Texturing the Tetrus

CS285, 2006 - Final Report - Steven An

Introduction

The purpose of my project was to try various methods of texture mapping the tetrus mesh. Then, I could use those methods with a height map to generate solid, geometric details over the tetrus. The main method I wanted to try is the one presented in Wei & Levoy [1]. That method takes a 2D texture sample and automatically synthesizes textures over arbitrary geometry that look like the original sample. A major part of my project was to implement this for the tetrus. Professor Sequin also had a way of texturing the tetrus for certain kinds of textures, so I wanted to implement that as well.

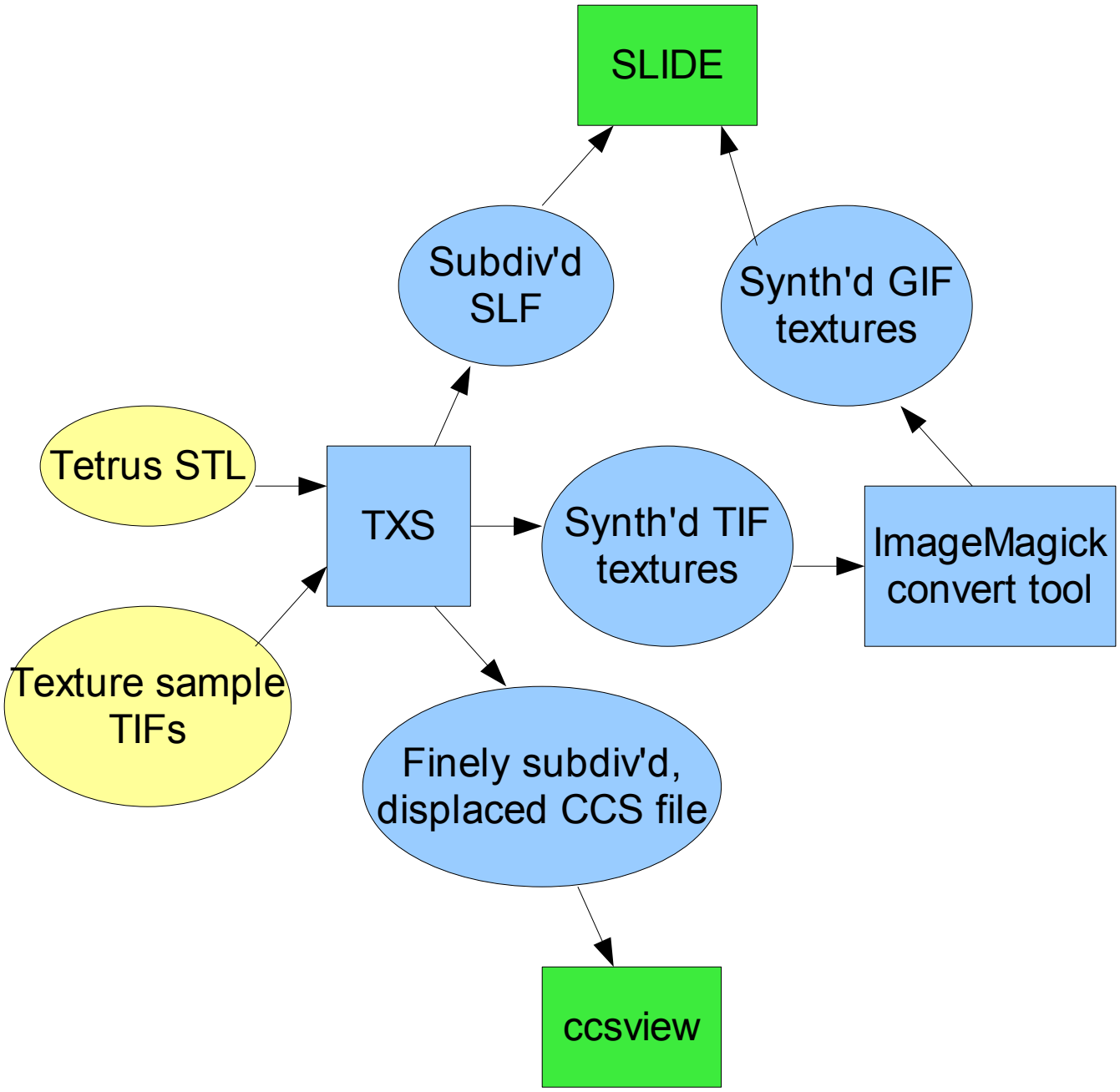
Once the tetrus has been textured with a height map, the height map must be applied to the actual geometry. Because the original tetrus mesh may not be fine enough, it must be subdivided in order to adequately express the height map details. Thus, the second part of my project was to implement Catmull-Clark subdivision surfaces and displace the vertices using the height map.

Overall Method and Process

My original plan was the following: Save the coarse tetrus mesh (provided by Prof. Sequin) from SLIDE as an STL. Write a program which takes the STL and texture samples and emits a SLIDE SLF file with the texture map and UV coordinate information. It would also emit the actual texture maps as .GIFs, of course. Then, I would modify the SLIDE source code's subdivision code to keep the UV coordinates so I could subdivide the texture mapped surface. Finally, I would modify the code to displace the subdivided mesh by the texture maps (which would act as height maps).

However, SLIDE turned out to be more trouble than it was probably worth. Me and a fellow student were both unsuccessful in modifying SLIDE's subdivision code. After discussion with Prof. Sequin, we decided it was better for me to just implement subdivision on my own. This turned out to be a good idea for various reasons. So in the end, I basically do everything in my own program, including subdivision and displacement. I only use SLIDE to view the texture mapped mesh, which is pre-subdivided by my program. The final work flow looks like the following:

(flow chart on next page)

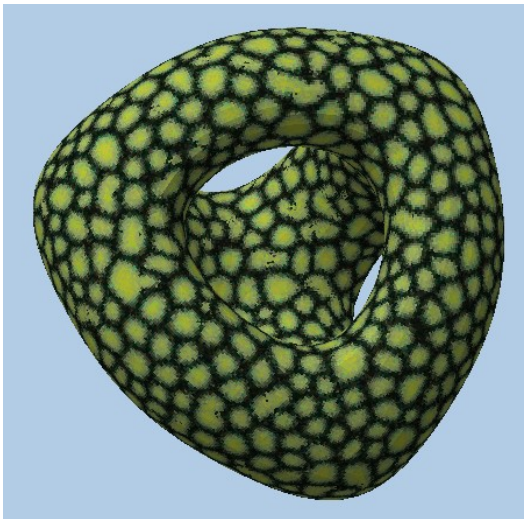
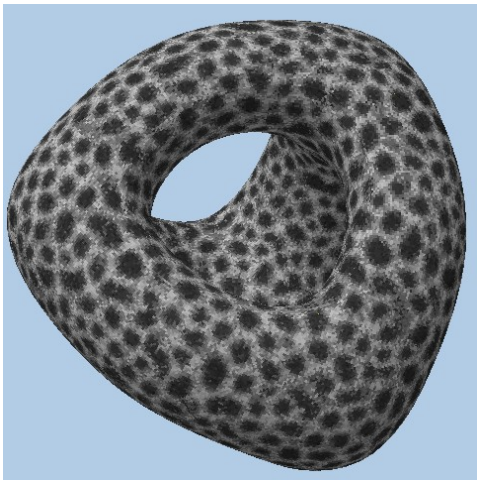
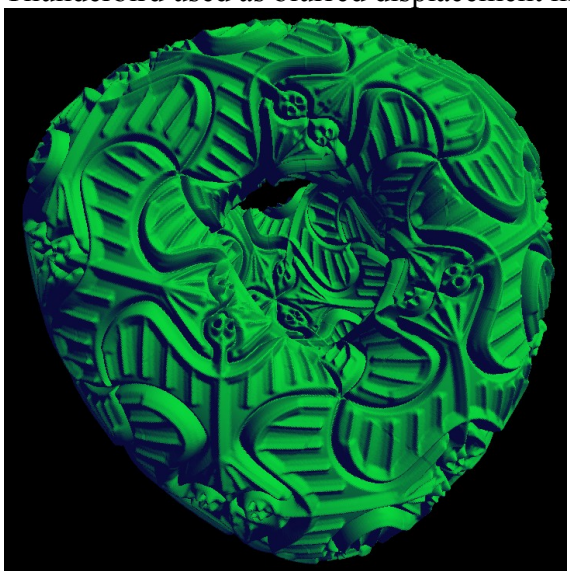


So my program (TXS) takes an STL and texture sample *.TIFs, builds a connectivity graph from the STL, performs the synthesis, outputs it to a SLIDE SLF file and *.TIF files, and finally outputs a subdivided and displaced *.CCS file. Then, I use ImageMagick to convert *.TIF's into *.GIF's so SLIDE can read it. To view the CCS files, I wrote a quick *ccsview* program especially for that.

Final Results

I succeeded in implementing most of Wei & Levoy's method, but not all of it. My code is able to synthesize an-isotropic textures over the tetrus, but not isotropic ones. This is because I could not get the orientation relaxation procedure working in time. I did implement Prof. Sequin's texturing pattern

for the thunder bird texture, and the subdivision with displacement is fully functional. I also built a specialized, light-weight viewer for the displaced subdivision surfaces, since they contain many facets, and SLIDE cannot practically load them. Some screenshots of the best results follow.

Wei & Levoy's snake skin synthesized over the tetrus: 	Wei & Levoy's Brodatz texture synth'd: 
Thunderbird with procedurally assigned UV-coordinates 	Thunderbird used as blurred displacement map 

All results were done with 4 Gaussian pyramid levels with neighborhood sizes of 3, 5, 7, and 9.

Wei & Levoy Implementation Details – Texture Map Generation

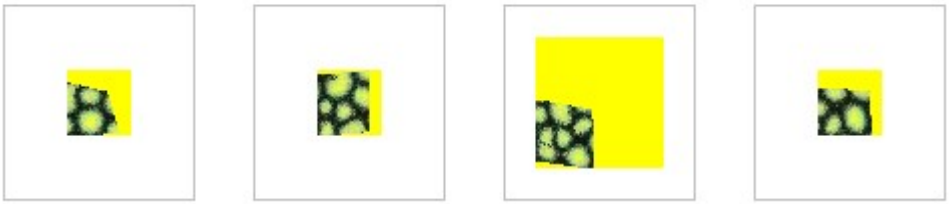
I will describe my particular implementation of Wei & Levoy's [1] method for the tetrus. I assume the reader is familiar with the algorithm**.**

The main difference between my implementation and Wei & Levoy's method is how the colors are actually assigned to the mesh. In their implementation, they re-mesh the input mesh to very fine levels and make sure that all triangles are about the same size. I chose not to do this, because implementing the re-meshing methods is probably very difficult. With the re-meshed input, they actually assign

colors to the individual vertices. When then render the synthesized result, they render the triangles with colored vertices and smooth-shade the triangles. So no 2D texture maps are actually created.

Originally, I figured I could just use Catmull-Clark to subdivide the tetrus mesh, and then assign vertex colors to that. However, after discussing this with Prof. Sequin, we decided it may be more efficient to actually generate 2D texture maps for the coarse tetrus mesh and perform the synthesis over the pixels of the maps. This is the approach I decided to take.

For each face in the mesh, I build a 2D image. I have a global setting that controls how many pixels per unit there should be, so the image sizes are determined by that value. For the tetrus mesh, which is quite small, I use about 1000 units per pixel. From that, I figure out how big the face is and determine what the size of the texture map should be for it. I make sure to keep my sizes in powers of 2 so I can halve them to make lower levels of the Gaussian pyramid. So actually, each face has a full 4-level Gaussian pyramid allocated. Some examples of the 2D images are shown below. The yellow pixels are the ones that don't map onto the face.



When I begin the algorithm, I essentially collect, over the whole mesh, all texture map pixels that actually map to some location on the geometry. Not all faces are perfectly square, so some pixels in the texture maps will not actually show up (the yellow ones). After I accumulate this giant list of "pixel locations," I begin the synthesis over them.

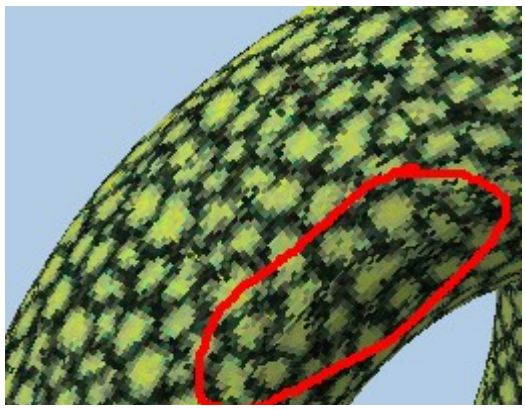
Note that because I did implement the Gaussian pyramid aspects of the algorithm, I actually accumulate pixel locations for each level of the pyramid. When I perform the synthesis for one level, I only use pixel locations from that level.

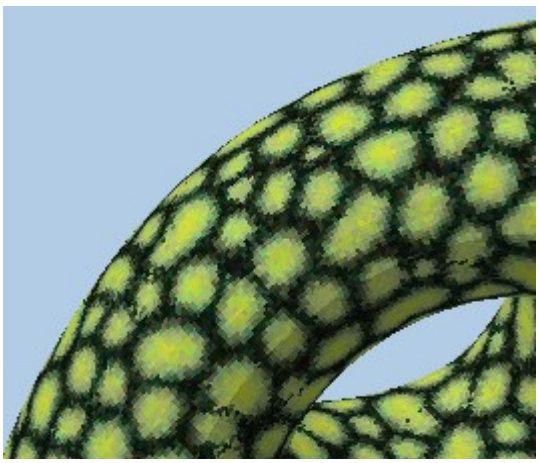
Wei & Levoy Implementation Details – Neighborhood Sampling

The second major difference in my implementation is how I determine the neighborhood of a given pixel location. In Wei & Levoy, they actually flatten out part of the mesh onto a 2D grid. Once the whole grid has been covered, they know they have enough pieces of the mesh flattened and sample colors for each grid point. They do the flattening to deal with sharp corners in the mesh, such as the ears of the Stanford bunny.

However, the tetrus does not have any very sharp corners. Thus, I chose a much simpler option: instead of flattening neighboring faces over my grid, I just shoot rays from my grid points and see what they hit. Whatever face they hit, I just look up the color for that face at the hit point and use that as the sampled color. I use the global units per pixel value to determine the size of my grid. Each grid square is the size of one pixel. Furthermore, for speed reasons, I do not even cast the ray through all faces of the mesh. I only try intersecting with the immediate adjacent faces of the current face (the current face being, the face that the given pixel location is located on). This does limit the synthesis in some ways: if my grids are too big (meaning the units per pixel value is large), then many rays may actually miss the immediate neighbors. However, I checked for this, and it never happens. It does happen when I increase the units per pixel, but for the values I used for my results, it turned out alright.

Actually, the coarsest tetrus mesh does contain some pretty sharp corners, and there were noticeable discontinuities at those edges. Thus, for my synthesis, I actually start with the tetrus after one run of Catmull-Clark. This smooths out those sharp edges and the results are much better. Below are some screenshots showing the difference. The left one is synthesized with the coarser mesh, and the red circle highlights the problems of the sharp edge. The right one is synthesized with the single subdivision, so it is much smoother along that edge.





Wei & Levoy Implementation Details – Orientation Relaxation Procedure

The one part I did not implement in time was the orientation relaxation procedure. Doing this would have allowed me to synthesis anisotropic textures. I was able to finish about half of the code, but I think there are some problems with it. I did not anticipate the math to be as tricky as it turned out to be. Wei & Levoy did not totally explain how they minimized the deviation energy function. It is not trivial, since the function involves modulo arithmetic. Their energy function is as follows:

$$E(p) = \sum_{q \text{ near } p} (a_{qp} - round[\frac{a_{qp}}{(360/N)}]\frac{360}{N})^2$$

Where *aqp* is the angle between orientation *q* and a possible orientation *p*. The goal is to choose *p* such that *E(p)* is minimized. The approach I took was to break the function into pieces such that within each individual piece, there are no discontinuities caused by the modulo arithmetic. For a texture with *N*-way symmetry, the discontinuities occur at $q + n*(360/N)$, for $n = 0$ to $N-1$, where *q* is the angle that near-by orientations make with the positive X-axis (although if I was doing this for a pixel on the mesh, I would compare to the pixel's texture orientation, not the X-axis). Then between any 2 discontinuity angles, we can derive the energy function (which turns out to be a simple quadratic function), minimize it using it, and not worry about discontinuities.

While I think the approach is correct, the actual implementation was more difficult than I expected. The main problems were with the modulo arithmetic involving angles and wrapping around at 360-degrees. So my energy-minimization code ended up with some edge-case bugs that were tough to track down.

If I had more time, I would have implemented the relaxation procedure much like the synthesis procedure. I would first generate "orientation maps" for each face, which would basically be images with orientation vectors instead of RGB colors, and then accumulate pixel locations for those maps. Then for each pixel location in each level, I would accumulate the orientations of near-by pixels to perform the energy minimization.

Catmull-Clark Subdivision Implementation Details

This part of the project was relatively straightforward. I implemented the standard Catmull-Clark subdivision scheme and subdivide the mesh after the synthesis is done. One thing I do additionally is the subdivision of the UV coordinates. After synthesis, each vertex of a face has a UV coordinate mapping onto its synthesized texture map. When I subdivide the face, I simply take averages of UVs. For example, when I calculate the new face point for Catmull-Clark, I average the UV coordinates of all vertices and set the average as the face point's UV. For edge points, I just take the average UV of its two end points. So after subdivision, there is no longer a 1-to-1 mapping of texture maps to faces. Many faces may share the same 2D texture map, but they of course use different UV coordinates.

After subdividing, the vertices must be displaced according to their height map. To determine the amount that a vertex should be displaced, I look up the vertex's color (or height value) from all incident faces and take the average. Usually, all incident faces use the same texture map, but sometimes this is not the case, so taking the average is necessary.

One slight bug I ran into was calculating the normal to displace along. At first, I basically calculated the average normal of all incident faces and used that to displace the vertex. However, in my implementation, I was calculating these normals during the actual displacement procedure. This means many normals were calculated using *displaced* vertices, which led to some badly slanting normals. The solution was to calculate all displacement normals *before* actually displacing any vertices, and this fixed the problem.

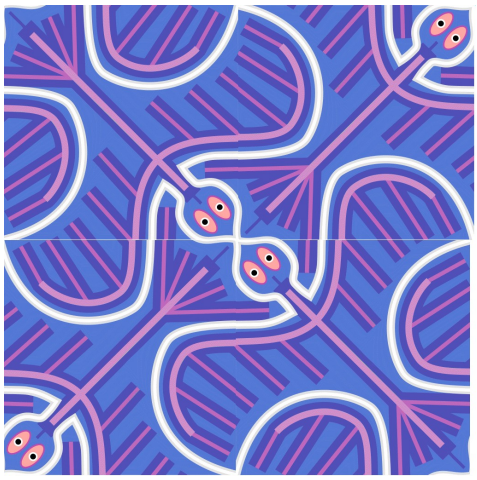
Dealing with STLs – Collapsing triangles to quads

One issue I ran into with Catmull-Clark subdivision is that when SLIDE exports to STL, it has to triangulate everything. Furthermore, it triangulates every quad into 4 triangles. This makes for very rough-looking subdivision surfaces. Thus, I needed to somehow preserve the original quads of the tetrus mesh. There was no way to export them from SLIDE, so I decided to do it procedurally in my program. I basically mark all vertices of the input mesh that have valence 4 (ie. vertices that are shared by 4 triangles), and then I collapse all of its incident triangles into a single quad. This makes the subdivision surface look much better. Of course, this would not really work for arbitrary meshes – so my program in its current state would not generalize very well. A better solution is probably to take quads as input from the beginning, instead of using STLs.

After the collapsing of the triangles, the quads are probably non-planar. To deal with this, I basically treat the quad as a planar rectangle when mapping it to a 2D texture map, but as 4 triangles when I do ray-casting. This leads to slight distortion of the texture map, but it is barely noticeable because the quads are not too non-planar – they're almost flat.

Mapping the Thunderbird Texture

The other texture mapping method I wanted to implement was procedural assignment of UV coordinates. The thunderbird texture provided by Prof. Sequin tiles very nicely in the following fashion:



The goal was to procedurally apply this tiling to the quads of the tetrus. This was pretty straightforward, since one quad's UV assignment completely determines the UV coordinates of all its neighbors. My program performs depth-first traversal of the quad-connectivity graph, assigning UV coordinates according to the tiling pattern as shown above. Although getting the assignment rules was a bit tricky, once I got it this was relatively straightforward. The end result is a very nice tiling of the thunderbird over the tetrus' quads.

For using the thunderbird as a height map, I found it was best to use the Gaussian-blurred height-map provided by Prof. Sequin. This resulted in very nice displaced geometry. Using the non-blurred height map resulted in very "pixelated" geometry due to the sudden changes in height.

Subdivision Surface Viewer

In order for the displaced and subdivided tetrus to look good, it must be subdivided many times. Otherwise, the resolution is too low to really express the height-map's features. My original plan was to output a highly subdivided mesh to SLIDE, but this had a major problem: SLIDE was not really designed to load *huge* SLF files, so this was not practical. I decided to write my own viewing program and output to a binary *.CCS format. CCS files basically have a vertex list and a quad list that references the vertex list. It is binary, so it is much more compact and easy to load than SLF files. The viewing program, *ccsview*, just loads the file and displays it using OpenGL. It provides a first-person-shooter navigation interface, which is not as nice as SLIDE's crystal ball interface, but it works. As a result, I was able to subdivide the tetrus 7 times and get very nice displacement geometry.

References

[1] Wei, L. and Levoy, M. 2001. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '01. ACM Press, New York, NY, 355-360. DOI= http://doi.acm.org/10.1145/383259.383298