Using Fragment Shaders to Manipulate Images

The Basic Idea: Wrap an Image Around a Piece of Geometry

In software, this is a very slow process. In hardware, this is very fast. The development of texture-mapping hardware was one of the most significant events in the history of computer graphics. This is really what finally enabled game development on a realistic scale.
The Basic Ideas

To prevent confusion, the texture image pixels are not called pixels. A pixel is an RGB dot in the final screen image. An RGB dot in the texture image is called a texture element, or texel.

Similarly, to avoid terminology confusion, a texture image’s width and height dimensions are not called X and Y. They are called S and T.

A texture image is not indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always $S=0$, the right side is $S=1$, the bottom is $T=0$, and the top is $T=1$.

Thus, you do not need to be aware of the texture’s resolution when you are specifying coordinates that point into it. Think of S and T as a measure of what fraction of the way you are into the texture.

Texture mapping is a computer graphics operation in which a separate image, referred to as the texture, is stretched onto a piece of 3D geometry and follows it however it is transformed. This image is also known as a texture map. This can be any image. It can also be data. After all, the contents of a texture are just numbers.
The Basic Ideas

The mapping between the geometry of the 3D object and the S and T of the texture image works like this:

\[(X_0, Y_0, Z_0, S_0, T_0)\]
\[(X_1, Y_1, Z_1, S_1, T_1)\]
\[(X_2, Y_2, Z_2, S_2, T_2)\]
\[(X_3, Y_3, Z_3, S_3, T_3)\]
\[(X_4, Y_4, Z_4, S_4, T_4)\]

Interpolated \((S, T) = (.78, .67)\)

\[[.78, .67] in S and T = (199.68, 171.52) in texels\]

You specify an \((s, t)\) pair at each vertex, along with the vertex coordinate. At the same time that OpenGL is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the \((s, t)\) coordinates. Then, when OpenGL goes to draw each pixel, it uses that pixel’s interpolated \((s, t)\) to lookup a color in the texture image.

Using a Texture: Assigning an \((s, t)\) to each vertex

Enable texture mapping:

```gl
glEnable( GL_TEXTURE_2D );
```

Draw your polygons, specifying \(s\) and \(t\) at each vertex:

```gl
glBegin( GL_TRIANGLES );
gTexCoord2f( s0, t0 );
gNormal3f( nx0, ny0, nz0 );
Vertex3f( x0, y0, z0 );

gTexCoord2f( s1, t1 );
gNormal3f( nx1, ny1, nz1 );
Vertex3f( x1, y1, z1 );

...
```

```gl
glEnd( );
```

(If this geometry is static, i.e., will never change, it is a good idea to put this all into a display list.)

Disable texture mapping:

```gl
glDisable( GL_TEXTURE_2D );
```
Index the image using the usual texture indexing

\((0. \leq s, t \leq 1.)\)

When you get back an RGB from the texture, remember that, if the texture’s numbers are colors:

\((0. \leq r, g, b \leq 1.)\)

If the texture contains data, then the numbers can be anything.

Also, if you need to know the texel resolution of this texture, do this:

\[
\text{ivec2 ires = textureSize( uImageUnit, 0 );}
\text{float ResS = float( ires.s );}
\text{float ResT = float( ires.t );}
\]

Thus, to get from the current texel’s \((s, t)\) to a neighboring texel’s \((s, t)\), add

\[
\pm \left( \frac{1}{\text{ResS}}, \frac{1}{\text{ResT}} \right)
\]

A Good Example of Manipulating RGB Numbers – the Image Negative

Image RGB values are just numbers – they can be manipulated any way you’d like!

\[
\begin{align*}
(R, G, B) & \quad \rightarrow \quad (1.-R, 1.-G, 1.-B)
\end{align*}
\]
Image Negative

### OpenGL GLIB

- Perspective 70
- LookAt 0.0.0.6.0.0.0.0.1.0.
- texture 5 image.bmp
- Vertex neg.vert
- Fragment neg.frag
- Program Neg TexUnit 5
- QuadXY .2 5.

Vertex shader

```glsl
#version 330 compatibility

out vec2 vST;

void main() {
    vST = gl_MultiTexCoord0.st;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

If you are using a Mac:
- Leave out the `#version` line
- Use `varying` instead of `out/in`
**Image Negative**

Fragment shader

```glsl
#version 330 compatibility
uniform sampler2D uTexUnit;
in vec2 vST;

void main()
{
    vec3 rgb = texture(uTexUnit, vST).rgb;
    gl_FragColor = vec4(1.-rgb.r, 1.-rgb.g, 1.-rgb.b, 1.);
}
```

If you are using a Mac:
- Leave out the `#version` line
- Use `varying` instead of `out/in`
- Use the `texture2D()` function instead

Could also have said:

```glsl
gl_FragColor = vec4(vec3(1.,1.,1.) - rgb, 1.);
```

**Image Distortion**

Fragment shader

```glsl
uniform float uS0, uT0;
uniform float uPower;
uniform sampler2D uTexUnit;
in vec2 vST;

void main()
{
    vec2 delta = vST - vec2(uS0, uT0);
    st = vec2(uS0, uT0) + sign(delta) * pow(abs(delta), uPower);
    vec3 rgb = texture(uTexUnit, st).rgb;
    gl_FragColor = vec4(rgb, 1.);
}
```
Image Un-masking:
Interpolation can still happen when $t < 0.$ or $t > 1.$

$$Q = (1 - t)Q_0 + tQ_1 = \text{mix}(Q_0, Q_1, t)$$

$t = -1.$
More dino, negative sphere

$t = 0.$
All dino, no sphere

$t = 1.$
All sphere, no dino

$t = 2.$
More sphere, negative dino

Image Un-Masking:
Abusing the Linear Blending Equation for a Good Purpose

$$I_{\text{out}} = (1 - t)I_{\text{dontwant}} + tI_{\text{in}}$$

$$Q = (1 - t)Q_0 + tQ_1$$

$$RGB_{\text{out}} = \text{mix}(RGB_{\text{dontwant}}, RGB_{\text{in}}, t)$$
**Brightness**

\[ I_{\text{dont want}} = \text{vec3}(0., 0., 0.); \]

\[ Q = (1 - t)Q_0 + tQ_1 \]

\[ RGB_{\text{out}} = \text{mix}(RGB_{\text{dont want}}, RGB_{\text{in}}, t) \]

**Contrast**

\[ I_{\text{dont want}} = \text{vec3}(0.5, 0.5, 0.5); \]

\[ Q = (1 - t)Q_0 + tQ_1 \]

\[ RGB_{\text{out}} = \text{mix}(RGB_{\text{dont want}}, RGB_{\text{in}}, t) \]
HDTV Luminance Standard

Luminance = 0.2125*Red + 0.7154*Green + 0.0721*Blue

Saturation

\[ I_{\text{d}ont\text{want}} = \text{vec3}(\text{luminance, luminance, luminance}); \]

\[ Q = (1 - t)Q_0 + tQ_1 \]

\[ RGB_{\text{out}} = \text{mix}(RGB_{\text{d}ont\text{want}}, RGB_{\text{in}}, t) \]
Difference

\[ I_{\text{dontwant}} = I_{\text{before}} \]
\[ I_{\text{in}} = I_{\text{after}} \]

\[ Q = (1-t)Q_0 + tQ_1 \]
\[ RGB_{\text{out}} = \text{mix}(RGB_{\text{dontwant}}, RGB_{\text{in}}, t) \]

ChromaKey

Replace the fragment if:
\[ R < t \]
\[ G < t \]
\[ B > 1-t \]
Blue/Green Screen Usage is ChromaKey

Loyal Studios

https://www.youtube.com/watch?v=Ldh6FKavxek

Blur

Blur Convolution:

\[
B = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad B = \frac{1}{100} \begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 4 & 8 & 16 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix}
\]
**Sharpening**

**Blur Convolution:**

Using the 3x3 Blur Convolution:

\[
B = \frac{1}{16} \begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1
\end{bmatrix}
\]

\[I_{\text{dontwant}} = I_{\text{blur}}\]

\[Q = (1-t)Q_0 + tQ_1\]

\[R_{\text{out}} = \text{mix}(R_{\text{dontwant}}, R_{\text{in}}, t)\]

\[\text{vec2 stp0} = \text{vec2}(1./\text{ResS}, 0.);\]
\[\text{vec2 st0p} = \text{vec2}(0., 1./\text{ResT});\]
\[\text{vec2 stpp} = \text{vec2}(1./\text{ResS}, 1./\text{ResT});\]
\[\text{vec2 stpm} = \text{vec2}(1./\text{ResS}, -1./\text{ResT});\]

\[\text{vec3 i00} = \text{texture(ulmageUnit, vST).rgb};\]
\[\text{vec3 im1m1} = \text{texture(ulmageUnit, vST-stpp).rgb};\]
\[\text{vec3 ip1p1} = \text{texture(ulmageUnit, vST+stpp).rgb};\]
\[\text{vec3 im1p1} = \text{texture(ulmageUnit, vST-stpm).rgb};\]
\[\text{vec3 im1m1} = \text{texture(ulmageUnit, vST+stpm).rgb};\]
\[\text{vec3 im10} = \text{texture(ulmageUnit, vST-stp0).rgb};\]
\[\text{vec3 ip10} = \text{texture(ulmageUnit, vST+stp0).rgb};\]
\[\text{vec3 i0m1} = \text{texture(ulmageUnit, vST-st0p).rgb};\]
\[\text{vec3 i0p1} = \text{texture(ulmageUnit, vST+st0p).rgb};\]

\[\text{vec3 blur} = \text{vec3}(0.,0.,0.);\]
\[\text{blur} += 1.*(\text{im1m1}+\text{ip1m1}+\text{ip1p1}+\text{im1p1});\]
\[\text{blur} += 2.*(\text{im10}+\text{ip10}+\text{i0m1}+\text{i0p1});\]
\[\text{blur} += 4.*(\text{i00});\]
\[\text{blur} /= 16.;\]

\[\text{gl_FragColor} = \text{vec4(\text{mix(blur, irgb, t)}}, 1.);\]
Sharpening

t = 0.

t = 1.

t = 2.

Embossing

```glsl
vec2 stp0 = vec2( 1./ResS, 0. );
vec2 stpp = vec2( 1./ResS, 1./ResT);
vec3 c00 = texture( uImageUnit, vST ).rgb;
vec3 cp1p1 = texture( uImageUnit, vST + stpp ).rgb;
vec3 diffs = c00 - cp1p1;
float max = diffs.r;
if( abs(diffs.g) > abs(max) )
  max = diffs.g;
if( abs(diffs.b) > abs(max) )
  max = diffs.b;
float gray = clamp( max + .5, 0., 1. );
vec4 grayVersion = vec4( gray, gray, gray, 1. );
vec4 colorVersion = vec4( gray*c00, 1. );
gl_FragColor = mix( grayVersion, colorVersion, t );
```
**Edge Detection**

**Horizontal and Vertical Sobel Convolutions:**

\[
H = \begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1
\end{bmatrix} \quad V = \begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1
\end{bmatrix}
\]

\[
S = \sqrt{H^2 + V^2} \quad \Theta = \text{atan2}(V, H)
\]

**Computer Graphics**

const vec3 LUMCOEFFS = vec3(0.2125, 0.7154, 0.0721);

\[
\begin{align*}
\text{stp0} & = \text{vec2}(1./\text{ResS, 0.}); \\
\text{st0p} & = \text{vec2}(0., 1./\text{ResT}); \\
\text{stpp} & = \text{vec2}(1./\text{ResS, 1./ResT}); \\
\text{stpm} & = \text{vec2}(1./\text{ResS, -1./ResT});
\end{align*}
\]

\[
\begin{align*}
\text{i00} & = \text{dot( texture( uImageUnit, vST ).rgb, LUMCOEFFS );} \\
\text{im1m1} & = \text{dot( texture( uImageUnit, vST-stpp ).rgb, LUMCOEFFS );} \\
\text{ip1p1} & = \text{dot( texture( uImageUnit, vST-stpp ).rgb, LUMCOEFFS );} \\
\text{ip11} & = \text{dot( texture( uImageUnit, vST-stpp ).rgb, LUMCOEFFS );} \\
\text{im10} & = \text{dot( texture( uImageUnit, vST-stpp ).rgb, LUMCOEFFS );} \\
\text{ip10} & = \text{dot( texture( uImageUnit, vST-stpp ).rgb, LUMCOEFFS );} \\
\text{ip01} & = \text{dot( texture( uImageUnit, vST-stpp ).rgb, LUMCOEFFS );}
\end{align*}
\]

\[
\begin{align*}
h & = -1.\text{im1p1} - 2.\text{ip0p1} - 1.\text{ip1p1} + 1.\text{im1m1} + 2.\text{ip0m1} + 1.\text{ip1m1}; \\
v & = -1.\text{im1m1} - 2.\text{im10} - 1.\text{im1p1} + 1.\text{ip1m1} + 2.\text{ip10} + 1.\text{ip1p1}; \\
mag & = \text{sqrt( h*h + v*v );}
\end{align*}
\]

\[
\text{vec3 target} = \text{vec3( mag, mag, mag );} \\
\text{color} = \text{vec4( mix( irgb, target, t ), 1. );}
\]
**Edge Detection**

$t = 0.$  
$t = 0.5$  
$t = 1.$

**Toon Rendering**

Hand-drawn cartoons have a unique style, typically characterized by:

1. Dark outlines between the important elements in the scene
2. A reduced collection of available colors (i.e., no smooth shading)

http://drawdo.com/draw/draw-winnie-the-pooh/
float mag = sqrt( h*h + v*v );
if( mag > uMagTol )
{
    gl_FragColor= vec4( 0., 0., 0., 1. );
}
else
{
    rgb.rgb *= uQuantize; // scale up
    rgb.rgb += vec3(.5, .5, .5 ); // round
    ivec3 irgb = ivec3( rgb.rgb ); // cast to all integers
    rgb.rgb = vec3( irgb ); // cast back to floats
    rgb /= uQuantize; // scale down
    gl_FragColor= vec4( rgb, 1. );
}

Quantizing example using the number 3.14159:

<table>
<thead>
<tr>
<th>uQuantize</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.</td>
<td>3.1</td>
</tr>
<tr>
<td>100.</td>
<td>3.14</td>
</tr>
<tr>
<td>1000.</td>
<td>3.141</td>
</tr>
</tbody>
</table>

These are just examples – uQuantize does not need to be a power of 10!
Toon Rendering for Non-Photorealistic Effects

Using shaders to enhance scientific, engineering, and architectural illustration

Photo by Steve Cunningham
Toon Rendering for Non-Photorealistic Effects

Mandelbrot Set

\[ z_{i+1} = z_i^2 + z_0 \]

How fast does it converge, if ever?
Julia Set

\[ Z_{i+1} = Z_i^2 + C \]

How fast does it converge, if ever?
Doing the Mandelbrot Math in Single vs. Double Precision

32-bit single precision floating point

64-bit double precision floating point
We Can Do Image Processing on Dynamic Scenes with a Two-pass Approach

Pass #1

Render a 3D dynamic scene → Texture

Pass #2

Render a quadrilateral → Framebuffer

Sharpen Image Shader

Original

Sharpened
Original

Sharpened