Hatching, Stroke Styles & Pointillism*

Kevin Buchin and Maike Walther

Introduction

Hatching is a common technique used in Non-Photorealistic Rendering (NPR). For hatching, series of strokes are combined into textures. These compositions of strokes can convey the surface form through stroke orientation, the surface material through stroke arrangement and style, and the effect of light on the surface through stroke density.

Up until now an important issue of real-time hatching techniques has been how to employ the limited programmability of the graphics hardware currently available. Pixel programmability has now reached a state where we can shift the focus to adding more flexibility to the hatching scheme and combining hatching with other techniques for creating new effects.

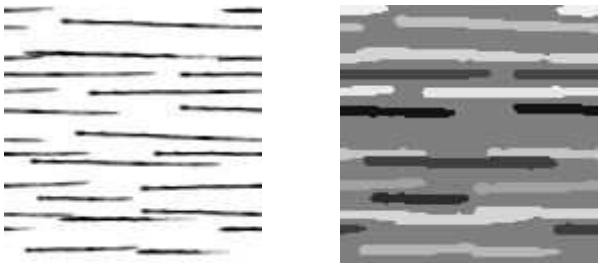
We present a hatching scheme and some extensions to it, namely interactively changing the stroke style, and hatching with specular highlights. As an application, we show how we integrate hand drawings into a scene taking into account the effect of lighting. Finally, we show how to choose a color for each stroke depending on the background color, which can be used for a pointillistic style.

Approaches to Hatching

For hatching, strokes have to be chosen from a collection of possible strokes to convey some tonal value. A possible approach to this problem is to think of each stroke having a priority and to choose strokes according to their priority, i.e. using only the most important strokes for light tonal values and adding less important strokes in areas of darker tonal values. Such collections of strokes are called *Prioritized Stroke Textures* [5] and can be seen as the basis for current hatching schemes.

For real-time hatching, stroke textures for some specific tonal values and different mipmap levels can be precomputed and blended at run-time according to the given tonal value [3]. To maintain a constant stroke width in screen space, the mipmap levels contain strokes of the same texel width. Thus, higher mipmap levels contain fewer strokes than lower mipmap levels for representing

*to appear in: W. Engel, editor, ShaderX^2 - Shader Tips & Tricks. Wordware, 2003



(a) (b)

Figure 1: (a) A texture containing the stroke colors as grey-scale values, (b) a texture containing the corresponding intensity thresholds.

the same tonal value. This technique can be implemented using pixel shaders, as presented in the first ShaderX book [1].

Prioritized Stroke Textures can also be implemented using a *thresholding scheme*, i.e. encoding intensity thresholds and information on resulting color values in a texture. For instance, this information can be differences in tone [4]. While strokes fade in gradually when blending stroke textures for given tonal values, using a thresholding scheme lets strokes appear more suddenly.

There is more to hatching than the actual rendering. In particular, texture-based hatching only works well with an appropriate texture parametrisation. An overview of the complete hatching process is given in [2]. Here we focus on the shader used for hatching.

Our Thresholding Scheme

Our approach is to encode a stroke by its color and intensity threshold. Figure 1 shows a sample stroke texture with gray-scale values (a) and corresponding intensity thresholds (b). An advantage of this approach is that we don't need to decide how the stroke color is combined with the background color - for instance adding, overlaying, modulating or replacing the background color - when creating the textures.

The stroke colors can be stored in the rgb-channels of a texture and the corresponding intensity thresholds in the α-channel of the same texture. To be able to distinguish the color values and intensity thresholds of the different strokes in one texture, the texture may not contain overlapping strokes. For drawing overlapping strokes, we use several textures, for instance two for horizontal strokes and two for vertical strokes. Instead of actually using several textures we can reuse one stroke texture by translating and/or rotating the original texture coordinates. To keep the pixel shader simple, we transform the in the vertex shader by adding several texture coordinates to the output `Out` and – for two horizontal and two vertical stroke textures – the following lines:

```
Out.Tex0 = Tex0;
Out.Tex1 = Tex0 + offset1.xy;
Out.Tex2 = Tex0.yx + offset1.zw;
Out.Tex3 = Tex0.yx + offset2.xy;
```

To each stroke texture we assign an intensity interval [*start*, *end*] and map the thresholds t in the α-channel to this interval by $start + t/(end - start)$. After computing a desired intensity we can modulate the background color with the stroke color using the following lines of code:

```
float4 stroke = tex2D(stroke_sampler, Tex0);
color *= (intensity < start + stroke.a/q) ?  stroke.rgb :  1.0;
```

with $q = end - start$. In the pixel shader, we compute an intensity using a lighting model and, again in the case of two horizontal and two vertical applications of one stroke texture, add the following lines:

```
float3 color = background_color.rgb;
float4 stroke = tex2D(stroke_sampler, Tex0);
color  *= (intensity < 0.75 + stroke.a/4) ?  stroke.rgb;
color  *= (intensity < 0.5 + stroke.a/4) ?  stroke.rgb :  1.0;
stroke = tex2D(stroke_sampler, Tex2);
color  *= (intensity < 0.25 + stroke.a/4) ?  stroke.rgb :  1.0;
stroke = tex2D(stroke_sampler, Tex3);
color  *= (intensity < stroke.a/4) ?  stroke.rgb :  1.0;

return float4(color.r, color.g, color.b, background_color.a);
```

A teapot rendered with this technique is shown in Figure 2.

Figure 2: A teapot hatched using our thresholding scheme.

(a) (b) (c) (d) (e)

Figure 3: (a) - (d) show the $rgb\alpha$-channels of a Stroke-Lookup Texture and (e) an illustration of a lookup. (a) shows the r-channel which contains the lookup in s, (b) the g-channel which contains the lookup in t, (c) the b-channel which contains the threshold, and (d) the α-channel which is used as a stencil.

Varying The Line Style

We can extend the above technique to allow variation of the hatching strokes at run-time. For this, we do not encode strokes directly into a stroke texture, but instead encode lookups into single-stroke textures. We call these textures *Stroke-Lookup Textures.*

A simple example of a Stroke-Lookup Texture is shown in Figure 3. The channels r and g store the lookups in t and s, the channel b stores the threshold and alpha is used as a stencil to prevent incorrect interpolation. For achieving a roughly uniform screen width of the strokes - as in hand drawn hatchings - we use mipmap-levels with strokes of the same texel size. Standard generation of mipmap-levels would half the texel width of a stroke in each level, thus strokes further away from the viewer would be thinner than those close to the viewer. For correct interpolation between these mipmap-levels we extend the stroke-lookup coordinates from $[0, 1]$ to $[-0.5, 1.5]$. For this we scale the texture coordinates appropriately, as illustrated in Figure 4.

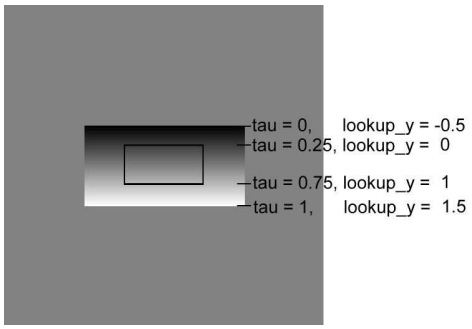


Figure 4: Illustration of the texture coordinates tau and scaled stroke-lookup coordinates `lookup_x` and `lookup_y`.

The above calculations have to be adapted in the following way:

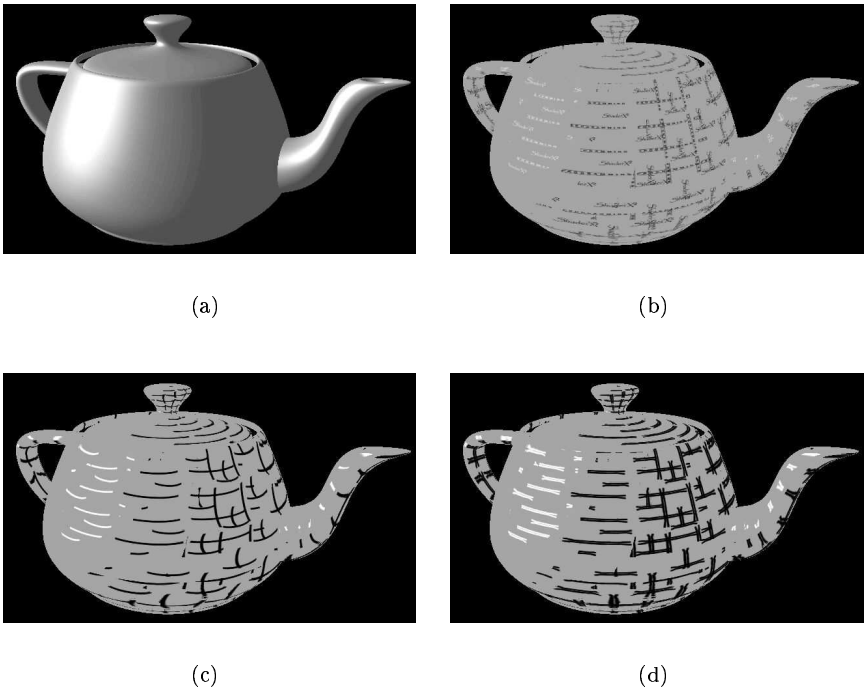
```
float4 lookup = tex2D(stroke_lookup_sampler, Tex0);
lookup.xy = (lookup.xy - 0.25)*2;
bool stroke_flag =
   (intensity < interv.x + interv.y*lookup.b) && (lookup.a > 0.99);
color *=
   stroke_flag ?  tex2D( single_stroke_sampler, lookup.xy) :  1.0;
```

For different strokes, we use lookups into several different single-stroke textures. For this, we use an additional texture with indices for single-stroke textures. Alternatively we could encode the indices into the stenciling channel. To keep the pixel shader simple, we assume that we have done the lighting computation in the vertex shader. The simple pixel shader, using lookups into two different single stroke textures - a short and a long stroke - could look like

```
float4 getStrokeColor(float2 texCoord, float shiftedIntensity)
{
   float4 lookup = tex2D(stroke_lookup_sampler, texCoord);
   lookup.xy = (lookup.xy - 0.25)*2;
   float stroke = tex2D(index, texCoord);
   float4 stroke_color = (stroke < 0.5) ?
   tex2D(short_stroke, lookup.xy) :  tex2D(long_stroke, lookup.xy);
   bool stroke_flag = (lookup.w > 0.99) && (shiftedIntensity < lookup.z/4.0);
   stroke_color = stroke_flag ?  stroke_color :  1.0;
   return stroke_color;
}

float4 main(
   float2 Tex0 :  TEXCOORD0,
   float2 Tex1 :  TEXCOORD1,
   float2 Tex2 :  TEXCOORD2,
   float2 Tex3 :  TEXCOORD3,
   float2 Tex4 :  TEXCOORD4,
   float4 Diff :  COLOR0 ) :  COLOR
{
   float4 color = 1.0;
   color *= getStrokeColor(Tex0, Diff.x - 0.75);
   color *= getStrokeColor(Tex0, Diff.x - 0.75);
   color *= getStrokeColor(Tex1, Diff.x - 0.5);
   color *= getStrokeColor(Tex2, Diff.x - 0.25);
   color *= getStrokeColor(Tex3, Diff.x - 0.0);
   return color;
}
```

Figure 5 shows the use of different stroke styles in combination with specular highlights.



(a) (b)

(c) (d)

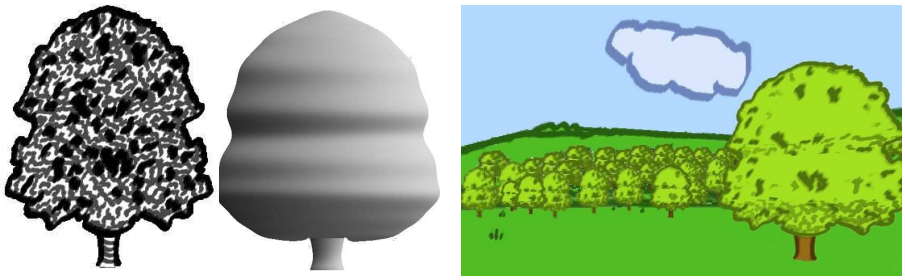
Figure 5: (a) A shaded teapot and (b) - (d) the same teapot hatched with specular highlights and different stroke styles.

Hatching With Specular Highlights

Up to now, we used strokes only to darken a rendition where the desired intensity is below a certain threshold. But we can also draw light strokes - assuming the background is not white. We use this for drawing specular highlights. To draw light strokes, we just need to check whether the intensity is above a given threshold. We can use the same stroke textures as for dark strokes - and possibly combine dark and light strokes - by taking one minus the previous threshold to maintain the stroke priorities. This effect is illustrated in Figure 5.

Lighting Hand drawn Illustrations

Hatching can be used for lighting a hand drawn illustration and integrating it into a 3D-scene. We use billboards for placing the illustration in the scene. For the lighting computation we need to provide normals for the illustration. We do this by roughly approximating our hand drawing by simple primitives and rendering these with a pixel shader outputting the color-encoded (normalized) normal vectors, or with a normalization cube map. Using the normals we choose



(a) (b) (c)

Figure 6: (a) A hand drawn image as input. (b) Approximation of the shape. (c) Resulting image in a 3D-scene.

which strokes to draw according to the light. Figure 6 (a) shows a hand drawn image that we approximated with the shape in (b) and placed in a scene in (c).

Stroke Colors & Pointillism

The last effect chose the stroke color according to the background color. The resulting strokes were uni-colored, simply because the background color of one stroke did not change. In the case of varying background or base color for a stroke, we would still like to draw uni-colored strokes – as is typical for stroke-based illustrations, such as mosaics, oil paintings and many others. We can do this by encoding offsets into the strokes, which are used for reading the base color, so that all points of a stroke read the same color. Figure 7 shows the r-channel of such a stroke. The brush uses values from black (0) to white (1). This value has to be scaled by the maximal relative brush width and

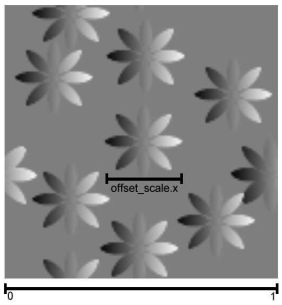


Figure 7: r-channel of the texture with the brush used in Figure 8 and the relative maximal brush width `offset_scale.x`.

height `offset_scale.xy` in the pixel shader. If brushes of different size are used simultaneously, smaller brushes should use values from a smaller interval. The code for modifying the texture coordinate used to determine the base color could like this

```
float2 offset = (tex2D(offset_sampler, Tex.xy*scale).xy  0.5)
   * offset_scale.xy / scale;
float2 newTex = Tex + offset;
```

Used on its own this technique can create a pointillistic style. Figure 8 shows some examples.

Conclusion

Pixel shaders offer great possibilities for implementing stroke-based rendering techniques. As examples of this we have shown a hatching scheme and several effects extending this scheme. We hope that these examples may serve as an inspiration for the many more effects that are possible.

References

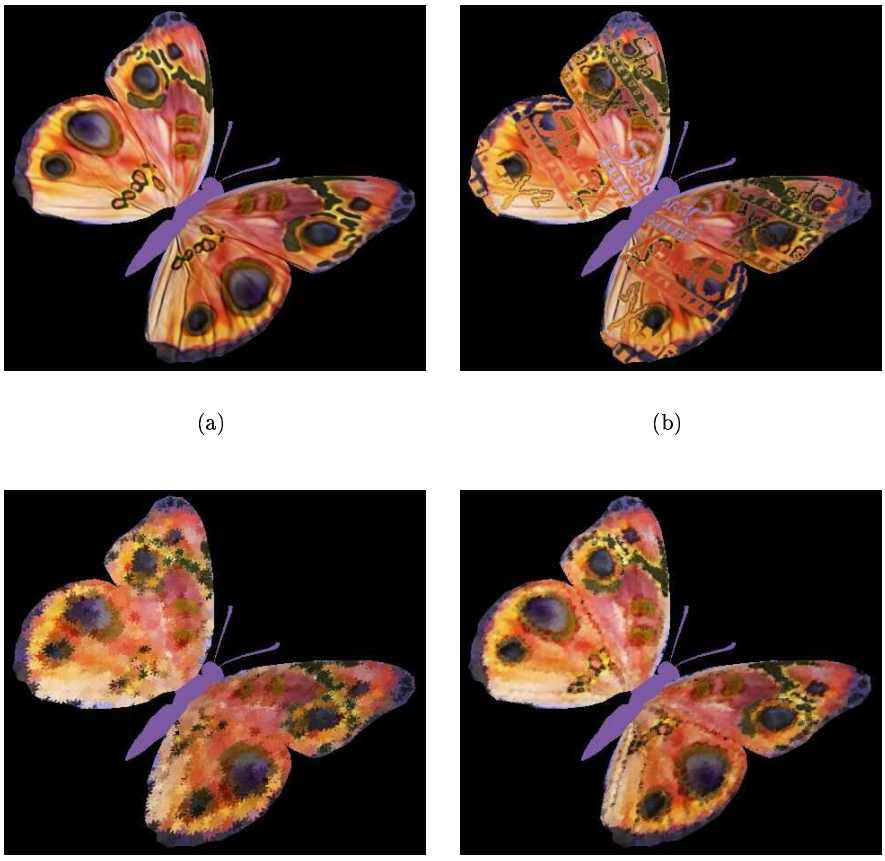
[1] Drew Card and Jason Mitchell. Non-Photorealistic Rendering with Pixel and Vertex Shaders. In Wolfgang Engel, editor, *Direct3d ShaderX: vertex and pixel shader tips and tricks*, pages 319 – 333. Wordware Publishing, Inc., 2002.

[2] Sébastien Dominé, Ashu Rege, and Cem Cebenoyan. Real-Time Hatching (Tribulations in). In *Game Developers Conference 2001*, 2001.

[3] Emil Praun, Hughes Hoppe, Matthew Webb, and Adam Finkelstein. Real-Time Hatching. In Eugene Fiume, editor, *Proceedings of SIGGRAPH'2001 (Los Angeles, August 2001)*, Computer Graphics Proceedings, Annual Conference Series, pages 581–586, New York, 2001. ACM SIGGRAPH.

[4] Matthew Webb, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Fine tone control in hardware hatching. In *Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering*, pages 53–58. ACM Press, 2002.

[5] Georges Winkenbach and David H. Salesin. Computer-Generated Pen-and-Ink Illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH'94 (Orlando, July 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 91–100, New York, 1994. ACM SIGGRAPH.



(a) (b)

(c) (d)

Figure 8: The RenderMonkey Iridescent Butterfly rendered (a) without the effect, (b) using a brush in form of the word. "ShaderX^2", (c) and (d) using the brush shown in Figure 7 with different values for `scale`.