Announcements

- **Assignment 1**
  - programming (due Friday)
  - submission directories are fixed use (submit -N A1b cscd18f08 a1_solution.tgz)
  - theory will be returned (Wednesday)

- **Midterm**
  - Will cover all of the materials so far including today's lecture
    - Lecture notes, lecture slides, readings, assignment are all fair game
  - Practice midterms are on-line (no solutions will be given)

- **Tutorial this week**
  - Life of the polygon
  - A1 theory questions

- **Office Hours**
  - I will have office hours today 1-2 pm
  - Alex will have office hours later in the week
  - I will also have office hours on Tuesday 4-5pm
Last week’s review …

- **Cameras (theory)**
  - Pinhole Camera
  - Thin Lens model
  - Virtual pinhole camera
  - Perspective and orthographic projections

- **Cameras (practice)**
  - Location of camera in space
  - Transformation of geometry from camera to world coordinate frame and (vice versa)
  - **Homogeneous Perspective Projection** (how do we represent perspective using a single 4x4 matrix)
  - Homogeneous Prospective Projection with **Pseudodepth**
Projecting Triangle

- Lets review steps in the rendering hierarchy
  - Triangle is given in the object-based coordinate frame as three vertices
Projecting Triangle

- Lets review steps in the rendering hierarchy
  - Triangle is given in the object-based coordinate frame as three vertices
  - Transform to world coordinated $\overrightarrow{p_i}^w = M_{ow}\overrightarrow{p_i}^o$
Projecting Triangle

- Lets review steps in the rendering hierarchy
  - Triangle is given in the object-based coordinate frame as three vertices
  - Transform to world coordinated \( \mathbf{p}_i^w = \mathbf{M}_{ow}\mathbf{p}_i^o \)
  - Transform from world to camera coordinates \( \mathbf{p}_i^c = \mathbf{M}_{wc}\mathbf{p}_i^w \)
Projecting Triangle

- Lets review steps in the rendering hierarchy
  - Triangle is given in the object-based coordinate frame as three vertices
  - Transform to world coordinated: $\overrightarrow{p_i^w} = M_{ow}\overrightarrow{p_i^o}$
  - Transform from world to camera coordinates: $\overrightarrow{p_i^c} = M_{wc}\overrightarrow{p_i^w}$
  - Apply homogeneous perspective: $\overrightarrow{p_i^*} = M_p\overrightarrow{p_i^c}$
  - Divide by last component
Projecting Triangle

- Lets review steps in the rendering hierarchy
  - Triangle is given in the object-based coordinate frame as three vertices
  - Transform to world coordinated \( \vec{p}_i^w = M_{ow}\vec{p}_i^o \)
  - Transform from world to camera coordinates \( \vec{p}_i^c = M_{wc}\vec{p}_i^w \)
  - Apply homogeneous perspective \( \vec{p}_i^* = M_p\vec{p}_i^c \)
  - Divide by last component
Visibility
Clipping

- **Idea:** Remove points and parts of objects outside view volume
- Sounds simple, but consider if we have an object on a boundary
Consider what we can actually see
Side note: Field of View

\[
\frac{1}{2} \tan(\alpha) = \frac{1}{2}(T - B) \quad f
\]
**View Volume**

- What does homogeneous perspective projection do to our view volume?

\[ M_p = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{2F}{f - F} & -\frac{1}{f} \left( \frac{f + F}{f - F} \right) \\
0 & 0 & \frac{1}{f} & 0 \\
\end{bmatrix} \]

\[ \text{parallelepiped} \]
Canonical View Volume

Can we alter homogeneous perspective projection to help us clip?

\[ M_p = \begin{bmatrix}
\frac{2}{R-L} & 0 & \frac{R+L}{T-B} & 0 \\
0 & 2 & \frac{T-B}{2F} & \frac{-1}{f} \left( \frac{f+F}{f-F} \right) \\
0 & 0 & \frac{f-F}{1/f} & 0 \\
0 & 0 & \frac{f-F}{1/f} & 0
\end{bmatrix} \]
**Back-face Removal**

- **Idea:** Remove surface patches that point away from the camera (like backside of the object as it viewed from the front)

- **Consider a cube**

  - 3 Back Faces
  - 4 Back Faces
  - 5 Back Faces

  We only need to render at most half of the sides depending on the view
Back-face Removal

- How do we know if the patch (triangle) points away from the camera?

Consider a normal of the patch (triangle)

- If \((\vec{p} - \vec{e}) \cdot \vec{n} > 0\) then triangle is part of the back-face and needs to be removed
- If \((\vec{p} - \vec{e}) \cdot \vec{n} < 0\) then triangle may be visible
Back-face Removal

- Does it matter which point we consider on the patch?
  - Not if this is a **planar** patch

Consider a normal of the patch (triangle)

- If $(\overrightarrow{p} - \overrightarrow{e}) \cdot \hat{n} > 0$ then triangle is part of the back-face and needs to be removed
- If $(\overrightarrow{p} - \overrightarrow{e}) \cdot \hat{n} < 0$ then triangle **may** be visible
Back-face Removal

- Does it matter which point we consider on the patch?
  - Not if this is a **planar** patch
- How do we compute $\vec{n}$
  - If $\vec{p}_1, \vec{p}_2, \vec{p}_3$ are patch vertices in CCW order
    - If $(\vec{p} - \vec{e}) \cdot \vec{n} > 0$ then triangle is part of the back-face and needs to be removed
    - If $(\vec{p} - \vec{e}) \cdot \vec{n} < 0$ then triangle **may** be visible
Back-face Removal

- Does it matter which point we consider on the patch?
  - Not if this is a **planar** patch
- How do we compute \( \mathbf{n} = \frac{(\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)}{\| (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \|} \)

- If \( (\mathbf{p} - \mathbf{e}) \cdot \mathbf{n} > 0 \) then triangle is part of the back-face and needs to be removed
- If \( (\mathbf{p} - \mathbf{e}) \cdot \mathbf{n} < 0 \) then triangle **may** be visible
Z-Buffer (a.k.a. Depth Buffer)

- We have a **frame-buffer** (this is where an image that we see on the screen is stored)
- We also have a **z-buffer** that keeps track of the $z^*$ coordinate for every pixel in the frame-buffer

- To draw point in the world with color $c$ that projects to $(x^*, y^* z^*)$ we can execute the following algorithm

```
if $z^* < z$-buffer$(x^*, y^*)$ then
    frame-buffer$(x^*, y^*) = c$
    $z$-buffer$(x^*, y^*) = z^*$
end
```
Z-Buffer (a.k.a Depth Buffer)

- We need to initialize the z-buffer with some value. What is the good value to initialize with?
  - If we are using canonical view volume then 1 would work

- To draw point in the world with color \( c \) that projects to \((x^*, y^* z^*)\) we can execute the following algorithm

\[
\begin{align*}
\text{if } z^* &< \text{z-buffer}(x^*, y^*) \text{ then} \\
&\text{frame-buffer}(x^*, y^*) = c \\
&\text{z-buffer}(x^*, y^*) = z^* \\
\text{end}
\end{align*}
\]
Z-Buffer (a.k.a a Depth Buffer)

- **Advantages of Z-buffering**
  - Simple and accurate
  - Independent of the order the polygons are drawn

- **Disadvantages of Z-buffering**
  - Memory for a Z-buffer (small consideration)
  - Wasted computation in drawing distant points first (this potentially can be a large drawback)
Z-Buffer (a.k.a Depth Buffer)

- We represent a patch using vertices
- How do we get a pseudodeph and proper rendering everywhere else?
Z-Buffer (a.k.a Depth Buffer)

- We represent a patch using vertices
- How do we get a pseudodeph and proper rendering everywhere else?

Linearly interpolate $z^*$ along a scan line
**Painter’s Algorithm**

- **Idea:** Order the patches and draw them in the order of depth (with most distant patches first)
  - This is an alternative to Z-buffering
Painter’s Algorithm

- How do we deal with intersecting patches?
  - Break patches into smaller patches

\[(x_1^*, y_1^*, z_1^*)\]

\[(x_2^*, y_2^*, z_2^*)\quad(x_3^*, y_3^*, z_3^*)\]
BSP Trees

- **Binary space partition tree** (BSP tree) is an algorithm for making back-to-front ordering of polygons efficient and to break polygons to avoid intersections.
BSP Tree

- If $\vec{e}$ and $T_2$ are on the same side of $T_1$ (left) then draw $T_1$ first then $T_2$
- If $\vec{e}$ and $T_2$ are on different sides of $T_1$ (right) then draw $T_2$ first then $T_1$
- How do we know if points are on the same side?

\[
f_i(\vec{x}) = (\vec{x} - \vec{p}_i) \cdot \vec{n}_i
\]

- $f_i(\vec{x}) = 0$ on the plane
- $f_i(\vec{x}) > 0$ "outside"
- $f_i(\vec{x}) < 0$ "inside"

outside-facing normals
BSP Tree Example

- Let’s try building a BSP tree for this scene

The tree will be the same regardless of the camera placement
Let’s try building a BSP tree for this scene.
Let’s try building a BSP tree for this scene.
Let’s try building a BSP tree for this scene.
Let’s try building a BSP tree for this scene.
Let’s try building a BSP tree for this scene.
BSP Tree Traversal

- Tree traversal algorithm

```plaintext
if eye in the outside half-space of the root
    Draw faces on inside sub-tree of the root
    Draw the root
    Draw faces is the outside of sub-tree of the root
else
    Draw faces is the outside of sub-tree of the root
    Draw the root
    Draw faces on inside sub-tree of the root
end
```

- Easy to modify to do back-face removal
BSP Tree

- **Advantages**
  - Can easily discard portions of the scene behind the camera
  - Artifacts of z-buffer quantization are not seen
  - Tree construction fixed for the static scenes

- **Disadvantages**
  - How can we handle dynamic scenes?

This is what is typically done in games, because it’s fast