# Introducing Assignment 4: Rasterizer
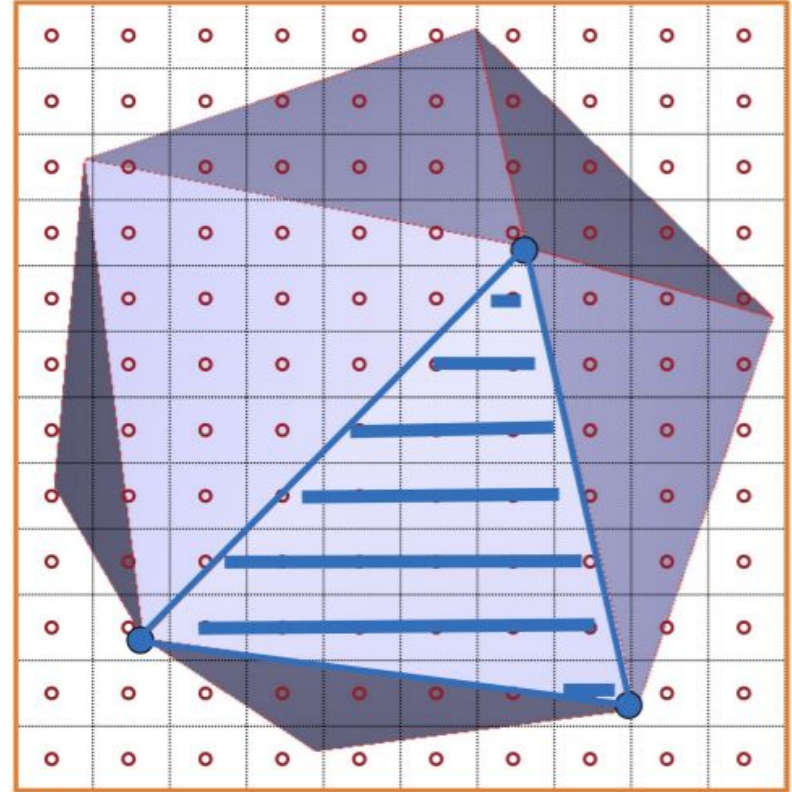
COS 426: Computer Graphics (Spring 2022)

Vivien Nguyen

# Agenda

- Overview of Rasterizer
- Rasterization Pipeline
  - Transformation Pipeline
  - Triangle Pipeline
  - Pixel Shading (Coloring)
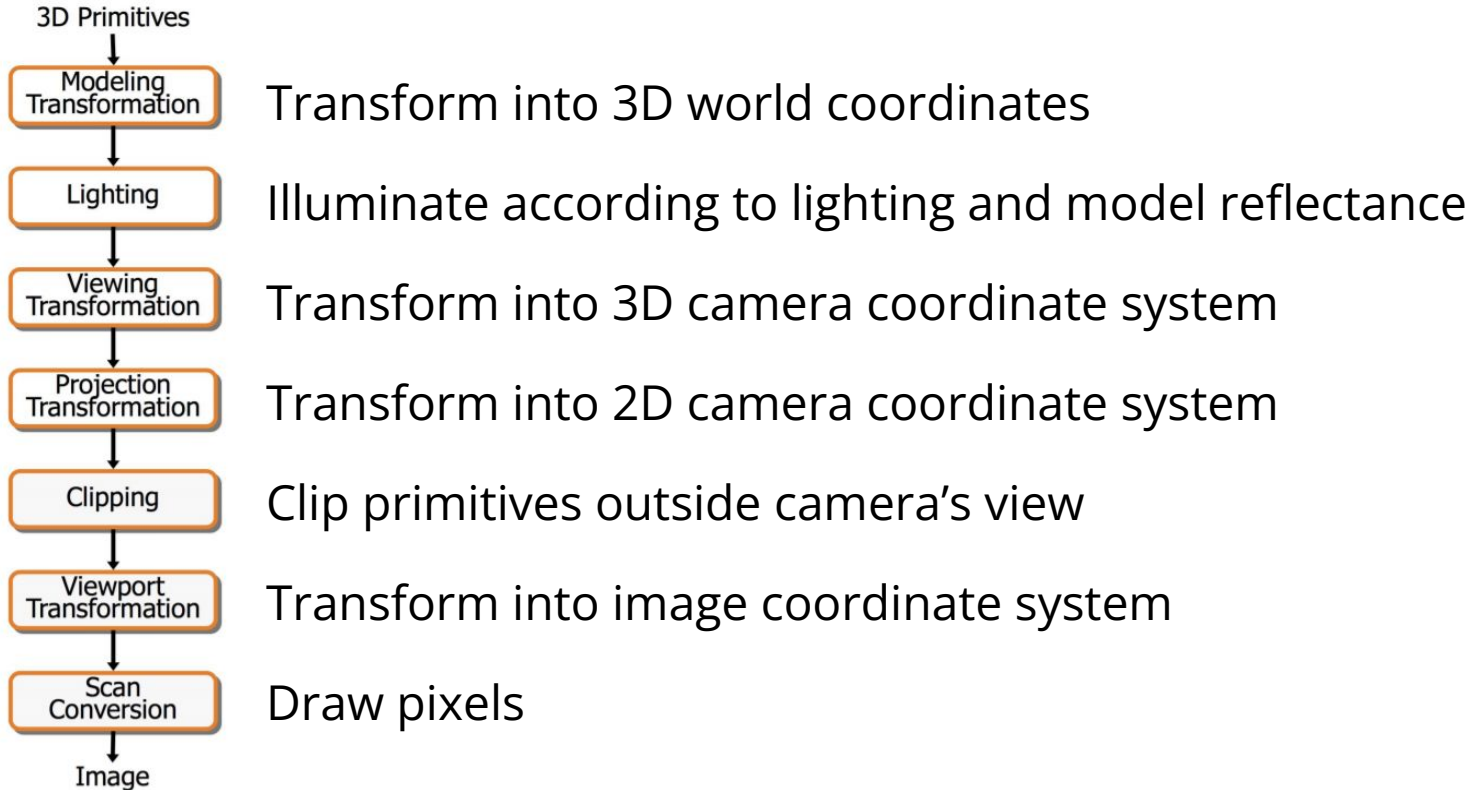  - Texture Mapping

# What is Rasterization?

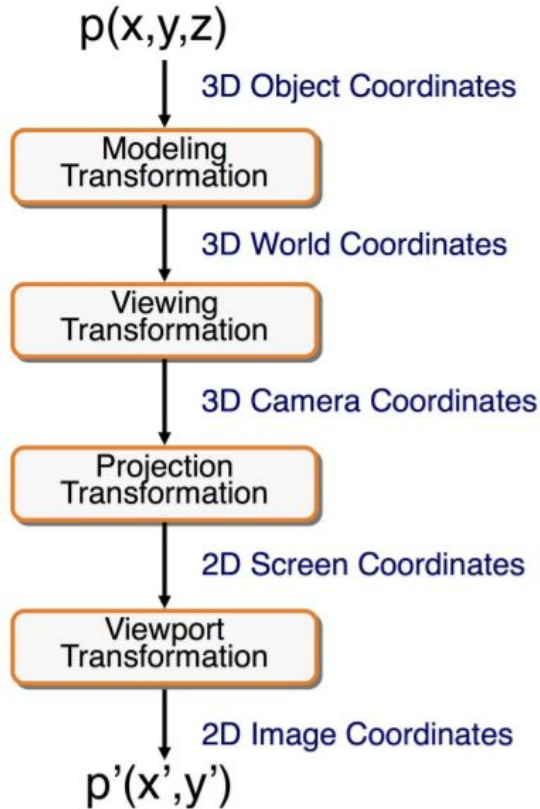- Renders 3D primitives to a 2D image using projection

# Rasterization vs Ray Tracing

- Pros:
  - Less computationally expensive
    - "Shoot rays from screen to objects" vs "Project objects to screen"
  - Takes advantage of spatial coherence of 3D objects
    - "Since this pixel is determined by a point on this triangle, then the neighboring pixels are likely determined by the same tri."
- Con: Less realistic light behavior than ray tracing
- Useful for live rendering
  - Video games, Assignment 2(!)

# Rasterization Pipeline

**3D Primitives**

| Stage | Description |
|-------|-------------|
| Modeling Transformation | Transform into 3D world coordinates |
| Lighting | Illuminate according to lighting and model reflectance |
| Viewing Transformation | Transform into 3D camera coordinate system |
| Projection Transformation | Transform into 2D camera coordinate system |
| Clipping | Clip primitives outside camera's view |
| Viewport Transformation | Transform into image coordinate system |
| Scan Conversion | Draw pixels |

**Image**

# Transformation Pipeline



p(x,y,z)

3D Object Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Projection Transformation

2D Screen Coordinates

Viewport Transformation

2D Image Coordinates

p'(x',y')

In A4:
All meshes are made of triangular faces

Your function transforms a triangle with 3D world coordinates to a projected triangle with 2D image coordinates

# Viewing Transformation

- Going from world coordinates to camera coordinates
- The "pose" of a camera is written as [R|t], a 4x4 matrix
  - R is a 3x3 rotation matrix, t is a 3x1 translation term
  - The last row exists because we are using homogeneous coords.
- [R|t] transforms a point represented in camera coordinates to world coordinates
- So to do the opposite, we apply the inverse of [R|t]

# Homogeneous Coordinates

- It has a fourth dimension, but think of it as another representation of 3D coordinates.
- To transform a 4D homogeneous coord to 3D coord:
  - $(x, y, z, w) \rightarrow (x/w, y/w, z/w)$
- A 3D coordinate $(x,y,z)$ is equivalent to $(x, y, z, 1)$ in 4D homogeneous coordinates.
- Important because the projection matrices we provide are in 4D homogeneous coordinates
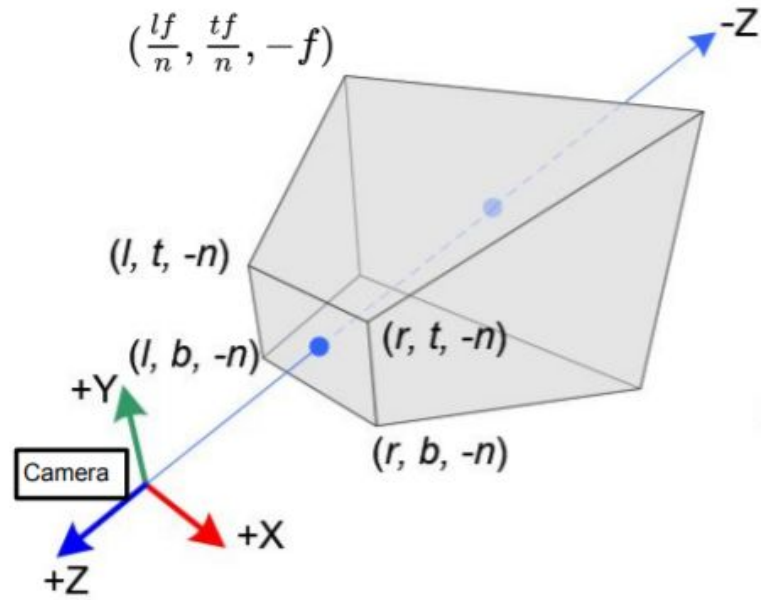
# Perspective Projection Transformation

- Going from 3D camera coordinates to 2D screen coordinates
  - More specifically, we want to convert to Normalized Device Coordinates (NDC)
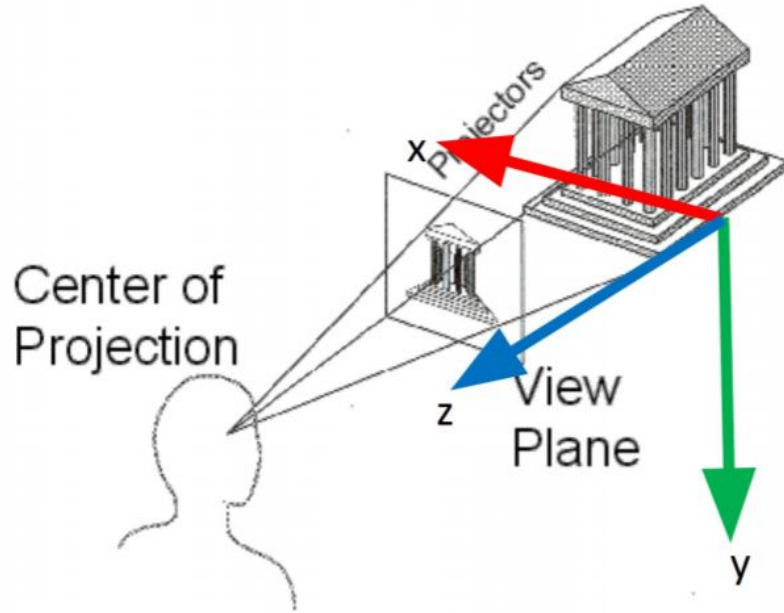
# View Volumes

- Camera Coordinates
  - A truncated pyramid frustum view, bounded by [l, r] in x, [b, t] in y, and [-n, -f] in z
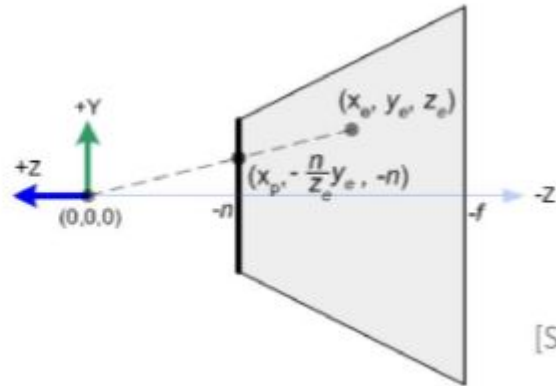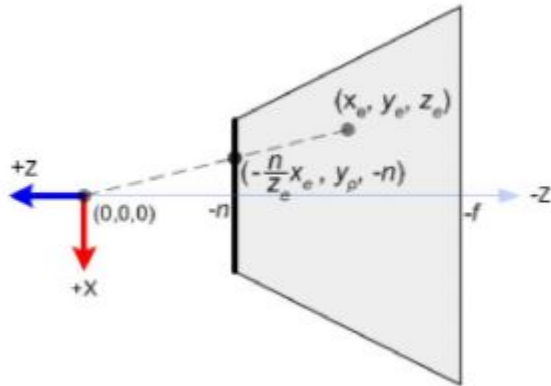  - Positive z axis is going towards camera

$(\frac{lf}{n}, \frac{tf}{n}, -f)$

-Z

$(l, t, -n)$

$(l, b, -n)$

$(r, t, -n)$

$(r, b, -n)$

+Y

Camera

+Z

+X

# View Volumes

- Normalized Device Coordinates
  - The "canonical" view volume bounded by a cube
  - Maps [l, r] -> [-1,1] in x,
  - [b, t] -> [-1, 1] in y,
  - [-n, -f] -> [-1, 1] in z



(-1, 1, 1)

(1, 1, 1)

+Y

+Z

+X

(1, -1, 1)

(-1, -1, -1)

(1, -1, -1)
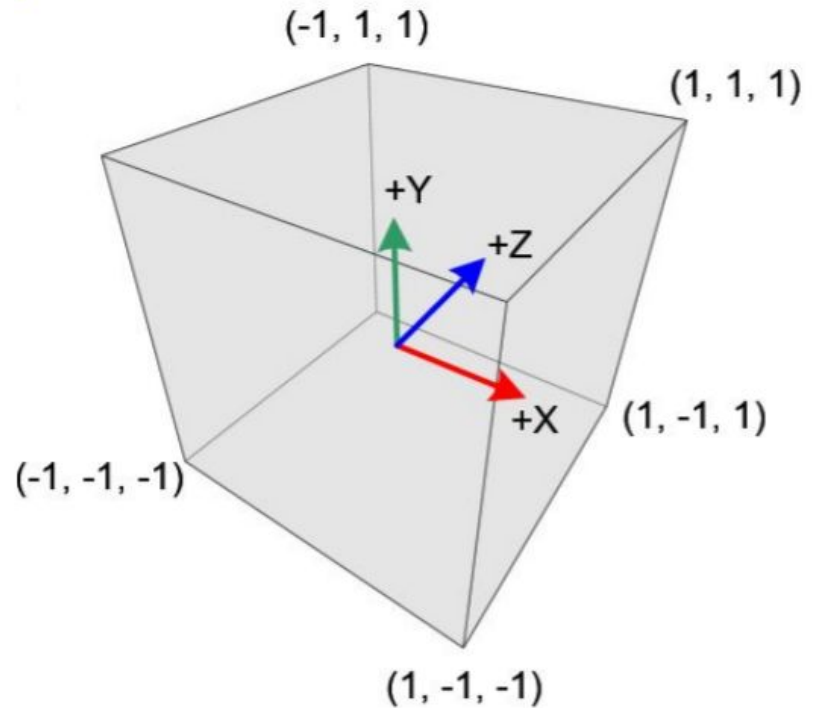
- Think about one dimension at a time
  - How can we scale it so the edges are going to be bound by [-1,1]?



[Song Ho Ahn]

# Perspective Projection Matrix

- The matrix that transforms from frustum view to canonical view

$$\text{projMat} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

  - Remember to divide your result by w to get the 3D equivalent
  - If your resulting z is not within the bounds of the canonical view, skip the triangle because it shouldn't be seen

# Viewport Transformation

- Going from Normalized Device Coordinates to image coordinates
  - X: [-1,1] -> [0, image width]
  - Y: [-1,1] -> [0, image height]
- Should we save Z?
  - Yes, need it for Z-buffering
    - Determining which object is closer to camera if they take up the same pixel, the closer thing gets rendered

# Implementation Hints for Transform

- The world to camera transformation and perspective projection matrices are already precomputed for you!
  - viewMat = projMat * $[R|t]^{-1}$
- You have to apply this matrix to the triangle in 3D space to project it to Normalized Device Coordinates, then scale it to image coordinates

# Pipeline of Rendering a Triangle

- Now we know how to transform, how do we render it?
- **Transform** a 3D triangle in world space to 2D triangle in image space
- Compute bounding box of the triangle
- For each pixel (x, y) in the bounding box:
  - Check if it's in the triangle w/ barycentric coordinates. If not, skip this pixel
  - Use barycentric coords to interpolate the z value for this pixel
  - If this z value is bigger than the value in z buffer for this pixel, skip this pixel
  - **Render the pixel**, and save this z value to z buffer for this pixel

# Barycentric Coordinates

- A point in a triangle can be represented as a convex combination of the three vertices
  - If any of the weights ($t_i$) are negative, then point is not in triangle
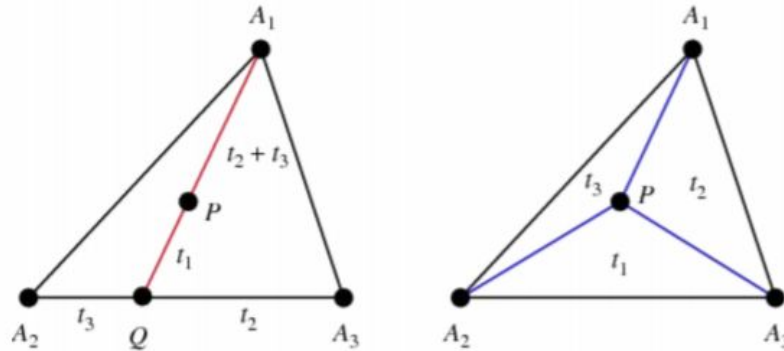


Efficient 2D algorithm on slides 30-33 at
https://www.cs.drexel.edu/~david/Classes/CS430/Lectures/L-10_NURBSDrawing.pdf

# Pipeline of Rendering a Pixel

- Now we know which pixel to render, how do we color it?
- For a pixel to render,
  - Compute the normal and position of this pixel in **world coords \*\***
    - Use barycentric coords to interpolate
  - Find view position of the camera in **world coords**
  - Find light source position(s) in **world coords**
  - Get the material of this pixel (getPhongMaterial) \*\*
  - Apply the given shader (either flat, Gourand, or Phong) using the above variables to color the pixel
    - Very similar to A3!

# Overview of Shaders

- Flat
  - Color of pixel is determined by face normal and centroid
    - Calculate color once per triangle
- Gouraud
  - Color of pixel is an interpolation of the colors at the vertices
    - Calculate color x3 per triangle
- Phong
  - Color of pixel determined by its normal found by interpolation
    - Calculate color once **per pixel** in triangle

# Texture Mapping

- If a mesh has a texture map, you have to define the uv coordinate for getPhongMaterial
- UV Coordinates
  - A vertex of the triangle with uv coordinate (u,v) in the texture map will have that color of texture map at (u,v)
  - Make sure uvs[] is defined for the triangle because not all meshes have texture maps

# Texture Mapping

- Normal Mapping
  - Adds additional detail to texture map
  - Stores normal vector information in an image I
  - The image uses same UV coordinates
  - For a vertex with UV coordinate (u,v)
    - Get RGB at I(u,v)
    - Compute normal XYZ = normalize(2*RGB - 1)
  - Use this new normal when calculating color with Phong Reflection Model