1 Introduction

In this code sample I will demonstrate my implementation of Object Space algorithms for drawing contour lines and suggestive contour lines on existing 3d meshes, developed for my Master’s thesis in the Graphics group at the Department of Computer Science of the Catholic University of Leuven, Belgium.

The Object Space algorithms implemented here are not the most ‘interesting’ features of my thesis (the real magic happens in the GLSL shaders), but I think they demonstrate my C++ coding skills and style best. All the algorithms in this sample work in Object Space: they perform calculations on the original mesh, and do not perform any image analysis technique on a specific render to extract contour lines: that’s the Image Space approach.
Figure 1: Regular contours in blue, suggestive contours in red. Golf ball model, 600k faces.

2 Theory

2.1 Line Drawings and Contour Lines

In Line Drawings, artists try to convey shape using only lines. Regular contour lines define the outline of the object, and although they form a strong visual cue, they are often not enough for a person to comprehend and recognize the shape which is depicted [Biederman, 1987]. Luckily, our visual system can relax the interpretation of drawn lines, to identify them not only as contours, but as lines where surfaces bend away from the viewer. This is where so called suggestive contours are introduced: they convey additional detail about the shape of the depicted model.

A good example to demonstrate the additional details suggestive contours depict can be found in figure 1. Without the suggestive contour lines, the object cannot be identified as a golf ball.

2.2 Definitions

The definitions of regular contours an suggestive contours are based on those given in the seminal paper for this topic, [Decarlo et al., 2003].

2.2.1 Regular Contours

Regular contours are defined as the set of points where:

\[ \mathbf{n}(\mathbf{p}) \cdot \mathbf{v}(\mathbf{p}) = 0 \]  

with \( \mathbf{n} \) being the unit normal vector in point \( \mathbf{p} \) and \( \mathbf{v} \) being the normalized vector pointing at the camera position \( \mathbf{c} \).

To compute and draw these contour lines on a mesh consisting of faces defined on discrete points in Object Space, there are two possibilities:

- **Contour lines on faces**: We can find zero crossings of \( \mathbf{n} \cdot \mathbf{v} \) on the faces of the mesh. Starting from 2 zero points, we can perform linear or higher (Hermite) interpolation. The more points we find, the smoother that contour segment drawn over the face will be. This algorithm is implemented in the FaceContourDrawer class.
2.2.2 Suggestive Contours

Suggestive Contours are defined using the radial curvature $\kappa_r$ of a point $p$ on a surface $S$. This is the inverse of the radius of the circle which best fits the surface $S$ when viewed in the direction of the camera.

To define Suggestive Contours, we also need the directional derivative $D_w\kappa_r$ of this radial curvature in the direction of $w$, which is the vector $v$ projected onto the radial plane at $p$. This involves some funky trigonometry and math which is thoroughly explained in [Rusinkiewicz, 2004]. For understanding this sample, it’s sufficient to know that we can estimate $D_w\kappa_r$ on any given vertex in the mesh.

Suggestive Contours are defined as points on the surface where:

$$\kappa_r = 0 \text{ and } D_w > 0 \quad (2)$$

The algorithm implemented in the class SuggestiveContourDrawer first tries to find zero points of $\kappa_r$, and then tests the $D_w > 0$ requirement. Using standard linear interpolation, this can result in two, one or no suggestive contour segments on a face. This is summarized on figure 2. Some additional trimming
is needed to retain stable suggestive contours only. This is defined by a fixed-value threshold, as explained in [Decarlo et al., 2004]. In order to be able to use the same threshold for every view level, we multiply the threshold by the so-called feature size of the model.

### 2.2.3 Suggestive Highlights

It’s also interesting to note that by drawing the dual points contained in the $\kappa_r = 0$ loops together with basic black/white toon shading, one can create a real-time Frank Miller-style (Sin City) effect, as demonstrated in figure 3. These lines are called Suggestive Highlights, and are easily constructed with just a sign change in the Suggestive Contour implementation.

### 3 Code structure

#### 3.1 Used libraries

I’m making use of the TriMesh2 library [Rusinkiewicz, 2006]. This is an academic library for reading and manipulating 3d meshes. By using this library, I did not have to re-invent the wheel on the following subjects:

- A performant implementation of vector and matrix operations.
- A parser for meshes in .OBJ and .PLY format.
- An implementation of an OpenGL camera.
- Calculation of basic view-independent per-vertex information like principal curvatures and building connectivity edge maps.

I’ve clearly mentioned in the source code where I’m using existing TriMesh code or functionality. The library file (for x86 systems) and the required headers are included in folders `include` and `lib` in the code sample.

The other libraries I’ve used are:

- **GLUT**: Easy window manager for OpenGL. [glm]
3.2 Application Structure

All methods and classes are documented in the source code. I will summarize the most important elements in the following sections.

3.2.1 Viewer

The main application is implemented in Viewer.cpp. This is an OpenGL viewer which allows the user to display a mesh and manipulate the camera (rotate/zoom) using the mouse. Using the command line, a set of Models can be passed to the viewer.

In order to position the camera accordingly in every frame, the viewer makes use of bounding spheres, which are calculated in update_boundingsphere(). All the transformations which are implicitly applied to the view matrix (by rotating, zooming, panning) are stored using xforms. These are basic classes representing matrix manipulations.

The most important part of the Viewer is contained in the draw()-method. For every model, the specific model transformations (contained in a matrix) are pushed on the OpenGL view matrix stack and the model is drawn. This method is called every time a redraw is needed. The lighting conditions are set once every frame by the setup_lighting method. In this demo, the only choices are fullbright and diffuse, which can be toggled on/off with a key.

The other functions in the Viewer are listener functions to act accordingly on mouse and keyboard events signaled by GLUT.

3.2.2 Model

The Model class represents a 3d mesh which was loaded into the program, and keeps track of all vertex information (view-independent and view-dependent) which was calculated for it. One of its attributes is a const TriMesh*, which is the actual triangular mesh vertex info as read and calculated from the input file. The actual reading and parsing of the model file is handled by the TriMesh2 library.

Every model contains a Drawer Stack, on which you can push and pop drawers (Last In, First Out), which each draw the model or parts of the model in a certain style. The first Drawer which is pushed into the stack gets drawn first. The Drawer Stack only retains pointers to the several defined Drawers, so any configured Drawer can be used for multiple models.

The model also contains pointers to Vertex Buffer Objects (VBO’s), which contain the vertex and normal information. This data is transferred to the GPU memory on creation of the Model object as static draw data: this sample assumes no changes will be made to model geometry. View-dependent vertex info can be calculated using needNdotV(vec camera_position) and needCurvDerivatives(vec camera_position). These functions can be called by Drawers, if they require this kind of information. It is only calculated once.
Figure 4: Drawers Class Hierarchy. (Colored classes are Abstract classes)

for each camera position, so no overhead work is done when two Drawers request the same info.

3.2.3 Drawers

The Drawers are the actual components which push vertices, faces and lines to the OpenGL draw buffer. The abstract class 

\[ \text{Drawer} \]

dictates that they should provide a virtual function 

\[ \text{draw(Model* m, vec camera position: vec)} \]

and provides a visibility flag for switching a drawer on/off. The class hierarchy for Drawers is displayed in figure 4.

Several Drawers are implemented for this sample:

- **BaseDrawer**: This just pushes all vertices with their normals to the OpenGL draw buffer, by using the triangle strips method. This is a compact way of describing long arrays of connected faces. The actual visual effect is determined by the lighting conditions. If no OpenGL lighting is configured, this just renders the model in fullbright color.

- **LineDrawer**: This abstract subclass of \[ \text{Drawer} \] represents Drawers who draw lines. It includes additional attributes to define line color and line width. It also offers buffers to hold the line segments while they are being calculated, so they can be pushed to the draw buffer with one OpenGL call, instead of pushing all the segments on-the-fly, which is notably slower.

The following LineDrawers are implemented:

- **EdgeContourDrawer**: Draws regular contour lines of a given model using Appel's algorithm [Appel, 1967]. In this class, the private method \[ \text{find_edges} \] represents the algorithm.

- **FaceContourDrawer**: Draws regular contour lines of a given model using linear interpolation over faces. In this class, the private method \[ \text{find_facelines} \] finds contour line segments on the faces by using the method \[ \text{construct_faceline} \] for the interpolation.
- **SuggestiveContourDrawer**: Draws suggestive contour lines of a given model. Has some additional attributes:
  The boolean `fading` controls whether or not a fading scheme is used.
  In the fading scheme, points which are closer to the limit imposed by equation (2) will have a higher alpha value. This way more important suggestive contours are drawn more clearly.
  The `sc_threshold` value defines the value at which suggestive contours get trimmed for stability reasons.
  The private functions `find_sc_segments` and `construct_sc_segments` are used to implement the algorithm described in section 2.2.2.

### 3.2.4 Helper functions and classes

The class FPSCounter contains a rudimentary framerate counter. The files `vertex_info` and `mesh_info` contain general-purpose helper methods to compute information about vertices and meshes.

### 4 Compiling

The code sample provided was successfully compiled using GCC 4.2 on an x86-based Linux system running Ubuntu 9.10. Requirements are OpenGL 1.2 and the libraries from section 3.1. These can be easily retrieved by installing the `freeglut3-dev`, `glew-utils` and `mesa-dev` packages.

A makefile is provided in the source folder.

### 5 Usage

Once the program is compiled, it can be started with the command `thesis_crytek`, followed by the relative or full path to the model files. Two example model files have been included in the `models` folder of the sample application.

The controls are defined as:

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMB Drag</td>
<td>Rotate object</td>
</tr>
<tr>
<td>LMB+RMB Drag</td>
<td>Move object</td>
</tr>
<tr>
<td>MMB Scroll</td>
<td>Zoom in/Zoom Out</td>
</tr>
<tr>
<td>LMB Click</td>
<td>Stop rotation</td>
</tr>
<tr>
<td>a</td>
<td>Toggle base mesh drawing</td>
</tr>
<tr>
<td>z</td>
<td>Toggle regular contour drawing</td>
</tr>
<tr>
<td>e</td>
<td>Toggle suggestive contour drawing</td>
</tr>
<tr>
<td>f</td>
<td>Toggle fading</td>
</tr>
<tr>
<td>d</td>
<td>Toggle diffuse rendering</td>
</tr>
<tr>
<td>g</td>
<td>Draw contours in different colors</td>
</tr>
<tr>
<td>h</td>
<td>Draw contours in black</td>
</tr>
<tr>
<td>w</td>
<td>Dump image file of current view</td>
</tr>
</tbody>
</table>

Table 1: Control scheme for Sample application.
Figure 5: Models rendered using this sample application. **Left Top:** Triceratops model (80k faces) using diffuse lighting, all contour lines and fading scheme. **Right Top:** Hippo model (120k faces) using fullbright lighting, all contour lines and suggestive contour lines in red. No fading. **Middle:** Armadillo model (350k faces) using all contour lines, diffuse lighting and fading.

6 Screenshots

Using the Drawers supplied with this sample, one can render models with several applied effects, as demonstrated in figure 5.

7 Contact information

For more info about this sample or to request additional code samples, please contact me at jerbae@gmail.com, or feel free to call me at +32497771327.
References


