.93	1.30	2.91	9.06
	Triangle rate (MΔ/s)	8.0	22.5	40.0	51.1
	Bandwidth utiliz. (%)	2.1	6.0	10.6	13.6
	Computation utiliz. (%)	13.6	42.4	77.0	99.0
CPU	Time (ms)	1.3	5.5	22.8	90.7
	Triangle rate (MΔ/s)	5.9	5.3	5.1	5.1

Table 2: *Timing results for subdividing a geometry image of size 31x31, including computation of limit positions and normals.*

7. Summary and future work

In summary, we have developed a scheme for modeling a closed genus-zero surface as single bicubic surface patch. This patch is C^2 everywhere except for four extraordinary vertices where the patch is C^1. We have demonstrated a simple approach to subdividing this surface patch that includes smooth level-of-detail transitions and displacement mapping. By representing the patch as a geometry image, we are able to realize all these computations using the GPU fragment shader pipeline.

Our method takes advantage of the widening gap between the processing powers of the GPU and CPU. In the future, we believe that other computations such as more sophisticated rendering techniques or computational simulations may profit from being computed on the GPU. The regular structure of geometry images will be a key to performing these computations on the GPU.

In terms of modeling and rendering smooth surfaces, future work will involve handling more general types of surface models such as those with handles and boundaries. Unfolding these surfaces involves cut curves with more complicated topologies that lead to extraordinary vertices of arbitrary valence on the boundary of the geometry image. Introducing creases into the interior of the geometry image is also an area of future work. Crease curves and corners would allow a larger class of models to be represented with geometry images.

Currently, parametrization misalignment in areas of high curvature sometimes gives rise to surface rippling. This may be overcome in the future by optimizing the parametrization.

We have shown how to perform some basic geometry manipulation in graphics hardware, but in the future one can envision using the hardware for many other types of geometric algorithms.

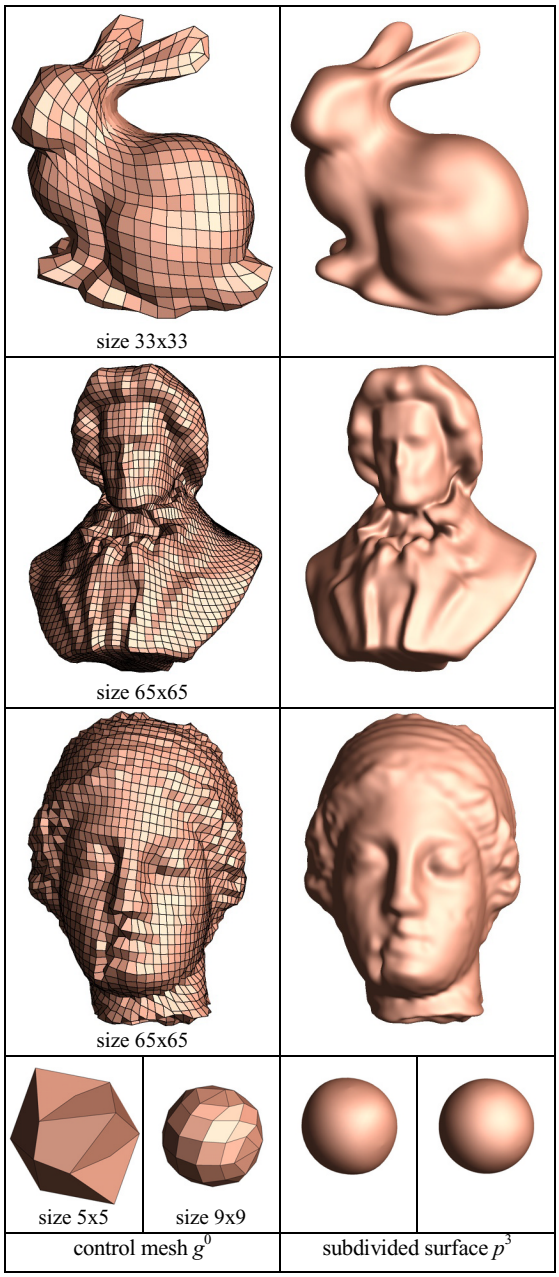


Figure 7: *Additional examples of subdivided surfaces.*

References

BISCHOFF, S., KOBBELT, L. P., AND SEIDEL, H.-P. 2000. Towards Hardware Implementation of Loop Subdivision. *SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 41–50.

BOLZ, J. AND SCHRÖDER, P. 2002. Rapid Evaluation of Catmull-Clark Subdivision Surfaces. *Web3D 2002 Symposium.*

CATMULL, E., AND CLARK, J. 1978. Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes. *Computer-Aided Design*, 10(6), pp. 350–355.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision Surfaces in Character Animation. *SIGGRAPH 1998*, pp. 85-94.

FORSEY, D. AND BARTELS. R. 1988. Hierarchical B-spline Refinement. *SIGGRAPH 1988*, pp. 205-212

FUNKHOUSER, T., AND SEQUIN, C. 1993. Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments, *SIGGRAPH 1993*, pp.247-254.

GU, X., GORTLER, S., AND HOPPE, H. 2002. Geometry Images. *SIGGRAPH 2002*, pp. 355-361.

HALSTEAD, M., KASS, M., AND DEROSE, T. 1993. Efficient, Fair Interpolation using Catmull-Clark Surfaces. *SIGGRAPH 1993*, pp. 35-44.

LANE, J. AND RIESENFELD, R. A Theoretical Development for the Computer Generation and Display of Piecewise Polynomial Functions. *Transactions on Pattern Analysis and Machine Intelligence*. 2(1), pp. 35-46.

LINDHOLM, E., KILGARD, M., AND MORETON, H. 2001. A User-Programmable Vertex Engine. *SIGGRAPH 2001*, pp. 149-158.

LOOP, C. 1994. A G^2 Triangular Spline Surface of Arbitrary Topological Type. Computer-Aided Geometric Design, 11(3), pp. 303-330.

MITCHELL, J., 2002. Image Processing with Direct3D Pixel Shaders. In *Vertex and Pixel Shaders Tips and Tricks*, Wolfgang Engel editor, Wordware.

PETERS, J. 2000. Patching Catmull-Clark Meshes. *SIGGRAPH 2000*, pp. 255-258.

PETERS, J. AND UMLAUF, G. 2001. Computing Curvature Bounds for Bounded Curvature Subdivision. *Computer-Aided Geometric Design*, 18, pp. 455-461.

PRAUN, E. AND HOPPE, H. 2003. Spherical Parametrization and Remeshing. *SIGGRAPH 2003*.

PULLI, K., AND SEGAL, M. 1996. Fast Rendering of Subdivision Surfaces. *Eurographics Rendering Workshop 1996*, pp. 61-70.

PURCELL, T., BUCK, I., MARK, W., AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3), pp. 703-712.

REIF, U. 1995. A Unified Approach to Subdivision Algorithms near Extraordinary Points. *Computer-Aided Geometric Design*, 12, pp. 153-174.

VLACHOS, A., PETERS, J., BOYD. C., AND MITCHELL, J. 2001. Curved PN Triangles. *ACM Symposium on Interactive 3D Graphics*, pp. 159-166.

WARREN, J., AND WEIMER, H. 2001. *Subdivision Schemes for Geometric Design*, Morgan Kaufmann.

Appendix: Smoothness at vertices *a, b, c, d*

Our task is to show that applying bicubic subdivision to the perturbed geometry image yields a smooth (C^1) surface at the cut vertices *a*, *b*, *c*, *d*. We first analyze the smoothness of the unperturbed scheme at the cut vertex *a*. Following Equation 1, the bicubic subdivision process for the one-ring of the cut vertex *a* has the form (with $h = \frac{m}{2}$)

$$\begin{pmatrix} g_{m,1}^{k+1} \\ g_{m-1,1}^{k+1} \\ g_{m,2}^{k+1} \\ g_{m-1,2}^{k+1} \\ g_{m+1,2}^{k+1} \end{pmatrix} = \frac{1}{32}\begin{pmatrix} 18 & 6 & 6 & 1 & 1 \\ 12 & 12 & 4 & 2 & 2 \\ 12 & 4 & 12 & 2 & 2 \\ 8 & 8 & 8 & 8 & 0 \\ 8 & 8 & 8 & 0 & 8 \end{pmatrix}\begin{pmatrix} g_{h,1}^{k} \\ g_{h-1,1}^{k} \\ g_{h,2}^{k} \\ g_{h-1,2}^{k} \\ g_{h+1,2}^{k} \end{pmatrix}.$$

This subdivision matrix *S* has eigenvalues $1, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{16}$. Unfortunately, the unperturbed scheme is only C^0 due to the fact that the subdominant eigenvalue $\frac{1}{4}$ has multiplicity three (as opposed to multiplicity two in smooth schemes). Technically, the three eigenfunctions associated with the eigenvalues $\frac{1}{4}$ provide three independent tangent directions at *a* and allow a cusp in the resulting limit surface (see Chapter 8 of [Warren and Weimer 2001] for details.)

However, note that one of the right eigenvectors of S corresponding to $\frac{1}{4}$ is exactly (-4,1,1,1,1). If we perturb $g_{h,1}$ as done in Equation 2,

$$g_{h,1}^{k} = \frac{1}{4}\left(g_{h-1,1}^{k} + g_{h,2}^{k} + g_{h-1,2}^{k} + g_{h+1,2}^{k}\right),$$

the effect on the subdivision process is to suppress the contribution of the eigenfunction corresponding to this eigenvector in the limit surface at *a*. More concretely, the subdivided control points again satisfy the same centroid relation

$$g_{m,1}^{k+1} = \frac{1}{4}\left(g_{m-1,1}^{k+1} + g_{m,2}^{k+1} + g_{m-1,2}^{k+1} + g_{m+1,2}^{k+1}\right).$$

Since the control point g_h^k is dependent in this new scheme, we can derive a new subdivision matrix of size four involving the remaining four independent control points of the form

$$\begin{pmatrix} g_{m-1,1}^{k+1} \\ g_{m,2}^{k+1} \\ g_{m-1,2}^{k+1} \\ g_{m+1,2}^{k+1} \end{pmatrix} = \frac{1}{32}\begin{pmatrix} 15 & 7 & 5 & 5 \\ 7 & 15 & 5 & 5 \\ 10 & 10 & 10 & 2 \\ 10 & 10 & 2 & 10 \end{pmatrix}\begin{pmatrix} g_{h-1,1}^{k} \\ g_{h,2}^{k} \\ g_{h-1,2}^{k} \\ g_{h+1,2}^{k} \end{pmatrix}.$$

The subdivision matrix for this perturbed scheme has a spectrum of the form $1, \frac{1}{4}, \frac{1}{4}, \frac{1}{16}$. To determine the smoothness of this subdivision scheme, we must extend the subdivision matrix to the two-ring of *a* due to the support of the B-spline basis functions. The resulting extension produces eigenvalues of magnitude less than or equal to $\frac{1}{8}$. Therefore, the two subdominant eigenfunctions corresponding to $\frac{1}{4}$ define two tangent directions and the perturbed scheme now has a well-defined tangent plane at *a*. To complete the proof of smoothness, we use the techniques of [Reif 95] to show that the characteristic map for this scheme is regular and injective (Figure 8), and thus the scheme is C^1 at *a*.

Finally, as shown in [Peters and Umlauf 2001], subdivision schemes with the spectrum $1, \alpha, \alpha, \alpha^2, \ldots$ have bounded curvature at the extraordinary vertex. Unfortunately, extending the subdivision process to the two-ring of *a* introduces several new eigenvalues of size $\frac{1}{8}$ to the spectrum of the extended subdivision matrix. Thus, the subdivision scheme while smooth has unbounded curvature at *a*.

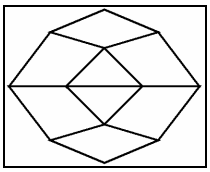


Figure 8: *The characteristic map at an extraordinary vertex of valence two.*