Drawing in 3D (again)  
(this time with depth)  
CS559 - Fall 2015  
Lecture 7  
September 24, 2015
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)
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)

---

Did some, will do more

A little for P4

Last time (**review**)

Not much to say

Last time (**review**)

(FCG 8)
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)
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
\[ e = M_e^{-1} M_p^{-1} M_t M_b p \]
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
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)

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

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

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

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

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 $n\&f$

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

\[
\text{perspective}(\text{fov}, \text{aspect}, z\text{Near}, z\text{Far}) \rightarrow \{\text{Mat4}\}
\]

\(\text{fov} = \text{field of view (specify focal length)}\)

\(\text{aspect ratio (width of image)}\)

assuming height is 1
Field of View

d
\[ \Theta \]
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)
```

It's making things Left Handed! (?)
Transformations between each

- Viewport Trans
- Projection
- Viewing
- Modeling
- Local
- Object . . .
- World
- Camera / Eye
- Normalized Device – [-1 1]
- Window (Screen) – in pixels
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
(this is OK for P4)
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 “farthest 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}@\texttt{pixel} becomes:
\texttt{readZ}@\texttt{pixel}
\texttt{test}
\texttt{writeZandColor}@\texttt{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!
Culling

 Quickly determine that things cannot be seen – and avoid drawing them

 Must be faster to rule things out than to draw them
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)
A Quick Word on Shading (for P4)

Color of triangle depends…

Color per triangle (OK for P4)
Color per vertex
Color per pixel
Lighting basics

To simulate light, we need to know where the triangle is in the world

Global Effects (other objects)
reflections, shadows, …

Local Effects (how the light bounces off)
shininess, facing the light, …
Local Geometry

Normal Vector – sticks “out” of the triangle

\[ N = \overrightarrow{AB} \times \overrightarrow{AC} \]
Transforming Normal Vectors

Transform triangle, re-compute the normal or...

Normal is transformed by the inverse transpose of the transform

If the triangle is transformed by $M$
The normal is transformed by $(M^{-1})^T$
Inverse Transpose?

Yes – ask Prof. Sifakis for the proof.
   (the book just asserts it as fact)

For a rotation, the inverse is the transpose
\[ M = (M^{-1})^T \]

But only for rotations …
What can I use a normal vector for?

Simplest lighting: Diffuse Shading

If surface is pointing towards light, it gets more light

\[
\text{brightness} \approx N \cdot L
\]

\(N = \text{unit normal vector}\)
\(L = \text{unit light direction vector}\)
We’ll look at this more in the future

Consider a fixed sized object:

\[
\text{amount of light that hits is } \approx \cos \theta \text{ where } \theta = \angle \text{ between light and normal}
\]

\[
D \propto n^i
\]
Simple things for P4

High noon...
\[ C' = \left( \frac{1}{2} + \frac{1}{2} N \cdot [0,1,0] \right) C \]

Top and bottom...
\[ C' = \left( \frac{1}{2} + \frac{1}{2} \text{abs}(N \cdot [0,1,0]) \right) C \]

Make sure N is a unit vector!
Program 4

Just like P3 (transform points) but...

1. Draw Triangles (solids)
2. Compute Normals (and shade)
3. Store triangles in a list and sort
   Painter’s Algorithm Visibility
What coordinate system to compute lighting in?

- Window (Screen)
  - Normals lost

- Normalized Device – [-1 1]
  - Projection loses normals

- Camera / Eye
  - Camera space is OK

- World
  - World space is good

- Object . . .

- Lights attached to objects?
  - 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)
Rasterization

Figure out which pixels a primitive “covers”

Turns primitives into pixels
Rasterization

Let the low-level library take care of it
Let the hardware take care of it

Writing it in software is different than hardware
Writing it today (with cheap floating point) is different than a few years ago
How does the hardware do it? (or did it last I learned about it)

Find a box around the triangle
For each pixel in the box
  compute the barycentric coordinates
  check if they are inside the triangle
Do pixels in parallel (in hardware)
  otherwise, really wasteful
Barycentric coordinates are useful
Barycentric Coordinates

Any point in the plane is a convex combination of the vertices of the triangle

\[ P = \alpha A + \beta B + \gamma C \]

\[ \alpha + \beta + \gamma = 1 \]

Inside triangle
\[ 0 \leq \alpha, \beta, \gamma \leq 1 \]
Where are we going next...

We’ve made a graphics pipeline

Triangles travel through steps... get turned into shaded pixels

How do we use the hardware to make this go fast...