Cel-Shaded Graphics
For Awesome Game Worlds

Gary Steelman, University of Texas at Dallas

CS 6366 Computer Graphics

gxs112030@utd.edu | gary.steelman42@gmail.com

Introduction

Modern video games are massively complex creations which require “style” in addition to technical achievement. A video game's style directly influences the the feelings induced in the player about the in-game world. The style is composed of multiple aspects including musical tone, graphical tone and setting, character speech patterns, and graphical style. In this paper I explore the cel-shaded graphical style, a non-photorealistic graphical effect akin to cartoon animation.

The cel-shading style exhibited is induced by discretizing the interpolated diffuse colors for vertices in an OpenGL fragment shader.

The results show that I can achieve the cel-shading style with a single equation based on a “number of shading levels” and a fragment color.

Procedure

In this section I explain in detail the procedure I followed to create a working cel-shader. I first created a Scene object, then populated it with a few models, introduced a virtual camera, then lights, then wrote phong illumination shaders, and finally converted the phong illumination shaders into cel-shaders.

All code written for this project is in C or C++ and utilizes the OpenGL, GLSL, and GLUT libraries.

The code is written to be as independent of OpenGL calls as possible. I wanted experience writing the matrix transformations and lighting calculations myself.

Please see the subsection “Cel-Shaders” for more information pertaining only to the created cel-shaders.

The Scene

(“GluLookAt Code – OpenGL.org”)

(“ZeusCMD - Design and Development Tutorials : GLUT Programming Tutorials - Keyboard Input”)

The Scene object consists of multiple other objects: a set of models, a set of lights, a keyboard controller, and a virtual camera for navigating the scene.

The Scene handles keyboard, mouse, resize, and redisplay callbacks from GLUT. The keyboard allows manipulation of any state variable in the Scene including things like counterclockwise drawing mode, wireframe rendering, enabling or disabling lighting, light color changes, and virtual camera movement.

The scene contains an array of Models which can be cycled through for display and visual manipulation.

The Models

Each model consists of triangles and materials. A material contains ambient, diffuse, specular, and shininess components. A triangle contains three vertices, three vertex normals, and a material index for each vertex. The model data is read from external files when the program runs. The data structure used for models is of my own creation – no glMaterial() calls are made.

Viewing the models is done through use of the virtual camera.

The Camera

(Angel and Shreiner)

The virtual camera contains functionality for exploring the 3D world in the Scene. It can translate, rotate, and automatically center on a specified Model object. The camera handles all modelview and projection matrix transformations externally from OpenGL – no glTranslate() or glRotate() calls are made.

To enhance the realism of the scene, lighting functionality was added.

The Lights

(Shreiner)

The lights can be point or spot lights. Each light contains an ambient, diffuse, and specular component, as well as a position. The data structure for the lights is of my own creation – no glLight() calls are made. Placing lights in the scene does no good without a way

to utilize them. I did this through implementing the Phong Illumination model.

Phong Illumination Shaders

(Angel and Shreiner)

(Shreiner)

I created a vertex and a fragment shader which implement the Phong Illumination model for a single light source. The color calculations take place in the vertex shader and are then interpolated and passed to the fragment shader. The final result is then rendered. One final step was necessary to render a cel-shaded model. I converted my Phong Illumination shaders into cel-shaders.

Cel-Shaders

("Toon-shader-version-ii[1]")

The cel-shaders are relatively simple and consist of a single vertex and a single fragment shader. The following sections discuss the mathematics behind the shaders. For code, please see the source code.

Using only the vertex color, vertex intensity, and a number of shading levels, I achieve the cel-shading sytle.

The Vertex Shader

The vertex shader calculates two values: the vertex color and the vertex "intensity".

The vertex color is computed as:

$$\vec{c} = \vec{diffuse}_{light} * \vec{diffuse}_{material}$$

which is the pairwise component multiplication of the light's diffuse color and the vertex's material diffuse color. In most demonstration implementations of cel-shaders only the material diffuse color is used. When multiplying the material diffuse with the light diffuse, however, I achieve the ability to alter the output color of the model based on the lighting in the environment. This is desirable because it gives more artistic flexibility.

The vertex intensity represents how directly a light is shining on the vertex. If a light is shining directly on the vertex, the intensity will be near to 1. If no light is shining directly on the vertex, the intensity will be near to 0. The intensity is calculated as:

$$intensity = \vec{L} \cdot \vec{N}$$

where L is the vector pointing to the light from the vertex:

$$\vec{L} = \text{normalize}\left(pos_{light} - pos_{vertex} \right)$$

and N is the vector's normal, normalized:

$$\vec{N} = \text{normalize}\left(\vec{n_{vertex}} \right)$$

Note that all three values, pos_{light}, pos_{vertex}, and n_{vertex} have been transformed into eye coordinates *prior* to these calculations.

I observe the following property about the calculated intensity:

$$intensity \in [0,1]$$

which is integral to the fomula for discretization of the intensity necessary to achieve the shading levels effect.

The new color vector, *c*, and the *intensity* are then passed to the fragment shader for interpolation and discretization.

The Fragment Shader

The fragment shader receives the *intensity* and *c* values from the vertex shader and makes a single call to the toonify() function.

The toonify() function discretizes *intensity* to a value in [0,1] based on *M*, the number of desired shading levels. The number of shading levels represents how many bands of color to show for the output on the model. If this number is 2, then only two varying levels of color will be shown on the model. The number of levels is specified at the application level.

The discretized intensity value gets multiplied against the the red, green, and blue components of the color sent from the vertex shader. The alpha transparency value is untouched to allow more artistic freedom with creation of transparent objects.

The discretization of the vertex's intensity value is:

$$intensity_{discrete} = \frac{\text{ceil}\left(intensity * M \right)}{M}$$

where ceil(x) rounds x up to the nearest integer.

Using the ceil() function here maintains the brightness of the object. Many implementations use round() or floor(), but in my experience this produces too dark of a model.

And the final output color for a fragment is:

$$\vec{c}_{final} = \begin{pmatrix} \vec{c}_{\text{red}} * intensity_{discrete} \\ \vec{c}_{\text{blue}} * intensity_{discrete} \\ \vec{c}_{\text{green}} * intensity_{discrete} \\ \vec{c}_{\text{alpha}} \end{pmatrix}$$

Challenges and Road-Blocks

There were multiple challenges overcome and multiple road-blocks encountered during development.

The first and most significant challenge was development of a framework that rendered models and allowed me to change the view see if lighting was working properly. This required significant design decisions and multiple rewrites of functions along the way.

The next challenge was altering the framework from using basic GL commands to using shaders. This required understanding how the application level interacts with the shader level and understanding the rendering pipeline more in depth than I did for the first version of the framework.

Then transforming the Phong shaders into cel-shaders took a trial-and-error approach:

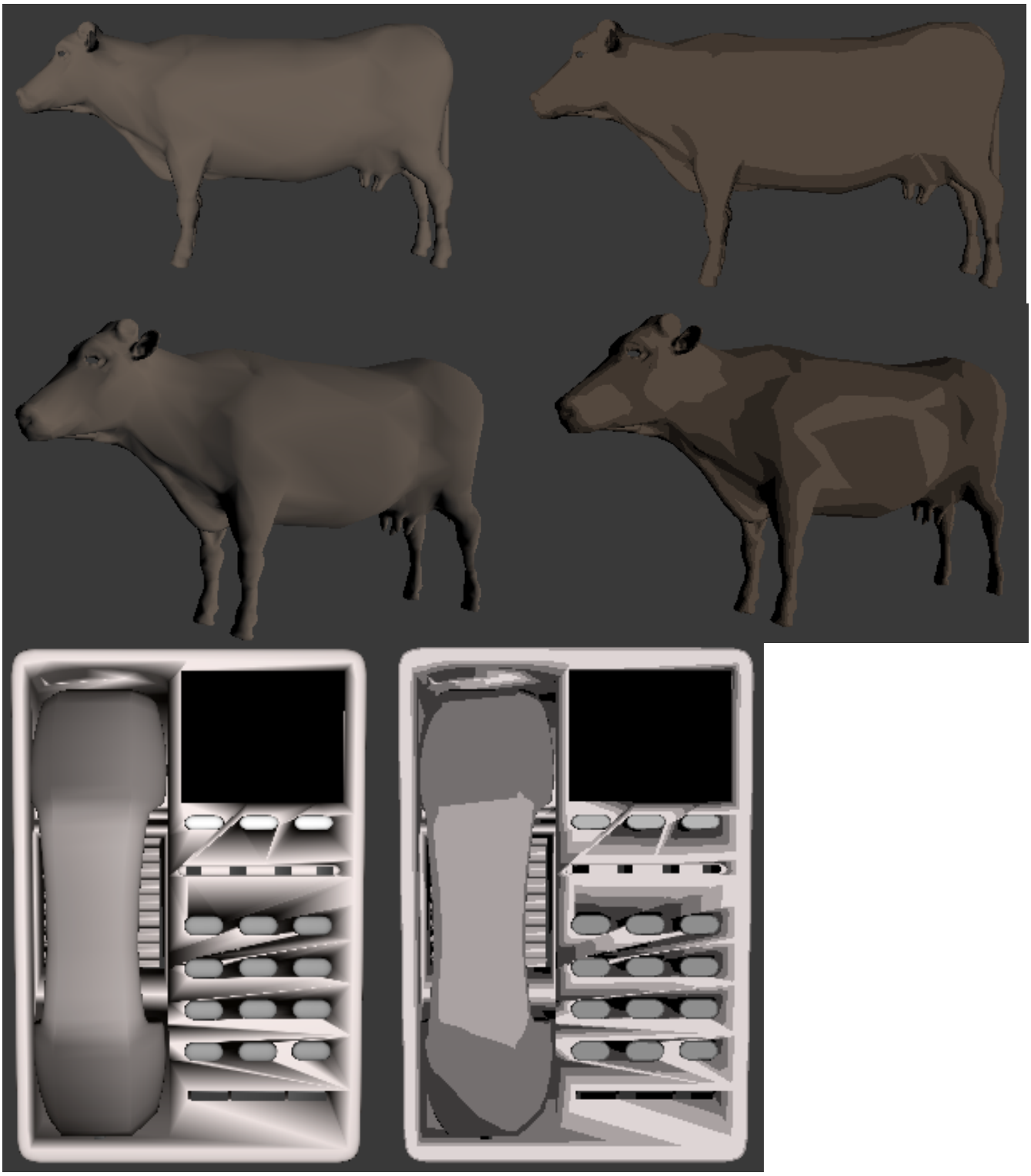
The first implementation of cel-shaders I had calculated the discretized intensity of a vertex and the color of a vertex in the vertex shader. These values were passed to the fragment shader (and interpolated along the way) and then displayed. The result was not the expected, smooth result. In fact it was quite ugly and not correct. This required me to realize that the values I computed in the vertex shader were interpolated before they were sent to the fragment shader. This interpolation was causing undesirable results. I realized what I wanted was for the colors and the *non*-discretized intensity to be interpolated. So I moved the discretization calculation to the fragment shader and the output produced was much more aesthetically pleasing.

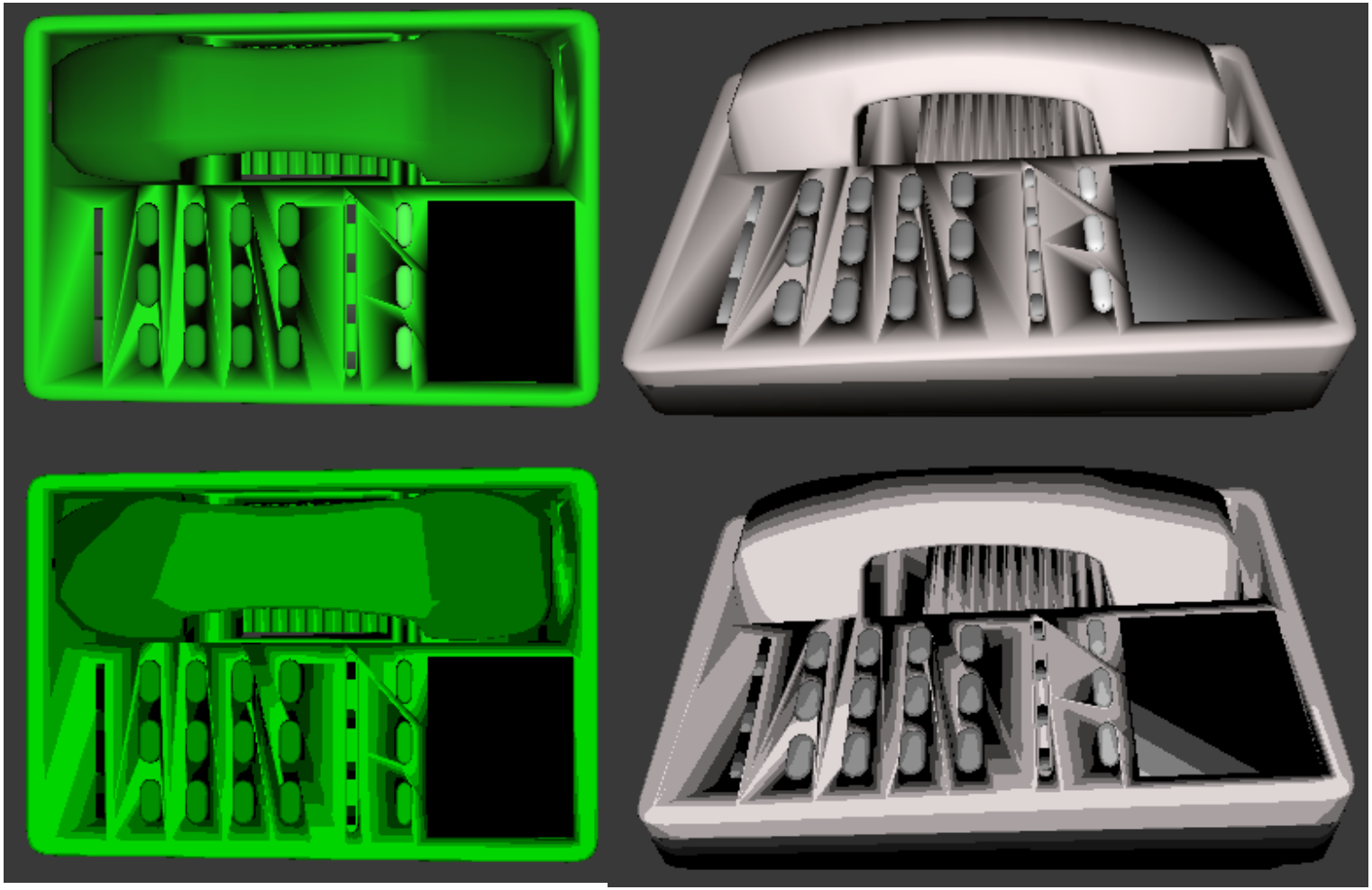
The implementation I had now was good, except that the discretization was a constant 4 levels and used an if/then/else block to calculate the discretized intensity. Something like “if (intensity > 0.95) return 0.95;” I wanted to experiment with the number of shading levels without having to rewrite my shader each time, so I created a mathematical function that discretizes the intensity for me, as presented in The Fragment Shader section of The Cel-Shaders.

The visual result of overcoming these challenges is presented in the next section and is quite beautiful.

Results

The results shown below are with M=4 levels of shading. So the output shows 4 different bands of color for the models. On the left is the smooth-shaded model produced from the Phong Illlumination model and on the right is the cel-shaded model.





Analysis and Conclusion

The results speak for themselves: a resounding success on all fronts. The cel-shading style was achieved and produced accurately.

All proposed goals were met. A vertex and fragment shader were created to achieve the style.

The equation derived in “The Fragment Shader” was key to having a customizable, easy to use, working cel-shader and this equation can be used in future projects.

Bibliography

Angel, Edward, and Dave Shreiner. *Interactive Computer Graphics: A Top-Down Approach With Shader-Based OpenGL*. 6th ed. Addison-Wesley, 2012. Print.

“GluLookAt Code - OpenGL.org.” Web. 11 May 2012.

Shreiner, Dave. *OpenGL Programming Guide*. 7th ed. Add, 2010. Print.

“Toon-shader-version-ii[1].” n. pag. Print.

“ZeusCMD - Design and Development Tutorials : GLUT Programming Tutorials - Keyboard Input.” Web. 11 May 2012.