Chapter 12

# Project: Cel-shading & convolution filtering

## 12.1 Introduction, overview & goals

For the CG project, we were inspired mostly by the power of shaders in an OpenGL environment. Both cel-shading and convolution filtering are specific examples of how shading can be used in an efficient and detailed way, to influence the look of a scene.

The outline, as set out by us, basically is the study of the effects of cel-shading and convolution filtering (specifically edge detection) on objects in a scene, where the former mostly affects the general lighting, while the latter impacts only the textured wrapped around the object.

With this report, we hope to be able to clearly state the principles, progress and results that have defined our approach.

## 12.2 The project: principles, implementation & results

The project can basically be divided into 2 main parts; convolution filtering and cel-shading. Both work very differently on their own, but contribute to the final result. In the next few sections a more detailed discussion is given.
*Note:* In the fragment shader, 3 booleans are used, to give the user easy control over which effects are applied. Basically, besides tweaking various parameters of the shader itself, one can select whether or not to use a texture, to apply cel-shading, or to do edge detection.

### 12.2.1 Convolution filtering

The convolution filtering only has an impact on the texture, wrapped around the teapot, and not on the general look and lighting of it. However, a near infinite range of effects can be achieved using this technique, which makes it worth implementing it. The main effect, used in this project, is edge detection and enhancement, since this contributes to the 'cartoonish' look of the scene. Various other manipulations, such as sharpening, blurring, feature detection,... are also easily achieved.

#### 12.2.1.1 Principles

The concept of convolution filtering is very general, and applicable in many engineering areas. Its roots lie in the domain of signal processing, where it can be shown that for any linear time-invariant system, the output (y[k]) triggered by a certain input (x[k]) is given by the convolution with the impulse response (h[k]). The general convolution operation between two operands looks like this:

$$y[k] = h[k] * x[k] = \sum_{i=-\infty}^{\infty} h[i]x[k-i] = \sum_{i=-\infty}^{\infty} h[k-i]x[i]$$

If we consider both the input signal and the 'filtering' response function 2-dimensional, convolutions can be applied on digital images, based on the 2D-coordinates of the pixels. The above formula can then be simplified to a feasible calculation, using a filter of width M and height N, as follows:

$$y[r,c] = \frac{\sum_{i=0}^{M}\sum_{i=0}^{N} h[j,i]x[r-j,c-i]}{\sum_{i=0}^{M}\sum_{i=0}^{N} h[i,j]}$$

The normalization factor in the denominator makes sure the resulting values remain contained to the original interval (in this case [0-255]).
It is now possible to apply any classic and/or exotic filter - e.g. low-pass, high-pass, band-pass,... - to an image, resulting in modifications according to the meaning of frequency in the visual domain. For example, one could make the following convolution filter, represented by a 3x3 matrix (to keep the calculations minimal, and localize the effect):

$$M_{sharpening} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The matrix above represents a typical edge-enhancement or sharpening filter. This can be seen intuitively by noticing that the convolution here implies the computation of the differences between the considered pixel and some surrounding pixels, and averaging out the result. More variable areas in the pictures will thus be emphasized.

The matrices for some other effects can be seem below, and are also pretty intuitive. An embossing filter is basically a non-symmetric sharpening filter, enhancing the edges only in a certain direction, thus creating a 3D-effect. The Gaussian blur, or low-pass filter smooths the values of the pixels in every direction. The absence of minus signs implies the averaging combination of various pixels, as opposed to the differentiating combination in the other 2 filters.

$$M_{embossing} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} M_{blurring} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

This is just a short overview of a very extensive topic, and many other effects are possible. For the scope of this project, it is sufficient to know the above. In the next section, the specific implementation, and various results, are highlighted.

#### 12.2.1.2 Project implementation

The convolution filter in the project is completely implemented in the fragment shader, and only works on the texture, if there is one present. This gives a very precise, and straightforward result. The code is broken down below, with additional explanation. The full shader file can be found in the project code appendix.

First, the edge detection section of the shader checks whether the texture is present. Obviously, without a texture, there can be no edge detection. To provide a base (diffuse) color, and make sure it's combined with the texture in case there is one, an additional multiplication is required (ANDing has a better effect on the result than ORing).

```
//Initialization//
 if(textureOn == false)
 {
  edgeDetection = false;
 }

//Edge detection shading section//
 vec4 sum = vec4(0.0);
 if(textureOn)
 {
  diffuse *= texture2D(shader, gl_TexCoord[0].st/2);
 }
```

The first part of the actual convolution filtering looks as follows. The calculated offset is used to position the convolution filter in the texture, so it can do 1 operation per pixel, and include the neighbouring pixels. The matrix of offsets determines the participating fragments in each calculations. The kernel is the core of the filter, implementing the edge enhancing effect, already discussed earlier. The matrix

used in the program is based on an amplification factor, called ampFactor, which determines the rate of filtering. The general structure is:

$$M = \begin{bmatrix} 0 & 1 * ampFactor & 0 \\ 1 * ampFactor & -4 * ampFactor & 1 * ampFactor \\ 0 & 1 * ampFactor & 0 \end{bmatrix}$$

The code looks like:

```
if(edgeDetection)
{
  float step_w = 1.0/width;
  float step_h = 1.0/height;

  offset[0] = vec2( -step_w, -step_h );
  offset[1] = vec2( 0.0, -step_h );
  offset[2] = vec2( step_w, -step_h );
  offset[3] = vec2(-step_w, 0.0);
  offset[4] = vec2(0.0, 0.0);
  offset[5] = vec2(step_w, 0.0);
  offset[6] = vec2(-step_w, step_h);
  offset[7] = vec2(0.0, step_h);
  offset[8] = vec2(step_w, step_h);

 float ampFactor = 4;
  kernel[0] = 0.0;
  kernel[1] = 1.0*ampFactor;
  kernel[2] = 0.0;
  kernel[3] = 1.0*ampFactor;
  kernel[4] = -4.0*ampFactor;
  kernel[5] = 1.0*ampFactor;
  kernel[6] = 0.0;
  kernel[7] = 1.0*ampFactor;
  kernel[8] = 0.0;
```

Finally, the calculated components need to be applied to the source image to generate the filtered result. This is done per fragment, by passing over all 9 elements involved, offsetting the texture lookup value accordingly, and multiplying this value with the kernel.

```
 int i;
  for( i=0; i<9; i++ )
  {
   vec4 offSettedValues = texture2D(shader,(gl_TexCoord[0].st/2) + ...
       offset[i] );
   sum+=offSettedValues*kernel[i];
  }
}
```

#### 12.2.1.3 Results

Doing all this renders some nice results. The picture below (12.1), illustrates the edge enhancing effect for various values of the ampFactor.

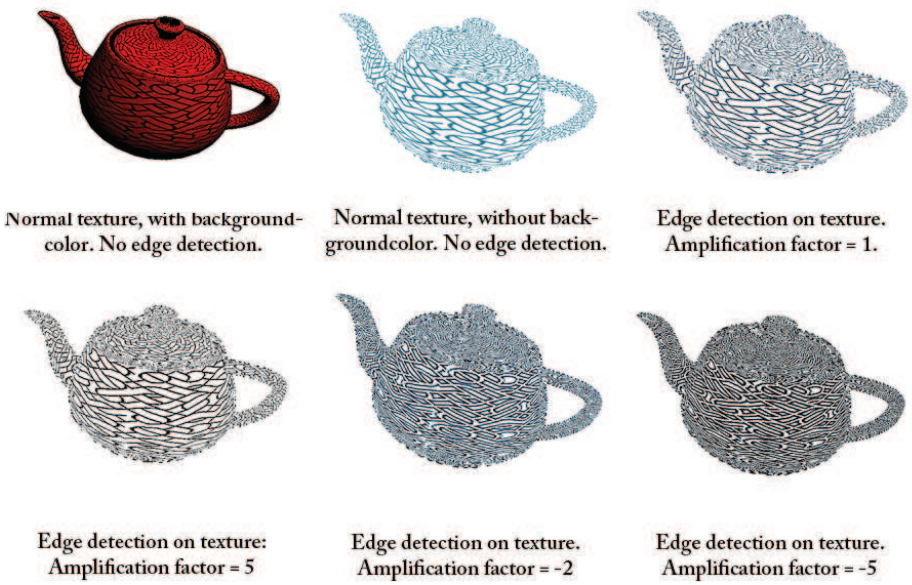



**Figure 12.1:** Various effects of edge enhancing filtering.

An example of another possible effect, namely embossing, is given in the figure below.

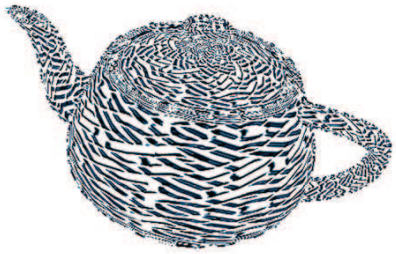


**Figure 12.2:** Embossing filtering.

### 12.2.2 Cel-shading and Inking

When we normally create synthetic imagery we normally strive to get as realistic results as possible, but in some cases that may not be the ideal. This is called non-photorealistic rendering. Comic book and cartoon artists for instance often strive towards a certain style in which shapes and edges are marked a thick black line and objects are shaded non-smoothly. An example of this can be seen in figure 12.3. Cel shading, or toon shading, is a technique used to mimic this style in computer graphics.

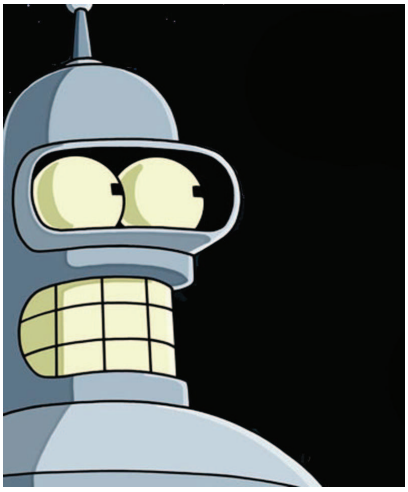


**Figure 12.3:** A typical cartoon-style rendition with sharp curves and border marked by a think black line and non-smooth shading.

#### 12.2.2.1 Principles

In order to achieve the blockly kind of shading that comic book artists often use, a regular Phong shader is used and the weight of the diffuse, ambient and specular color contribution is rounded of so that the can only have a few values, usually 3 or 4. An illustration of this concept can be seen in figure 12.4.

Often only the diffuse color contribution is used in the shading, because it is often enough to achieve the desired effect.

There are multiple approaches for making the black outline, which is known as inking. The types of inking that we are interested in are the following: Edges that are not shared by multiple polygon (borders), edges that are shared by a front-facing and back-facing polygon (silhouettes), and edges that are shared by polygons have sharp angle between them (hard edges) **?**.

One technique is to have the fragment shader draw black fragments whenever the angle the between viewing direction vector, and the surface normal get above a certain amount. That amount should be somewhere slightly less than 90°, depending on thick the line should be. This will however only provide inking for silhouettes,

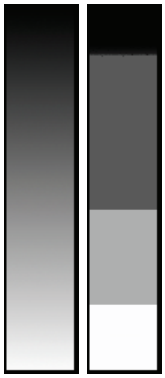


**Figure 12.4:** The figure illustrates the difference between smooth and cel shading shading, with smooth shading on the left and cel shading on the right.

and the thickness of the line will depend on how sharply the surface curves. For instance the neck of a teapot will have a thin outline, because there are only few fragments near the edge that an eye-normal angle within the threshold, however the body of the teapot has many such fragment and will therefore get a thick outline.

Another technique is to draw the model twice; the first normally and them as a black wire-frame with the front faces culled. Most of the wire-frame will be covered by the original rendering, but the border edges of the wire-frame be partially in front of the original rendering. These edges will appear as borders and silhouettes. An artifact of using this technique is that it can be possible to see the wire-frame through the model if it is somehow open.

#### 12.2.2.2 Project implementation & results

To simplify shading we have chosen only to use the diffuse color contribution, which is sufficient for the desired effect. Transform regular Phong shading to cel shading actually quite trivial. We simple calculate `Kd` as we normally would, and then rewrite to the middle value of whatever interval it is within. The intervals we have chosen are: 0 to 0.3, 0.3 to 0.4, 0.4 to 0.6 and 0.6 to 1. The code for these calculation can be seen in listing 12.2.2.2.

```
float Kd = max(dot(Normal, Light), 0.0);
...
if (Kd > 0.60)
  Kd = .8;
else if (Kd > 0.4)
  Kd = 0.5;
else if (Kd > 0.3)
  Kd = 0.35;
else
  Kd = 0.15;
```

The result of this can be seen in figure 12.5

**Figure 12.5:** A cel shaded teapot.

The code for inking using angle between the vector to the eye and the surface normal can be seen in listing 12.2.2.2. First the angle is calculated, then we check if its beyond the threshold and if it is, we reduce `Kd` to zero, so that the color becomes black.

```
float normalEyeAngle = acos(dot(Normal, Eye));

if (normalEyeAngle > 1.35)
{
  Kd = 0.0;
}
else
{
  //filter Kd normally
}
```

The result of this can be seen in figure 12.6.

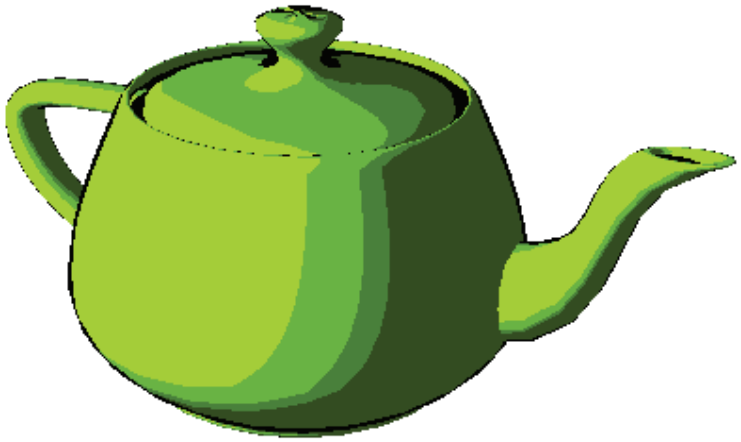
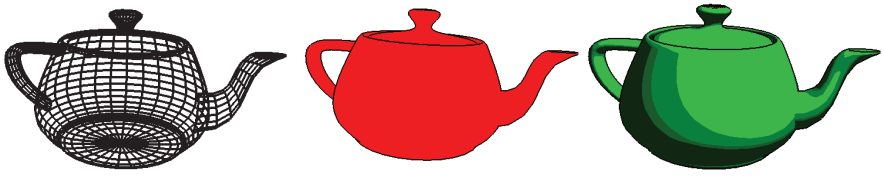


**Figure 12.6:** Inking using angle between the eye vector and the surface normal.



**(a)** Wireframe **(b)** Ink **(c)** Inked and shaded

**Figure 12.7:** Inking and shaded.

This type of inking made the thickness of the outline vary a lot, which was not the goal, we instead tried inking using the front face culled wire-frame. The code for this can be seen in figure 12.2.2.2. First we enable culling and set it to cull front-faces as well as set up the depth function. Then we change the drawing mode so that the model is drawn as a black wire-frame. Finally we the the wire-frame and reset polygon mode and culling.

```
//Draw wire-frame for inking
glEnable(GL_CULL_FACE);
glDepthFunc(GL_LEQUAL);
glCullFace(GL_FRONT);
glPolygonMode(GL_BACK, GL_LINE);
glLineWidth(4.0f);
const float wireColor[] = {0.0f, 0.0f, 0.0f, 1.0f};
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, wireColor);

//glColor3f(0.0f, 0.0f, 0.0f);

drawCurrentModel();

//Reset culling and polygon mode
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glDisable(GL_CULL_FACE);
```

The inked teapot can be seen in figure 12.7.

### 12.2.3 Total implementation

The combined implementation of both parts results in some cool-looking objects.

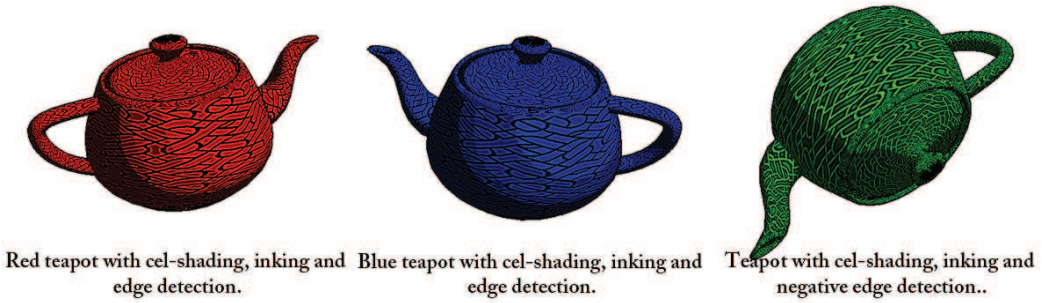



**Figure 12.8:** Fully rendered sample objects.

. . . In the fragment shader, there is some additional code to allow the user to turn on/off and tweak certain aspects of the code in it. However, the user can switch between certain scenarios in the executable as well, by using the 'R' key (display raster), 'I' key (display inking), and 'M' key (change model). Due to time limitations, there are no executable controls over the convolution filtering, but as said, this can be modified in the shader file easily.

## 12.3 Team work: experiences & participation

Although most of the project work was done together, each member had its own specialization. Jens-Kristian mostly worked on the cel-shading and inking part, while Naim was responsible for the convolution filtering sections.

The project, in the end, might have turned out rather large, mostly due to our expanding interest while making it. We can however say that with this project, we've learned a lot about the discussed domains, and computer graphics in general.

## 12.4 Conclusions

The CG project was a demanding one, but interesting at the same time. The unbounded topic choice made it possible for us to include two separate areas into one whole, with nice results as a consequence.