Topic 6 - Rendering and the Graphics Pipeline

Cameras

We have a virtual camera which acts as you the viewer's position within the scene and takes the 3D objects and converts them into images composed of pixels

Elements within the process of going from 3D objects to pixels

1 Modeling (3D objects)
2 Model View Transform (Camera)
3 Projection (Process from 3D to 2D)
4 Rasterization (Transforming into pixels)
5 Lightning
6 Texturing/ appearance

3D World has world coordinates and objects which has their own coordinates

Modeling

Model Space

- The x, y, and z, and the 0, 0, 0 position of the model of each object. That's the space in which the vertices of the objects are expressed in.
- These need to be converted into world space which is different for each object.

World Space

- This is what the transform matrix of that object does, it transforms from the local model space of the polygons into the world space of unity with that matrix transform.

Camera Coordinate System (CCS) / View Space

- Where an object is relative to the camera, how far away it is from the camera, what directions is relative to the camera
- The conversion from world space coordinates to camera space coordinates is still a straightforward matrix, it's the transform matrix of the camera

Q Projection happens to vertices in which coordinate system?

view space

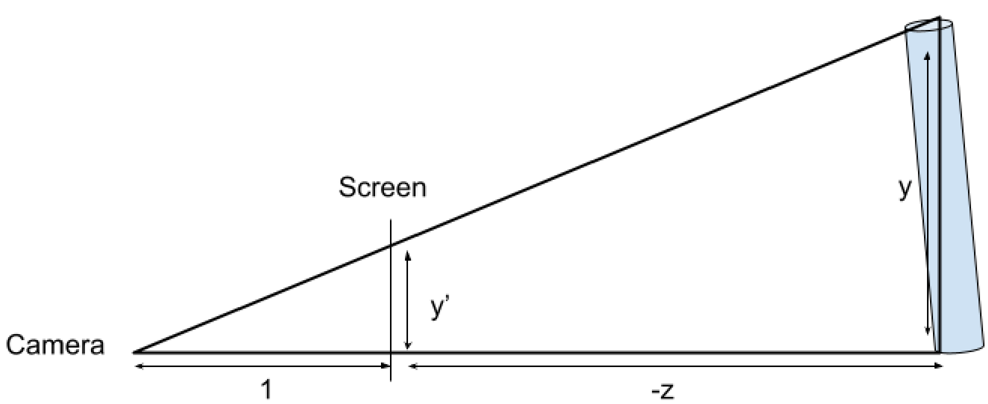
Q What is the correct order of transformations?
Model Space->World Space->View Space

Projection

- This is the process of projection going from 3D to 2D
- **Parallel (Orthographic):**
    - you just get rid of the z's. What that means is that the x and y distances stay the same, and you get quite a flat projection like this.
    - If you want to do a technical drawing where you're preserving distances so you can make measurements of it, you need to do a parallel projection.

$$\overset{4x4}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}} \times \overset{4x1}{\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}} = \overset{4x1}{\begin{pmatrix} x' \\ y' \\ 0 \\ 1 \end{pmatrix}}$$

- **Perspective:**
    - The further away an object is, the smaller it looks on the screen, just as it does in real life
    - If you want something that looks real, use perspective.



Find Transformation value by first using the tangents

$$\frac{y'}{1} = \frac{y}{1-z} \qquad\qquad y' = \frac{y}{1-z}$$

Then use this method to get the value with matrixes

$$[x\,y\,z\,1]$$

$$[x\,y\,z\,w] = [\tfrac{x}{w}\ \tfrac{y}{w}\ \tfrac{z}{w}]$$

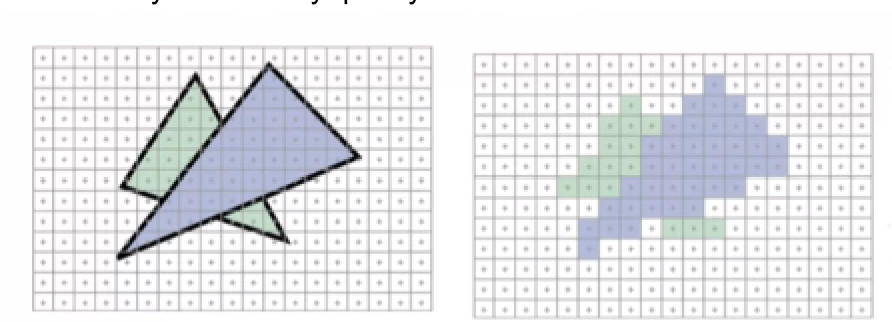
$$[x\ \ y\ \ 0\ (1 - z)] = [\tfrac{x}{1-z}\ \tfrac{y}{1-z}\ 0]$$

Final Matrix:

$$\overset{\text{4x4}}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix}} \times \overset{\text{4x1}}{\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}} = \overset{\text{4x1}}{\begin{pmatrix} x \\ y \\ 0 \\ 1-z \end{pmatrix}} = \overset{\text{3x1}}{\begin{pmatrix} x/(1-z) \\ y/(1-z) \\ 0 \end{pmatrix}}$$

Rasterization

- The process of turning 2D objects into pixels
- After rasterization, we still need to calculate lighting and texture before displaying them.
- Primitives are “continuous” geometry objects, screen is discrete(pixels)
- Rasterization computes a discrete approximation in terms of pixels
- Have to find a way to do it very quickly

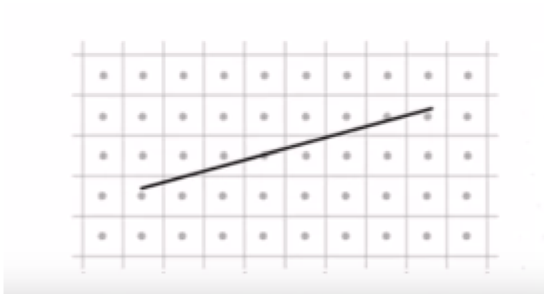


Line Rasterization

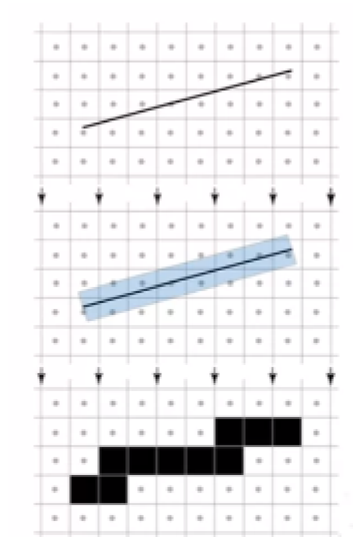
- **Approximate a line with a collection of pixels**

- **Desirable properties:**
  - Uniform thickness
  - Continuous appearance (no holes)
  - Efficiency
  - Simplicity (for hardware implementation)

$y = mx + b$
$p_0(x_0, y_0)$
$p_1(x_1, y_1)$

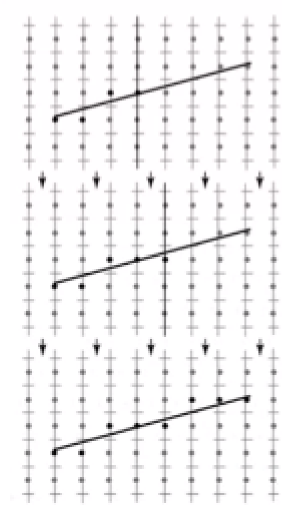


Point-sampled line rasterization



- How do I turn this line equation to pixels?
- I can assume that the line is actually a rectangle.
- For the center of each pixel, we can test if it is inside this rectangle.
- If it is, then we turn it black, otherwise, we leave it white.
- Not very accurate and efficient
- **The line doesn't have a constant thickness**

Midpoint line rasterization

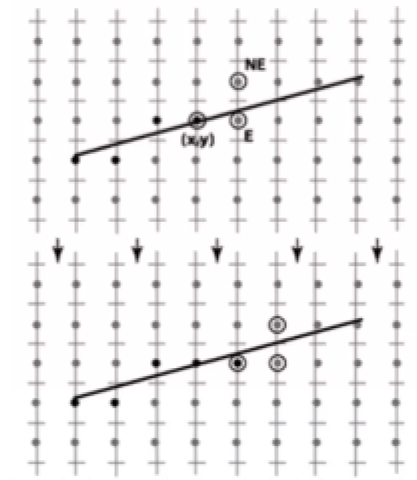


- For each column only turn on closest pixel
- Simple algorithm
  - given line equation
  - evaluate equation for each column between endpoints
- **The line has a constant thickness**

```
For x = round(x0) to round(x1)
{
      y = m*x + b
      write (x, round(y))
}
```

- Evaluating is slow! We assume that m in [0,1)
- At each pixel (Xp, Yp) only two options for next step:
  - Equal: **E**(Xp + 1, Yp)
  - Not Equal: **NE**(Xp + 1, Yp+1)

Bresenham's line rasterization



- All we need to decide for the next step is do we have the same y or do we move up by one
- We can calculate the distance between the real value of the next Y and the current Y. Because we know the line is between 0 to 45 degrees from the x axis, we know that this distance should be something between 0 and 1
- $d = (x_p + 1)\, m\ + b - y_p$
    - **if** $d > 0.5\ then\ NE$
    - **else** E
- We still need to calculate d with the original line equation. But actually, we don't.

- We need to calculate d once in the beginning, and then we update it for each x. If we're making an equal step, we update d to d plus m, and if we're making a not equal step, we'll increase y by one, then we update d to d plus m minus 1.
- Can evaluate **d** using incremental differences:
    - E: d = d+m
    -
    - Ne: d= d+m-1

```
x = round(x0)
y = round (m*x + b)
d = m*(x + 1) + b - y
while x < round(x1)
{
    write (x , y, 1)
    x += 1
    d += m
    if d > 0.5
    {
        y +=1
        d -=1
    }
}
```