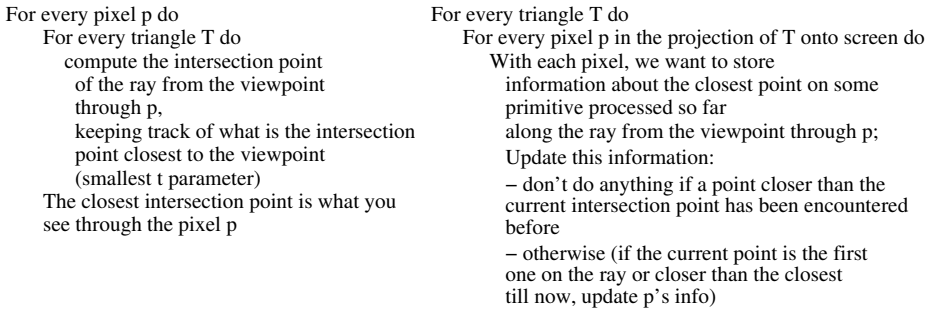

Figure 1: Two ways of handling visibility: left: Ray tracing, right: graphics pipeline.

1 Pipeline

1.1 Ray tracing vs Pipeline

Let us focus on determining what is visible through a pixel and what is not, which is obviously an important step in getting the right image.

Ray tracing computes visibility by running a double loop shown in Figure 1, left. Graphics pipeline runs the equivalent double loop but nested the other way around, Figure 1, right. Graphics pipeline requires storing *depth* for each pixel. Depth measures the distance between the viewpoint and the closest intersection point of a ray originating from the viewpoint through the pixel. It does not need to be equal to that distance, but it has to be a monotonous function of it, i.e. needs to increase with the distance between the viewpoint and the projected point. For every triangle we update the depths of all pixels in its projection to the screen so that they correctly reflect which of the primitives processed up to now is the closest along the ray from the viewpoint through that pixel.

1.2 Pipeline structure

All that can be organized in the following way. We break the computation into three major stages shown in Figure 2 (actually, there are a lot more, but let's focus on the big picture).

1. **Vertex Processing.** The input is a stream of vertex records (containing coordinates, normals, colors or reflective properties and possibly other data). The output is a stream of vertices in screen coordinates (coordinates of the vertex projected to the screen, depth, results of operations on the original vertex data). This stage includes, the perspective transformation.

2. **Triangle Setup and Rasterization.** Here, triangles are built from the projected vertices. One of the ways to do that is to split the vertex stream into triples and treat each triple as vertices of a triangle. For each triangle, we generate a *fragments*, which correspond to pixels in the projection

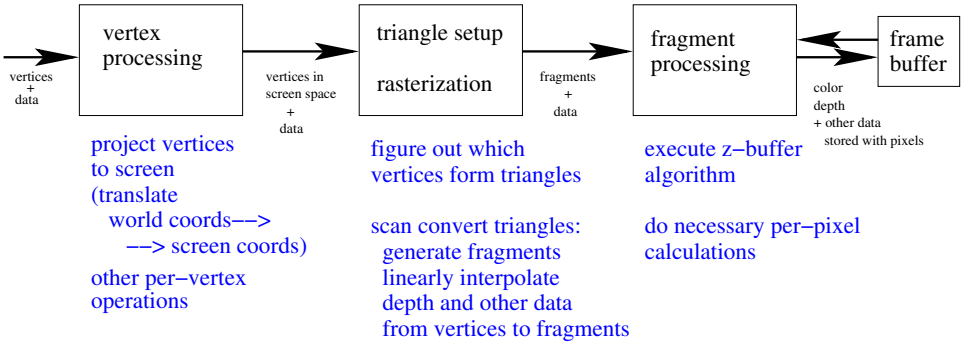

Figure 2: A schematic and somewhat simplistic view of a standard graphics pipeline.

of that triagle to the screen. Each fragment has x- and y-coordinates (describing its location on the screen), depth and other associated data. All this data (including depth) is obtained from the data associated with the triangle's vertices by linear interpolation.

3. **Fragment Processing.** Some calculations are possibly done on the fragment's data. Color and depth is written to the frame buffer. The z-buffer algorithm is applied to resolve the visibility correctly: a fragment's color and depth is written into the frame buffer only if the depth of the fragment is less than the depth of the pixel in the frame buffer at the same location.

2 Color in the pipeline: shading models

There are three classical ways of dealing with light and shading in the context of graphics pipeline. They are referred to as *shading models*. All of them are based on an illumination model similar to the one we discussed for ray tracing. Recall that computing illumination requires knowing the normal vector and material properties (ambient, diffuse and specular coefficients and specular exponent). We are going to be sending all of the above with each vertices pushed into the graphics pipeline. We will also inform the pipeline of global parameters needed to compute illumination: location, color and intensities of light source(s) and color and intensity of ambient light.

2.1 Flat shading

Flat shading is fast, but has lowest quality. The idea is to compute color for each vertex at the vertex processing stage and send it out with the projected vertices to the rasterizer. For each triangle, the rasterizer determines a **single** color that it will use to color all fragments obtained from that triangle. The

precise way of doing this may depend on the specific graphics library, it could be a color of one of the vertices or the average of vertex colors.

Flat shading uses just one color for each triangle, so it doesn't look well, especially if triangles used to tile the surfaces are large: typically one can easily see many of the triangles' boundaries since the color is discontinuous there. On the other hand, it is very fast: pretty much all of the necessary work is performed during the vertex processing stage. The only thing the rasterizer needs to do is to decide what color to use for each triangle. No linear interpolation of color data from vertices is needed.

2.2 Gouraud shading

Gouraud shading also computes colors on per-vertex basis during vertex processing stage. However, linear interpolation of color is used to determine the colors of the fragments generated for each triangle. This leads to colors that are continuous across edges. Gouraud shading leads to better quality images, but it requires more work than flat shading (the difference is in linear interpolation of color that needs to be done during the rasterization stage).

One drawback of Gouraud shading is that it has serious problems with highlights, especially for smooth surfaces. Imagine you are drawing a very shiny surface. A perfect image would show small specular highlights on the surface. But if the triangles are tile the surface in such a way that their vertices are somewhat off the place where the highlight should be, we are likely to miss it (or at least make it darker than it should be) in a Gouraud-shaded image. This is because color is computed on per-vertex basis and therefore all colors are going to be darker than the desired brightness of the highlight. Linearly interpolated colors will also be too dark.

2.3 Phong shading

Phong shading attempts to deal with the highlight problem by computing colors on per pixel rather than per triangle basis. This works in the following way. No illumination calculation is done on the vertex processing stage. The original vertex normals are simply passed on as projected vertex data to the rasterizer. The rasterizer linearly interpolates the normals and sends the resulting interpolated normals with the fragments. Finally, the illumination formula is evaluated for each fragment (using the interpolated normal). In the case of Phong shading, most work is done on the pixel processing stage. Like in the case of Gouraud shading, interpolation is also needed during rasterization (but here we interpolate normals, not colors). Nothing exciting happens on the vertex processing stage. Note that one of the things that needs to be done for each fragment is the normalization of its interpolated normal vector (even if the vertex normals are unit length, the interpolated normals need not be unit length) – this is expensive.

Phong shading generally produces better highlights for shiny materials: the interpolated normals are an attempt to mimic the distribution of the smooth

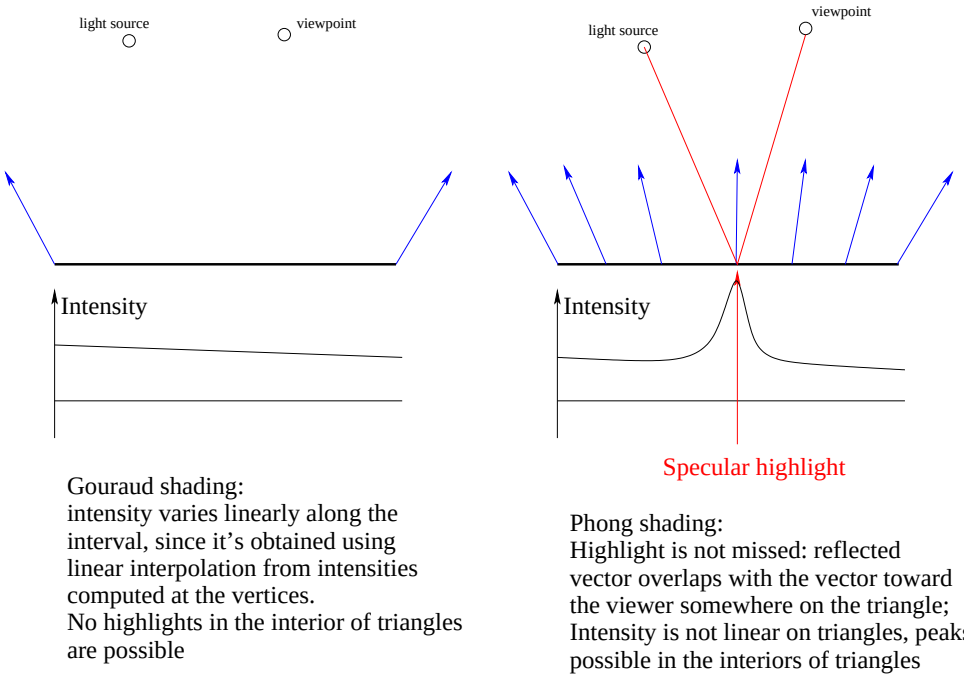

Figure 3: Comparison of Gouraud shading with Phong shading.

surface that is represented using the triangles, so highlights of roughly correct intensity in roughly right place will appear under reasonable conditions (see Figure 3).