

CS 314: Clipping, Hidden Surfaces

Robert Bridson

September 30, 2008

1 View Clipping

We do have a small technical issue with our projection it turns out. We constructed a projection matrix followed by a homogenization (division by the homogeneous coordinate) which maps points in the view frustum to the canonical view volume $[-1, 1]^3$, and points outside the frustum to outside it. If we were just rendering points, this is simple enough: transform each point, and only rasterize those that end up inside the canonical view volume. However, for triangles it's more complicated.

The first thing to notice is that a triangle with one, two or even three vertices outside the view frustum can still partially overlap with it. In particular, some vertices may have their camera z coordinates closer than the near clipping plane or further than the far clipping plane, meaning we can't rasterize the entire triangle, but part of the triangle may still be visible.

1.1 Clipping during Rasterization

In the orthographic projection case, it is possible to fix this in the rasterization stage, by computing the projected z coordinate of each pixel (using the barycentric coordinates to interpolate from the projected z coordinates of the vertices of the triangle), and decided not to draw the pixel if this lies outside the range $[-1, 1]$. However, this doesn't work for perspective projection.

The real problem with perspective projection is that points that are behind the camera (i.e. have positive camera z) get mapped—after projection and the homogenization divide—to be in front of the camera. Let's take a more detailed look at this. From last time, the relevant parts of the 4×4 projection matrix—the bits involving z and the homogeneous coordinate w , with n and f being the near and far

clipping values—are:

$$\begin{pmatrix} \cdot \\ \cdot \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} \cdot \\ \cdot \\ z \\ 1 \end{pmatrix}$$

For simplicity I'm using 1 for the input homogeneous coordinate in camera space. Multiplying this out gives:

$$\begin{aligned} z' &= -\frac{f+n}{f-n}z - \frac{2fn}{f-n} \\ w' &= -z \end{aligned}$$

Finally, after homogenizing (dividing by w') we get the normalized device coordinate z'' , a.k.a. the depth:

$$z'' = \frac{f+n}{f-n} + \frac{2fn}{(f-n)z}$$

It's a great exercise to sketch the graph of this function. Verify that, as expected, the near clipping plane $z = -n$ gets mapped to depth $z'' = -1$, and the far clipping plane $z = -f$ gets mapped to depth $z'' = +1$. When z is in between $-n$ and $-f$, the depth is between -1 and 1 .

If the point is in front but far away from the camera, $z < -f$, it's easy to see the depth is between 1 and $(f+n)/(f-n)$, with it approaching this latter limit as $z \rightarrow -\infty$. If the point is between the near clipping plane and the camera, $-f < z < 0$, then the depth ends up in the range $(-\infty, -1)$. However, if the point is behind the camera, $z > 0$, the depth 'wraps around' infinity and ends up positive: $z'' > (f+n)/(f-n)$.

The danger is that if one vertex of a triangle is behind the camera and another in front, if you interpolate between the depth z'' values, you get ghost points that really aren't in the triangle. Refer to the pictures drawn in class.

(As an aside, you should also foresee some possible danger if a triangle vertex just happens to have $z = 0$ in camera coordinates!)

What this boils down to is an argument that it's too late to deal with clipping in z at the rasterization stage, after the homogenization division. We need to clip against the near and far planes earlier; conceptually we'll do this with camera space coordinates, though as Shirley section 12.2 expounds upon there is an intermediate possibility adopted by most implementations (which also clip against the other sides of the view frustum or view volume, to reduce the work-load of the rasterization stage).

1.2 Clipping against Near and Far Planes

We'll simplify the problem of clipping against both near and far planes by doing one at a time: for definiteness we'll just focus on the $z = -n$ near plane. Again, we'll assume for these notes this is all in camera space coordinates, and we'll ignore clipping against the other sides of the view frustum or volume.

Clipping a triangle against a plane means cutting off the part of the triangle on the wrong side of the plane. This breaks up into four cases:

- All vertices (hence the entire triangle) are on the correct side, $z \leq -n$. The triangle can be left as is.
- All vertices (hence the entire triangle) are on the wrong side, $z \geq -n$. The entire triangle gets clipped away, leaving nothing.
- One vertex is on the correct side $z < -n$ and the others are on the wrong side, $z \geq -n$. In this case, the clipped part is a smaller triangle.
- Two vertices are on the correct side $z < -n$ and one is on the wrong side $z > -n$. The clipped region we're left with is a quadrilateral.

The first two cases are trivial; the third and especially the fourth need a little more effort.

For the third case, label the vertex on the correct side as 0 (so $z_0 < -n$) and the others as 1 and 2 ($z_1 \geq -n, z_2 \geq -n$). The clipped result is a triangle with vertex \vec{x}_0 unchanged, and new vertices \vec{x}_1^* and \vec{x}_2^* . It should be clear these new vertices are on the clipping plane, $z_1^* = z_2^* = -n$, and also are on the edges $0 \leftrightarrow 1$ and $0 \leftrightarrow 2$ of the original triangle: draw a diagram if this isn't clear to you, and study it until it is. All we have to do then is figure out the point of intersection of each edge with the plane. One way to do this is to parameterize each edge (an explicit representation) with parameter s . Edge $0 \leftrightarrow 1$ can be defined by:

$$\vec{x}(s) = \vec{x}_0 + s(\vec{x}_1 - \vec{x}_0), \quad s \in [0, 1]$$

Now solve for the value of s which gives $z(s) = -n$ to get the point of intersection:

$$\begin{aligned} z(s) = z_0 + s(z_1 - z_0) &= -n \\ \Rightarrow s &= \frac{-n - z_0}{z_1 - z_0} \end{aligned}$$

Finally, plug this value of s in to $x(s)$ and $y(s)$ to evaluate the camera x and y coordinates of the clipped triangle vertex \vec{x}_1^* . The coordinates of \vec{x}_2^* can be found in similar fashion (with a different value for s obviously).

For the fourth case, where the triangle gets clipped to a quadrilateral, we have a further issue: our rasterization algorithm doesn't handle quadrilaterals, just triangles. We can fix this by splitting the quadrilateral into two triangles by cutting it along a diagonal (it doesn't matter which diagonal). The vertices of the two triangles are found with edge-plane intersections in exactly the same way as above.

Note that thanks to the fourth case, one input triangle may be split into two triangles for the rasterizer, and in fact every time we clip against another plane (such as $z = -f$) triangles might be split further. Clipping can increase the amount of geometry being dealt with!

2 Hidden Surface Elimination

So far we have constructed the following for our 3D rendering pipeline: transformations for modeling and viewing, transformations for orthographic and perspective projection, clipping of triangles outside the view volume or frustum, and rasterization of the resulting 2D triangles. The biggest part we're missing is dealing with **hidden surfaces**: making sure that a point closer to the camera than another point is rasterized "in front". Right now, depending upon the order in which triangles are processed for rasterization, a triangle that should be completely **occluded** (blocked from sight) by a closer triangle might instead get drawn over the closer triangle.

For simplicity—and this is by far the most common case—we'll assume for now that all surfaces we want to render are fully opaque.¹ Full opacity means every pixel in the final image should have the colour of the closest surface to the camera at that pixel.

There are several approaches to this problem, eliminating hidden surfaces. We can loosely divide them into three categories:

- clipping away hidden surfaces in camera space and rasterizing in any order,
- sorting triangles in camera space and then rasterizing from back to front,
- and figuring out the nearest surface on a pixel-by-pixel basis.

¹In fact we've made an additional hidden assumption right from the start of this course, that we're rendering surfaces—there are **volumetric** phenomena like smoke and fog and fire that don't correspond to any surface model but still are relevant to computer graphics; we won't broach the subject of **volume rendering** for now.

2.1 Clipping Hidden Surfaces in Camera Space

The first category isn't particularly attractive anymore. Here we generalize the work in the last section, clipping triangles against planes, to clip a triangle against the silhouette of any triangles that appear in front of it. To do this efficiently and robustly, taking into account floating point rounding error, is a severe implementation challenge. Moreover, it can cause an explosion of geometry: even with just two triangles, one partially in front of the other, the output from clipping could be ten sub-triangles (I've leave it as a challenge for you to figure out how that happens).

2.2 Painter's Algorithm

The second category revolves around something called the **Painter's Algorithm**. Though this isn't exactly how painters would approach a painting (for watercolours it's definitely not) the idea is that if you paint, or rasterize, triangles that are further way first, any closer triangles can overwrite any of those pixels safely. The key is that the triangles have to be sorted by depth, and then rasterization can proceed from the biggest depths to the smallest depths.

The biggest problem with this is that it's not always possible to sort triangles by depth: a triangle can occupy a whole range of depth values (from its nearest vertex to the furthest), and two triangles' ranges can overlap. In particular, if two triangles intersect each other, at some points in the image the first triangle will be in front of the other, whereas elsewhere the second triangle will be in front of the first. Figure 8.2 in Shirley's book shows an example of three non-intersecting triangles which similarly have no consistent ordering: triangle 1 is partly in front of triangle 2, which is partly in front of triangle 3, which is partly in front of triangle 1.

Some specialized rendering applications can rule these cases out, reducing the implementation complexity to figuring out an efficient means of sorting by depth, but in general this approach will require that some triangles get split into sub-triangles which can be fully depth-sorted. Note that this is not the same as the clipping algorithm above, and in comparison typically many less triangles will need to be cut up to get to the point where the painter's algorithm can work, but it still is done in 3D space independent of pixels. A classic approach to deciding how to cut up triangles involves a BSP (binary space partition) tree, discussed in Shirley's book in section 8.1, which simultaneously makes sorting efficient; it was popular before graphics hardware become common, featuring in games such as the original Doom.

2.3 Z-Buffer

The first example of an algorithm from the third category, where sorting by depth is done pixel-by-pixel, is the **Z-buffer algorithm**. This is what is implemented in virtually all graphics hardware for real-time graphics work, and a more sophisticated variation allowing for partial transparency (albeit with a vastly different approach to rasterization) is used for almost all film rendering.²

The idea is to rasterize every triangle—in any order—but keep track of which triangle ends up nearest in each pixel: the pixel gets the colour of the closest triangle. Note that a full sort is not needed, since we only care about finding the minimum depth value. The obvious algorithm for finding the minimum of a set of numbers is to keep a running count of the smallest number seen so far. This is what the Z-buffer algorithm does with every pixel.

More specifically, we introduce a **Z-buffer**, which means in addition to storing RGB values in every pixel we also allocate room for a single number, the depth (z). Initially the Z-buffer values are set to the maximum depth possible (usually 1). The algorithm then rasterizes each triangle. However, instead of always overwriting a pixel that the triangle overlaps, the following logic is added:

- Use barycentric coordinates to interpolate the depth of the triangle at that pixel from the depths of the vertices, just like we do for colour.
- If the new depth is less than that stored current in the Z-buffer for this pixel, overwrite the colour and the Z-buffer value.
- Otherwise, this triangle is behind a triangle that has already been rasterized—at this pixel—and so we move on without modifying anything.

This logic is simple enough (and can be run on all pixels in parallel potentially), and the memory for the Z-buffer is cheap enough nowadays, that this algorithm has proven to be by far the most efficient for hardware rendering. This may change in the future of course.

2.4 Raycasting

The other major example of the pixel-by-pixel approach to hidden surface elimination is **raycasting**. The previous Z-buffer algorithm loops over all the triangles, and for each triangle updates (or leaves alone if hidden) all the pixels it touched. Raycasting takes the loops the other way around: raycasting loops over all the pixels, and for each pixel figures out which triangle is nearest.

²This is the REYES algorithm, which is covered in cs426.

This is usually thought of in terms of casting a ray from the camera (the origin of camera space) through the near clipping plane at the location that gets mapped to a particular pixel. The ray may intersect several objects in the scene; the first intersection is the one that is visible through the pixel and the rest are hidden, so it is the surface of first intersection which provides the colour for the pixel.