Hardware Rendering

Brian Curless
CSE 557
Fall 2015

Reading

Required:

- Shirley, Ch. 7, Sec. 8.2, Ch. 18

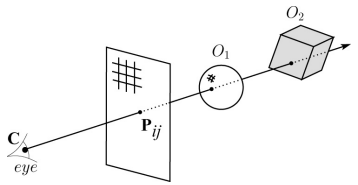
Further reading:

- Foley, et al, Chapter 5.6 and Chapter 6
- David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, 2nd Ed., McGraw-Hill, New York, 1990, Chapter 2.
- I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1-55, March 1974.

Going back to the pinhole camera…

Recall that the Trace project uses, by default, the pinhole camera model.

If we just consider finding out which surface point is visible at each image pixel, then we are **ray casting**.

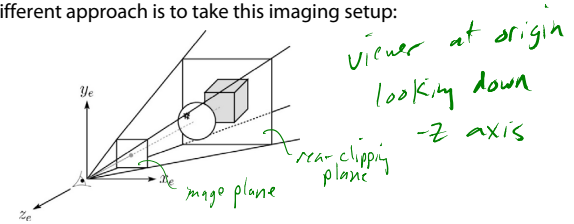


For each pixel center $\boldsymbol{P}_{ij}$

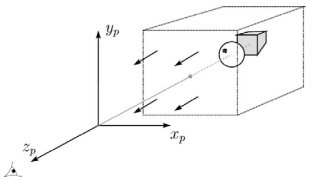
- Send ray from eye point (COP), $\mathbf{C}$, through $\boldsymbol{P}_{ij}$ into scene.
- Intersect ray with each object.
- Select nearest intersection.

Warping space

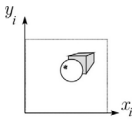
A very different approach is to take this imaging setup:



then warp all of space so that all the rays are parallel:



and then just drop the z-coordinate and draw:

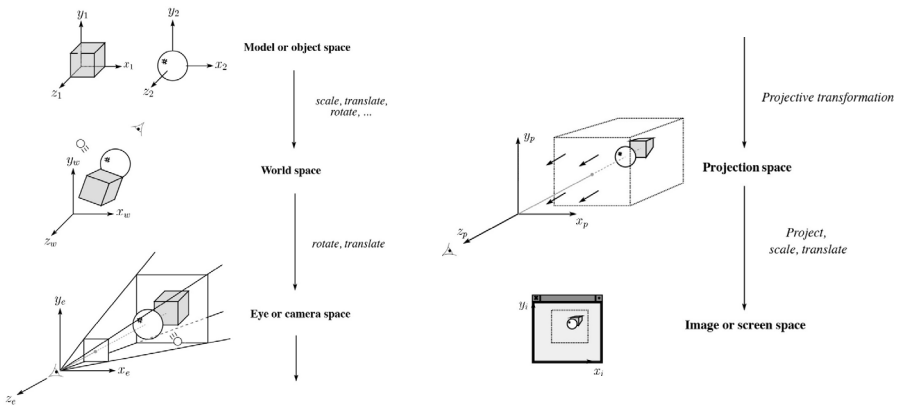


In practice, we keep track of the z-coordinate during drawing to determine visibility.

3D Geometry Pipeline

Graphics hardware follows the "warping space" approach.

Before being turned into pixels, a piece of geometry goes through a number of transformations...



Z-buffer

The **Z-buffer** or **depth buffer** algorithm [Catmull, 1974] can be used to determine which surface point is visible at each pixel.

Here is pseudocode for the Z-buffer hidden surface algorithm, for a viewer looking down the z axis (bigger – i.e., more positive – z's are closer):

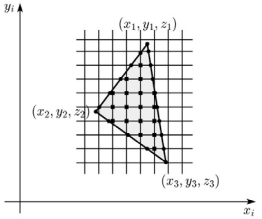
```
for each pixel (i, j) do
    Z-buffer [i, j] ← FAR
    Framebuffer [i, j] ← <background color>
end for
for each triangle A do
    for each pixel (i, j) in A do
        Compute depth z of A at (i, j)
        color ← shader(A, i, j)
        if z > Z-buffer [i, j] then
            Z-buffer [i, j] ← z
            Framebuffer [i, j] ← color
        end if
    end for
end for
```

Q: What should *FAR* be set to? ~∞ - BIG NUMBER

Rasterization

The process of filling in the pixels inside of a polygon is called **rasterization**.

During rasterization, the z value can be computed incrementally (fast!).



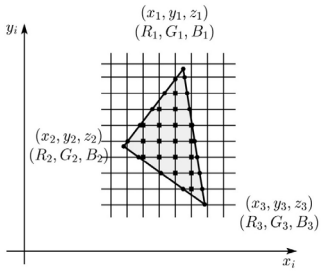
Curious fact:

- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for huge memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.

Rasterization with color

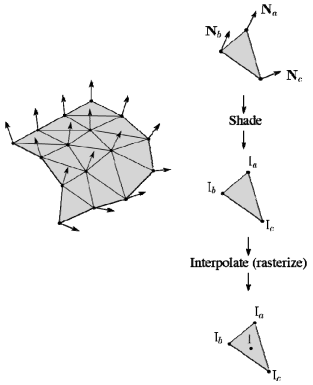
During rasterization, colors can be smeared across a triangle as well:



Gouraud interpolation

Recall from the shading lecture, rendering with per triangle normals leads to faceted appearance. An improvement is to compute per-vertex normals and use graphics hardware to do **Gouraud interpolation**:

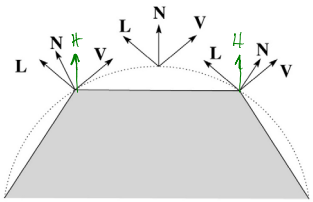
1. Compute normals at the vertices.
2. Shade only the vertices.
3. Interpolate the resulting vertex colors.



Gouraud interpolation artifacts

Gouraud interpolation has significant limitations.

1. If the polygonal approximation is too coarse, we can miss specular highlights.



2. We will encounter **Mach banding** (derivative discontinuity enhanced by human eye).

This is what graphics hardware does by default.

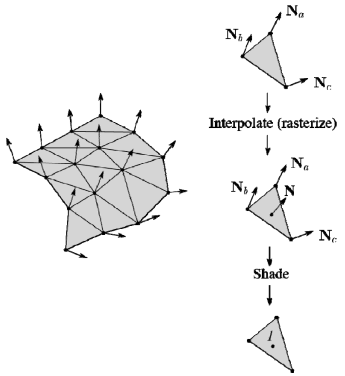
A substantial improvement is to do…

Phong interpolation

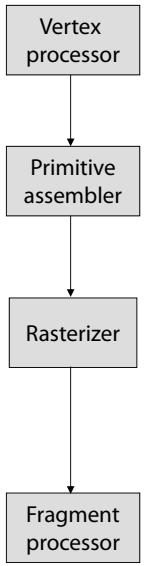
To get an even smoother result with fewer artifacts, we can perform **Phong *interpolation***.

Here's how it works:

1. Compute normals at the vertices.
2. Interpolate normals and normalize.
3. Shade using the interpolated normals.



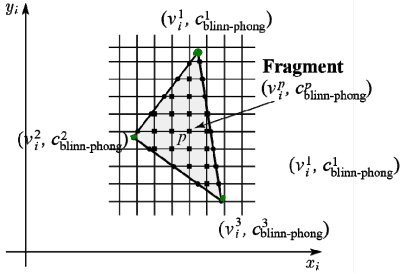
Old pipeline: Gouraud interpolation



Default vertex processing:

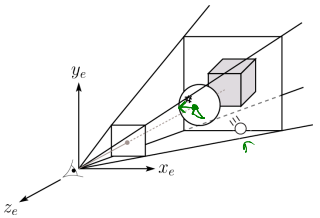
- $L \leftarrow$ determine lighting direction
- $V \leftarrow$ determine viewing direction
- $N \leftarrow$ normalize(n_e)
- $c_{\text{blinn-phong}} \leftarrow$ shade with L, V, N, k_d, k_s, n_s
- attach $c_{\text{blinn-phong}}$ to vertex as "varying"
- $v_i \leftarrow$ project v to image

$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle

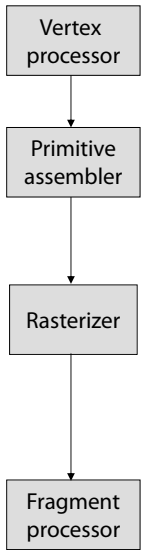


Default fragment processing:

color $\leftarrow c^p_{\text{blinn-phong}}$

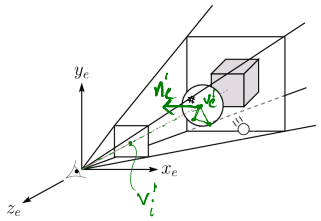


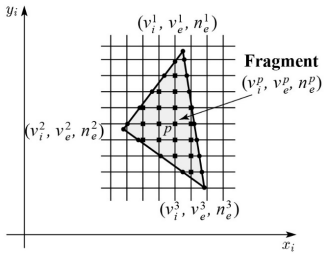
Programmable pipeline: Phong-interpolated normals!



Vertex shader:
attach n_e to vertex as "varying"
attach v_e to vertex as "varying"
$v_i \leftarrow$ project v to image

$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle





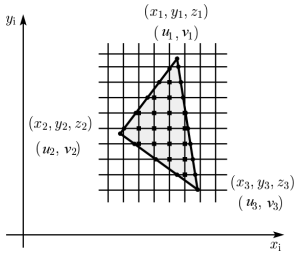
Fragment shader:
$L \leftarrow$ determine lighting direction (using v_e^p)
$V \leftarrow$ normalize$\left(-v_e^p\right)$
$N \leftarrow$ normalize(n_e^p)
color $\leftarrow$ shade with L, V, N, k_d, k_s, n_s

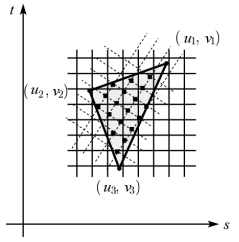
Texture mapping and the z-buffer

Texture-mapping can also be handled in z-buffer algorithms.

Method:

- Scan conversion is done in screen space, as usual
- Each pixel is colored according to the texture
- Texture coordinates are found by Gouraud-style interpolation





Note: Mapping is more complicated to handle perspective correctly!

Shading in OpenGL

The OpenGL lighting model allows you to associate different lighting colors according to material properties they will influence.

Thus, our original shading equation (for a point light):

$$I = k_e + k_a I_{La} + \sum_j \frac{1}{a_j + b_j r_j + c_j r_j^2} I_{L,j} B_j \left[k_d \left(\mathbf{N} \cdot \mathbf{L}_j \right) + k_s \left(\mathbf{N} \cdot \mathbf{H}_j \right)_+^{n_s} \right]$$

becomes:

$$I = k_e + k_a I_{La} + \sum_j \frac{1}{a_j + b_j r_j + c_j r_j^2} \left[k_a I_{La,j} + B_j \left\{ k_d I_{Ld,j} (\mathbf{N} \cdot \mathbf{L}_j) + k_s I_{Ls,j} (\mathbf{N} \cdot \mathbf{H}_j)_+^{n_s} \right\} \right]$$

where you can have a global ambient light with intensity I_{La} in addition to having an ambient light intensity $I_{La,j}$ associated with each individual light, as well as separate diffuse and specular intensities, $I_{Ld,j}$ and $I_{Ls,j}$, repectively.

Materials in OpenGL

The OpenGL code to specify the surface shading properties is fairly straightforward. For example:

```
GLfloat ke[] = { 0.1, 0.15, 0.05, 1.0 };
GLfloat ka[] = { 0.1, 0.15, 0.1, 1.0 };
GLfloat kd[] = { 0.3, 0.3, 0.2, 1.0 };
GLfloat ks[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat ns[] = { 50.0 };
glMaterialfv(GL_FRONT, GL_EMISSION, ke);
glMaterialfv(GL_FRONT, GL_AMBIENT, ka);
glMaterialfv(GL_FRONT, GL_DIFFUSE, kd);
glMaterialfv(GL_FRONT, GL_SPECULAR, ks);
glMaterialfv(GL_FRONT, GL_SHININESS, ns);
```

Notes:

- The `GL_FRONT` parameter tells OpenGL that we are specifiying the materials for the front of the surface.
- Only the alpha value of the diffuse color is used for blending. It's usually set to 1.

Shading in OpenGL, cont'd

In OpenGL this equation, for one light source (the 0th) is specified something like:

```
GLfloat La[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat La0[] = { 0.1, 0.1, 0.1, 1.0 };
GLfloat Ld0[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat Ls0[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat pos0[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat a0[] = { 1.0 };
GLfloat b0[] = { 0.5 };
GLfloat c0[] = { 0.25 };
GLfloat S0[] = { -1.0, -1.0, 0.0 };
GLfloat beta0[] = { 45 };
GLfloat e0[] = { 2 };

glLightModelfv(GL_LIGHT_MODEL_AMBIENT, La);
glLightfv(GL_LIGHT0, GL_AMBIENT, La0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, Ld0);
glLightfv(GL_LIGHT0, GL_SPECULAR, Ls0);
glLightfv(GL_LIGHT0, GL_POSITION, pos0);
glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a0);
glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION, b0);
glLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, c0);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, S0);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, beta0);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, e0);
```


Shading in OpenGL, cont'd

Notes:

You can have as many as GL_MAX_LIGHTS lights in a scene. This number is system-dependent.

For directional lights, you specify a light direction, not position, and the attenuation and spotlight terms are ignored.

The directions of directional lights and spotlights are specified in the coordinate systems *of the lights*, not the surface points as we've been doing in lecture.