Non-Photorealistic Experimentation

Jhon Adams

Danny Coretti

Abstract

Photo-realistic rendering techniques provide an excellent method for integrating stylized rendering into an otherwise dominated field of computer graphics world of trying to replicate the real-world scenes is the prime objective. Many of these techniques emulate the drawings and paintings of artists and provide excellent mock-ups of real world artistic techniques. The trouble is many of these techniques cannot be implemented on commonly available hardware to run in real-time. Through our research and work implementing and tweaking several NPR techniques, we have explored the trade off between precise detail and frame rate in order to create aesthetically pleasing scenes without sacrificing frame rates.

1 Introduction

1.1 Motivation

Much of the research into NPR techniques has focused thus far on creating precise replications of real-world artistic styles. These potentially computationally heavy techniques have a down side in that they are unable to be rendered more than interactive rates, if even that. Through reading several NPR related papers, we decided to try and see what kind of techniques could we implement in real-time while sacrificing the least possible amount of detail. Our aim became to implement and combine several commonly used NPR techniques and tweak them to get the best frame rates, while producing aesthetically pleasing cartoon style images.

1.2 Related Work

In the realm of cartoon and painterly NPR techniques, there have been numerous papers and developments. One commonly used technique is cell shading or toon shading, where the shading of a model is limited to a specific set of colors, resulting in a banded, drawn shading appearance. From there, we have looked into several techniques involving artistic brush stroke methods where a picture is approximated by numerous “strokes”, giving a painted look to a scene [Hertzmann 2004]. There are also several papers involving the creation of accurate strokes along contours and suggestive contours of a 3D model in order to provide a representation of that object as if it were hand drawn [Goodwin et al. 2007; DeCarlo et al. 2004].

Methods such as the Isophote Distance [Goodwin et al. 2007] and stroke based rendering [Hertzmann 2004] provided excellent results on their techniques, but only at interactive rates. Other techniques such as cell shading can be easily done in real time and were used to improve the overall appearance of the resulting scenes.

2 Algorithm/Technique

2.1 Cell Shading

To define the basis of our experiment, we first wanted to implement one form of the cell shading technique. There are multiple versions of this technique, involving everything from the GPU hardware and predefined textures, to the main CPU and procedurally generated textures. Implementations of all of these methods are readily available throughout the web. The general procedure that is constant throughout the variations in method is as follows:

1. Compute the surface normal n of each vertex point on the mesh.

2. Compute the dot product of the normal with the view vector.

$$n \cdot v$$

$$v = l - p$$

l is the light's position and p is the position of the vertex. The result of subtracting the two is the view vector v.

3. The result of step 2 is the intensity of the light at that point. This value is multiplied by the vertex color stored in the mesh data structure. The result is a coordinate on the cell shading texture, which is to be displayed at that point.

Many methods of cell shading involve the computation of one specific color's intensity values procedurally in order to create the cell shading values possible for a color. In our method we implemented a 3D volumetric texture representing a cell shaded model of the RGB color space. Our implementation creates a 3D texture with n samples of color intensity at each point. We found that by using 16 samples in the 0-255 color intensity range that the best looking renderings could be achieved. This however could be changed to allow any arbitrary number of samples.

The computations to map all the texture coordinates are O(n) and even when calculated every frame create a small impact to frame-rates.

2.2 Silhouettes, Contours, and Suggestive Contours

Our next step was to implement silhouette and contour detection in order to outline specific features along the model to enhance the appearance that the models are drawn. The algorithm for detecting basic silhouette edges involves finding edge crossings where on one side of the edge a vertex is facing the camera, and on the other it is away from the camera. In mathematical terms this means that a point is on the silhouette when:

$$n \cdot d = 0$$

$$v = d - p$$

n is the normal of the vertex, and v, the view vector, is the camera position, d, minus the point position p. Therefore a specific point is on the edge of a model relative to the camera when its surface normal is perpendicular to the camera's direction. Since Silhouette edges are the global minima of the dot product presented above, suggestive contours are then the local minima. Using the curvature of the surface along with the derivatives of this curvature at each point we were able to determine local minima. Curvature values were computed at load time by the mesh library we utilized in our program (Trimesh2).

More on how suggestive contours are implemented.

Like the toon-shading, these values only need to be computed if the camera moves relative to the object, thereby cutting down on some of the recomputation time.

2.3 Stroke Thickness

Isophote distance, otherwise known as stroke thickness [Goodwin et al. 2007] provides a method of emulating artistic strokes that define depth and curvature of an object using lines of varying thickness along contours and suggestive contours. One of the requirements for Isophote distance drawing is to compute isophotes across the mesh. An Isophote is defined as a curve of constant light intensity. Thus it is computed exactly as we computed cell shading intensity, only lines are included in a particular Isophote line only if they have a particular intensity. Given an Isophote of intensity r0 and a contour or suggestive contour, the Isophote distance, or stroke thickness is the image space distance from a point p to an Isophote curve. In our implementation we approximate this distance using formulas outlined in Goodwin et al.

Once this is calculated you then draw the lines using a spine that lies between the Isophote and the contour line. This ensures that the line does not go beyond the bounds of the mesh and more accurately represents the model's shape. This spine line is created by extending a vector from a contour point p in the direction of the inward facing surface normal at that point. Strokes are then defined as quads along this spine with a thickness defined by the Isophote distance, clamped to a user defined value.

2.4 Painterly Rendering

The next concept we implemented was the idea of stroke based rendering. In Hertzmann's paper "Stroke Based Rendering", we decided to see if we could integrate and experiment several painterly rendering techniques. The one that we focused on was based upon the idea of drawing strokes in the direction of the gradient of the image [Hertzmann 2002]. The basic algorithm involved looking at the curvature of a point's neighbors and choosing points with the highest change in curvature. Then a line is drawn between the two points with a color that is the average of the neighbors of the points. We experimented heavily with this technique as it produced very poor frame rates. The result from our experimentation was an array of non-depth-tested dots we affectionately called "Gradient Dabs".

Based slightly on the technique described above, the "Stroke Based Rendering" that is actually implemented is a much cruder version. For every point on the mesh, the neighbor vertex farthest away is chosen and a line is drawn between the two. Special attention has been paid so that no points will draw lines to each other more than once, creating a cross-hatched style in some cases.

A shading style is included, although not painterly, that creates "Curvature Lines", connecting two vertices of above average curvature that are above each other. This creates strong lines along the y axis of the mesh around areas with strong curves.

3 Results

Once the toon shading was calculated for a particular lighting setup, it does not need to be recalculated. This means that with toon shading, a mesh will render at the same speed it did without it. Only when the light is constantly changing does the toon shading create an impact of O(n) before the mesh can be drawn.

The volumetric texture being used for the cell shading worked out very well but, as we learned toward the end of the project, means that the cell shading implementation cannot be used under Microsoft Windows or even Cygwin do to OpenGL being frozen at version 1.1.

Calculating the contours of a mesh was also significantly improved. All previous implementations of contours and suggestive contours ran at only around 20 frames per second on the machine used for testing. By comparison, the optimizations that we used to improve these computations resulted in above 100 frames per second.

The curvature lines were by far the most frame-rate friendly non-photorealistic rendering style that we experimented with as it did not affect the frames per second in any significant way. This is followed by the stroke based rendering that requires n*2 vertices to be drawn to properly render. To the techniques advantage, these vertices only need to be calculated once.

The last NPR technique that was experimented with, “Gradient Dabbing”, resulted in sub-interactive frame-rates. This technique only requires drawing n vertices. The large dots that create the dabbing style are expensive to draw. If the size of the dots is even decreased by half, the frame-rate increases ten-fold.

4 Conclusion

4.1 Discussion

We implemented a variety of techniques and were able to combine them to produce real-time frame rates. In the case of painterly rendering, rendering time was severely lowered.

4.2 Known Bugs and Limitations

We had a lot of trouble getting the Isophote distance stroke thickness method to work properly. We currently are computing stroke thickness properly but have problems rendering along a spine on the mesh. In this bug, lines often jut out from the surface of the mesh at regular intervals.

4.3 Work Distribution

This project was completed over 20 days. We each worked in tandem throughout the various features of the project. Jhon completed Toon shading and fixed Danny’s silhouette detection code as we experimented with various methods for their computation. Danny wrote the base rendering engine and integrated the mesh library, as well as worked on painterly rendering and silhouette detection by deciphering papers. Both Danny and Jhon experimented with NPR techniques based on papers and on their own intuition at whim. Only finished styles and code have been included.

4.4 Future Work

We believe that with more time we could properly integrate more NPR techniques into the program and create scenes using multiple objects to render a walkthrough scene. Our current code allows such scenes but we have mainly tested using single models of varying polygon counts.

5. References

[1] Szymon, R., “Estimating Curvatures and Their Derivatives on Triangle Meshes”. Symposium on 3D Data Processing, Visualization, and Transmission. September 2004.

[2] Goodwin, T., Vollick, I., Hertzmann, A., “Isophote Distance: A Shading Approach to Artistic Stroke Thickness”. NPAR 2007.

[3] DeCarlo, D., Finkelstein, A., Szymon R.,“Interactive Rendering of Suggestive Contours with Temporal Coherence”. NPAR June 2004.

[4] Hertzmann, A., “Stroke Based Rendering”. SIGGRAPH 2002.

[5] Rusinkiewicz, S. “Trimesh2 Mesh loading Library.” Mar. 2008 <http://www.cs.princeton.edu/gfx/proj/trimesh2/>

[6] Rusinkiewicz, S. “Suggestive Contours Sample Code." Mar. 2008 http://www.cs.princeton.edu/gfx/proj/sugcon/