

## OpenGL Rendering (9/5/6)

*Lecturer: James Kuffner**Scribes: Rosen Diankov***Rendering Pipeline**

- Primitives - Vertices, Normals, Colors
- Transformations - *move objects*
- Clipping against view frustum - *Is it visible?*
- Projection -  $3D \rightarrow 2D$
- Rasterization - *fill in triangles*

**1 Color**

**Color Gamut** - Subset of colors that are represented by a given output device.

**Color Interpolation** - Can imagine a cube in 3D with each coordinate representing red, green, and blue. Interpolation happens on the straight line joining two points  $(R_1, G_1, B_1)$  and  $(R_2, G_2, B_2)$ .

**HSV Color Scale** - There are many colors scales besides RGB. Another popular color scale is Hue, Saturation, and Value where.

- Value - how bright the color is
- Saturation - vibrancy of the color
- Hue - color shade (or color type)

**2 Lighting**

The simplest description of the final color on an object can be described by the combination of 3 main types of light: ambient, diffuse, and specular.

- Ambient - Viewer independent, Light independent. The color usually comes from light bouncing around the whole scene.
- Diffuse - Viewer independent, Light dependent. The more a face is oriented toward the light, the brighter it is.
- Specular - Viewer dependent, Light dependent.

Consider that a face is defined by 3 points in counter-clockwise order:  $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ , then the vector normal (orthogonal) to the face is defined by:

$$\mathbf{n} = \frac{\mathbf{a} \times \mathbf{b}}{\|\mathbf{a} \times \mathbf{b}\|}$$

where

$$\mathbf{a} = \mathbf{v}_2 - \mathbf{v}_1 \quad \text{and} \quad \mathbf{b} = \mathbf{v}_3 - \mathbf{v}_1.$$

The cross product can be represented nicely by the determinant of a 3x3 matrix:

$$\mathbf{N} = \begin{vmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = (a_y b_z - a_z b_y)\hat{\mathbf{x}} - (a_x b_z - a_z b_x)\hat{\mathbf{y}} + (a_x b_y - a_y b_x)\hat{\mathbf{z}}$$

$$\text{then, } \mathbf{n} = \frac{\mathbf{N}}{\|\mathbf{N}\|}$$

Before computing lighting, note that there will be faces that face away from the viewer. The process of removing those faces is called *backface culling*. Imagine that the vector from the viewer position to the face is  $\mathbf{d}$  and the face normal is  $\mathbf{n}$ . Then faces facing toward the viewer will have  $\mathbf{d} \cdot \mathbf{n} < 0$ , any other faces should be ignored.

Once all the face normals are computed, the diffuse and specular components can be computed. If a simple directional light is used, each face will be colored with only one color. But the color where two faces meet will be discontinuous because the normals themselves aren't. This is because each face only roughly approximates the underlying surface of the object. Imagine that an array of faces is used to approximate a sphere, then each face will be representing a small curved surface of the sphere. So a face's normal is really just a rough approximation of the surface's normal. To get a smooth continuous color from the object, we can use vertex normals instead of face normals.

A grossly rough and fast approximation to calculate the colors is to calculate the diffuse and specular terms once for each vertex using the vertex normals, and then linearly interpolate across the rest of the pixels on the face. Triangles that share the same vertices will share the same color, so there won't be anymore discontinuities.

A more accurate scheme for getting smoother colors is called **Phong Shading**. The idea is to interpolate the vertex normals across each pixel rather than the colors. This falls under the category of *per-pixel* shading because all the lighting equations have to be recomputed for each pixel. The other methods that compute the lighting equations for each vertex fall under *per-vertex* shading. The traditional OpenGL pipeline supports *per-vertex* shading only; however, due to recent advances in GPUs, *per-pixel* shading is possible<sup>1</sup>.

### 3 OpenGL Notes

OpenGL usually has a render target for storing color and a depth buffer for storing the depths of the pixels. Usually a render loop will consist of: clearing all buffers, drawing stuff, and swapping the buffers. Swapping is an old technique that alternates the buffers that the application renders on. The reason is so that the application doesn't have to wait until the OS is finished displaying the previous frame before the application

<sup>1</sup>The newest generation of game consoles use per-pixel shading only.

can start overwriting the data for the newer frame. Only the color render targets are swapped, the depth buffer can be reused across frames.

Each color render target usually has a 4th channel, the alpha channel, whose main function is to control transparency (alpha blending). Turn it on with **glEnable(GL\_BLEND)**. Depending on the OpenGL blending mode, if a pixel with color *src* is drawn with an alpha value of  $\alpha$  and the previous color in the render target is *dest*, then the final color is a blend between the two:

$$final = \alpha src + (1 - \alpha) dest$$

See **glBlendFunc** for more details.

Another point worth noting is that the *depth writes* are usually turned off by **glDepthMask** when using alpha blending to avoid depth testing artifacts.