1 Rasterization

How do we draw polygons on the screen?

1.1 DDA algorithm

Since we know that the equation for a line is

$$\frac{dx}{dy} = \frac{y_2 - y_1}{x_2 - x_1}$$

, we can focus on drawing the closest integral value of y for every x. However, for steep lines, there will be many blank spaces until y gets rounded to the next whole number. The solution is to do the rounding for every y instead of every x.However, this takes an extra addition per step.

1.2 Bresenham's algorithm

With BA, we don't need the extra floating-point addition. Instead, we replace it with multiplication and comparisons, which are easier to implement. We basically compare the δy to see if it's greater than δx. This makes it way more efficient(w/ bit shifting).

1.3 Circle Drawing Algorithms

We can use the same idea as Bres. to plot circles. Since circles are defined by $f(x, y) = x^2 + y^2 - r^2$, we can just measure the value of a pixel M, and compare $f(M)$ with r. If $f(m) > r^(2)$, choose SE point, otherwise NE, and so on. Same idea can apply to other polynomials.

1.4 Polygon filling

Convex polygons are preferred b/c they have well-defined interiors. Easier to break down into triangles, b/c any interior edge will only remain in the interior. Several ways to test for points:

1. **Odd-even filling**: A point in p must cross an odd num of edges to go to infinity. Outside points cross even num edges. The **winding test** measures how many times a point is encircled by travelling along a direction. If 0, don't color, otherwise yes.
2. **scanlines**: we draw a horizontal line across the screen and measure all the intersections. Then we sort the intersections by x-coordinate. We fill the consecutive vertices that are part of the polygon.
3. **flood-fill**: The flood-fill algorithm is similar to the CPE, we recursively call a fill function on neighboring points if they are inside the polygon too.

2 Hidden Surface

To fill the polygons that are visible, we need to be able to determine intersections within the polygons.

2.1 Culling

If the viewing vector v and the normal of the polygon have an angle $90 \geq \theta \geq -90$, poly will be visible. We have two options: **object space approach** and **image space approach**.

2.1.1 object space approach

We use pairwise testing for every pair of polygons in the scene. We determine if either one completely obstructs the other one, if there is only partial obstruction, or neither is overlapping. Takes minimum $O(n)$.

2.1.2 Image space approach

From the COP, we can draw other rays coming out of the projection plane for each pixel. We follow the ray's intersections with the k polygons in the scene, and color it the same color of the first polygon it intersects. For an nxm display, it takes $O(nmk)$.However, since in practice the algorithms perform better, this is mostly used.

2.1.3 z-buffer algorithm

We maintain a z-buffer, which will store the color of the closest(z-direction) element. Done at the same time as fragment processing. Efficient b/c can be implemented in hardware. Whether perspective is screen space or world coords doesn't matter, relative distances preserved.

2.1.4 Depth Sort and Painter's Algorithm

"Paint over" the farthest, obscured elements, b/c we do *back-to-front rendering*. For two different polygons with identical z-coordinates, we try to apply Depth-sort to figure out an order in which to draw the figures. For cyclic overlap, we can just divide the polygon into smaller ones and render them separately, since an ordering might not be posssible.

2.2 Screen Space vs. 3D Space

By following the formula $P(m) = P_1 + mP(P_2 - P_1)$, we can find the relation between tow points on the two different coordinate systems.

2.3 Space Partitioning

To avoid rendering unnecessary objects, we can create different data structures that partition the space.

2.3.1 Octree

In 2D, we can partition the space into 4 different quads, 3D it's octs. We can do it while there are more than one color per area.

2.3.2 BSP Trees

By drawing planes, we can divide the 3D plane into two half-spaces, and subsequent divisions allow us to more efficiently calculate visibility of different objects w/ the camera. Creating the ordering of how to draw the polygons becomes easier since we have to compare fewer elements.

2.4 Portal Culling

Here, we divide the space into cells and can only move to other cells through "portals"?

3 Buffers & Mapping Techniques

Mapping takes place during the fragment processing. For more complicated shapres, we can first map to an intermediate shape before passing on to the desired shape. We can use parametric equations to map textures to different objects.

3.1 Interpolation

Interpolation will take the samples of different adjacent spaces in the world. This can lead to *funky* effects. We can further subdivide the figure and interpolate again if we map to screen space. *point sampling* can be the easiest, but aliasing errors can still happen. A best approach is to average the values of an entire area.

3.2 minimaps

Minimaps are textures of different resolutions. Depending on the distance of the object, we can apply different sized images. This helps with different object interpolations.

3.3 Environment Mapping

For highly reflective surfaces, we can apply the environment onto the object to get a reflective look.

3.4 Bump & Normal Mapping

With normal mapping, we just map the texture normally. For bump mapping, we apply a distortion to the normal vectors. $P' = P + bN$.

3.5 Opacity, Blending, etc...

To establish opacity, we can provide a constant factor and a blending equation to map a texture to objects behind it.

3.6 Accumulation Buffer

To apply compositing and fog, we need a bigger buffer space than allowed by the frame buffer. We use the *Accuulation Buffer* for things like Compositing, Image Filtering, anti-aliasing, motion effects, etc..

4 Curves and Surfaces

4.1 parametric curves

These curves map some parameter space to 3d space (maybe each var is defined by different equation?) Use them bc they are more general than polygon meshes, normals easier to define, easier to animate, texture coords easy to map.

Polynomial curves allow us to define curves, but finding the coefficients is not efficient (can use least squares).

4.2 Hermite Curves

Way of approxing curves by using polynomial functions with certain derivatives given points $x_1, x_2, ..., x_n$. To connect points x_i and x_{i+1}, we need to know the values at those points and their derivatives. We can approx the curve by the derivatives at those points.

4.3 Bezier Curves

A bezier curve involves the tangent of the source and control points. At each time value t, we draw a line from the line segment P_0P_1 to the proportional equivalent on the other control line. This new segment will be tangential to the point on the Bezier curve we wish to draw. The point is then drawn on the proportion of the new segment. We can approx them using *Bernstein Polynomials*. We join the control points and draw their tangents as the line. Different types of continuity:

- C_0: No breaks in surface
- G_1: tangent at joing has same direction.
- C_1:tangent at joint has same dir. and magn.
- C_n:curve continuous through nth derivative.

5 Beyond Rendering

Normal lighting models don't take other surfaces for calcs. Rays can come out of light source or eye. Refraction occurs out along the normal or is reflected if $\theta > 90$

5.1 whitted ray tracing

For each pixel, we trace a primary ray to the first visible surface.

- **primray ray**: ray from eye to scene.
- **shadow ray**: in dir L to light sources.
- **reflected ray**: in dir R off of the normal.
- **refracted ray**:(transmitted ray) in dir T.

In spuersampling, we emit multiple rays off of the pixel to find avg. of area(slow).Also this is img. space. Complicated rendering equation.

5.2 radiosity

We divide the scene into "patches", and all surfaces are diffuse reflectors.

$$b_i a_i = e_i a_i + \rho_i \sum_{j=0}^{n} f_{ji} b_j a_j$$

- ρ_i:reflectance of element i(given).
- b_i: color of patch i.(unknown)
- a_i:the area of patch i.(computable)
- e_i:emmissive component (given).
- f_{ji}:form factor (j->i)(computable)

5.3 Form Factor

F_{ji} Fraction of light leaving element j and arriving at element i. Depends on shape of the patches, their orientation, distance, and visibility.
We have different ways of solving the equation:

- **Jacobi's method**:we need two copies of radiosity vector B, doesn't converge quickly.
- **Gauss-Siedel**:no additional copies, converges quickly.

5.4 Calculating Form Factors

Simple way is to use ray tracing & point-to-area form factors.

5.4.1 Hermitude Algorithm

On top of every patch, we have a hermicube, which is divided into pixels. Then, we project the patch on the faces of the hermicube. Then the form factor for a particular patch is just the sum of the form factors of the individual pixel.

6 filler

- geometry shader: Can draw new geometric shapes, sits between vertex and fragment shader.
- vertex shader: applies effects per vertex, calcs normal, and light reflection.
- fragment shader: calcs color of every vertex.

mipmapping: refers to creating smaller textures since one texture coordinate might be smaller/greater than one individual vertex/pixel. This means that we can save ourselves some of the hard work if there is no need to render all the texture.

6.1 photon mapping

Also composed of bi-directional paths, which take into account the paths from the light sources as well as from the eye. Then we cache the photons that went along the path from the source.