

Graphics pipeline: Depth buffer, Back-face culling, Triangle Strips and Fans

Philosophical idea: instead of processing all the primitives (triangles) for each pixel, like in the simple variant of ray tracing you implemented for the first project, we go over all the primitives and, for each of them process all pixels covered by it. It turns out that processing of all the pixels covered by one primitive can be performed very efficiently, with tiny amount of work per each such pixel (how – we will see later on). Thus, the total cost of this approach can be broken into two major components:

1. Vertex processing cost: e.g. projecting to the screen and color computation
2. Primitive processing: basically, writing the triangles into the frame buffer; This involves finding all the pixels covered by the projection of the primitive, computation of their depth and color; we skip the details for now, let's just say that the amount of work needed to be done per pixel is VERY small.

1 Depth Buffer

The idea: keep the depth (something much like the value of the t parameter you computed in the ray-tracing program) for each pixel. The value of t will allow to figure out whether the primitive we are currently processing is behind whatever was processed before or not. If it is, it need not be drawn out since it is invisible. Otherwise, we will draw it out since there is a chance that it is visible (I said 'there is a chance...' since there might be other primitives which will be processed in the future which will obscure the currently processed one; even if the depth of the current primitive is less than any other one we have looked at so far, we cannot be sure that it will stay visible until the end). For each primitive and pixel covered by its projection, we compute the depth too; the pixel is drawn out only if the new depth value is smaller than what was stored at that pixel before. By 'drawn out' we mean not only drawing the polygon in its color, but also updating the depth value. Before the drawing starts, the depth is initialized to something larger than any possible depth; think of it as plus infinity.

2 Back Face Culling

It is used to spare some of the work (mostly in the primitive processing component). In order to make it work, all the triangles have to be oriented so that their vertices appear counterclockwise when looked at from outside (just a convention.... clockwise would work too with obvious changes). Now, after the triangle's vertices are projected, and they turn out to come in the clockwise order, the projection of that triangle is not drawn out at all. Even more than

that: we even do not compute the covered pixels, depths nor colors. Thus, no work is done in step 2.

3 Triangle Strips

This is a way to reduce the number of vertices that are processed. All in all, it is a rather dirty compromise between this goal and the fact that we need to be able to code the procedure in hardware, using a small fixed number of 'registers' (here: three). The triangle strip definition is shown in Figure 1. The right way to use strips is to cover the object to be rendered with them; the strips should be made as long as possible: the longer they are, the more savings in the rendering time (Why? Can you count the number of vertices which have to be processed?). Computing the optimal layout of triangle strips is a hard problem, but it often can be solved nicely for special cases.

Triangle fans work like triangle strips. See Figure 1.

Polygons are often broken into triangles and so they are processed pretty much like triangle fans are 1.

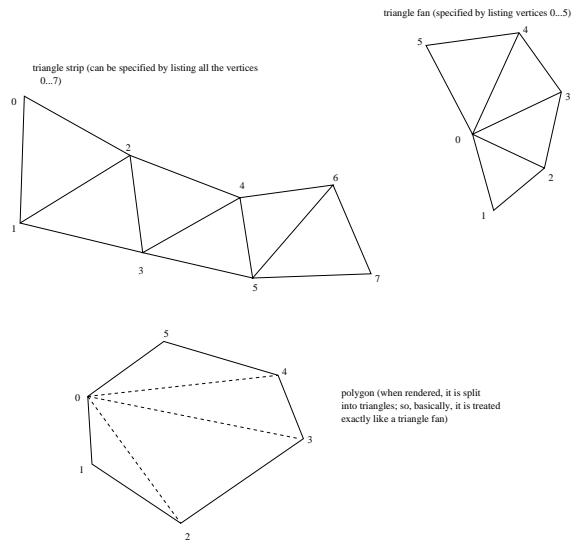


Figure 1: Triangle strips, fans and polygons

How does it work? If the triangles on the strip on the Figure were to be processed separately, we would have to project a vertex as many as 18 times (3 times per triangle). If we draw the triangles as a triangle strip, we need to project each vertex only once, which means we need to do 8 projections. Much less than 18. Triangle fans work in a similar way. To draw a triangle strip, you first tell OpenGL to interpret the forthcoming sequence of vertices as a specification of a triangle strip. Then you specify the strip's defining vertices

in order. OpenGL will know that it has to interpret each consecutive three as bounding a different triangle.

4 Display Lists

A display list is a data structure whose purpose is to store OpenGL calls in a compact and efficient form. They help improve the efficiency by storing all the arguments of all the calls in a way close to the internal format used by the graphics hardware. Also, the arguments are stored as *constants*, which means that no arithmetic calculations (like the trig functions etc which we need to do when drawing a torus or a sphere) are needed to compute the vertex data. Thus, the calls are executed much more efficiently. A nice thing is that building this data structure is no different than actual rendering from the programmers perspective. All you have to do is to tell OpenGL: 'put the calls I am making to a display list without drawing things out into the frame buffer' or 'put those calls to a display list and draw everything simultaneously', say that you are done after you are and your display list ready to use!