

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

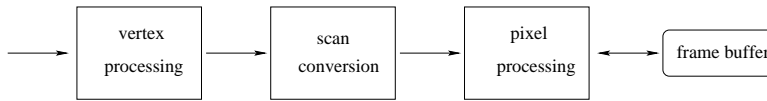


Figure 1: A simplified view of the graphics pipeline

Ray tracing as we talked about so far can be viewed as based on the following loop (we concentrate only on resolving visibility here):

```
for each pixel p do
  for each primitive X do
    compute the intersection of ray from the viewpoint through
      the pixel p and X, keeping track of the closest one
    compute the color for p based on this closest intersection
```

The graphics pipeline implemented in today's graphics hardware is based on a similar double loop in which the inner and outer `for` statements are swapped:

```
for each triangle T do
  for each pixel p in the projection of T onto the screen do
    plot the pixel, using T's material properties
    to compute color
```

Of course, there is a problem with the second loop: it's not hard to see that, unless the triangles are drawn in back-to-front order, when we draw triangles that are more distant we can overwrite the triangles that are closer to the viewpoint. Fortunately, this problem can be solved by applying the *depth buffer algorithm*, described below.

One of the reasons for the success of this approach is that it can be implemented as a pipeline architecture (Figure 1), simple enough to be possible to realize in hardware. Note that Figure 1 shows a highly simplified view: each of the stages is typically implemented as a pipeline too. The three main stages of the pipeline are:

1. **Vertex processing:** The modeling transformation and the perspective transformation are applied to the vertices; For each vertex, we compute the colors (typically, the vertices will have associated normal vectors) or some other quantities derived from vertex data that can be used e.g. to do more complex shading during the pixel processing stage or for some other purpose
2. **Rasterization:** The triples bounding individual triangles are extracted from the vertex stream. For each triangle, pixels belonging to its projection onto

the screen are computed. All data travelling with the vertices is linearly interpolated and assigned as pixel data (in particular, this happens to depth or color in Gouraud shading)

3. Pixel processing: For each of the pixels generated in the previous stage, we execute the depth test and draw the pixel out if it passes that test (see below). In newer graphics hardware, it's possible to program this stage, e.g. to perform complex shading (e.g. one of the common Phong shading implementations is actually performed here).

## 1 Depth Buffer

Let's keep the depth value for each pixel. A good depth is some quantity behaving like the value of the  $t$  parameter you computed in the ray-tracing program: it has to increase as the distance from the viewpoint increases. Take a look at the transformation (projection) notes for more precise description of what depth is based on. Depth will allow to figure out whether the primitive we are currently processing is behind whatever was processed before or not (this is as simple as comparison of the depth of the current primitive at a pixel to the depth stored for the same pixel in the depth buffer). If it is, it need not be drawn since it is invisible. Otherwise, we will draw it 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 all triangles are processed). Remember that the depth test is performed on per-pixel basis and depth buffer stores a depth value for each pixel.

For each triangle and pixel covered by its projection, we compute the depth of that pixel by linearly interpolating depth values from the vertices. Note that this works because the perspective transformation takes triangles into triangles (that's why we don't use the  $t$  parameter as the depth: even though it's much more intuitive, linear interpolation of  $t$  from vertices wouldn't produce the correct  $t$  values in the interior of the triangles!). 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 assigning the RGB values (computed in the usual way using the triangle's material properties) in its color, but also updating the depth value.

## 2 Back Face Culling

It is an optimization technique allowing to (conservatively) reject some of the back-facing triangles. 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 triangle is not drawn. In particular, do not even have to compute the pixels of the screen its projection covers, their depths or colors. Thus, some work is spared in the pixel processing stage.

### 3 Triangle Strips

This is a way to reduce the number of vertices that are processed. All in all, it is a 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 2. 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 2.

Polygons are often broken into triangles and so they are processed pretty much like triangle fans are 2.

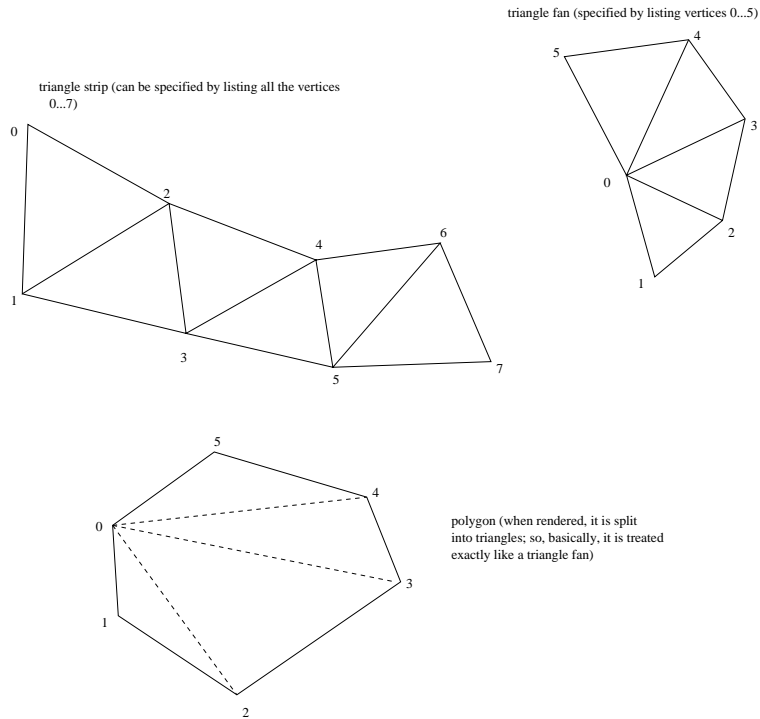


Figure 2: Triangle strips, fans and polygons

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. The information how to interpret the vertex stream is mainly used by the rasterizer (when it puts together triangles to be rasterized; this is sometimes called triangle setup).