Drawing Fast
(this time with depth)
Partial lecture: most time spent on P3
Rasterization

CS559 – Fall 2015
Lecture 8
September 29, 2015
Some questions that come up

Are my normals pointing the right way?
Your real questions...

What would have helped you with P3?
If you got stuck – where?
If you didn’t– where could we have helped?

What would help most with P4?
The Key Ideas of Computer Graphics (559 Learning Goals)

by MIKE GLEICHER on AUGUST 18, 2015 [EDIT]

A list of the key ideas of Computer Graphics that we want students to understand in CS559.

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.
5. The abstractions of interactive rendering provide building blocks to do lots of things – beyond what they are obviously intended for. Many tricks involve putting these building blocks together in interesting ways to achieve interesting effects. The specific tricks change over time, but the paradigm of assembling the basic elements in interesting ways is fundamental.
6. Organize your computations for the hardware. Use programmable shading and block buffer transfers to make stuff go fast (if you’re using an API that lets you talk to the hardware).
7. Determine apparent color of objects based on lighting models. Use simple, local lighting models for efficiency, and use other tricks for non-local effects.
8. Use texture to describe detail (such as color patterns) on objects. Set up texturing with texture
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.
What does it take to draw in 3D?

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**    Get triangles
2. Figure out what color it should be
   **Shading**    Do lighting
3. Position relative to the Eye
   **Viewing**    View transform
4. Get rid of stuff behind you/offscreen
   **Clipping**    Don’t worry
5. Figure out where it goes on screen
   **Projection**  Projection Transform
6. Figure out if something else blocks it
   **Visibility**  Z-Buffer
7. Draw the 2D primitive
   **Rasterization**  (Let library do it)
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...
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
Rasterization

Input:
    primitive (in screen coords)

Output
    list of pixels “covered”
What primitives

Points
Lines
Triangles

Generally build other things from those
Approximate curves
Rasterizing Points

Easy! 1 pixel – and we know where

Issues:

What if we want different sizes?
(points smaller than a pixel?)

Discretization?
(pixels are an integer grid)
Welcome to the world of Aliasing

The real world is (effectively) continuous
Our models are continuous

Our displays are discrete

This is a deep problem – we’ll come back
Do I care about Aliasing?

Jaggies
Crawlies
Things not moving smoothly
Can’t make small things
Can’t put things where you want
Errors add up to weird patterns
(or, simply, Yes)
Preview: Dealing with Aliasing

Little Square Model
Not preferred
Simpler to start
Preview: Dealing with Aliasing

Simple Drawing

Pick the:
Nearest pixel (center)

Fill the pixel
Preview: Dealing with Aliasing

Simple Drawing

Pick the:
Nearest pixel (center)
(cover multiple pixels)
Fill the pixel
Preview: Dealing with Aliasing

Simple Drawing

Pick the:

Nearest pixel (center) (cover multiple pixels)
Fill the pixel (partially fill pixel)
Dealing with Aliasing?

Simple:
Aliased (jaggies, ...)
Crisp

Anti-Aliased:
Less aliased
Blurry
Other Anti-Aliasing Issues

Z-Buffer is a binary choice

Partially filling can be a problem

Depends on lots of other stuff

Really elegant math! Good theory!
Lines

Historical:
Vector graphics hardware
Simulate with “new” pixel-based (CRT)

Brezenham’s Algorithm (1960s)
Integer only line drawing
No divisions
Today?

Floating point is cheap
Division isn’t too expensive

Make lines into degenerate triangles
Triangles (Polygons)

- The really important primitive
- Determine which pixels are covered
  - Also do interpolation (UV, color, W, depth)
- Scan conversion
  - Generically used as a term for rasterization
  - An old algorithm that isn’t used by hardware
- Not to be confused with Scanline rendering
  - Related, but deals with whole scenes
Scan Conversion Algorithm

• Idea:
  – Scan top to bottom
  – “walk edges” (active edge list)
  – Scan left to right

Active Edges (for this scanline)

Brezenham’s Alg (or equiv) to get begin/end

Change active list at vertex
Scan-Conversion

• Cool
  – Simple operations, very simple inner loops
  – Works for arbitrary polygons (active list management tough)
  – No floating point (except for interpolation of values)

• Downsides
  – Very serial (pixel at a time) / can’t parallelize
  – Inner loop bottle neck if lots of computation per pixel
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

\[ 0 \leq \alpha, \beta, \gamma \leq 1 \]
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

\[ 0 \leq \alpha, \beta, \gamma \leq 1 \]
Linear Interpolation

Interpolative coordinate (t)

$0 \leq t \leq 1$ then in line segment