A Portable Graphics Program for Rapid Prototyping File Display, Repair, and Interchange

Mike Bailey, Ph.D[*] mjb@sdsc.edu San Diego Supercomputer Center

Dru Clark dru@sdsc.edu Scripps Institution of Oceanography

John Rapp jlrapp@ucsd.edu San Diego Supercomputer Center

University of California San Diego

Abstract

This paper describes a portable interactive graphics program that can be used to preview, repair, and interchange rapid prototype files. Named *glstl*, this program uses the OpenGL graphics API to achieve high 3D performance. Because it uses the GLUT windowing toolkit and the GLUI user interface package, *glstl* has been able to be ported to UNIX, Windows, and Macintosh systems. *glstl* is also freely available on the web.

glstl is able to::

- Import ASCII and binary STL files
- Import color STL files using an ASCII color STL extension
- Display and interactively transform in 3D
- Display as points, lines, reduced lines, shaded surfaces, or shrunk triangles
- Scale and re-orient parts
- Detect and fix model cracks and reversed triangles
- Use color to show difficulty of LOM de-scrapping in particular orientations
- Use color to show difficulty of mold creation in different orientations
- Display parts using 3D ChromaDepth
- Export parts in STL, color STL, or PLY (used by color Z Corp machine)

The paper shows examples of these features and will explain the algorithms behind them. Using this program, our RP operation has become very efficient. We have used RP not just for mechanical design prototyping, but for scientific visualization as well.

[*] Primary contact information:
Mike Bailey
San Diego Supercomputer Center
University of California San Diego
9500 Gilman Drive
La Jolla, CA 92093-0505
W: +1.858.534.5142
Fax: +1.858.534.5152
Email: mjb@sdsc.edu

Introduction

Rapid Prototyping, or Solid Freeform Fabrication, is a mainstream technology in the drive to improve product development productivity. Obtaining a prototype part from a new design is becoming a required prerequisite before entering final production.

One of the strengths of the RP business is the universal acceptance of the STL file format. However, this file format is also one of its key weaknesses. The STL format has been described as a "bucket of triangles". There is no adjacency information between the triangles. There is also no guarantee that the set of triangles form a legal solid or that the triangle vertices are all oriented CCW or that the normals are all outward-facing.. Thus, before RP can be used productively, better ways must be found to deal with the STL file format.

glstl, and a robust data structure

Researchers at the San Diego Supercomputer Center (SDSC) at the University of California San Diego are using RP as a 3D visualization hardcopy device. The Center for Visualization Prototypes (CVP) collaborates with scientists all over the United States (and some outside as well) to enhance scientific understanding by making 3D models. Some examples of this will be shown later.

If STL files that originate from robust solid modeling CAD systems are suspect, then STL files that come from scientific datasets are even more suspicious. To alleviate some of these problems, CVP researchers have developed a program called *glstl* to preview and repair STL files.

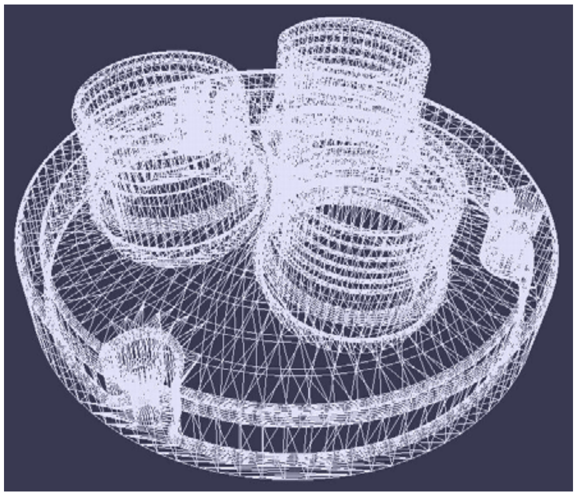
The most important aspect of *glstl* is that it turns the "bucket of triangles" into a robust winged-edge adjacency data structure. It does this by accumulating unique vertices in a balanced B-tree upon reading the STL file. Once the vertices in common have been determined, common triangle edges are searched for, which also gives the adjacency information.

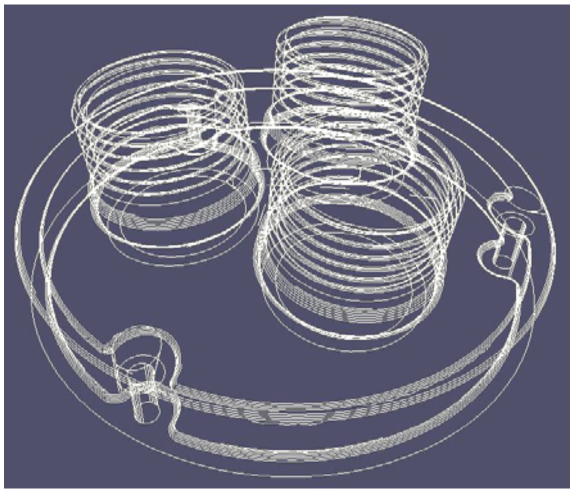
Graphics Portability

glstl is quite portable. We have ported it to UNIX systems, Windows, and the Macintosh. It gets this graphics portability by using the OpenGL graphics API [WOO1999]. This also allows us to take advantage of hardware 3D graphics acceleration.

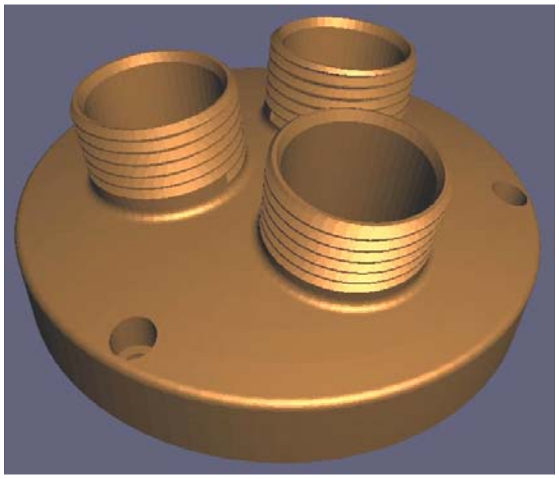
Window system portability is achieved by using the GL Utility Toolkit, or GLUT [GLUT2002]. GLUT collects common window systems operations into a single API with UNIX, Windows, and Macintosh-specific drivers. On top of that, we also use the GL User Interface, or GLUI [GLUI2002], which sits atop GLUT and OpenGL. Thus, *glstl* is a common code that takes advantage of device-independent APIs to achieve portability.

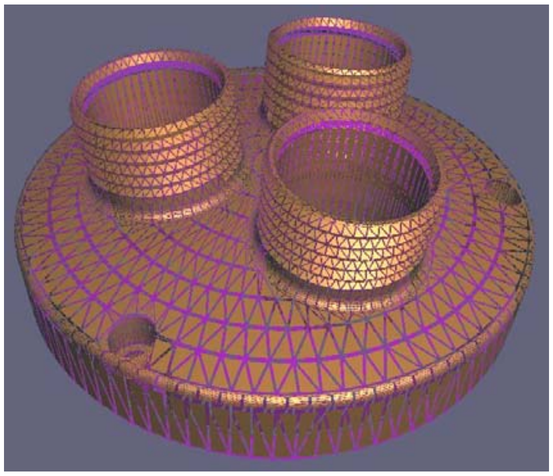
glstl has several graphics display modes, some of which are shown below. The figure on the left below shows a wireframe display using all unique edges. The figure on the right below shows a reduced wireframe, which removes edges between triangles that are nearly coplanar.


Wireframe


Reduced wireframe

Glstl also can display objects using shaded, lighted triangles as shown below on the left. It can also shrink each triangle to show the degree of triangularization, as shown on the right:


Shaded, lighted triangles


Shrunk triangles

Flaw Detection and Fixing

STL file flaws are usually (1) surface normals pointing in the wrong directions, and (2) missing triangles that form cracks in the surface. *glstl* detects and fixes both of these.

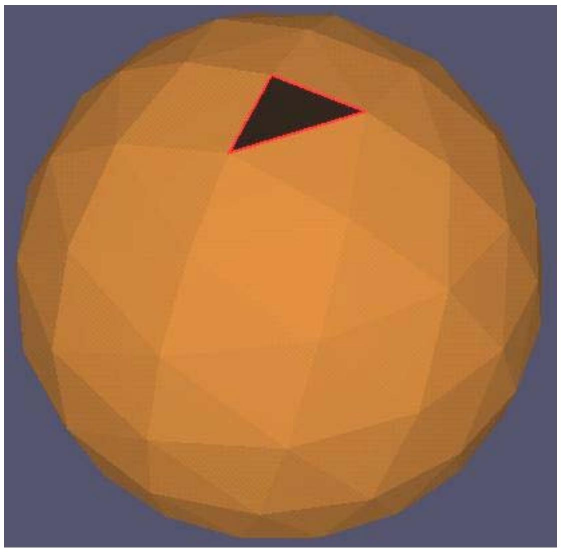
There are basically two stages to fixing the normals. The first stage divides the triangles into what is basically a Binary Space Partition (BSP) structure. The actual division could probably be done better, as it does not subdivide areas with large amounts of triangles, and very often it has spaces with no triangles whatsoever. However, this seems to work for the models we have been using, and it was pretty easy to implement.

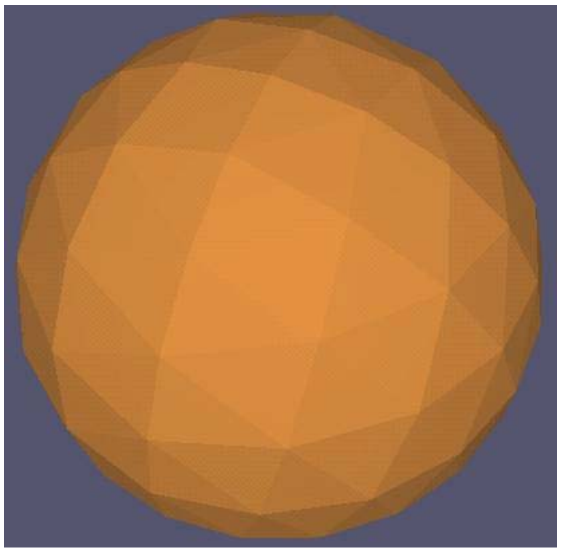
The second stage goes through each triangle and randomly draws a ray out from the center of the triangle. It then intersects that ray with each of the BSP boxes, and if it hits a box, then every triangle in that box is added to a list to be checked for collisions. In order to make sure it doesn't check the same triangle twice, a flag is set in each triangle after it has been tested.

If the ray hits an odd number of faces behind it, and an even number of faces in front of it, then it was originally facing the right way. If not, then the program flips the face because it was facing the wrong way. It is easiest to see this in a picture than anything else. If something strange happens and it finds an even (or odd) number both in front and behind, then it tries again with a randomly perturbed ray.

In terms of speed, this generally runs a few orders of magnitude faster than checking each triangle against every other triangle. In terms of correctness, this usually finds the correct orientation much more often than other faster options where you compare it with its neighboring faces to see which way it should be pointing.

Cracks are found by looking for edges that only have one triangle attached to them. *glstl* puts these edges into a list and graphically identifies these edges with a red line as shown on the left below. It then looks for two edges in this list with a point in common. When it finds these two edges, it completes the third edge between them to create a new triangle, called a patch. If that newly-created edge does not complete another triangle, then it is added to the bounds-a-single-triangle list. This list continues to be processed until it is empty.


Crack detected


Crack patched

Special Tricks -- ChromaDepth

Using OpenGL allows us to take advantage of graphics acceleration hardware to manipulate the STL objects in 3D. However, there are many times where the ability to understand the three dimensionality of the object needs some extra help.

ChromaDepth™ was invented by Richard Steenblik ([STEENBLIK87]) as a way to amplify the common chromostereoscopy phenomenon into a useful display tool. ChromaDepth consists of two pieces: a simple pair of glasses and a display methodology.

The glasses, shown here, contain very thin diffractive optics that have the efficiency of refractive optics. While being very thin and inexpensive[†], they behave like thicker glass prisms. The optics are designed so that red light is bent more than green and green more than blue. The lenses are oriented sideways, so the overall bending effect looks like parts of the scene have been shifted horizontally inwards (ie, towards the center of your nose). The red hues are shifted more than the greens and the greens are shifted more than the blues. Thus, red elements in the 3D scene appear to converge closest to the viewer and the blue elements appear to converge the farthest away.

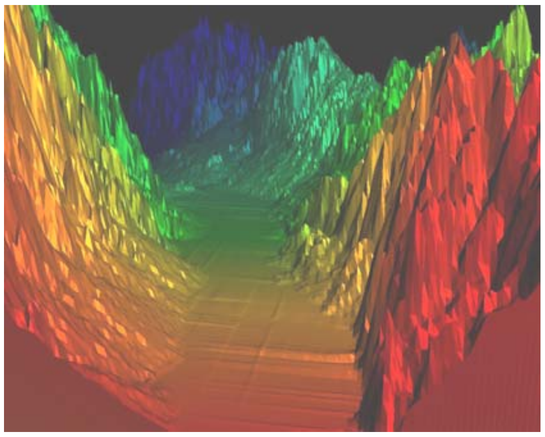
The corresponding display methodology is then quite simple: color code the scene in linear a rainbow spectrum based on depth so that those elements that are close to the eye are displayed as red and those farthest away are displayed as blue.

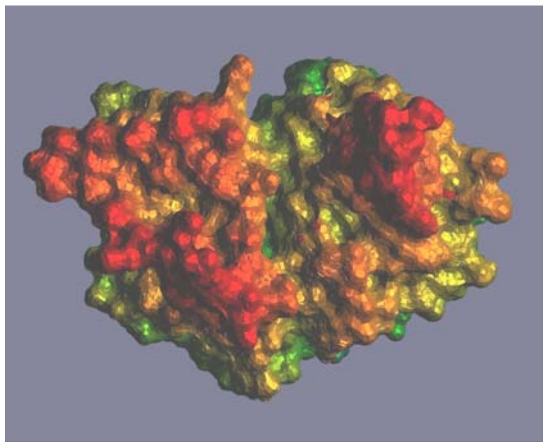
In order to create a ChromaDepth scene, *glstl* creates a 1D texture representing a color ramp from red to green to blue needs to be created. This can be done in a number of ways. We did ours by interpolating in hue-saturation-value (HSV) space.

The next step is to specify how to compute the texture coordinate based on scene depth. [WOO1999] provides a good description of the OpenGL *automatic texture-coordinate generation* capability. This can be used to generate contours on a 3D model in world or eye space, or can be used for dramatic environment mapping effects. It can also be used to automatically generate coloring for ChromaDepth.

The following are ChromaDepth images from two of our scientific models:

[†] Prices for the "opera-style" ChromaDepth glasses shown above range from $.10 to $.90 USD, depending on volume.

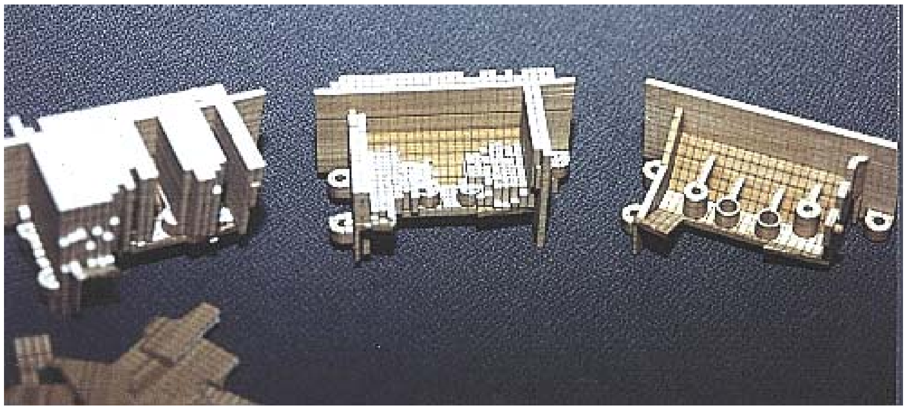

Earthquake fault


Protein kinase

Special Tricks -- Optimal Orientation for De-scrapping

One of the RP machines in the CVP is a Laminated Object Manufacturing (LOM) machine)the other is a Z Corporation Z402). In the LOM process, the 3D object is made from layers of paper. In the LOM process, new paper is spooled into place and laminated to the layers beneath it with a hot roller. A laser cuts the part outlines at this level. The outlines are essentially the contour lines for this height on the 3D object.

After the laser cuts the part outlines, it needs to do something with the portion of this layer that is not part of the object being fabricated. This portion of the layer cannot be made to "fall away" on its own like some of the liquid-based SFF processes, so the LOM process crosshatches it as scrap. Layer-after-layer of this scrap is built up. These scrap "columns" must then be plucked from the resulting part after it is completed. A part in various stages of de-scrapping is shown below. Some of these scrap columns fall away trivially, but some are so difficult to remove that one ends up accidentally ripping some paper that was meant to belong to the part's exterior surface. Clearly having the scrap fall away trivially is a major requirement for a quality part.


Various stages of LOM de-scrapping

The ease with which a scrap column falls away is a function of how weak the connection is between the base of the column and the part exterior. This in turn is proportional to the density of contour lines at the base-part interface. The more contour lines, the more likely the scrap column will just fall off. From our experience, a contour line density of about 100 lines/linear inch will make this scrap column weak enough to fall off easily and preserve the quality of the part exterior.

We needed a graphical method to visually represent the de-scrapping difficulty so we could optimally orient a part. It could not require the application to do much computing between reorientations. With models typically having 100,000-300,000 polygons, this would have made the interactive response too slow to be useful.

Without loss of 3D generality, we will look at this problem in 2D by treating the vertical height as ΔZ and treating the horizontal distance as $\Delta H = \sqrt{\Delta X^2 + \Delta Y^2}$. The contour line density is then related to the slope of the part surface. This is well known to anyone who has used a topographic hiking map. Regions with very dense contour lines represent a steep slope.

Most OpenGL programmers are familiar with the 2D texture coordinates (s,t) and the texture matrix. The typical way to use texture mapping is to specify an (s,t) pair at each polygon vertex and possibly use the texture matrix to rotate or scale (s,t) to become (s',t'), which then index into the texture image during display update. But, a lesser-known feature of OpenGL texturing is that it can use 4D texture coordinates, each specified by an (s,t,r,q) quadruple. These are passed through the 4x4 homogeneous texture matrix as follows:

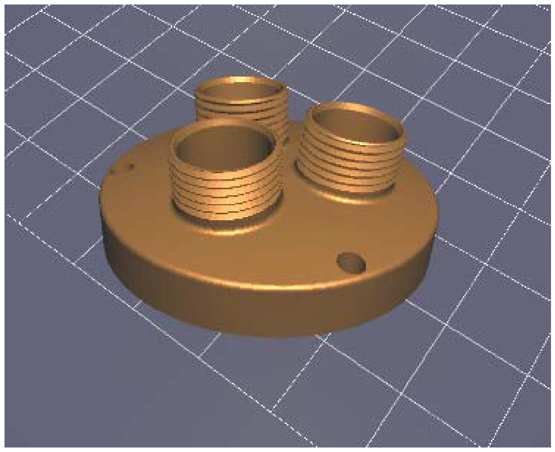
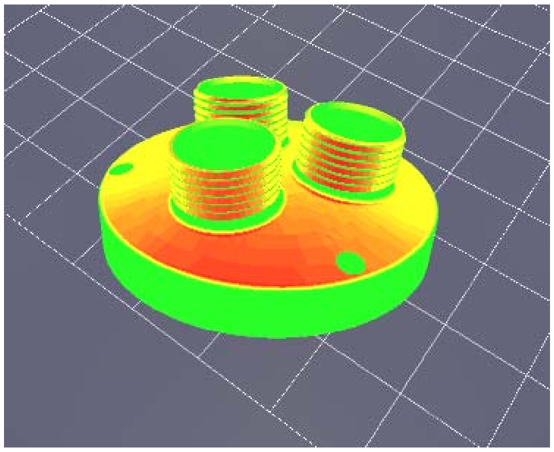
$$\left\{\begin{matrix} s' \\ t' \\ r' \\ q' \end{matrix}\right\} = \begin{bmatrix} Texture \\ Matrix \end{bmatrix} * \left\{\begin{matrix} s \\ t \\ r \\ q \end{matrix}\right\}$$

In the same way that homogeneous vertex coordinates are handled, the elements (s',t',r') are each divided by q' before use. (s't') are then used to index into the texture image.

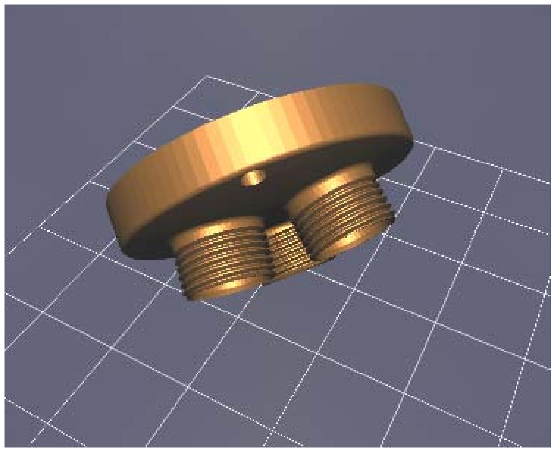
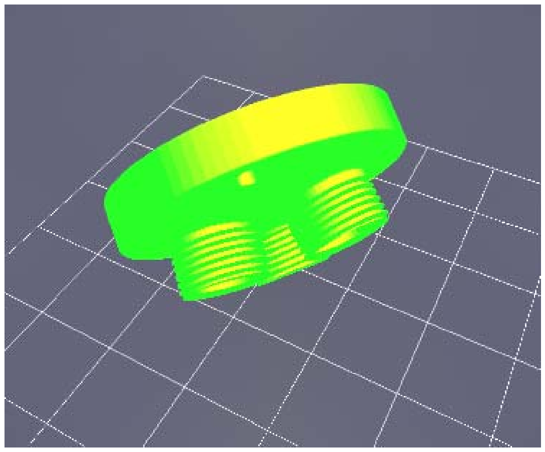
In this method, the (s,t,r,q) quadruple was set to be the unitized surface normal (nx,ny,nz,1.). The texture matrix was two concatenated matrices, one to rotate the normal the same amount as the part has been rotated, and one to turn the rotated Z component of the surface normal into the proper texture coordinates. We established a 256x1 pixel texture where the colors were used to represent the de-scrapping difficulty.

The range of colors in the texture corresponded to a range of Z normal components of -1. to 1. To establish colors, a general definition of a "bad" Z component had to be established. From experience we knew that a CLD of 100 or more (which corresponds to $nz \leq 0.922$) made de-scrapping easy. To these values, we assigned green. We then ramped the hue from green to red in the hue-saturation-value color space as CLD decreased from 100 to 0. This gave us good green-to-yellow-to-red behavior when the contour line density dropped below 100 per inch.

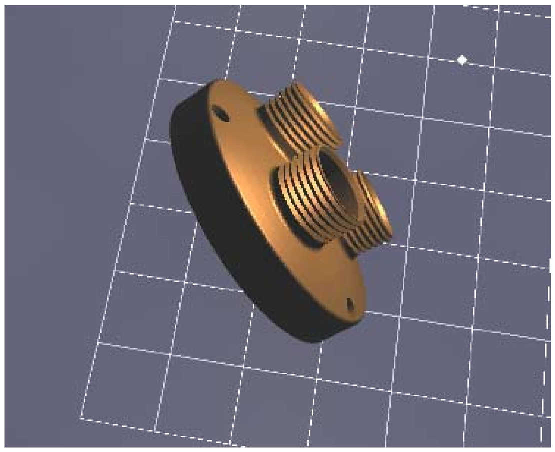
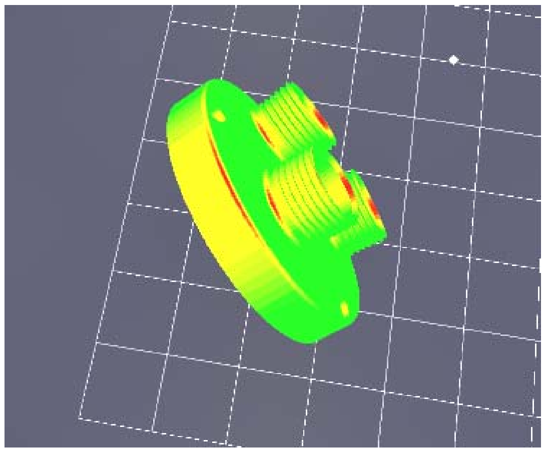
The following figures show how we can visualize de-scrapping difficulty, and how we can re-orient the part to move the difficult area to a non-critical portion of the part.

The difficult de-scrapping area is in a bad place – the threads

A better orientation for a quality finish on the threads

The difficult de-scrapping area has moved to a better place

Special Tricks -- Mold Overlap

While RP is a good technology for one-of manufacturing, it is not good for mass production. When we have needed to consider making a number of the same part, we have turned to molding. To do this, we have needed to determine an optimal orientation and parting-line location.

A good first-order approximation to finding an optimal orientation and optimal parting line is to view the object from above using an orthographic and determining how vertically convex it is. At the optimum, the part is vertically convex at all locations. Seeing how often different portions of the model occupy the same pixels is a rough measure of vertical convexity. The trick is being able to count this pixel overlap on a pixel-by-pixel basis.

Fortunately, the OpenGL graphics API has a useful construct called the *stencil buffer*. The stencil buffer allows the graphics pipeline to perform some simple pixel-by-pixel arithmetic and comparisons. Details of the OpenGL stencil buffer can be found in [WOO1999].

We use the stencil buffer in a new and novel way as follows:

1. Orient the part in its trial orientation such that the eye is looking at the top of the part.

2. Set the viewing projection to orthographic with X and Y window boundaries that exceed the size of the part. Set the near clipping plane to be between the top of the part and the eye. Set the far clipping plane to be at the parting surface.

3. Disable the Z-buffer test.

4. Enable backface culling.

5. Clear the stencil buffer to all 0's.

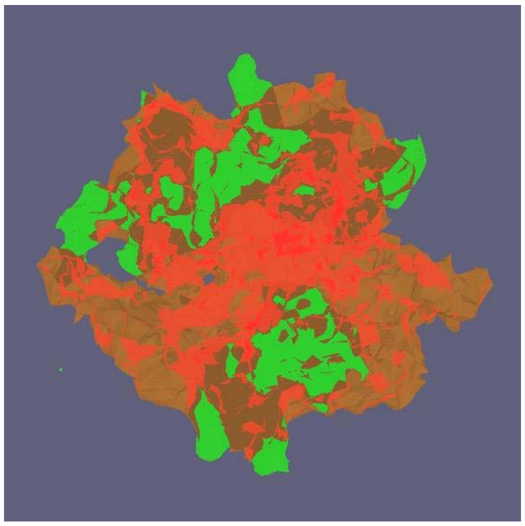
6. Set the stencil function and stencil operator to increment the count as new pixels are attempted to be drawn:

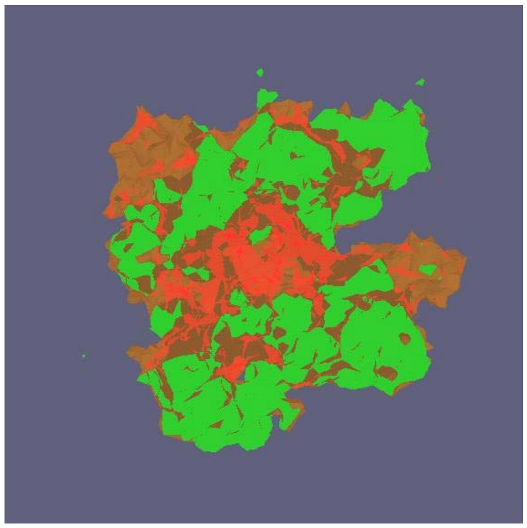
   ```
   glStencilFunc( GL_ALWAYS, 0, 1 );
   glStencilOp( GL_INCR, GL_INCR, GL_INCR );
   ```

7. Draw the part.

When the drawing is completed, the stencil buffer is used to determine the colors in which to draw the part again – green to indicate that only one surface was drawn onto this pixel and shades of red to indicate that multiple surfaces tried to be drawn here.

The part can be reoriented interactively.. The object of the exercise is to maximize the amount of green that shows in the display. Thus, this technique essentially uses the hardware graphics pipeline as a "pixel computer" to quickly discern some information about a particular part in a particular orientation.


One orientation/parting line


A better orientation/parting line

A molded molecule

Color Parts

One of our RP machines is a Z Corporation Z402C which is capable of making parts in color. The Z402C expects the colors to be specified triangle-by-triangle. Unfortunately, the STL file format does not allow any color specification, so the Z402C expects color parts to be specified in a special binary polygon format called PLY. By the time this came about, we had a considerable amount of development tied up in producing STL files from scientific data, and did not want to re-tool all those applications to produce PLY.

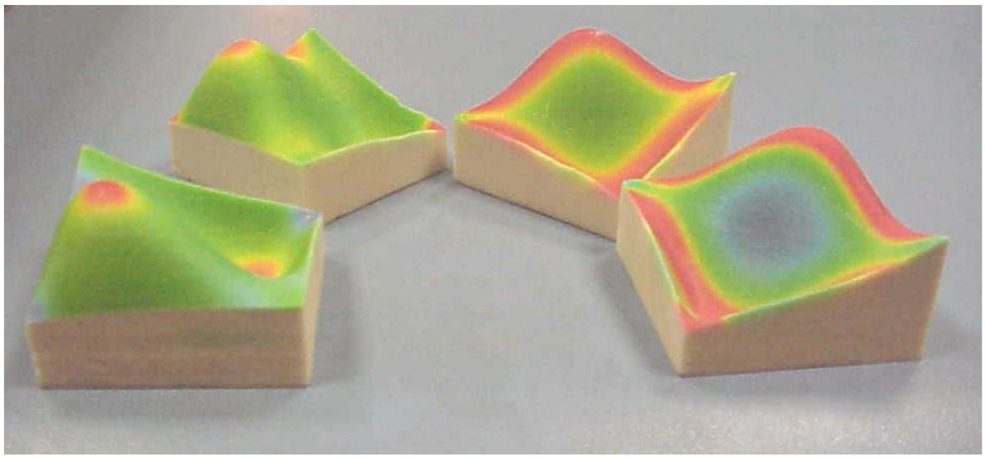
So, instead we added our own extension to STL. In our version of STL, a line of the form:

```
        color r g b
```

Can be included anywhere in the STL file and can be included as many times as needed. When read by *glstl*, it has the effect of coloring all triangles after this line to the color (r,g,b), until the next `color` line is encountered. This allows us to preview color parts. We then added a PLY export option to *glstl* so that these color parts can be sent to the Z402C. The following figures show some of the color parts we have made:

Colored mars globe

Hemoglobin molecule showing electrical charges


Mathematical curvature analysis

Conclusions

glstl is a grass-roots program designed to increase the productivity of those who work with STL files. It allows 3D display, inspection, flaw-detection, and some flaw-fixing. It also contains a number of "tricks" that increase its power to display and analyze the parts.

glstl can be downloaded free from the web at: `http://dvl.sdsc.edu/glstl`

Acknowledgements

This project gratefully acknowledges funding from the National Science Foundation under grant 9809224. We also acknowledge the San Diego Supercomputer Center for its support.

References

[BAILEY1998]
M. J. Bailey and D. Clark, "Encoding 3D Surface Information in a Texture Vector," *Journal of Graphics Tools*, Volume 2, Number 3, August 1998, pp. 27-35.

[BAILEY1999]
M. J. Bailey and D. Clark, "Using OpenGL and ChromaDepth to obtain Inexpensive Single-image Stereovision for Scientific Visualization", *Journal of Graphics Tools*, Volume 3, Number 3, August 1999, pp. 1-9.

[GLUI2002]
`http://www.cs.unc.edu/~rademach/glui`

[GLUT2002]
`http://www.opengl.org/developers/documentation/glut/index.html?GLUT`

[STEENBLIK1987]
Richard Steenblik, "The chromostereoscopic process: a novel single image stereoscopic process, " *Proceedings of SPIE: True 3D Imaging Techniques and Display Technologies*, January 1987.

[WOO1999]
Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner, OpenGL [1.2] Programming Guide, Addison Wesley, 1999.