Fast Backface Culling for Real Time 3D Rendering

Vadim Manvelyan
Faculty Advisor: Dr. Norman Badler

Abstract:
Speed up backface culling by discarding sets of faces at a time, as opposed to individual faces.

Basic Approach:
If the faces are sorted by the direction their normals point at, and e.g. it is known that the observer is to the north of the mesh, we can discard all southward faces without performing individual visibility checking on each. (Assuming non-transparent northward faces, of course).

Implementation:
Place an imaginary sphere around an object. Project the normals of the mesh’s faces onto sectors of the sphere bounded by its parallels and meridians. Data structure-wise, each “sector” is represented by a list of faces whose normals fall into that sector (See Figure 1). When rendering an object, draw a vector from the center of this sphere to the eye location. Note which sector that vector passes through. Faces that do not belong to the hemisphere centered around that sector should be discarded. Figure 2 shows that (simplified to 2D): the vector to the camera passes through the orange sector. White sectors are discarded, orange and blue ones are rendered. Note that only the direction to the eye matters, not distance (for convex objects)—even at infinity, the observer will not see any faces that do not “touch” the grey area, and the rendered sectors always overestimate the number of faces seen. The smaller the sector angle (angle between neighboring parallels/meridians), the more precise the approximation is, the less extra geometry is rendered. However, as the angle decreases further, the list of faces associated with each sector gets shorter and the rendermap gets larger, lowering the efficiency.

The actual rendering takes place as follows: we have a rendermap—a 2D boolean matrix representing all sectors. As the central “orange” sector gets determined, a square of \((180/\text{sector-angle} + 1)^2\) cells around it is set to ‘true’. This is quite a trivial procedure, of \(O((180/\text{sector-angle} + 1)^2)\)—very fast if the sector angle is not too small. However, the manner in which spherical coordinates project onto a rectangular grid creates some complications. When the camera looks over a pole, sometimes sectors that should be visible are not rendered (Figure 3, the eye is in the plane of the page, very far to the right—in this case, distance matters).

This happens because projection of a line onto an inclined sphere will get farther from the axis meridian as it moves up the latitude to the pole, and the span covering the visible surface should include more sectors. In Figure 3, the sectors colored yellow are mistakenly omitted, even though everything to the right of the red line should be rendered. This is corrected by “smoothing the corners” of the omitted region—the sectors likely to be “yellow” are added to the render list. For 10 degree sector angles, this decreases rendering efficiency by less than 2% (6 extra sectors are added, out of 648 possible sectors) without any complicated math, and takes care of the problem nicely. Another solution, for e.g. objects on the ground, is rotating the projection sphere such that its polar axis lies on the ground plane—then “looking over” a pole is impossible, since the terrain surface prevents the camera from seeing the problematic region altogether. (Figure 4).

Another problem is that the algorithm gets confused when the eye is very close to (or inside) a concave object (Figure 5). It is easy to construct a situation when the normal of an eye-facing surface does not belong to the set of rendered sectors (note how, given the eye position, the faces marked red will not be seen, irrespective of the view vector). To take the view vector into account meaningfully, it would be necessary to keep track of eye distance to the object relative to the object size. Breaking up concave objects into several convex ones or simply switching to a different culling method when up close are easier fixes. Even if the alternative method is not as efficient, being close to the object in question means that a large part of the scene is occluded and will not be rendered—therefore, we have some performance to spare.

Miscellaneous tweaks:
Smaller rendermap makes bookkeeping more manageable—for example, we can reuse the rendermap from the previous frame if the central sector did not change. A per-face culling approach would require us to keep tabs on many thousands of faces if we wanted to reuse previous frame data, which would be detrimental to performance.

Performance:
This algorithm really shows its advantage over the brute force rendering in complex scenes, where both the memory bandwidth and the rendering capacity of the graphics subsystem get overwhelmed (Table 1). The absolute performance of the brute force approach is much lower, as is evident from the graph. However, our algorithm also scales much better with increased load. The actual number of triangles/sec rendered by the brute force approach is 4,998,200 in the simplest scene to 4,038,360 in the most complex one—a decrease in efficiency of 10%. In contrast, our algorithm achieves 192,780\(\times 43 = 8,289,540\) triangles in the most complex scene versus 7,068,600 in the simplest scene—an efficiency increase of 17% while in the 5 degree sectors mode. (It also seems that having sectors spanning less than 5 degrees will not gain any more performance). Such efficiency could benefit certain real-time 3D applications, games in particular—it would allow a painless increase in the detail level of static objects in the world, which traditionally have quite economical polygon budgets compared to the characters.