A PLATFORM-AGNOSTIC APPROACH

Game Programming ALGORITHMS and TECHNIQUES

Sanjay **MADHAV**

Game Programming Algorithms and Techniques

One interesting thing to note is that regardless of the shading method used, the silhouette is identical. So even if Phong shading is used, the outline can be a dead giveaway if an object is a low-polygon model.

Visibility

Once you have meshes, coordinate space matrices, lights, a reflection model, and shading, there is one last important aspect to consider for 3D rendering. Which objects are and are not visible? This problem ends up being far more complex for 3D games than it is for 2D games.

Painter's Algorithm, Revisited

As discussed in Chapter 2, "2D Graphics," the painter's algorithm (drawing the scene back to front) works fairly well for 2D games. That's because there's typically a clear ordering of which 2D sprites are in front of each other, and often the 2D engine may seamlessly support the concept of layers. But for 3D games, this ordering is not nearly as static, because the camera can and will change its perspective of the scene.

This means that to use the painter's algorithm in a 3D scene, all the triangles in the scene have to be resorted, potentially every frame, as the camera moves around in the scene. If there is a scene with 10,000 objects, this process of resorting the scene by depth every single frame would be computationally expensive.

That already sounds very inefficient, but it can get much worse. Consider a split-screen game where there are multiple views of the game world on the same screen. If player A and player B are facing each other, the back-to-front ordering is going to be different for each player. To solve this, we would either have to resort the scene multiple times per frame or have two different sorted containers in memory—neither of which are desirable solutions.

Another problem is that the painter's algorithm can result in a massive amount of **overdraw**, which is writing to a particular pixel more than once per frame. If you consider the space scene from Figure 2.3 in Chapter 2, certain pixels may have been drawn four times during the frame: once for the star field, once for the moon, once for an asteroid, and once for the spaceship.

In modern 3D games, the process of calculating the final lighting and texturing for a particular pixel is one of the most expensive parts of the rendering pipeline. If a pixel gets overdrawn later, this means that any time spent drawing it was completely wasted. Due to the expense involved, most games try to eliminate as much overdraw as possible. That's never going to happen with the painter's algorithm.

And, finally, there is the issue of overlapping triangles. Take a look at the three triangles in Figure 4.16. Which one is the furthest back?

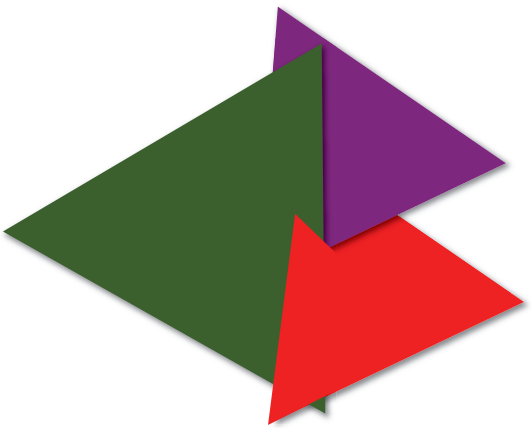


Figure 4.16 Overlapping triangles failure case.

The answer is that there is no one triangle that is furthest back. In this instance, the only way the painter's algorithm would be able to draw these triangles properly is if some of them were split into multiple triangles.

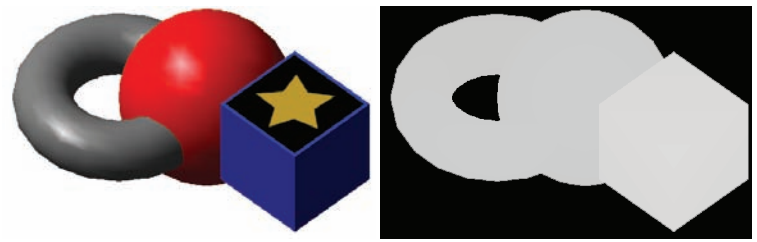
Because of all these issues, the painter's algorithm does not see much use in 3D games.

Z-Buffering

In **z-buffering**, an additional memory buffer is used during the rendering process. This extra buffer, known as the **z-buffer**, stores data for each pixel in the scene, much like the color buffer. But unlike the color buffer, which stores color information, the z-buffer (also known as the **depth buffer**) stores the distance from the camera, or **depth**, at that particular pixel. Collectively, the set of buffers we use to represent a frame (which can include a color buffer, z-buffer, stencil buffer, and so on) is called the **frame buffer**.

At the beginning of a frame rendered with z-buffering, the z-buffer is cleared out so that the depth at each pixel is set to infinity. Then, during rendering, before a pixel is drawn, the depth is computed at that pixel. If the depth at that pixel is less than the current depth value stored in the z-buffer, that pixel is drawn, and the new depth value is written into the z-buffer. A sample representation of the z-buffer is shown in Figure 4.17.

So the first object that's drawn every frame will always have all of its pixels' color and depth information written into the color and z-buffers, respectively. But when the second object is drawn, if it has any pixels that are further away from the existing pixels, those pixels will not be drawn. Pseudocode for this algorithm is provided in Listing 4.2.



Scene Z-Buffer

Figure 4.17 A sample scene and its corresponding z-buffer.

Listing 4.2 Z-Buffering

```
// zBuffer[x][y] grabs depth at that pixel
foreach Object o in scene
   foreach Pixel p in o
      float depth = calculate depth at p
      if zBuffer[p.x][p.y] > depth
         draw p
         zBuffer[p.x][p.y] = depth
      end
   end
  end
```

With z-buffering, the scene can be drawn in any arbitrary order, and so long as there aren't any objects with transparency, it will look correct. That's not to say that the order is irrelevant—for example, if the scene is drawn back to front, you would overdraw a lot of pixels every frame, which would have a performance hit. And if the scene were drawn front to back, it would have no overdraw at all. But the idea behind z-buffering is that the scene does not need to be sorted in terms of depth, which can greatly improve performance. And because z-buffering works on a per-pixel basis, and not a per-object basis, it will work well even with the overlapping triangles case in Figure 4.16.

But that's not to say z-buffering by itself solves all visibility problems. For one, transparent objects won't work with z-buffering as is. Suppose there is semi-transparent water, and under this water there's a rock. If pure z-buffering is used, drawing the water first would write to the z-buffer and would therefore prevent the rock from being drawn. To solve this issue, we first render all the opaque objects in the scene using normal z-buffering. Then we can disable depth buffer writes and render all the transparent objects, still checking the depth at each pixel to make sure we aren't drawing transparent pixels that are behind opaque ones.

As with the representation of color, the z-buffer also has a fixed bit depth. The smallest size z-buffer that's considered reasonable is a 16-bit one, but the reduced memory cost comes with some side effects. In **z-fighting**, two pixels from different objects are close to each other, but far away from the camera, flicker back and forth on alternating frames. That's because the lack of accuracy of 16-bit floats causes pixel A to have a lower depth value than pixel B on frame 1, but then a higher depth value on frame 2. To help eliminate this issue most modern games will use a 24- or 32-bit depth buffer.

It's also important to remember that z-buffering by itself cannot guarantee no pixel overdraw. If you happen to draw a pixel and later it turns out that pixel is not visible, any time spent drawing the first pixel will still be wasted. One solution to this problem is the **z-buffer pre-pass**, in which a separate depth rendering pass is done before the final lighting pass, but implementing this is beyond the scope of the book.

Keep in mind that z-buffering is testing on a pixel-by-pixel basis. So, for example, if there's a tree that's completely obscured by a building, z-buffering will still individually test each pixel of the tree to see whether or not that particular pixel is visible. To solve these sorts of problems, commercial games often use more complex **culling** or **occlusion** algorithms to eliminate entire objects that aren't visible on a particular frame. Such algorithms include binary spatial partitioning (BSP), portals, and occlusion volumes, but are also well beyond the scope of this book.

World Transform, Revisited

Although this chapter has covered enough to successfully render a 3D scene, there are some outstanding issues with the world transform representation covered earlier in this chapter. The first problem is one of memory—if the translation, scale, and rotation must be modified independently, each matrix needs to be stored separately, which is 16 floating point values per matrix, for a total of 48 floating points for three matrices. It is easy enough to reduce the footprint of the translation and scale—simply store the translation as a 3D vector and the scale as a single float (assuming the game only supports uniform scales). Then these values can be converted to the corresponding matrices only at the last possible moment, when it's time to compute the final world transform matrix. But what about the rotation?

There are some big issues with representing rotations using Euler angles, primarily because they're not terribly flexible. Suppose a spaceship faces down the z-axis in model space. We want to rotate the space ship so it points instead toward an arbitrary object at position P. In order to perform this rotation with Euler angles, you need to determine the angles of rotation. But there isn't one single cardinal axis the rotation can be performed about; it must be a combination of rotations. This is fairly difficult to calculate.

Another issue is that if a 90° Euler rotation is performed about a coordinate axis, the orientation of the other coordinate axes also changes. For example, if an object is rotated 90° about the

z-axis, the x- and y-axes will become one and the same, and a degree of motion is lost. This is known as **gimbal lock**, as illustrated in Figure 4.18, and it can get very confusing very quickly.

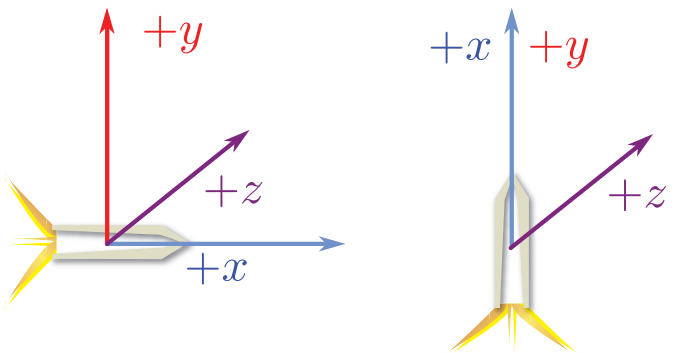


Figure 4.18 A 90° rotation about the z-axis causes gimbal lock.

Finally, there is a problem of smoothly interpolating between two orientations. Suppose a game has an arrow that points to the next objective. Once that objective is reached, the arrow should change to point to the subsequent one. But it shouldn't just instantly snap to the new objective, because an instant change won't look great. It should instead smoothly change its orientation, over the course of a second or two, so that it points at the new objective. Although this can be done with Euler angles, it can be difficult to make the interpolation look good.

Because of these limitations, using Euler angle rotations is typically not the preferred way to represent the rotation of objects in the world. Instead, a different math construct is used for this.

Quaternions

As described by mathematicians, quaternions are, quite frankly, extremely confusing. But for game programming, you really can just think of a **quaternion** as a way to represent a rotation about an arbitrary axis. So with a quaternion, you aren't limited to rotations about x, y, and z. You can pick any axis you want and rotate around that axis.

Another advantage of quaternions is that two quaternions can easily be interpolated between. There are two types of interpolations: the standard lerp and the **slerp**, or spherical linear interpolation. Slerp is more accurate than lerp but may be a bit more expensive to calculate, depending on the system. But regardless of the interpolation type, the aforementioned objective arrow problem can easily be solved with quaternions.

Quaternions also only require four floating point values to store their information, which means memory is saved. So just as the position and uniform scale of an object can be stored as a 3D vector and float, respectively, the orientation of the object can be stored with a quaternion.

For all intents and purposes, games use **unit quaternions**, which, like unit vectors, are quaternions with a magnitude of one. A quaternion has both a vector *and* a scalar component and is often written as $q = [q_v, q_s]$. The calculation of the vector and scalar components depends on the axis of rotation, $\hat{a}$, and the angle of rotation, θ:

$$q_v = \hat{a} \sin\frac{\theta}{2}$$
$$q_s = \cos\frac{\theta}{2}$$

It should be noted that the axis of rotation must be normalized. If it isn't, you may notice objects starting to stretch in non-uniform ways. If you start using quaternions, and objects are shearing in odd manners, it probably means somewhere a quaternion was created without a normalized axis.

Most 3D game math libraries should have quaternion functionality built in. For these libraries, a `CreateFromAxisAngle` or similar function will automatically construct the quaternion for you, given an axis and angle. Furthermore, some math libraries may use x, y, z, and w-components for their quaternions. In this case, the vector part of the quaternion is x, y, and z while the scalar part is w.

Now let's think back to the spaceship problem. You have the initial facing of the ship down the z-axis, and the new facing can be calculated by constructing a vector from the ship to target position P. To determine the axis of rotation, you could take the cross product between the initial facing and the new facing. Also, the angle of rotation similarly can be determined by using the dot product. This leaves us with both the arbitrary axis and the angle of rotation, and a quaternion can be constructed from these.

It is also possible to perform one quaternion rotation followed by another. In order to do so, the quaternions should be *multiplied* together, but in the reverse order. So if an object should be rotated by q and then by p, the multiplication is pq. The multiplication for two quaternions is performed using the **Grassmann product**:

$$(pq)_v = p_s q_v + q_s p_v + p_v \times q_v$$
$$(pq)_s = p_s q_s - p_v \cdot q_v$$

Much like matrices, quaternions have an inverse. Luckily, to calculate the inverse of a unit quaternion you simply negate the vector component. This negation of the vector component is also known as the **conjugate** of the quaternion.

Because there is an inverse, there must be an identity quaternion. This is defined as:

$$i_v = \langle 0,\ 0,\ 0 \rangle$$
$$i_s = 1$$