11 Image Manipulation with Shaders

The OpenGL computer graphics API is primarily intended for rendering 3D synthetic scenes from geometric primitives, but some capabilities for manipulating images were built into the system from the beginning. With the addition of shader capabilities, OpenGL can now use texture access and manipulation operations to carry out a number of new image functions. In this chapter, we describe some of these functions. Our main tools will be the ability to get texels directly from a texture and the ability to do arithmetic on texel values.

The general form of the GLIB file is as below, including a uniform slider variable T, used in case you use a parameterized operation such as image blending, and variables for the resolution of the image file. Each texture needs to be assigned to a texture unit. Here we have set up the GLIB file for two textures, because some of the later examples in this chapter operate on two

images. Of course there will be changes if we work on a single image (using only one file, `sample.bmp`), if we use a different image (replacing the name of the image file), or if we include additional uniform variables to support other computations.

```
##OpenGL GLIB

Ortho -1. 1. -1. 1.

Texture 5 sample1.bmp
Texture 6 sample2.bmp

Vertex sample.vert
Fragment sample.frag
Program Sample  uT <0. 0. 5.>   \
                uImageUnit 5 uImage2Unit 6

QuadXY .2 5.
```

If you are using *glman*, do not use texture units 2 and 3, because *glman* uses those to hold its built-in 2D and 3D noise textures.

This GLIB file puts the two texture images on texture units 5 and 6. You can arbitrarily pick which texture units to use, up to the total number supported by your graphics card.

The vertex shader is short and uses our familiar conventions for variable names:

```
out vec2 vST;

void main( )
{
  vST = aTexCoord0.st;
  gl_Position = uModelViewProjectionMatrix * aVertex;
}
```

Throughout this chapter we will be looking at different image manipulation functions that we can build into fragment shaders.

Basic Concepts

GLSL deals with images by treating them as textures and using texture access and manipulation to set the color of each pixel in the color buffer. This color buffer may then be displayed, letting you see the effect of your manipulation,

or it may be saved as another texture or output as a file.

In Figure 11.1, we see a texture image as it might have been read from an image file. This texture file may be treated as an image raster by working with each texel individually. A built-in GLSL function `textureSize( )` will tell you the resolution of the texture, called *ResS* and *ResT* in Figure 11.1.

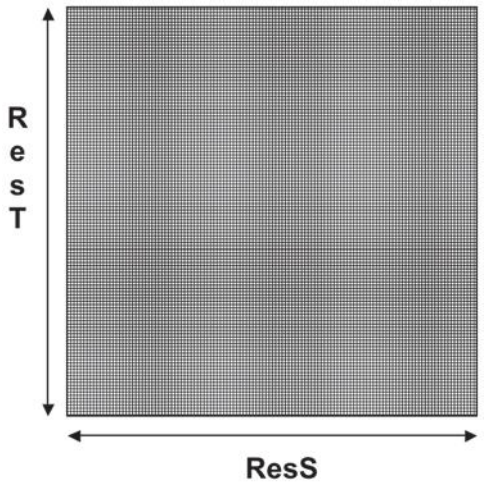

Figure 11.1. A texture raster that could be created from an image file.

There are two ways to access a single texel in a texture. Since any OpenGL texture has texture coordinates ranging from 0.0 to 1.0, the coordinates of the center of the image are `vec2(0.5,0.5)` and you can increment texture coordinates by 1./*ResS* or 1./*ResT* to move from one texel to another horizontally or vertically, respectively. Alternately, if you are working with GLSL 1.50 (OpenGL 3.2) or higher, you can access any texel with the `texelFetch( )` function. We will use these GLSL texture-access capabilities in the fragment shader to identify and calculate colors for pixels in the color buffer.

In order to be as general as possible, we will address and increment texture coordinates with real numbers rather than integers, in spite of the weakness in this approach, since it can lead to some unintentional interpolations of pixel values.

Single-Image Manipulation

In the next several sections, we work with an individual image and compute the color of output pixels by using information contained in the image. This is in contrast to some later sections in this chapter, when we use two different images as textures loaded into different texture units in our computation.

Luminance

The luminance of a color is the overall brightness of the color, with no reference to the color's hue. Luminance is a more complex property than it might seem, because our eyes respond to different primary colors differently. Luminance has been studied because of the need to give luminance cues to persons who have deficient color vision, as described in [15, Chapter 5], and because it was

necessary to consider luminance when creating a color system that could support both black-and-white and color television.

The sRGB specification (also known as IEC 61966-2-1) is emerging as a standard way to define colors across various monitors and applications [42]. In sRGB, luminance is defined as a linear combination of red, green, and blue. The weight vector for luminance in sRGB is

```
const vec3 W = vec3( 0.2125, 0.7154, 0.0721 );
```

We use this set of weights in much of the upcoming sample vertex shader code to compute the luminance of a pixel by taking the dot product of the vector `.rgb` with this weighting vector as follows:

```
vec3 irgb = texture( uImageUnit, vST ).rgb;
float luminance = dot( irgb, W );
```

Note that these numbers in the weight vector W sum to 1.0000 so that dotting this vector with a legitimate RGB vector will produce a luminance between 0 and 1. We will find luminance to be an important concept in several image manipulation techniques, such as grayscale. Grayscale conversion of an image is accomplished by replacing the color of each pixel with its luminance value. When you compute each pixel's luminance, as shown in the code fragment above, you can create a grayscale representation of the image by setting the pixel color to a vector of the luminance value:

```
fFragColor = vec4( luminance, luminance, luminance, 1.);
```

A conversion from a color image to grayscale in this way is shown in Figure 11.2.

Figure 11.2. A supermarket fruit image (left) and its grayscale equivalent (right).

CMYK Conversions

A common function when you are doing graphics that will be published using standard printing process is converting your RGB-color images to CMYK-color. The RGB color model is based on emissive colors, adding color components to black, as used by computer monitors. The CMYK color model is a transmissive model, created by subtracting color components from white. Standard printing uses four subtractive color components: cyan, magenta, yellow, and black. Converting RGB colors to CMYK colors and outputting the four single-color images is called creating CMYK separations. The single-color images are used to create four printing plates. The conversion from the RGB color space to the CMYK color space is straightforward, although there are different approaches. The examples shown here are taken from [5].

RGB to CMYK conversion works like this. First, convert RGB to CMY by subtracting the RGB color from white. Then calculate the amount of black in each color and segregate it out as the K value, then adjust each of the CMY colors to reflect the fact that this K is present. Sample fragment shader code to convert a variable `vec3 color` to a variable `vec4 cmykcolor` is shown here.

```
vec3 cmycolor = vec3(1., 1., 1.) - color;
float K = min( cmycolor.x, min(cmycolor.y, cmycolor.z) );
vec3 temp = (cmycolor - vec3(K,K,K,) )/(1.0 - K);
vec4 cmykcolor = vec4(temp, K);
```

A more complex, but much more satisfactory, conversion scales the values of `cmycolor` above by modifying the value of K used to convert to `cmykcolor`. This approach, which yields a good approximation of the Adobe Photoshop CMYK conversion, is given by

$$\begin{pmatrix} C' \\ M' \\ Y' \\ K' \end{pmatrix} = \begin{pmatrix} C - f_{UCR}(K) \\ M - f_{UCR}(K) \\ Y - f_{UCR}(K) \\ f_{BG}(K) \end{pmatrix}$$

where the functions f_{UCR} and f_{BG} are given by

$$f_{UCR}(K) = S_K * K$$

$$f_{BG}(K) = \begin{cases} 0 & K < K_0 \\ K_{\max} * \dfrac{K - K_0}{1 - K_0} & K \geq K_0 \end{cases}$$

Figure 11.3. A color image (top) and the four CMYK separations (shown in grayscale) in C-M-Y-K order.

where $S_K = 0.1$, $K_0 = 0.3$, and $K_{\max} = 0.9$. This approach is used in developing Figure 11.3.

A separation is a grayscale image that captures one of the C, M, Y, or K components of the image. These are output as files to be used in printing either on film or digitally. To create a separation, you use code such as that above and replace each pixel's color with the single-color grayscale. For example, to create the magenta separation, we could use

```
fFragColor = vec4( cmykcolor.yyy, 1.);
```

Since there is no "cmyk" nameset, and since namesets pay no attention to the meaning of the components, we have used the xyzw nameset for the `vec4 cmykcolor` in this example.

An example of creating separations is shown in Figure 11.3, which shows an original color image and four separations created with this technique. The separations are shown in grayscale to emphasize the amount of ink that would be required to print each; darker values in the separations indicate that more ink of that color will be used at that point. The most obvious effect in this fruit image is the yellow tones in the fruits and the foliage, along with the magenta tones from the red fruit colors.

The fragment shader for this CMYK conversion is shown below, with the variables in the discussion hard-coded for this example.

```
#define CYAN
#undef   MAGENTA
#undef   YELLOW
#undef   BLACK

uniform sampler2D uImageUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec3 irgb = texture( uImageUnit, vST ).rgb;

  vec3 cmycolor = vec3( 1., 1., 1. ) - irgb;
  float K = min( cmycolor.x, min(cmycolor.y, cmycolor.z) );
  vec3 target = cmycolor - 0.1 * K;
  if (K < 0.3) K = 0.;
  else K = 0.9 * (K - 0.3)/0.7;
  vec4 cmykcolor = vec4( target, K );

#ifdef CYAN
  fFragColor = vec4( vec3(1. - cmykcolor.x), 1. );
#endif

#ifdef MAGENTA
  fFragColor = vec4( vec3(1. - cmykcolor.y), 1. );
#endif

#ifdef YELLOW
  fFragColor = vec4( vec3(1. - cmykcolor.z), 1. );
#endif

#ifdef BLACK
  fFragColor = vec4( vec3(1. - cmykcolor.w), 1. );
#endif
}
```

Hue Shifting

Along with the conversion to CMYK color, you can also convert among the other major color models. We assume that you are familiar with the HLS and HSV color models [14], and we will implement hue shifting by converting RGB to either HLS or HSV color, changing the hue in the new color model, and then shifting back to RGB. The effect of this kind of image shifting is shown in Figure 11.4.

Figure 11.4. A color image and the same image with hue shifted by 240 degrees.

Some sample fragment shader code to do this is shown below, using the HSV color model. This color model is used because the hue is an angular function, and you can shif color easily by adding a numeric value to the hue and taking the result mod 360. The color conversions from RGB to HSV and back from HSV to RGB use two functions from [18]. The hue-shifting shader is written to use the *glman* slider variable *T*, with range [0., 360.], to control the amount of the hue shift.

```
uniform float uT;
uniform sampler2D uImageUnit;

in vec2 vST;

out vec4 fFragColor;

vec3
convertRGB2HSV( vec3 rgbcolor )
```

```
{
  float h, s, v;

  float r = rgbcolor.r;
  float g = rgbcolor.g;
  float b = rgbcolor.b;
  float v = float maxval = max( r, max( g, b ) );
  float minval = min( r, min( g, b ) );
  if (maxval==0.) s = 0.0;
  else s = (maxval - minval)/maxval;

  if (s == 0.)
    h = 0.; // actually h is indeterminate in this case
  else
  {
    float delta = maxval - minval;
    if ( r == maxval ) h = (g - b)/delta;
    else
      if (g == maxval) h = 2.0 + (b - r)/delta;
      else
       if (b == maxval) h = 4.0 + (r - g)/delta;
    h *= 60.;
    if (h < 0.0) h += 360.;
  }
  return vec3( h, s, v );
}

vec3
convertHSV2RGB( vec3 hsvcolor )
{
  float h = hsvcolor.x;
  float s = hsvcolor.y;
  float v = hsvcolor.z;
  if (s == 0.0) // achromatic- saturation is 0
  {
    return vec3(v,v,v); // return value as gray
  }
  else // chromatic case
  {
    if (h > 360.0) h = 360.0; // h must be in [0, 360)
    if (h < 0.0) h = 0.0; // h must be in [0, 360)
    h /= 60.;
    int k = int(h);
    float f = h - float(k);
    float p = v * (1.0 - s);
    float q = v * (1.0 - (s * f));
    float t = v * (1.0 - (s * (1.0 - f)));
```

```
    if (k == 0) return vec3 (v, t, p);
    if (k == 1) return vec3 (q, v, p);
    if (k == 2) return vec3 (p, v, t);
    if (k == 3) return vec3 (p, q, v);
    if (k == 4) return vec3 (t, p, v);
    if (k == 5) return vec3 (v, p, q);
  }
}

void main( )
{
  vec3 irgb = texture( uImageUnit, vST ).rgb;
  vec3 ihsv = convertRGB2HSV( irgb );
  ihsv.x += uT;
  if (ihsv.x > 360.)  ihsv.x -= 360.; //add to hue
  if (ihsv.x < 0.)    ihsv.x += 360.; //add to hue
  irgb = convertHSV2RGB( ihsv );
  fFragColor = vec4( irgb, 1. );
}
```

This example includes an implicit conversion between the RGB color representation and the HSV color representation, showing how more general color conversions may be done.

Image Filtering

A number of image manipulations are based on filtering images. A filter is a process that convolves a pixel with its neighbors by using a matrix to weight neighboring pixels. The size of the filter, the values in the filter, and the meaning of different values that are returned when a filter is applied, all vary from algorithm to algorithm.

As two examples of filters, consider the following. One is a three-by-three Sobel filter that is used to detect horizontal edges. The other is a five-by-five blur filter that can be used to smooth (or blur) an image:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad \frac{1}{273} * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}.$$

The filters are used as weights in creating a weighted sum of the values in an adjacent set of pixels. For pixel values P_{ij} and filter elements F_{ij} and a filter

width of $2 * n + 1$, we can express this weighted sum as

$$\sum_{i=-n}^{n} \sum_{j=-n}^{n} F_{ij} * P_{ij}$$

There are some general properties that filters may have. Filters are often square matrices, usually of odd size, so we can often talk about a 3×3 or 5×5 filter. The sum of the weights in the filter is often one, especially when the overall content of an array is to be preserved, so applying a filter usually does not change the overall magnitude of whatever the filter is applied to.

Image Blurring

Image blurring can be done by applying a simple symmetric filter to the image, so that each pixel's color is influenced by the color of each of its neighbors. You can use a simple 3×3 blur convolution filter like the one below or a larger 5×5 blur convolution filter like the 5×5 example shown above:

$$\frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

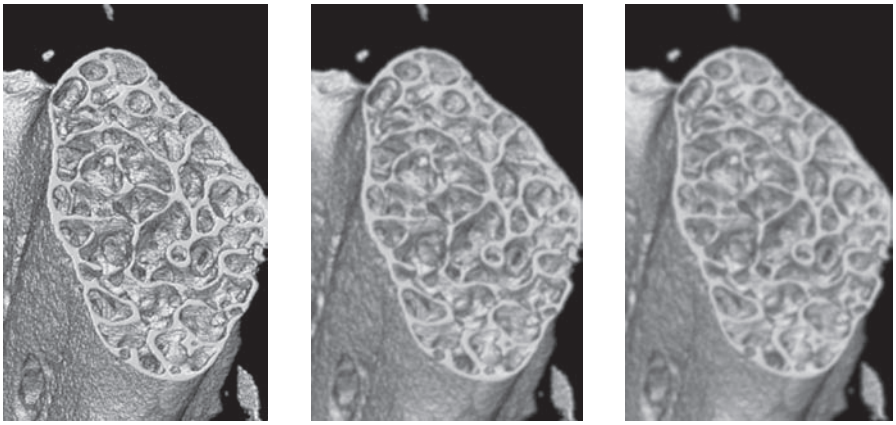

Figure 11.5. An original image (left) blurred by a 3 × 3 filter (center) and a 5 × 5 filter (right).

Examples of these two filters' effects are shown in Figure 11.5 and are compared with an original unblurred image, both to show you the blurred images and to let you compare the amount of blurring generated by each of these filters. Because blurring is not very easy to see in reduced-size naturalistic images, we have chosen an original visualization image whose edges are particularly pronounced.

Below is a fragment shader that applies the 3×3 blur convolution filter above to a set of pixels to blur an image. The computation is done without formal matrix multiplication as the pixels with weight 1.0 are gathered, as are those with weight 2.0 and the single pixel with weight 4, and the result is divided by the overall weight. The code for a 5×5 blur filter would look quite similar, and both are included with the resources for this book; the difference is that four additional pixel addresses are needed, and 25 individual pixel colors are generated, instead of the nine shown here.

```
uniform sampler2D uImageUnit;

in vec2 vST

out vec4 fFragColor;

void main( )
{
  ivec2 ires = textureSize( uImageUnit, 0 );
  float ResS = float( ires.s );
  float ResT = float( ires.t );
  vec3 irgb = texture( uImageUnit, vST ).rgb;

  vec2 stp0 = vec2(1./ResS, 0. ); // texel offsets
  vec2 st0p = vec2(0.   , 1./ResT);
  vec2 stpp = vec2(1./ResS, 1./ResT);
  vec2 stpm = vec2(1./ResS, -1./ResT);

// 3x3 pixel colors next

  vec3 i00   = texture( uImageUnit, vST ).rgb;
  vec3 im1m1 = texture( uImageUnit, vST-stpp ).rgb;
  vec3 ip1p1 = texture( uImageUnit, vST+stpp ).rgb;
  vec3 im1p1 = texture( uImageUnit, vST-stpm ).rgb;
  vec3 ip1m1 = texture( uImageUnit, vST+stpm ).rgb;
  vec3 im10  = texture( uImageUnit, vST-stp0 ).rgb;
  vec3 ip10  = texture( uImageUnit, vST+stp0 ).rgb;
  vec3 i0m1  = texture( uImageUnit, vST-st0p ).rgb;
  vec3 i0p1  = texture( uImageUnit, vST+st0p ).rgb;
```

```
    vec3 target = vec3(0.,0.,0.);
    target += 1.*(im1m1+ip1m1+ip1p1+im1p1); //apply blur filter
    target += 2.*(im10+ip10+i0m1+i0p1);
    target += 4.*(i00);
    target /= 16.;
    fFragColor = vec4( target, 1. );
}
```

Chromakey Images

Chromakey image manipulation is used in "green screen" or "blue screen" image replacement. This lets you take any image and replace any regions that have the same color as the key color or are very near the key color with a background texture or portions of another image. The chromakey replacement effect is shown in Figure 11.6.

For chromakey computation, two textures are required, an "image texture" and a "before texture." The "image texture" is the one that may contain pixels in the color key that would need to be replaced, and the "before texture" is the one that would replace any color-keyed pixels. The process then is relatively simple: read the image texture, and for each pixel, either keep this pixel, or if the pixel color is sufficiently near the key color, replace the pixel with the corresponding pixel in the before texture. The code fragment below uses pure green as the color key, simulating a green-screen process. The value of uT is a tolerance, or a measure of how near a color must be to the color key before its pixel will be replaced. This is typically very small, so that only colors very near green, `vec3(0., 1., 0.)`, will pass the limit test and will be replaced by the "before" texture color.

A fragment shader for this process is shown below, with uniform slider variables `uT` and `uAlpha` from a GLIB file. The foreground image comes from the `BeforeUnit` and the background image is from the `AfterUnit`. The `uAlpha` variable controls the alpha value for the foreground image as seen in the figure.

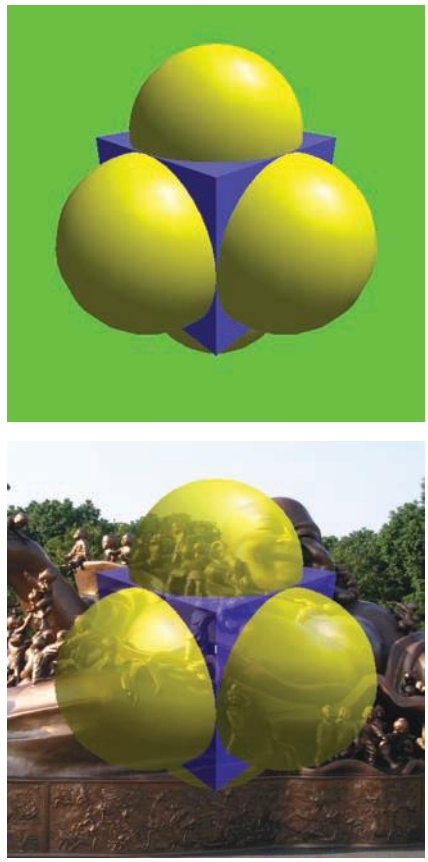

Figure 11.6. A synthetic image (top) and the result of green-screen chromakey processing to replace the green color and blend the foreground image with a background with an alpha value of 0.7 (bottom).

```
uniform float uT;
uniform float uAlpha;
uniform sampler2D uBeforeUnit, uAfterUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit, vST ).rgb;
  vec4 color;

  float r = brgb.r;
  float g = brgb.g;
  float b = brgb.b;
  color = vec4( brgb, 1. );
  float rlimit = uT;
  float glimit = 1. - uT;
  float blimit = uT;
  if( r <= rlimit && g >= glimit && b <= blimit )
    color = vec4( argb, 1. );
  else
    color = vec4( uAlpha*brgb + (1.-uAlpha)*argb, 1. );

  fFragColor = color;
}
```

Stereo Anaglyphs

A very interesting and fun use for image-based fragment shaders is to produce *stereo anaglyphs*. These have long been used in comic books and movies, and are still popular today. These are sometimes called "red-blue stereo," although today most glasses are actually red-cyan, with the convention that the red filter is over the left eye and the cyan filter is on the right.

Before writing the shader, we need to see how the glasses actually work. Our shader will produce a composite image that incorporates both the left and right eye views. When the composite image is viewed through the red filter, we want to see just the left eye image. The right eye image needs to be blocked, or in image terms, it needs to be blacked out. Similarly, when the composite image is viewed through the cyan filter, we want to see just the right eye image, so the left eye needs to be blocked. Since a red filter passes red light and blocks cyan light, this means that the left eye image needs to be coded in red

and the right eye image needs to be coded in cyan (i.e., greens and blues). The cyan filter on the right eye blocks red and passes green and blue, so the reverse needs to happen for the right eye. Thus, the final composite image needs to get its red component from the left eye image and its green and blue components from the right eye image.

An example of doing this is shown in Figure 11.7. You need to have some red-cyan glasses to see the effect.

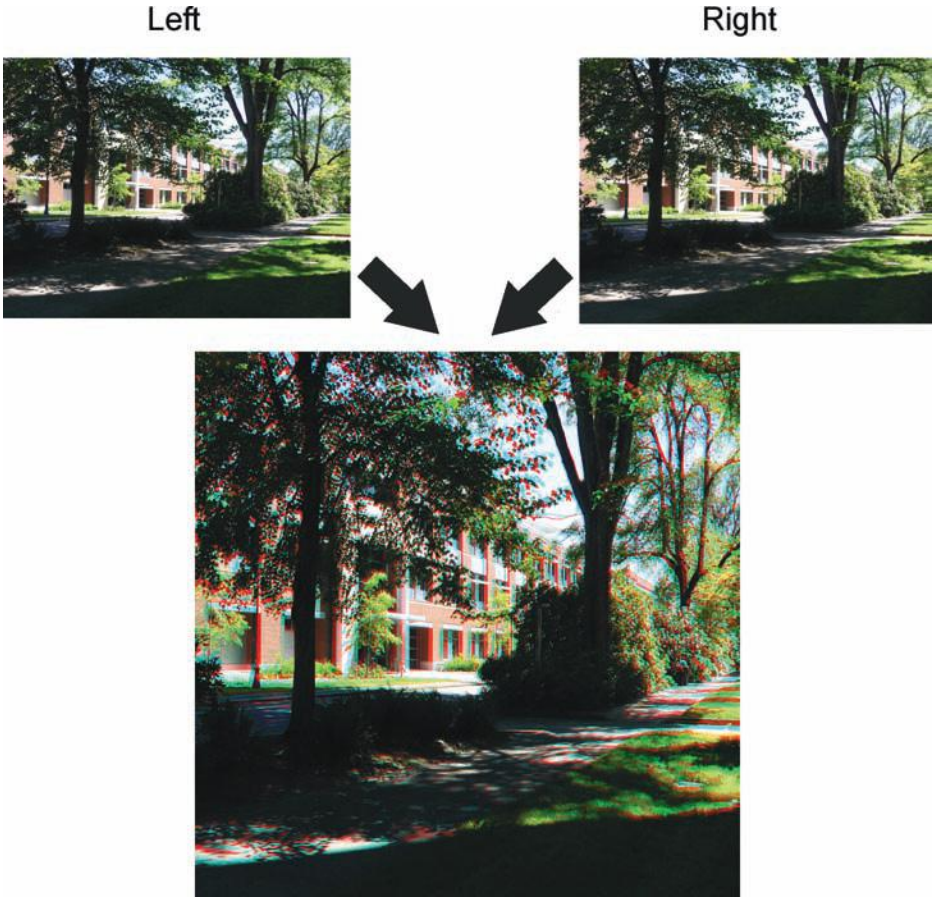

Figure 11.7. A pair of stereo images (top) and the composite anaglyph (bottom).

The strategy for creating anaglyph images, then, is this:

1. Start with left and right images of a 3D scene. These can be produced with a camera or by separately rendering two views of the same scene. Much of this is discussed in [27]. For photographs, take two pictures from points about 4 inches to 6 inches apart that frame the same area, preferably in the photos' foregrounds or middlegrounds.
2. Create a composite image using the red from the left eye image and the green and blue from the right eye image.
3. Because there is often vertical disparity between stereo images, especially when shot with a handheld camera, allow one image to be repositioned vertically.
4. Whatever objects appear in the same location in the two images will appear to live in the plane of the screen or paper. This depth is known as the plane of zero parallax. While it is cool to place the plane of zero parallax towards the back of the scene so that most of the scene seems to hover in midair, it is awkward. Here's why. In our everyday existence, things can appear in midair in front of us, a flying bird for example. Graphics scenes get clipped on the left, right, bottom, and top, but that midair bird doesn't. So, a graphics scene hovering in front of us has the potential to look rather unworldly when it gets clipped in midair for no apparent reason. A more natural-looking approach is to place the plane of zero parallax in the front, so that most of the 3D scene appears to live inside the monitor or book. If we were watching the midair bird through a window, and the bird suddenly got clipped against the window sides, we would think nothing of it. So a graphics scene that goes into the page and gets clipped will look like something we are used to seeing. So, in producing this anaglyph, it is also a good idea to allow one image to be repositioned horizontally to change the plane of zero parallax.
5. Because the red and cyan filters are not usually perfectly balanced, allow the color components to be scaled to compensate for any inequities.

The GLIB file needs to bring in both image files and set up the sliders:

```
##OpenGL GLIB

Texture 5 left.bmp
Texture 6 right.bmp

Vertex anaglyph.vert
```

```
Fragment anaglyph.frag
Program Anaglyph                    \
       uOffsetS <-.25 0. .25>       \
       uOffsetT <-.25 0. .25>       \
       uRed     <0. 1. 5.>          \
       uGreen   <0. 1. 5.>          \
       uBlue    <0. 1. 5.>          \
       uLeftUnit 5 uRightUnit 6

QuadXY .2 5.
```

Like the other examples in this chapter, most of the work is in the fragment shader:

```
uniform sampler2D uLeftUnit, uRightUnit;
uniform float uOffsetS, uOffsetT;
uniform float uRed, uGreen, uBlue;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec4 left = texture(uLeftUnit, vST );
  vec4 right = texture(uRightUnit,vST+vec2(uOffsetS,
                                           uOffsetT));

  vec3 color = vec3( left.r, right.gb );
  color *= vec3( uRed, uGreen, uBlue );
  color = clamp( color, 0., 1. );

  fFragColor = vec4( color, 1. );
}
```

Notice that the fragment shader uses five uniform slider variables that are set up in the GLIB file. The variables `uOffsetS` and `uOffsetT` control the offset in the right image, to make up for differences in registering the images, and the three uniform variables `uRed`, `uGreen`, and `uBlue` let you adjust the color balance to make up for variations in the colors in the glasses. When you create an anaglyph image, you may want to adjust the image with these variables to get the best effect.

Figure 11.8 shows another example of creating stereo anaglyphs.

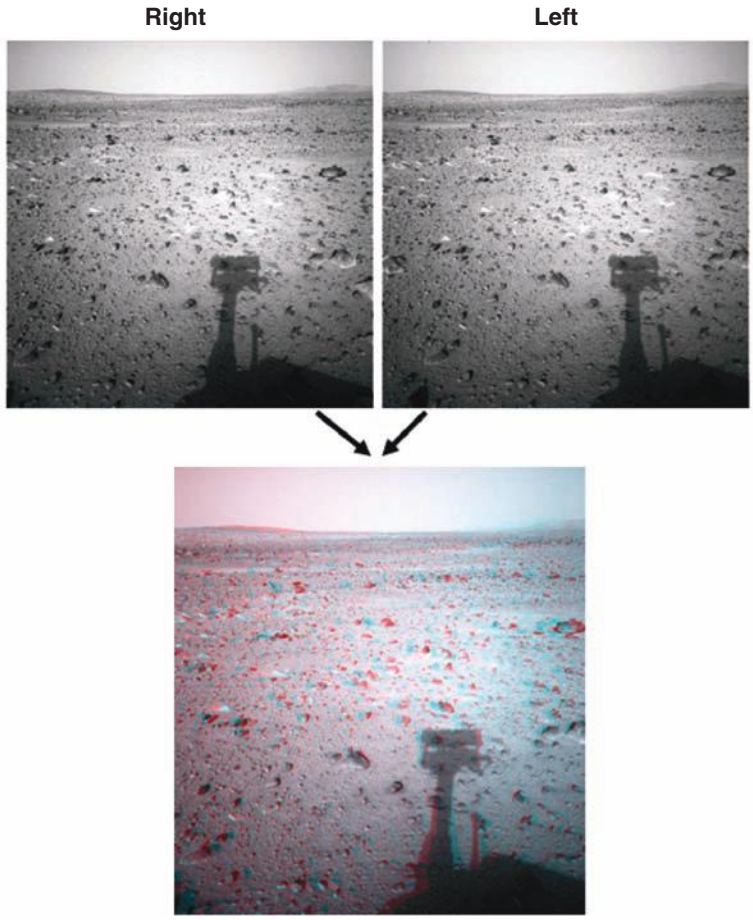

Figure 11.8. An anaglyph made from stereo pairs of images of Mars, from NASA's website [7]. Note that in the top of this figure, the left eye view is on the right and the right eye view is on the left. This makes it possible to free-view these images if you are good at crossing your eyes. If you are good at parallel free-viewing, try the top of Figure 11.7.

3D TV

While we are on the subject of stereographics, let's go one step farther. One proposal for 3D television ("3DTV") has been a technique called "SmoothPicture" [21] which transmits a single stereo image by spatially interlacing the left and right images into it, as shown in Figure 11.9. Each separate image is decimated in complementary checkerboard patterns before being combined. A

3D-enabled digital television decomposes the single image back into decimated left and right images, doubles its refresh rate to 120 Hz, and alternately displays the images: left-right-left-right-… Viewers wear shutterglass stereo eyewear to channel the proper image into the proper eye.

Fortunately, existing programs can be adapted fairly easily to produce this spatially-interlaced signal. A left eye and right eye view would need to be rendered, each to its own texture. A fragment shader, shown below, would then create the checkerboard interlace pattern. A simple way to do that would be to use the built-in `gl_FragCoord` window-relative pixel-space coordinates, to decide whether this fragment should receive the left eye image or the right.

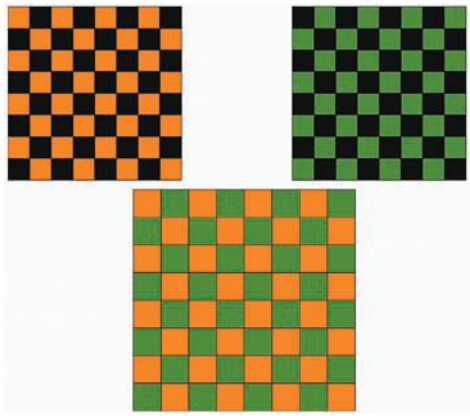

Figure 11.9. Left and right eye views being combined into a single spatially interlaced image.

```
uniform sampler2D uLeftUnit, uRightUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  int row = int( gl_FragCoord.y );
  int col = int( gl_FragCoord.x );
  int sum = row + col;

  vec4 color;
  if( ( sum % 2 ) == 0 )
    color = texture( uLeftUnit, vST );
  else
    color = texture( uRightUnit, vST );

  fFragColor = vec4( color.rgb, 1. );
}
```

Here is an example of what this looks like. Figure 11.10 shows left and right eye images. (In this case they were taken with a stereo camera, but they could just have easily been computer-generated.) The spatially interlaced image is also shown as part of Figure 11.10, along with a zoomed-in view more clearly showing the checkerboard interlacing pattern.

Figure 11.10. Left and right eye images, top, and the spatially interlaced result and a zoom-in view, bottom.

Shutterglass stereo , in which each eye sees an image separately through polarized lenses, has been a visualization mainstay for many years, finding important applications in architecture, biology, chemistry, computer-aided design, geology, etc. In addition to the obvious entertainment applications, 3DTV should become an important tool for science and engineering.

Edge Detection

Edge detection is a classic image processing technique, and is relatively easy to do in a fragment shader. The edge detection process we present uses a pair of Sobel filters, one for horizontal components and one for vertical components. The horizontal Sobel filter was shown above. The vertical Sobel filter is the same, but rotated 90 degrees. Specifically, the horizontal and vertical filters are, respectively,

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}.$$

The effect of the Sobel filters is to compare two columns (or rows, depending on which filter you are using) that are one column (or row) apart; if there is no edge, the colors should be quite close and the filter should return a very small value. If the returned value or values are "large", the process infers that an edge is present. The test may be done on the original image or the luminance-only image.

Notice that in the rightmost image of Figure 11.11, the filter results are interpreted as colors. Where there is no edge, the output figure is very dark;

Figure 11.11. The edge detection operation, with the edge-showing image combined with the original image in proportions 0.0 (left), 0.5 (middle), and 1.0 (right).

where there is an edge, the output color is light. This visually validates the concept of detecting edges, though in many applications you would go on to make processing decisions based on these edges rather than simply displaying them.

Below, you see an example of some fragment shader code that implements these ideas. The colors of the 3×3 set of pixels are retrieved from the image texture, a dot product of each is done with the luminance weight vector to convert the 3×3 image to grayscale, and then the Sobel filters are applied and the two results combined to set a single grayscale output value. Finally, that output value is mixed with the original color according to a *glman* uniform slider variable, *uT*.

```
ivec2 ires = textureSize( uImageUnit, 0 );
float ResS = float( ires.s );
float ResT = float( ires.t );
vec3 irgb  = texture( uImageUnit, vST ).rgb;

vec2 stp0 = vec2(1./ResS, 0. );
vec2 st0p = vec2(0.   , 1./ResT);
vec2 stpp = vec2(1./ResS, 1./ResT);
vec2 stpm = vec2(1./ResS, -1./ResT);

const vec3 W  = vec3( 0.2125, 0.7154, 0.0721 );
float i00    = dot( texture( uImageUnit, vST ).rgb, W );
float im1m1  = dot( texture( uImageUnit, vST-stpp ).rgb, W );
float ip1p1  = dot( texture( uImageUnit, vST+stpp ).rgb, W );
float im1p1  = dot( texture( uImageUnit, vST-stpm ).rgb, W );
float ip1m1  = dot( texture( uImageUnit, vST+stpm ).rgb, W );
float im10   = dot( texture( uImageUnit, vST-stp0 ).rgb, W );
float ip10   = dot( texture( uImageUnit, vST+stp0 ).rgb, W );
float i0m1   = dot( texture( uImageUnit, vST-st0p ).rgb, W );
float i0p1   = dot( texture( uImageUnit, vST+st0p ).rgb, W );
float h= -1.*im1p1-2.*i0p1-1.*ip1p1+1.*im1m1+2.*i0m1+1.*ip1m1;
float v= -1.*im1m1-2.*im10-1.*im1p1+1.*ip1m1+2.*ip10+1.*ip1p1;

float mag   = length( vec2( h, v ) );
vec3 target  = vec3( mag, mag, mag );
fFragColor = vec4( mix( irgb, target, uT ), 1. );
```

Embossing

We can modify the idea of edge detection to include replacing color by luminance and highlighting images differently depending on the edges' angles. The result is the emboss operation that is commonly found in image manipulation programs. The result of an emboss operation is shown in Figure 11.12, and the code for a fragment shader to accomplish this is shown below. This

Figure 11.12. An original photo (left) along with the emboss operation results (right).

code includes #define statements to create grayscale or color embossing; both are shown in the figure.

```
#define GRAY

uniform sampler2D uImageUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  ivec2 ires = textureSize( uImageUnit, 0 );
  float ResS = float( ires.s );
  float ResT = float( ires.t );
  vec3 irgb  = texture( uImageUnit, vST ).rgb;

  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;  // vector difference
  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. );
  vec3 color = vec3( gray, gray, gray );
  fFragColor = vec4( color, 1. );
}
```

Toon Shader

There are various kinds of shader that are known as *toon shaders*. One is a shader for 3D graphics, in which the colors are quantized, and the edges are enhanced by coloring them black. This is the "toon shader" used by many commercial 3D graphics packages, so named because the resulting images look like a hand-drawn cartoon.

Figure 11.13. The original fruit image (top) and with toon shading applied (bottom).

This shader operates in a relatively simple fashion and uses the edge-detection filtering discussed above and some color quantization. This models both kinds of enhancement seen in the 3D toon shader. At a high level, the 2D toon shader's operations are

1. Calculate the luminance of each pixel.
2. Apply the Sobel edge-detection filter and get a magnitude.
3. If magnitude > threshold, color the pixel black
4. Else, quantize the pixel's color.
5. Output the colored pixel.

This is shown in the following fragment shader, which is set up with uniform slider variables `MagTol` and `Quantize` to manipulate the image through *glman*. Notice that this gets the nine texture values needed for a 3×3 filter, converts each to its saturation value, and then applies both horizontal and vertical Sobel filters and tests their combination for edges The color is then quantized to simulate the behavior of hand-drawn cartoons.

```
uniform sampler2D uImageUnit, uBeforeUnit, uAfterUnit;

uniform float uMagTol;
uniform float uQuantize;

in vec2 vST;

out vec4 fFragColor;
```

```
void main( )
{
  ivec2 ires = textureSize( uImageUnit, 0 );
  float ResS = float( ires.s );
  float ResT = float( ires.t );

  vec3 irgb = texture( uImageUnit, vST ).rgb;
  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit, vST ).rgb;

  vec3 rgb = texture( uImageUnit, vST ).rgb;
  vec2 stp0 = vec2(1./uResS, 0. );
  vec2 st0p = vec2(0.   , 1./uResT);
  vec2 stpp = vec2(1./uResS, 1./uResT);
  vec2 stpm = vec2(1./uResS, -1./uResT);

  const vec3 W = vec3( 0.2125, 0.7154, 0.0721 );
  float i00  = dot( texture( uImageUnit, vST).rgb, W );
  float im1m1= dot( texture( uImageUnit, vST-stpp ).rgb, W );
  float ip1p1= dot( texture( uImageUnit, vST+stpp ).rgb, W );
  float im1p1= dot( texture( uImageUnit, vST-stpm ).rgb, W );
  float ip1m1= dot( texture( uImageUnit, vST+stpm ).rgb, W );
  float im10 = dot( texture( uImageUnit, vST-stp0 ).rgb, W );
  float ip10 = dot( texture( uImageUnit, vST+stp0 ).rgb, W );
  float i0m1 = dot( texture( uImageUnit, vST-st0p ).rgb, W );
  float i0p1 = dot( texture( uImageUnit, vST+st0p ).rgb, W );

  // next two lines apply the H and V Sobel filters at the pixel
  float h= -1.*im1p1-2.*i0p1-1.*ip1p1+1.*im1m1+2.*i0m1+1.*ip1m1;
  float v= -1.*im1m1-2.*im10-1.*im1p1+1.*ip1m1+2.*ip10+1.*ip1p1;
  float mag = length( vec2( h, v ) );// how much change
                                     // is there?
  if( mag > uMagTol )
  { // if too much, use black
    fFragColor = vec4( 0., 0., 0., 1. );
  }
  else
  {               // else quantize the color
    rgb.rgb *= uQuantize;
    rgb.rgb += vec3( .5, .5, .5 );      // round
    ivec3 intrgb = ivec3( rgb.rgb );    // truncate
    rgb.rgb = vec3( intrgb ) / Quantize;
    fFragColor = vec4( rgb, 1. );
  }
}
```

Artistic Effects

If you use a commercial image manipulation program such as Photoshop or a general-distribution version such as GIMP, you will find a number of "artistic filters" that you can use to make an image look more like a painting or as if any of several other kinds of thing had been done to it. It is interesting to consider how you might be able to create such artistic effects with a GLSL fragment shader.

A general approach might be to select a region of several texels from the image relative to the current texel, and apply some sort of process that results in a single color value. For example, you might choose the texel in the region that has the greatest luminance. In Figure 11.14 we have done that to create a painting effect for the familiar cherry blossom figure of this chapter. The process is fairly straightforward; we develop a full 5×5 texel rectangle R around an individual pixel, as well as a 5×5 mask rectangle M whose values are simply zero or one. We then look at the luminance values of each texel in the set $R * M$ and use the texel with the highest luminance value in the place of the particular pixel. The code is rather long because, at this writing, GLSL does not allow variables to be used as array indices, so we will not include it here. It is available with the resources for the book.

It is straightforward to choose which values are zero and which are one in the mask, and changing the "shape" of the mask will change the effect of the filter. You can use other criteria besides the maximum luminance to select the color for the pixel. There is ample ground here for fruitful experimentation!

Figure 11.14. An image (left) with a painting-effect filter applied (right).

Image Flipping, Rotation, and Warping

In the previous examples of single-image manipulation, we worked with each pixel in place. However, we can also compute the color for a pixel by manipulating the coordinates of the pixel to get that pixel's color from another place in the image.

While we create image warps to achieve particular effects on specific images, it can be useful to have some kind of uniform benchmark image for warps. There are many such benchmark images, depending on just what effects you want to see, but in Figure 11.15 we see a simple rectangular grid that we will use to examine the details of changes in the images.

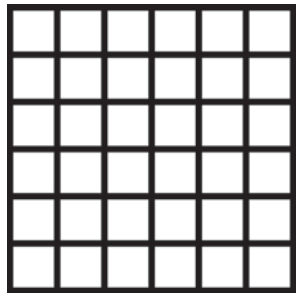

Figure 11.15. The rectangular grid image.

We will work with images by treating them as texture maps, as we have throughout this chapter so far. When we compute the source address for a pixel, we are applying a function from the pixel address space to itself. Since the texture space is the unit square, $[0, 1] \times [0, 1]$ we are looking for functions that map that space to itself. Sometimes, however, we may want to double the size of that space and shift it so that the space we are working with is $[-1, 1] \times [-1, 1]$ to make it convenient to apply familiar functions (such as trigonometric functions) to the space. Examples and exercises will help clarify what we mean by this.

One of the simplest kinds of address-based image manipulation is *image flipping*. There are two kinds of flipping: horizontal and vertical. In vertical image flipping, you exchange the top pixels in the image with the bottom pixels, effectively mirroring the image around a horizontal line. In horizontal image flipping, you exchange the left and right pixels in the image, effectively mirroring it around a vertical line.

You can flip an image by a very simple calculation on the texture coordinates. Since the texture coordinates are in the interval [0, 1], the function $t = 1 - t$ will reverse the order of the coordinate t in this interval. If this is applied to the texture coordinates in the fragment shader (with the common *glman* setup) as

```
vec2 st = vST;
st.t = 1. - st.t;
vec3 irgb = texture( uImageUnit, st ).rgb;
fFragColor = vec4( irgb, 1. );
```

then the resulting image will be displayed "upside down" or flipped vertically. It is quite easy to see how a horizontal flip could be implemented by manipulating the s texture coordinate.

Simple image rotation (that is, rotation through a multiple of 90 degrees) can be done similarly. If you want to rotate an image by 90 degrees counterclockwise, for example, you can simply replace the s-coordinate of the texture by the original texture t-coordinate, and the t-coordinate of the texture by one minus the s-coordinate. (See if you can quickly figure out why the "one minus" is needed.) In terms of functions of two variables, the function $f(st) = (t, 1 - s)$ captures this operation. The other simple rotations are similarly easy. More general rotations are straightforward applications of the usual graphics rotation operations, but are complicated by the need to preserve the rectangular form factor in the domain and are thus not considered here.

Filling a pixel with a pixel from somewhere else in the image is more interesting. You can apply any function or procedure that you like to manipulate the address of any particular pixel, so long as it stays within the unit

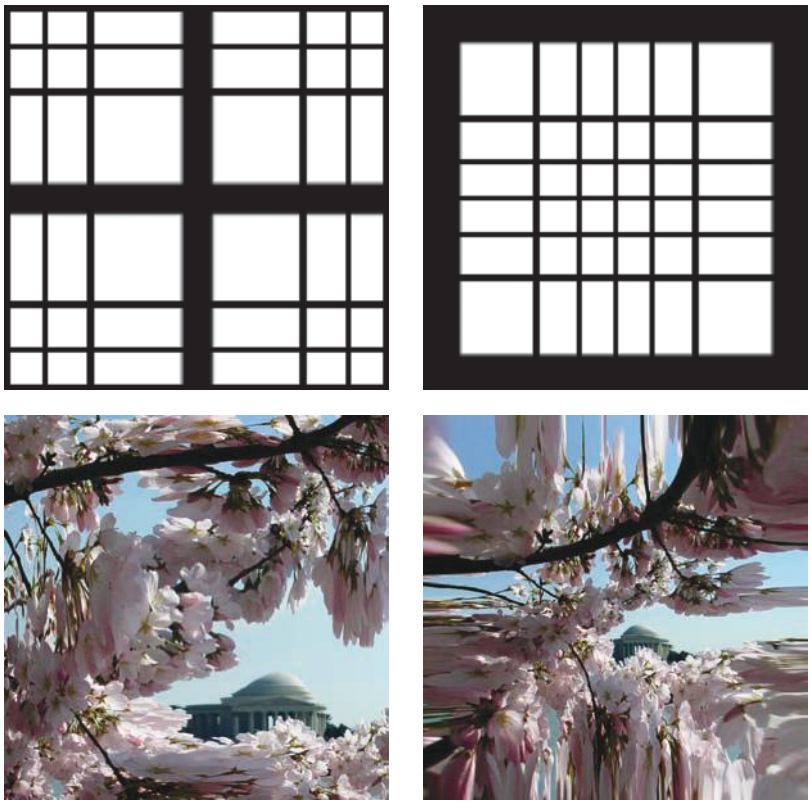

Figure 11.16. The grid (above) and cherry blossom image (below), manipulated to magnify (left) or compress (right) the center part of the image.

square of the pixel space. The process of manipulating the image by applying a function to the pixel address space is called *image warping* [46] and has many potential uses.

In most image warping applications, the effect of the function can vary quite a bit if different parameter values are used in the function. Fortunately, *glman* is easy to set up so you can create uniform slider variables for these parameters. For example, if we consider the image warping with the function $x = x + t * \sin(\pi * x)$ applied to both coordinates of the texel, we see in Figure 11.16 the effect of two different values of the parameter t for this warping on both the grid above and on the cherry blossom image.

The fragment shader that defines this effect is shown below.

```
const float PI = 3.14159265;

uniform sampler2D uImageUnit;
uniform float uT;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec2 st = vST;
  vec2 xy = st;
  xy = 2. * xy - 1.;         // map to [-1,1] square
  xy += uT * sin(PI*xy);
  st = (xy + 1.)/2.;       // map back to [0,1] square
  vec3 irgb = texture( uImageUnit, st ).rgb;
  fFragColor = vec4( irgb, 1. );
}
```

Other kinds of image warping apply more complex kinds of operations to pixel coordinates. The *twirl transformation* is one example, and others are explored in the exercises. For the twirl transformation, we work in pixel coordinates, so we start by transforming texture coordinates to pixel coordinates, apply the twirl transformation, and then come back to texture coordinates to select the actual pixel colors.

The twirl transformation rotates the image around a given anchor point (x_c, y_c) by an angle that varies across the space from a value α at the center, decreasing linearly with the radial distance as it proceeds toward a limiting radius r_{max}. The image remains unchanged outside the radius r_{max}. The notation has (x', y') as the original pixel coordinates and (x, y) as the coordinates

of the pixel whose color you use; look for this in the shader code. The inverse mapping function for this transformation is given by

$$T_x^{-1} : x = \begin{cases} x_c + r\cos(\beta) & \text{for } r \le r_{\max} \\ x' & \text{for } r > r_{\max} \end{cases}$$

$$T_y^{-1} : y = \begin{cases} y_c + r\sin(\beta) & \text{for } r \le r_{\max} \\ y' & \text{for } r > r_{\max} \end{cases}$$

with

$$d_x = x' - x_c \qquad r = \sqrt{d_x^2 + d_y^2}$$

$$d_y = y' - y_c \qquad \beta = \arctan_2(d_y, d_x) + \alpha\left(\frac{r_{\max} - r}{r_{\max}}\right)$$

The resulting image effect is shown in Figure 11.17 for the rectangular grid and for the cherry blossom image.

In the twirl transformation fragment shader below, look for the changes to and from pixel coordinates, and note the two parameters (angle α and limiting radius $r_{\max}$) that are set up as uniform variables, so they can be defined as *glman* uniform slider variables.

```
const float PI = 3.14159265;

uniform sampler2D uImageUnit;
uniform float uD, uR;

in vec2 vST;
```

Figure 11.17. The twirl transformation on the grid (left) and on the cherry blossom image (right).

```
out vec4 fFragColor;

void main( )
{
  ivec2 ires = textureSize( uImageUnit, 0 );
  float Res = float( ires.s ); // assume it's a square
                               // texture image
  vec2 st = vST;
  float Radius = Res * uR;
  vec2 xy = Res * st;          // pixel coordinates from
                               // texture coords

  vec2 dxy = xy - Res/2.; // twirl center is (Res/2, Res/2)
  float r  = length( dxy );
  float beta = atan(dxy.y,dxy.x) + radians(uD)*
                                   (Radius-r)/Radius;

  vec2 xy1 = xy;
  if (r <= Radius)
  {
    xy1 = Res/2. + r * vec2( cos(beta), sin(beta) );
  }
  st = xy1/Res;  // restore coordinates

  vec3 irgb = texture( uImageUnit, st ).rgb;
  fFragColor = vec4( irgb, 1. );
}
```

Image warping need not be uniform, of course, and you can readily use noise functions, as described in the previous chapter, to modify the address of a source pixel in an image. Some code for a fragment shader to do this is below, and the result is shown in Figure 11.18.

Figure 11.18. The grid (left) and cherry blossom image (right) with noise as pixel offset.

```
uniform sampler2D uImageUnit;
uniform float uT;
uniform sampler3D Noise3;

in vec3 vMCposition;
in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec2 st = vST;
  float x = st.x;
  float y = st.y; // extract coordinates
  vec4 noisevecx = texture( Noise3, vMCposition );
  vec4 noisevecy = texture( Noise3,
                            vMCposition+vec3(noisevecx) );
  x += uT*(noisevecx[.r]-noisevecx[.g]+noisevecx[.b]
           +noisevecx[.a]-1.);
  y += uT*(noisevecy[.r]-noisevecy[.g]+noisevecy[.b]
           +noisevecy[.a]+1.);
  st = vec2( x, y );  // restore coordinates
  vec3 irgb = texture( uImageUnit, st ).rgb;
  fFragColor = vec4( irgb, 1. );
}
```

You can also see *image morphing*, the transition over time from one image to another, as a combination of image warping and image blending. The image blending part of this is discussed in the sections below, while the image warping for morphing is a very specialized process where a fixed set of points on one image are mapped to a fixed set in another, and the image blending is parameterized so that at the beginning, the geometry of one image is fixed, and at the end, the geometry of the second image is achieved. This is well beyond the scope of our discussion here, however; you can read more in [46].

The Image Blending Process

There are several kinds of image manipulation in which you create a linear combination of the pixels from one image with those from a constant or from another image. This kind of combination is shown in Figure 11.19. In the next few sections, the base value will be a constant color or a value derived directly from each pixel, so only the image being manipulated is used. Later in this

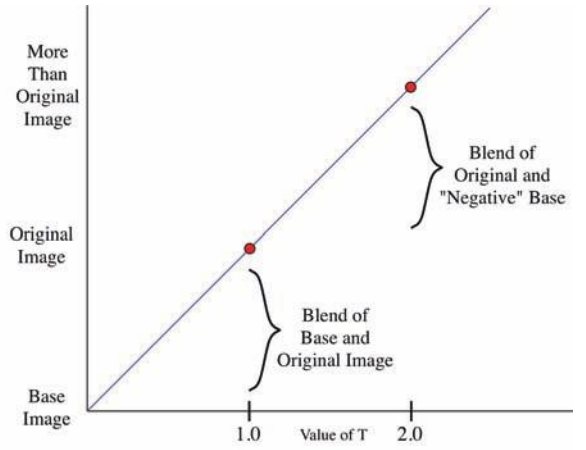

Figure 11.19. The meaning of the parameter T in the blending process.

chapter, we will present examples where one of these images is a particular base image, and the other is an image that you want to manipulate. The general form of the linear combination is

$$I_{\text{out}} = (1. - T) * I_{\text{base}} + T * I_{\text{source}}.$$

We are used to equations such as this being limited by having the parameter T restricted to the range [0., 1.]. However, for some of these applications, we don't make any such limitation, because for some effects it is easier to ask for what you don't want than to ask for what you do. Going outside the [0., 1.] range will allow us to extrapolate to the effect we want to achieve.

The parameter in the blend can be varied to get different results, and here *glman*'s ability to attach a uniform variable to a slider can be very helpful in experimenting with the effects of a parameter. The built-in `GLSL mix( )` function supports the actual blending sum.

Blending an Image with a Constant Base Image

There are several image-manipulation processes that involve blending each pixel of an image with a constant value. The operations that result are quite common and are very useful. Many of the examples below have been set up for the *glman* environment with a uniform slider variable T that performs the blending operation shown in Figure 11.19.

Color Negative

The *color negative* models the way photographic negatives work. A photographic negative blocks the complement of a color from getting to photographic paper, so the negative of an image is computed by subtracting the color of each pixel from white:

```
vec3(1.0, 1.0, 1.0) - color.rgb
```

If you use that negative as the base image, you get an image that looks just like the photographic negative, as shown in Figure 11.20.

The following code for the negative fragment shader sets up a color and its negative so you can blend between them with the variable uT. At `uT = 0` you have the original image, and at `uT = 1` you have the negative.

```
uniform sampler2D uImageUnit;
uniform float uT;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec3 irgb = texture( uImageUnit, vST ).rgb;
  vec3 neg = vec3(1.,1.,1.) - irgb;
  fFragColor = vec4( mix( irgb, neg, uT ), 1. );
}
```

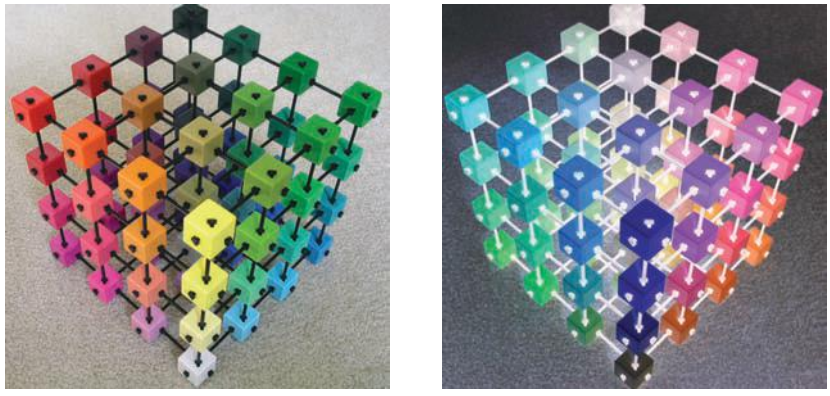

Figure 11.20. An image (left) and its color negative (right).

Brightness

Informally, brightness can be thought of as the amount of not-black in a color. For RGB color, "less" black means that the color components are nearer to 1.0, and "more" black means that the components are nearer to 0.0. To manipulate the brightness of an image, use a black image with color

```
target = vec3(0.0, 0.0, 0.0)
```

as the base. Values of uT less than 1.0 will darken each component of the color, while values greater than 1.0 will brighten each component of the color up to the point where the color is clamped. This can, of course, wash out colors if the colors are already bright or if you use uT too large. This is shown in Figure 11.21.

Sample code for a very simple fragment shader that adjusts brightness is shown below. In effect, brightening the image is done by subtracting black from it.

```
uniform sampler2D uImageUnit;
uniform float uT;
in vec2 vST;
out vec4 fFragColor;

void main( )
{
  vec3 irgb  = texture( uImageUnit, vST ).rgb;
  vec3 black  = vec3( 0., 0., 0. );
  fFragColor = vec4( mix( black, irgb, uT ), 1. );
}
```

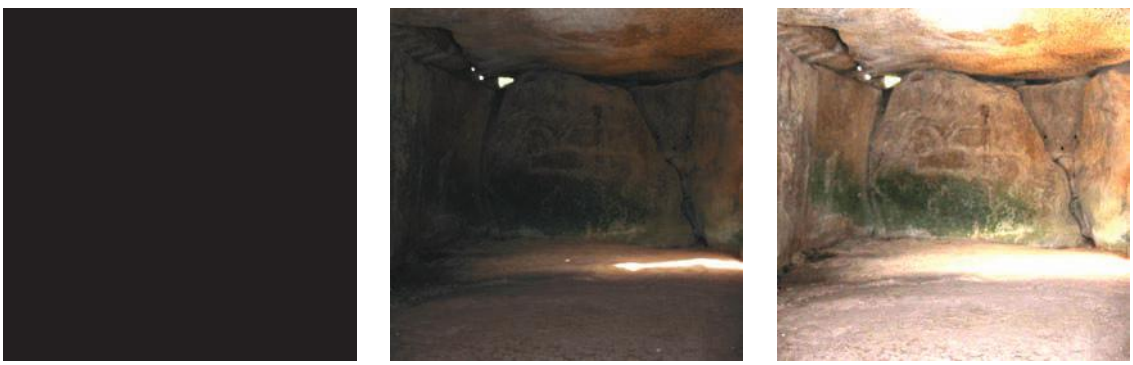

Figure 11.21. Brightness manipulation in a photograph from a prehistoric French tomb with uT = 0.0 (left), 1.0 (middle), and 2.0 (right).

Contrast

The contrast in an image describes how much the colors stand out from gray. To manipulate the contrast in an image, use as a base image a constant 50% gray image, which is easily computed as

```
target = vec3(0.5,0.5,0.5);
```

Parameter values of T less than 1 will move each color component toward 0.5, reducing the contrast in the image, while values greater than 1 will move each color component away from 0.5, increasing the contrast, as shown in Figure 11.22.

Sample code for a very simple fragment shader that adjusts either brightness or contrast is shown below. In effect, brightening the image is done by subtracting black from it, and contrast is increased by subtracting 50% gray from it.

```
#define BRIGHTNESS
#undef CONTRAST

uniform sampler2D uImageUnit;
uniform float uT;

in vec2 vST;

our vec4 fFragColor;

void main( )
{
  vec3 irgb = texture( uImageUnit, vST ).rgb;
```

Figure 11.22. Contrast manipulation in a photograph of a ruined French abbey with $T = 0.0$ (left), 1.0 (middle), and 2.5 (right).

```
#ifdef BRIGHTNESS
  vec3 target = vec3( 0., 0., 0. );
#else
  vec3 target = vec3( 0.5,0.5,0.5 );
#endif
  fFragColor = vec4( mix( target, irgb, uT ), 1. );
}
```

Blending an Image with a Version of Itself

Another common kind of image manipulation involves creating a base value that is computed from the image itself. This might be a grayscale image or a blurred image in the examples below. Again, these are common and very useful kinds of manipulation. And again, we show examples that have been set up for *glman* as described above.

Saturation

We think of *color saturation* as a description of the "purity" of the color, or how far the color is from gray. This is consistent with the notion of saturation in the HLS color system, where saturation is the distance from the pure grays that are at the center of the HLS double cone. If saturation is reduced, the color is more gray; if it is increased, the color is purer and more vivid.

To manipulate the saturation of an image, you create a grayscale base image by replacing the color at each point by its luminance that we defined earlier in this chapter:

```
target = vec3( luminance, luminance, luminance);
```

and mix the color with this target, as we have seen. Values of uT less than 1 will move each color component toward its luminance, making the color less saturated, while values greater than 1 will move each color component away from the luminance, making it more saturated, as shown in Figure 11.23.

A simple fragment shader to manipulate saturation is shown below.

```
const vec3 W = vec3( 0.2125, 0.7154, 0.0721 );

uniform sampler2D uImageUnit;
uniform float uT;

in vec2 vST;
```

Figure 11.23. Saturation manipulation of the supermarket fruit image with $uT = 0.0$ (left), 1.0 (middle), and 2.0 (right).

```
out vec4 fFragColor;

void main( )
{
  vec3 irgb = texture( uImageUnit, vST ).rgb;
  float luminance = dot( irgb, W );
  vec3 target = vec3( luminance, luminance, luminance );
  fFragColor = vec4( mix( target, irgb, uT ), 1. );

}
```

Sharpness

We think of sharpness as the degree of clarity in both coarse and fine image detail in an image. Alternately, you could think of sharpness as the opposite of blurred. Manipulating the sharpness of the image takes advantage of this fact by creating an extrapolation from a blurred version of the image through the image itself. The blurred image is created by the blurring process discussed earlier in the chapter. An example of sharpening an image is shown in Figure 11.24; the left and middle images are larger versions of those of Figure 11.5.

A fragment shader to manipulate sharpness would contain code something like the following. The code uses the same filter and computation described in the image blur example earlier in the chapter, except that it ends by mixing the blurred image ("target") and the original image ("irgb"). The shader files are included in the materials with this book.

```
...
fFragColor = vec4( mix( target, irgb, uT ), 1. );
```

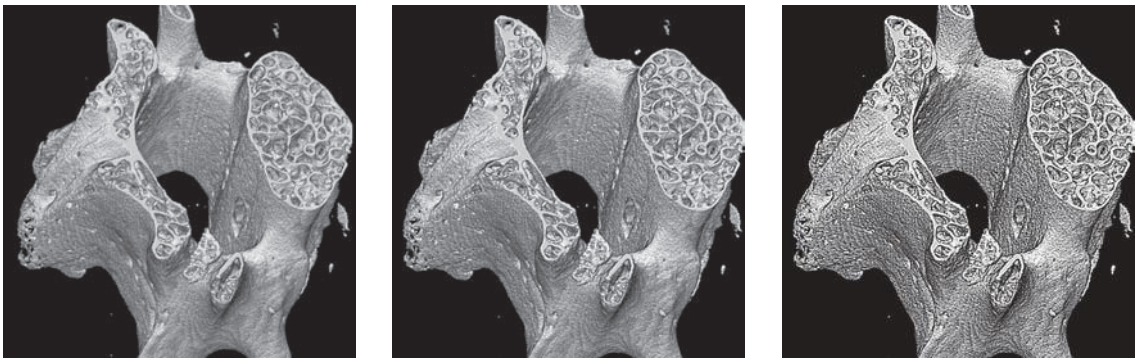

Figure 11.24. The result of the sharpness operation with the 5×5 blurred image ($T = 0$, left), the original image ($T = 1$, middle), and the sharpened image ($T = 5$, right).

This is a special case of unsharp masking, a standard image manipulation technique to sharpen photographic images for printing. The general technique uses a blur filter of adjustable radius and an adjustable blend; this example uses radius 1 and a limited adjustable blend.

Blending Two Different Images

The two-image manipulations in the sections above have really been about creating effects in a single image, using another reference image as a tool. However, sometimes you have two images that each have content, and you

Figure 11.25. Two sample images we will use to illustrate blending: Washington cherry blossoms (left) and Xidi, an ancient Hong village in Anhui province, China (right).

want to blend both images. There are a number of different common kinds of blends. In the sections below, we will sketch a few of them and show examples. It should be straightforward for you to complete any implementations that we do not give completely. In addition, we have included a few more blends as chapter exercises. Figure 11.25 shows two sample images that we will use to illustrate many of the blending operations we discuss.

Other Combinations

Complex and interesting interpolations of two images are possible because you can use any function that takes two RGB color values and returns another RGB color value. The function could act on entire RGB vectors or it could act on the individual color components separately. We explore a few of these below, and there are a few more in the exercises.

Cosine Interpolation

As an example, consider a cosine-based interpolation from [20] that looks interesting; Figure 11.26 shows the effect. The same pixel from both images is read, and the color components of the two pixels are combined, using cosine multipliers. The cosine is applied to each component, so components nearer one are increased. If we take *Argb* as the color of the "after" image and *Brgb* as the color of the "before" image, as above, then the blended color is given by

Figure 11.26. The cosine interpolation of the two sample images.

$$color = \rho - \alpha * \cos(\pi * Argb) - \beta * \cos(\pi * Brgb)$$

where ρ is a base color, basically an overall luminance, and α and β are chosen to weight the two images (and either $|\rho| + |\alpha| + |\beta|$ cannot exceed 1 or you must clamp the result).

Sample fragment shader code for this operation is given below. Notice that we have used values of 0.5 and –0.25 as the base value and cosine multiplier, respectively; in an exercise, we encourage you to experiment with these (and we suggest that you use *glman* uniform slider variables to do so).

```
const float PI = 3.14159265;

uniform sampler2D uBeforeUnit, uAfterUnit;

in vec2 vST;
```

```
out vec4 fFragColor;

void main( )
{
  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit,vST).rgb;
  vec3 target = 0.5 - 0.25*cos(PI*brgb) - 0.25*cos(PI*argb);
  fFragColor = vec4( target, 1. );
}
```

Multiply

The *multiply* operation does exactly as the name suggests. You read a pixel from each image and multiply the color components together to get the final color of the pixel. In this way, one image is being used as a subtractive filter for the other.

Since all the color components are less than or equal to one, the final image will likely be darker than either original. In order to account for that, you can balance the colors by computing the luminance of the original colors, *argb*, *brgb*, and *target*, and adjusting the final output color of each pixel so its luminance is the average of the two input pixels' colors. Some sample fragment shader code for this is shown below. The result, both without and with the color balancing, is shown in Figure 11.27.

```
const vec3 W = vec3(0.2125, 0.7154, 0.0721)

uniform sampler2D uBeforeUnit, uAfterUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit, vST ).rgb;
  vec3 target = argb * brgb;

  float alum = dot( argb, W );
  float blum = dot( brgb, W );
  float tlum = dot( target, W );
  target = (alum + blum)/(2.*tlum);
  fFragColor = vec4( target, 1.);
}
```

Figure 11.27. The results of the multiply without the color balancing (left) and with the color balancing (right) operations on our sample images.

Darken and Lighten

The *darken* and *lighten* operations are very similar, so we discuss them together. The darken operation on two images uses one image to darken the other. You read a pixel from each image, and you take the smaller of the values of each color component for each pixel. Some sample fragment shader code for this is shown below.

The lighten operation is the converse of the darken operation above; you read a pixel from each image, and you take the larger of the values for each color component for each pixel. The fragment shader code for this is left as an exercise. The result for both operations is shown in Figure 11.28.

```
uniform sampler2D uBeforeUnit, uAfterUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit, vST ).rgb;
  vec3 target = min( argb, brgb ); // alternately max(...)
  fFragColor = vec4( target, 1.);
}
```

Figure 11.28. The result of the darken (left) and lighten (right) operations on our sample images.

Image Transitions

In addition to combining two images into one, we should think about ways to move from one image to another over time. One example of this is the set of slide transitions in Powerpoint, but the control we have with fragment shaders lets us go well beyond the options available there.

The basic principle is that we start with each pixel from one image, which we will call the *Before* image, and we manipulate each pixel in a way that finishes with a second image, which we will call the *After* image. We can replace *Before* pixels with *After* pixels in any way we like, and we will try to create some interesting effects in doing so. In all our examples in this section, we start with the two images of Figure 11.25, the Washington cherry blossoms and the Hong village.

Horizontal Replace

The first transition we will consider moves the *Before* image off the display to the right while simultaneously moving the *After* image onto the display from the left. However, as we go through the transition, both images are displayed in their entirety; each is simply compressed into the part of the display that is available to it. An example of the transi-tion partly completed is shown in Figure 11.29.

The .glib file and vertex shader source are essentially identical to the image blending examples above, so we will focus on the fragment shader source, shown below.

```
uniform float uT; //0. <= uT <= 1.
uniform sampler2D uBeforeUnit, uAfterUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec2 st = vST;
  vec3 brgb = texture( uBeforeUnit, st ).rgb;
  vec3 argb = texture( uAfterUnit, st ).rgb;
  vec3 color;

  if ( st.x < uT )
  {
    st = vec2( st.x/uT, st.y );
    vec3 thisrgb = texture( AfterUnit, st ).rgb;
    color = thisrgb;
  }
  else
  {
    st = vec2( (st.x-uT)/(1.-uT), st.y );
    vec3 thatrgb = texture( BeforeUnit, st ).rgb;
    color = thatrgb;
  }

  fFragColor = vec4( color, 1.);
}
```

Figure 11.29. The Hong village image replacing the cherry blossom image.

Here the two halves of the if statement represent the two halves of the display: the side where the *s*-component of the texture coordinate is less than `uT` and the side where it is greater than `uT`. For each pixel coordinate, the *s*-component of the appropriate image (i.e., texture) is calculated by a proportional computation, and the resulting texture coordinate is used to select the texel to be displayed.

As `uT` goes from 0. to 1., the effect in this example is to create the transition from the *Before* image to the *After* image over that same period. No static figure can capture the full effect; an exercise invites you to create your own transition and see it work.

Dissolve

The *image dissolve* operation computes a weighted average of the *Before* and *After* images that determines how much of each image's color is used in the output image. This weight can be given by a parameter that changes over time, giving the effect of moving from one image to another, as can be done for slideshows. This is shown in Figure 11.30 and in the weighted-average fragment shader code below. As the value of `uT` ranges from 0. to 1., the *Before* image dissolves into the *After* image.

Figure 11.30. A dissolve of the two sample images with `uT` = 0.5.

```
uniform sampler2D uBeforeUnit, uAfterUnit;
uniform float uT;

in vec2 vST;

out vec4 fFragColor;

void main( )
{

  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit, vST ).rgb;
  fFragColor = vec4( mix( argb, brgb, uT ), 1. );
}
```

Burn-Through

Another transition can be made where the *After* image "burns through" the *Before* image; that is, where the parts of the *After* image with the strongest luminance replace the same parts of the *Before* image. We will leave this exact transition for the exercises, but we will consider an example where we approximate the luminance by the average of the R, G, and B colors in the *After* image. The effect of this transition is almost like the *After* image burning through the *Before* image, which is why we have chosen this name for it. In Figure 11.31 we see this transition partway through. It is not difficult to see some of the darker architectural features of the village scene coming through the cherries image.

Figure 11.31. The Hong village image burning through the cherry blossom image.

Again, the .glib file and vertex shader are essentially the same as previous ones, and the fragment shader is shown below.

```
uniform float uT;
uniform sampler2D uBeforeUnit, uAfterUnit;

in vec2 vST;

out vec4 fFragColor;

void main( )
{
  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit, vST ).rgb;
  vec3 color;

  if ( (argb.r + argb.g + argb.b)/3. < uT )
    color = argb;
  else
    color = brgb;

  fFragColor = vec4( color, 1.);
}
```

There is even less computation in this fragment shader; the average of the *After* color components is calculated and compared with the parameter *uT*, and the *After* color is used instead of the *Before* color when the color values are low (that is, when the colors are dark). As the value of *uT* moves from 0. to 1., more and more of the texels in the *After* image satisfy the condition and become part of the final display.

Break-Through

What if we had some other way for the *After* image to replace the *Before* image over time? What if, for example, we generated a random texture with a `Noise( )` function and used the values of that random texture to determine whether the *Before* or *After* image is used for each pixel? An example of this kind of transition is shown in Figure 11.32. This is something like the burn-through transition, but the image that controls the pixel selection is hidden and there is no apparent relation between this intermediate image and either of the two original images.

Because this process uses noise operations, the .glib and vertex shader are somewhat different from the ones we have seen before in this chapter. The .glib file simply selects a 3D noise texture and proceeds as in previous examples.

```
##OpenGL GLIB

Noise3D 128
Ortho -1. 1. -1. 1.

Texture 6 cherries.bmp
Texture 7 Hong.village.bmp

Vertex  transition.vert
Fragment transition.frag
Program Transition uBeforeUnit 6 uAfterUnit 7

QuadXY .2 5.
```

Figure 11.32. A break-through transition with the Hong village image replacing the cherry blossom image under the control of a noise function.

The vertex shader adds an input variable, the familiar *MCposition*, that holds the model coordinates for each vertex in the initial quad and, when it is interpolated across the quad, will hold the model coordinates for each pixel in the display.

```
out vec3 vMCposition;
out vec2 vST;

void main( )
{
  vMCposition = vec3(aVertex);
  vST     = aTexCoord0.st;
  gl_Position = uModelViewProjectionMatrix * aVertex;
}
```

Finally, the fragment shader gets the pixel colors for each image as usual, but then gets a noise value (the variable *nv*) for the pixel by querying the 3D sampler function `Noise3` at a position determined by the pixel's model coordinates. Since the original quad was 10 units across, we divide the model coordinates by 10 to get the actual texture coordinate for the pixel. The octaves of the noise value are then used to compute a numeric value whose fractional value is used for the comparison that selects the image.

```
uniform float uT;
uniform sampler3D Noise3;
uniform sampler2D uBeforeUnit, uAfterUnit;

in vec3 vMCposition;
in vec2 vST;
```

```
out vec4 fFragColor;

void main( )
{
  vec3 brgb = texture( uBeforeUnit, vST ).rgb;
  vec3 argb = texture( uAfterUnit, vST ).rgb;
  vec3 color;

  vec4 nv = texture(Noise3, vMCposition/10.);
  float sum = nv.r + nv.g + nv.b + nv.a;
  sum = ( sum - 1. ) / 2.;   // 0. to 1.
  sum = fract( sum );
  if ( sum < uT )
    color = argb;
  else
    color = brgb;

  fFragColor = vec4( color, 1.);
}
```

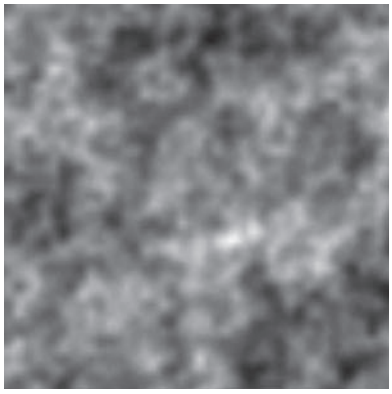

Figure 11.33. The grayscale texture used in the break-through transition.

Although we do not save it for any other use, this numeric value *sum* actually provides a noise texture that acts as the controller for the transition; if we set

```
color = vec3( sum, sum, sum );
```

instead of setting color in the if statement, we can see that texture, shown in Figure 11.33.

There are obviously many other ways you could control which image contributes the actual value for any pixel. For example, almost any of the image blending operations that involves taking part of one image and part of another image under control of a parameter could be used to create a transition by varying that parameter. Further developments are left for the curious reader.

Notes

These sections have discussed a number of techniques that are all rather similar, but that differ in how an image is processed on its own, is compared with a reference image, or is combined with a different image. The techniques are

straightforward; choosing the right one to use when you want to create a particular effect takes experience and some time.

Exercises

1. Complete the work of the CMYK separation example by presenting the four separations in their actual color, instead of in grayscale. You may use any image you like, but the file `Figure-11.3.tif` is included in the resources for the book so that you may compare your work to that in this chapter.
2. Create an anaglyph of a familiar scene, such as part of your home or campus, by taking two digital photographs from nearby points that frame the same portion of the middle ground of the scene, and combining them as described in this chapter.
3. Implement *image rotation* by any multiple of 90° by taking the original texture coordinates and applying trigonometric functions to them.
4. Implement *image flipping* or *image inversion*, the process where the top and bottom of an image are reversed. Do the same for reversing the left and right sides of an image.
5. Implement some different *image warping* approaches than the one we discussed in the chapter. Consider how you might use trigonometric functions in only one direction, exponential functions, or other kinds of manipulations to the texture coordinates.
6. Implement *selective coloring*. Using the luminance value, it is simple to convert an image to grayscale, but this can be done selectively. Get an image for which one thing stands out in a different color (for example, an apple on a windowsill) and make everything grayscale except that one thing. Use color testing on each pixel to decide whether or not to change it.
7. In the discussion of the interpolation operation, we use the equation below to combine the two colors we are blending:

```
vec3 target = vec3(0.5) - 0.25*cos(PI*brgb) -
     0.25*cos(PI*argb);
```

   Experiment with the values used to control the blending. As a first try, you might vary the base color b and the subtractive terms s in the equation

```
vec3 target = b - s*cos(PI*brgb) - s*cos(PI * argb);
```

with the relationship $s = (1 - b)/2$. Make b a slider uniform variable in a GLIB file and use *glman* to experiment with this concept. Record and comment on your results.

8. You can combine the manipulation techniques described in this chapter to achieve other specific effects. For example, if you have a photograph of a green apple but you want an image of a red apple, you can use the technique from the chromakey to select the greens of the apple and then use the hue shifting operation to change the green to red, while retaining some of the character of the green apple. Pick one of your images that has a strong area of some color, and change that color to another color.
9. Combine some of the effects from this chapter and see what you get. For example, you can sharpen images with one technique and then make them grayscale with another. (Does it matter in which order you do that?) You can take the image output of one technique and use it as the input to the next. If you push some of the techniques beyond their logical bounds (for example, take a *very* large mixing factor for sharpness) you may get some images that could effectively be taken into another technique (for example, grayscale). See what you can do!

The next two exercises consider other examples of image warping, similar to the example shown in Figure 11.16. Like that example, these come from [6, Chapter 16].

10. The ripple transformation displaces pixels in waves in both the x- and y-directions. This transformation has four parameters: the period lengths $\tau_x, \tau_y \neq 0$ (in pixels) and the wave magnitudes a_x, a_y (in pixels) in both directions:

$$x = x' + a_x \sin\left(2\pi y'/\tau_x\right) \text{ and } y = y' + a_y \sin\left(2\pi x'/\tau_y\right).$$

Create a shader that implements the ripple transformation, and apply it to both a grid image and a natural image. In [4] an example uses the parameters (in pixels) $\tau_x = 120$, $\tau_y = 250$, $a_x = 10$, and $a_y = 15$, so you might use these.

11. The spherical transformation simulates viewing the image through a hemispherical lens. If we assume that the lens is centered on the image, the parameters of this transformation are the radius of the lens $r_{\max}$ and its refraction index η. The functions that implement this transformation are

$$x = x' - \begin{cases} z\tan(\beta_x) & \text{for } r \leq r_{\max} \\ 0 & \text{for } r > r_{\max} \end{cases}$$

$$y = y' - \begin{cases} z\tan(\beta_y) & \text{for } r \leq r_{\max} \\ 0 & \text{for } r > r_{\max} \end{cases}$$

with

$$d_x = x' - x_c, \quad r = \sqrt{d_x^2 + d_y^2}, \quad \beta_x = \left(1 - \frac{1}{\rho}\right)\arcsin\left(\frac{d_x}{\sqrt{d_x^2 + z^2}}\right)$$

$$d_y = y' - y_c, \quad z = \sqrt{r_{\max}^2 - r^2}, \quad \beta_y = \left(1 - \frac{1}{\rho}\right)\arcsin\left(\frac{d_y}{\sqrt{d_y^2 + z^2}}\right)$$

Implement the spherical transformation and apply it to a grid image and to a natural image. A good value for the refraction index is $\eta = 1.8$.

The next few exercises ask you to examine some operations on pairs of images and see the results.

12. The *screen* operation is similar to the multiply operation, but you take the complement of each pixel's color components, multiply the components together, and take the complement of the result. Implement this operation. The result will be lighter than either original; explain why. As we did in the multiply example in the text, use a luminance computation to balance the screen operation results with the originals.
13. The *difference* between two images is defined by the absolute value of the color difference between the images' pixels. Implement this image operation.
14. *Negation* and *exclusion* are similar to difference, but treat the colors somewhat differently. For the negation operation, the color target is

```
vec3 target = vec3(1.,1.,1.) - abs(1. - argb - brgb );
```

while for the exclusion operation, the color target is

```
vec3 target = argb + brgb – 2.0 * argb * brgb;
```

Implement both the negation and the exclusion operations. The target for the negation operation is automatically in the legal range for color, but the target for exclusion may not be; you will probably want to clamp it to [0., 1.].

15. Color *burn* and *dodge* are two other related operations. The color burn operation is given by

```
vec3 target = vec3(1.,1.,1.) – (1.-argb)/brgb;
```

Since you are dividing by the value of color components that are no larger than one, you may get results greater than one, so you may need to clamp this result to [0., 1.]:

```
vec3 result = clamp( target, 0., 1. );
```

The color dodge operation involves a divide instead of a multiply and involves the inverse rather than the original of the second image. Again, some clamping may be needed.

```
vec3 target = argb/(vec3(1. - brgb);
```

Implement both the color burn and dodge operations.

16. Modify the burn-through transition to replace the RGB average value with the computed luminance. Can you see any subjective difference between these two transitions? Discuss why you think the difference, or lack of difference, you see is reasonable.
17. In a variation on the break-through transition, create a systematic gray-scale pattern texture and use that to control the selection of the image for each pixel. Look at the selection of transitions available in Powerpoint and identify the transitions that can be implemented by this approach.
18. In the break-through transition discussion, we said that you could actually display the noise texture used to control which image is presented at each stage of the transi-tion. Do this. Then capture one frame part way through the transition and compare that capture to the noise texture to see if you can identify the texture's action in the transition.
19. In the break-through transition discussion, Figure 11.33 shows how you can create an output image from the noise texture, by assigning the same value to all three color components. What if you assign `nv[0]` to red, `nv[1]` to green, and `nv[2]` to blue? What do you get? Why?
20. If you declare a variable

```
uniform float Timer;
```

then *glman* will fill it with a value from 0. to 1. over the course of 10 seconds. Try using `Timer` instead of `uT` in the image transitions to create an animated effect.