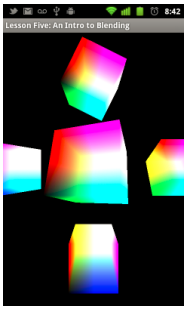
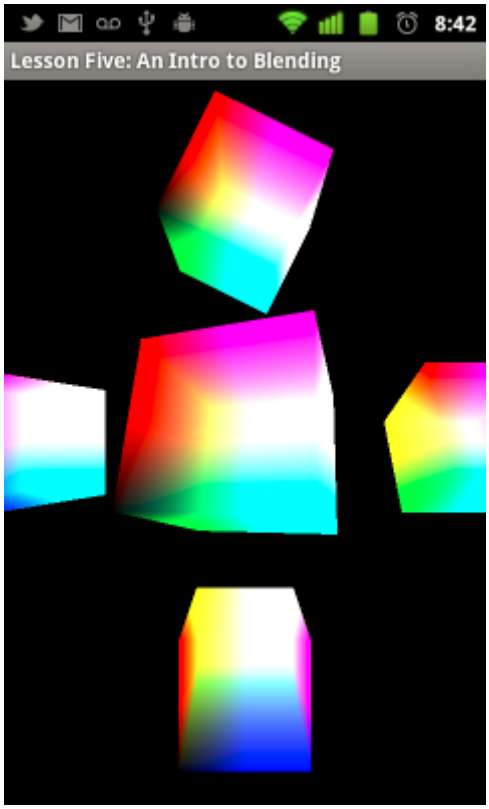
Learn OpenGL ES
Learn how to develop mobile graphics using OpenGL ES 2

Android Lesson Five: An Introduction to Blending



In this lesson we'll take a look at the basics of blending in OpenGL. We'll look at how to turn blending on and off, how to set different blending modes, and how different blending modes mimic real-life effects. In a later lesson, we'll also look at how to use the alpha channel, how to use the depth buffer to render both translucent and opaque objects in the same scene, as well as when we need to sort objects by depth, and why.

We'll also look at how to listen to touch events, and then change our rendering state based on that.



Basic blending.

ASSUMPTIONS AND PREREQUISITES

Each lesson in this series builds on the one before it. However, for this lesson it will be enough if you understand Android Lesson One: Getting Started. Although the code is based on the preceding lesson, the lighting and texturing portion has been removed for this lesson so we can focus on the blending.

BLENDING

Blending is the act of combining one color with a second in order to get a third color. We see blending all of the time in the real world: when light passes through glass, when it bounces off of a surface, and when a light source itself is superimposed on the background, such as the flare we see around a lit streetlight at night.

OpenGL has different blending modes we can use to reproduce this effect. In OpenGL, blending occurs in a late stage of the rendering process: it happens once the fragment shader has calculated the final output color of a fragment and it's about to be written to the frame buffer. Normally that fragment just overwrites whatever was there before, but if blending is turned on, then that fragment is *blended* with what was there before.

By default, here's what the OpenGL blending equation looks like when `glBlendEquation()` is set to the default, `GL_FUNC_ADD`:

```
output = (source factor * source fragment) + (destination factor * destination fragment)
```

There are also two other modes available in OpenGL ES 2, `GL_FUNC_SUBTRACT` and `GL_FUNC_REVERSE_SUBTRACT`. These may be covered in a future tutorial, however, I get an UnsupportedOperationException on the Nexus S when I try to call this function so it's possible that this is not actually supported on the Android implementation. This isn't the end of the world since there is plenty you can do already with `GL_FUNC_ADD`.

The source factor and destination factor are set using the function `glBlendFunc()`. An overview of a few common blend factors will be given

below; more information as well as an enumeration of the different possible factors is available at the Khronos online manual:

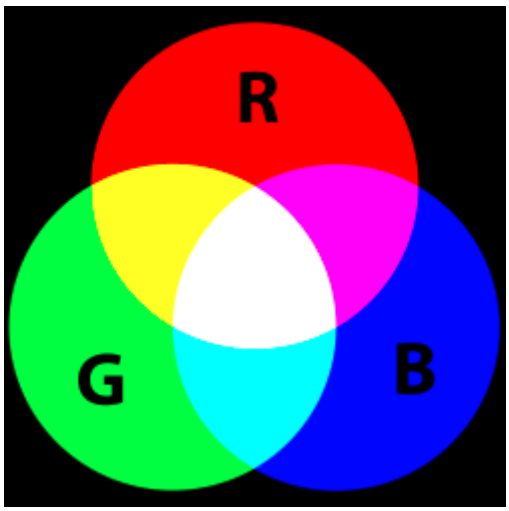
- glBlendFunc()
- glBlendEquation()

The documentation appears better in Firefox or if you have a MathML extension installed.

Clamping

OpenGL expects the input to be clamped in the range [0, 1] , and the output will also be clamped to the range [0, 1]. What this means in practice is that colors can shift in hue when you are doing blending. If you keep adding red (RGB = 1, 0, 0) to the frame buffer, the final color will stay red. However, if you add in just a little bit of green so that you are adding (RGB = 1, 0.1, 0) to the frame buffer, you will end up with yellow even though you started with a reddish hue! You can see this effect in the demo for this lesson when blending is turned on: the colors become oversaturated where different colors overlap.

DIFFERENT TYPES OF BLENDING AND HOW THEY RELATE TO DIFFERENT EFFECTS



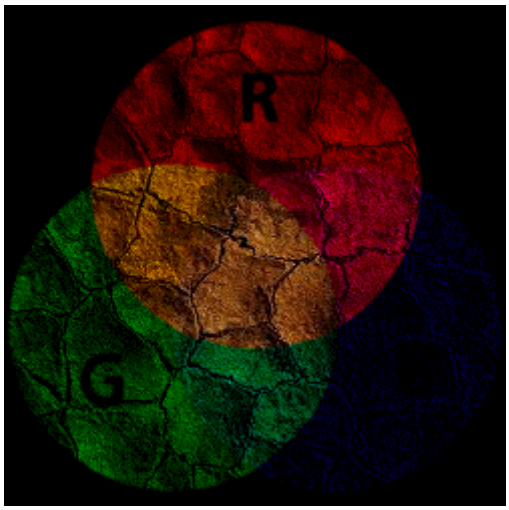
The RGB additive color model. Source: Wikipedia

Additive blending

Additive blending is the type of blending we do when we add different colors together and add the result. This is the way that our vision works together with light and this is how we can perceive millions of different colors on our monitors — they are really just blending three different primary colors together.

This type of blending has many uses in 3D rendering, such as in particle effects which appear to give off light or overlays such as the corona around a light, or a glow effect around a light saber.

Additive blending can be specified by calling `glBlendFunc(GL_ONE, GL_ONE)`. This results in the blending equation `output = (1 * source fragment) + (1 * destination fragment)`, which collapses into `output = source fragment + destination fragment`.



An example of lightmapping.

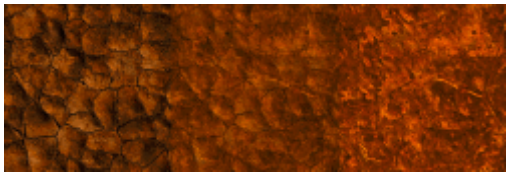
Multiplicative blending

Multiplicative blending (also known as modulation) is another useful blending mode that represents the way that light behaves when it passes through a color filter, or bounces off of a lit object and enters our eyes. A red object appears red to us because when white light strikes the object, blue and green light is absorbed. Only the red light is reflected back toward our eyes. In the example to

the left, we can see a surface that reflects some red and some green, but very little blue.

When multi-texturing is not available, multiplicative blending is used to implement lightmaps in games. The texture is multiplied by the lightmap in order to fill in the lit and shadowed areas.

Multiplicative blending can be specified by calling `glBlendFunc(GL_DST_COLOR, GL_ZERO)`. This results in the blending equation `output = (destination fragment * source fragment) + (0 * destination fragment)`, which collapses into `output = source fragment * destination fragment`.



An example of two textures interpolated together.

Interpolative blending

Interpolative blending combines multiplication and addition to give an interpolative effect. Unlike addition and modulation by themselves, this blending mode can also be draw-order dependent, so in some cases the results will only be correct if you draw the furthest translucent objects first, and then the closer ones afterwards. Even sorting wouldn't be perfect, since it's possible for triangles to overlap and intersect, but the resulting artifacts may be acceptable.

Interpolation is often useful to blend adjacent surfaces together, as well as do effects like tinted glass, or fade-in/fade-out. The image on the left shows two textures (textures from public domain textures) blended together using interpolation.

Interpolation is specified by calling `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`. This results in the blending equation `output =`

(source alpha * source fragment) + ((1 - source alpha) * destination fragment). Here's an example:

Imagine that we're drawing a green (0_r, 1_g, 0_b) object that is only 25% opaque. The object currently on the screen is red (1_r, 0_g, 0_b) .

output = (source factor * source fragment) + (destination factor * destination fragment)
output = (source alpha * source fragment) + ((1 - source alpha) * destination fragment)
output = (0.25 * (0_r, 1_g, 0_b)) + (0.75 * (1_r, 0_g, 0_b))
output = (0_r, 0.25_g, 0_b) + (0.75_r, 0_g, 0_b)
output = (0.75_r, 0.25_g, 0_b)

Notice that we don't make any reference to the destination alpha, so the frame buffer itself doesn't need an alpha channel, which gives us more bits for the color channels.

USING BLENDING

For our lesson, our demo will show the cubes as if they were emitters of light, using additive blending. Something that emits light doesn't need to be lit by other light sources, so there are no lights in this demo. I've also removed the texture, although it could have been neat to use one. The shader program for this lesson will be simple; we just need a shader that will pass out the color given to it.

Vertex shader

```
uniform mat4 u_MVPMatrix;               // A constant representing the
combined model/view/projection matrix.

attribute vec4 a_Position;              // Per-vertex position
information we will pass in.
```

```
attribute vec4 a_Color;                 // Per-vertex color information
we will pass in.

varying vec4 v_Color;                   // This will be passed into the
fragment shader.

// The entry point for our vertex shader.
void main()
{
        // Pass through the color.
        v_Color = a_Color;

        // gl_Position is a special variable used to store the final
position.
        // Multiply the vertex by the matrix to get the final point in
normalized screen coordinates.
        gl_Position = u_MVPMatrix * a_Position;
}
```

Fragment shader

```
precision mediump float;        // Set the default precision to medium.
We don't need as high of a
                                                                    //
precision in the fragment shader.
varying vec4 v_Color;           // This is the color from the vertex
shader interpolated across the
                                                                    //
triangle per fragment.

// The entry point for our fragment shader.
void main()
{
        // Pass through the color
    gl_FragColor = v_Color;
}
```

Turning blending on

Turning blending on is as simple as making these function calls:

```
// No culling of back faces
GLES20.glDisable(GLES20.GL_CULL_FACE);

// No depth testing
GLES20.glDisable(GLES20.GL_DEPTH_TEST);

// Enable blending
GLES20.glEnable(GLES20.GL_BLEND);
GLES20.glBlendFunc(GLES20.GL_ONE, GLES20.GL_ONE);
```

We turn off the culling of back faces, because if a cube is translucent, then we can now see the back sides of the cube. We should draw them otherwise it might look quite strange. We turn off depth testing for the same reason.

Listening to touch events, and acting on them

You'll notice when you run the demo that it's possible to turn blending on and off by tapping on the screen. See the article "Listening to Android Touch Events, and Acting on Them" for more information.

Further exercises

The demo only uses additive blending at the moment. Try changing it to interpolative blending and re-adding the lights and textures. Does the draw order matter if you're only drawing two translucent textures on a black background? When would it matter?

WRAPPING UP

The full source code for this lesson can be downloaded from the project site on GitHub.

A compiled version of the lesson can also be downloaded directly from the Android Market:

As always, please don't hesitate to leave feedbacks or comments, and thanks for stopping by!

Author: Admin

Kevin is the author of OpenGL ES 2 for Android: A Quick-Start Guide. He also has extensive experience in Android development. View all posts by Admin

Admin / October 3, 2011 / Android, Android Tutorials / additive blending, blending, GLSurfaceView, interpolation, Khronos, modulation, Nexus S, RGB color model, touch events

23 thoughts on "Android Lesson Five: An Introduction to Blending"

Mislav

October 27, 2011 at 12:30 am

Great tutorials, waiting for #6!!

DudeBro

November 22, 2011 at 1:05 am

Great tut as always, keep up the good work!!

---

steve

December 25, 2011 at 7:55 pm

Thanks for these great tuts, please include some transformation stuff in upcoming ones.

Driant

January 15, 2012 at 12:00 pm

These are awesome, I'm really happy to have stumbled upon your site, the opengl examples are just too well written 🙂 thank you very much, can't wait for the sixth tutorial

---

Admin

January 18, 2012 at 3:19 am

Thank you Driant!

Kyle

January 16, 2012 at 6:02 pm

Why are you using vertex arrays?! I came here for VBOs!

Admin

January 18, 2012 at 3:19 am

I think that parts of that API don't work on Froyo, but sure, why not.

Kyle

January 22, 2012 at 5:38 am

Froyo is kind of dying out, though. Gingerbread has been out for a while, and with ICS out on the market now, it's a lot more relevant.

Admin

March 9, 2012 at 2:46 pm

Hi Kyle,

My newest tutorial goes into VBOs. A future tutorial will also cover IBOs (index buffer objects). Hope you enjoy. 🙂

Doug

February 2, 2012 at 12:52 am

Great Tutorials, I am now looking for how to use a color texture, normal texture and a point light. I think I have it working, but I would love to see how you do it!

Keep up the great work.

Pingback: Listening to Android Touch Events, and Acting on Them | Learn OpenGL ES

Admin

March 9, 2012 at 3:39 pm

So thanks for all of the feedback, everyone! As more and more of the basics are covered I plan to go more in depth. 🙂

Fabien R

June 6, 2012 at 2:19 pm

Isn't there a typo in the section "Multiplicative blending" ?
output should be something like destination color * source fragment...

Great tutorial anyway 🙂

Admin

June 13, 2012 at 11:16 pm

Hi Fabien,

I actually can't see the typo — it looks good to me? At least, it did when I based it on the OpenGL blending equation at the time. 😉

Best,

Kevin

davor

January 22, 2013 at 10:59 pm

Hi,
thanks for great tutorial. I spent weeks learning opengl es 2.0, and this is the best tut I have found. These are my questions:

1. First I rendered sphere using compound light(ambient and diffuse), and I have got what I expected to. Then I rendered sphere using ambient light and blended it with the same sphere using diffuse light, only. The result was significantly worse than in first rendering, I guess that was a case due to z-fighting. How to improve the result of the second method, which I need to implement for volume shadow rendering?
2. (Little bit off-topic) I need to build app, for drawing sketches in visual studio. I would like to do it with opengl. What opengl x.y is the most similar to opengl es 2.0, so I can use most of I have learned in opengl es 2.0?
best regards.

Admin

January 30, 2013 at 3:16 pm

For 1, you can try a couple things:

1) Disable depth writes when you draw the first sphere. You can do that like this:

```
glDepthMask(false);
«draw»
glDepthMask(true);
```

2) Instead, change the depth test when you draw the second sphere, as follows:

```
glDepthFunc(GL_LEQUAL);
«draw»
glDepthFunc(GL_LESS);
```

You can also try playing around with glPolygonOffset()

For 2, I believe that OpenGL 2 and OpenGL 3 are the most similar to OpenGL ES 2.0, though you will have some differences. I think that there are some comparison charts at opengl.org and khronos.org which you could probably find with a quick search.

Pradeep

January 24, 2013 at 9:49 pm

Hi,

First of all thank you so much for posting about OpenGL2.0 for android. This site is helpful to me a lot.

I have a question for you. Could you post something on How to have layers and be able to draw on them and have scrolls for each of those layers with an example.

Thanks & Regards
Pradeep

Admin

January 30, 2013 at 3:18 pm

Good idea 🙂 I haven't covered much 2D stuff, so that could be interesting.

Jeff

November 16, 2013 at 3:09 am

Thanks, finally this equation makes sense. LOL previously I would cycle through every possibility of src, and dst blend combinations, and wait for the one that looked correct.

omgnooblike

February 25, 2014 at 9:33 am

Hello,

Can you create an explosion style particle effect? PLEASE :). Currently I am reading your book but the particle system there is like a fountain 🙂

Please.

Thanks

Caleb

April 5, 2014 at 2:44 am

How could you let cube emitters light，I can't find any code about that.
Could you share that?

Pingback: [Android]OpenGL Study | Dion's Note

Kapil Goyal

May 19, 2016 at 8:03 am

Hello,

How can apply blending for a single image with different colors.
I have an UIImage 100*100. i want apply four colors on a single UIImage with different blending mode.

like
0 to 25: Red color
25 to 50: Green color
50 to 75: Blue Color
75 to 100: Purple color

So these four color will show different effect on a single Image.

I have apply single color on a UIImage, but not getting success four colors at a time.

Please help me about it.

Reference app: ColorBurn IOS App

Thanks