8-2010

PARALLEL √3-SUBDIVISION with ANIMATION in CONSIDERATION of GEOMETRIC COMPLEXITY

Stephen Cooney
Clemson University, Cooney.Stephen@gmail.com

PARALLEL √3-SUBDIVISION with ANIMATION in CONSIDERATION of GEOMETRIC COMPLEXITY

A Thesis
Presented to
The Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

Stephen Timothy Cooney
August 2010

Accepted by:
Dr. Donald House, Committee Chair
Dr. Tim Davis
Dr. Brian Malloy

Abstract

We look at the broader field of geometric subdivision and the emerging field of parallel computing for the purpose of creating higher visual fidelity at an efficient pace. Primarily, we present a parallel algorithm for √3-Subdivision. When considering animation, we find that it is possible to do subdivision by providing only one variable input, with the rest being considered static. This reduces the amount of data transfer required to continually update a subdividing mesh. We can support recursive subdivision by applying the technique in passes. As a basis for analysis, we look at performance in an OpenCL implementation that utilizes a local graphics processing unit (GPU) and a parallel CPU. By overcoming current hardware limitations, we present an environment where general GPU computation of √3-Subdivision can be practical.

Acknowledgements

I offer special thanks to my advisor Dr. Donald House for providing me guidance in my research and interests. His experience and support has significantly enabled me to filter and develop my ideas into this paper. Beyond the immediate contents of this paper, Dr. Donald House has introduced me to fields of study that have contributed towards these interests.

I also thank Dr. Robert Geist for introducing OpenCL and GPGPU development to me. Many methods introduced in this paper are central to the knowledge I gained while learning from him.

Dr. Tim Davis and Dr. Brian Malloy have frequently provided open guidance for me along the way and I am thankful that. Their engagement with my peers and myself has always been a reinforcement for our hard work.

Finally, I owe my deepest appreciation to my mother and father whom have always supported me, even when my ideas were and are terrible.

Table of Contents

List of Figures

1 Introduction

Over the past decade, graphics have improved to a point that is approaching photorealism. Realtime graphics have followed offline rendering techniques at a close pace thanks to rapidly advancing graphics hardware. Bit by bit, the realtime graphics pipeline has opened up which enables developers to create software implementations for custom and high quality graphics. In particular, realtime surface quality has improved tremendously over the past decade as a result of graphics processing units (GPU).

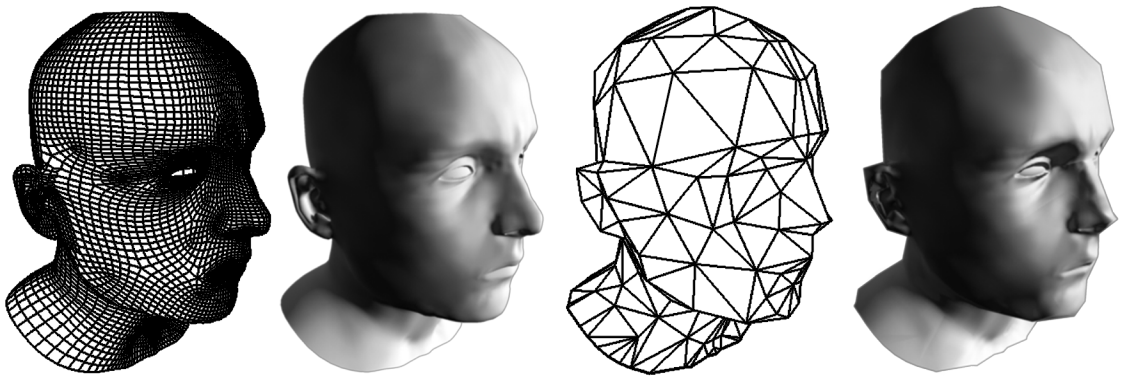

Figure 1: Modern realtime techniques optimize surface quality by baking high-resolution information (left 2 figures) onto low-resolution representations (right 2 figures).

Fragment shaders allow programmable surface operations such as Phong lighting and normal mapping. As a programmable operation, it allows the developer any combinatory shading technique within processing limitations. Its openness has allowed innovation to advance the industry as a whole. An example of normal mapping is depicted in Figure 1. Notice that the surface quality retains complex lighting information on the low resolution's interiors. Modern hardware has enabled this technique to run in realtime.

While hardware focusing on surface quality has proceeded at a rapid pace, pipeline improvements for dealing with geometry have only recently begun advancement. As a result, surface interiors tend to appear smooth, though it is not uncommon to find silhouette artifacts due to low polygon counts. This results in a degraded visual fidelity as depicted in Figure 2; the high resolution mesh on the left has smooth contours where the normal-mapped mesh on the right has creased contours. While vertex processing power does increase with hardware iterations,

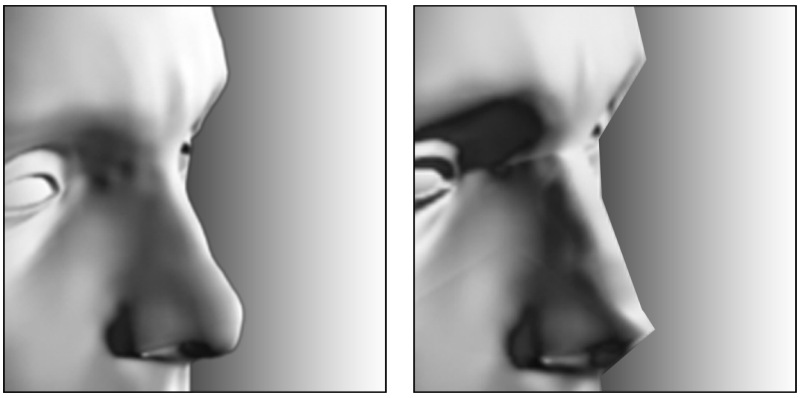

Figure 2: Normal-map artifacts. Left: High-resolution source w/ smooth contours; Right: Low-resolution Normal-mapped mesh w/ sharp edges.

the significant kinds of improvements that came about via fragment shaders have been somewhat absent for processing geometry.

Many current modeling techniques for generating smooth shapes rely on a branch of study known as subdivision surfaces. Subdivision surfaces work by taking a coarse geometric mesh and refining it until it becomes acceptably smooth (two to four times for most subdivision algorithms). These subdivided meshes can contain most of their original information, reducing the complexity for the artist. Special effects studios frequently use subdivision surfaces for highly believable models. Subdivision techniques have been too expensive to compute for real-time applications.

Recent hardware developments have begun to lean towards geometric processing. DirectX10 and OpenGL3 introduced programmable geometry shaders into the pipeline. These shaders allow semi-arbitrary geometry to be output based on varying inputs. In practice, this specific implementation does not provide an efficient solution for rendering higher fidelity geometry. It is useful for generating geometry that is often reliant on operations that happen in the vertex shader, which is normally inaccessible to the cpu-side implementation. With DirectX11 and OpenGL4, tessellator engines are built directly into the pipeline and hardware accelerated making real-time subdivision techniques feasible.

In conjunction, advancements are being made in general purpose graphics processing unit (GPGPU) development. Many architectures and standards are now moving to a pipeline that allows generic computation to be done on the GPU. This enables developers to leverage the power that is used to advance graphics for any computationally parallel problem. While before only hardware specific solutions were available such as CUDA ("CUDA Architecture Overview [...]" 2009) and FireStream ("ATI Stream Computing [...]" 2009), Khronos and Microsoft are releasing the cross hardware OpenCL and DirectCompute. Both in their infancy, they hint towards a future pipeline that is almost entirely programmable.

We believe that programmable pipelines allow the developer flexibility and control. By leveraging hardware accelerated functions with open ended possibilities, GPGPU programming can open up new possibilities in a short time. Specifically, we believe that GPGPU implementations may be used to efficiently execute subdivision methods.

In this paper, we start by introducing concepts required to understand geometric subdivision surfaces and those surfaces which are most commonly employed. We then expand to provide an overview of related advancements in hardware and general GPU computation. To explore our ideas we present a parallel algorithm built on Leif Kobbelt's $\sqrt{3}$-Subdivision technique. Investigating our parallel implementation, we find a scenario involving animation that provides performance optimizations. We will explore current limitations, and provide insight into future development trends.

2 Background

2.1 Geometric Complexity

In this paper, we look at geometric complexity by its relationship to visual fidelity. When talking about geometric complexity, we use this term to describe the bridge from a coarse, faceted geometric representation to a smooth representation. The concept of a smooth representation does not necessarily mean that the greater structure of a surface is continuous. We look at a surface as being smooth if it's underlying geometric structure, its topology, is not visible in a common environment.

We look for higher geometric complexity, with smoother surfaces to create more realistic imagery. To achieve this, here we look at various techniques for achieving geometrically complex representations. Generally, our goal is to present techniques that take a simple form of data and then convert the data into a more pleasing representation with considerations to performance.

2.2 Subdivision Surfaces

Subdivision surfaces are a technique for taking coarse, low-resolution meshes and generating higher resolution mesh with finer detail. Techniques with a smoothing component often work recursively. With each recursive pass these meshes refine towards an infinitely smooth, though highly detailed limit surface. This surface is often considered a limit surface because infinitely smooth meshes are not representable on a computer. The coarse meshes are generally represented by a collection of polygonal faces. Some subdivision techniques operate on specific collections of faces, such as quadrilaterals or triangles.

Of the subdivision techniques, two of the most common high-quality techniques are Catmull-Clark (Catmull 1978) and Loop-subdivision (Loop 1987). Catmull-Clark subdivision operates most efficiently on quadrilaterals, whereas Loop subdivision operates on triangles. Numerous other techniques provide alternative solutions such as √3-Subdivision (Kobbelt 2000) presented here; some are more efficient at the expense of quality and vice-versa.

2.2.1 Comparison to NURBS Surfaces

Traditionally, NURBS have been used for representing non-faceted surfaces. Created to aid manufacturing, they were adapted for graphics and became the primary tool for 3D representation. However, NURBS have drawbacks that limit a designers ability to create arbitrary surfaces. These drawbacks were the motivation for creating modern mesh subdivision schemes.

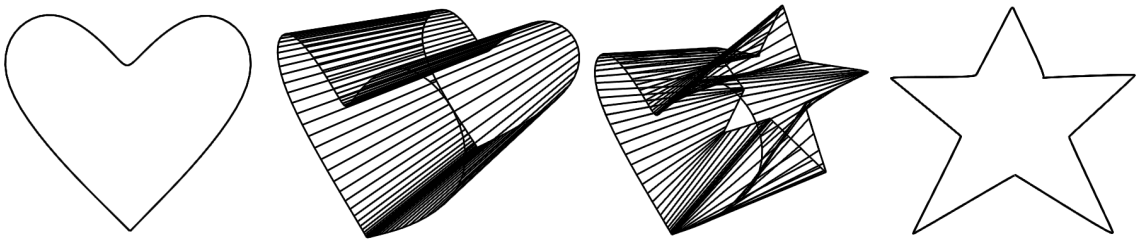

Figure 3: B-Splines are extruded to form a surface.

NURBS stands for "non-uniform rational basis spline." A NURBS surface is generally a grid of b-spline control points that represent two directions of curvature. Primitive shapes, such as a plane, cylinder or sphere are parametrized and then control points are editable to create the desired shape. *The NURBS Book* (Piegl 1997) is an excellent reference covering the subject. A basic cylinder extrusion is demonstrated in Figure 3.

The primary drawback is that NURBS surfaces do not operate on arbitrary shapes and non-regular topology, such as where a triangle and a quadrilateral may meet. Modern research has made progress to mitigate this as demonstrated in the paper "NURBS with Extraordinary Points: High-degree, Non-uniform, Rational Subdivision Schemes" (Cashman 2009) which allows extraordinary vertices.

NURBS are still critical when developing or analyzing a subdivision scheme. At NURBS' core are b-splines which at various 'orders' represent curvature. Here, 'order' represents depth of the curve's complexity in relation to its parametric representation, which is described later in terms of continuity. A subdivision scheme's goal is often to achieve NURBs like surfaces with arbitrary mesh topology, for example: Catmull-Clark subdivision was originally presented as

"Recursively generated B-spline surfaces on arbitrary topological meshes," which coincidentally sums the general field of study.

2.2.2 Combination with Displacement Maps

In practice, creating continuously smooth surfaces is not the goal when creating a continuous surface. Often when provided with a coarse mesh, subdivision is applied to create a basis for mesostructure complexity. The concept of a mesostructure relates to the medium level detail of a surface. While the shape of an object may be considered its macrostructure, the internal shapes that make up the macrostructure may be considered the mesostructure. Likewise, small details at the edge of perceptible complexity (and sometimes beyond) may be considered micro-structure.

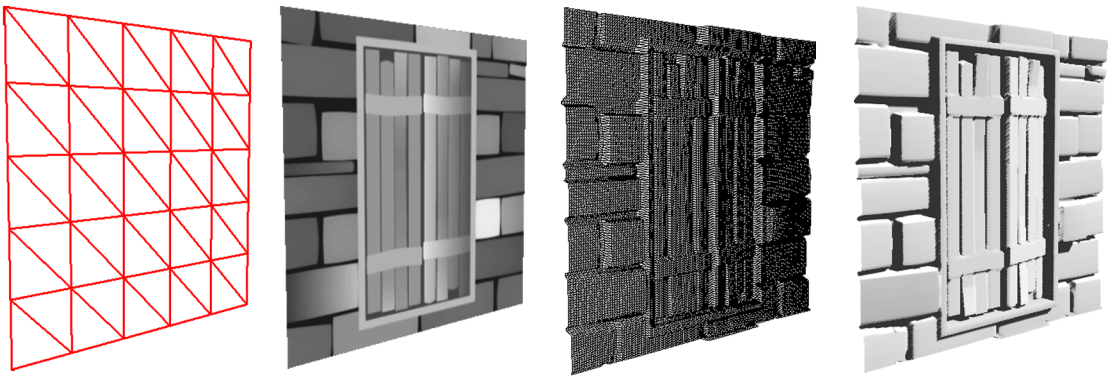

Figure 4: Displacement surface. Left to right: Control surface, displacement map, subdivided/displaced wireframe, and the final surface.

Since a coarse mesh defining only the macrostructure can not normally contain mesostructure information, displacement maps are defined which create mesostructure detail when combined with the subdivided mesh. Depending on the implementation, displacement maps may be represented in multiple ways. For example, displacement may be parameterized to represent magnitude along the original normal or dimensional components may be included allowing displacement in any direction. An example of displacement along the surface normal is depicted in Figure 4.

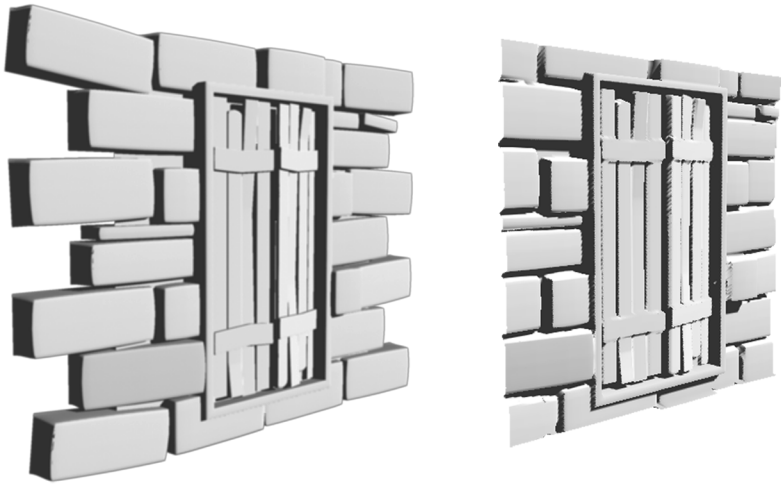

Figure 5: Original mesh vs displaced grid.

Depending on the surface, construction of displacement maps may also vary. Though in the simplest system, a displacement is a magnitude along a surface normal, the objects represented often vary. Commonly, high resolution and highly complex shapes are baked onto the shape that will be displaced. This enables artists to generate complex surfaces without worrying about local complexity. The baking technique is represented in Figure 5 where a high resolution surface (left) was used to generate the displacement map for the surface on the right.

2.2.3 Parametric & Geometric Continuity

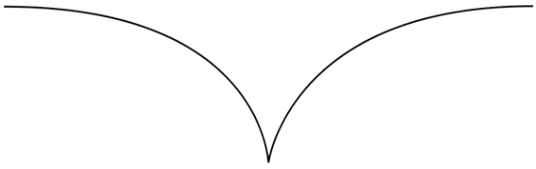

Figure 6: Two curves meet with C^0/G^0 continuity.

The concept of parametric and geometric continuity defines the smoothness over a region of a curve. In subdivision algorithms continuity often describes behavior at each vertex of a mesh after subdividing and smoothing operations. Because the goal of a smoothing algorithm is to generate continually smooth surfaces and continuity describes curvature, it is often easiest to

describe the results of a subdivision-smoothing algorithm in terms of either geometric or parametric continuity.

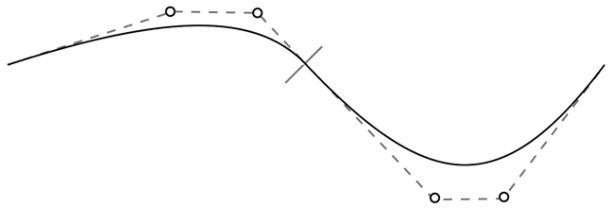

Figure 7: Two bezier curves that meet with C^1/G^1 continuity.

Each order of parametric/geometric continuity may be described in the form C^n/G^n, with "n" representing the order of continuity. For the most part, geometric and parametric continuity represent identical structures. In order for a curve to have C^n continuity at the intersection of two curves, the curves' derivatives up to the n^{th} level must be equivalent. In this case, a curves derivative may be represented in the form $d^n s/dt$ with s representing the output of a curve function. Generally, curves are not described with more than C^2 continuity, because parametric curves are normally implemented as cubic. C^0 requires that curves only intersect (Figure 6); C^1 requires the curves' tangents be the same (Figure 7); C^2 requires the curves' accelerations to be the same which dictates the motion of their tangents over time. (Figure 8).

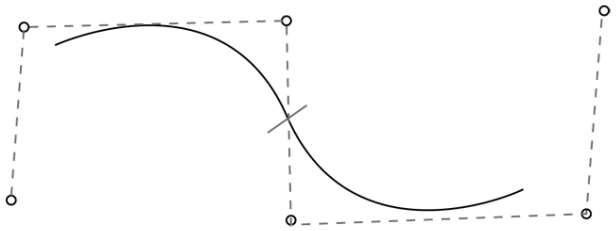

Figure 8: Piecewise B-Spline generates a curve with C^2/G^2 continuity.

Geometric continuity at G^n is implicitly considered true if C^n continuity is deemed true. However, G^n continuity does not necessarily indicate C^n continuity. Geometric continuity is used to describe the shape of a curve or manifold whereas parametric continuity more specifically describes parametric curvature functions. Order descriptions are fairly similar: G^0 indicates that

the curves meet; G^1 requires that curve tangents be in the same direction; G^2 requires that the center of curvature is shared.

2.2.4 Extraordinary Faces/Vertices

The concept of extraordinary faces and vertices is often central to a subdivision method's functionality and analysis. Certain smoothing algorithms vary depending on a polygonal configuration in relation to its neighbors. In a 'normal' quadrilateral mesh, each vertex would be expected to have four neighbors. The introduction of arbitrary topology may remove or introduce a new neighbor. This face may then be deemed extraordinary. Likewise, triangle vertexes often expect a valence of six. Here, valence represents the number of neighbors. Any variation would indicate an extraordinary environment.

Impact on a subdivision algorithm may vary. Normally, extraordinary vertex valences alter the continuity of the smoothing algorithm. For example: a subdivision algorithm may declare C^2 continuity everywhere except at extraordinary vertexes where it is C^1 continuous. The algorithm in focus here, $\sqrt{3}$-Subdivision, becomes C^1 continuous at extraordinary vertices.

2.2.5 Subdivision Extensions

Various studies are often done after a subdivision technique is presented to extend a given subdivision technique to support common features. Some of these extensions include: direct evaluation, adaptive tessellation and border retention. Understanding these features becomes useful when analyzing subdivision techniques for a specific implementation.

Studies in direct evaluation are attempting to take a recursive algorithm for generating up to a smooth limit mesh and make it possible to complete in one pass. Normally, this leads to a speed increase in the overall processing time. Also, some hardware still will not allow recursion, so direct evaluation methods are required. For example, the GPU hardware advancements which are discussed later require direct evaluation methods. Difficulty with generating direct evaluation methods relates to the methods required to create accurate smoothing coefficients. With arbitrary subdivision sizes, these smoothing coefficients often change, requiring not only a dynamic mesh-

handling algorithm, but a dynamic smoothing coefficient, which at even minor subdivision levels can be very complex.

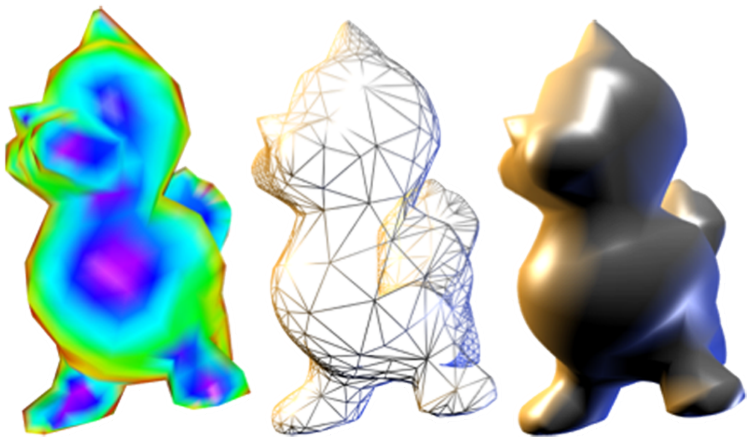

Figure 9: "Adaptive view-dependent Phong Tessellation[...]" (Boubekeur 2008).

Adaptive tessellation algorithms allow variable subdivision across a surface with as few artifacts as possible. These algorithms require smooth transition zones from higher to lower polygon density. If the algorithms are fast enough, they can assist in optimizing an efficient rendering solution that utilizes the subdivision scheme. For example, silhouette tessellation as depicted in Figure 9 subdivides a mesh in regions where the normals are more perpendicular to the viewing angle. This leads to smoother edges, without over-tessellating interiors.

The goal of border retention, or creased edges, is to retain original border edges and discontinuities that would otherwise be smoothed out of the mesh by progressive subdivisions. Techniques that retain the original mesh topology are more easily adapted to maintain borders by ignoring smoothing factors (generally vertex neighbors) across these borders. Other techniques such as √3-Subdivision discussed in this paper, require a more specific approach that is discussed later.

2.2.6 Linearly-Interpolated Subdivision

The most straightforward method of subdividing a mesh retains it's original form, but increases the resolution of mesh composition. Subdividing by linear interpolation causes vertices

at any level of subdivision across a face to be placed upon the face. As linear interpolation requires only the local face, subdivision of any level is allowed outright. For example, a quad may be triangulated towards the limit surface with no recursion required. As the only data required is local and the math is computationally inexpensive, linearly-interpolated subdivision my be generally regarded as one of the fastest ways to subdivide a surface.

Initially, this method provides no real visual benefit. Even as geometric complexity increases as the resolution increases, linear interpolation causes the output to be effectively the same as the input. When used as a platform in combination with other data, such as a displacement map, linear interpolation can provide an efficient level-of-detail system.

Semi-planar surfaces such as a brick-road or terrain benefit from this form of subdivision combined with displacement because the form of the overall surface generally has little effect on the apparent mesostructures. These may include stones, erosion and other locally distinguished forms. Displaced linearly-subdivided surfaces are in Figure 4. Implicit LOD systems can be created by subdividing individual faces at varying levels dependent on angle of incidence and distance.

2.2.7 Terrain Subdivision

Terrain subdivision is an entire subset of study in geometric complexity. Accomplishing high complexity in terrain is a unique topic because representation of a terrain surface is often different than of an arbitrary surface. While real-world terrains are as complex as any other physical entity, virtual terrains can be abstracted as planar surfaces that vary in height only. From there, features may be added to increase the terrain's fidelity.

Simplification of terrain allows for larger areas to be feasibly represented. Then, terrain subdivision algorithms tend to optimize rendering and storage methods while retaining final visual fidelity. Basic terrain often rests upon a linearly subdivided planar mesh combined with a displacement surface, as is described in section 2.2.2. Even complex terrain methods utilize a

displacement map, but use advanced techniques for determining variable subdivision resolution across a planar mesh.

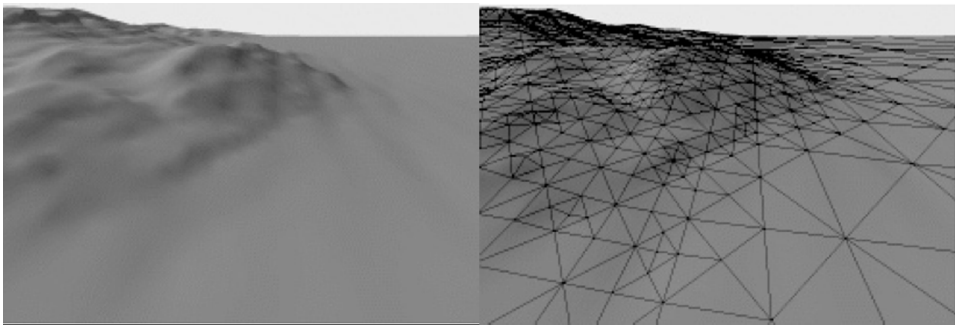

Figure 10: ROAM optimized terrain (Duchaineau 1997).

One common realtime terrain algorithm is presented in the paper "ROAMing Terrain: Real-time Optimally Adapting Meshes" (Duchaineau 1997). This technique attempts to optimize the mesh based on the viewpoint of the user while providing a runtime system for updating the mesh. The system determines subdivision priority by distance and surface variation. Areas of the terrain that vary greatly have higher priority for higher resolution subdivision than those areas that are relatively planar as shown in Figure 10.

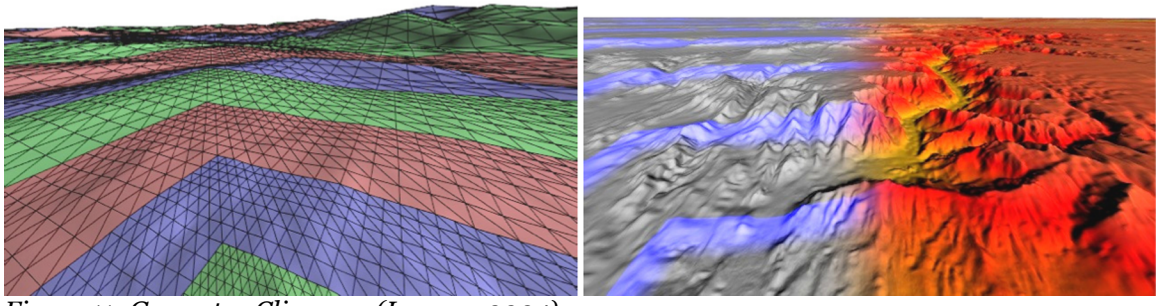

Figure 11: Geometry Clipmaps (Losasso 2004).

As GPUs become more powerful, techniques have been developed for terrain algorithms that rely on the GPU for performance and quality gains. A more common recent technique is presented in "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids" (Losasso 2004) which is shown in Figure 11. The technique provides high-resolution terrain-geometry when close. As terrain at a point moves away from the camera, triangle-density recedes in increments towards the horizon. Part of the goal is to retain relatively constant projected density.

This is achieved by having higher density grids closer to the camera. However, the main innovation in Clipmaps is its utilization of the GPU. Instead of pre-calculating the entire mesh on the host machine and then uploading it to the hardware, Clipmaps arrange small set-pieces dynamically on the GPU and displaced them at render-time. All set-pieces are reused over the entire mesh so resources consumed are relatively low.

2.2.8 Phong Tessellation

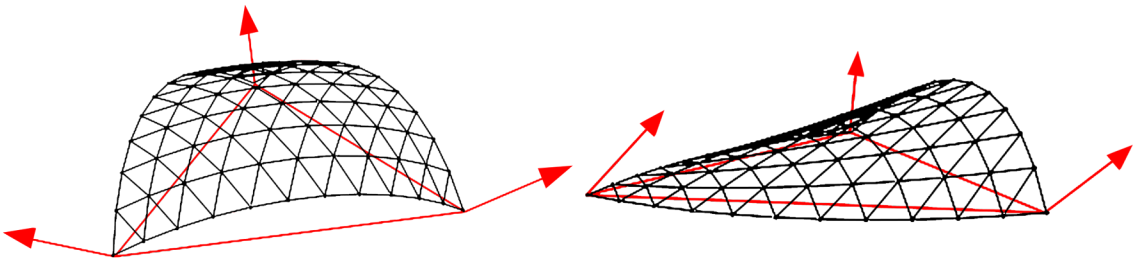

Figure 12: Phong tessellation on a triangle.

"Phong Tessellation" (Boubekeur 2008) is directly related to it's lighting counterpart known as Phong shading (Phong 1975). Where Phong shading operates by interpolating between the vertex normals, Phong tessellation operates by interpolating between projections onto the vertex normals' planes. Though the implementation is fairly simple, the resultant surface is only quadric. Note that in Figure 12 the triangle on the right has no inflection point, though a cubic curve with the same configuration would. This is similar to the behavior of a Phong shading implementation. Unfortunately, the surface is C^0 continuous everywhere except at the vertexes where it is C^1 continuous.

The technique allows for direct evaluation of the subdivided surface. This means that any level of subdivision and any configuration of subdivision may be done in one pass. Also, the algorithm is is relatively efficient and does a modest job at approximating contours of an object given little information. For both of these reasons, it has been presented as a new technique for modern hardware with tessellation support.

2.2.9 Curved PN Triangles

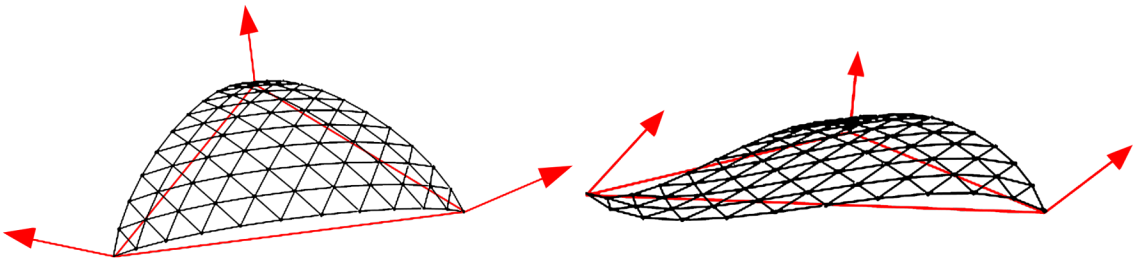

Figure 13: PN Triangles subdivision on a triangle.

"Curved PN Triangles" (Vlachos, 2001) presents a technique that is similar to Phong Tessellation; PN triangles require same input data: vertex positions and normals. The technique separates the normal calculation and the geometric subdivision calculation to achieve more pleasing visual results. Geometrically, the algorithm relies on a Hermite configured bezier triangle, that uses normals for input as opposed to control points. This derivation allows a cubic surface. The triangle on the right in Figure 13 has a visible inflection point. For lighting normals, instead of being calculated from the resultant geometry or being linearly interpolated (as Phong would), the normals are calculated using a quadric blending function which allows more accurate curves than Phong would. Unfortunately, the geometric subdivision implementation is still only C^0 continuous over the greater surface.

Curved PN triangles are one of the first GPU oriented tessellation algorithms. Having been developed coincident with ATI's "Truform" technology in 2001, curved PN triangles are a featured implementation of this technology. We revisit this later in 2.3 GPU Hardware Tessellation.

2.2.10 Bivariate Hermite Subdivision

"Bivariate Hermite Subdivision" (Damme 1997) is also very similar to both Curved PN Triangles and Phong Tessellation in that Hermite subdivision takes the same input to generate its subdivided surface: vertex positions and normals. Bivariate Hermite surfaces are higher order than both PN and Phong methods. Like PN triangles, Bivariate Hermite Subdivision can create

surfaces with inflection. By using normals at each vertex, a Bivariate Hermite surface normally achieves C^1 continuity. Bivariate Hermite surfaces are based off of Hermite curves, which are a modification of the cubic Berzier curve. Unlike both PN and Phong, Damme's technique is presented as a recursive operation, without specifying any direct evaluation technique.

Though it is generally desired for surfaces to be at least C^1 continuous, Bivariate Hermite subdivision is an interpolatory method of subdivision. This causes the original vertexes to remain stationary which often causes unwanted wobbles in the subdivided surface. For instance, at the corner of a brick there may be a sharp corner. Instead of slightly rounding off the edge, Bivariate Hermite subdivision would cause an outward kink in order for the flow of the geometry to meet the tangent at the edge.

There are techniques that counter the inherent wiggle in interpolatory schemes. In "NonInterpolatory Hermite Subdivision Schemes" (Han 2004) a technique is presented to "iteratively refine Hermite data in a not necessarily interpolatory fashion." Though, in general these methods are avoided since more prevalent techniques, such as Catmull-Clark are naturally noninterpolatory.

2.2.11 Catmull-Clark Subdivision

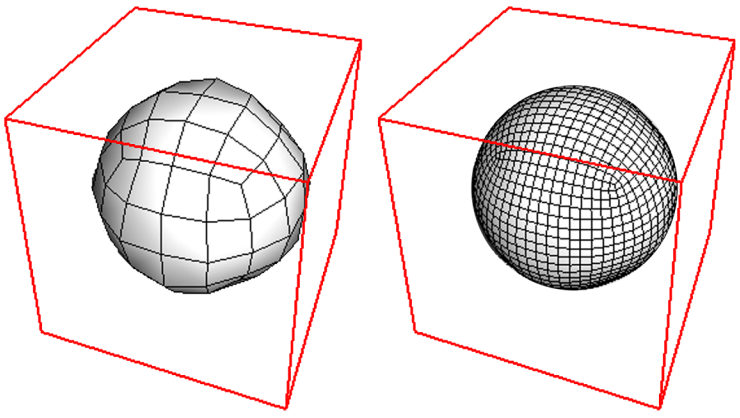

Figure 14: Catmull-Clark applied to a quadrilateral cube in 2 and 4 passes.

Catmull-Clark Subdivision was first presented in the paper "Recursively Generated B-Spline Surfaces on Arbitrary Topological Surfaces" (Catmull 1978). Not only was this technique

one of the first subdivision methods presented, it has since become the primary form of subdivision used in computer graphics. The technique operates most optimally on quadrilateral (quad)-dense meshes and generates b-spline quality surfaces that are normally C^2 across regular regions of a mesh. In one normal pass, for every original quad: four more are generated. It is also understood that recursively applying the technique to a cube several times generates a sphere, as shown in Figure 14.

In the artist's pipeline, one generally works with Catmull-Clark surfaces by creating a basic shell of the modeled object while keeping to quads as much as possible. Quads not only subdivide to smooth surfaces with the Catmull-Clark algorithm but allow other modeling techniques which can ease the modeling task. For example, the loop-slice is a common technique which splits a strip of faces by an edge. This edge may be slid up and down along the quads, which is much more difficult with triangles or arbitrary polygons that do not often form perfect loops. The artist may switch between low and high-resolution surfaces while focusing the majority of his/her work on the low-resolution surface. This approach eases the overall amount of data for an artist to handle.

In 2008, Charles Loop and Scott Schaefer presented a technique that attempts to create the Catmull-Clark without the complex cases at extraordinary vertexes. In their paper "Approximating Catmull-Clark Subdivision Surfaces with Bi-Cubic Patches" they present a technique that is C^2 continuous everywhere vertex valence is normal (touches 4 quads), but allow C^0 continuity at extraordinary vertexes. To retain smooth shading and displacement, they separate the tangent/normal field into a separate calculation for smooth shading and displacement.

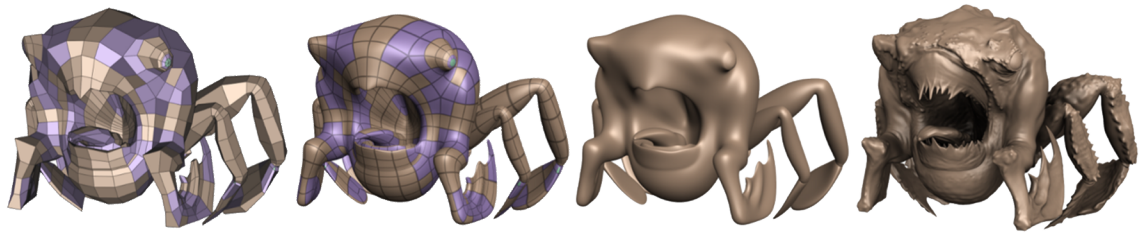

Figure 15: Stages of "Approximated subdivision surfaces[...]" (Loop, 2009).

In 2009, Charles Loop and Scott Schaefer extend their Approximated Catmull-Clark in the paper "Approximating subdivision surfaces with Gregory patches for hardware tessellation." In many cases, the algorithm produces visibly convincing Catmull-Clark surfaces such as in Figure 15. Here they move toward an explicit focus on hardware GPU computation and DirectX 11 hardware. They provide a general solution that works for both triangles and quadrilaterals. In it they argue that the memory requirements are low enough to be efficiently executed on the GPU.

2.2.12 Loop Subdivision

In 1987, Charles Loop takes many of the concepts presented in the Catmull-Clark paper but applies them to triangles in his thesis "Smooth Subdivision Surfaces Based on Triangles." This paper is later analyzed by Jos Stam in "Evaluation of Loop Subdivision Surfaces" (1998). The technique operates by splitting each edge of a triangle in the center and then smooths the original vertex. For every input triangle: four are output. Like Catmull-Clark subdivision, Loop subdivision retains b-spline like behavior with normal meshes. For triangular meshes, regular triangle vertexes have a valence of six.

2.2.13 √3-Subdivision

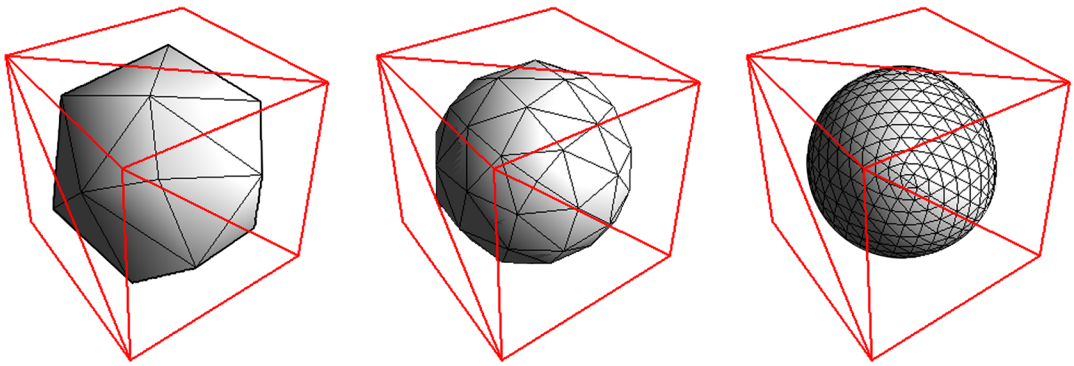

Figure 16: √3-Subdivision on a triangulated cube after 1, 2 and 4 applications.

In his paper, "√3-Subdivision" (2000), Leif Kobbelt explains a new method for smoothing geometry represented by a coarse triangle meshes. This method generates at a rate of 1-to-3 triangles for each subdivision pass. It produces a surface that is C^2 continuous everywhere, except at extraordinary vertexes (valence ≠ 6) where it is considered C^1.

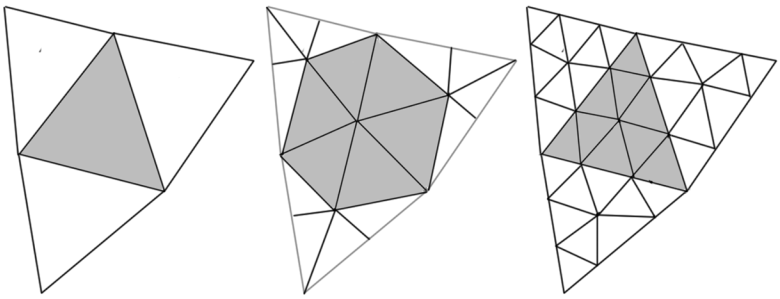

Figure 17: Topological behavior at 0, 1 and 2 levels of subdivision.

Figure 17 illustrates the topological behavior of normal subdivision. Note that for every other subdivision, the original topology is retained. Also, after two levels of subdivision nine triangles are created for the original, hence the √3 connection.

Kobbelt's subdivision method is a challenge to the more common "Loop" (aka. dyadic split) method which generates triangles at a rate of 1-to-4. The motivating difference is that with Loop subdivision, each edge of the original mesh is split, whereas in √3-Subdivision each face is split. Kobbelt argues that the smoothing operator is more representational by splitting a face in the center. Also since √3 subdivides slower than Loop, there is more level of control when resources are constricted.

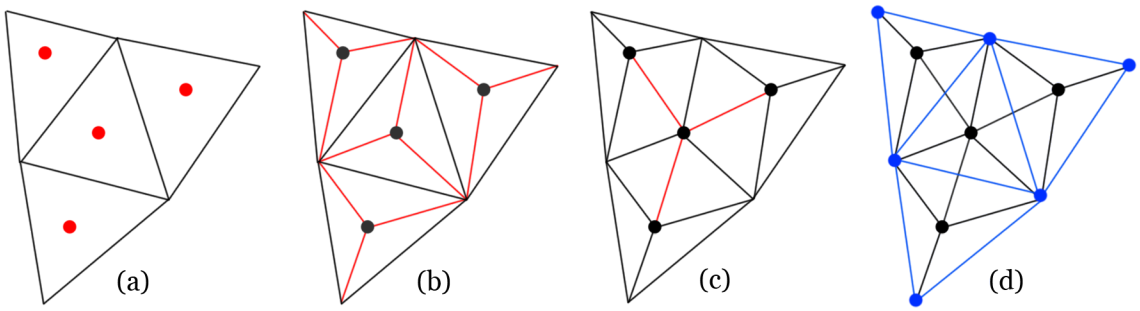

Figure 18: (a) Split triangles by finding centroids. (b) Connect new vertexes to old vertexes. (c) Rotate original edges. (d) Smooth original vertexes.

The algorithm requires three broad steps. In the first step (Figure 18(a), Figure 18(b)), faces are split by inserting a new vertex at the center of a triangle and creating edges connecting the new vertex to each of the three original vertexes. Next (Figure 18(c)), each of the original edges are spun to connect neighboring split vertexes as opposed to the old vertexes. Finally

(Figure 18(d)) , using a stencil that encompasses only direct neighbors, the original vertexes are relaxed.

If each triangle is represented by the points $\triangle(p_i, p_j, p_k)$, the location of the split vertexes are as follows

$$q := \frac{1}{3}(p_i + p_k + p_k)$$

As a basis for determining the smoothed location of the old vertexes, Kobbelt defines

$$S(p) := (1-\alpha_n)\, p + \alpha_n \frac{1}{n} \sum_{i=0}^{n-1} p_i$$

Here, n indicates the valence (# of neighbors) and p_i each of the neighbors to p . Then, α_n represents the smoothing coefficient defined as

$$\alpha_n = \frac{4 - 2\cos(\frac{2\pi}{n})}{9}$$

Kobbelt constructs α_n by calculating the eigenvalues for a subdivision matrix as explained in his paper. Aside from the edge-spinning operation, the above three functions outline the majority of work required for basic √3-Subdivision.

√3-Subdivision also supports many extensions intuitively. Since subdivision is performed per face through splitting each face into three, it is possible to support adaptive tessellation. Smooth boundary lines may be maintained by skipping edge-spinning operations on odd subdivision passes, and feature lines (border retention) may be maintained with variate smoothing rules. For border retention with √3-Subdivision, vertexes that lie on border edges and should remain sharp only need use neighbors connected on that line when smoothing.

2.3 GPU Hardware Tessellation

Modern GPUs are beginning to focus on tessellation techniques for realtime engines. By providing a hardware accelerated interface, GPU manufacturers are enabling developers higher levels of geometric complexity, and therefore potential visual fidelity. Demonstrations on current technology by both NVidia and ATI provide promising outlooks into the future of tessellation engines.

Though GPU based tessellation is a current focus for both major manufacturers and graphics API developers, ATI has engaged in GPU tessellation research since 2001. Their technology was initially named "Truform" and first supported on the ATI Radeon 8500, during the DirectX 8 generation. The technology implemented "Curved PN Triangles" which is described in 2.2.9 Curved PN Triangles as a technique for generating a cubic face that requires only vertex positions and normals. This technique removes extensive requirements and provides a direct-evaluation, non-recursive technique for subdividing triangles. ATI's technology is described in their whitepaper "Truform" (2001).

Due to Truform's lack of third party developer support, "Truform" was dropped as an advertised feature, though ATI cards still supported tessellation. On ATI hardware, tessellation is possible in DirectX 9 and DirectX 10 implementations via extension APIs. In conjunction, ATI's Xenos graphics chip for the Xbox 360 enables hardware tessellation on a console. ATI provides an in-depth discussion about tessellation integration with DirectX 9 and newer cards in their paper "Programming for Real-Time Tessellation on GPU" (Tatarchuk 2009).

Starting with DirectX 11 and OpenGL 4, hardware tessellation is supported across both main APIs. With ATI already supporting tessellation, NVidia is now also producing chipsets with hardware tessellation. It is reasonable to assume that with nearly identical tessellation systems across hardware and APIs will entice developers to fit tessellation into their pipelines.

2.3.1 Tessellation Pipeline

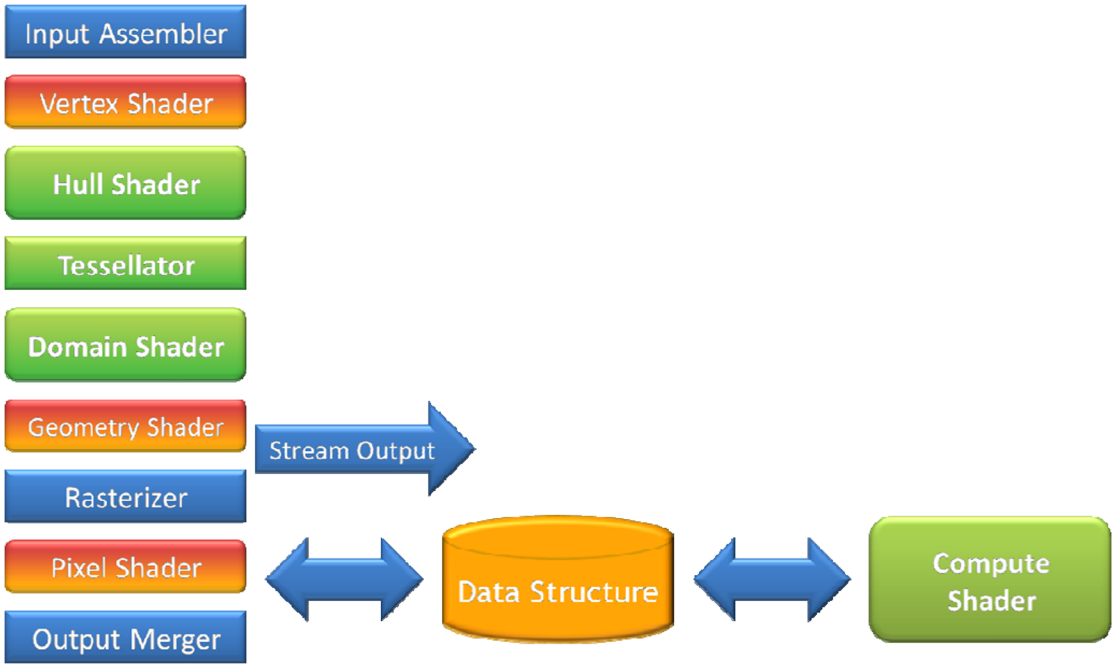

Figure 19: DirectX 11 rendering pipeline ("Programming for Real-Time Tessellation on GPU" 2009).

DirectX 11 and OpenGL 4 hardware tessellation adds three operators to the standard pipeline. They are injected between the vertex shader and the geometry shader stages (see Figure 19). Their placement allows deformation applied to the low resolution input mesh before subdivision during the vertex shader shader stage, and geometric operations that rely on the subdivided mesh to be computed in the geometry shader. For example, skinned animation may be applied in the vertex shader and volumetric shadow projection can be generated in the geometry shader.

The first new operator is the programmable "hull" shader. This shader operates once per every control primitive. This is basically a primitive from the input mesh that is to be subdivided. This allows the developer to compute tessellation factors for adaptive tessellation and to modify patch control points that are used later for generating the smooth surface.

The next operator in the pipeline is the fixed "tesselator" stage. Here, the tesselator uses the tesselation factors that were output from the hull shader to subdivide the input patch. The output is a set of uv coordinates which get passed into the next stage one at a time. At this point, the subdivided vertex points are represented in their relative positioning, but have no real world positions.

The last operator is the "domain" shader. Here uv coordinates and a patch are provided as input and its intent is to determine the location of the newly generated vertex. A system that did no smoothing would do linear interpolation between the patch points based on the uv coordinates. Then, the domain shader may sample from a displacement texture and generate complex silhouettes.

New programmability empowers the programmer with options when implementing techniques to apply visibly convincing curvature in a given situation. This programming model does not specifically limit hardware tessellation models to those that are presented in this paper, leaving potential for future research in this area.

2.3.2 GPU Subdivision Techniques

There are already a diversity of techniques that may be applied to hardware tessellated surfaces, including: Linearly-interpolated subdivision, "Phong Tessellation" (Boubekeur 2008), "Curved PN Triangles" (Vlachos 2001), and Approximated Catmull-Clark ("Approximating Subdivision [...]" Loop 2009). The general goal of these real-time tessellation techniques is different than the general body of subdivision surface techniques. While a pure subdivision surface refines towards an infinitely smooth limit mesh, most hardware tessellation techniques are targeted at generating a large number of triangles to show mesostructure detail. In order to achieve mesostructure detail, a significant part of the pipeline is to integrate displacement maps which are applied after surfaces are subdivided. Each of these techniques are described in 2.2.

2.3.3 Limitations

While performance gains from GPU based tessellation engines may provide a way into the future for real-time games and media, they have a few significant drawbacks. None of the currently advertised subdivision techniques provide dependable solutions to real continuous subdivision. Although the approximation of Catmull-Clark subdivision is very close to actual Catmull-Clark, there are cases in which it will break down to being C^0 continuous ("Approximating Catmull-Clark [...]" 2008).

Because each of the techniques are approximations of actual subdivision algorithms and are generally targeted at providing dynamically complex surfaces for displacement, we believe there is room for improvement and a necessity for optimizing accurate subdivision techniques. This drawback is a motivation and justification for implementing √3-Subdivision in parallel with OpenCL. With OpenCL, it is possible to achieve a higher level of accuracy and control.

2.4 Alternatives for Mesostructure Complexity

It is worth expressing that geometric subdivision techniques are not the only investigated disciplines for the purpose of generating higher mesostructure complexity. Fundamentally, hardware systems that support a tremendous number of polygons at every level of the pipeline solve the need for active subdivision algorithms. This requires systems that can handle taxing memory requirements, and the computational ability to transfer, transform, deform, and render the required data at every frame.

There are creative solutions that treat mesostrucure complexity as a raytracing problem. Parallax mapping (Kaneko 2001) began research into raytraced shaders. The algorithm transforms the view-direction onto the surface into texture-coordinate tangent space. It then raymarches until passing through a sampled displacement height. The algorithm produces noticeable artifacts that appear as sheets from the fixed step sizes. Modifications to this technique vary step sizes depending on viewing angles and have provided self-shadowing techniques. These techniques are explained in "Per-Pixel Displacement Mapping with Distance Functions." (Donelly

Figure 20: Per-pixel displacement. A Displaced Stone Surface (Donelly 2005)

2005). While these techniques significantly improve mesostructure detail, the do nothing for sillhouettes. Notice in Figure 20 that the surface of the top plane appears very detailed but retains the sharp borders.

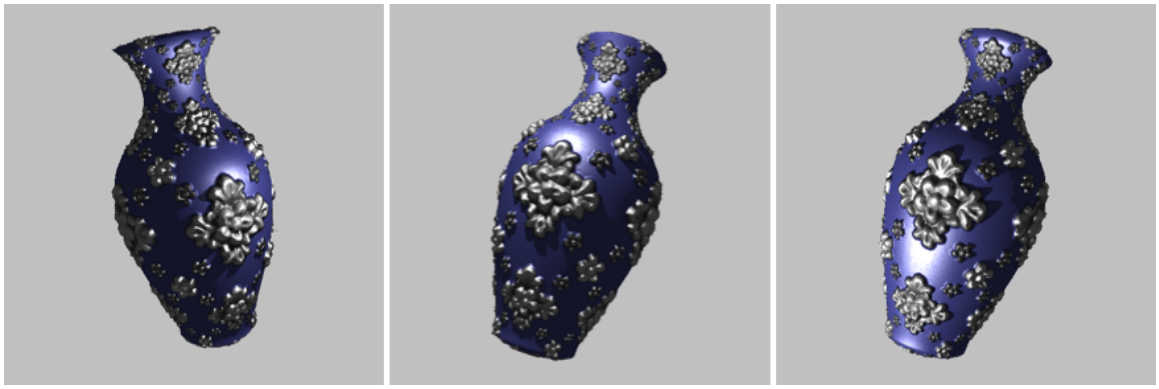

Figure 21: "Examples of VDM rendering under different lighting and viewing directions[...] 8640 tri, 130 fps" (Wang 2003).

"View-Dependent Displacement Mapping" (Wang 2003) uses similar techniques to the above mentioned. Once rasterizing a polygon, raytracing techniques are used to determine the output. By baking a displacement texture's information across multiple angles/curvatures, VDM

is able to provide complex mesostructure parallax and complex silhouettes (see Figure 21). Unfortunately, the memory requirement is exceptionally prohibitive. In the referenced paper's implementation, "All VDM data in this paper is of resolution 128x128 and sampled for 32x8 viewing directions and 16 curvatures" (Wang). Realistic texture sizes (512x512 and upwards) would dominate memory requirements of an application.

2.5 OpenCL and GPGPU

OpenCL is an open standard for "General Purpose Graphics Processing Unit" (GPGPU) development as managed by the Khronos Group and outlined in its specification ("The OpenCL Specification" 2009). It is a cross-platform, cross-hardware supported API for interfacing with GPU drivers. The goal of GPGPU development is to harness excess spare power available in modern graphics cards and their highly parallel processing structures to efficiently handle tasks that benefit from parallel processing.

Many modern computer problems involve highly parallel tasks such as a physical simulation that includes many members that may be computed independently. In these simulation environments, the problems are most efficient when they are also computationally expensive (though still local in their scope)

In our experiments, we choose OpenCL as a platform because subdivision schemes are extremely parallel. Most subdivision operations are fairly simple tasks over enormous amounts of data. We will show later that performance benefits in our implementation are limited because they are not as computationally expensive as they are memory-management expensive.

OpenCL does not limit development to GPU implementations. By presenting the concept of a "compute device," OpenCL provides a framework for any form of parallel computing. This allows easy portability to different devices given they have OpenCL support. In our experiments we use two scenarios which include a CPU based compute device for comparison with the GPU implementation. OpenCL specification advertises for implementation on "multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs" (OpenCL, 2010).

We believe OpenCL may represent a future trend for highly parallel computing in the future. The major hardware developers, NVidia and AMD, develop more "open" hardware each year. Intel has experimented with developing an almost entirely general purpose multi-processor under the name “Larrabee” (Seiler 2008). General purpose multiprocessors provide the developer with a higher level of control over one's applications and may provide the backbone for future scientific processing.

2.5.1 Other GPGPU Libraries

OpenCL is one of the newer of a legacy of parallel GPU computing architectures. NVidia CUDA and ATI FireStream represent hardware specific GPGPU APIs and both were introduced in 2007. OpenCL has the added benefit of integrating with OpenGL implementations, and DirectCompute integrates DirectX.

Traditionally, fragment shader support could represent the start of GPGPU. Developers sometimes encode information in textures, compute shader operations on these textures, and then read the results back as another texture. An example of this is in "Sparse Matrix Solvers on the GPU..." (Bolz, 2003) which uses fragment shaders for smoothing meshes and fluid simulation. NVidia and ATI developer websites provide numerous tricks for simulation using older GPU techniques.

2.5.2 Parallel Operations

The primary focus of computing when viewing GPGPU APIs like OpenCL is the abstracted parallel operation. In OpenCL they are known as "kernels" and are smaller, isolated functions that are built to be run on volumes of data. A kernel's isolation allows the API core to handle the thread resources and provides scalability.

In the most efficient kernels, data to kernel ratios are generally one-to-one. In these scenarios, kernels are instantiated for every unit of data and operate exclusively on that unit of data. However, they are often more complex and reference data and information relevant to all permutations of the kernel. In modern hardware, routines that allow thread synchronization and

more advanced resource management, such as arbitrary linked-lists, enable complex memory relationships. However, any arbitrary synchronization comes with a performance cost.

When determining whether or not an operation can be made parallel, it is important to look at how dependent a series of operations are on one another. The less coupled, the more an operation will benefit from parallelization, and possibly GPGPU acceleration.

2.5.3 GPGPU Overhead

After determining if operations can be made parallel, it is important to determine what operations are taxing enough to overcome GPU overheads. Unfortunately, communication between multiprocessing devices, such as GPUs, and the host memory is often very slow. These penalties do not occur as significantly for multi-core CPUs, though parallel CPUs do not normally have as many processors as a GPU. Currently many operations that are highly parallel do not benefit from GPGPU computation because they become memory locked.

In our implementation, we find that our biggest challenge is overcoming the memory bottlenecks. As we will explain later, considering the memory implications of what is being computed often makes the difference in results.

3 Method

Through our research, we have implemented a few systems based off of Leif Kobbelt's √3-Subdivision technique presented in 2.2.13. We present a parallel algorithm of √3-Subdivision, and an implementation of an environment running our algorithm on current hardware can produce beneficial results.

The components presented here are the: parallel algorithm, data-requirements, animation extension and an OpenCL implementation. The parallel algorithm describes the technique that we used to implement √3-Subdivision and the data-requirements section expands upon data-types and complications that are required for the parallel algorithm to work. We go on to describe a simple animation environment that the method can work in. Finally, we will describe specifics about our OpenCL implementation which drives our results and discussion.

3.0.1 Parallel Implementation

Our implementation requires modifications to Leif Kobbelt's base explanation. A literal interpretation of √3-Subdivision technique would be inefficient. For example, the base interpretation describes the mesh topology/connectivity as changing multiple times during one pass. In this paper, efficiency and algorithmic technique is the focus.

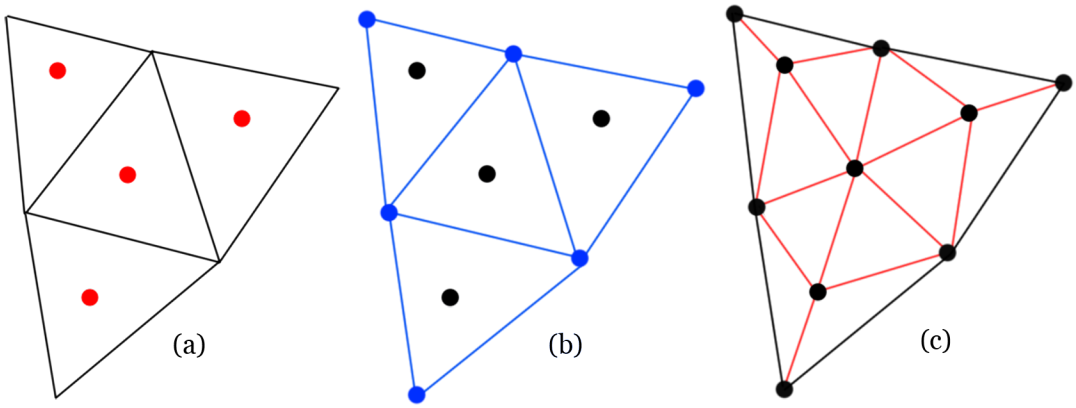

Figure 22: Parallel √3-Subdivision. (a) Split/find-centroids. (b) Smooth the original vertexes. (c) Re'wind' the mesh topology.

Our implementation presents 3 passes. First (Figure 22(a)), the centroid of every triangle is found. Then (Figure 22(b)), the original vertexes are smoothed. Lastly (Figure 22(c)), the mesh topology is rewound. These passes describe an optimized approach to subdividing a mesh. Unfortunately, implementation is more complex as each of these passes require numerous structures informing the passes about a mesh's organization. As some mesh storage methods contain the information required, as described in each section below, we do not include this in our parallel algorithm.

The biggest hurdle in executing these passes in parallel is overcoming divergent threads and predicting memory organization. This divergent behavior generally occurs when determining where to read/write from memory and how much to read/write. Our analysis has simplified the technique so that information is provided to the passes to make operation as linear as possible. With these optimizations, computational complexity becomes relatively inconsequential as the following sections depict.

3.0.1.1 Split/Find Centroids

The first pass determines the centroid of every triangle-face. This is calculated by executing a thread for every triangle-face and averaging the positions. These centroids are saved out as new vertex positions and used for reconstructing topology in a later pass.

In order for this operation to be in parallel and avoid collision, we need to determine where these vertexes are written to so that later passes can quickly access the new data. Fortunately, since one vertex is generated for each face, we can use the relative face index to determine where to output this new vertex. This new index can then reference a point beyond the last of the original vertexes for a direct write.

3.0.1.2 Smooth Original Vertexes

The second pass runs a smoothing mask on every original vertex as described in 2.2.13. It uses its original vertex neighbors, which are connected via an edge, as part of a weighting algorithm to determine the new vertex location.

In review the smoothing function is

$$S(p) := (1-\alpha_n)\,p + \alpha_n \frac{1}{n}\sum_{i=0}^{n-1} p_i$$

The smoothing coefficient is

$$\alpha_n = \frac{4 - 2\cos(\frac{2\pi}{n})}{9}$$

The dynamic nature of the input provides a few complications. As the smoothing mask can take any number of neighbors (n), the number of inputs at each vertex changes. By using a dynamic storage structure it's possible to reference any number of neighbors. For many implementations, a library such as a STL vector would provide this functionality.

However in our implementation, we bake this information down into two linear vectors of data. This provides the benefit of being serializable and generally lightweight. One vector collects indexing data required for each vertex to reference its neighbors. Another vector contains a flat collection of all neighbor indexes that the former vector references. The vector containing neighbors can be of any possible length, however the vector that indexes the neighbor list contains two components. The first component represents the first index into the neighbor list whereas the second component represents how many neighbors the vertex has. This is explained and illustrated in further detail in 3.0.4.

3.0.1.3 Reconstruct Mesh Faces

The third pass reconstructs the mesh topology to represent the next level of connectivity. An investigation of Kobbelt's general technique reveals that all topology can be determined by information relevant to the original mesh's edges. Essentially, for every edge there will be two new faces in the edge's place. This provides our approach for making the topology determination parallel and independent.

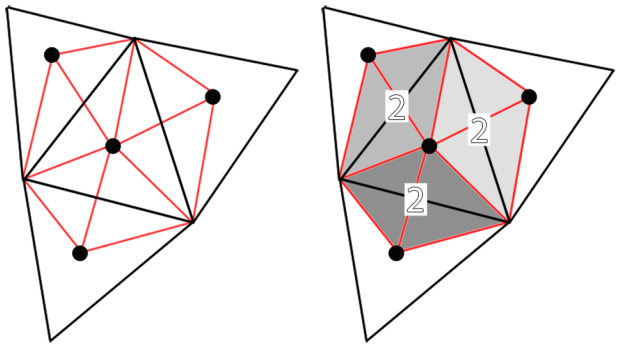

Figure 23: Two faces for every original edge.

Because we know that there are two faces for every original edge (Figure 23), we immediately know how much data we need to store for the new topology. We can also determine the indexes for the new face data based on the implicit connection to the producing edge's index.

However since we generate two faces for every edge, not only do we have to choose which edge is placed first, but we need to determine correct and consistent face winding patterns. For example, we normally want all faces to be wound counter-clockwise facing the exterior or the mesh. To approach this problem, we first have to look at where all of our required data is coming from. To construct a face based off indexes, we only need the index of each vertex involved. This is assuming that vertex components, such as normals, reside in the same index space.

For a parallizeable algorithm, we require an edge data structure that references indexes to the edge vertexes and to adjacent faces. We require indexes into the array of faces because we previously placed the centroid vertexes at indexes based on their respective faces. We can use this connectivity to easily access the generated centroids. Furthermore, the order that these are referenced is significant to a correct winding pattern.

In our implementation, we refer to these components as the edge A, B vertex and A, B face. If we organize the indexes to reference a pattern where A-vertex to B-vertex to A-face is counter-clockwise and A-vertex to B-vertex to B-face is clockwise, we can create a consistent winding pattern in parallel.

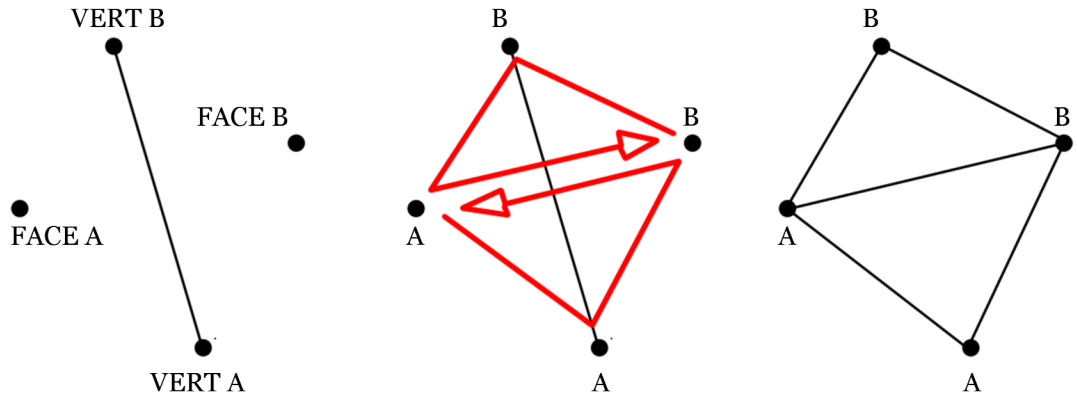

Figure 24: Face winding technique.

This counter-clockwise winding pattern comes down to A-face to A-vertex to B-face and B-face to B-vertex to A-face. The winding pattern is illustrated in Figure 24. Given this information, all we require for the winding pass is the collection of edge information. We can then execute a thread for every edge and generate two faces as an output.

3.0.2 Recursive Implementation

In order to do multiple levels of subdivision it is possible to run the operation multiple times; each level can take the last level's output as input. Required inputs that are not generated by the algorithm must be calculated at each level. If relationships such as neighbor and edge information do not exist, calculation can be time consuming.

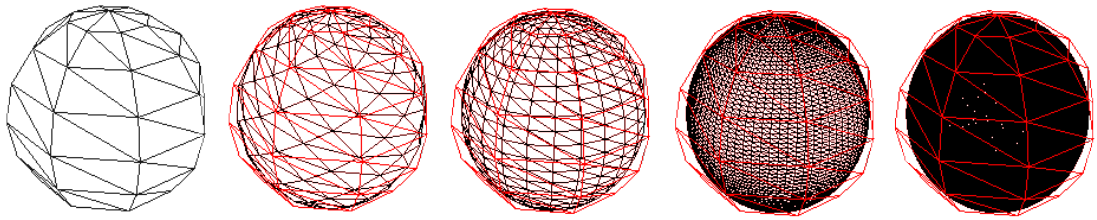

Figure 25: Recursive subdivision onto a triangular control sphere. Subdivision levels are: 0, 1, 2, 4 and 6.

In our implementation, the only dynamic data at each level of subdivision is the input and output vertex data. Other data is static and only uploaded the first time the operation is run. With this approach we find that it is unnecessary to run the winding operation multiple times; the topology will not normally change. This eliminates and partially optimizes the algorithm.

3.0.3 Data Requirements and Caching

As we've mentioned before, the initial efficiency of the entire algorithm may depend on what data structures are available when starting the operation. Calculation of many required data structures is not parallelized in our implementation. These requirements generally do not change; once they are calculated they do not need to be recalculated.

In order to combat the initial calculation overhead, we implemented a basic level of caching. In our implementation we hash the source mesh and using this key to locate a cache that stores initial and constant input. Failure to locate the cache will result in generation of a new cache. By using a hash test, we are able to provide a fairly secure, unique and dynamic environment.

This becomes even more useful when doing recursive subdivision as outlined above. After a few levels of subdivision on a regular object our system became bottlenecked handling the requirements. For example, the edge information collection can require an $O(N^2)$ search through the mesh to determine connectivity. However if this information is generated with the mesh and stored, this complication can be mitigated.

The dynamic input and output vertex information, does not need to be cached because it is expected to change. Depending on the subdivision environment, the output index buffer may be cached if it remains constant. Most every other input is static and includes the source: index buffer, neighbor information, neighbor list and edge information.

3.0.4 Data Type Definitions

This section expands upon the data types discussed and required by the parallel implementation in 3.0.1. There are five primary data collections needed for the actual execution of the operations. These include: vertexes, faces/indexes, neighbor information, neighbor collections, and edge information.

A vertex (Figure 26) represents a point in space that makes up the sub-divisible mesh. An individual vertex consists of three floating point values which represent the standard coordinate

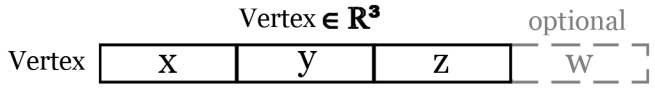

Figure 26: Vertex definition.

system. Depending on the implementation, a four component vector may be used with the last component representing the w-coordinate. Our OpenCL implementation uses this w coordinate to map to four component OpenCL types. Individual implementations may expand a vertex to contain more complex information such as lighting-normals or texture-coordinates.

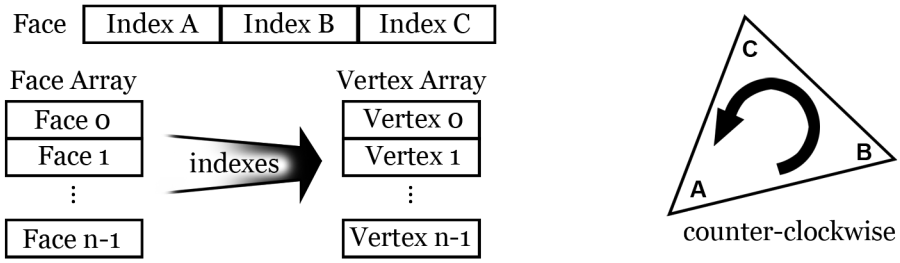

Figure 27: Face definition.

A face (Figure 27) represents a collection of three indexes into the vertex buffer that make up a triangle face. In our implementation, each face is represented by an integer value and ordered in a counter-clockwise winding fashion. Like with the vertex definition, our OpenCL implementation uses four component vectors to contain the faces, with the fourth component aligning our data to some cards' registers. Again, we implement it this way to map to OpenCL four-component data types.

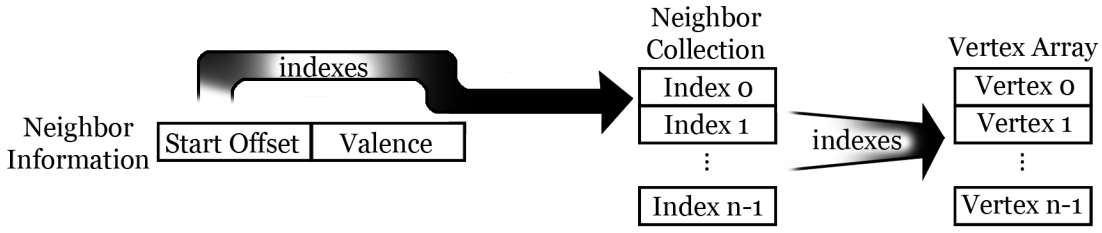

Figure 28: Neighbor information & collection definition.

"Neighbor information" and "neighbor collection" data represent information about an individual vertex's vertex-neighbors (Figure 28). In our implementation, these are split into two

separate arrays because of the unpredictable count of neighbors. A Neighbor information object contains two components. The first component represents an offset into the neighbor collection which indicates the first neighbor. The second component represents the vertex's number of neighbors. This is also known as a vertex's valence. These two components map the dynamic number of neighbors. Then, the neighbor collection is a collection of indexes to the vertex array. Since the neighbor information object maps the data in the neighbor collection, all of the neighbor indexes are compressed into one array. Mapping the neighbor collection requires the information data.

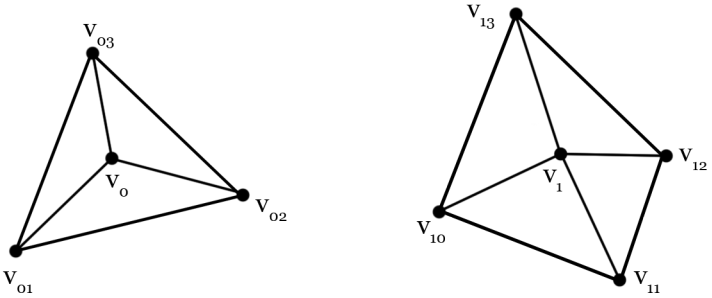

Figure 29: Two arbitrary vertexes and their neighborhood.

To explain this further, consider the two triangular objects in the above Figure 29. For the purposes of this explanation, each object represents a vertex index V_i and it's neighbor indexes V_{ij}. In conjunction with a vertex array, this represents the information we need to know to calculate our smoothed V_i. Notice that the number of neighbors for each vertex is inconsistent. This makes indexing information about neighbors not directly predictable. In order to access the information in our limited environment, we need to unroll this information into a linear format.

In Figure 30 we illustrate how the neighbor information array and the neighbor collection array linearizes the neighbor information for vertex V_i. The dynamic arrays of neighbors are added to a single “collection” list one after another. Their placement and count are then stored in the information array. This information array is can be directly mapped to an individual vertex, which then maps into an array of dynamic arrays.

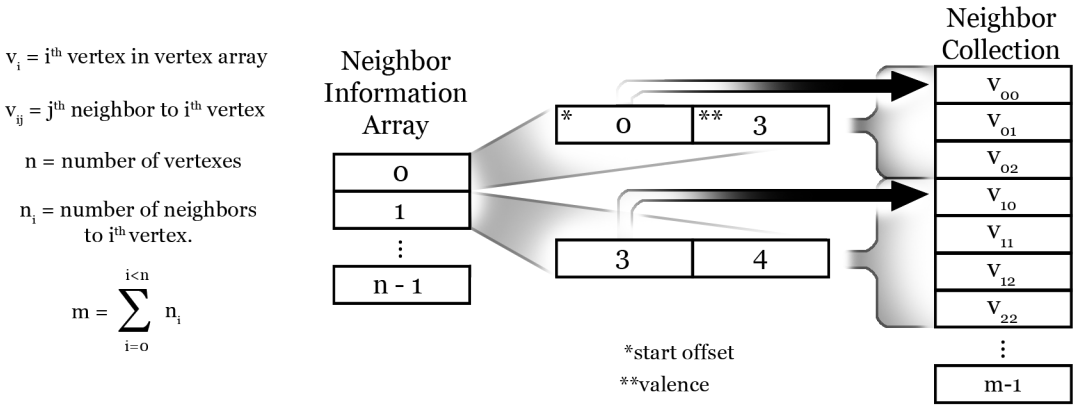

Figure 30: Neighbor information and collection array bakes dynamic nested vectors.

Finally, we collect edge information which we use to generate the new face windings (Figure 31). An edge is represented by four components. The first two components reference the vertexes in the vertex buffer that make the edge. The second two components index into the faces buffer that the edge is a part of. A counter-clockwise representation of the edge information would find face A to vertex A to face B to vertex B. This order of data storage is expected by the winding algorithm to determine consistently ordered faces.

3.0.5 Animation

We use animation to demonstrate the algorithm's efficiency. A deforming low poly-count mesh is provided as input to the subdivision method, which then generates a smooth surface. This works because for most simple mesh animation systems, one of the only changing components is the vertex buffer. Other elements may be keyed, but in our implementation, we just observe mesh deformation.

Our animation system uses an animating buffer. In an external program, a version of the vertex buffer is copied at each frame and keyed with a time. Each mesh with its time-key is then written to a file. The application loads from this file and interpolates between neighboring meshes in time based on an integrating time. Our character used for testing is depicted in Figure 32.

Figure 32: Our animating character in wireframe. Red is the control cage and black is the subdivided mesh.

This method of animation is very simple. It is more common to implement animation as the result of bone animations in time. In those systems, bone animation would be stored to a file and used to transform the low resolution mesh. Input to the subdivision algorithm would not change since deformation would most efficiently be done before subdivision. Animation deformation after subdivision would require more operations to compute.

Upon implementing the algorithm, we find that the parallel winding pass is not required as part of an animation deformation pass. Since topology is not expected to change, it only needs to be run once. While we can not attribute it's parallelism in our OpenCL animated system to a performance gain, we can expect a performance increase by removing it.

3.0.6 OpenCL Implementation

Using the parallel implementation described previously, translation into a GPGPU language such as OpenCL is straightforward. The OpenCL implementation includes three kernels: split, smooth and wind as explained in 3.0.1. Other operations, generally those that bake information needed for the actual subdivision operation are ignored since they only need to be run once. However, the wind operation may also be run only once for animating meshes since the organizational topology will not change from frame to frame. In this case, using the parallel implementation with OpenCL would not be as beneficial.

Most of the communication with OpenCL is done at the start. In this stage: the context is built, the programs/kernels are linked, the OpenCL shared memory objects are created, and all of the initial and constant data is uploaded to the device. These stages assume that the data required for the operation to run have already have been calculated and loaded in client memory so that they may be uploaded to whatever device context is chosen. Furthermore, when the data is uploaded in a recursive environment, each level of data is required for an uninterrupted combination of passes. The varying vertex data at each level is not expected to be set until runtime.

After the initial setup, only a few calls are used for the entire process. These calls encompass uploading new vertex data to the device, running each of the kernels (split, smooth, and wind when appropriate) and then downloading the updated, high-resolution vertex buffer. If recursive subdivision is being applied to the mesh, the output from one level may be switched to the input of the next level without requiring data transfer between the host and device. This adds to our performance gains, since more computation is done in comparison to data transfer in a non-recursive setup.

Our OpenCL implementation uses four-component vectors instead of three-component vectors to represent vertex positions. This is because four-component vectors map more efficiently to certain kinds of hardware. Some cards such as those developed by ATI are vector based hardware and recommend sticking to vectors. In ATI's “ATI Stream SDK – OpenCL Programming Guide” (2010), they note “Use float4 and the OpenCL built-ins for vector types (vload, vstore, etc.). These enable ATI Stream’s OpenCL implementation to generate efficient, packed SSE instructions when running on the CPU. Vectorization is an optimization that benefits both the AMD CPU and GPU” (pg. 4-55). In contrast, NVidia's hardware is scalar based. In their “NVidia OpenCL Best Practices Guide” (2009) they state “in particular, there is no register-related reason to pack data into float4 or int4 types” (pg. 27).

4 Performance

4.1 Approach

Very early in our development, we found our parallel implementation to be usably faster on a multi-core CPU system than a single-core system. However, one of the main motivations for our implementation is to accelerate subdivision via the GPU. In our initial tests, our GPU implementations did not show performance benefits. Upon investigation, we found that the overhead for transferring the initial required data became the primary bottleneck.

Naturally, the volume of required data is positively associated with the number of faces to subdivide. While attempting to shrink the amount of geometry subdivided reduces the amount of data to transfer, the initial overhead from utilizing OpenCL GPU implementations outweighs this optimization. Since neither increasing or decreasing the volume of geometry helped us in this situation, we chose to approach the problem differently.

To reduce overall data transfer, we changed our system from a one-time operation to an iterative, animated system. This reduces the volume we "normally" have to transfer. Since most of the data required is static, we upload the majority of data required only once. After this, only the input and output vertex data needs to be transferred to the device.

While animation reduced the overhead, we found that GPU implementations were less optimal than CPU implementations. Though we believe variations hardware would have varying effects on our result, we wanted to explore the possibilities with common, modest hardware. We describe our testing platform in 4.2.

Upon reviewing our pipeline, we found that data-transfer could be further reduced. Since we upload the resultant data to the OpenGL for rendering, all of the implementations have to transfer their data to the GPU. For our early GPU subdivision implementations, we also required the data to be read from the device before uploading it to the GPU as a renderable object. After

benchmarking the amount of time it takes each implementation to transfer data, we found the most optimal solution would be to keep the resources on the card.

To solve this, we use resource sharing between the OpenGL and OpenCL device. After integrating sharing, we have found our OpenCL GPU meets and marginally passes the multicore CPU implementation on our system.

4.2 Testing Environment

Our primary machine for testing, and the machine responsible for our performance results is a late-2008 model MacBook Pro. The machine contains two graphics processors and a dual core processor. The more powerful card, the NVidia GeForce 9600M GT, is used for OpenCL GPU tests. This device has 32 cores in total running at 1.25 GHz. The CPU is an Intel Core2 Duo running at 2.53 GHz with 2 cores.

When testing our parallel √3-Subdivision implementation, we looked at three primary scenarios: an OpenCL GPU implementation, an OpenCL CPU implementation, and a serial CPU implementation. Each of these implementations used the same algorithm and timers were placed in the same configuration. The OpenCL implementations used the kernel definitions and manage their parallel placement. The serial implementation is a C++ version of the same code. We use the serial implementation as our control group.

We also tested with different source objects. In our first set of tests, we subdivided a uniform triangular sphere every frame. Although animation is not applied to the sphere, constant tessellation simulates this effect. The static sphere was mostly used for checking the validity of the implementation. In our second set of tests, we use an animating character as described in 3.0.5 to demonstrate the actual system. The majority of our statistics come from this animation implementation.

All of the tests are done in conjunction with rendering. Although the parallel algorithm is fast enough on its own to be used for other systems, we are partly limited by our current GPU

implementation. We need to utilize OpenGL/OpenCL sharing which effectively removes the GPU implementation's main bottleneck, memory-management.

4.3 Results

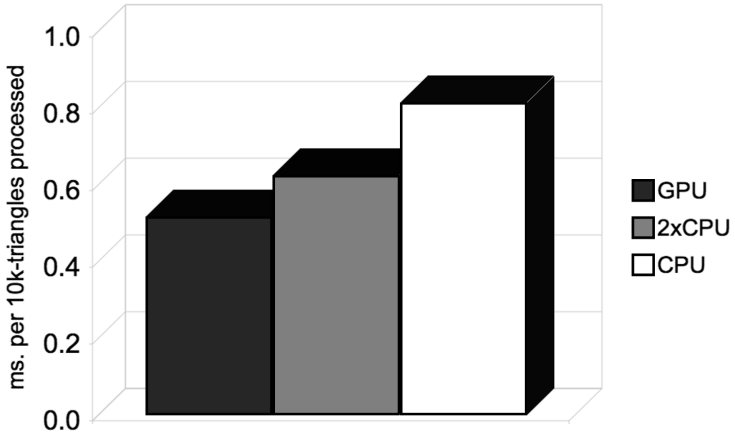

Figure 33: Cost of subdivision. milliseconds per 10k-triangles processed. This includes: parallel operations as well as any data-transfer stages.

After compiling our data we find that, in general, our GPU implementation is more efficient than the other implementations. Figure 33 shows us the general cost of subdivided a number of triangles. We also find that the OpenCL CPU version (2xCPU) which is also parallel is faster than the serial CPU version (CPU). This comparison includes costs for both computation and data transfer.

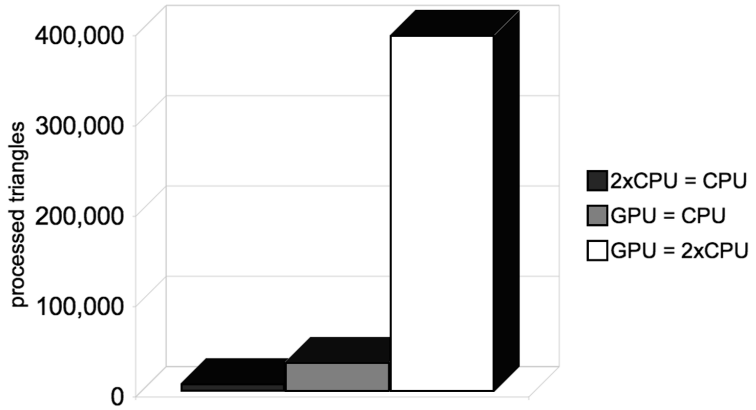

Figure 35: Overcoming overhead. Depicts the intersection point at which one method surpasses another.

Even though Figure 33 depicts the GPU implementation as being faster, it does not include the initial overhead that is a consequence of GPGPU computation and OpenCL's current implementation. Here we can consider overhead as how long it takes for a technique to become efficient. It takes the GPU approximately 32,000 triangles to surpass the serial implementation and almost 400,000 triangles to surpass the OpenCL CPU implementation (Figure 35). Unfortunately, the triangle counts required to make the GPU more efficient than a parallel CPU implementation take our GPU implementation out of real-time frame-rates. However, the parallel CPU implementation surpasses the serial CPU implementation at just under 10,000 triangles.

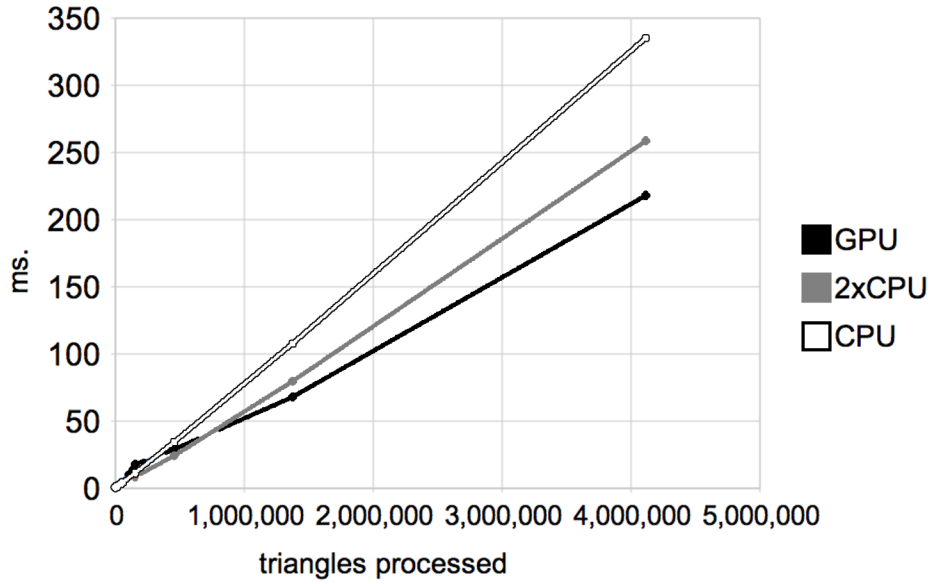

Figure 36: Triangle subdivision compared to time.

Here, performance is depicted as the amount of time it takes to subdivide a set of triangles in each implementation. Note that each technique is relatively linear over a number of triangles over a broader length of time. The information in Figure 36 could be used to dynamically switch between implementations.

Our first GPU implementations required a read from the compute device before writing to the render device. Since the render device and read device are the same in our application, it makes sense to use OpenGL/OpenCL sharing. Without sharing (Figure 37) the average speed was sampled at about twice as slow.

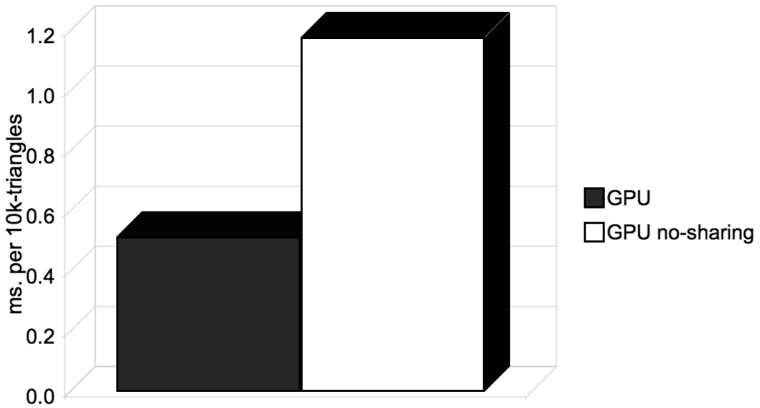

Figure 37: OpenGL-OpenCL sharing. Presents the effects of utilizing interoperability when it is available.

The more triangles processed, the quicker the GPU implementation without sharing quickly separates itself from the shared implementation (Figure 38). The subdivision of more triangles requires larger volumes of data to be transferred from and then back to the card. Host to device transfer speeds are much slower than internal memory transfer speeds.

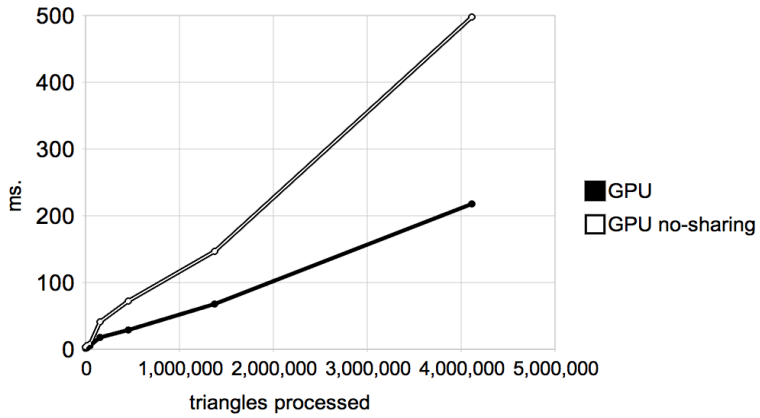

Figure 38: OpenGL-OpenCL sharing over time.

5 Discussion

In short, we believe that the parallel implementation is successful. We have performance gains with both the parallel CPU implementation and the parallel GPU implementation. Here, we are able to show OpenCL provides us performance without any sort of exceptionally unique hardware. However, our experienced performance would most likely vary depending on a platform's implementation.

We also find many limitations that need to be overcome for this particular approach to be implemented into any regular system. Currently our technique only applies to surface subdivision but does not compute many of the common components of a mesh, such as normals and texture coordinates. Also, seams are not addressed. Initial overhead created by OpenCL and the device's transfer speed also limit our ability to effectively integrate parallel subdivision.

5.1 GPU vs CPU

It is fortunate that both the CPU and GPU show reasonable performance. We believe that the performance documented here supports specialized forms of subdivision. For example, it has been shown that the GPU implementation is more computationally efficient than both the parallel and serial CPU implementation. However, initial overhead is significant enough to make GPU computation impractical in many situations. This would indicate support for large, intensive subdivision environments.

OpenCL/parallel CPU implementations provided efficient subdivision at low levels and stayed near GPU subdivision performance even after being surpassed. In most of our cases, the OpenCL CPU implementation is the most reliably consistent and efficient method. Naturally, parallel CPUs are more accustomed to dealing with generic computation. Practically, the parallel CPU implementation is easily adaptable into most software pipelines.

5.2 Realtime Subdivision

In our implementation, on our hardware, realtime subdivision using the GPU is highly practical. In our tests, our models could smooth to relatively high levels. This was all possible in reasonable real-time framerates.

While this is realistic on the GPU, it is more practical at realistic subdivision levels to use the algorithm as a parallel CPU implementation. On our hardware, once GPU subdivision became more efficient than the parallel CPU implementation, real-time frame rates had dropped below 15 frames-per-second. It is generally unrealistic to try and render millions of triangles in real-time.

It is important to reiterate that these results most likely vary significantly from machine to machine. GPGPU devices are organized in a fashion that allows uniform API development, but performance commonly varies. More modern generations of graphics cards have many more parallel cores. Many CPUs are also more parallel than our dual-core.

5.3 Possible Extensions

A realistic system would require the entire pipeline to be optimized in order to receive serious acceleration benefits in subdivided objects. In our project we found that independently computing normals was so slow that it limited any realistic real-time implementation. At higher levels of subdivision, surface normal calculation often took the longest. In this case, it takes only one part of the pipeline to damage the whole system.

We find it is more productive to take the ideas presented in this paper and apply them to the whole system. While advancements may be made to our presented approach, our future research would more extend the breadth of this system as opposed to focusing on the local performance.

When considering what breadth extensions could cover, properties beyond the most basic are not integrated into this algorithm. Practical meshes contain color, texture and normal information. This leads to most meshes containing one or more seams where data is not

continuous. Practical storage methods for realtime applications do not normally contain complex relationships between vertexes and their non-continuous data. This leads to the internal vertex topology often being disconnected, which leads to cracks in a subdivided mesh.

For a mesh supporting these extra features, we may require an advanced relationship between the low-poly control-mesh and a high-poly output-mesh that allows for data seams, while retaining surface continuity. One method for doing this would be to merge vertexes for subdivision, but then separate them when finished. In this case, mesh topology would have to remain the same. Also, It may be necessary to parameterize the surface components so that they may follow with accurate smoothing.

6 Conclusion

We have presented a parallel √3-Subdivision algorithm and considered its possible usage with parallel and OpenCL GPGPU applications. By abstracting the parallel algorithms inherent in the √3-Subdivision algorithm, our technique maps easily to parallel systems. We have observed its implementation in various environments and found generally pleasing results. In order to develop efficient GPGPU implementations, we only needed to overcome memory management bottlenecks and architecture overheads. In conjunction with support for animation, recursion and resource-sharing, we have mitigated many of the bottlenecks often standing in the way of GPU implementation.

We have also provided a background on the study of subdivision and geometrically complex surfaces. Furthermore, we have reviewed the current trends in specialized parallel processing. Since many modern hardware trends are leaning towards geometry-focused features, we believe that supporting research and development can contribute significantly. We also hope this paper has provided enough resources and inspired future extensions for our Parallel √3-Subdivision algorithm.

7 Appendix

7.1 OpenCL Kernels

7.1.1 Split

```
__kernel void Split(//in
                                        __global float4 *vertex_data,
                                        __global uint4          *index_data,
                                        __global uint           split_begin,
                                        // out
                                        __global float4 *smoothedvertex_data)
{
        int i = get_global_id(0);

        float4 centroid = (float4)(vertex_data[index_data[i].s0] + vertex_data[index_data[i].s1] +
        vertex_data[index_data[i].s2]);
        centroid.x = native_divide(centroid.x, 3.0f);
        centroid.y = native_divide(centroid.y, 3.0f);
        centroid.z = native_divide(centroid.z, 3.0f);

        smoothedvertex_data[split_begin + i] = centroid;
}
```

7.1.2 Smooth

```
__kernel void Smooth(//in
                                        __global float4 *vertex_data,
                                        __global uint2 *valence_data,
                                        __global uint *valenceindex_data,
                                        // out
                                        __global float4 *smoothedvertex_data)
{
        uint i = (uint)get_global_id(0);

        // valence_data
        // s0 - valence indexes begin
        // s1 - valence (number of valence indexes)

        uint uin = valence_data[i].s1;
        float n = (float)uin;

        float4 summed = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

        for(uint j=0; j<uin; ++j)
           summed += vertex_data[valenceindex_data[valence_data[i].s0+j]];

        summed.x = native_divide(summed.x, n);
        summed.y = native_divide(summed.y, n);
        summed.z = native_divide(summed.z, n);

        float B = native_divide(4.0f - 2.0f *native_cos(native_divide(2.0f*3.1416f,n)),9.0f);
```

```
        smoothedvertex_data[i] = (1.0f-B) * vertex_data[i] + B * summed;
}
```

7.1.3 Wind

```
__kernel void Wind(//in
                                    __global int4 *edge_data,
                                    __global uint split_begin,
                                    // out
                                    __global uint4 *face_data)
{
        int i = get_global_id(0);

        // edge_data member explanation
        // s0 - vertex index A
        // s1 - vertex index B
        // s2 - face index A - ccw is a->b
        // s3 - face index B - ccw is b->a

        uint4 faceA, faceB;

        faceA.s0 = split_begin + (uint)edge_data[i].s2;
        faceA.s1 = (uint)edge_data[i].s0;
        faceA.s2 = split_begin + (uint)edge_data[i].s3;
        faceA.s3 = 0;
        face_data[i * 2] = faceA;

        faceB.s0 = split_begin + (uint)edge_data[i].s3;
        faceB.s1 = (uint)edge_data[i].s1;
        faceB.s2 = split_begin + (uint)edge_data[i].s2;
        faceB.s3 = 0;
        face_data[(i * 2) + 1] = faceB;
}
```

8 References

“ATI Stream Computing Technical Overview.” *AMD, Inc.* (March 2009): Web. 20 July 2010. <http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf>

“ATI Stream SDK OpenCL Programming Guide” *AMD, Inc.* (2010): Web. 12 July 2010. <http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf>

Bolz, Jeff, Ian Farmer, Eitan Grinspun, Peter Schröder. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid." ACM SIGGRAPH 2003 Papers (2003): 917-924. Print.

Boubekeur, Tamy, Marc Alexa. "Phong Tessellation." ACM Transactions on Graphics (TOG) 27.5 (December 2008): Print.

Cashman, Thomas J, Ursula H. Augsdörfer, Neil A. Dodgson, Malcolm A. Sabin. "NURBS with extraordinary points: high-degree, non-uniform, rational subdivision schemes." ACM Transactions on Graphics (TOG) 28.3 (August 2009): Print.

Catmull, E, J Clark. "Recursively Generated B-Spline Surfaces on Arbitrary Topological Surfaces." Computer Aided Design 10.6: (November 1978). 350-355. Print.

Castaño, Ignacio. "Next-Generation Rendering of Subdivision Sufaces." Siggraph 2008 (2008): Session Slideshow, Digital Distribution.

Damme, Ruud van. "Bivariate Hermite subdivision." Computer-Aided Geometric Design 14.9 (December 1997): 847-875. Print.

Donnelly, William. "Per-Pixel Displacement Mapping with Distance Functions." GPU Gems 2 (2005): 123-136. Print.

Duchaineau, Mark, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein. "ROAMing Terrain: Real-time Optimally Adapting Meshes." IEEE Visualization. Proceedings of the 8th conference on Visualization '97 (1997): 81 - 99. Print.

Kaneko, Tomomichi, Toshiyuki Takahei, Masahiki Inami, Naoki Kawakami, Yasayuki Yanagida, Taro Maeda, Susumu Tachi. "Detailed Shape Representation with Parallax Mapping." Proceedings of the ICAT 2001 (December 2001): 205-208.

Kobbelt, Leif. "√3-Subdivision." International Conference on Computer Graphics and Interactive Techniques (2000): 103-112.

Labisk, U; G. Greiner. "Interpolatory √3 Subdivision." Eurographics 2000: 19.3. Print.

Losasso, Frank, Hugues Hoppe. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." ACM Siggraph 2004 Papers (2004): 769-776. Print.

Loop, Charles. "Smooth Subdivision Surfaces Based on Triangles." M.S. Mathematics Thesis. University of Utah (1987): Print.

Loop, Charles, Scott Schaefer. "Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches." ACM Transactions on Graphics (TOG) 27.1 (March 2008): Article 8. Print.

Loop, Charles, Scott Schaefer, Tianyun Ni, Ignacio Castaño. "Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation" ACM Transactions on Graphics (TOG) 28.5 (December 2009): Article 151. Print.

“NVidia CUDA Architecture Introduction & Overview.” *NVidia Corporation* (2009): Web. 20 July 2010. <http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf>

“NVidia OpenCL Best Practices Guide.” *NVidia Corporation* (2009): Web. 12 July 2010. <http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_BestPracticesGuide.pdf>

"OpenCL." The Khronos Group: Open Standards, Royalty Free, Dynamic Media Technologies: Web. 7 July 2010. <http://www.khronos.org/opencl/>.

Phong, Bui Tuong. “Illumination for Computer Generated Pictures.” *Communications of the ACM* 18.6 (June 1975): 311-317. Print.

Piegl, Les, Warne Tiller. *The NURBS Book* (1997): 2nd Edition. Print.

Seiler, Larry, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Pat Hanrahan. “Larrabee: A Many-Core x86 Architecture for Visual Computing.” *Proceedings of ACM SIGGRAPH 2008. ACM Transactions on Graphics (TOG)* 27.3 (August 2008): Print. Article #18.

Tatarchuk, Natalya, Joshua Barczak, Bill Bilodeau. "Programming for Real-Time Tessellation on GPU." AMD, Inc. (2009): Web. 8 July 2010. <http://developer.amd.com/gpu_assets/Real-Time_Tessellation_on_GPU.pdf>

“The OpenCL Specification.” Khronos OpenCL Working Group (October 6, 2009): Version 1.0. Web, Digital-Copy. <http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf>

"Truform." ATI Technologies, Inc. (2001): Web. 8 July 2010. <http://developer.amd.com/media/gpu_assets/truform.pdf>

Vlachos, Alex, Jörg Peters, Chas Boyd, Jason L. Mitchell. "Curved PN Triangles." Proceedings of the 2001 symposium on Interactive 3D graphics. (2001): 159-166. Print.

Wang, Lifeng, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, Heung-Yeung Shum. "View-Dependent Displacement Mapping." ACM Transactions of Graphics 22.3 (July 2003): 334-339. Print.