

COMP3201 – Computer Graphics

Module 2: Transformations and Scene Creation

2.8 Rendering

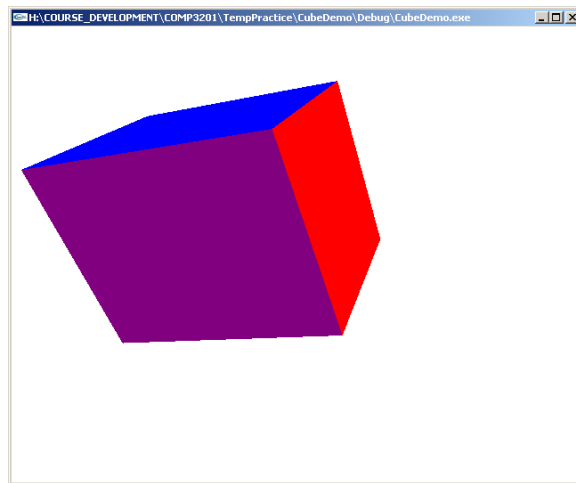
2.8.1 Introduction

This section covers some of the issues in rendering a scene. In particular, we look at hidden surfaces, depth buffering and culling.

2.8.2 Hidden-Surfaces

Hidden-surface removal is performed to create greater realism in the scene being rendered. An associated problem is that of hidden-line removal, where edges behind others should not be drawn.

Consider, for example, a solid cube that has 6 faces. All these faces have been specified in drawing the cube, and all are sent down the graphics pipeline to be rendered. But depending on our viewpoint, we should only see three of these faces at any one time. Hidden surface removal is the technique of not rendering the faces that shouldn't be visible.



One step in solving the problem of hidden surfaces is to eliminate all back-facing polygons – this is called culling and is described later. Another technique is depth-buffering, to work out which objects are deeper in the picture and hence are obscured.

Another approach is to order all objects according to their distance from the viewer. This is called the Painter's Algorithm; the objects are then rendered from furthest-away to closest. Closer objects are drawn on top of ones further away, and so this algorithm is computationally intensive. Another problem that can arise with this algorithm is when two objects intersect at an angle – then part of the object is in front while the rest of the object is behind; the Painter's Algorithm does not render such a scene correctly.

2.8.3 Depth-Buffering

A widely used hidden-surface removal algorithm is the z-buffer algorithm. In this section we investigate in detail the z-buffer algorithm, the effects of perspective projections on the operation of the z-buffer, and using depth buffering in OpenGL.

2.8.3.1 The z-buffer algorithm

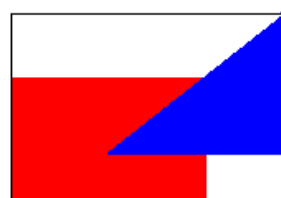
The viewport transformation maps the normalized device coordinates to the screen coordinates. As well as the (x,y) screen coordinates, the viewport transformation also calculates a depth coordinate value. This depth coordinate represents the distance from the view plane to the object. The depth coordinate is clamped to the range [0,1] where 0 represents the near clipping plane and 1 represents the far clipping plane.

The rasterisation step in the OpenGL pipeline converts each object into the set of pixels that the object will occupy on the screen. Each individual pixel has two values for the screen coordinates as well as a depth value. The depth value is generated by interpolating between the known depth coordinates for the vertices of the object that is being rasterised.

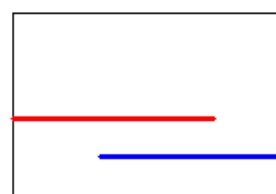
The z-buffer (also called the depth buffer) is similar to the colour buffer in that there is a position for every pixel. This position will hold the current depth value for that pixel. The depth buffer is usually initialised at the start of every frame. The initialisation value is the maximum depth value.

If depth buffering has *not* been enabled then the pixel that has just been calculated in the rasterisation step will now be written into the colour buffer. If depth buffering has been enabled then the depth value of the incoming fragment (proto-pixel) is compared to the depth value currently recorded for the target pixel. If the incoming fragment has a depth value less than the current value, then the incoming fragment represents a point closer to the view plane and so the colour buffer value is updated to the incoming fragment colour and we replace the depth buffer value with the incoming depth value. If the incoming fragment has a depth value greater than the current value, then the incoming fragment is dropped and the colour buffer and depth buffer are left in their current state.

Let us consider the following example. Suppose we wish to draw a rectangle and a triangle, where the triangle is in front of the rectangle.



Front View



Top View

Now examine the colour and depth buffers for an extremely low resolution display (only 5x5 pixels) as we execute the following pseudo-code.

- Clear both of the buffers (choose black as the colour buffer clear value).
- Draw the triangle.
- Draw the rectangle.

If there was no depth buffering, the above pseudo-code would draw the rectangle on top of the triangle simply because the rectangle is drawn last. However with depth buffering the following occurs.

- Clear both of the buffers

| Depth Buffer | | | | |
|--------------|-----|-----|-----|-----|
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

| Colour Buffer | | | | |
|---------------|-------|-------|-------|-------|
| Black | Black | Black | Black | Black |
| Black | Black | Black | Black | Black |
| Black | Black | Black | Black | Black |
| Black | Black | Black | Black | Black |
| Black | Black | Black | Black | Black |

- Draw the triangle

| Depth Buffer | | | | |
|--------------|------------|------------|------------|------------|
| 1.0 | 1.0 | 1.0 | 1.0 | 0.1 |
| 1.0 | 1.0 | 1.0 | 0.1 | 0.1 |
| 1.0 | 1.0 | 0.1 | 0.1 | 0.1 |
| 1.0 | 0.1 | 0.1 | 0.1 | 0.1 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

| Colour Buffer | | | | |
|---------------|-------------|-------------|-------------|-------------|
| Black | Black | Black | Black | Blue |
| Black | Black | Black | Blue | Blue |
| Black | Black | Blue | Blue | Blue |
| Black | Blue | Blue | Blue | Blue |
| Black | Black | Black | Black | Black |

- Draw the rectangle

| Depth Buffer | | | | |
|--------------|------------|------------|------------|------------|
| 1.0 | 1.0 | 1.0 | 1.0 | 0.1 |
| 1.0 | 1.0 | 1.0 | 0.1 | 0.1 |
| 0.2 | 0.2 | 0.1 | 0.1 | 0.1 |
| 0.2 | 0.1 | 0.1 | 0.1 | 0.1 |
| 0.2 | 0.2 | 0.2 | 1.0 | 1.0 |

| Colour Buffer | | | | |
|---------------|-------------|-------------|-------------|-------------|
| Black | Black | Black | Black | Blue |
| Black | Black | Black | Blue | Blue |
| Red | Red | Blue | Blue | Blue |
| Red | Blue | Blue | Blue | Blue |
| Red | Red | Red | Black | Black |

2.8.3.2 Perspective Projection and the z-buffer

It is easy to think that values in the z-buffer will correspond to values in eye coordinates in some linear fashion. That is, a depth value of 0.1 represents one tenth of the distance from the near to the far clipping plane and the depth value 0.5 represents half way between the near and far clipping planes. If an orthographic projection has been used then the above simple description is true. However if a perspective projection has been used then the above linear assumption is not true. The reason for the nonlinearity is due to the way that we constructed the transformation of the z-coordinate in perspective projection.

It is easily shown (look up the lecture notes on the perspective transformation) that the z and w values in clip coordinates are related to eye coordinates by

$$z_c = -\frac{far + near}{far - near} z_e - 2 \frac{far \times near}{far - near} w_e$$

$$w_c = -z_e$$

Performing the homogeneous scaling to move into normalized device coordinates yields

$$z_{ndc} = +\frac{far + near}{far - near} + 2 \frac{far \times near}{far - near} \frac{w_e}{z_e}$$

The viewport transformation moves the x and y normalized device coordinates into x and y screen coordinates (pixels) and also transforms the -1 to $+1$ range of the z coordinate into a 0 to $+1$ range. Explicitly,

$$z_s = \frac{1}{2} \left[\frac{far + near}{far - near} + 2 \frac{far \times near}{far - near} \frac{w_e}{z_e} \right] + 0.5$$

$$= \frac{1}{2} \frac{far + near}{far - near} + \frac{far \times near}{far - near} \frac{w_e}{z_e} + 0.5$$

Most implementations of the depth buffer are actually integer based. We will denote the number of bits assigned to each element of the depth buffer by n (n is usually 16). The depth value is given by scaling the screen z coordinate to the range $[0, 2^n - 1]$.

$$z_{db} = \text{int} \left((2^n - 1) z_s \right)$$

$$= \text{int} \left((2^n - 1) \left[\frac{1}{2} \frac{far + near}{far - near} + \frac{far \times near}{far - near} \frac{w_e}{z_e} + 0.5 \right] \right)$$

Now to see how the eye coordinate z values correspond to values in the depth buffer we will rearrange the above equation to solve for $\frac{z_e}{w_e}$ (the details are an exercise for the reader).

$$\frac{z_e}{w_e} = \frac{far \times near}{(far - near) \frac{z_{db}}{2^n - 1} - far}$$

We can see that the relationship is clearly non-linear. To make things more concrete we will now examine some numeric cases:

| near | far | depth precision | near | far | depth precision |
|-------|--------------|-----------------|------|--------------|-----------------|
| 0.01 | 100 | 16 | 0.01 | 100 | 8 |
| zdb | ze/we | | zdb | ze/we | |
| 0 | -0.01 | | 0 | -0.01 | |
| 1 | -0.010000153 | | 1 | -0.010039366 | |
| 2 | -0.010000305 | | 2 | -0.010079043 | |
| 65533 | -76.61955035 | | 253 | -1.25907273 | |
| 65534 | -86.76225276 | | 254 | -2.486834406 | |
| 65535 | -100 | | 255 | -100 | |

In both of these cases you can see that the precision close to the near clipping plane is ridiculously fine and that the precision close to the far clipping plane is ridiculously coarse. For example, in the 8 bit depth buffer case, all objects at eye coordinates lying between $z = -2$ and $z = -100$ get mapped to only two depth values. This would defeat the purpose of having a depth buffer.

Now that the problem is clear, what is the solution? A programmer usually has no control over the precision of the depth buffer. Most times the depth buffer is in hardware and hence cannot be changed. However as a programmer you can control the near and far values.

Look at the following example:

| near | far | depth precision |
|-------|--------------|-----------------|
| 1 | 100 | 16 |
| | | |
| zdb | ze/we | |
| 0 | -1 | |
| 1 | -1.000015107 | |
| 2 | -1.000030214 | |
| 65533 | -99.69878143 | |
| 65534 | -99.84916354 | |
| 65535 | -100 | |

It is the ratio of near to far that the programmer can use to control the effective depth buffer precision. The first 16 bit example had a ratio of 1:10 000 whereas the second 16 bit example had a ratio of 1:100 and consequently the second example had much better effective precision. For many applications the 1:100 ratio will provide adequate effective depth precision but for some algorithms (for example shadow mapping) even higher effective precision will be required to achieve a realistic effect.

2.8.3.3 Depth-Buffering in OpenGL

A widely-used hidden-surface removal algorithm is the z-buffer algorithm. This algorithm computes the depth of each object from the near clipping plane; if the depth is less than the value in the z-buffer, then this object is closer; if the depth value is greater than that in the z-buffer, then it means that an object that is closer has already been processed. At the end of the algorithm, it is known which objects are in front and hence visible to the viewer.

OpenGL has built-in support for the z-buffer algorithm for hidden-surface removal. To implement it, z-buffer or depth-buffer storage must be requested:

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

The algorithm is enabled by calling

```
glEnable( GL_DEPTH_TEST );
```

Finally, the depth buffer must be reset before the scene is re-drawn, so `glClear` must contain an extra parameter:

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

2.8.4 Culling

Let us first review polygon faces. A polygon has two faces – a front face and a back face. By default, the front face of the polygon is defined to be the face where the vertices are specified in an anti-clockwise order. This default behaviour can be

changed. The front and back faces of a polygon may have different drawing attributes, i.e., one side may be red and the other side blue. Only one face of a polygon is drawn on the screen for a given viewing angle. If the viewing angle causes the polygon to appear on the screen in such a manner that the vertices appear in an anti-clockwise fashion, then the front face of the polygon will be drawn. Otherwise the back face will be drawn. This means that for a given viewing angle a polygon can be considered to be front-facing or back-facing.

Back-face culling is a form of hidden surface removal based on the following observation. Most 3D objects are made so that the outside of the object is composed of front faces of polygons, i.e., the back faces of the polygons are all on the insides of objects. To visualise this, consider drawing a cube. Even though the cube is constructed so that the outside is all front faces of polygons, because of the viewing angle some of the polygons must be backward-facing.

The trick to notice is that if you never slice open one of these objects, then there is never a need to draw a backwards-facing polygon because it will always be obscured by a front-facing polygon. Thus the back-face culling technique is simply to not draw any polygons that have been determined to be backward-facing from the current viewing angle. This technique has the pleasant side effect of improving the performance of the application as fewer polygons are now rendered.

There are instances where culling is not an appropriate technique. For example consider a crystal ball. You need to partially see through the ball, so this means that the back-facing polygons on the other side are not obscured by the front-facing side, and hence must be drawn to achieve a realistic effect.

2.8.4.1 Culling in OpenGL

Culling in OpenGL is an easy technique to implement. To tell OpenGL which face we are interested in culling, simply call the function

```
GLvoid glCullFace( GLenum mode );
```

where mode is one of `GL_BACK`, `GL_FRONT`, `GL_FRONT_AND_BACK`. If the mode is set to `GL_FRONT`, then front-facing polygons are not drawn, while back-facing polygons are not drawn if the mode is `GL_BACK`.

The culling calculation is enabled with the call

```
glEnable( GL_CULL_FACE );
```

The use of culling improves computer performance. However there are instances when culling would not be required, and so it is essential to be able to disable culling. For example, objects made of glass or plastic are transparent and so we need to be able to see both the front and back faces of such objects.