

Frustum Culling

Author: Max Wagner, mwagner@digipen.edu

Pre-Reqs

Familiarity with terms and concepts from Linear Algebra, including the ideas behind vectors and vector spaces. Additionally, the 3D graphics transformation pipeline, including the View (or Camera) and Projection transformations. Also, you may want to read my tutorial on Planes and Half-Spaces (find it at www.emeyex.com).

Notation

Vectors AND points in bold, i.e. \mathbf{u} or \mathbf{n} . I leave it to the reader to discern from the context whether the object is a point or a vector. Scalars will be weighted normally, that is un-bolded. Often I will refer to a vector \mathbf{u} , and then shortly afterward refer to its components as u_x , u_y , u_z , etc.

What is a Frustum?

The simple answer is that it's a pyramid with its top cut off, such that the planes corresponding to the top and the bottom are parallel. We often refer to the "view frustum" in computer graphics, referring to that volume sitting in world space which corresponds to the visible real estate. This view volume is subject to various parameters, which I will not go into here. Suffice it to say that the most common type of frustum we deal with in computer graphics is this view frustum, and that the information we have from which to construct our frustum depends on the camera's View and Projection matrices.

Representing the Frustum

For our purposes (culling), the most useful way to represent our frustum is using six plane vectors, one for each face. We will call these faces front, back, left, right, top, and bottom. By convention, we will assume that these plane vectors have outwardly facing normals, that is, normals that point away from the frustum (as opposed to pointing toward the interior of the frustum); hence, the frustum volume is defined as the intersection of the negative half-spaces of the six frustum planes. As a quick example, we could test whether a point were in our frustum by performing a half-space test on the point for each face plane; if the point is found to be in the positive half-space of any of the six planes, it is not inside the frustum; otherwise, it is inside.

So this sounds pretty simple, right? Right. But how do we get these six plane vectors, given only the View and Projection Matrices?

Finding the Frustum's Plane Equations

Let's break it down. The view frustum, like anything else in our simulated world, can live in multiple spaces: world space, view space, projection space. Which space do we want it in? Which space is most useful? This can depend on your circumstances. We already know what the frustum is in projection space (or we will shortly, at least). But most often, we want it in view or world space. This discussion will be assuming we want to get the frustum in world space, but it will end up demonstrating how to get it into view space if that is preferred.

So let's look at the life cycle of a point. Assuming that it has already been transformed into the world, the standard concatenation of the View and Projection transformations will put the point into projection space, what is also referred to as Normalized Device Coordinate space, or just NDC for short. In this space, the view frustum has been simplified to become a box, with extents of -1 and 1 for each of the x , y , and z axes¹. We know that to get the point back out into world space, we would have to multiply it by the inverse of its View-Projection transformation. So why don't we just do the same thing for the view frustum? Fortunately, a similar transformation will work (*not* an identical one, however).

Let's start with the point. Specifically, to put a point \mathbf{p} in the world into projection space, we multiply it by the concatenated View-Projection transformation:

$$\mathbf{p}' = \mathcal{P}\mathcal{V}\mathbf{p}$$

Getting the point back into the world looks like (this assumes you have not performed the division by w on the point, of course):

$$\begin{aligned} (\mathcal{P}\mathcal{V})^{-1}\mathbf{p}' &= \mathbf{p} \\ \mathcal{V}^{-1}\mathcal{P}^{-1}\mathbf{p}' &= \mathbf{p} \end{aligned}$$

The primary difference for transforming the frustum planes is that they must be transformed using the rules for the transformation of normal vectors (see my notes on Planes and Half-Spaces, where I derive this transformation).

Thus, given an arbitrary frustum plane f' in projection space, its transformation to the world would look like:

$$\begin{aligned} (((\mathcal{P}\mathcal{V})^{-1})^T)f' &= f \\ (\mathcal{P}\mathcal{V})^Tf' &= f \\ \mathcal{V}^T\mathcal{P}^Tf' &= f \end{aligned}$$

But what do these frustum planes look like in projection space? Above I discussed the extents of the frustum for each axis. This information can be translated into the following plane vectors:

¹ This is according to OpenGL. In DirectX, the NDC z -axis of the view frustum has extents of 0 and 1

$$\begin{aligned}
 f'_{right} &= [1 \ 0 \ 0 \ 1]^T \\
 f'_{left} &= [-1 \ 0 \ 0 \ 1]^T \\
 f'_{top} &= [0 \ 1 \ 0 \ 1]^T \\
 f'_{bottom} &= [0 \ -1 \ 0 \ 1]^T \\
 f'_{back} &= [0 \ 0 \ 1 \ 1]^T \\
 f'_{front} &= [0 \ 0 \ -1 \ 1]^T
 \end{aligned}$$

Again, I remind you that these plane vectors are constructed using the convention of outwardly facing normals; it would be just as easy to define the six plane vectors using the convention of inwardly facing normals; it's just important to be consistent.

So there you have it, the recipe for getting your six frustum planes into whichever space you like. To recap, if you want to perform your frustum culling in world space, multiply the above plane vectors by the transpose of your combined View-Projection matrix; if you want to perform your frustum culling in view space, multiply the plane vectors by the transpose of your Projection matrix. Notice too that given the simple form of the above plane vectors (all those zeroes), you can optimize the vector-matrix multiplies. I'll leave that as an exercise for the reader.

Another important note: you will likely want to normalize your plane vectors after these transformations, according to the proper rules for plane normalization (again, see my notes on Planes and Half-Spaces). Otherwise you will get incorrect distance measurements when performing culling.

Culling Spheres

To get our feet wet, let's see how can apply this fancy new frustum object we have to some simple bounding sphere culling. While there are myriad other, more sophisticated bounding volume hierarchy strategies, it's hard to beat the combination of simplicity and speed of bounding spheres.

The question of computing a bounding sphere aside, let us assume that we have an object with a bounding sphere S represented by a center \mathbf{c} and a radius r ; this bounding sphere is initially in the object's own space (not world space). Thus we need to transform the sphere to world space, transform the view frustum to world space, and then perform a sphere/frustum intersection test.

The beauty of bounding spheres is that they fully contain the object at any orientation. Thus, the transformation to world space for a bounding sphere requires only a translation (one vector addition), as opposed to a full 4x4 matrix-vector multiply³. Our sphere, in the world, becomes S' , where $\mathbf{c}' = \mathbf{c} + \mathbf{t}$, $r' = r$, for the object's world translation \mathbf{t} .

² Again, this is for the case of OpenGL. In DirectX, the equation for the front plane would be $[0 \ 0 \ -1 \ 0]$.

³ This assumes we are not allowing scaling in our world transformations.

We now know how to get our frustum into world space, let's assume we have the six plane vectors, and let's label them as an array f , where this array is indexed by face.

The sphere/frustum intersection test is all that remains. This test involves testing the signed distance of the sphere's center to each of the six frustum planes. If the (signed) distance is ever found to be greater than the sphere's radius, then the sphere must be wholly outside the frustum (and thus can be discarded from rendering). Otherwise, if we get to the end (testing the sphere against each plane), then the sphere must either be wholly contained or intersecting a plane, in which case we would render the object. This case is a good example of where it's important to ensure the frustum planes are normalized, otherwise the distances reported will be inaccurate.

In pseudocode, this test looks like

```
for each face i in frustum
    if ( SignedDistance(f[i], c') > r' )
        return outside
    end if
end for

return contained/intersecting
```

Where $f[i]$ is i^{th} frustum plane vector, c' is the sphere's center in world space, and r' is the sphere's radius in world space.

Sometimes you will want to differentiate between the cases of wholly contained and merely intersecting. Fortunately, this is handled easily as a modification to the initial algorithm. Instead of testing if the sphere is outside a given a plane, we will add a test for intersection. This test asks if the absolute value of the signed distance is less than the sphere's radius; if the answer is yes, then we have an intersection.

In pseudocode:

```
for each face i in frustum
    d ← SignedDistance(f[i], c')
    if ( d > r' )
        return outside
    else if ( Abs(d) < r' )
        return intersecting
    end if
end for

return contained
```

This latter approach of separating the contained and intersecting cases is useful when using volume hierarchies. If you had a parent object containing many other objects, and that object's bounding sphere tested as wholly contained in the view frustum, you could dispense with testing the children; they would by necessity be wholly contained as well.