

# Lecture on Alpha Transparency

## 1 Alpha

The “color” of an object or material can have a fourth component, called *alpha*. This is notated the RGBA system or, occasionally,  $RGB\alpha$ .

The alpha component has no fixed meaning, but we will see today what meaning it typically has, namely the *opacity* of the material:

- 1 is perfectly opaque
- 0 is perfectly transparent

To use RGBA, you have to initialize your OpenGL program as follows:

```
glutInitDisplayMode( GL_RGBA ... );
```

But since  $GL\_RGB$  is an alias for  $GL\_RGBA$ , we’ve been using it all along.

## 2 Blending

Given the pipeline model, we understand that at some moment during the rendering process, some of our objects have been drawn and exist only in the frame buffer and some of our objects have not yet been drawn. So, there is a time when the rendering of the next object is being *combined* with the rendering of some previous object. In the usual case, the new object’s pixels *overwrite* the old pixels.

In general, though, OpenGL allows you to *blend* the two sets of pixels in the following way. The pixels already in the frame buffer are known as the *destination* pixels and a particular pixel is colored  $(R_d, G_d, B_d, A_d)$ . The new pixels are called the *source* pixels and a particular one is colored  $(R_s, G_s, B_s, A_s)$ . You can choose the blending factors,  $s$  and  $d$  so that the combined color is computed as:

$$\begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix} = d \begin{bmatrix} R_d \\ G_d \\ B_d \\ A_d \end{bmatrix} + s \begin{bmatrix} R_s \\ G_s \\ B_s \\ A_s \end{bmatrix}$$

The result components are clamped to the range  $[0, 1]$ . The  $s$  and  $d$  factors are given to OpenGL using a constant from the following list, most of which we will ignore.

```
GL_ZERO  
GL_ONE  
GL_SRC_COLOR  
GL_ONE_MINUS_SRC_COLOR  
GL_SRC_ALPHA  
GL_ONE_MINUS_SRC_ALPHA  
GL_DST_ALPHA  
GL_ONE_MINUS_DST_ALPHA  
GL_DST_COLOR  
GL_ONE_MINUS_DST_COLOR  
GL_SRC_ALPHA_SATURATE  
GL_CONSTANT_COLOR  
GL_ONE_MINUS_CONSTANT_COLOR  
GL_CONSTANT_ALPHA  
GL_ONE_MINUS_CONSTANT_ALPHA
```

Note that any of these that need the destination ALPHA will require an ALPHA buffer.

You also need to use

```
glEnable(GL_BLEND);
glBlendFunc(source_factor, destination_factor)
```

The default blend function is

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

which just replaces (overwrites) the destination with the source.

See the man page for `glBlendFunc` for more information. However, we will quote one important sentence from that man page:

Transparency is best implemented using blend function (`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) with primitives sorted from farthest to nearest.

Let's try to understand the first half of that:

- We don't care what the alpha of the framebuffer (the destination) is. We only care about the opacity of the new object.
- We want to use a convex sum, so that things have the "right" relative weight (don't get progressively brighter or darker).

So, we use the alpha of the source object for the source factor and the complement for the destination. In the next section, we'll try to understand the second half of that sentence.

First, let's look at the `~cs307/public_html/demos/tutor.cc`. This lets you adjust the alpha values for three quads, drawn either furthest to nearest (in keeping with the advice from the man page) or in the reverse order.

Another demo is `~cs307/public_html/demos/TransparentQuads.cc`

- The red surface is opaque
- The green surface is slightly transparent
- The blue surface is mostly transparent
- You can toggle the background from black/white with the "b" key
- You can modify the transparency of the green and blue quads from 0.7/0.3 to 0.5/0.5 with the "M" key. (Note that it is not important that they add to 100 percent; I just wanted to start out with one that was mostly opaque and the other mostly transparent.)
- Notice the use of `glClearColor` and `glClear`. The first is like `glColor` except it sets the color used for clearing the framebuffer. (The TW default is to use 70 percent gray.) The second clears the frame buffer and associated buffers. We'll see about the depth buffer soon.
- Notice how the blending is defined.
- Notice how the colors are defined, including the alpha value.
- You can toggle the background color using 'b.' Notice the effect it has: the green and blue quads are mixed with the background color when they are drawn. Also, the red quad may look different: surrounding colors can affect our psychological perception of color. We'll see some of the effect of surrounding color on our perceptions of color by switching to "immerse" mode and comparing.

### 3 Hidden Surface Elimination

Suppose we render a scene with surfaces that overlap or even interpenetrate. For example:

- a blue teapot sitting on a brown table. Some pixels in the framebuffer are "both" blue and brown.
- a teddy bear (its ears and nose are spheres penetrate its head)
- a teddy bear with a knife in its guts

How does a graphics system determine *which* color to use for any pixel? There are two major algorithms: depth sort, which is *object*-based, and depth buffer, which is *pixel*-based.

### 3.1 Depth Sort

Determine which object is farthest from the camera, draw that first, then the next, and so forth. Since the nearer stuff always overwrites the farther stuff, this works well. But:

- We would have to draw things in different orders depending on the position of the camera.
- What about objects that inter-penetrate. How do you handle that?

Sometimes, we break up objects into smaller pieces, just so that we can sort them by distance. If we take that to its logical extreme, and re-organize our thinking, we come to the next algorithm.

### 3.2 Depth Buffer

For each pixel, keep track of the depth of that pixel. (This buffer needs to be initialized to some maximum at the beginning of rendering.) Whenever we consider drawing a pixel, first compute the new depth and compare to the old depth (looking it up in the depth buffer). If the new depth is *less*, update the color buffer and the depth buffer. Computing the depth is easy, because we have the original  $(x, y, z)$  coordinates of the object at the beginning of the transformation process, and we maintain all of them to the end.

### 3.3 Depth in OpenGL

OpenGL uses the depth buffer algorithm, AKA the  $z$  buffer algorithm. You use it, you have to

```
glutInitDisplayMode(... | GLUT_DEPTH);  
glEnable(GL_DEPTH_TEST);
```

TW does the latter for you. In plain OpenGL, you have to remember to enable the depth test.

Then, in your display function, you have to:

```
glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

This is what initializes the buffer to the maximum value (1.0).

Finally, the depth buffer is only updated if `DepthMask` is true. This is the default value, but you can temporarily turn it off with:

```
glDepthMask(GL_FALSE);
```

A simple and useful demo is `demos/transparency/SameDepth`, which draws two quads that occupy the same space. By “occupy the same space,” I don’t mean just that their 2D *projections* overlap, I mean that in the 3D world, their volumes overlap. Because OpenGL retains depth information through projection (that’s the “ $z$  coordinate”), if projections overlap, it can still tell which one is “in front.” However, if the volumes coincide, it can’t tell which one is in front because, in fact, neither one is. Therefore, if the depth test is enabled, OpenGL will make the decision based on the depth buffer, where tiny roundoff errors may differ from pixel to pixel, so that sometimes it decides that the red one is “in front” and sometimes the green one. Thus, we get a speckling effect. If you turn off the depth test, the second quad (the one drawn later) always wins.

## 4 Depth and Transparency

The depth buffer algorithm has real trouble with transparency. Why?

If you update the depth buffer when you draw a transparent object, then an opaque object that is drawn *later* but is *farther* won’t be drawn.

Let’s pause to make sure we understand that. Let’s also return to the demo.

Let’s go back to the `TransparentQuads` demo and look at the effect of depth mask and depth test.

- If the depth test is OFF, notice what happens when the green quad overlaps the red one. Where the green and blue quads pass behind the red one and should be invisible, they are still visible. This is the effect of the (lack of the) depth test. We would want to use the depth test for drawing opaque objects, so that only the nearest is visible.

Notice that the area where the green and blue quads overlap is a blend of both the green and blue, and the two halves (green behind blue on the right and blue behind green on the left) should look identical. This is because if we ignore depth, we're just mixing green and blue, and it doesn't matter which is SRC and which is DEST.

Notice that if the depth test is off, the depth mask makes no difference. This makes sense, since the depth buffer is ignored, so it doesn't matter whether you update it.

So, it seems that the depth test should be on.

- If the depth test is ON,
  - If the depth mask is TRUE, the fact that the blue quad passes behind the green one is noted: we only see the right half of the overlap area, because in the left half, the blue quad is *behind* the green one, and so we don't see it at all. This is the effect of the *z*-buffer algorithm. So, even though the green quad is partly transparent, we can't see *any* blue through it. That's bad.
 

Try finding a view where you can compare red through green with red through blue through green. They look the same! Should they?
  - If the depth mask is FALSE, the blue and green colors mix where the projections overlap. That's what we want. However, since the depth buffer isn't being updated, OpenGL can't properly interleave this surface with others.

Solutions:

- don't update the depth buffer. You can turn off updating, temporarily, with

```
glDepthMask(GL_FALSE);
```

- Draw all the opaque objects first, then disable the depth test and draw all the transparent ones, from furthest to nearest (switching to the other hidden-surface algorithm). The coding scheme is shown in figure 1.

```
glEnable(GL_DEPTH_TEST);
glDepthMask(GL_TRUE);          // this is the default
glDisable(GL_BLEND);

// draw all opaque Objects
...

// done

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
glDepthMask(GL_FALSE);
// draw all transparent Objects

// done
glutSwapBuffers();
```

Figure 1: Code that draws the opaque objects first

Compare these. In particular, when Opaque objects are drawn last, try it with and without the depth test. It just doesn't work.

**Demos:** `~cs307/public_html/demos/transparency/tutor` With this demo,

- the squares are all at the same distance
- they are drawn in order from lower left (first) to upper right (last)
- Try the following:
  - bottom (0.5, 0.5, 0, 1)
  - middle (0,1,0,0.5)
  - top (1,0,0,0.5)

Compare. Does this make sense?

- Change the middle to (0,1,0,1). Compare. Does this make sense?

You can use this tutor to experiment with different combinations of RGBA values.

## 5 Depth Resolution

There are a limited number of bits in the depth buffer; the actual number depends on the graphics card. Quoting from the OpenGL Reference Manual page for `gluPerspective`:

Depth buffer precision is affected by the values specified by  $zNear$  and  $zfar$ . The greater the ratio of  $zFar$  to  $zNear$  is, the less effective the depth buffer will be at distinguishing between surfaces that are near each other. If

$$r = zFar/zNear$$

roughly  $\log_2 r$  bits of depth buffer precision are lost. Because  $r$  approaches infinity as  $zNear$  approaches 0,  $zNear$  must never be set to 0.

So, even though it seems realistic to set *near* to zero and *far* to infinity, the practical result is that the depth buffer algorithm won't be easily able to tell which of two surfaces is closer if they are similar in distance. Some of you have discovered that if they are exactly the same distance, the color can be almost random, based on small roundoff errors in the calculation.