Drawing in 3D (again)
(this time with depth)

CS559 – Fall 2016
Lecture 8
October 3, 2016
What does it take to do this?

1. Put a 3D primitive in the World
2. Figure out what color it should be
3. Position relative to the Eye
4. Get rid of stuff behind you/offscreen
5. Figure out where it goes on screen
6. Figure out if something else blocks it
7. Draw the 2D primitive
1. Put a 3D primitive in the World
   **Modeling**
2. Figure out what color it should be
   **Shading**
3. Position relative to the Eye
   **Viewing** / Camera Transformation
4. Get rid of stuff behind you/offscreen
   **Clipping**
5. Figure out where it goes on screen
   **Projection** (sometimes called Viewing)
6. Figure out if something else blocks it
   **Visibility** / Occlusion
7. Draw the 2D primitive
   **Rasterization** (convert to Pixels)
Viewing / Projection

How to get from the object to the screen?

A **transformation** between coord systems

Once we get to the screen, then draw a 2D primitive. Like a painter.
In case it wasn’t obvious. . .

We transform **points**

If you want to transform a line/triangle
   Transform its points
   Re-assemble it after transforming
   (e.g. draw the 2D primitive)
3D to 2D

Do we lose a dimension?

No – we actually need to keep it
Yes – but we’ll just ignore Z

The screen as a fishtank
Canonical View Volume
Normalized Device Coordinates

-1 to 1 (zero centered; note TWGL variations)

XY is screen (y-up)

Z is towards viewer (right handed)

   Negative Z is into screen
   (so some prefer left-handed)

Viewport transform: NDC -> Pixels
All the coordinate systems

Window (Screen) – in pixels

Normalized Device – [-1 1]

Camera / Eye

World

Object . . .

Local
Transformations between each

Viewport Trans

Projection

Viewing

Modeling

Modeling

Window (Screen) – in pixels

Normalized Device – [-1 1]

Camera / Eye

World

Object . . .

Local
From object to eye: ModelView

Modeling matrix: object to world

Viewing matrix: world to eye / camera
  Rigid Transformation (rotate/translate)

Invert the camera’s model matrix
Build a “LookAt / LookFrom” matrix
How to describe cameras?

Rotate and translate (and scale) the world
The camera is a physical object
(that can be rotated and)
Easier ways to specify cameras
Look from / Look at / vup
Lookfrom / Lookat / Vup

Don’t compute angles!

Eye point = LA
Z axis (camera sight) = LF-LA
X axis (camera right) = Z x Vup
Y axis (camera up) = X x Z
(normalize, and flip directions as needed)
Next Problem: Projection

Convert 3D (eye coordinates) to 2D (screen)
A transformation

Types:
- Orthographic
- Perspective
- some others we won’t talk much about
View Volumes / Transformations

Viewing transformation puts the world into the viewing volume

A box aligned with the screen/image plane
Orthographic Projection

Projection = transformation that reduces dimension

Orthographic = flatten the world onto the film plane
Orthographic Projection

Scale X and Y to fit things on screen

Note: we can look in any direction
    we are already in camera coordinates!
Orthographic Projections

Simple
Preserves Distances

Objects far away same size as close
Looks weird
Orthographic Projections

Note the TWGL Convention! (Documentation is either wrong or misleading)

Depth range [znear,zfar] gets remapped to [0,-1] !!!
Perspective Projection

Farther objects get smaller

Eye (or focal) point
Image plane
View frustum (truncated pyramid)

Two ways to look at it:
- Project world onto image plane
- Transform world into rectangular view volume (that is then orthographically projected)
Perspective Assumptions

There is a single focal point

Simplifying Assumptions: (not required)
Image plane orthogonal to view direction
Image plane centered on view direction
Perspective

Eye point
Film plane
Frustum

Simplification
  Film plane centered with respect to eye
  Sight down Z axis
  • Can transform world to fit
Pinhole Camera
Basic Perspective

Similar Triangles

Warning = using d for focal length (like book)
F will be “far plane”

\[
\frac{y}{z} = \frac{y'}{d}
\]

\[
y' = \frac{d}{z}y
\]
Use Homogeneous coordinates!

Use divide by w to get perspective divide

Issues with simple version:
Font / back of viewing volume
Need to keep some of Z in Z (not flatten)

\[
\begin{bmatrix}
 x' \\
 y' \\
 z' \\
 w'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 \\
 0 & 0 & 1 & 0 \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z \\
 1
\end{bmatrix} =
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix} =
\begin{bmatrix}
 x/z \\
 y/z \\
 z/z = 1
\end{bmatrix}
\]
Simplest Projective Transform

\[
\begin{bmatrix}
  d & 0 & 0 & 0 \\
  0 & d & 0 & 0 \\
  0 & 0 & 0 & 1 \\
  0 & 0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1 \\
\end{bmatrix}
= \begin{bmatrix}
  dx \\
  dy \\
  1 \\
  z \\
\end{bmatrix}
= \begin{bmatrix}
  dx / z \\
  dy / z \\
  1 / z \\
  1 \\
\end{bmatrix}
= \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1 \\
\end{bmatrix}
\]

After the divide by w...
- Note that this is \(dx/z, \ dy/z\) (as we want)
- Note that \(z'\) is \(1/z\) (we can’t keep \(Z\))

Fancier forms scale things correctly
The real perspective matrix

\[ P = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & \frac{n+f}{n} & -f \\
0 & 0 & -n & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix} \]

N = near distance, F = far distance
Z = n put on front plane, z=f put on far plane
Shirley’s Perspective Matrix

After we do the divide, we get an unusual thing for $z$ – it preserves order, keeps n&f

\[
P_x = P \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \frac{n}{z} \\ y \frac{n}{z} \\ n + f - \frac{fn}{z} \\ 1 \end{bmatrix}
\]
The TWGL perspective matrix

\[
\text{perspective(fov, aspect, zNear, zFar)} \rightarrow \{\text{Mat4}\}
\]

\(\text{fov} = \) field of view (specify focal length)
\(\text{aspect ratio} = \) width of image
assuming height is 1
Field of View

\[ \theta \]

\[ d \]
znear and zfar are distances the camera sights down the \(-Z\) axis

```javascript
var pmat = m4.perspective(toRadians(60), 1, 1, 10);
writeMatrix(pmat);

writePoint(m4.transformPoint(pmat, [0, 0, -1]));
  // 0,0,-1 (near)
writePoint(m4.transformPoint(pmat, [0, 0, -10]));
  // 0,0,1 (far)
```

```
<table>
<thead>
<tr>
<th>2.23</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1.22</td>
<td>-2.22</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

[0,0,-1]
[0,0,1]
```
Transformations between each

Viewport Trans
Projection
Viewing
Modeling
Modeling

Window (Screen) – in pixels
Normalized Device – [-1 1]
Camera / Eye
World
Object …
Local
1. Put a 3D primitive in the World
   **Modeling**
2. Figure out what color it should be
   **Shading**
3. Position relative to the Eye
   **Viewing** / Camera Transformation
4. Get rid of stuff behind you/offscreen
   **Clipping**
5. Figure out where it goes on screen
   **Projection** (sometimes called Viewing)
6. Figure out if something else blocks it
   **Visibility** / Occlusion
7. Draw the 2D primitive
   **Rasterization** (convert to Pixels)
Visibility:
What objects do you see?
What objects are offscreen?
  To avoid drawing them
    (generally called clipping)
What objects are blocked?
  Need to make things look solid

Assumes we have “filled” primitives
  Triangles, not lines
Now we’re in Screen Coordinates with depth
Bad ideas...

Last drawn wins
sometimes object in back
what you seen depends on ...

Wireframe (nothing blocks anything)
hard to see what’s going on if complex
How to make objects solid

Physically-Based
Analytic Geometry

Object-space methods (order)
Image-space methods (store per pixel)
Painter’s Algorithm

Order the objects

Draw stuff in back first

Stuff in front blocks stuff in back
Simple version

Pick 1 point for each triangle
Sort by this one point
What about triangles that ... Intersect? Overlap?

Need to divide triangles that intersect (if you want to get it right)

A triangle can be in front of and behind
Downsides of Painters Algorithm

Need to sort
\[ O(n \log n) \]

need all triangles (not immediate)

Dealing with intersections = lots of triangles

Need to resort when the camera moves
Binary Space Partitions

Fancy data structure to help painters algorithm
Stores order from any viewpoint

A plane (one of the triangles) divides other triangles
Things on same side as eye get drawn last

T2 divides into groups
T3 is on same side of eye
Using a BSP tree

Recursively divide up triangles

Traverse entire tree
   Draw farther from eye subtree
   Draw root
   Draw closer to eye subtree

Always $O(n)$ to traverse
   (since we explore all nodes)
No need to worry about it being balanced
Building a BSP tree

Each triangle must divide other triangles
Cut triangles if need be

Goal in building tree: minimize cuts
Painters Problem 2: Overdraw

All triangles get drawn

Even if something else will cover it

Depth Complexity = # of things at each pixel

Inefficient, uses lots of memory bandwidth
Z-Buffer

An image space approach

Hardware visibility solution
Throw memory at the problem

Every pixel stores color and depth
Z-buffer algorithm

Clear all pixels to “farthestest value” (-inf)

for each triangle
    for each pixel
        if new Z > old Z:  // in front
            write new color and Z
Simple

The only change to triangle drawing:

test Z before writing pixels

\texttt{writeColor}@pixel) becomes:
 \texttt{readZ}@pixel)

\texttt{test}

\texttt{writeZandColor}@pixel)
Notice...

Order of triangles *usually* doesn’t matter

Except...

   If the Z is equal, we have a tie
   We can decide if first or last wins
   Either way, order matters

Z-Fighting
Z-Fighting

Z Equal? Order matters

Z Really close?
    random numerical errors cause flips
Z-Resolution

Remember – we don’t have real Z
we have 1/Z (bunches resolution)

Old days: integer Z-buffer was a problem
Nowadays: floating point Z-buffers
Z-resolution less of an issue
Keep near and far close
Transparent Objects

Draw object in back
Draw transparent object in front

But...

Draw transparent object in front
Draw object in back (Z-buffer prevents)
Overdraw

Still drawing all objects – even unseen

Can save writes if front objects first

Early z-test...
  Avoid computing pixel color if it will fail z-test
Using the Z buffer

Give polygons in any order (except…)
Use a Z-Buffer to store depth at each pixel

Things that can go wrong:
Near and far planes matter
Culling tricks can be problematic
You may need to turn the Z-buffer on
Don’t forget to clear the Z-Buffer!