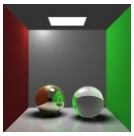


2018/2019, 4th quarter

INFOGR: Graphics / *NeedToKnow*

Lecture 13 – Visibility

Author: Jacco Bikker

TL;DR

Visibility is an important topic when rendering using the rasterization algorithm. Triangles can be off-screen, or behind other triangles; in fact, entire meshes can be invisible. Detecting this efficiently (ideally, with as little computation as possible) is crucial to rendering massive scenes, where the number of visible triangles is often a fraction of the total number of triangles.

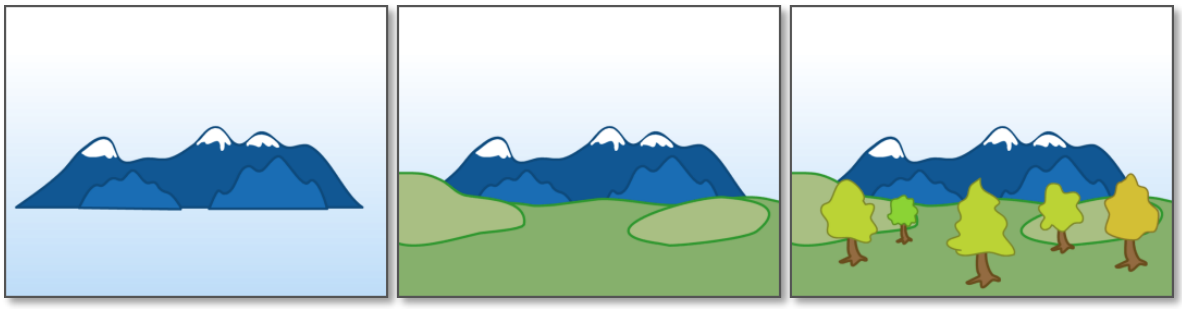
Visibility

Visibility determination is a broad topic, with many aspects:

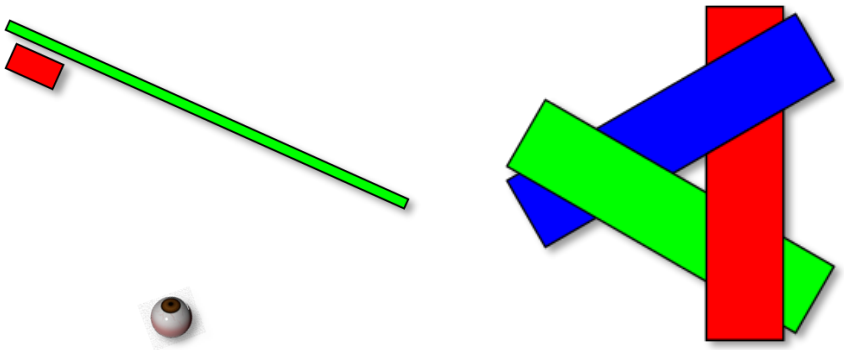
At a fine level, we need to determine which (parts of) triangles are in front of others. We also need to determine which (parts of) triangles are off-screen. At a somewhat coarser level we want to skip trees that are behind us. Ideally not one triangle at a time, but the whole mesh at once. Then we have the objects that are far away: we either skip them (beyond a certain distance) or draw them with reduced detail. Visibility also affects our lighting: a shadow occurs when some point is invisible from a light source.

The Closest Triangle

After drawing all the triangles, the ones that we actually see are the ones that are closest. We can achieve this by simply sorting the polygons. If we draw them back-to-front, the nearby ones will be drawn over the further away ones. This is similar to how a painter builds an image on canvas, which is why the technique is named the *Painter's algorithm*.



The algorithm is not flawless. Sorting requires a single depth per triangle, which is typically the distance of the center of the triangle to the camera. It is however possible for a triangle to be closer than some other triangle, while still being in front of it. And in some cases, a correct order is impossible: only splitting the triangles would help in that case.



A simple solution to the triangle sorting problem is the z-buffer. With a z-buffer, we don't sort triangles, but pixels: for each pixel of a triangle, we test the depth against the value stored in the z-buffer. If the pixel is closer than any other pixel drawn to that location, we draw the pixel, and update the value in the z-buffer. Although this yields pixel-perfect ordering, it is substantially more work for a CPU. Instead of just writing a pixel, we now:

1. read the z-buffer
2. compare the depth
3. write to the z-buffer and
4. write the pixel.

These days every GPU implements a z-buffer in hardware. It can still be beneficial to sort polygons (at least somewhat): if we draw the triangles front-to-back, we reduce the number of pixels that pass the comparison (step 2).

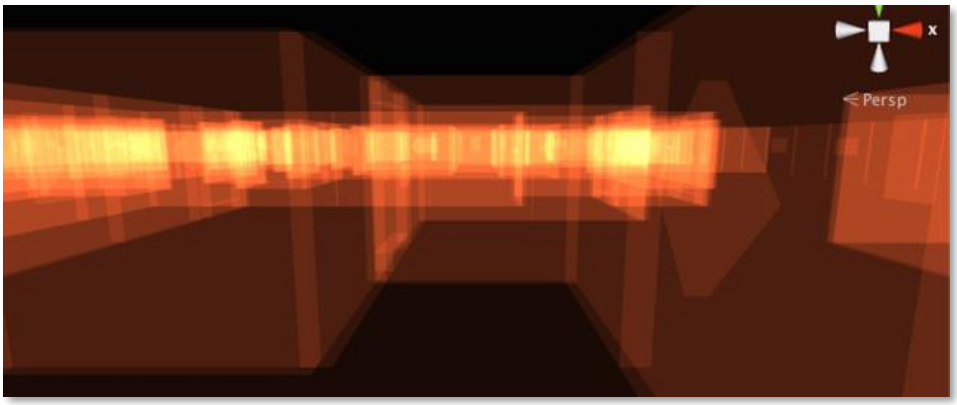
Z-Buffer Details

So what do we store in a z-buffer? The depth, obviously, but there are many ways to represent a depth, and the original floating point z value is not the best way. We can use $1/z$: if $A < B$, then $1/A < 1/B$. The benefit is that we get more precision up close, which is where the detail is needed. We can take this further however.

With a trick we can limit our values to the range $0..1$. In OpenGL, $1/(-z+1)$ is 1 if $z = 0$ and shrinks to 0 as z approaches -infinity (recall that z is negative in front of the camera). A number between 0 and 1 can then be converted to a 32-bit integer. The final depth becomes: $(2^{32}-1)/(-z+1)$. Storing the depth this way is significantly more precise than storing the generic float.

Overdraw

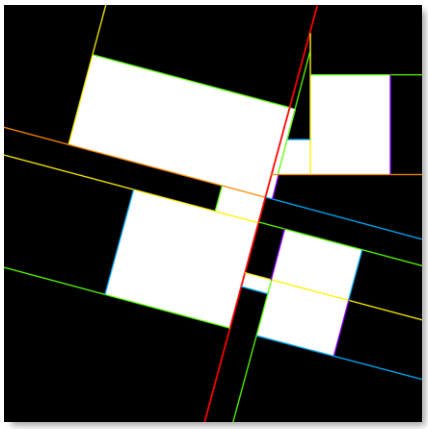
A z-buffer ensures that the final image is correct. It does however not prevent overdraw: the effect that a single pixel changes color multiple times per frame. For translucent objects this is inevitable, but if we have a complex car behind a simple building, drawing the car (and then the building over it) is just a waste of time.

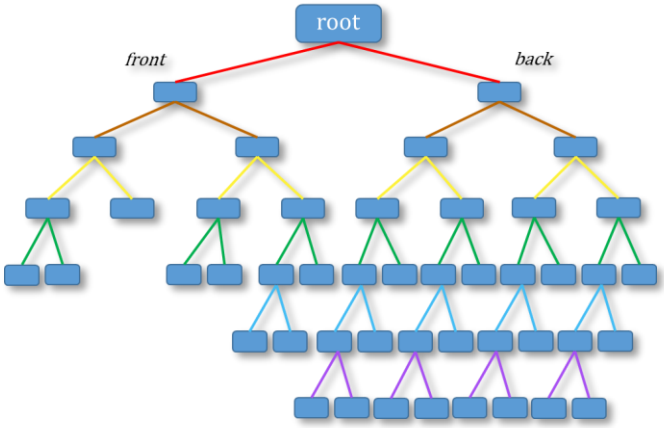


Overdraw in a maze.

BSP

Quake 1 (1996) used a number of novel techniques to render its 3D worlds efficiently. In the game, static geometry and dynamic entities are handled by different code paths. The static geometry is rendered using a BSP tree: a *binary space partitioning* tree. This structure subdivides the world in two *half spaces*, using the plane of a polygon in the scene. This is done recursively: each of the half spaces is split again, until only half spaces remain that cannot be split by polygons (i.e., convex volumes).





Using a BSP we can draw the scene with perfect sorting, without using a z-buffer. Starting at the root, we determine on which side of the split plane the camera is located. We then first process the far half space, then the near one. Once we get to the leafs of the tree we draw the triangles they contain. These will now be drawn back-to-front, without any explicit sorting.

Quake used a z-write for the static geometry. This means that the z-buffer is unconditionally written to, but it is never read, which also saves the comparison. Once all static geometry has been processed, the dynamic entities are drawn with full z-buffering (read-test-write-write). The resulting renderer has no sorting problems at all, and is efficient.

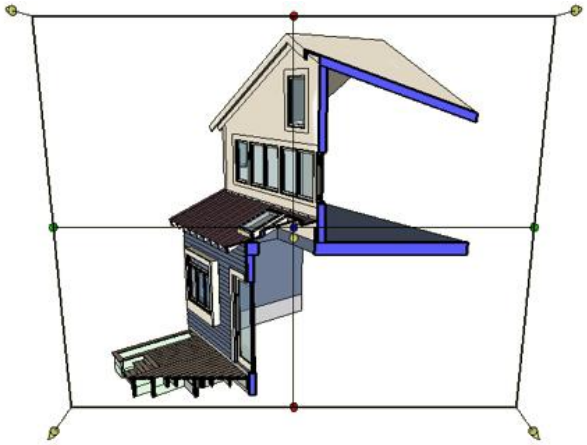
PVS

The BSP provides correct sorting, but it does not eliminate overdraw. For this, Quake used another technique: the PVS, or potential visibility set.

The PVS is a large table which stores the mutual visibility of areas in the game. Constructing the PVS is a complex and time consuming procedure, so it is only useful for static geometry. Once it has been calculated we can use it to quickly find the areas that may be visible from the area that the camera is in. The test is *conservative*: often an area is visible from the area the camera is in, but not from the camera location itself.

Clipping

A triangle that is only partially on the screen needs to be clipped. Clipping is also needed when a triangle crosses the $z = 0$ plane, and ideally, we also clip to the far clipping plane. OpenGL takes care of this for us, but in some cases, we may actually want to do the clipping ourselves, e.g. when we want an arbitrary cut of a scene.



To clip an *ngon* (i.e., planar polygon with any number of vertices) to an arbitrary plane, we use *Sutherland-Hodgeman clipping*. The algorithm takes a list of vertices, and emits a new list of vertices, which forms the clipped ngon. To do so, it loops over the *edges* of the ngon. For each edge, it looks at the start vertex v_0 and the end vertex v_1 to classify the edge. There are four cases:

1. The edge is 'in': v_0 and v_1 are both in front of the plane.
2. The edge is 'out': v_0 and v_1 are both behind the plane.
3. The edge is 'coming in': only v_1 is in front of the plane.
4. The edge is 'going out': only v_0 is in front of the plane.

If the edge intersects the plane, we calculate the intersection point C. Depending on the case, we now emit 0, 1 or 2 vertices for the clipped ngon:

- Case 1: emit v_1.
- Case 2: emit nothing.
- Case 3: emit C and v_1.
- Case 4: emit C.

Make sure you practice this on a piece of paper at least once.

Guard Bands

To prevent excessive clipping for detailed meshes, the PS2 introduced guard bands. By making the screen slightly larger, triangles that are partially off-screen but not outside the guard band can be safely drawn without clipping.

Large Scale Culling

Although the z-buffer and clipping allow us to render scenes of arbitrary size, this is not exactly efficient for large scenes.

Culling is the process of preventing the rendering of (groups of) triangles. This can be done at several levels:

Backface culling eliminates the triangles that are not facing the camera. If we assume that triangles are single-sided, this reduces the number of visible triangles by 50%, at the cost of a dot product per triangle.

We can also cull using *bounding spheres*. If we know the bounds for a mesh, we can check if these bounds intersect the *view frustum*. If not, the object can be safely ignored.

Multiple meshes (with their bounding spheres) can be grouped. We now have a *bounding volume hierarchy*, which allows us to very efficiently cull large groups of objects.

Alternatively, if we have a world that is organized in a grid, we can simply limit our efforts to the tiles that the view frustum overlaps.

THE END

There is more, but this document is already six pages! The rest of the topics discussed in the lecture will not appear in this year's exam.

If you have any questions, feel free to ask by email or on Slack!



INFOGR2019 – NEEDTOKNOW