

COMP3201 – Computer Graphics

3. Module 3: Realism and Performance

3.2 Blending and Anti-Aliasing

3.2.1 Blending

Normally when a fragment (proto-pixel) comes to the framebuffer it replaces the colour of the current pixel with the new value. (If depth testing is enabled it does a comparison of the z values and makes a choice to replace or not.) Blending, however, is the process of combining the colour of the pixel in the framebuffer with the colour of the incoming fragment. The final outcome of the combining of the colours is dependent on each of the alpha values and on the choice of blending function.

The colour of a fragment is its RGB value and the opacity of a fragment is its A (alpha) value. An alpha value of zero means that the fragment is totally transparent. An alpha value of one means that the fragment is totally opaque.

Blending is harder to understand than to code. There are three easy steps to including blending in an application:

1. Disable depth buffering. `glDisable(GL_DEPTH_TEST);` See the section on troubleshooting for further explanation of this point.
2. Enable blending with `glEnable(GL_BLEND);`
3. Specify the blending functions (which are internally used to produce scaling vectors).

And voila, you have blending.

When blending is enabled OpenGL calculates the final colour of the pixel in the following manner. Let (I_r, I_g, I_b, I_a) represent the colour of the incoming fragment (also called the source fragment) and let (C_r, C_g, C_b, C_a) represent the current pixel colour (also called the destination fragment). The choice of blending function specifies how to calculate a scaling vector for both the incoming fragment and current pixel. For the moment we will ignore the specifics of the blending function and just work with the output of the blending function, i.e. the scaling factors. Denote the incoming scaling vector by (SI_r, SI_g, SI_b, SI_a) and the current pixel scaling vector by (SC_r, SC_g, SC_b, SC_a) . The final pixel colour is denoted by

$$\begin{pmatrix} I_r SI_r + C_r SC_r \\ I_g SI_g + C_g SC_g \\ I_b SI_b + C_b SC_b \\ I_a SI_a + C_a SC_a \end{pmatrix}$$

Each of the components of the final colour is restricted to the range [0,1]. Note the addition in the above calculation. Some implementations of OpenGL allow a choice of operation here with the default being addition.

The blending functions are specified by using

```
void glBlendFunc( GLenum ifunc, GLenum cfunc);
```

where ifunc and cfunc enumerated constants are chosen from the following tables.

Enumerated constant	Computed scaling vector
GL_ZERO	(0, 0, 0, 0)
GL_ONE	(1, 1, 1, 1)
GL_SRC_ALPHA	(I _a , I _a , I _a , I _a)
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (I _a , I _a , I _a , I _a)
GL_DST_ALPHA	(C _a , C _a , C _a , C _a)
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (C _a , C _a , C _a , C _a)
GL_CONSTANT_COLOR	(R, G, B, A)
GL_ONE_MINUS_CONSTANT_COLOR	(1, 1, 1, 1) - (R, G, B, A)
GL_CONSTANT_ALPHA	(A, A, A, A)
GL_ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1, 1) - (A, A, A, A)

Table 1: Both ifunc and cfunc can be chosen from this table.

Enumerated constant	Computed scaling vector
GL_DST_COLOUR	(C _r , C _g , C _b , C _a)
GL_ONE_MINUS_DST_COLOUR	(1, 1, 1, 1) - (C _r , C _g , C _b , C _a)
GL_SRC_ALPHA_SATURATE	(s, s, s, s) : s=min(I _a , 1-C _a)

Table 2: Specific ifunc blending functions

Enumerated constant	Computed scaling vector
GL_SRC_COLOUR	(I _r , I _g , I _b , I _a)
GL_ONE_MINUS_SRC_COLOUR	(1, 1, 1, 1) - (I _r , I _g , I _b , I _a)

Table 3: Specific cfunc blending functions

3.2.2 Uses of Blending

There are often many ways of achieving the same effect so the following list is by no means exhaustive. It is also interesting to note that some of the techniques listed below only depend on the incoming fragment (source) alpha. These source-only techniques can also be used when the framebuffer isn't storing alpha values (i.e. when it is in RGB mode and not RGBA mode).

3.2.2.1 To draw a picture which is an equal blend of two completely overlapping images [REDBOOK]

Set the source function to `GL_ONE`, the destination function to `GL_ZERO` and draw the first image. Now set the source function to `GL_SRC_ALPHA`, the destination function to `GL_ONE_MINUS_SRC_ALPHA`, and draw the second image with all its alphas set to 0.5. If the desired picture is an unequal blend of the two images then set the alpha of the second image to the percentage that you wish it to be of the final image. This tip can be extended to more than two images.

3.2.2.2 Incremental paint [REDBOOK]

Suppose that you're writing a paint program and you want to have a brush that gradually adds colour so that each stroke blends in a bit more colour with whatever is currently in the image. To do this, draw the image of the brush with an alpha of say 10% and use `GL_SRC_ALPHA` for the source function and use `GL_ONE_MINUS_SRC_ALPHA` for the destination function. Note that the alpha value can be varied across the brush to achieve an antialiasing effect. Similarly erasers can be implemented by setting the eraser colour to the background colour.

3.2.2.3 Complex holey images (e.g. trees) [REDBOOK]

Creating a complex image which has see-through holes (e.g. a tree) by construction with polygons is a computationally intensive task. A simpler way of achieving the same effect is to draw a simple polygon in the approximate shape of the tree and then apply a texture (textures are discussed in a later lecture). The parts of the rectangular texture that we don't want to see (e.g. gaps in the foliage) are assigned an alpha of zero, and the rest is assigned an alpha of one.

3.2.2.4 Glass [OpenGL FAQ]

First render all opaque objects in your scene. Disable lighting, enable blending, and render your glass geometry with a small alpha value. This should result in a faint rendering of your object in the framebuffer. (Note: You may need to sort your glass geometry, so it's rendered in back to front Z order.)

Now, you need to add the specular highlight. Set your ambient and diffuse material colors to black, and your specular material and light colors to white. Enable lighting. Set `glDepthFunc(GL_EQUAL)`, then render your glass object a second time.

3.2.2.5 Screen door transparency [OpenGL FAQ]

You don't want to use OpenGL blending for this. Screen door transparency is best achieved by specifying a polygon stipple pattern with `glPolygonStipple()` and by rendering the transparent primitive with polygon stippling enabled (`glEnable(GL_POLYGON_STIPPLE)`). The number of bits set in the stipple pattern determines the amount of translucency and opacity; setting more bits results in a more opaque object, and setting fewer bits results in a more translucent object. Screendoor

transparency is sometimes preferable to blending, because it's order independent (primitives don't need to be rendered in back-to-front order). Polygon stippling is part of the fourth year Advanced Computer Graphics course.

3.2.2.6 In place scale and bias (i.e. brightness and contrast) [Kilgard]

A common computer graphics and image processing operation is to scale and/or bias components in an image. A scale & bias operation lets an application quickly adjust the contrast and/or brightness of an image.

OpenGL's pixel path has scale & bias parameters that permit an application to scale & bias drawn, copied, or read back pixels rectangles as well as textures during download. See `glPixelTransfer`.

It is often useful to do an "in place" scale & bias of an image already rendered in the framebuffer. This can be useful to perform a scale & bias *after* texture filtering (OpenGL's pixel path scale & bias is before texture filtering). Post-texture filtering scale & bias is often needed by volume rendering and image processing applications. A post-texture filtering scale & bias can also be used by mainstream 3D applications to easily change the contrast & brightness of textured geometry without having to re-download (and re-filter) the texture. For example, a texture mapped spaceship might brighten briefly when it absorbs power from a fuel cell.

A straightforward means to implement an "in place" scale & bias is with an "in place" `glCopyPixels` to scale & bias the designated pixels. This ends up being rather expensive however since many low-end OpenGL implementations do not fully optimize this path and it involves streaming pixels out of the framebuffer to then be immediately written back into the framebuffer. This degree of data movement is expensive.

An alternative approach is to use OpenGL's blending capabilities to perform the "in place" scale & bias as blended rendering. Many low-end hardware accelerators fully implement OpenGL's blending modes in hardware.

Here's a simple example. Say you want to scale & bias a rectangular region of the frame buffer. The desired scale factor is 0.7 and the desired bias factor is 0.1. This can be done like this:

```
glBlendFunc(GL_ONE, GL_SRC_ALPHA);
glColor4f(0.1, 0.1, 0.1, 0.7);
glRect(x, y, x + width, y + height);
```

This ends up computing:

```
REDnew = 0.1 * 1 + 0.7 * REDprevious
GREENnew = 0.1 * 1 + 0.7 * GREENprevious
BLUEnew = 0.1 * 1 + 0.7 * BLUEprevious
```

This turns out to perform exactly the intended "in place" scale & bias operation. The expense is no more than rendering a blended rectangle, substantially cheaper than a glCopyPixels. On an O2, this blending approach can be 4 to 25 times faster depending on how many pixels are affected and the particular scale & bias parameters.

Notice that you can actually scale & bias an arbitrary region in the frame buffer, not simply a rectangular region with this technique by drawing triangles instead of a single rectangle. OpenGL stencil can make sure you don't scale & bias a particular pixel multiple times.

This blending approach becomes more complicated when the scale factor exceeds 1.0, but this can be handled with $\log_2(n)$ extra passes where n is the integer portion of the scale factor. Another problem is how to handle a negative bias. An EXT OpenGL extension called the "blend subtract" extension makes it possible to efficiently handle the negative bias case. Machines like SGI's O2 workstation implement the "blend subtract" extension in hardware making a negative bias just as fast as a positive bias.

3.2.2.7 Antialiasing

This is the subject of section 3.2.4.

3.2.3 Troubleshooting

The most common cause of blending "not working" is the depth buffer. Both blending and the depth test are fragment operations and the depth test occurs before blending. When depth testing is enabled only one fragment makes it through to the blending stage (which does rather simplify the blending operation :). The cure is to either do without the depth buffer or to temporarily disable the depth buffer.

3.2.4 Antialiasing

Almost all modern displays are made up of small square pixels. This means that horizontal and vertical straight lines are displayed as (where an x represents a pixel turned on)

```

                                     x
                                     x
                                     x
                                     x
                                     x
                                     x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  x
```

which is visually acceptable. The problem arises when lines at angles in between those just discussed are drawn. These lines cannot be exactly rendered by square pixels so the closest approximation is used. For example

```

XXXXXXXXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXX

```

These sharp discontinuities are known as aliasing, or jaggies. The technique for removing these jaggies is called antialiasing.

Antialiasing works (unintuitively) by blurring the object. This blurring makes the scene much more visually acceptable. The details of this are as follows. The ideal object would like to cover only a fraction of certain pixels rather than having to make a choice between on or off. Although we can't do fractions of pixels, blending provides a mechanism to allow partial coverage of a pixel. If the ideal object touches a pixel then the pixel will be given the object's colour but the alpha assigned to the colour is the fraction of the pixel that the object should be covering. This RGBA value for the pixel (representing the object) is then blended with the existing colour of the pixel.

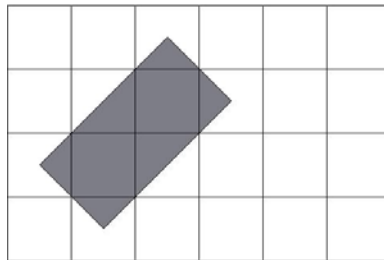


Figure 1: Ideal fat line on coarse square pixels

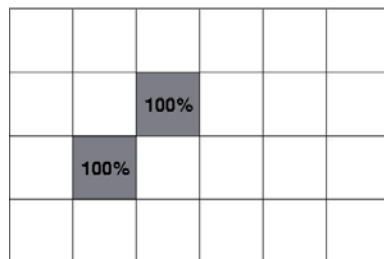


Figure 2: Drawing of the fat line without antialiasing

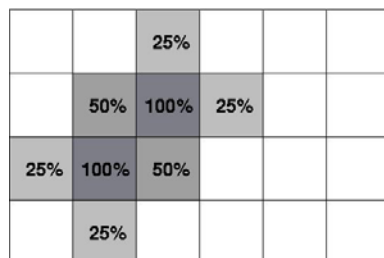


Figure 3: Drawing of the fat line with antialiasing

3.2.4.1 Point and Line Antialiasing

There are three steps to achieving point and line antialiasing in OpenGL

1. Enable the calculations with `glEnable(GL_POINT_SMOOTH);` and `glEnable(GL_LINE_SMOOTH);`
2. (Optionally) Give a hint on how the OpenGL engine should do the calculation. Hints are given by

```
void glHint( GLenum target, GLenum hint );
```

where `target` is either `GL_POINT_SMOOTH_HINT` or `GL_LINE_SMOOTH_HINT` and `hint` is one of `{GL_NICEST, GL_FASTEST, (default =) GL_DONT_CARE}`. `GL_NICEST` gives the best quality antialiasing whereas `GL_FASTEST` obviously gives the fastest implementation at the expense of quality.

3. Enable blending and specify the blending functions. Usually the blending functions are set to be `GL_SRC_ALPHA` for the source and `GL_ONE_MINUS_SRC_ALPHA` for the destination. If you want the intersection of lines to be brighter try using `GL_ONE` for the destination function.

3.2.4.2 Polygon Antialiasing

Polygon antialiasing is only necessary if polygons are rendered in `GL_FILL` mode. If the polygon is rendered in `GL_POINT` or `GL_LINE` mode then the usual point and line antialiasing are applied.

Obviously, polygon antialiasing is only necessary on the edges of the polygon.

There are five steps to achieving polygon antialiasing in OpenGL

1. Enable the calculation with `glEnable(GL_POLYGON_SMOOTH);`
2. (Optionally) Give a hint on how the OpenGL engine should do the calculation. Hints are given by

```
void glHint( GLenum target, GLenum hint );
```

where `target` is `GL_POLYGON_SMOOTH_HINT` and `hint` is one of `{GL_NICEST, GL_FASTEST, (default =) GL_DONT_CARE}`. `GL_NICEST` gives the best quality antialiasing whereas `GL_FASTEST` obviously gives the fastest implementation at the expense of quality.

3. Disable depth buffering.

4. Enable blending and specify the blending functions. Usually the blending functions are set to be `GL_SRC_ALPHA_SATURATE` for the source and `GL_ONE` for the destination. With these blending functions the final colour is the sum of the destination colour and the scaled source colour. Remember that for this choice of source function the scaling factor is the minimum of the incoming alpha and one minus the current alpha value. Note that for large alpha values one minus the destination alpha is almost zero. Thus for a pixel that has a large alpha value, further incoming fragments have little effect on the final colour.
5. Order the polygons from front to back and draw them.

3.2.4.3 Full Scene Antialiasing

The implementation of full scene antialiasing is computationally more expensive than doing the other types of antialiasing. It also involves using the accumulation buffer which isn't covered in this course. The advantage of using FSAA is that it provides antialiasing without having to do depth ordering first.

3.2.5 References

[REDBOOK] “The OpenGL Programming Guide” Third edition, Woo *et. al.* p224–5

[OpenGL FAQ] <http://www.frii.com/~martz/oglfaq/transparency.htm>

[Kilgard] Mark J. Kilgard

<http://www.opengl.org/developers/code/mjktips/InPlaceScaleAndBias.html>