Simplified Mesh Generation from Renders

Thorvald Natvig

Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
Sem Sælandsvei 7-9, NO-7491 Trondheim
Email: thorvan@idi.ntnu.no

Abstract

Scientific visualization presents data or real-world objects in 3D to visually verify and observe results. It has been used in everything from car manufacture to seismic analysis. The 3D models used frequently have far too much detail to be rendered at acceptable frame-rates on current computers, and while the speed of graphics hardware continually increases, so does the resolution of stored data. Quite often, the models displayed include very high resolution submodels that were originally intended to be displayed on their own, like including a fully detailed pressure gauge with all mechanical parts on a large engine. While many methods exist to reduce the complexity of the regular meshes such objects are built from, they all require detailed knowledge of mesh data. The only way to get this data is to integrate extended data export into the tool used to produce the mesh, a task which is difficult unless the software is self-written or comes with full source code.

Presented here is a new method to generate a well-formed approximate mesh with color and normal information, based on just the ability to draw the object on screen. The resulting mesh contains no hidden or intersecting surfaces and hence is perfectly suited for simplification.

1 Introduction

A common problem when displaying scientific models in 3D is that they are too complex, reducing frame-rate below acceptable levels. Especially in virtual reality environments, a frame rate of at least 25 frames per second is necessary to achieve the illusion of real objects. A solution to this problem is to use level-of-detail rendering, which is to have multiple representations of objects in the model, each at a different resolution, and switch between

This paper was presented at the NIK-2006 conference; see http://www.nik.no/.

resolutions depending on distance to the camera. To enable this, a method is needed to generate a simpler (lower resolution) model from a more complex one.

A multitude of different algorithms have been made to reduce mesh complexity, but all of these require the original mesh data to be structured and ordered in a specific way for the given algorithm. Furthermore, they all work on a single model outline at a time, so while the model may contain multiple surfaces, those surfaces are expected to share edges with each other and not intersect any point.

Quite often, the original mesh with all necessary details is unavailable. A common example are programs featuring industry standard VRML or Performer binary exports; these formats seldom contain enough information to do any processing on the mesh. They export the mesh as a series of separate triangles, but without any information on which triangles belong to the same surface.

Even with a fully detailed output, a problem with mesh simplification algorithms is that they generally do not remove hidden geometry, such as objects contained entirely inside other objects. If the object is rendered at a distance (which is the reason we can simplify in the first place), such hidden geometry will never be seen, and hence could be removed. However, mesh simplification algorithms work on one object at a time, and hence will waste allocated geometry complexity on the contained objects.

Likewise, few algorithms handle separate objects that intersect, e.g. a fully modeled nut on a screw where a lot of the allocated complexity will go into modeling the threads on the inside of the nut, even if they will never be visible.

Mesh Simplification Methods

Given a well-formed mesh, numerous algorithms can dynamically reduce its complexity. All of these work on the same basic principle of creating one or more new meshes $\mathbf{M}'$ which minimized a user-defined error-metric $\varepsilon(\mathbf{M}, \mathbf{M}')$.

One of the best performing algorithms is "Progressive Meshes" by Hughes Hoppe [1], with data formats described in detail in "Efficient Implementation of Progressive Meshes" [2]. Numerous improvements on the original algorithm have been made, in particular the use of Garland et.al's "Quadric Error Metrics" [5] which introduce a better error metric for the difference ε, and Lindstrom's memory-reduction scheme [6].

Numerous implementations of Progressive Meshes exist in everything from hardware 3D rendering packages (D3DXPMesh in Microsoft's Direct3D) to raytracing applications (Maya, 3D Studio). In particular, its ease of use in Direct3D games has pushed it forward as the current de-facto standard.

Progressive Meshes

The basic idea in progressive meshes is to construct a multi-resolution representation of the model by iteratively collapsing edges. The simplified (base) model is then stored, along with the inverse of the edge collapses.

When the model is rendered, the base model can be refined to the desired resolution by performing edge splits (the stored inverse edge collapses). Furthermore, it is possible to gradually split a vertex, making the transition from $\mathbf{M}_n$ to $\mathbf{M}_{n+1}$ seamless (where n is the number of edges in the current representation).

To recreate the object as progressive meshes, one needs both discrete attributes for the faces, and scalar attributes for the vertexes. To represent the basic geometry of the mesh, 3D coordinates for all vertexes are hence needed as well as the vertex connectivity (edges). A face (polygon) is defined by its bounding edges. Each face needs material information, which is either color or texture. In addition, each corner of the face requires a normal.

Creating the structured mesh

This paper will detail a new method to regenerating a structured mesh of any 3D object that can be rendered, using only screen grabs of renderings from different angles and different lighting conditions. The resulting mesh will satisfy the constraints of most mesh simplification methods and hence make it easy to compute a lower complexity version of the model. In addition, as it is based on visual representation, any hidden geometry will be removed and intersecting objects will be merged into one.

The generation of meshes from depth images is not new. Much research has been done in this field over the years, since it is a basic requirement for turning laser-scanner images of real objects into computer-based 3D models. Sun et.al. have published a paper on mesh-based reintegration with color [4]. The major new contribution in the method presented in this paper is the accurate regeneration of the normals present in the original object, instead of just recalculating them based on surface orientation. Many objects actively use normals that are not necessarily "normal" to the surface to achieve visual effects, and that information will be accurately reflected here.

2 Method overview

To recreate the geometry of the object, the object is drawn from all 6 directions (top, bottom, left, right, front and back) with a orthographic projection matrix and then the depth buffer is polled to create a volumetric model from which surfaces are constructed. This will give a good enough model of the outer surface to recreate vertex and 3D surface information. This is explained in detail in Section 3.

To recreate basic color information, the model is lit with ambient lighting only and rendered from the same 6 directions, but this time the color buffer is polled. As the object might occlude details of itself that will not be visible when only considering the 6 angles, multiple slices are rendered if such occlusion is detected.

To recreate normal information for accurate lighting, the process to recreate color information is repeated, but this time with diffuse lighting from multiple directions. By computing the difference in color intensity between the ambient and diffuse cases, the normal can be computed and stored. Details on this are in Section 4.

With the geometric, color and normal information, vertexes and surfaces are then created.

3 Generating basic mesh geometry

In order to regenerate the basic geometry, one first needs to decide on a sampling resolution (n), and assume that the object is to be represented inside a box of n^3 evenly spaced vertexes. The overall strategy is then to first use OpenGL depth rendering to create a volumetric model

where each such vertex is "present" or "not present" (section 3), and then turn that volumetric data into proper 3D surfaces (section 3).

Creating the 3D volume

Start defining a volume consisting of n^3 tiny cubes ("sub-cubes") where all cubes are initially filled. Render the original model from each face of the box, grabbing a depth image each time. In these depth images, each pixel is actually the depth of the model at that point (between 0.0 and 1.0), so if the pixel at position $0,0$ of the front depth-image is 0.5, this means we can remove the sub-cubes from $0,0,0$ to $0,0,n/2$.

Surface detection

The nest step is detecting which parts of the volume are part of a surface and hence will need color and normal information. This is simply the elements that are still filled but which contain a cleared neighbor in any of the 6 directions.

In addition, we now create a separate data-structure for each cube and replace the true/false state in the volume model with an index representing the sub-cube number.

OpenGL rounding problems

Unfortunately OpenGL has a few implementation problems which introduce inaccuracies in the depth rendering stage. As we wish to work with models for which we lack the underlying data-structure and only have a method of displaying, we need to work around these inaccuracies. It should be noted that these problems are not OpenGL specific, but are present also in e.g. Direct3D. The worst problem is that we cannot easily determine if these inaccuracies are there or not, so we simply have to assume they are always present.

Imagine a single dot at coordinate $0.51, 0.51, 0.51$ in an orthogonal view from $(-1,-1,-1)$ to $(1,1,1)$. If this dot is rendered to a 4*x*4 pixel viewport, it should be placed at coordinate $3,3$, the properly rounded position. Unfortunately, some OpenGL implementations will just truncate the value rather than round it, placing it at $2,2$.

If using a 32-bit depth buffer, the depth buffer has more than enough precision to place the depth quite accurately at 0.51. If we use proper rounding for the sampled Z buffer, the depth should be 3, and so the final coordinate is $(2,2,3)$ with an approximate x and y position and an accurate Z position. However, if we follow the algorithm and remove everything but this coordinate, we run into a problem when rendering from the left side. There we'll have accurate x positions (as that will be the depth buffer), but only approximate y and z, and we end up removing the last sub-cube.

This problem is easily solved by adapting the rounding of the depth value to match OpenGL, but as mentioned it is unfortunately different from implementation to implementation. To make matters even worse in the truncating case, that same dot, if rotated 180 degrees around the Y axis, will be placed at coordinate $0,2$ and hence it is not in the same place when viewed from the front as the mirror-image of the view from the back.

Even with proper rounding, the same end result can be observed if the modelview matrix has had a large number of transformations (rotations or translations). The limited precision of the 32-bit floats used to represent vertexes means that rotating 1 degree 180 times is not the same as rotating 180 degrees. When just viewing the object, any such inaccuracies will be

equally applied all over the object and hence the viewer will not notice it, but as we use the image for processing we require as accurate a representation as possible. The partial solution we used is to use just one transformation for each rotation, and then reinterpret the direction of the axis of the resulting image (flipping horizontally/vertically).

For any kind of object with volume, none of these faults introduce any real problems, just very minor inaccuracies in the model. The problem arises with surfaces that have no thickness. For example, the sprout of the Utah teapot as defined in GLU is a single surface; if you look down into the sprout all you see is the back-side of the polygons defining the sprout exterior. This unfortunately means that part of the sprout is removed by this method. Increasing to resolution will not help, as the surface will always be just a single pixel thick. For an example of this, see Figure 1.

Original model Model after regeneration with cubified method.

Figure 1: Rounding problem: Surfaces without any thickness will unfortunately be removed.

To work around this problem, we mark detected cubes so that they will not be removed by subsequent depth images. For example, if the image indicates that we should clear from $(1,1,0)$ to $(1,1,7)$, we will mark $(1,1,7)$ so it will remain intact in subsequent passes. Note that as these preserved edges are not visible from all directions, the color and normal information for them might be off.

4 Capturing color and normal information

Color information is captured in much the same way depth information is. A simplified version of the OpenGL lighting model [3] with ambient lighting is expressed as:

$$I = L_d \cdot M \cdot cos\theta + G_a \cdot M$$

where I is the reflected intensity, L_d is the light diffuse color, G_a is the global ambient light color, M is the color of the material, and θ is the angle between the light vector $\vec{L}$ and the normal of the surface $\vec{N}$.

Our goal is to compute M and $\vec{N}$ by sampling with various values for L_d, G_a and $\vec{L}$.

Capturing material color

Renderings from each of the 6 sides are done with ambient white light only, so that $I_a = M_a$. Taking the render from the front, the color at pixel i, j is assigned to the front face of the first nonzero cube along $i, j, 0 < k < n$. Repeating this for all 6 faces, the model has color information for all faces that were not occluded by the model itself. However, imagine 2 identical large cubes placed along the x axis. Cube A will occlude the left face of cube B and vice versa. To find such surfaces, the volume-model is iterated, and if a mini-cube is found

that is missing color information on a face where it does not have a neighbor, we render again with the clipping plane adjusted to start at that point, and iterate as necessary until the model is complete. In an absolute worst-case scenario, this will require n renderings for each facing.

Note that this problem of "missing surface" also applies to the mesh generation itself, but if we iterate there as well we will recreate internal geometry that would be mostly invisible from the outside, a feature we do not want when the goal is to simplify the mesh.

Capturing normals

To capture normals, the process of color capture is repeated, but this time 6 renders are done, each with diffuse lighting from a different direction.

The diffuse part of the lighting model states

$$I_d = L_d \cdot M_d \cdot cos(\theta)$$

Since θ is the angle between the normal and the light, $\vec{N} \cdot \vec{L} = |\vec{N}||\vec{L}|cos(\theta)$, both $\vec{N}$ and $\vec{L}$ are normalized and we always use white light, we can simplify this to

$$\frac{I_d}{M_d} = \vec{N} \cdot \vec{L}$$

If $I_d > 0$ when light is applied from the front, it is naturally 0 when light is applied from the back, and vice versa for the other sides. To help calculations we create a relative intensity, defined as

$$R_{front,back} = \begin{cases} \frac{I_{d(front)}}{M_d} & \text{if } I_{d(front)} > 0 \\ \frac{-I_{d(back)}}{M_d} & \text{otherwise} \end{cases}$$

and similar for the other sides.

If we assume that $M_d = M_a$, we can use the 6 measured values of I_d with their corresponding $\vec{L}$ to set up a series of equations as follows

$$\begin{array}{lcll} R_{front,back} & = & [N_x N_y N_z] \cdot [0,0,1] & \Rightarrow \quad N_z = R_{front,back} \\ R_{left,right} & = & [N_x N_y N_z] \cdot [1,0,0] & \Rightarrow \quad N_x = R_{left,right} \\ R_{top,bottom} & = & [N_x N_y N_z] \cdot [0,1,0] & \Rightarrow \quad N_y = R_{top,bottom} \end{array}$$

This process is then applied to all sub-cubes. Implementation-wise, this should be interleaved with the captured of material color, as we have to iterate through sub-planes here as well.

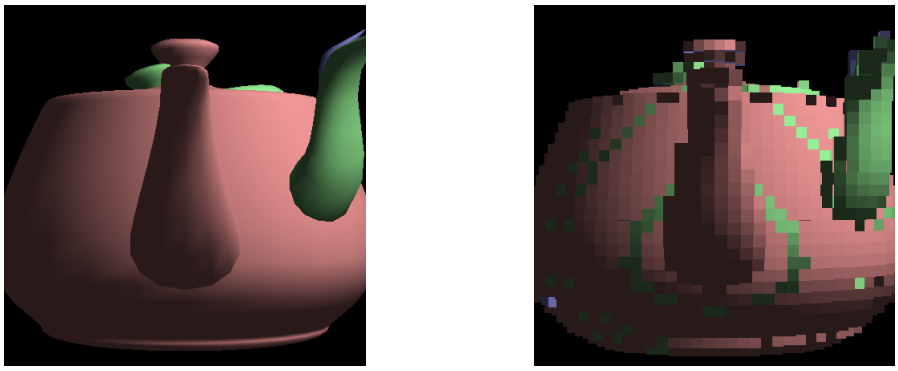
The problem of clipping

OpenGL clipping does not work as a per-pixel solution. Rather, it simply disregards vertexes that are outside the clipping plane, which means any polygons including that vertex will be discarded.

Imagine a red plane at $z = x$ and a blue plane at $z = x - 0.5$ and bound by the viewing box. On the first pass, the clipping plane will be at the very front of the box and both planes will be rendered, with the red in front, and we correctly capture color information for all the sub-cubes along the red plane. On the next pass, we move the clipping plane further into the box

to capture the front-face of the sub-cubes belonging to the blue plane. However, as the bounds of both planes were identical, their starting Z depths are also identical, and our clipping plane just clipped away the entire blue plane as well as the red plane. For an example of this effect as seen on the twopot model, see Figure 2.

We try to work around the problem by using multiple clipping positions around the target area, then choosing the closest value obtained. There other solutions to this problem, for example adapting the sampling algorithm to split each of the 6 surfaces into smaller areas, as this will reduce the chances of this problem appearing. Another solution is to use a pixel shader to implement the clipping, rejecting pixels based solely on their eye-space Z value. While this requires more advanced hardware, it will provide better results.



Original model Regenerated model without adjusting for clipping

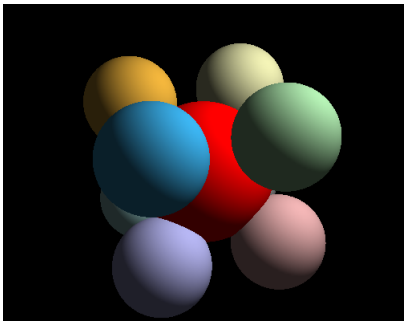
Figure 2: Clipping problem: Multiple Utah teapots reveal the interior teapot rather than the exterior.

Back-facing polygons

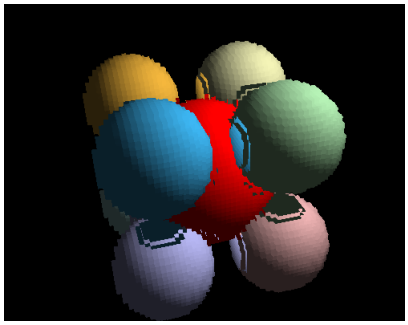
Good models implement support for back-face culling. In OpenGL, a face is said to be front-facing if it is screen-transformed vertexes are in a counter-clockwise order. If they are not, we are looking at the backside of the polygon and can disregard it. Unfortunately most models aren't good models; for example most of the GLU example models (spheres and teapots) do not properly define polygons facing, and so back-face culling has to be turned off.

This turns into a problem when we optimize the number of passes necessary to render. Imagine two cylinders standing on the xz-plane, placed along the z axis, with a red cylinder A at $(0,0,-5)$ and a blue cylinder B at $(0,0,5)$ with each cylinder having a radius of 2. On the first pass, we'll capture color information from the front-face of cylinder A. Now, assume there is some other geometry present in the scene which causes the next capture-depth to be at -4. Starting at depth -4 and going into the model, the next front-facing cube waiting for color assignment is part of cylinder B. However, the color will be red, as what is present in the color buffer are the back-facing polygons of A. A similar problem occurs if there should happen to be an object inside cylinder A; our volume model will not include it, but it will become visible once we start clipping. See Figure 3 for an example of this effect on the spheres model.

To work around this problem, we always check the actual depth of captured color information and compute the difference between the depth of the sub-cube and the depth found in the rendering. By only updating the color of the face if this difference is less than on the



Original model



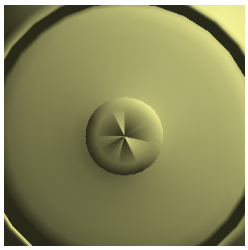
Regenerated model without verifying depth results

Figure 3: Back-facing or interior polygons that are closer to the eye than the volume model reflects lead to the wrong color being assigned to sub-cubes.

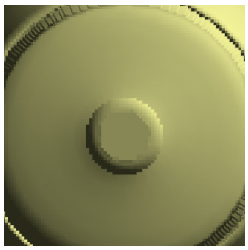
previous assignment, the "most accurate" color is used. This works well for most models, but for models where interior surfaces are very close to the exterior (like the twopot example), this will cause the interior surface color to be used in the case where that surface is closer to the edge of the sub-cube. We therefore recommend to try both methods and choose one based on visual inspection. It is important that the final model uses counter-clockwise winding polygons, meaning back-facing polygons can be culled and ignored by the 3D hardware.

Invalid normals

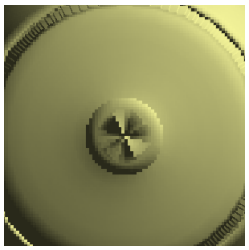
Models that are defined with invalid normals ($length \neq 1.0$ or pointing into the surface), will have invalid normals in the regenerated mesh as well, but those normals might be invalid in other ways. For an example of this, look at the top of the Utah teapot in GLU, which comes with normals pointing into the teapot (Figure 4). The figure also shows what the result looks like if you recalculate the normals based on surface geometry (done just for the knob).



Original model with invalid normals (knob should be smooth)



Regenerated model, but with calculated normals for knob



Regenerated model accurately reflecting invalid normals

Figure 4: Invalid normals are accurately reflected in the regenerated model.

5 Results

To verify that the generated data is accurate and indeed similar to the original, we iterate all the sub-cubes, and draw a tiny square for each defined face of each defined cube. While this model has a high complexity, it is a perfect input for any mesh simplification program, and allows for

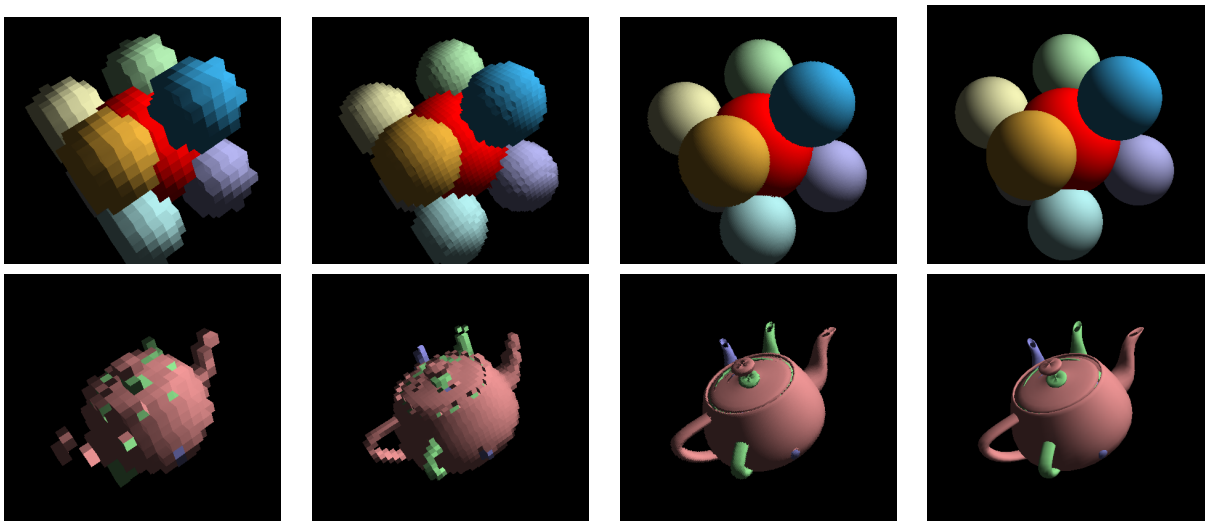
easy visual comparison. Table 1 shows geometric accuracy after complexity reduction with a progressive mesh of both the full original mesh and the regenerated one using the Utah teapot model, and Table 2 shows the same for the spheres model. Figure 5 shows the effect of different sampling resolution for both models.

Edges	Original	Cubified (n=160)	Cubified (n = 40)
782	0.000	0.031	0.092
500	0.011	0.047	0.121
250	0.082	0.096	0.135
100	0.107	0.118	0.156

Table 1: Creating progressive mesh based on the Utah teapot and a regenerated Utah teapot: Average error in final simplified mesh.

Edges	Solid Geometry	Cubified (n=400)
5000	0.001	0.001
500	0.007	0.008
250	0.022	0.022
100	0.036	0.037

Table 2: Creating progressive mesh of spheres object constructed via constructive solid geometry or the cubified method presented here: Average error in final simplified mesh.



$n = 20$ $n = 40$ $n = 160$ Original

Figure 5: Visual effect of sampling resolution

6 Discussion

Validation of the method was done by taking the Utah teapot and run both the original model as well as the cubified regenerated mesh through a progressive mesh generation system. The error

metric used was sum of distances from each point p in the original model to the closest surface in the simplified model. p was chosen as all the vertexes as well as the center of all polygons. Edges are the number of edges left in the original mesh; all three meshes are simplified to this level before comparison. This error metric only measures the correctness of the geometry; it does not take color or normals into account.

Please note that the cubified model has a very high complexity due to its high number of sub-cubes, and so has to be simplified by about 90% just to match the complexity of the original. As almost off these reductions are of the trivial kind (removing the middle vertex of 3 vertexes along straight line), it doesn't affect visual quality, but does increase the time it takes to generate the progressive mesh.

The original mesh for the Utah teapot had 782 edges, so naturally the progressive mesh fully expanded doesn't have any errors at all. The problem with the hole at the end of the sprout due to single-sided polygons unfortunately means the cubified model has errors even before being simplified at all. The $n = 40$ cubified model really has too low sampling resolution to include all the features, as can be clearly seen by its high initial error metric. As complexity is decreased, the difference between the original and the cubified model decreases, clearly indicating that our cubified approach works.

To avoid the problems of the teapot and test the method with a well-behaved object, the "spheres" test-object was recreated using solid geometry in a 3D modeling package and then converted to a single high-resolution mesh. As both the solid geometry model and the cubified model were constructed with a very high polygon count (a little over 5 million), they are both greatly simplified before they reach the 5000 edges target, and the computation took a little over 30 hours and consumed 2.2 GB of memory for the cubified data. However, as this clearly shows, given sufficient resolution the cubified approach very accurately approximates the original.

Sampling resolution

The number of sample points along each axis (n) define the sampling resolution. As no feature on the model smaller than a sub-cube will be represented, it is important to have n large enough that all needed visual details are included. At the same time, computation scales as n^3, so n cannot be defined to an arbitrarily large amount.

Figure 5 shows different sampling resolutions for both the spheres and the twopot object. The cubified model is shown without color smoothing so the actual cubes can be seen.

In general, it makes sense to define n such that it is less than twice the maximum pixel size the simplified object is to be rendered at. So if the finished object will never be displayed with a width or height more than w, then $n = 2w$ will make sure all the details that would have been visible in the unsimplified model are also visible in the simplified model of the cubified result.

Future work and improvements

Some preliminary work was done to use more advanced mesh creation algorithms than simply connecting filled sub-cubes. Some initial work has been done using adaptive Marching Cubes with good results, but in the end even better results were seen by simply increasing the sampling resolution and running an extra pass of the progressive mesh simplification algorithm.

On today's graphics processors, it would be more efficient to reduce the complexity of the model even further, and use texture and normal maps with a pixel shader to recreate the look of the original model. Creating texture maps are easy (store a reduced-resolution version of the color-samples), and the normal maps can be generated from the 3d volume model at the same depth points as the chosen texture maps.

There are a lot of optimization tasks left in this method. For example, if we cache the last and next depth for each pixel when searching for the next slice for color capture, we would speed up the computation part of color capture tremendously. It would also be a good idea to just store the needed slices after the first pass, instead of recomputing it for each pass of directional lighting.

It should be possible to not store the entire volume, but just store the depth samples and calculate the data as it is needed. This would enable much higher resolution without memory scaling as n^3.

7 Conclusion

A new method for recreating a well-formed regular mesh based purely on a method to display an object was introduced. The method can convert any 3D object that can be rendered into a fully detailed mesh, without any prior knowledge of the object or the data it is based upon. Hidden geometry and features smaller than the desired resolution are removed; making sure all of the geometric complexity is devoted to visible features.

As results show, the method accurately captures mesh geometry as well as color and normal information with enough detail to create an effective simplified model of the object for inclusion in larger scenes.

The method is a general approach and can easily be extended to cover other attributes, such as specular lighting, or to use alternate methods of surface representation, such as texture and normal maps coupled with an even lower complexity mesh.

References

[1] H. Hoppe: *Progressive meshes*, ACM SIGGRAPH 1997, pp 189-198.

[2] H. Hoppe: *Efficient implementation of progressive meshes*, Computers & Graphics, Vol. 22, No. 1, 1998, pp 27-36

[3] *GL Red Book*, Addison-Wesley Publishing Company, 1993, ISBN 0-201-63274-8.

[4] Y. Sun, C. Dumont, M. A. Adibi: *Mesh-based integration of range and color images*, SPIE Conf. on Sensor Fusion, **Vol** 4051, pp 110-117.

[5] M. Garland and P. Heckbert: *Surface Simplification Using Quadric Error Metrics*, Proceedings of SIGGRAPH 97, pp 209-216.

[6] P. Lindstrom and G. Turk: *Fast and Memory Efficient Polygonal Simplification*, IEEE Visualization '98, pp 279-286.