2.10

Hardware Accelerated Charcoal Rendering

Markus Nuebel

Introduction

With the arrival of programmable pipelines, the variety of visual effects used in modern computer games has increased dramatically. Although it has taken some time for shader hardware to become widespread in the consumer market, we have now reached a point where some games no longer support older hardware, and rely more and more on shader technology. While developers have ventured deeply into improving photorealistic rendering using shaders, many have also started to explore non-photorealistic rendering techniques and improvements. In a market where visual brilliance is taken as being "matter of fact," non-photorealistic rendering is increasingly interesting, as it allows the creation of unique effects that have not been available before.

In this article, we describe a non-photorealistic rendering technique based on [Majumder02]. It produces results similar to classic charcoal drawings, i.e., those created by artists drawing on paper using charcoal sticks. We will discuss how to simulate smooth tonal variation and grainy strokes, which are typical for charcoal paintings, as well as the closure effect used to achieve soft silhouettes.

Overview

Below is a short overview of the steps involved in hardware accelerated charcoal rendering. Each step will then be discussed in greater detail.

1. **Grayscale Diffuse Lighting:** Start by calculating the brightness of each vertex according to the Lambert diffuse lighting model.
2. **Illumination Contrast Enhancement:** A modified brightness value is calculated by applying a contrast enhancement operator to the value calculated in step 1.
3. **Contrast-Enhanced Noise Texture Generation:** The same contrast enhancement operator used in step 2 is applied to a grayscale noise texture. This produces a contrast-enhanced texture that is used in the following steps.
4. **Model Texturing:** For each pixel, the contrast-enhanced brightness is used as one component of a 2D texture coordinate. The other component is generated randomly, and the model is textured using the contrast enhanced noise texture created in step 3.

5. **Color Blending:** The color from the contrast enhanced noise texture is modulated by the brightness from step 2.
6. **Background Paper Blending:** 2D clip space coordinates are calculated and look up a color from a background paper texture, and are then blended with the result from the previous step. The technique is implemented using exactly one vertex (vs_1_1) and one pixel shader (ps_1_4).

Details Description

Grayscale Diffuse Lighting

In the vertex shader, we transform the surface normal to world-space, and then calculate the Lambert diffuse lighting term for use in shading.

Optionally we can oversaturate the intensity to more emphasis the closure effect, described later on.

```
// Calculate light intensity
float3 vecNormal = mul(Input.Normal, matWorld);
float fIntensity = saturate(dot(vecLight1, vecNormal));
float fLambertIntensity = saturate(fAmbientIntensity + fIntensity);
// Optionally oversaturate, to increase the closure effect
fLambertIntensity = saturate(fLambertIntensity * 1.5);
```

Figure 2.10.1 shows the model of a chameleon, lit with the above shader. (Note that the bumpmap and light vectors are not yet used in the pixel shader, only LambertIntensity is used for this figure.)

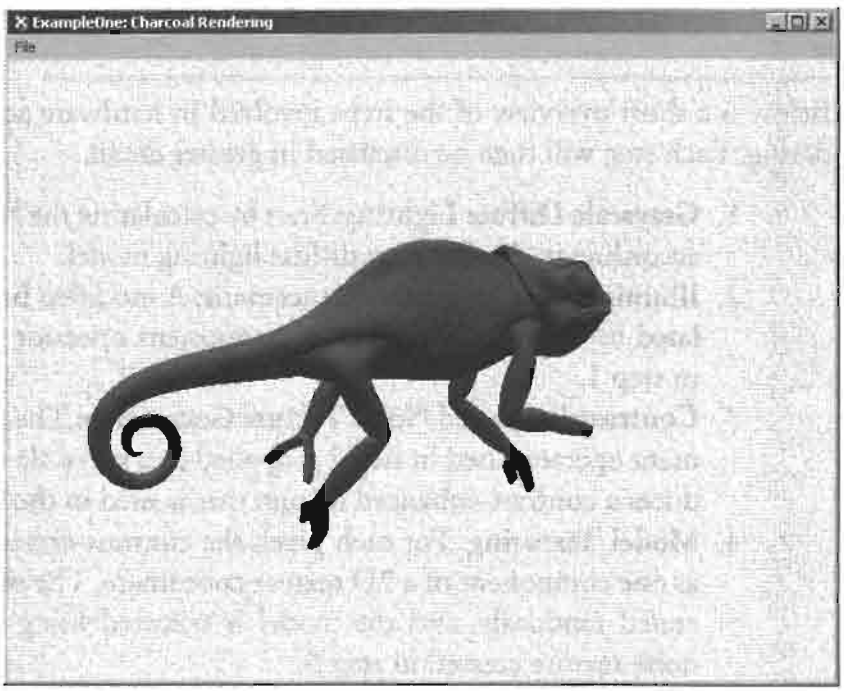


FIGURE 2.10.1 *Grayscale diffuse-lit model.*

Illumination Contrast Enhancement

A contrast enhancement operator *OP* is a function that maps a coordinate x, $x \in [0, 1]$ to a value y, $y \in [0, 1]$. For our purposes, we will use an exponential function $y = x^{\alpha}$ with $\alpha > 0$.

By applying *OP* to the calculated brightness, we get a new contrast-enhanced brightness value for every vertex. The nature of the exponential function results in more noticeable shadows in areas that are not directly lit, and produces a sharper rendering result.

The following code snippet shows the application of *OP* to the previously calculated brightness value.

```
// Applying a contrast enhancement operator.
// y = x^exponent
float fEnhIntensity = pow(fIntensity, fContrastExponent);
Out.Diffuse = fEnhIntensity.xxxx;
```

The result is stored in an output variable to be used in the pixel shader.

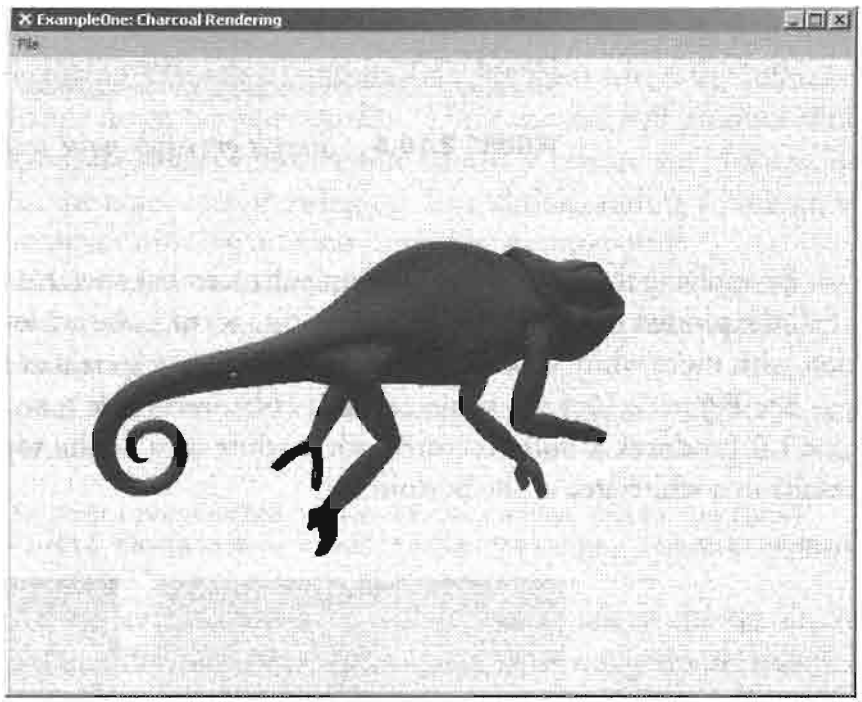


FIGURE 2.10.2 *Intensity-enhanced shaded model.*

Finally, the vertex shader is used to pass 2D clip-space coordinates for each vertex to the pixel shader, for use as paper-texture coordinates in Step 6.

```
// Calculate the clip space position
Out.Position     = mul(Input.Position, matWorldViewProj);
// Calculate 2D clip coords
Out.CoordPaper   = (Out.Position / Out.Position.w) * 0.5f + 0.5f;
```

Contrast-Enhanced Noise Texture Generation

In a pre-processing step we calculate a contrast-enhanced grayscale noise texture. This texture is static and used by the pixel shader to obtain grayscale noise.

We start with a simple noise texture that is generated by randomly distributing black spots on a white background. See Figure 2.10.3 for this intermediate step.

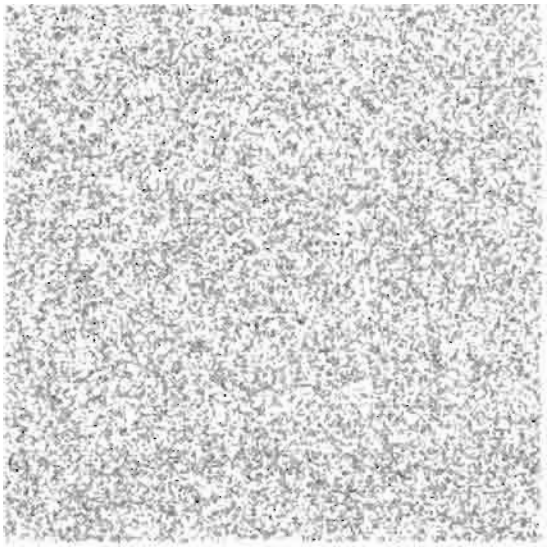


FIGURE 2.10.3 *Simple grayscale noise texture.*

By applying the exponential contrast enhancement operator *OP* to the y-coordinate of all the plotted pixels of the generated noise texture, we achieve a polarized distribution with more white areas at one end and more black areas at the other end.

See Figure 2.10.4 for some contrast-enhanced noise textures. Notice that using $\alpha < 1.0$ produces a noise texture with a white area at the top while using $\alpha > 1.0$ results in a white area at the bottom.

FIGURE 2.10.4 *Contrast-enhanced grayscale noise textures using exponents:* $\alpha = 0.5$, $\alpha = 1.5$, $\alpha = 4.5$.

Texturing the Model Using the Enhanced Grayscale Noise Texture

The next step is to texture the model with the enhanced noise texture using the contrast-enhanced intensity value calculated in step 2 as a texture coordinate. This in fact, results in applying the enhancement operator twice.

Normally, the u/v texture coordinates are interpolated across each triangle, and are used in the pixel shader to sample the texture. In our case, the artist-assigned texture coordinates of the model are not used directly. Instead they are used in the following way:

- The artist-assigned coordinates index into a random noise texture, effectively generating random values.
- The u coordinate is set to one of the random values from the random noise texture.
- The v coordinate is set to the contrast enhanced intensity value of the vertex, plus a different random value from the noise texture.

By using the contrast-enhanced intensity value as the v coordinate we are in fact applying the contrast enhancement operator *OP* to texture the model. The reason we use random values in the u and v coordinates is to prevent the structure of the noise texture showing up in the final output. Imagine areas where the influence of lighting does not change much between vertices. These vertices will produce similar contrast-enhanced intensity values and therefore similar v texture coordinates, increasing the possibility of the noise map showing up. In addition, adding a random value to the v coordinate reduces banding artifacts caused by mipmapping.

To generate a random value in the pixel shader, we use a static noise texture that is also generated during pre-processing. This texture is constructed by calculating a random color for each pixel in the texture.

In the pixel shader we use the artist-assigned texture coordinate to sample the random noise texture.

```
// Retrieve random value (from random noise texture)
float4 colRandom = tex2D(texRandomNoise, Input.CoordRandom);
```

These values are simply noise in all four channels of the texture (the different channels contain different noise—it is colored noise not grayscale noise). Now we just have to fill the two components of a `float2` variable with a combination of the random value and the intensity value already computed in the vertex shader.

```
float2 vecCoordLookup;
vecCoordLookup.x   = colRandom.x * 0.05f;
vecCoordLookup.y   = Input.CoordContrast.y + colRandom.y * 0.05f;
```

Finally we sample the contrast-enhanced grayscale noise texture with the computed texture coordinate.

```
// Sample the contrast enhanced grayscale noise texture with
// the above calculated texture coordinate
float4 colEnhanced = tex2D(texContrastEnhancedNoise,
                           vecCoordLookup);
```

The scaling by 0.05 stops the texture accesses from being too far apart. If this happens, the mipmap hardware selects very high mipmap levels, and thus a very small texture, and banding becomes obvious. Using a small variance in the u coordinate prevents this. Perturbing the v coordinate also helps prevent banding.

Figure 2.10.5 shows an example of the textured model. Referring back to Figure 2.10.4, look at the two noise textures with $\alpha > 1.0$. Notice that the noise pattern is much more visible in the darker areas at the top, while it is not so noticeable at the bottom, where the texture appears much lighter.

Figure 2.10.5 shows that the noise pattern is more intensive in darker regions of the model, while there is hardly any pattern in the very bright locations. This is the result of using an indirect texture read instead of a calculated gradient, and creates the grainy stroke effect that is typical in charcoal drawing.

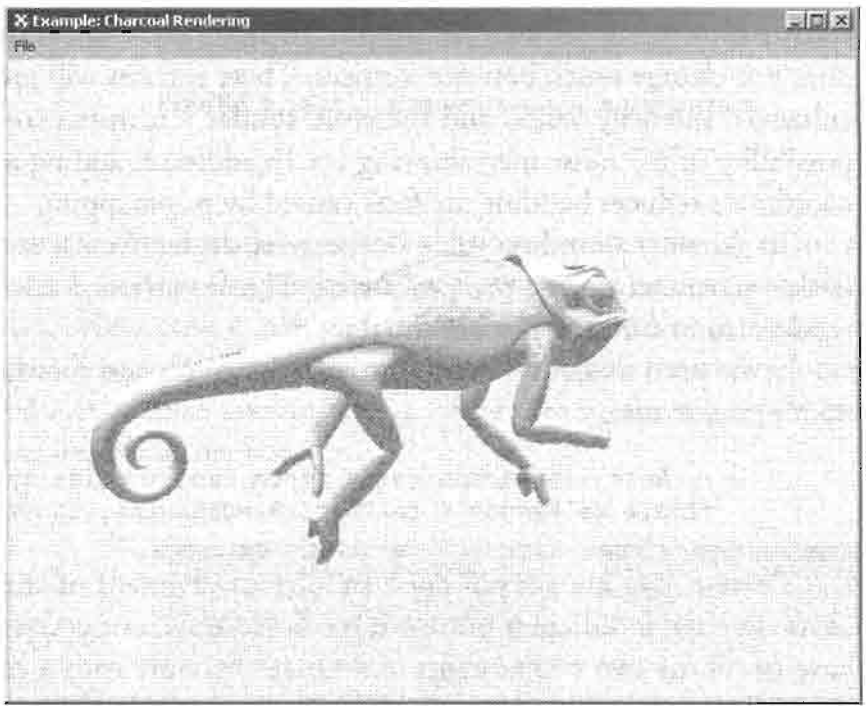


FIGURE 2.10.5 *Model index-textured with contrast-enhanced grayscale noise texture.*

The contrast enhancement done in step 3 helps achieve an effect known as "closure." Closure is used when artists do not explicitly draw parts of an object's silhouette, leaving the closure of the object to the interpretation of the viewer.

Figure 2.10.6 shows the sample model rotated at a slightly different angle compared to the previous screenshots. The closure effect at the top of the chameleon's head and tail is more noticeable in this perspective.

It is also used where you would normally expect a specular highlight caused by a bright light. Modifying the oversaturation factor in the vertex shader controls the visibility of this effect.

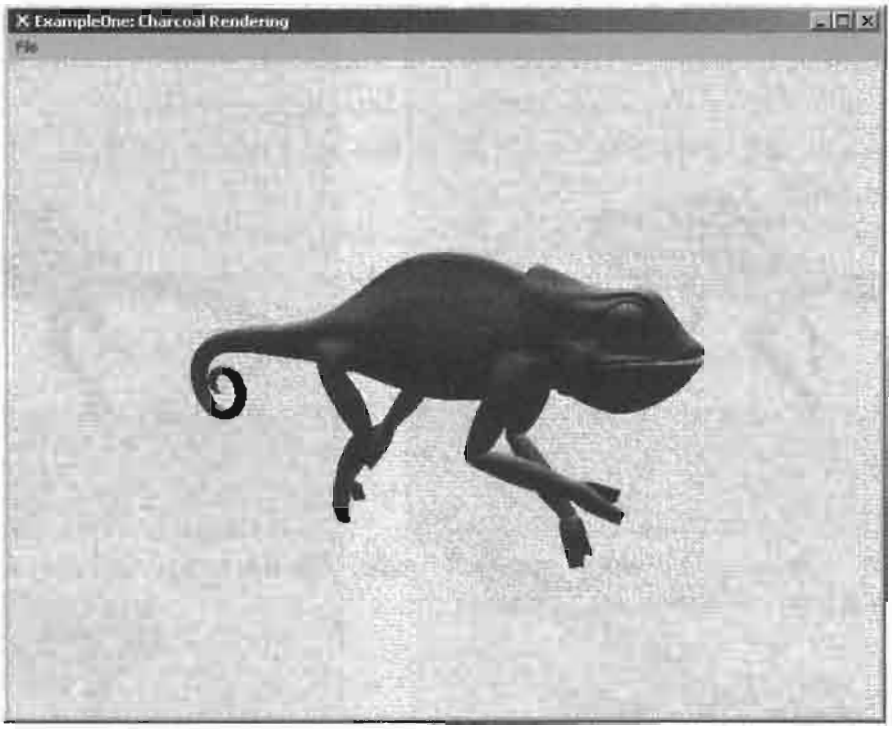


FIGURE 2.10.6 *Final output with closure effect visible at head and tail of the chameleon.*

Blending the Textured Model with the Enhanced Model

In this step we blend the colors looked up from the contrast enhanced texture with those calculated earlier using the diffuse reflection model.

```
// Retrieve the blend ratio from a pixel shader constant
float fRatio = vParams.x;
// Blend the color from the texture with the enhanced diffuse color
float4 colSmudged = lerp (Input.Diffuse.xxxx, colEnhanced, fRatio);
```

The value of the blend ratio is passed to the shader by the application in a pixel shader constant register. This blending operation controls the sharpness of the final output. By blending toward the noise texture components (`colEnhanced`), the image is

more blurred, while blending in more of the intensity-enhanced pixel color (stored in `Input.Diffuse`) results in a sharper and brighter output.

Figure 2.10.7 shows the sample model blended with two different ratios. There is no "rule of thumb" that defines how to choose the blend ratio for special kinds of materials. Depending on your application, you might want to have the same model shown sharper in one place, while it may be displayed more smudged in other places.

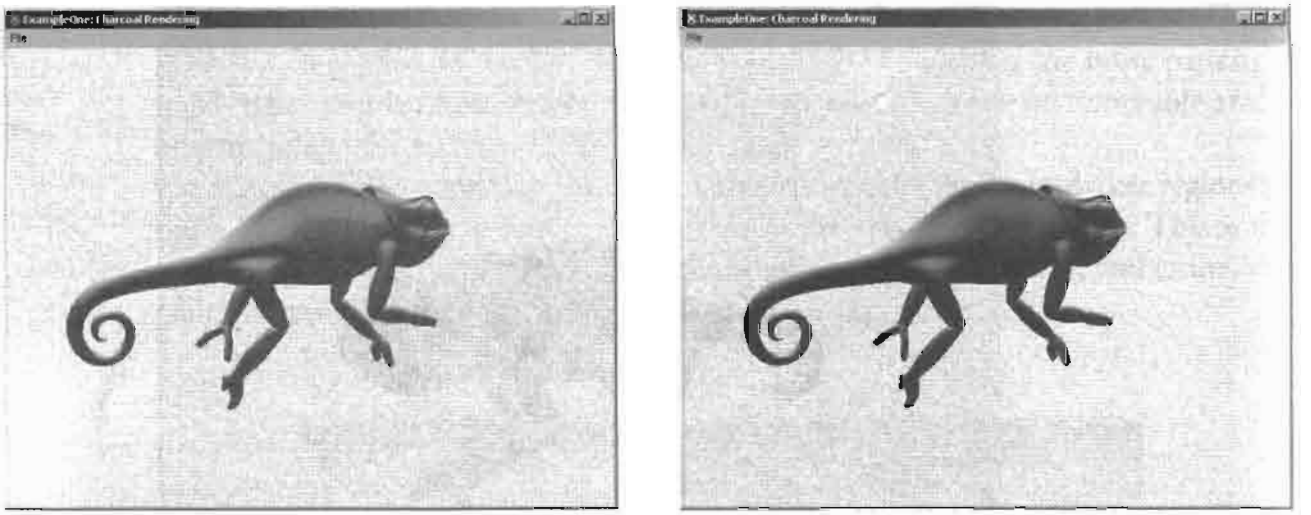


FIGURE 2.10.7 *Result of blending: smudged model. The left image uses a ratio of 60/40, the right one a ratio of 40/60.*

Blending the Final Result with a Background Paper Texture

The last step is optional and is only needed to exactly simulate the effect of artistic charcoal drawings.

The 2D clip coordinates calculated in the vertex shader at the end of step 2 are used in the pixel shader to sample the paper texture. The scene background is this same paper texture.

The texture is used to bump map the model by multiplying the color value calculated so far with the dot product of the normal extracted from the paper texture and the interpolated light vector passed in from the vertex shader.

This happens in the pixel shader.

```
// Bumpmap with paper texture
float4 vecBump  = tex2D(texPaper, Input.CoordPaper) *2.0f - 1.0f;
float3 vecLight = Input.Light*2.0f – 1.0f;
float colPaper    = saturate(dot(vecLight.xyz, vecBump));

// Multiply with previously calculated pixel color
float4 colFinal   = colPaper * colSmudged;
```

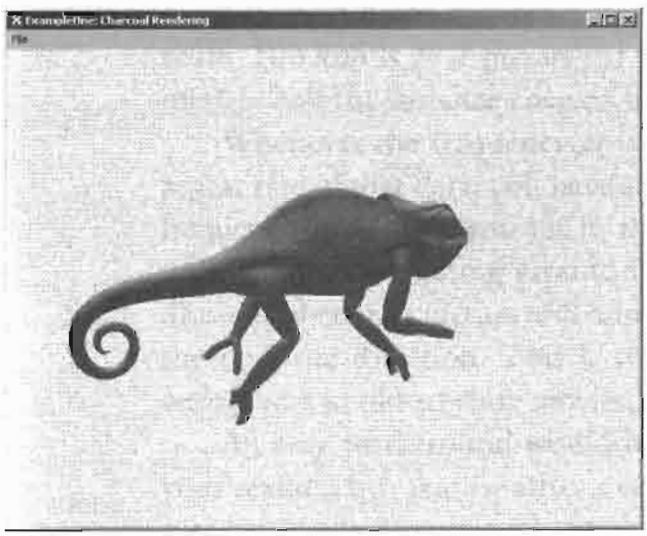
Note here that we are not transferring the light normal into tangent space. Instead, we use the 2D clip space coordinates to look up a normal for the paper texture and calculate the dot product with the worldspace light direction. This bump maps the model in a 2D way that achieves the effect of the model drawn on paper.

Doing tangent space bump mapping here would produce the effect of a paper-skinned model which would destroy the impression of a charcoal drawing, because it would result in "breaks" in the surface paper structure near the model boundaries.

The contrast enhancement results in bright areas being shaded with white colors. Doing the final multiplicative blend with the background paper texture makes the background structure more dominant in areas where the model is brightly lit.

This is the reason that color enhancement also helps to simulate the "closure effect," since multiplication with values around 1.0 makes the background more visible.

Figure 2.10.8 (along with Figure 2.10.9) shows the final result, where our index-textured and lighting-enhanced model is blended with a bumpy paper texture.



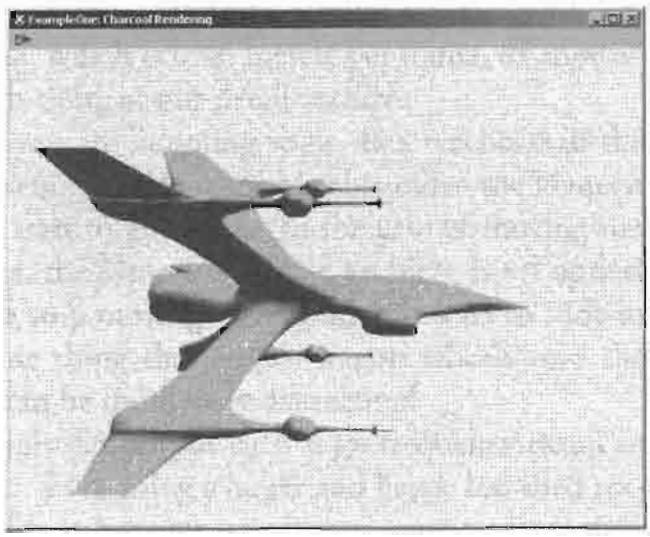


FIGURE 2.10.8 AND FIGURE 2.10.9 *The final result.*

Applications

Due to the non-photorealistic nature of the presented charcoal rendering technique, it can be used to implement a wide variety of real and unreal effects, such as flashback scenes or dream sequences, as well as old-fashioned black and white TV displays, or unnatural and maybe confusing environment lighting. As always in the field of computer graphics, there are nearly no limits for the application of such effects, just the limits to our imagination.

Sample Code

All of the code discussed above is on the book's CD-ROM. This article has talked about most of the shader code, but it may also be interesting to look at other parts we have not discussed, such as the generation of the random noise texture and the contrast enhanced grayscale noise texture as well as all the general shader management code.

Reference

[Majumder02] A. Majumder and M. Gopi, "Hardware Accelerated Real-Time Charcoal Rendering," NPAR 2002: Symposium on Non-photorealistic Animation and Rendering, 2002.