Adaptive Displacement $\sqrt{3}$ Subdivision
May 12th, 2015

Jim McCarthy
# Table of Contents

1 Abstract ................................................................. 2
2 Introduction ............................................................ 2
3 Motivation .............................................................. 2
4 Related Works ......................................................... 2
   4.1 Displacement Mapping Techniques ............................ 2
   4.2 Quadrilateral Subdivision ....................................... 3
      4.2.1 Catmull-Clark Subdivision ................................. 3
   4.3 Triangle Subdivision ............................................ 3
      4.3.1 Loop Subdivision ........................................... 3
      4.3.2 $\sqrt{3}$ Subdivision ...................................... 3
5 Overview .............................................................. 4
6 The Process ............................................................ 4
   6.1 Quadrilateral Subdivision and Displacement ............... 4
   6.2 Triangle $\sqrt{3}$ Adaptive Subdivision ....................... 5
      6.2.1 $\sqrt{3}$ Implementation .................................... 5
   6.3 Displacing the Mesh ............................................ 6
7 Tests ................................................................. 7
8 Limitations ............................................................ 7
9 Future Work ........................................................... 7
10 Conclusions ........................................................ 7
11 Results ............................................................ 8
   11.1 Images ............................................................ 8
   11.2 Numbers ........................................................ 12
1 Abstract

Subdivision as a modeling tool has been around for decades, but frequently smooths desired details or amplifies coarse patches. Subdivision influenced by texture maps, such as displacement or normal maps, is often limited to post-processing or shader-driven effects. I propose a tool for generating detail on a mesh during the modeling process. This process uses a slow subdivision technique for more delicate refinement and controlled triangle counts. The process also uses adaptive refinement based on displacement mapped information to increase detail where it can best benefit the mesh.

2 Introduction

Photo-realism and detailed models are at high demand in today’s computer graphics industry. Any method which allows a modeler to get more detail cheaply is important. The computationally cheapest way to increase a model’s detail is to use texture maps to paste detail onto a mesh to mimic tactility, granularity, specularity, and emissivity. Each of the aforementioned qualities has an associated map that goes along with it, the most prominent being diffuse color maps, specular maps, and bump or normal maps. The map that this paper is particularly interested in is displacement mapping which is halfway between bump mapping and actual modeling. Displacement mapping is used for surface details larger than bump maps, but small enough that a modeler wouldn’t want to model it by hand. These maps hold grey-scale values on the texture plane which represent how much to perturb the model’s surface in its normal direction. Displacement maps are easily generated by either painting them (in 2D or directly on the model) or by using a base texture and estimating height data from that texture. These maps lend themselves easily to altering models by creating geometry on the surface and elevating it in the normal direction. The major benefit of this technique is correct silhouettes, which bump and normal mapping do not offer.

3 Motivation

Frequently 3D modeling artists use a bottom-up strategy for design. This entails starting with something as simple as a box or a sphere and using detailing techniques such as extruding, creating edge loops, and direct vertex/edge/face manipulation to build up a model. After the rough form of a model is complete the next step is adding detail, which can be rather difficult. Displacement-map-based subdivision could allow artists to enhance rough base meshes by using a custom displacement map to generate detail.

4 Related Works

4.1 Displacement Mapping Techniques

There are many tools out there for subdividing, the simplest of which is Autodesk’s 3D software: Maya. Maya supports Catmull-Clark subdivision and allows tweaks for smoothing, and subdivision levels. This does not however provide any extra detail to the model (simply more geometry). This type of subdivision actually blurs details such as sharp points and creases. Algorithms to maintain such creasing have been done [Hoppe], but this still adds no further detail. Some have even extended the Catmull-Clark subdivision algorithm to use displacement map details, which allows for adaptive refinement, and keeps a watertight mesh [Bunnell]. However, the quadrilateral patch management (for water-tightness) and the GPU involvement (to increase speed), complicate the implementation significantly. The patch management can also frequently leave the quad mesh with triangles, and working with mixed quad-triangle meshes is undesirable.

The first displacement map implemented was part of the Reyes Image Rendering Architecture and it used micropolygons to displace the surface. The basis behind the method was to subdivide the mesh’s geometry to be about one fourth the size of a pixel from texture space, and then displace the geometry.[Cook]

Figure 1: Displacement surface above a mesh [Cook]

More recently, endeavors have produced progress with raytracing using displacement maps [Pharr]. Rayracing with displacement maps is particularly difficult because tessellating geometry increases the complexity of the scene geometrically and increases the number of comparisons for each ray shot into the scene. Pharr minimized the amount of comparisons by using a voxel-based spatial data structure along with caching to subdivide and store geometry only when a ray enters...
its voxel space, or if the patch is used frequently and cached.

Jan Kautz and Hans-Peter Seidel pioneered an image based, hardware accelerated displacement mapping technique for real-time applications. This method takes the displacement map and creates alpha masks for several heights above the given surface. The algorithm then takes the underlying texture, copies it for each layer, and applies the alpha mask (making a layer-cake-like stack of images). This method was the closest to real time, but suffered because the effect was poor when viewed from a glancing angle. [Kautz]

The final method is the one I will be using in this paper, tessellation. Tessellation takes the base mesh and alters the geometry based on the displacement map (instead of simply approximating a change the the geometry). Most implementations done this way do it from the camera’s point of view instead, because their focus is rendering, but I will do it from object-local space because the focus in this project is modeling.

4.2 Quadrilateral Subdivision

4.2.1 Catmull-Clark Subdivision

Arguably the most well-known subdivision algorithm is the Catmull and Clark subdivision algorithm. In this method, for each face, a new edge point is created for each edge of the face. These edge points are placed evenly between two adjacent face centers and between the two vertices of the shared edge. Vertex points are created for each existing vertex of the face using the formula \[ \frac{Q}{n} + \frac{2R}{n} + \frac{S(n-3)}{n} \] where Q is the average of the face centers around the target vertex, R is the average of all the edge mid points around the target vertex, S is the original vertex, and n is the number of incident edges around the current vertex. [Catmull]²

4.3 Triangle Subdivision

4.3.1 Loop Subdivision

Loop Subdivision is a simple method for subdividing a triangle mesh by a factor of three. For each face, a vertex is created at the midpoint of the three edges. The old triangle is then deleted and four new triangles are created by connecting the edge-midpoint vertices. This makes a triangle in the center of the face, and forms three corner triangles with the old vertices.

The next step is to move the new vertices based on the old mesh. If a vertex is part of a border edge then its new position is an average of its two adjacent border vertices. If the vertex is an extraordinary vertex then it is influenced by its old neighboring vertices according to this formula: \[ \frac{1}{n}v_n + (\alpha)n \] where n is the number of adjacent vertices, \( v_n \) are each of the adjacent vertices and \( (\alpha)n \) is \[ \frac{5}{8} - \frac{1}{n} \times (1 + 2 \cos \left( \frac{2\pi}{n} \right))^2 \]. The method also has rules for crease edges, which are very similar to rules for borders. The crease edge rules maintain important non-smooth features during subdivision, allowing for hard edge models and variable levels of smoothness. [Loop]

4.3.2 √3 Subdivision

Kobbelt developed a triangle subdivision method that increases the mesh’s triangle count slower than the Loop subdivision algorithm. If a mesh is subdivided twice using the \( \sqrt{3} \) algorithm, each triangle subdivides into three new triangles and for this reason it’s called \( \sqrt{3} \) subdivision. \( \sqrt{3} \) is based around three important factors, subdivision generations (or depth), mate triangles, and edge swapping.

A subdivision generation represents how many subdivisions led to the current triangle’s existence. For each time a triangle is destroyed, and replaced with a different triangle, the subdivision generation increases. This triangle creation and destruction happens during both the subdivision process and the edge swapping process. Mates and edge swapping are a more complicated and nuanced part of the algorithm and could be best explained by example.
Figure 4: $\sqrt{3}$ Subdivision example [Kobbelt]\(^7\)

The algorithm starts by creating a new vertex at the center of each (e.g. generation 0) triangle. It then makes edges between the new vertex and each of the triangle’s original vertices. These edges form three new triangles that now all have an odd generation (1). All odd generation triangles must have a mate unless they are border triangles. The mate of a triangle can be found by first finding the center vertex from which it was originally split, then looking at the edge opposite that vertex. The triangle on the opposite side of that edge is said to be the triangle’s mate. When two adjacent triangles are mated to each other and share a generation they undergo a swap. This swap takes the edge between the two, deletes it, and instead draws it from what used to be the center vertex of each of the triangles. [Kobbelt]\(^7\)

5 Overview

The goal was to subdivide a mesh in such a way that the detail from a displacement map is added to a base mesh. I started off using basic quad subdivision, similar to Loop’s subdivision. The quad subdivision created four quads for each one initial quad by splitting edges at the midpoints, creating a center vertex, and connecting it to the midpoint vertices. I used displacement information from the map to determine where the new edge vertices should be placed. Based on those edge vertices and displacement values, I calculated a center vertex between those four points, specifically between pairs of opposite pair-edge vertices. After seeing the unevenness and multitude of divisions for locations with minimal displacement values, I was convinced I should change my method. I decided I wanted an adaptive subdivision scheme. To maintain a watertight mesh and good topology, that also means switching to a triangle-based subdivision algorithm, for which $\sqrt{3}$ has great support. I then rewrote the raytrace assignment’s framework to convert quadrilateral meshes into triangles meshes, subdivide adaptively using $\sqrt{3}$, and use displacement values for triangle subdivision choices.

6 The Process

The beginning of the project was dominated largely by learning how to implement displacement maps, which I learned using the existing raytracing homework’s code. I found that in most cases, finding a U and V coordinate is simply done by taking the U and V of the adjacent vertices and using the correct proportion of them to find the current location.

6.1 Quadrilateral Subdivision and Displacement

Next, I wanted to influence subdivision using this displacement data. I started out by allowing the user to choose an increment variable, then the algorithm would sample that many points along each edge to be subdivided, and finally place a vertex on the sample with the largest differential displacement value. I used a differential displacement value because every vertex would be moved based on its displacement value, meaning that although a point may have a large displacement value, if the two vertices next to it have the same value, then the vertex is in roughly the place it would be if it was not divided. For this reason, I first calculated where the point will be in space when the vertices around it are raised to their displacement height. This is done by interpolating between its two neighboring vertices’ displacement values. I then took the projected displacement value at the point in question (through displacement map lookup) and subtracted from it the interpolated displacement, finding the net displacement to be gained from a subdivision. Next I had to choose how to place the center vertex. The first algorithm was focused on being fast because as the mesh grows large the work becomes geometric, which would suffer if every subdivision had an $O(n^2)$ complexity to choose the best location. I decided to use the user-input increment value to sample the points directly between the two opposite-pair edge vertices. This yielded decent results in finding significant displacement features, but it split the mesh very unevenly and left me with an ugly mesh. The final step after this was to apply transformations to the new vertices based on their displacement values. I did this by duplicating the old mesh and moving each vertex its displacement value distance in the vertex normal direction. The results confirmed my suspicion that the quad subdivision and displacement-based vertex movement were creating a noisy mesh that didn’t improve on the original mesh significantly and needed a more targeted and controlled approach.
6.2 Triangle $\sqrt{3}$ Adaptive Subdivision

I changed my plans to triangles, $\sqrt{3}$ subdivision, and adaptive refinement. $\sqrt{3}$ offered a good controlled way to subdivide at a slower rate than Loop’s method which allowed more control over mesh complexity. $\sqrt{3}$ also offered an algorithm for adaptive refinement which had minimal drawbacks and seemingly simple implementation. The conversion of quads to triangles seemed like it should be simple, but there were some speed-bumps to navigate. The starting point was obvious, to create four vertices (a, b, c, and d) for each of the quads vertices and make two triangles ABC and ACD.

A speed-bump I didn’t realize until later in the project is that when you start converting adjacent quads into triangles, you need to use some existing vertices, not create all new ones. Otherwise, you would have no way of linking the edges of the triangles to adjacent quads. This is extremely important for $\sqrt{3}$ subdivision because of the aforementioned mate triangles which are based on adjacent triangles. The workaround I found for this was putting pointers to triangle edges on the quad’s edges. Then when splitting a quad you get, for example, the QuadA-QuadB edge, looking to its opposite edge to see if it has a triangle edge initialized. If it is set, the TriB-TriA edge’s vertices are used in the construction of this new triangle. I didn’t actually realize this disconnect was happening until deep into the $\sqrt{3}$ implementation because locally, within a test case that was a single quad, those triangles can find each other.

The next step was to create new InitializeVBOs, SetupVBOs, DrawVBOs, and CleanupVBOs methods, which given my limited OpenGL knowledge was difficult because the buffer code can be quite cryptic. I based my code heavily off the radiosity’s equivalent methods, but changed certain branching statements and converted it to triangles instead of quads.

6.2.1 $\sqrt{3}$ Implementation

With the triangle mesh displaying, and the quad meshes hidden, it was time to start work on the $\sqrt{3}$ subdivision algorithm. One thing to note moving forward is that the subdivision-generations for triangles follow different rules. For this reason I personally treat odd generation triangles as half-subdivided because they can’t be subdivided themselves until combining (or edge swapping) with their’s mate. The algorithm is summed up nicely in the pseudocode found in Kobbelt’s paper using two methods, split and swap: [Kobbelt]"
which had a generation two greater than the current triangle will now only be one greater meaning it is an even generation. We now split that triangle, resulting in a mate of equal generation which causes the original triangle to edge swap with the mate, finally splitting the target triangle. This recursion would seem to lend itself to increased complexity, but in practice my recursion depths tended to either be 0 or 1 when going from 1024 triangles to 2048 triangles.

\[
\text{swap}(T_1, T_2) \\
\begin{align*}
&\text{change } T_1(A,B,C), T_2(B,A,D) \text{ into } T_1(C,A,D), \\
&T_2(D,B,C) \\
&T_1.\text{index}++ \\
&T_2.\text{index}++
\end{align*}
\]

The swap method is the second part of the $\sqrt{3}$ algorithm and its importance is to bring the mesh back to the original triangle’s proportions. In Figure 4, you can see the third step of the subdivision has skewed triangles that are no longer near-equilateral and this is undesirable. The swap method takes the common edge shared by two odd and equal generation mates, deletes it, and creates a new edge joining the vertices that were opposite to the original edge.

This function seems simple, but its important to realize that this function is destroying two triangles and creating two new ones which personally led to issues with null pointer problems. The first problem that occurs is in the split function. If you have any additional code after the second split in the odd-generation-else-case then the triangle you passed will be edge-swapped with a neighbor meaning your original split triangle target has been deleted and will be a null pointer (except if it was a border triangle). Another issue is that if you are storing all the triangles in a heap or a vector then you now need to find the two deleted triangles and remove them. After implementing the $\sqrt{3}$ algorithm, I created a heap data structure for finding the triangle with the largest displacement discrepancy, but these unexpected triangle deletions caused a lot of issues maintaining the heap, so I had to remove it.

It is important to realize that with this algorithm, odd generation border edges, and all other odd triangles that recursively lead to that border triangle will never split. This is because the border triangle’s mate is null and it can never edge swap with its mate. I added a boolean to triangles that indicated if a triangle was a border triangle or lead to a border triangle, which eliminates recursions that lead to no progress.

Figure 6: Border Edges Indivisible

One method to counteract this block is to duplicate and mirror the current odd triangle across the mate-edge, and edge swap with that. Then when these two newly even triangles split after an edge swap, the border will line up with the old border (with exception to two triangles that will be hanging off the side which can be deleted). This method was ill-suited to my needs because it requires new U V space (or mirrored U V space), which would not look good. It is also unfavorable when using adaptive refinement as opposed to global refinement because both the swapped triangles need to be subdivided to restore the old border. This is why I kept odd-border edges locked from splitting.

6.3 Displacing the Mesh

The last addition to the algorithm was the addition of displacement values and mesh perturbation. After every split I duplicated the base unperturbed subdivided mesh and loop through all the vertices to displace them. It’s not the prettiest method, but it involves a lot less book-keeping than recording triangles that are deleted and added, as well as making sure the $\sqrt{3}$ algorithm is never affected by the displacement. Keeping only one mesh would also require one to change the displacement map based on the mesh’s new form for every pixel of a face to account for changed elevation, which is quite unreasonable compared to just duplicating a mesh.

As with the prior attempt at sampling displacement values, I struggled with ways to assign a trian-
gle an average (or representative) displacement value. Since the subdivision algorithm divides at the center of a triangle I started by simply taking the displacement value at the center of a triangle and using that. This could, however, miss displacement features in a triangle if they didn’t intersect with the center point. I broadened the estimation by also taking the midpoints between the center point and all the triangle’s vertices into a weighted average, but this doesn’t solve the problem, it just minimizes it for reasonably dense meshes.

7 Tests

My main test was a simple displacement map I made that consists of an S with two dots, all with Gaussian gradient fall-offs to the edges. The two dots were to represent small displacement details that may be missed by displacement sampling and the S was a good smooth feature that would display the adaptive refinement portion of the algorithm well.

I tested rigid edge subdivision by using a solid black and white brick texture. This would test the effectiveness of the displacement sampling within a triangle.

And finally I used a superman logo to test an in-between image that used smooth shading, but also had large blocks of value like the brick texture. This was intended to be a more natural image that had elements of the previous two test images.

I wanted to add tests on new surfaces like a sphere and the bunny model, but formatting the UVs from Maya’s .obj export format to that of the raytracer framework would have taken more time than I had.

8 Limitations

The results are fairly robust and scalable but there are blind spots in the implementation. The most noticeable problem is that displacement sampling needs a better way to sample displacement within a triangle. It currently suffers from the same problem as pixel color-sampling, it senses colors at the center of its sample region but it can miss details that pass by smaller than a pixel.

Another limitation of the project is that it only yields good results for displacement maps that are relatively smooth, have medium to large sized details, and the resulting mesh must be subdivided significantly if a detailed model is desired. You can see these problems exemplified in Figure 9 and 14, in which the solid and contrasty brick texture yielded virtually useless results.

One of the most limiting factors of the algorithm is its inability to handle border triangles. This can lead to some ugly artifacts, like in the S-test I used, one of the edges of the S is close to a border, so a large triangular chunk remained un-subdivided if the initial mesh is not already reasonably dense. Although, this was minimized when I switched from only sampling the center to also sampling the midpoints between the center and the vertices.

9 Future Work

This project badly wants a better displacement sampling algorithm. This would increase the effectiveness of the adaptive refinement feature, and would produce a better detailed mesh at the end for cheaper triangle budgets.

The project could also easily be extended for normal maps instead of displacement maps. I currently have the displacement map stored in a vec3, so the only change needed would be using the normal direction for the perturbation.

The project could also use some optimizations. I mentioned I wanted to use a heap for selecting the next triangle to subdivide, but the complications required me to cut it for time’s sake. If a heap could be implemented correctly it would save on what is currently a linear scan through all the triangles for every subdivision. It would also be beneficial to cut out unnecessary data (as I mentioned I’m using a vec3 for displacement values which could be cut down to a float).

The algorithm also needs an elegant way to deal with border cases to eliminate artifacts around the edges.

10 Conclusions

The algorithm holds up well to simple needs, like extruding medium to big details from a mesh. However, for fine detail it either fails to capture the details or needs unreasonable dense collections of triangles to capture the details. I would consider the project a success because the adaptive refinement works well for medium detailed maps. If not for the displacement sampling issue, I think this algorithm would be quite viable for several personal applications.
11 Results

11.1 Images

Figure 7: Adaptive Snake results 1562 triangles

Figure 8: Adaptive Superman Results 1562 triangles
Figure 9: Adaptive Brick Results 1562 triangles

Figure 10: Adaptive Snake Results Perspective 1562 triangles
Figure 11: Uniform Snake Results Perspective 1562 triangles

Figure 12: Adaptive Superman Results Perspective 1562 triangles
Figure 13: Uniform Superman Results Perspective 1562 triangles

Figure 14: Adaptive Brick Results Perspective 1562 triangles
11.2 Numbers

<table>
<thead>
<tr>
<th>Model</th>
<th>512-640</th>
<th>640-800</th>
<th>800-1000</th>
<th>1000-1250</th>
<th>1250-1562</th>
</tr>
</thead>
<tbody>
<tr>
<td>S and Dots</td>
<td>0.46s</td>
<td>0.75s</td>
<td>1.21s</td>
<td>1.99s</td>
<td>3.37s</td>
</tr>
<tr>
<td>Superman Logo</td>
<td>0.49s</td>
<td>0.77s</td>
<td>1.19s</td>
<td>1.98s</td>
<td>3.35s</td>
</tr>
<tr>
<td>Brick</td>
<td>0.45s</td>
<td>0.74s</td>
<td>1.20s</td>
<td>1.99s</td>
<td>3.30s</td>
</tr>
<tr>
<td>S and Dots Adaptive</td>
<td>1.00s</td>
<td>5.79s</td>
<td>19.60s</td>
<td>45.79s</td>
<td>77.33s</td>
</tr>
<tr>
<td>Superman Logo Adaptive</td>
<td>1.25s</td>
<td>5.20s</td>
<td>20.75s</td>
<td>40.54s</td>
<td>96.85s</td>
</tr>
<tr>
<td>Brick Adaptive</td>
<td>0.60s</td>
<td>0.96s</td>
<td>20.36s</td>
<td>49.02s</td>
<td>96.85s</td>
</tr>
</tbody>
</table>

The numbers in the categories to the right of models are all triangle counts in the form of "before subdivision triangle count"."after subdivision triangle count". All tests performed on a Macbook Pro, Mid-2010, 2.66 GHz, Intel Core i7, 8 GB RAM, 1067 MHz DDR3, NVIDIA GeForce GT 330M 512 MB.

References


