Laboratory 6
Blending

Alpha Blending

You have no doubt noticed the color quadruplet of "RGBA". As mentioned earlier, the *A* part is the *alpha channel*, and it is traditionally used for specifying translucency in an image. In a bitmap used for texturing, the alpha layer forms an 8-bit image, which can be translucent in one section, transparent in another, and completely opaque in a third. If an object isn't using texturing but instead has its color specified via its vertices, lighting, or overall global coloring, alpha will let the entire object or scene have translucent properties. A value of 1.0 means the object or pixel is completely opaque, while 0 means it is completely invisible.

For alpha to work as with any blending model, you work with both a source and a destination image.

We start with the `BouncySquare` project. Solid squares of colors are used here first instead of textured ones, because it makes for a simpler example.

The `onDrawFrame()` method of the `BouncySquareRenderer` is modified as follows. First of all, the buffer is cleared to black, making it easier to see any blending later.

```
gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
gl.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
gl.glMatrixMode(GL11.GL_MODELVIEW);
```

Next, we draw one square that is moved up and down and then back by 3 units, while given a blue color. Because there is no coloring-per-vertex, this call to `glColor4f` will set the entire square to blue. However, notice the last component of 1.0. That is the alpha, and it will be addressed shortly. And immediately following `glColor4f` is the call to actually draw the square.

```
gl.glLoadIdentity();
gl.glTranslatef(0.0f,(float)Math.sin(mTransY), -3.0f);
gl.glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
mSquare.draw(gl);
```

We then address the second square, coloring it red and moving it left and right. Moving it away by 2.9 instead of 3.0 units ensures that the red square will be in front of the blue one.

```
gl.glLoadIdentity();
gl.glTranslatef((float)(Math.sin(mTransY)/2.0f),0.0f, -2.9f);
gl.glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
mSquare.draw(gl);
mTransY += .075f;
```

It's not much to look at, but this will be the framework for the next several experiments. The first will switch on the default blending function.

As with so many other OpenGL features, turn blending on with the call `gl.glEnable(GL10.GL_BLEND)`. Add that anywhere before the first call to `mSquare.draw()`. Recompile, and what do you see that nothing has changed. That's because there's more to blending than that. We must specify a *blending function* as well, which describes how the source colors (as expressed via its fragments, or pixels) mix with those at the destination. The default, of course, is when the source fragments always replace those at the destination, when depth cueing is off. As a matter of fact, proper blending can be assured only when z-buffering is switched off.

Blending Functions

To change the default blending, we must resort to using `glBlendFunc`, which comes with two parameters. The first tells just what to do with the source, and the second, the destination. To picture what goes on, note that all that ultimately happens is that each of the RGBA source components is added, subtracted, or whatever, with each of the destination components. That is, the source's red channel is mixed with the destination's red channel, the source's green with the destination's green, and so on. This is usually expressed the following way: call the source RGBA values Rs, Gs, Bs, and As, and call the destination values Rd, Gd, Bd, and Ad. But we also need both source and destination blending factors, expressed as Sr, Sg, Sb, Sa, and Dr, Dg, Db, and Da. And here's the formula for the final composite color:

$$(R, G, B) = ((Rs * Sr) + (Rd * Dr), (Gs * Sg) + (Gd * Dg), (Bs * Sb) + (Bd * Db)).$$

In other words, multiply the source color by its blending factor and add it to the destination color multiplied by its blending factor.

One of the most common forms of blending is to overlay a translucent face on top of something that has already been drawn—that is, the destination. As before, that can be a simulated windowpane, a heads-up display for a flight simulator, or other graphics that just might look nicer when mixed with the existing imagery. Depending on the purpose, you may want the overlay to be nearly opaque, using an alpha approaching 1.0, or very tenuous, with an alpha approaching 0.0.

In this basic blending task, the source's colors are first multiplied by the alpha value, its blending factor. So, if the source red is maxed out at 1.0 and the alpha is 0.75, the result is derived by simply multiplying 1.0 by 0.75. The same would be used for both green and blue. On the other hand, the destination colors are multiplied by 1.0 minus the source's alpha. Why? That effectively yields a composite color that can never exceed the maximum value of 1.0; otherwise, all sorts of color distortion could happen. Or imagine it this way: the source's alpha value is the proportion of the color "width" of 1.0 that the source is permitted to fill. The leftover space then becomes 1.0 minus the source's alpha. The larger the alpha, the greater the proportion of the source color that can be used, with an increasingly smaller proportion reserved for the destination color. So, as the alpha approaches 1.0, the greater the amount of the source color is copied to the frame buffer, replacing the destination color.

Now we can examine that in the next example. To set up the blending functions described earlier, you would use the following call:

```
gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
```

The `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA` are the blending factors described earlier. And remember that the first parameter is the source's blending, the object being written at present. Place that line immediately after where you enable blending. Then compile and run.

So, what's happening? The blue has an alpha of 1.0, so each blue fragment completely replaces anything in the background. Then the red with an alpha of 0.5 means that 50 percent of the red is written to the destination. The black area will be a dim red, but only 50 percent of the specified value of 1.0 given in `glColor4f`. Now on top of the blue, 50 percent of the red value is mixing with a 50 percent blue value:

$$Blended\ color = Color\ Source * Alpha\ of\ source + (1.0 - Alpha\ of\ Source) * Color\ of\ the\ destination.$$

Or looking at each component based on the values in the previous example:

$$Red = 1.0 * 0.5 + (1.0 - 0.5) * 0.0$$

$$Green = 0.0 * 0.5 + (1.0 - 0.5) * 0.0$$

$$Blue = 0.0 * 0.5 + (1.0 - 0.5) * 1.0.$$

So, the final color of the fragment's pixels should be 0.5, 0.0, 0.5, or magenta. Now the red and resulting magenta are a little on the dim side. What would you do if you wanted to make this much brighter? It would be nice if there were a means of blending the full intensities of the colors. We can't use alpha values of 1.0, because with blue as the destination and a source alpha of 1.0, the earlier blue channel equation would be $0.0 * 1.0 + (1.0 - 1.0) * 1.0$. And that equals 0, while the red would be 1.0, or solid. What you would want is to have the brightest red when writing to the black background, and the same for the blue. For that you would use a blending function that writes both colors at full intensity, such as `GL_ONE`. That means the following:

```
gl.glBlendFunc(GL10.GL_ONE, GL10.GL_ONE);
```

Going back to the equations using the source triplet of $red = 1.0$, $green = 0.0$, $blue = 0.0$, and destination of $red = 0.0$, $green = 0.0$, $blue = 1.0$ (with alpha defaulting to 1.0), the calculations would be as follows:

$$Red = 1.0 * 1.0 + 0.0 * 1.0$$

$$Green = 0.0 * 1.0 + (0.0 - 0.0) * 1.0$$

$$Blue = 0.0 * 1.0 + (1.0 - 0.0) * 1.0.$$

And that yields a color in which $red = 1.0$, $green = 0.0$, and $blue = 1.0$. And that is magenta.

Now it's time for another experiment. Take the code from the previous example, set both alphas to 0.5, and reset the blend function to the traditional values for transparency:

```
gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
```

After you run this modified code, take note of the combined color, and notice that the further square is blue at -4.0 away and is also the first to be rendered, with the red one as the second. Now reverse the order of the colors that are drawn, and run.

The intersections are slightly different colors. This shows one of the issues in OpenGL: as with most 3D frameworks, the blending will be slightly different depending on the order of the faces and colors when rendered. In this case, it is actually quite simple to figure out what's going on. In the first case, the blue square is drawn first with an alpha of 0.5. So, even though the blue color triplet is defined as 0, 0, 1, the alpha value bring that down to 0, 0, 0.5 as it is written to the frame buffer. Now add the red square with similar properties. Naturally the red will write to the black part of the frame buffer in the same manner as the blue, so the final value will be

0.5, 0, 0. But note what happens when red writes on top of the blue. Since the blue is already at half of its intensity, the blending function will cut that down even further, to 0.25, as a result of the destination part of the blending function, $(1.0 - Source\ alpha) * blue + destination$, or $(1.0 - 0.5)0.5 + 0.0$, or 0.25. The final color is then 0.5, 0, 0.25. With the lower intensity of the blue, it contributes less to the composite color, leaving red to dominate. Now in the second case, the order is reversed, so the blue dominates with a final color of 0.25, 0, 0.5.

One final method here that might be really handy in some blending operations is that of `glColorMask`. This function lets you block one or more color channels from being written to the destination. To see this in action, modify the red square's colors to be 1, 1, 0, 1; set the two blend functions back to `GL_ONE`. The red square is now yellow and, when blended with blue, yields white at the intersection. Now add the following line:

```
gl.glColorMask(true, false, true, true);
```

The preceding line *masks*, or turns off, the green channel when being drawn to the frame buffer.

Multicolor Blending

Now we can spend a few minutes looking at the effect of blending functions when the squares are defined with individual colors for each vertex. Add the following code to the constructor for the `Square`. The first color set defines yellow, magenta, and cyan. The complementary colors to the standard red-green-blue are specified in the second set.

```
float squareColorsYMCA[] =
{
    1.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 1.0f, 1.0f,
    0.0f, 0.0f, 0.0f, 1.0f,
    1.0f, 0.0f, 1.0f, 1.0f
};
float squareColorsRGBA[] =
{
    1.0f, 0.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f, 1.0f
};
```

Assign the first color array to the first square (which has been the blue one up until now) and the second to the former red square. Do this in `SquareRenderer` and pass the color arrays in via the square's constructor. Of course, now we need two squares, one for each color set instead of just the one. And don't forget to enable the use of the color array.

You should be familiar enough now to know what to do. Also, notice that the arrays are now normalized as arrays of floats as opposed to the previously used unsigned bytes, so you will have to tweak the calls to `glColorPointer`. Run the project with the blending disabled, and with blending enabled using the traditional function for transparency.

One solution to seeing some real transparency would be to use the following:

```
gl.glBlendFunc(GL10.GL_ONE, GL10.GL_ONE);
```

This works because it ditches the alpha channel altogether. If you want alpha with the "standard" function, merely change the 1.0 values to something else, such as 0.5, and change the blend function to the following:

```
gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
```

Texture Blending

Now, we can approach the blending of textures. Initially this seems much like the alpha blending described earlier, but all sorts of interesting things can be done by using multitexturing.

First let's rework the previous code to support two textures at once and do vertex blending. You will have to use the `draw()` and `createTexture()` methods from the Laboratory 5 examples. The square will need to support texture coordinates, and each instance of the square will need their own unique texture. First disable blending, then activate the colors from the previous exercise and enable blending using the `GL_ONE` functions from earlier in this laboratory. Use the `hedly.png` texture for one of the squares, and the `goldengate.png` texture for the other.

Using a single bitmap and colorizing it is a common practice to save memory. If you are doing some UI components in the OpenGL layer, consider using a single image, and colorize it using these techniques. You might ask why is it a solid red as opposed to merely being tinted red, allowing for some variation in colors. What is happening here is that the vertex's colors are being multiplied by the colors of each fragment. For the red, we used the RGB triplet of 1.0, 0.0, 0.0. So when each fragment is being calculated in a channel-wise multiplication, the green and blue channels are going to be multiplied by 0, so they are completely filtered out, leaving just the red. If you wanted to let some of the other colors leak through, you would specify the vertices to lean toward a more neutral tone, with the desired tint color being a little higher than the others, such as 1.0, 0.7, 0.7.

You can also add translucency to textures quite easily. To enable this, we will introduce a small simplifying factor here. You can colorize the textured face by one single color by simply using `glColor4f` and eliminate the need to create the vertex color array altogether. So, for the second square, the closest one, color it using `glColor4f(1.0f, 1.0f, 1.0f, 0.75f)`, and make sure to reset the coloring for the first square with `glColor4f(1.0f, 1.0f, 1.0f, 1.0f)`; otherwise, it will darken with the second one. Also, ensure that blending is turned on and that the blending function uses the `SRC_ALPHA/ONE_MINUS_SRC_ALPHA` combination.

Multitexturing

So now we have covered blending for colors and mixed mode with textures and colors, but what about two textures together to make a third? Such a technique is called *multitexturing*. Multitexturing can be used for layering one texture on top of another while performing certain mathematical operations. More sophisticated applications include simple image processing.

Multitexturing requires the use of *texture combiners* and *texture units*. *Texture combiners* let you combine and manipulate textures that are bound to one of the hardware's texture units, the specific part of the graphics chip that wraps an image around an object.

To set up a pipeline to handle multitexturing, we need to tell OpenGL what textures to use and how to mix them together. The process isn't that much different (in theory at least) than defining the blend functions when dealing with the alpha and color blending operations previously. It does involve heavy use of the `glTexEnvf` call, another one of OpenGL's wildly overloaded methods. This sets up the texture environment that defines each stage of the multitexturing process.

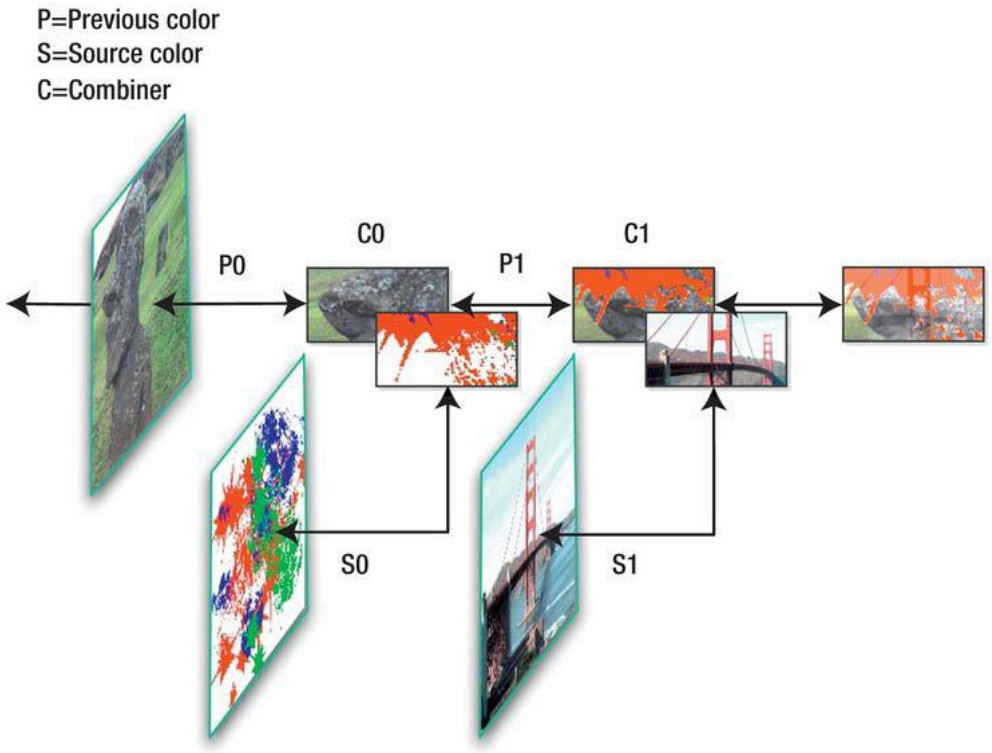

Figure 1: The texture combiner chain

Figure 1 illustrates the combiner chain. Each combiner refers to the previous texture fragment ($P0$) or the incoming fragment for the first combiner. It then takes a fragment from a "source" texture ($S0$ in the figure), combines it with $P0$, and hands it off to the next combiner if needed, $C1$, and the cycle repeats.

In the following example, two textures are loaded together, bound to their respective texture units, and merged into a single output texture. Several different kinds of methods used to combine the two images are tried with the results of each examined in depth. The first texture is `hedly.png` and the second is `splash.png`.

First off, we revisit our `draw()` method from the `Square` class. We are back to only a single texture, going up and down. The color support has also been turned off. And make sure you are still loading a second texture. The code should look as follows:

```
gl.glEnable(GL10.GL_TEXTURE_2D);
gl.glBindTexture(GL10.GL_TEXTURE_2D, mTexture0);
gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
```

then the usual

```
gl.glFrontFace(GL11.GL_CW);
gl.glVertexPointer(2, GL11.GL_FLOAT, 0, mFVertexBuffer);
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glColorPointer(4, GL11.GL_FLOAT, 0, mColorBuffer);
gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
```

Now, we use two calls to `glClientActiveTexture`, which sets what texture unit to operate on. This is on the client side, not the hardware side, and indicates which texture unit is to receive the texture coordinate array. Don't get this confused with `glActiveTexture`, used below, that actually turns a specific texture unit on.

```
gl.glClientActiveTexture(GL10.GL_TEXTURE0);
gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mTextureCoords);
gl.glClientActiveTexture(GL10.GL_TEXTURE1);
gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mTextureCoords);
```

Now, call the method that configures the texture units.

```
multiTexture(gl,mTexture0,mTexture1);
```

Lastly, draw the square:

```
gl.glDrawElements(GL11.GL_TRIANGLES, 6, GL11.GL_UNSIGNED_BYTE, mIndexBuffer);
gl.glFrontFace(GL11.GL_CCW);
```

`mTexture0` and `mTexture1` are `private int` fields of the `Square` class, which are initialized in the `setTextures(GL10 gl, Context context, int resourceID0, int resourceID1)` method of the same class:

```
mTexture0 = createTexture(gl,context,resourceID0);
mTexture1 = createTexture(gl,context,resourceID1);
```

In this case, the `createTexture()` method must return an `int`, namely, add to the end of the `createTexture()` method the following line:

```
return textures[0];
```

Let us turn our attention to the method `multiTexture(GL10 gl, int tex0, int tex1)`. First of all, we must specify what the combiners should do

```
float combineParameter = GL10.GL_MODULATE;
```

Then we set up the two textures. `glActiveTexture` makes active a specific hardware texture unit.

```
gl.glActiveTexture(GL10.GL_TEXTURE0);
gl.glBindTexture(GL10.GL_TEXTURE_2D, tex0);
gl.glActiveTexture(GL10.GL_TEXTURE1);
gl.glBindTexture(GL10.GL_TEXTURE_2D, tex1);
```

Now we should tell the system what to do with the textures in the final line of the method. Set the texture environment mode for the textures to combine:

```
gl.glTexEnvf(GL10.GL_TEXTURE_ENV, GL10.GL_TEXTURE_ENV_MODE, combineParameter);
```

Finally, it's time to play with other combiner settings. Try `GL_ADD` for the `combineParameter` above replacing `GL_MODULATE`. Then follow this by `GL_BLEND` and `GL_DECAL`.