

CS-4470
Image Synthesis

Part 3

Blending

Notes and Programs by:

Dr. Michael J. Zyda

Blending 3
Blending - The Source and Destination Values 4
Blending - The Blending Function 5
Sample Uses of Blending 6
3D Blending with the Depth Buffer 9
3D Blending with the Depth Buffer - Solution 10

Blending

- Alpha values have been part of our color specifications with `glColor*()` and `glClearColor()` and part of our material specifications but we have not yet discussed how they are used.
- When blending is enabled, the alpha value is used to combine the color value of the fragment being processed with that of the pixel already stored in the framebuffer.
- Blending occurs after the scene has been rasterized and converted to fragments, but just before the final pixels are drawn in the framebuffer.
- Alpha values can also be used in the alpha test to accept or reject a fragment based on its alpha value.
- Without blending, each new fragment overwrites any existing color values in the framebuffer, as though the fragment is opaque.
- With blending, you can control how much of the existing color value should be combined with the new fragment's value.
- Thus you can use alpha blending to create a translucent fragment, one that lets some of the previously stored color value "show through".
- Color blending lies at the heart of techniques such as transparency, digital compositing, and painting.
- The most natural way to think about of blending operations is to view the RGB components of a fragment as representing its color, and the alpha component as representing opacity.
- Thus transparent or translucent surfaces have lower opacity than opaque ones.
- Imagine viewing an object through green glass - the color you see is partly green from the glass and partly the color of the object - the percentage varies depending on the transmission properties of the glass.
- We could say that the glass transmits 80% of the light that strikes it so that the color we see is 20% of the glass color and 80% of the color of the object behind the glass.

Blending - The Source and Destination Values

-- During blending, color values of the incoming fragment (the *source*) are combined with the color values of the corresponding currently stored pixel (the *destination*) in a two-stage process.

-- First, you specify how to compute source and destination factors.

-- These factors are RGBA quadruplets that are multiplied by each component of the RGBA values in the source and destination, respectively.

-- Then, the corresponding components in the two sets of RGBA quadruplets are added.

-- Let's look at this mathematically...

-- Let (Sr, Sg, Sb, Sa) be the source blending factor.

-- Let (Dr, Dg, Db, Da) be the destination blending factor.

-- Let (Rs, Gs, Bs, As) be the RGBA of the source.

-- Let (Rd, Gd, Bd, Ad) be the RGBA of the destination.

-- Then the final, blended RGBA values are given by:

$$(RsSr + RdDr, GsSg + GdDg, BsSb + BdDb, AsSa + AdDa)$$

-- Each component of this quadruplet is clamped to [0, 1].

-- Now let's look at how the source and destination blending factors (S & D) are generated.

-- We use `glBlendFunc()` to supply two constants: one that specifies how the source factor should be computed and one that indicates how the destination factor should be computed.

-- We also need to enable blending: `glEnable(GL_BLEND)`.

Blending - The Blending Function

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
```

-- Controls how color values in the fragment being processed (the source) are combined with those already stored in the framebuffer (the destination).

-- The argument sfactor indicates how to compute a source blending factor;

-- dfactor indicates how to compute a destination blending factor.

-- See the table for possible values.

-- The blend factors are assumed to lie in the range [0, 1]; after the color values in the source and destination are combined, they're clamped to the range [0, 1].

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(Rd, Gd, Bd, Ad)
GL_SRC_COLOR	destination	(Rs, Gs, Bs, As)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1) - (Rs, Gs, Bs, As)
GL_SRC_ALPHA	source or destination	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1, 1, 1, 1) - (As, As, As, As)
GL_DST_ALPHA	source or destination	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPHA	source or destination	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f = min(As, 1-Ad)

Source and Destination Blending Factors

Sample Uses of Blending

-- Not all of the combinations of source and destination factors make sense.

-- The majority of applications use a small number of combinations.

-- The following describe a few typical uses for particular combinations of the source and destination factors.

-- Some of these examples use only the incoming alpha value, so they work even when alpha values aren't stored in the framebuffer.

1. One way to draw a picture composed half of one image and half of another, equally blended, is to set the source factor to `GL_ONE`, draw the first image, then set the source and destination factors to `GL_SRC_ALPHA`, and draw the second image with alpha equal to 0.5 (remember, this means to set the alpha values for ALL the pixels in the second image to 0.5).

-- If the picture is supposed to be blended with 0.75 of the first image and 0.25 of the second, draw the first image as before, and draw the second image with an alpha of 0.25, but with `GL_SRC_ALPHA` (source) and `GL_ONE_MINUS_SRC_ALPHA` (destination). This pair of factors probably represents the most commonly used blending operation.

-- Note: When we wrote the first image with `GL_ONE`, that set all the alpha planes in the destination to 1.0.

2. To blend different images equally, set the destination factor to `GL_ONE`, and the source factor to `GL_SRC_ALPHA`. Draw each of the images with an alpha equal to 0.3333333.

-- With this technique, each image is only one-third of its original brightness, which is noticeable where the images don't overlap.

3. Suppose you're writing a paint program, and you want to have a brush that gradually adds color so that each brush stroke blends in a little more color with whatever is currently in the image (say 10% color with 90% image on each pass).

-- To do this, draw the image of the brush with alpha of 10% and use `GL_SRC_ALPHA` (source) and `GL_ONE_MINUS_SRC_ALPHA` (destination).

-- Note: you can vary the alphas across the brush to make the brush add more of its color in the middle and less on the edges, for an antialiased brush shape. Similarly, erasers can be implemented by setting the eraser color to the background color.

Sample Uses of Blending continued

4. The blending functions that use the source and destination colors - `GL_DST_COLOR` or `GL_ONE_MINUS_DST_COLOR` for the source factor and `GL_SRC_COLOR` or `GL_ONE_MINUS_SRC_COLOR` for the destination factor - effectively allow you to modulate each color component individually.

-- This operation is equivalent to applying a simple filter - for example, multiplying the red component by 80%, the green component by 40%, and the blue component by 72% would simulate viewing the scene through photographic filter that blocks 20% of the red light, 60% of the green, and 28% of the blue.

-- Note: Basically the idea is that the image we wish to filter is already in the destination planes - we filter that image say using a specially set up second image...

5. Suppose you want to draw a picture composed of three translucent surfaces, some obscuring others and all over a solid background. Assume the farthest surface transmits 80% of the color behind it, the next transmits 40%, and the closest transmits 90%.

-- To compose this picture, draw the background first with the default source and destination factors (`GL_ONE`, `GL_ZERO`), and then change the blending factors to `GL_SRC_ALPHA` (source) and `GL_ONE_MINUS_SRC_ALPHA` (destination). Next, draw the farthest surface with an alpha of 0.2, then the middle surface with an alpha of 0.6, and finally the closest surface with an alpha of 0.1.

-- Note: Please note the required ordering of the surfaces! Z-buffering will NOT take care of this. the ordering is necessary.

6. If your system has alpha planes, you can render objects one at a time (including their alpha values), read them back, and then perform interesting matting or compositing operations with the fully rendered objects.

-- See "Compositing 3D Rendered Images" by Tom Duff, SIGGRAPH '85 Proceedings, pp. 41- 44. for examples of this technique.

-- Note that objects used for picture composition can come from any source - they can be rendered using OpenGL commands, rendered using other techniques such as ray tracing or radiosity that are implemented in another graphics library...

Sample Uses of Blending continued

7. You can create the effect of a nonrectangular raster image by assigning different alpha values to individual fragments in the image. Assign an alpha of 0 to each "invisible" fragment and an alpha of 1.0 to each opaque fragment.

-- For example, you can draw a polygon in the shape of a tree and apply a texture map of foliage; the viewer can see through parts of the rectangular texture that aren't part of the tree if you've assigned them alpha values of 0.

-- This method, sometimes called *billboarding*, is much faster than creating the tree out of 3D polygons. This, combined with rotating the billboard about its center so that its always facing the viewer, makes quite a believable tree.

3D Blending with the Depth Buffer

- As stated previously, the order in which polygons are drawn greatly affects the blended result.
- When drawing 3D translucent objects, you can get different appearances depending on whether you draw the polygons from back to front or front to back.
- You also need to consider the effect of the depth buffer (z-buffer) when determining the correct order.
- Remember, the z-buffer keeps track of the distance between the viewpoint and the portion of the object occupying a given pixel in a window on the screen; when another candidate color arrives for that pixel, its drawn only if the object is closer to the viewpoint, in which case its depth value is stored in the depth buffer.
 - With this method, obscured portions of surfaces aren't necessarily drawn and therefore aren't used for blending.
- We normally want to render both opaque and translucent objects in the same scene, and you want to use the depth buffer to perform hidden surface removal for objects that lie behind the opaque objects.
 - If an opaque object hides either a translucent object or another opaque object, you want the depth buffer to eliminate the more distant objects.
 - If the translucent object is closer, however, you want to blend it with the opaque object.
 - If everything in the scene is stationary, it is pretty easy to compute the correct drawing order for the polygons but the problem becomes too hard if either the viewpoint or the object is moving.

3D Blending with the Depth Buffer - Solution

-- The solution is to enable depth-buffering but make the depth buffer read-only while drawing the translucent objects.

-- First draw all the opaque objects, with the depth buffer in normal operation.

-- Then preserve these depth values by making the depth buffer read-only.

-- When the translucent objects are drawn, their depth values are still compared to the values established by the opaque objects, so they aren't drawn if they're behind the opaque ones.

-- If they're closer to the viewpoint, however, they don't eliminate the opaque objects, since the depth buffer values can't change. Instead, they're blended with the opaque objects.

-- To control whether the depth buffer is writable, use `glDepthMask()`; if you pass `GL_FALSE` as the argument, the buffer becomes read-only, whereas `GL_TRUE` restores the normal, writable operations.

-- Note: this solution method is exact only when no pixel in the framebuffer is drawn more than once by a transparent object. If transparent objects overlap, resulting in multiple blended renderings to individual pixels, this solution is only a useful approximation to the correct (sorted) result.

-- Sketch Of Program:

In `initCB`:

```
glDepthFunc(GL_LEQUAL);  
glEnable(GL_DEPTH_TEST);
```

...

In display loop:

Draw solid objects with blending off.

```
glEnable(GL_BLEND);  
glDepthMask(GL_FALSE);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

Draw translucent stuff.

```
glDepthMask(GL_TRUE);  
glDisable(GL_BLEND);
```