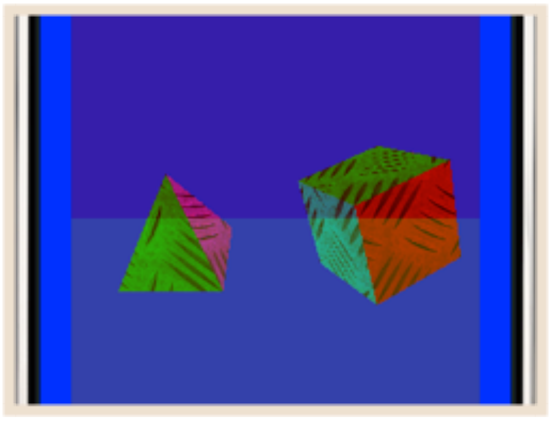
SIMON MAURICE - IPHONE - OPENGL ES



Tuesday, 21 April 2009

OpenGL ES 09 - Blending Without Mr. Buzzy Part 1

NOTE: I'm not particularly happy with this tutorial at the moment but I have posted it here in it's current state so you can get what you can out of it rather than hold back. Don't worry if there are errors, but the code all works. I'll come back and revise this shortly as well as writing part 2.

What's Mr. Buzzy? That's my kitchen blender! We're obviously not talking about my kitchen blender or the Blender 3D software, we're talking OpenGL blending. However, this is going to be part one of two or three tutorials. The effect of blending is quite simple but the detail can take quite a bit to cover. This will be an overview of blending with the detail to follow at a later stage.

I remember the first time I used blending in code. Fortunately, I didn't have to write the algorithm myself, but the effect was really cool. It was on an SGL Indigo II with a GR3-Elan graphics board (if you've ever seen inside an SGI Indigo, or other of their 1990's era graphics powerhouses, you'll know it's a full sized graphics board!). The processing power of that box is practically miniscule compared to this

Mac that I'm using right now, but for the time, *it was spectacular!*

The main reason it was an exciting moment was that, at the time, none of us had ever seen blending executed real time on texture mapped objects, with translations through a scene and lighting effects. Needless to say, we'd seen it in LightWave and other ray-tracing software, but that was hardly real time. Yes, those were the days when coding was something that you did because you wanted to do something new and cool; not because you thought you could output a game in a couple of weeks and make some coin off it.

Anyway, let's get onto learning blending. Blending, is the process of ***combining*** two images together so that the front image appears partially transparent. For example, just say you have a piece of red perspex and look through it. The world will appear red because the colour of the perspex will alter the colours of all objects you view through it.

So, to make blending work, we need an object in the foreground which is partially transparent (ie an alpha value of less than 1.0 in our RGBA colour definition) and something behind it which it will "blend" with.

That is the critical point about blending: you ***must*** have two objects with the foreground object being partially transparent. Blending ***is not*** transparency. Blending cannot work without some transparency but a partially transparent object can be blended with an object behind it.

I was always taught that the correct term for blending is in fact "alpha compositing". I won't call it that (never have) but you may see the term around. Computer graphics has jumped a long way since my formal studies but when I was taught this, I believe that blending was just one form of alpha compositing, and alpha compositing is a method of drawing single or combined images with and alpha channel.

Getting Started
Download our project from tutorial #7: AppleCoder-OpenGLES-07.zip

We'll use this project and use the texture mapped pyramid and cube in the background for our blending effects to be superimposed upon. Fire this up in Xcode.

Blending in OpenGL
To make blending work in OpenGL, all you need to turn on the "state" within OpenGL that makes it use blending. I'm sure you know how to do this:

```
    glEnable(GL_BLEND);
```

The place to put this line in the `drawView[]` method right before we draw our partially transparent objects and then turn it off again with `glDisable()`.

OK, so blending is switched on (remember to turn it off if you don't need it). One of the great things about OpenGL, is that the blending can be executed using different blending methodologies to produce different blending effects. These are called *blend functions*. I'll just turn one on and discuss them in detail below.

Now, go to the `drawView[]` method. We are going to define a rectangle which we will put onto the display in front of our two objects.

```
    const GLfloat blendRectangle[] = {
        1.0, 1.0, -2.0,
        -1.0, 1.0, -2.0,
        -1.0, -1.0, -2.0,
        1.0, -1.0, -2.0
    };
```

We want to render two of these. So, in the drawing code will use the `glPushMatrix()` and `glPopMatrix()` pair to translate them independently.

Scroll down in the `drawView[]` method until you get **past** the drawing code for the pyramid and the cube. In order to correctly render the blending effects, we need to draw the opaque objects first, then the transparent ones just like the old painter's algorithm you've probably come across in graphics programming.

Now, think about the current state of OpenGL. We currently have texture mapping turned on (of course) so do we want this same texture on our rectangles? Not for this example so let's turn off texture mapping with this line:

```
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
```

Now, turn on blending:

```
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
```

This turns on blending with the `glEnable()` function. Turning on blending is not enough, we need to tell OpenGL *how* we want the blending to work otherwise the default state will do nothing interesting. That's where `glBlendFunc()` comes into play. Don't worry about this for now, I'm going to discuss the blending function in detail below; right now, let's just get it working and then we can experiment.

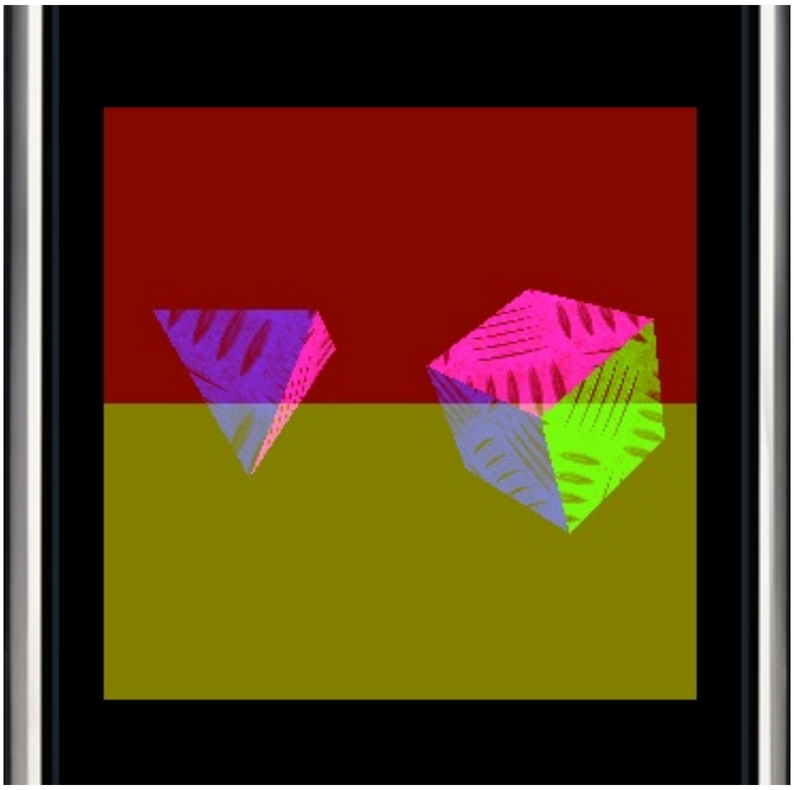
So, texture mapping is turned off, blending is turned on, and we've told OpenGL how to blend with the `glBlendFunc()` call. It's time to draw two rectangles.

```
glPushMatrix();
{
    glTranslatef(0.0, 1.0, -4.0);
    glVertexPointer(3, GL_FLOAT, 0, blendRectangle);
    glEnableClientState(GL_VERTEX_ARRAY);
    glColor4f(1.0, 0.0, 0.0, 0.4);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();

glPushMatrix();
{
    glTranslatef(0.0, -1.0, -4.0);
    glColor4f(1.0, 1.0, 0.0, 0.4);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();
glDisable(GL_BLEND);
```

Apart from the call to `glDisable()` at the end, nothing is new here. We've just taken our `blendRectangle` definition, positioned it on the display, coloured it and drawn it. We then need to disable blending otherwise OpenGL will blend our pyramid and cube next time this function is called (you can experiment later on if you like).

That's it! Hit "Build and Go" and you'll get the following:



Note the colours of the pyramid and cube change as they cross the two colours of our rectangles? That's blending at work. The colours with the two front rectangles which are partially transparent (40%

transparent) and combining or blending with the colour on the objects to create a new colour.

Hence, the yellow rectangle "brightens" the colours slightly compared to the red rectangle (just a bit more colour saturation).

If you got just a red and yellow rectangle and no objects, you've put the drawing code in the wrong spot: you need to draw the transparent objects after drawing the opaque objects; or you haven't enabled blending properly.

That's blending in it's most basic form. Nothing particularly special, but we're getting somewhere in learning about blending.

Quick Recap Before Continuing
So, to use blending, we need a partially transparent object in the foreground of our scene. We draw the solid objects first. Then we enable blending with a call to `glEnable()`, set the blend factor with `glBlendFunc()`, and finally draw our partially transparent objects from back to front.

Now we need to get into the single real complexity of blending. Setting the blend factors.

Blending Factors
I've just deleted several pages of text which I had just written. It was all about calculating the final pixel colour and was loaded full of what, is essentially, matrix maths (although it was not presented in that way). Nope, I don't think that's the right thing to do otherwise I might as well be regurgitating the man page.

It's time for me to get more creative and more practical but you do need some theory first: how does blending actually work.

As I described earlier, you need a partially transparent object, and something in the background for blending to work against. You can actually do blending with nothing more than the background as we call `glClear()` every time we redrew a scene in the tutorials so far so you can just blend against that if you like.

So, for each pixel on the display, OpenGL knows a couple of things:

1. There is a *source* pixel. This is the new pixel we are currently drawing from the partially transparent object (the red and yellow rectangles in our case).
2. The *destination* pixel. This is the pixel which is currently in our display buffer. It is what our partially transparent object is being drawn over. In our example above, it is either an object (cube or

pyramid), or the background.
3. The *source pixel blend factor specifier*. This is the first parameter passed to `glBlendFunc()`. It tells OpenGL how to process the new pixel.
4. The *destination blend factor specifier.* This is the second parameter passed to `glBlendFunc()` and tells OpenGL how to process the destination pixel.

Knowing the above four pieces of information, OpenGL will calculate new source and destination blend factors and apply it to the destination buffer. That is, in a nutshell, all the crap I just deleted!

The key is the two parameters we pass to `glBlendFunc()`.

In the above example, we used the following code to set the blending function:

```
glBlendFunc(GL_ONE, GL_ONE);
```

The two parameters are the same: `GL_ONE`. The first parameter affects the *source* or *incoming* pixel value and the second parameter affects the *destination* pixel value.

There are a whole host of different blend factor specifiers which can be passed to `glBlendFunc()`. What I need you to understand right now is that it's not what they individually mean, *it's how they are used in combination that affects our blending effects*.

That's an important point. I know I haven't listed the possible options yet but I need you to get through this bit of theory first.

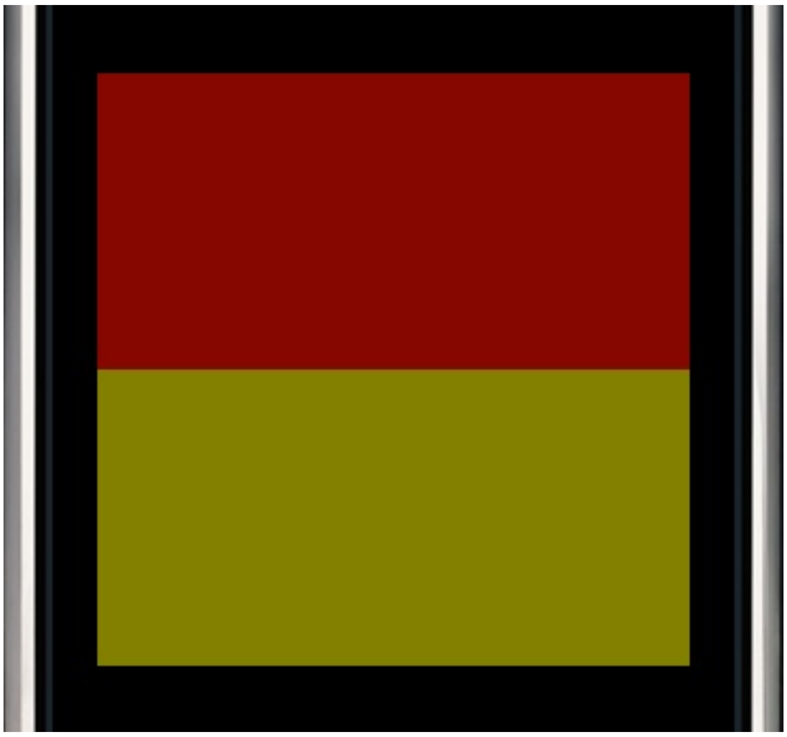
It needs to be clear in your mind if you're to get the most out of blending that, because there is the interaction of the new colour being the partially transparent object (our rectangles), and the pre-existing solid objects (our cube, pyramid or our background), that it is the *combination* of the two blending factors that control the finished effect.

To get this point across, let's get some examples out of the way. What the two parameters means right now is not important! What I am trying to show you is what happens when you change parameters

This one is the OpenGL default when you turn on blending with `glEnable()`:

```
glBlendFunc(GL_ONE, GL_ZERO);
```

We've changed the second parameter to `GL_ZERO`. Remember the second parameter affects our pre-existing "solid" objects in the buffer and not the partially transparent objects. It gives us the following:



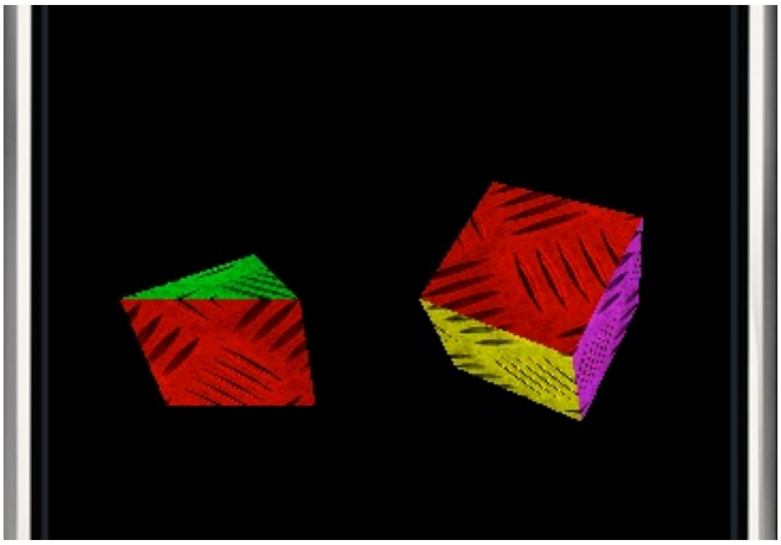
So, what's happened to our background objects. Without much explanation, the `GL_ZERO` is one hell of a clue.

This is not a "no blend" situation, but the destination pixel is being multiplied by the RGBA colour (0, 0, 0, 0) resulting in nothing being rendered. The source (the two front rectangles are being multiplied `GL_ONE` by the RGBA One (1, 1, 1, 1). That is a bit of an oversimplification but it should get the point across.

This would be the same result if you did not make any call to `glBlendFunc()` as it is the default state when blending is enabled.

Swap the two parameters around like this:

```
    glBlendFunc(GL_ZERO, GL_ONE);
```



Not surprisingly, the two rectangles have vanished (it's what you get when you multiply things by zero).

OK, so you know that `GL_ZERO` and `GL_ONE` obviously do what their names suggest, what about the other options. It's time to list the full set of possible parameters for `glBlendFunc()`.

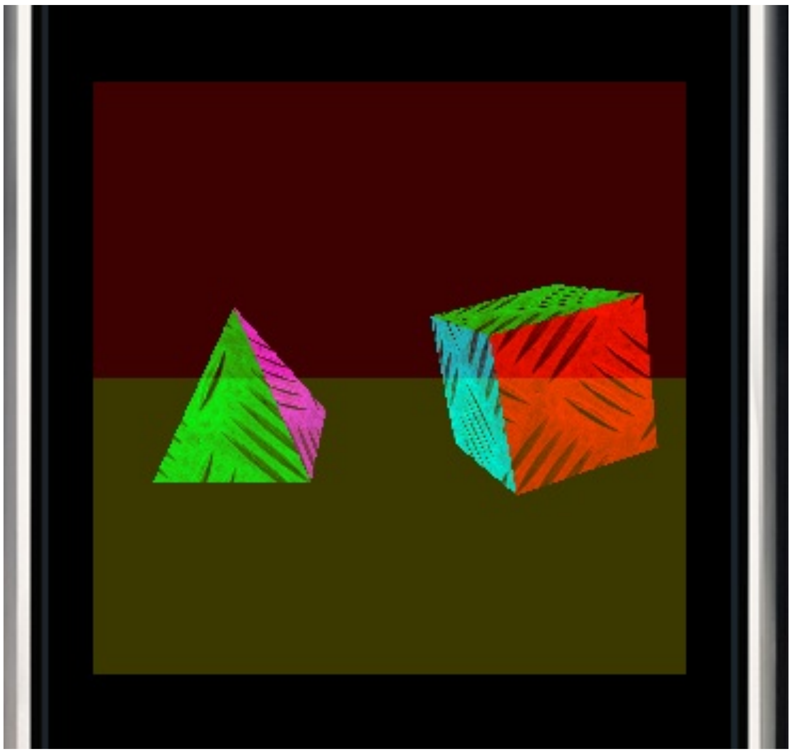
```
GL_ZERO
GL_ONE
GL_SRC_COLOR
GL_ONE_MINUS_SRC_COLOR
GL_DST_COLOR
GL_ONE_MINUS_DST_COLOR
GL_SRC_ALPHA
GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA
GL_ONE_MINUS_DST_ALPHA
GL_SRC_ALPHA_SATURATE
```

I'll go into the detail of each definition in later tutorials. Some of these are source only, some are destination only, check the man page if you're interested in working out what's what.

Blending Examples

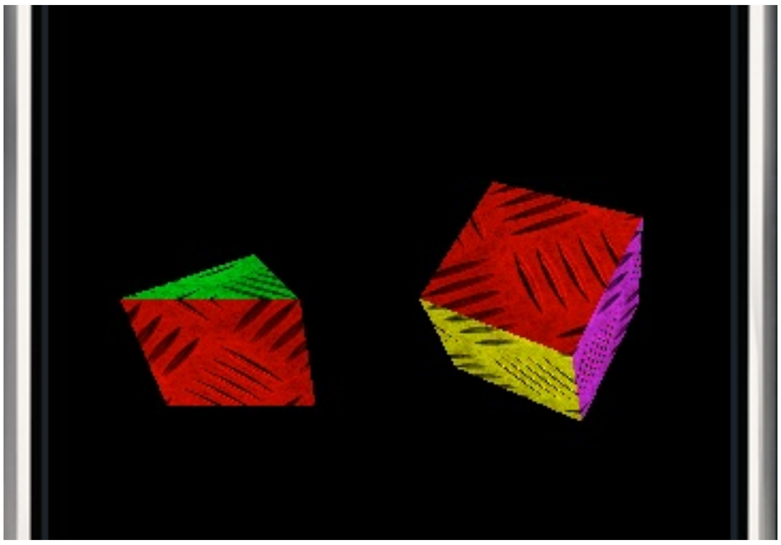
I'm going to fire through a few examples now. Just to give you an idea to experiment with different blend functions.

```
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```



Note that the front two rectangles are more transparent? Compared to the original blend we did, the objects "appear" brighter and the rectangles "appear" darker. This is your eye being tricked a little.

```
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
```

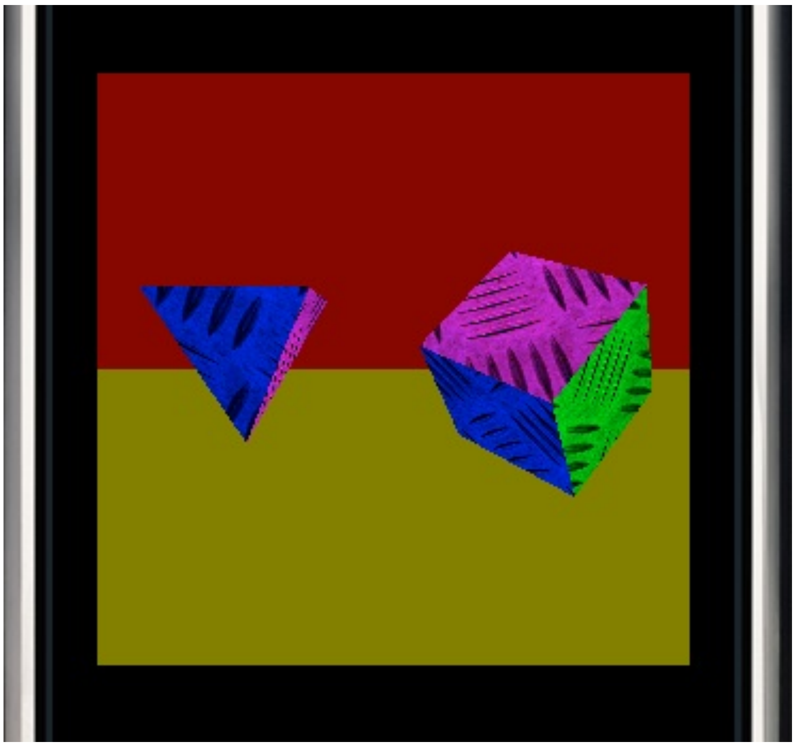


Our rectangles have disappeared! Well, not exactly. Their alpha value is being factored by One Minus Destination Alpha. So, our destination has the alpha of 1.0 so 1.0 - 1.0 = 0.0.

Notice one thing though: the background colour is being multiplied against the colour set by `glClearColor()` which also has an alpha of 1.0. Go back to the `setupView[]` method and change `glClearColor()` function call to:

```
glClearColor(0.0, 0.0, 0.0, 0.0);
```

Only change this one line. Now look what you get:



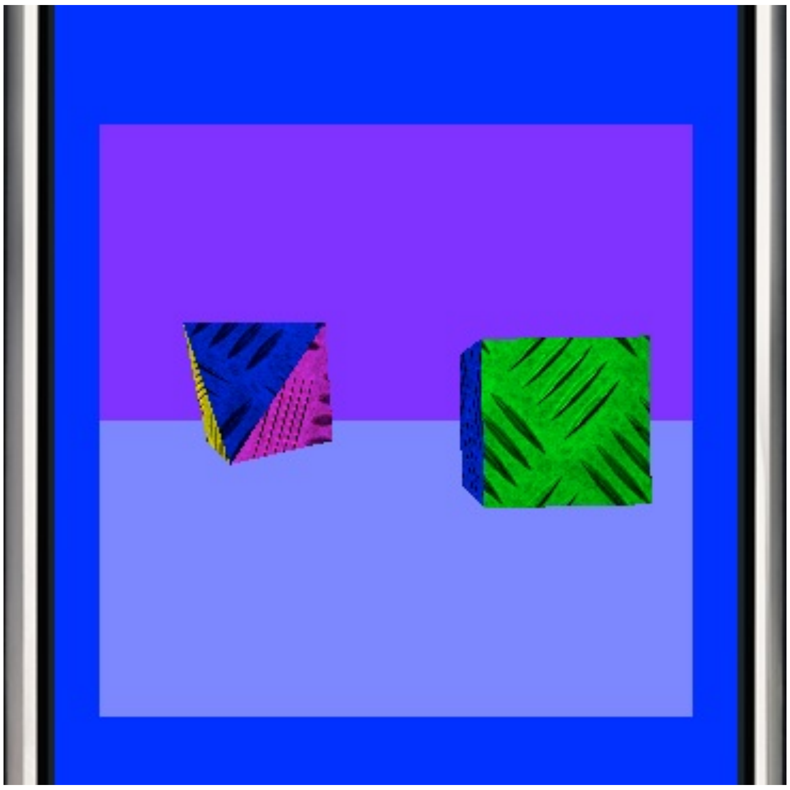
Can you work out what's going on? Whilst it looks like the rectangles have moved to the background, it's really blending at work again. From the previous example, wherever OpenGL renders against a pixel (ie the pixels of our objects) which have an alpha of 1.0, the rectangles become 100% transparent. Now that the background has an alpha of 0.0, the alpha level of our rectangles are then 1.0 - 0.0 which renders a colour.

Let's look at that a little closer:

Our red rectangle has a colour of: (1.0, 0.0, 0.0, 0.4). So against a black background with zero alpha, the rendered or blended colour is (1.0, 0.0, 0.0, 0.4). While that is simplistic due to the background being black, let's change the clear colour to blue. Go to `setupView[]` and change the `glClearColor()` call to:

```
    glClearColor(0.0, 0.0, 1.0, 0.0);
```

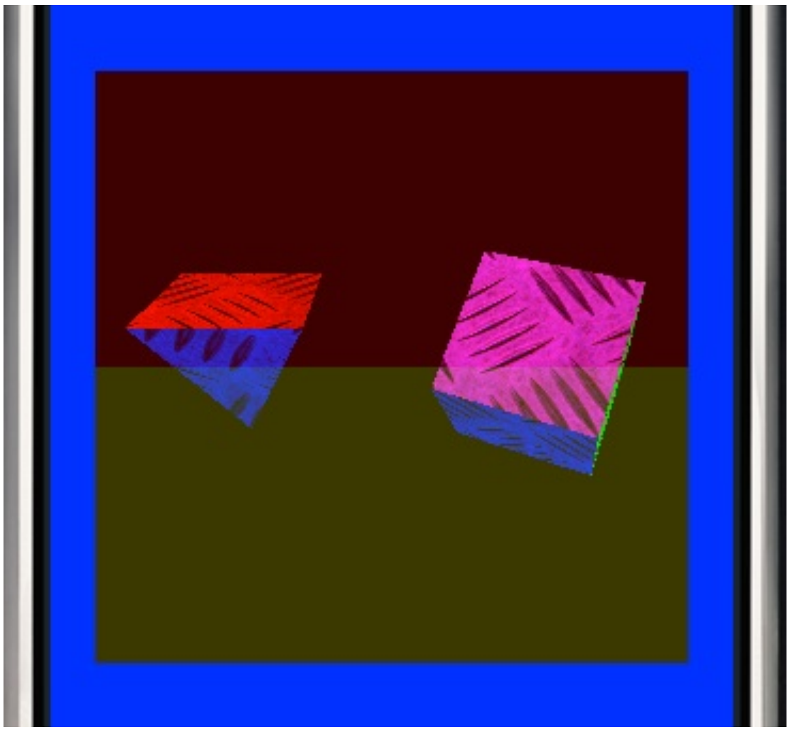
This sets the background to blue, full intensity. Hit "Build & Go" and you'll get:



The blue background is obvious. Now the cube and the pyramid are still being rendered without blending effects (ie the alpha of the front two rectangles are zero where they intersect with our objects) but the blending effect against our background colour has caused the red to change to purple and the yellow to a grey/blue.

I am going to go through one final example, then go through the calculations and we'll work out a pixel colour together.

```
    glBlendFunc(GL_SRC_ALPHA, GL_DST_ALPHA);
```

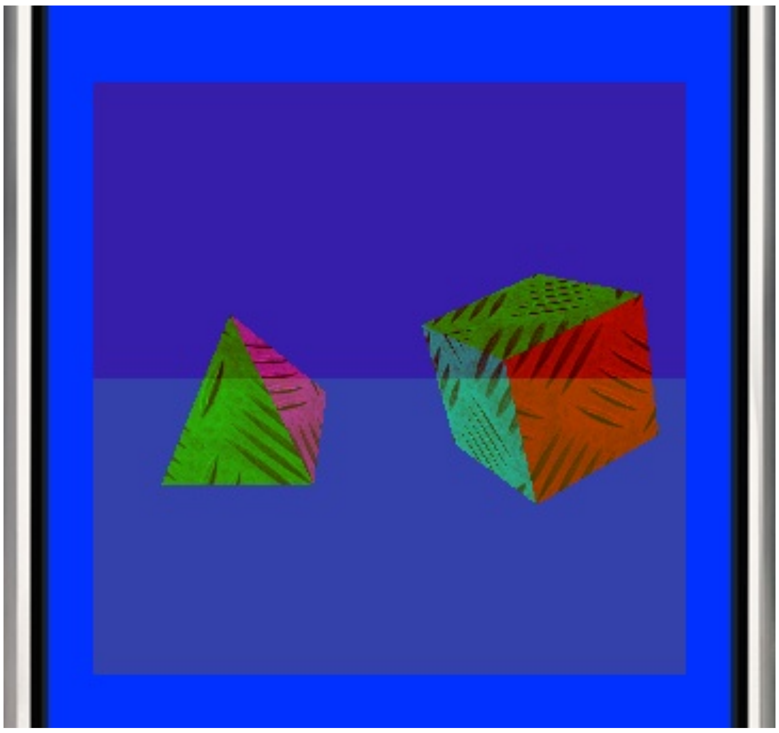


One Final Blending Combination...

Let's now look at what is the most common parameter pair:

```
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

In our test project here, the result would look like the following:



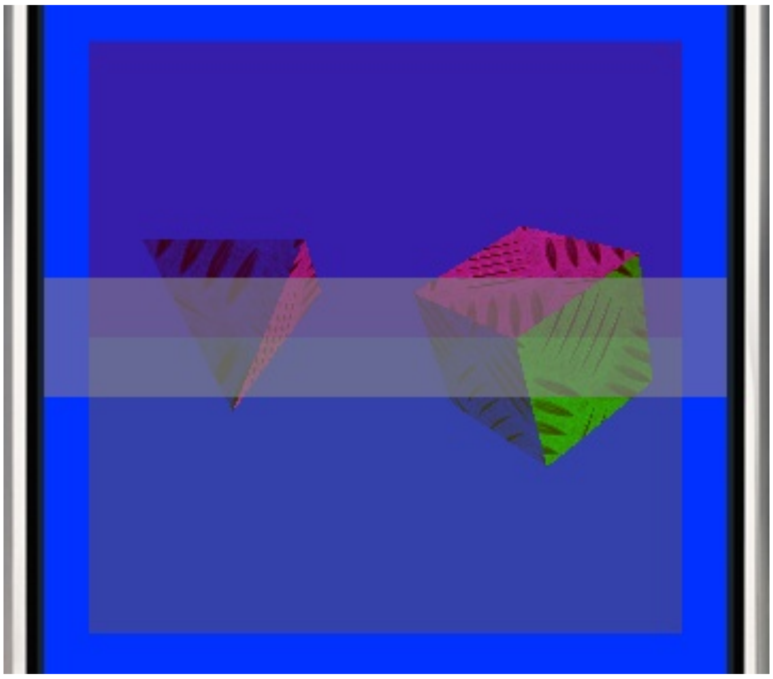
If there was panacea for choosing the right parameters for `glBlendFunc()`, this would it. It's your bog standard blend and probably the most utilised. If in doubt, start with that!

Adding Additional Blends

You don't need to only blend one object per pixel. You can have many partially transparent objects layered on top of each other. Add the following code to the `drawView[]` method after the drawing of the second rectangle:

```
    glPushMatrix();
    {
        glTranslatef(0.0, 0.0, -3.0);
        glScalef(1.0, 0.3, 1.0);
        glColor4f(1.0, 1.0, 1.0, 0.6);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    }
    glPopMatrix();
```

All we're doing is adding a third rectangle in front of the other two. OpenGL will combine the blending together quite happily producing the following:



Of course, I've put something in there I haven't covered before: `glScalef()`.

Scaling - Something I Haven't Covered Yet!

Scaling is simply resizing of an object, much like what you do in a paint program (except that's 2D). It takes three parameters: `xscale`, `yscale`, and `zscale`.

In the example above, `xscale` and `zscale` have a scaling factor of 1.0 meaning original size. Numbers larger than 1.0 magnify or make them larger, smaller values shrink them. So with the `yscale` value of 0.3, the height of the rectangle has been reduced to 30% of it's original size. I did that so the cube and the pyramid would pass in and out of the "double blended" area to produce maximum effect.

It's pretty straight forward and should have covered it earlier when dealing with transformations.

Conclusion to this Tutorial

That will do me for this tutorial for now. Whilst I know there's a lot more to cover in blending, particularly the detail in how the different blending combinations work, I'm going to leave that for another day. What I do want you to get out of this lesson is how to turn blending on, and the basic operation of the `glBlendFunc()` function call.

As usual, here's the code for this tutorial:

AppleCoder-OpenGLES-09.zip

The home of the tutorials is in the "Tutorials" section of the iphonedevsdk.com forums. Check out the thread there.

Until next time, hooroo!
Simon Maurice

previous *next*