Drawing in 3D
(viewing, projection, and the rest of the pipeline)

CS559 – Spring 2017
Lecture 6
February 2, 2017
The first 4 Key Ideas...

1. Work in convenient **coordinate systems**. Use **transformations** to get from where you want to be to where you need to be. Hierarchical modeling lets us build things out of pieces.

2. Use **homogeneous coordinates** and transformations to make common operations easy. Translation, projections, coordinate system shift all become simple matrix multiplies.

3. Create **viewing transformations** with **projection**. The geometry of imaging (pinhole camera model) leads to linear transformations in homogeneous coordinates.

4. Implement primitive-based rendering (interactive graphics) with a **pipeline**. The abstractions map nicely onto hardware, and let you do things like visibility computations easily. Be aware that there are other paradigms for drawing.
The first 4 Key Ideas...

1. Work in convenient **coordinate systems**. Use **transformations** to get from where you want to be to where you need to be. Hierarchical modeling lets us build things out of pieces.

2. Use **homogeneous coordinates** and transformations to make common operations easy. Translation, projections, coordinate system shift all become simple matrix multiplies.

3. Create **viewing transformations** with **projection**. The geometry of imaging (pinhole camera model) leads to linear transformations in homogeneous coordinates.

4. Implement **primitive-based rendering (interactive graphics)** with a **pipeline**. The abstractions map nicely onto hardware, and let you do things like visibility computations easily. Be aware that there are other paradigms for drawing.

Last 2 Weeks

This Week (and next)
ReCap: Basic Idea

To draw we need a coordinate system

Transformations can
  Move between coordinate systems
  Move objects around

Useful transformations as matrices
ReCap: Math

Points, Vectors, Matrices
Transformations as linear operators
  Matrices (homogeneous coordinates)

3D Coordinate Systems
  cross product, right hand rule
ReCap: Code

Matrix stack
• Save
• Transform (concat)
• Draw (current trans)
• Restore

```javascript
this.context.save();
this.context.translate(50,50);
this.context.rotate(this.frontPropAngle);
this.drawProp();
this.context.restore();
```
What about 3D?
How to draw in 3D?

We are making a **2D** picture!

Do it …

Like nature does
Like artists do
Do it like Da Vinci!
Primitive-based Rendering

Draw 2D Objects in the image
  Paint strokes on canvas
  Lines / Triangles on screen

Map (Transform)
3D primitives (world) to 2D primitives (screen)
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**
In terms of the readings...

A lot of the pieces are there
Not necessarily in any particular order

Good for the details
    Today we’ll try for the big picture
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, lots later
Today (FCG 7)
Not much to say (FCG 8)
Today (FCG 7)
This week (FCG 8)
Not much to say (FCG 8)
In case you’re wondering... (We’ll come back to this. It’s a detail)

For the kinds of projection we will use...

3D Points map to 2D Points
3D Lines map to 2D Lines
3D Triangles map to 2D Triangles

 Doesn’t work for curves (even ellipses)
Viewing

How to get from the object to the screen?

A transformation between coord systems

A little weird...
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
Let’s build a viewing transform

A toy example (with code)
My Neighbor ...
A (simple) bird

```javascript
function birdGeometry(ct) {
    "use strict";
    ct.fillStyle = "#88C";
    ct.beginPath();
    ct.arc(0,0,20, 0, 2 * Math.PI, false);
    ct.arc(15,15,10, 0, 2 * Math.PI, false);
    ct.fill();
    ct.fillStyle = "#CC0";
    ct.beginPath();
    ct.moveTo(24,10);
    ct.lineTo(24,20);
    ct.lineTo(32,15);
    ct.closePath();
    ct.fill();
}
```
A Tree for the Bird
function treeGeometry(ct) {
    "use strict";
    ct.strokeStyle = "#8B4513";
    ct.lineWidth = 5;
    ct.beginPath();
    ct.moveTo(30, 0);
    ct.bezierCurveTo(20, 50, 20, 50, 20, 100);
    ct.lineTo(-20, 100);
    ct.bezierCurveTo(-20, 50, -20, 50, -30, 0);
    ct.stroke();
    ct.strokeStyle = "#0C0";
    ct.fillStyle = "#DFE";
    ct.beginPath();
    ct.arc(0, 175, 75, 0, 2 * Math.PI, false);
    ct.fill();
    ct.stroke();
}
function drawTree(ct) {
  treeGeometry(ct);
  ct.save();
  birdInTree.apply(ct);  // transform
  drawBird(ct);
  ct.restore();
}
Bird in Tree, Tree in Park
And a person to look at the bird...
The Eye Coordinate System
(or camera)
Look At (the bird)
Perspective (just wait)
Look At (the bird)
Where is everything in the Park?

Points in Park Coords

Tree in Park

Bird in Tree

Points in Tree Coords

Persons in Park

Points in Person Coords

Eye in Person

Eye Point 0

Points in Bird Coords
Bird in Eye (Camera) Coordinates

- Bird in Tree
- Tree in Park
- Points in Park Coords
- Person in Park
- Eye in Person
- Points in Person Coords
- Points in Eye Coordinates
- Points in Bird Coords
\[ e = M_{e}^{-1} M_{p}^{-1} M_{t} M_{b} p \]
\[ e = M_e^{-1} M_p^{-1} M_t M_b p \]
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
Next Problem: Projection

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

Types:
  Orthographic
  Perspective
  some others we won’t talk much about
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 Projections

Objects that are far away look smaller
My sample P3 . . .

Orthographic

Perspective
Drawing in Perspective
A draughtsman drawing a portrait, Albrecht Dürer, 1532. E.44-1894

http://www.vam.ac.uk/content/articles/d/drawing-techniques/
Perspective Imaging

Eye Point
a.k.a.
Focal Point

Image Plane
Perspective Assumptions

There is a single focal point

Simplifying Assumptions: (not required)
Image plane orthogonal to view direction
Image plane centered on view direction
Image plane in front of eye
Image plane behind eye

d is the focal length
Pinhole Camera

d
Perspective math

- $e$
- $g$
- $y_s$
- $y$
- $d$
- $z$
Perspective math

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

d, z, e, g, view plane
Perspective Projection

This is linear in homogeneous coordinates

\[
x_s = \frac{d}{z} x \quad y_s = \frac{d}{z} y
\]

Rotate so \( w = z \), the divide by \( w \) does the job

Details in the book (or next class)
Homogeneous Coordinates

What does this $\mathbf{w}$ axis do?

So far, it’s always been:

1 (for points)
0 (for vectors)
Projective Equivalence

In homogeneous coordinates, points correspond to rays through the origin.

In 3D...

The point \(x, y, z\)

Has a set of homogeneous coordinates:

\([x, y, z, 1] \quad [2x, 2y, 2z, 2], \ldots\)
The divide by w

For a 3D point \(x, y, z\)
It can be anything of the form:
\[
[\alpha x, \alpha y, \alpha z, \alpha]
\]

We consider the “hyperplane” \([w=1]\) to be our “regular” 3D space. So we can convert:
\[
\left[\frac{x}{\alpha}, \frac{y}{\alpha}, \frac{z}{\alpha}, 1\right] \text{ - divide by } w
Projective coordinates for 1D

1D \((x)\) becomes \((x, w)\)
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
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

Window (Screen) – in pixels
Normalized Device – [-1 1]
Camera / Eye
World
Object . . .
Local
Program 3 Hints

All the transforms (even projection) are matrices.

The matrix library has them implemented. You don’t have to implement them yourself, but it’s important to understand them.

TWGL’s “screen” coordinates are [-1, 1]. You have to convert to Canvas Pixels.
Demos