

Subdividing Reality

Employing Subdivision Surfaces for Real-time Scalable 3D

Stephen Junkins and Allen Hux

stephen.junkins@intel.com & allen.hux@intel.com

Intel Architecture Labs, Intel Corporation

Introduction

Vast choices of CPUs, graphics cards, instruction sets, memory speed, bus technologies, graphics drivers, and graphics APIs have created a tremendous heterogeneity of a consumer graphics platforms on which consumers may run real time 3D games. Building applications that deliver compelling visual experiences across a wide range of platforms is referred to as the *Scalability Problem* [Rosenzweig97]. Solving the scalability problem requires using modeling and rendering technologies that can dynamically target 3D content to deliver the best visual experience possible on a particular rendering platform.

One important axis of 3D content scalability is the number of polygons (typically triangles) used to render a particular 3D model within a game's virtual world. The importance of polygon scalability will become increasingly important, as higher performance graphics cards shift rendering bottlenecks from pixel fill rate to triangle throughput. Generally, the more triangles allocated to rendering a particular model, the greater the possible visual fidelity. Implicit surfaces and parametric surfaces such as NURBS have been long hailed as *the* solution to the content scalability problem while also providing a more organic modeling capability. However, NURBS have yet to take the industry by storm for many reasons: NURBS tools are just beginning to mature, NURBS surface trimming and crack filling are difficult problems, and adaptive tessellation of NURBS surfaces is not fine-grained. Considerable industry and academic resources *have been* invested in creating sophisticated triangle-based 3D content modeling tools and high-performance triangle based rendering hardware. For the moment, triangles are a very mature modeling paradigm and it is important that any polygon scalability solution fully leverage this maturity.

Subdivision surfaces are a powerful surface modeling technique for creating scalable 3D content. They provide the organic "look and feel" of a NURBS surface, but take full advantage of triangle based modeling tools and triangle based rendering hardware. Furthermore they are well suited to real-time adaptive evaluation and triangulation of 3D content. In this paper will provide a brief background of subdivision surfaces, demonstrate several scalable applications of subdivision surfaces, and then concentrate on describing solutions to many challenging implementation issues often neglected by the academic literature.

Subdivision Surface Taxonomy and Terms

Subdivision surfaces were first introduced in missing words [Catmull87]. In recent years there has been a research renaissance that has addressed many open problems. However, software developers have only recently begun employing them in commercial products. For example, Nichimen Graphics's uses subdivision surfaces in their Mirai* Modeling tool and more famously, Pixar uses Catmull-Clark surfaces in their custom animation tools used for films like Gerri's Game* and Toy Story 2.* [DeRose98][Porter00].

Given a simple control mesh of polygons, subdivision surfaces implementations algorithmically generate a more finely tessellated surface that yields a smoother visual representation. The subdivision algorithms are local in nature and typically yield new vertices by computing weighted linear combinations of a small neighborhood of vertices from the control mesh. The term subdivision comes from the fact that we subdivide each coarse polygon into a collection of smaller polygons that more smoothly model the intended surface.

There is a rich taxonomy of subdivision algorithms [Schröder98]. The algorithms are classified as *approximating* or as *interpolating* based on the kind of surface they produce, relative to their control mesh. Algorithms such as Catmull-Clark and Loop's Scheme [Loop87] are said to be approximating subdivision schemes. Working with these surfaces is similar to working with NURBS surfaces in that the control mesh is spatially separate from the computed surfaces. Approximating subdivision surfaces are said to approximate the control mesh. Algorithms such as the Butterfly Scheme and the Modified Butterfly Scheme are said to be *interpolating subdivision schemes*. In these schemes, all of the vertices of the control mesh also lie on the surface yielded by subdivision. The subdivision surface interpolates the control mesh.

For both interpolating and approximating subdivision schemes, there is a theoretical "final surface" called the *limit surface*. The limit surface can be achieved by infinite application of the subdivision algorithm. However, for the purpose of building virtual environments with smooth modeling, satisfactory approximations of the limit surfaces can be achieved with just a few applications of the chosen subdivision scheme.

The subdivision surface implementation discussed in this paper, implements interpolating subdivision using the Butterfly Scheme. Several characteristics make this scheme desirable for use in real-time 3D applications:

*Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owner's benefit, without intent to infringe.

1. The butterfly scheme is interpolating. Because the vertices of the control mesh actually lie on the limit surface of the butterfly scheme, we can trivially use the control mesh for rendering the coarsest representation of our model. Furthermore, we may be able to trivially enhance pre-existing content by using those models as control meshes. (Approximating schemes often require post-subdivision re-scaling.)
2. The butterfly scheme is triangle-based. Triangle-based control meshes can be easily authored using any number of 3D authoring tools. The subdivision surface itself will also be composed of triangles and is easily rendered.
3. The butterfly scheme employs a very simple local neighborhood so neighbor gathering is not extensive.
4. Sharp (non-smooth) surface features are more easily modeled with interpolating schemes than with an approximating scheme.

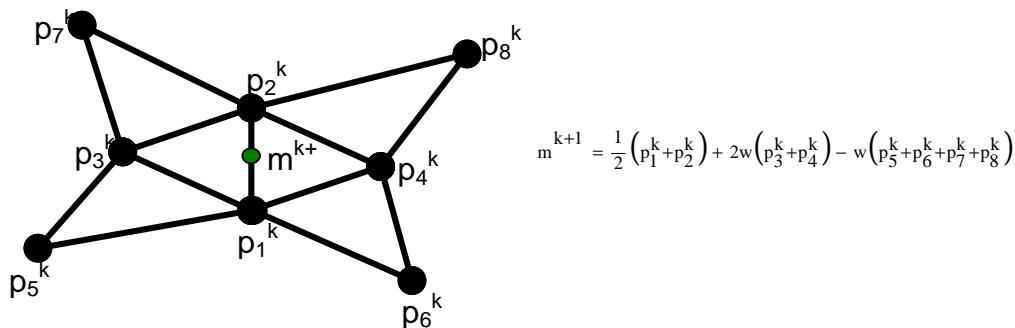


Figure 1. The Butterfly Scheme's local neighborhood mask and its linear combination rule. m^{k+1} is the midpoint to be generated for subdivision level $k+1$. To generate this midpoint, we'll need all of the vertices $p_1^k - p_8^k$ from subdivision level k . The equation shows how these vertices are combined. The factor w is the *global tension parameter* and controls the degree to which the kernel interpolates the surface.

Figure 1 shows the local neighborhood mask of triangles and vertices necessary for each butterfly subdivision computation. To subdivide a single triangle, we apply this mask and its associated linear combination rule three times, once to each edge of the triangle, yielding four new triangles. To subdivide an entire mesh one level, we must apply the mask to every edge in the mesh and re-triangulate each triangle into four new triangles.

It is worth noting that the modified butterfly scheme does provide stronger surface continuity guarantees for certain mesh connectivity patterns, but at the expense of gathering a larger local neighborhood. Such an expense may not matter for author-time or load-time subdivision surface applications, but it is a critical consideration for real-time applications. It is also worth noting that all interpolating schemes can exhibit some small surface *wiggling* between subdivision levels due to the nature of the interpolation. The wiggling can be somewhat alleviated by adding more vertices to the control mesh, to further constrain the interpolation.

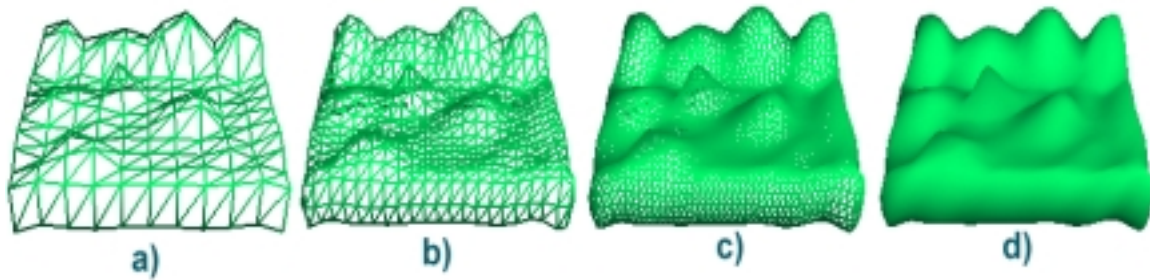


Figure 2. The Butterfly Subdivision Scheme applied to a simple surface control mesh. Because the scheme is an interpolating subdivision scheme, each denser subdivision interpolates (includes) all of the points from the less dense subdivision levels. a) No subdivision; this is the base control mesh. b) 1 level of subdivision. c) 2 levels of subdivision. d) 2 levels of subdivision shaded using computed surface normals and a single light source.

Subdivision Surface Applications

We now list several possible applications of subdivision surface algorithms relevant for real-time 3D application development.

Load Time Content Enhancement: Subdivision surfaces can be used by a 3D software application to subdivide a base mesh either uniformly or adaptively to produce a mesh of a higher resolution. The depth of subdivision can be chosen relative to the detected graphics performance of the executing platform. The refined mesh could then be used by the 3D application as it would any other mesh.

Transmission and Compression: Subdivision surfaces can be used as a powerful tool for compressing 3D models for Internet transmission. In this scenario, a server transmits the control mesh to a smart client. The client uses a subdivision surface algorithm to "fill out" the control mesh by refining and smoothing it. A typical refinement may require only 3 or 4 levels of uniform subdivision. The amount of refinement can be customized on the rendering performance of the client. Also, traditional mesh compression techniques may be applied to the base mesh for even greater mesh compression.

Enhancement of Existing 3D Content: The primary requirement of the control mesh is that major surface features are represented by coarse polygons that provide a rough outline of the features. Many pre-existing models from real-time 3D applications already meet this criterion. Subdivision surfaces can be used at load time to add smoother surface detail to these models, to make them more appropriate for rendering on higher performance processors.

Adaptive Subdivision of Static Meshes: Static meshes are meshes whose vertex positions do not change between rendering frames. These include environment meshes such as terrain and objects on the terrain, as well as stationary buildings and their interiors. Adaptive subdivision works to dynamically generate a mesh that provides the same visual quality as a uniformly subdivided mesh, but with a smaller triangle count requiring fewer rendering sources. Adaptive subdivision adds surface detail only where needed based on a subdivision metric. Example

subdivision metrics might be a distance metric, or a screen space metric, or perhaps a specular lighting effect quality metric.

Adaptive Subdivision of Dynamic Meshes: Dynamic meshes are meshes whose vertex positions are transformed dynamically and do not exhibit good or easily predicted frame-to-frame coherence. Examples include the skins of bones-based character animations or perhaps meshes modeling cloth. It is important to note that one cannot cache subdivision computations between frames for dynamic meshes. In fact, all subdivision computations must be recomputed every frame. If computing resources are available, one can use adaptive subdivision to enhance the animated model. For instance, an adaptive metric could add additional surface at places where joints cause extreme "pinching" on the coarse mesh.

Data Structures and Computational Efficiency

There are some important data structures required for implementing subdivision surfaces. The memory footprint and traversal efficiency of these data structures will dominate the end performance of the subdivision surface evaluator. Implementers must realize that subdivision surface computations are generally not floating point bound. As described in Figure 2, the butterfly subdivision scheme requires 3 multiplies, 6 adds, and 1 subtract for each dimension to be computed. So computing 3 dimensional positions via subdivision requires $3 \times (3 \text{ mults, } 6 \text{ adds, } 1 \text{ sub})$ for each computation. While this may seem like a lot, it is less work than the computation and random memory access operations that must be performed to gather all 6 triangles and all 8 vertices that serve as inputs to the computation. We must have data structures that can efficiently deliver these local neighborhoods relative to a particular triangle or triangle edge.

Winged Edge Control Mesh: We first need a *winged edge* style data structure that explicitly models the triangle neighbor relationships of the control mesh. Such data structures have many useful properties such as oriented vertex navigation, however we need them primarily to efficiently deliver the triangle neighbors in the control mesh. We use a somewhat simpler data structure that stores three triangle neighbor pointers for each triangle. Through this data structure we can provide all of the local neighborhoods needed to compute the first level of subdivision.

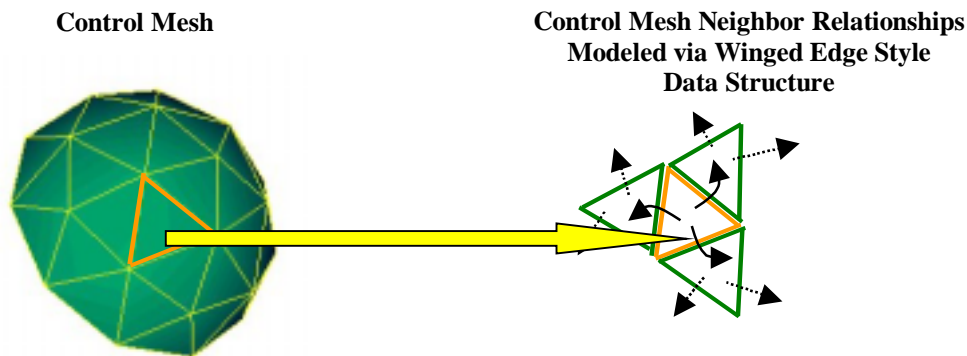


Figure 3. Modeling neighbor relationships in the control mesh via a winged edge style data structure.

However, gathering the local neighborhoods for deeper levels of subdivision presents a challenging problem. Uniform subdivision will generate a completely new mesh to replace the control mesh, thus invalidating the neighbor relationships modeled in the winged edge structure. The algorithm for constructing a new winged edge structure has $O(n)$ complexity and must examine the entire mesh. Such an algorithm is inefficient for real time applications of uniform subdivision and totally unsuitable for real time adaptive subdivision. Therefore we need a more efficient data structure for modeling the dynamic neighbor relationships of triangles as they are dynamically generated and consolidated during the subdivision process.

Restricted Triangular Quad Trees: Classic subdivision surface implementations have employed a *restricted triangular quad tree* to model the dynamic neighbor relationships of triangles generated via subdivision. In a quad tree, each parent triangle stores four pointers to its four subdivision generated child triangles. Thus each triangle in the control mesh will store all of its subdivision progeny in a unique quad tree. Insertions and removals are local, simple, and efficient. Furthermore, any level of subdivision already computed is readily available by just descending the tree to the desired subdivision depth.

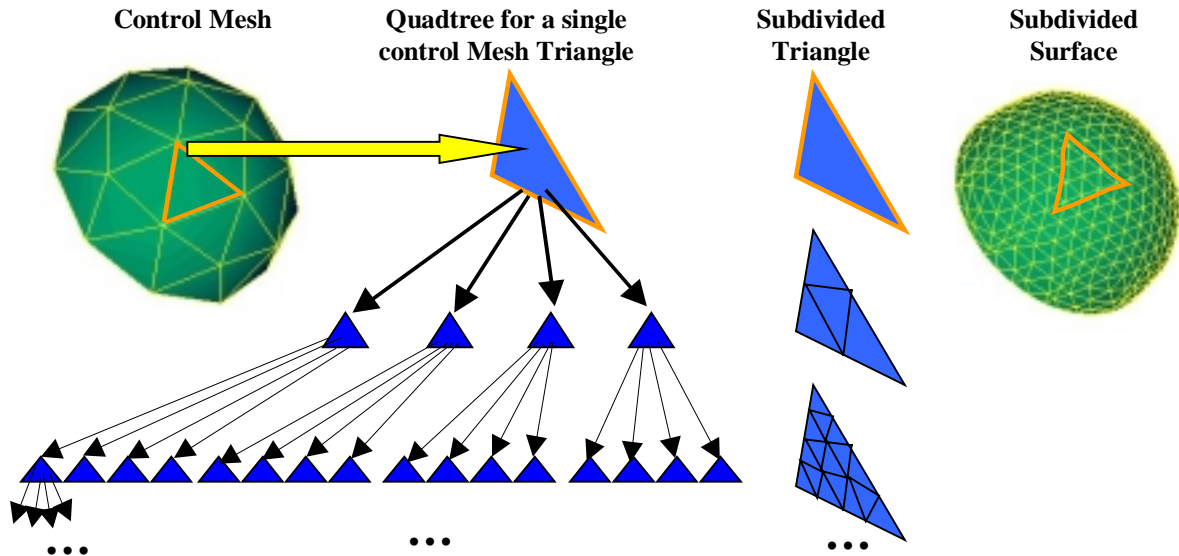


Figure 4. Each triangle in the control mesh points to a quad tree that stores all of its subdivision generated triangles.

In the quad tree, neighbor-finding operations incur some additional expense. Any triangle neighbors local to the quad tree can be found by using a *Nearest Common Ancestor* traversal algorithm. This algorithm delivers a triangle neighbor by first ascending the tree to the shared ancestor, then descending to the neighbor triangle. For a majority of the neighbors, this may require ascending only a level or two. cursory analysis would seem to suggest that this is an $O(\log n)$ operation. However, Hanan Samet proved in [Samet90] that the expected cost is actually $E(1)$. This result is due to the fact that half of all neighbor relationships have the immediate parent as a least common ancestor, requiring traversal to just a single node. Half of the remaining neighbor relationships can be resolved by traversing just two nodes, and so on. This geometric series simplifies to an expected constant cost, $E(1)$.

Finding triangle neighbors that are not local to the same quad tree can be a bit trickier. Such triangle neighbors will have a triangle in the control mesh as their least common ancestor. In this instance, one uses the winged edge style data structure to bridge the gap between the two quad trees. An alternative algorithm for computing triangle neighbors can be found in [Junkins99].

One final note about triangular quad trees. Subdivision to level n can not be performed unless all triangles in the local neighborhood are already subdivided to level $n-1$. As we will see, satisfying this condition is critical to supporting adaptive subdivision. Thus all quad trees are *restricted* so that sibling triangles are: 1) at the same subdivision level, 2) one subdivision level greater, or 3) one subdivision level less. This "differ by at most one" restriction satisfies the inputs to the subdivision computation and enables a very trivial crack filling policy for adaptive subdivision.

Practical details: Normals and Discontinuities

Implementers must attend to several important details to make subdivision surfaces practical. First, if we plan on using dynamic lighting models, we must employ techniques to efficiently compute surface normals for our new surface. Second, the subdivision surface evaluator must deal with any surface or attribute discontinuities.

Computing Vertex Normals: We need to compute new normals for all of the new vertices yielded by the subdivision computation. Furthermore we need to re-compute the normals for the input vertices. In general, any vertex adjacent to a new subdivision triangle, will need to have its normal computed. There are several strategies for computing these normals.

- One can compute and average all triangles incident to each vertex requiring a normal.
- Most subdivision schemes also provide techniques for computing a vertex's normal at the limit surface. Once computed, these limit surface normals need not change.
- Zorin and Schröder suggest in [Zorin96] an eigen vector technique that gathers the 2-neighborhood, computes two surface tangents, then computes the normal by cross product.

Each of these techniques require expensive triangle neighbor gathering and floating point computation and potentially a normalization step. Such expense may not be practical for real time applications of subdivision.

We employ a fourth technique that takes advantage of the local neighborhood gathering already employed for the subdivision computation. In fact we apply the very same subdivision mask to the surface normals that we applied to the surface positions. Using the subdivision mask in this manner produces a vertex normal that corresponds perfectly to the vertex position yielded by the subdivision computation. Also, implementing this technique requires fewer floating point computations than any of the other techniques. Finally, if the graphics API accepts un-normalized normals, then we can optimize further by neglecting the normalization step. We also employ the subdivision mask in the interpolation of texture coordinates and other attributes as DeRose did in [DeRose98]. This normal interpolation technique does not work if procedural surface detail or distortion is to be incorporated into the subdivision computation. Such an application must compute exact vertex normals.

Surface Discontinuities: Anytime that a control mesh does not model a perfect closed manifold surface such as sphere, we will encounter a mesh boundary that models the surface discontinuity. Subdividing along mesh boundaries is problematic since the empty space beyond the boundary does not provide

triangles and vertices as inputs to general case subdivision mask. We break this problem into two special cases. In the first case, we wish to subdivide an edge directly on the mesh boundary. In the second case we wish to subdivide an edge adjacent to the boundary. The resolutions to both cases are relatively simple.

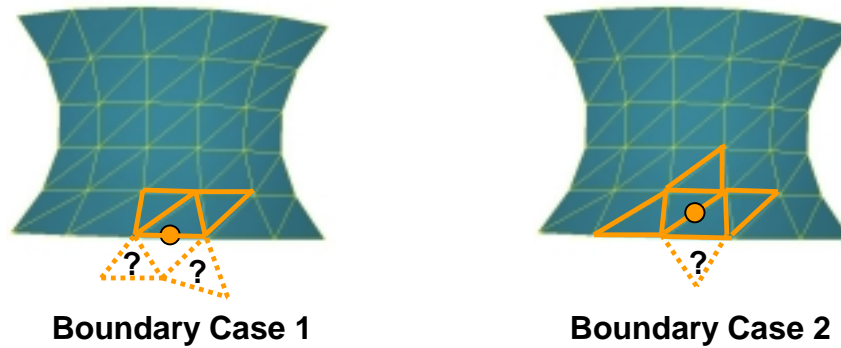


Figure 5. The two special case of mesh boundary subdivision. Case 1 shows the application of the butterfly mask to an edge on the boundary. Case 2 shows the application of the butterfly mask to an edge adjacent the boundary. In both cases, the general butterfly mask needs inputs that do not exist.

We resolve case 1 by employing a special four point subdivision mask instead of the general case butterfly mask. This mask was first suggested in [Zorin96]. The four point mask is simply a weighted average of four points along the boundary. It uses the weights as shown in Figure 6. One nice property of this mask is that if a physically separate mesh happened to share the boundary, the four point mask will guarantee that the subdivided boundary will be the same for both meshes.

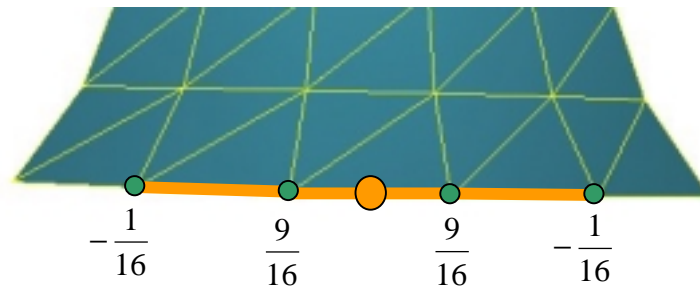


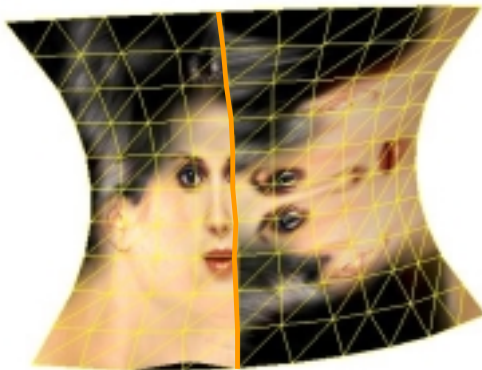
Figure 6. The four point mask is used to resolve boundary case 1.

We resolve case 2 by once again borrowing a technique [Zorin96]. In this situation, we synthesize the missing vertex position by projecting across the boundary the corresponding vertex position that is present. While this is a crude "guess" at how the surface might cross the boundary, the continuity of the surface is maintained and the results are visually satisfactory.

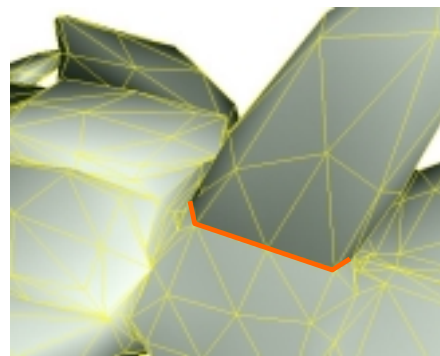
Attribute Discontinuities: Naïve subdivision surface implementations will assume that each vertex will associate with only one normal and one texture coordinate. While this may be convenient for the sake of implementation, it is a severe

restriction for content creators and the content authoring tools; both of which are very adept at modeling multiple surface attributes per vertex.

Arbitrary attribute association can be easily supported in subdivision, provided the underlying data structures can easily identify the attribute discontinuities at run time. Because our system employs the subdivision mask for modifying the texture coordinates and normals as well as the vertex position, attribute discontinuities can be handled exactly like surface discontinuities. We treat edges along a texture discontinuity or a normal discontinuity as we would a mesh boundary, applying the four point mask and the synthesizing transformations where appropriate.



Texture Coordinate
Discontinuities



Normal
Discontinuities

Figure 7. Two examples of subdivided meshes with attribute discontinuities. The orange lines indicate triangle edges whose vertices have multiple associations, texture coordinates or normals.

The Seemingly Unattainable Goal: Adaptive Subdivision

Adaptive subdivision has appeared as somewhat of an unattainable goal among application developers who are straining beneath polygon budgets or triangle throughput limitations. The general principle in adaptive subdivision is to dynamically generate triangles only where they contribute visually to the current image frame. The challenge is that one must spend precious run time cycles and data structure management to compute those triangles which are visually relevant. Ideally, the gain is less than the pain. To date, there have been few published results or shipping applications that demonstrate adaptive subdivision in a real-time setting.

Adaptive Subdivision: Our subdivision implementation provides a per triangle callback to programmatically determine whether a triangle should be subdivided, consolidated into its parent, or left unchanged. Though any adaptive subdivision

metric may be trivially programmed, we use a three stage adaptive metric incorporating 1) view frustum, 2) back face culling, and 3) screen-space error.

Most of our rendering performance is gained by limiting the subdivision to the portion of the mesh within the view frustum. We test each triangle using the frustum test described by [Abrash97]: for each frustum plane, compare the dot product of the eye position and frustum plane normal against the dot product of each vertex position and the frustum plane normal.

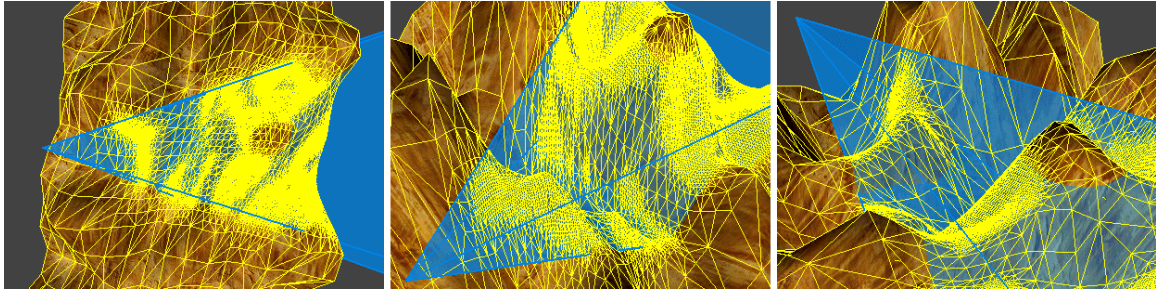


Figure 8. Above are three views of a single scene from different viewpoints. The pyramid represents the frustum used for the adaptive metric computation. It can be seen that triangles are not subdivided if they are outside the frustum (left), perpendicular to the line-of-sight (center), or back-facing (right).

We next evaluate the frustum visible triangles to see if they are back facing or not. Like the frustum test, this eliminates substantial numbers of triangles as candidates for subdivision. This test also consolidates other subdivisions as they transform away from the camera.

Finally, we evaluate all remaining triangles using an error metric from [Hoppe97], shown in Equation 1. This metric nicely combines error due to projection, proximity, and surface orientation. For example, faces which are relatively perpendicular to the line-of-sight or faces that are far from the viewpoint produce less visual error and thus are not indicated as candidates for subdivision. The screen space metric is further modulated by a per triangle surface error term, δ . Triangles with higher error values are more likely to be subdivided regardless of proximity or orientation. Figure 8 shows the metric applied to a scene. Notice in the center view how the faces that form the opposite valley wall are perpendicular to the line-of-sight, and therefore are not subdivided.

$$\delta^2 \left(\|v - e\|^2 - ((v - e) \cdot n)^2 \right) \geq k^2 \|v - e\|^4$$

Equation 1. Screen space error metric. δ is the per-triangle surface error, v is the center of the triangle, e is the viewpoint, n is the triangle normal, k is the pixel tolerance (user-settable quality term).

Success with the screen space error metric depends on finding a per-triangle surface error term. This term must measure the variance of the triangle from the limit surface. Subdividing this triangle should decrease its error and move the representation closer to the limit surface. We simply use the variance of the face

normal from the vertex normals. As the triangle is successively subdivided, this variance approaches 0. This crude metric fails for planar objects, like saucers, where all vertex normal equal their face normals but whose edge curvature changes with subdivision. However it is easy to compute and works well for a large class of models. Implementing a better per triangle error term is an important subject for future study.

As a triangle's error value approaches a specified tolerance, there can be some instability in the subdivision between frames. Visually, it will appear as if the subdivision for a particular triangle is flickering on and off. We abate the flickering by providing a tolerance zone at each stage of the error metric. Triangles whose error is in the tolerance zone will not be subdivided or consolidated until they leave the zone. The width of the tolerance zone is a user settable parameter.

Crack Filling: Adaptive subdivision requires dynamic crack filling to support the transitions between different levels of subdivision. Crack filling implementations are much simplified thanks to the sibling triangle restrictions guaranteed by our restricted triangular quad tree. Only the sibling triangles that differ by one level of subdivision must be crack filled. This presents exactly three crack fill scenarios depicted in Figure 9. Consider these scenarios from the standpoint of the triangle with the coarser subdivision of depth n :

1. One side of the triangle with subdivision depth n has a neighbor with subdivision depth $n+1$.
2. Two sides of the triangle with subdivision depth n have neighbors with subdivision depth $n+1$.
3. All three sides of the triangle with subdivision depth n have neighbors with subdivision depth $n+1$.

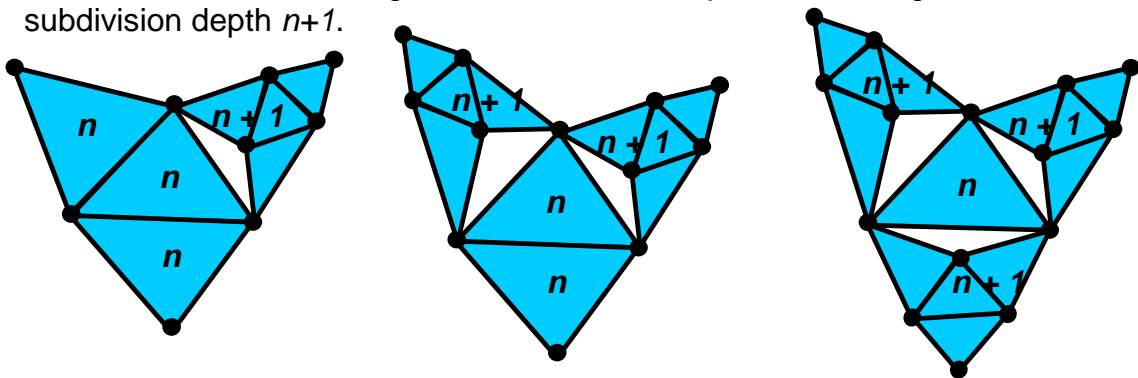


Figure 9. The three possible scenarios requiring crack filling.

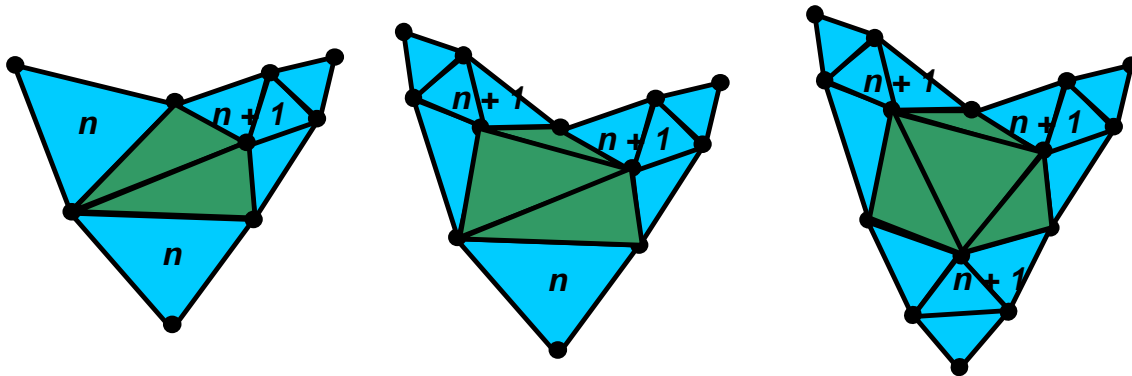


Figure 10. The crack fill triangulation.

The resolution to the first two scenarios is to replace the coarse triangle with 2 and 3 triangles respectively that perfectly seam the vertices of the two neighbors. In the third scenario, the triangle is effectively subdivided since all of its neighbors are already subdivided. We simply replace the coarse triangle with the subdivision triangulation. Figure 10 demonstrates the crack fill triangulation. Note that the crack fill triangles are for rendering purposes only and are not valid as inputs to subsequent subdivision computations.

Lazy Evaluation of Adaptive Subdivision Surfaces: Adaptive subdivision can benefit greatly if the implementation can cache partial subdivision results from frame to frame. For instance, in a terrain flyby simulation, it is likely that the majority of the subdivision is not changing significantly between frames. Ideally the adaptive subdivision metric dictates that the subdivision changes most dramatically as distant objects come into the foreground and also at the edges of the current view frustum as objects move on and off screen.

We employ a caching strategy called *Lazy Evaluation*. Lazy Evaluation traverses the quad trees in a breadth-first manner and limit the amount of traversal time permitted each frame. The breadth-first traversal ensures that coarser subdivisions are evaluated first. It is likely that our adaptive metric will deem these coarser triangulations as the areas of the high visual error, thus we can quickly correct them. Placing a time limit on the amount of traversal allows us to balance frame rate against surface quality. Implementing these two rules allows the adaptive subdivision to deliver a coarse surface when the view frustum is changing rapidly. Conversely, once the view frustum settles, the adaptive subdivision quickly delivers fine grained visual detail. Yet frame rates can be guaranteed throughout.

It is important to point out that Lazy Evaluation is only practical for models whose vertex positions are static, e.g. terrain and virtual environments. Lazy Evaluation is not appropriate for models whose vertex positions are dynamically modified each frame, e.g. IK systems or articulated skins. This is because subdivision surfaces are not invariant under transformation (NURBS are invariant). Dynamic

models require full re-computation of the subdivision each frame. Such computation is prohibitively expensive.

Conclusions

We have presented several practical techniques for implementing subdivision surfaces in real time 3D applications. In our experience, subdivision is a powerful solution to the scalability problem for load time content scaling, enhancement of pre-existing content, and for Internet compression applications. Though the implementation is more difficult, real time adaptive subdivision are a powerful technique for building high performance virtual environment simulations that scale visually and use rendering hardware efficiently.

Copyright © 2000 Intel Corporation. All rights reserved.

*Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owner's benefit, without intent to infringe.

Bibliography

- [Abrash97]** Michael Abrash. *Michael Abrash's Graphics Programming Black Book*. Coriolis Group Books, 1997.
- [Catmull78]** Ed Catmull and J. Clark. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer Aided geometric design*, Vol. 10, No 6, (1978).
- [DeRose98]** Tony DeRose, Michael Kass, and Tien Truong. Subdivision Surfaces in Character Animation. *Computer Graphics Proceedings, ACM SIGGRAPH 98* (1998).
- [Dyn78]** Nira Dyn and David Levin. A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control. *ACM Transactions on Graphics*, Vol. 9, No 2, (1990).
- [Hoppe97]** Hughes Hoppe. View-Dependent Refinement of Progressive Meshes. *Computer Graphics Proceedings, ACM SIGGRAPH 97* (1997).
- [Junkins99]** Stephen Junkins. Fast Triangle Neighbor Finding for Subdivision Surfaces. Intel Architecture Labs White Paper. (September, 1999)
- [Loop87]** Charles Loop. Smooth Subdivision Surfaces Based on Triangles. M.S. Thesis. Department of Mathematics. University of Utah. (August 1987).
- [Porter00]** Tom Porter and Galyn Susman. Creating Lifelike Characters in Pixar Movies. *Communications of the ACM*. Vol 43, No 1, January 2000.
- [Rosenzweig97]** Michael Rosenzweig, Techniques for Writing Scalable 3D Games. *Computer Game Developers Conference Proceedings 1997*. Miller Freeman.
- [Samet90]** Samet, Hanan, *Applications of Spatial Data Structures*. Addison-Wesley, Reading, MA, (1990).
- [Schröder98]** Peter Schröder, Denis Zorin, Tony DeRose, David Forsey, Leif Kobbelt, Michael Lunsbery, Jörg Peters. *Subdivision for Modeling and Animation, ACM SIGGRAPH 98 Course Notes* (1998).
- [Zorin96]** Zorin, D., Schröder, P., and Sweldens, W. Interpolating Subdivision for Meshes of Arbitrary Topology. *Computer Graphics Proceedings, ACM SIGGRAPH 96* (1996).