Drawing Fast
The Graphics Pipeline

CS559 – Fall 2016
Lectures 10 & 11
October 10th & 12th, 2016
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** / Camera Transformation
4. Get rid of stuff behind you/offscreen
   **Clipping** Don’t worry
5. Figure out where it goes on screen
   **Projection** (sometimes called Viewing)
6. Figure out if something else blocks it
   **Visibility** / Occlusion Z-Buffer
7. Draw the 2D primitive
   **Rasterization** Let library do it
Rasterization

Figure out which pixels a primitive “covers”

Turns primitives into pixels
Lines
Triangles
Hardware Rasterization

For each point:
Compute barycentric coords
Decide if in or out
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
Linear Interpolation

\[ P = (1-t) A + t B \quad (t \text{ is the coord}) \]

Interpolative coordinate (t) \[ P = (1-t) A + t B \]

\[ 0 \leq t \leq 1 \text{ then in line segment} \]

- \[ 1A + 0B \]
- \[ \frac{1}{2} A + \frac{1}{2} B \]
- \[ 0A + 1B \]
- \[ -\frac{1}{2} A + 1\frac{1}{2} B \]
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 Coords are Useful!

Every point in plane has a coordinate
\((\alpha \ \beta \ \gamma)\) such that: \(\alpha + \beta + \gamma = 1\)

Easy test inside the triangle
\(0 \leq \alpha, \beta, \gamma \leq 1\)

Interpolate values across triangles
\[x_p = \alpha \ x_1 + \beta \ x_2 + \gamma \ x_3\]
\[c_p = \alpha \ c_1 + \beta \ c_2 + \gamma \ c_3\]
Hardware Rasterization

For each point:
Compute barycentric coords
Decide if in or out
Wasteful?

Can do all points in parallel

We want the coordinates (coming soon)

Does the right things for touching triangles
Each point in 1 triangle
Note

Triangles are independent

Even in rasterization

(they are independent throughout process)
The steps of 3D graphics

Model objects (make triangles)
Transform (find point positions)
Shade (lighting– per tri / vertex)
Transform (projection)
Rasterize (figure out pixels)
Shade (per-pixel coloring)
Write pixels (with Z-Buffer test)
A Pipeline

Triangles are independent
Vertices are independent
Pixels (within triangles) are independent
  (caveats about sharing for efficiency)

Don’t need to finish 1 before start 2
(might want to preserve finishing order)
A Pipeline

transf  light  project  raster  shade  write
A Pipeline
Vertices are independent
Parallelize!

1

transf
transf
transf
proj
proj
raster
shade
write
Parallelization

Vertex operations
- split triangles / re-assemble
- compute per-vertex not per-triangle

Pixel (fragment) operations
- lots of potential parallelism
- less predictable

Use queues and caches
Why do we care?

This is why the hardware can be fast.

It requires a specific model.

Hardware implements this model.

The programming interface is designed for this model. You need to understand it.
Some History...

Custom Hardware (pre-1980)
  rare, each different
Workstation Hardware (early 80s-early 90s)
  increasing features, common methods
Consumer Graphics Hardware (mid 90s-)
  cheap, eventually feature complete
Programmable Graphics Hardware (2002-)

Graphics Workstations 1982-199X

Implemented graphics in hardware

Providing a common abstraction set

Fixed function –

it was built into the hardware
Silicon Graphics (SGI)

Stanford Research Project 1980
Spun-off to SGI (company) 1982

The Geometry Engine
   4x4 matrix multiply chip
   approximate division
Raster engine (Z-buffer)
1988: The Personal Iris
The 4D-2X0 series

4 processors (240)

Different graphics

1988 – GT/GTX
1990 - VGX
Why do you care?

It’s the first time the abstractions were right later stuff adds to it
It’s where the programming model is from it was IrisGL before OpenGL
It’s the pipeline at it’s essense we’ll add to it, not take away
The Abstractions

Points / Lines / Triangles
Vertices in 4D
Color in 4D (RGBA = transparency)
Per-Vertex transform (4x4 + divide by w)
Per-Vertex lighting
Color interpolation
Fill Triangle
Z-test (and other tests)
Double buffer (and other buffers)
What’s left to add?

All of this was in software in the 80s
1990 – texture
1992 – multi-texture (don’t really need)
2002 – programmable pipelines
2005 – more programmability
The pipeline (2006-current)
The pipeline (1988)
The full fixed-function pipeline (1992)
The parts you **have** to program

1988-2014

Now (in addition to above)
A Triangle’s Journey
Things to observe as we travel through the pipeline...

What does each stage do?
What are its inputs and output?
   important for programmability
Why would it be a bottleneck?
   and what could we do to avoid it
The pipeline (1988) (no texturing)
Start here
Setup modes (window, ...)
Setup transform, lights
Draw a triangle
Position, color, normal
Drawing a triangle

**Modes** per triangle

which window, how to fill, use z-buffer, ...

**Data** per-vertex

position

normal

color

other things (texture coords)
Per Vertex?

**Modes** per triangle

which window, how to fill, use z-buffer, ...

**Data** per-vertex

position

normal ← allow us to make non-flat
color ← allows us to interpolate

other things (texture coords)
Per-Vertex not Per-Triangle

Allows sharing vertices between triangles

Or make all the vertices the same (color, normal, ...) to get truly flat
Triangle
V1: (x1, y1, z1), (r1, g1, b1), (nx1, ny1, nz1)
V2: (x2, y2, z2), (r2, g2, b2), (nx2, ny2, nz2)
V3: (x3, y3, z3), (r3, g3, b3), (nx3, ny3, nz3)
Triangle
V1: (x1, y1, z1), (r1, g1, b1), (nx1, ny1, nz1)
V2: (x2, y2, z2), (r2, g2, b2), (nx2, ny2, nz2)
V3: (x3, y3, z3), (r3, g3, b3), (nx3, ny3, nz3)
Is this a potential bottleneck?

Function calls to the driver
3 vertices + triangle + ...
Old style OpenGL

begin(TRIANGLE);
c3f(r1,g1,b1);
n3f(nx1,ny1,nz1);
v3f(x1,y1,z1);
c3f(r2,g2,b2);
n3f(nx2,ny2,nz2);
v3f(x2,y2,z2);
c3f(r3,g3,b3);
n3f(nx3,ny3,nz3);
v3f(x3,y3,z3);
end(TRIANGLE);

11 function calls
35 arguments pushed

Old days:
This is a lot less than the number of pixels!

Nowadays:
Just the memory access swamps the process
Coming Soon...

Block transfers of data

Data for lots of triangles moved as a block
Try to draw groups of triangles
Triangle
V1: (x1,y1,z1), (r1,g1,b1),(nx1,ny1,nz1)
V2: (x2,y2,z2), (r2,g2,b2),(nx2,ny2,nz2)
V3: (x3,y3,z3), (r3,g3,b3),(nx3,ny3,nz3)
Split up triangles into *vertices*

V1: \((x_1, y_1, z_1), (r_1, g_1, b_1), (nx_1, ny_1, nz_1)\)

V2: \((x_2, y_2, z_2), (r_2, g_2, b_2), (nx_2, ny_2, nz_2)\)

V3: \((x_3, y_3, z_3), (r_3, g_3, b_3), (nx_3, ny_3, nz_3)\)
Buffer / Queue the **vertices**

V1: \((x_1, y_1, z_1), (r_1, g_1, b_1), (n_{x1}, n_{y1}, n_{z1})\)

V2: \((x_2, y_2, z_2), (r_2, g_2, b_2), (n_{x2}, n_{y2}, n_{z2})\)

V3: \((x_3, y_3, z_3), (r_3, g_3, b_3), (n_{x3}, n_{y3}, n_{z3})\)
Buffering Vertices

Old Days:
- Vertex processing expensive
- Try to maximize re-use
- Process once an use for many triangles

Nowadays
- Getting vertex to hardware is expensive
- Process vertices in parallel
Buffer / Queue the **vertices**

V1: \((x_1, y_1, z_1), (r_1, g_1, b_1), (nx_1, ny_1, nz_1)\)

V2: \((x_2, y_2, z_2), (r_2, g_2, b_2), (nx_2, ny_2, nz_2)\)

V3: \((x_3, y_3, z_3), (r_3, g_3, b_3), (nx_3, ny_3, nz_3)\)
Process each vertex independently

Transform – compute $x'$ and $n'$

Clip

Light – compute $c'$
Vertex in $\rightarrow$ Vertex out
In: $x, n, c$
Out: $x, n, c, x', n', c'$
Vertex Processing

Just adds information to vertices

Computes transformation
screen space positions, normals
Computes “lighting”
new colors

(in the old days, clipping done here hence TCL)
Vertex Processing:
Each vertex is independent

Inputs are:
vertex information for this vertex
any “global” information
current transform, lighting, ...

Outputs are:
vertex information for this vertex
Looking ahead...
When we program this pipeline piece
It will still be: Vertex in ➔ Vertex out

Looking ahead…
When we program this pipeline piece
It will still be: Vertex in ➔ Vertex out
Store processed vertices in a **cache**

Store several so that we can re-use if many triangles share a vertex
Vertex Caching

Old days:
Big deal, important for performance

Now:
Not even sure that it’s always done
Put triangles back together
This is one of the few places where triangles exist
In the fixed-function pipeline, there is no “triangle processing” step (maybe clipping)
**Rasterizer:**
Convert triangles to a list of pixels

**Input:** Triangle – with values \((x', c', \ldots)\)

**Output:** Pixels – with values \((x', c', \ldots)\)
Pixels or Fragments

I am using the terms interchangably (actually, today I am using pixel)

Technically...

Pixel = a dot on the screen
Fragment = a dot on a triangle

might not become a pixel (fails z-test)
might only be part of a pixel
Where do pixel values come from?

Each vertex has values
Each pixel comes from 3 vertices

Pixels interpolate their vertices’ values
  Barycentric interpolation
All values (in a pixel) are interpolated
Each triangle is separate

Careful processing of edges so no cracks

1 triangle → many pixels
Rasterizer: Convert triangles to a list of pixels

**Input:** Triangle per-vertex info*3

**Output:** Pixels interpolated info

---

Application Program

Graphics Driver

Command Buffer (Triangle Queue)

Vertex Queue

Vertex Processing (TCL)

Vertex Cache

Assembly

Triangle Processing

Rasterize

Pixel Queue

Pixel Processing

Pixel Tests

Texture Memory

Render to texture

Frame Buffer
Each triangle can make lots of pixels

We need to process pixels in parallel (all of the remaining steps)
Process each pixel to get its final values

Usually color, but sometimes Z

This step is a no-op in the 1988 pipeline
Pixel in $\rightarrow$ Pixel out, each independent

All we do is change its values (potentially re-compute)
Coming attractions...

This step will get to be pretty exciting
Pixel Processing Ground Rules

Pixels are independent
Pixel in $\rightarrow$ Pixel out
Changing its position \((x,y)\) makes it a different pixel (so you can’t)

Can change other values
Or “reject”
Application Program

Graphics Driver

Command Buffer (Triangle Queue)

Vertex Queue

Vertex Processing (TCL)

Vertex Cache

Assembly

Triangle Processing

Geometry Shading

Rasterize

Pixel Queue

Pixel Processing

Pixel Shading

Pixel Tests

Texture Memory

Render to texture

Frame Buffer

Coming attractions...

This step will get to be pretty exciting
Consider the pixel and it’s destination Z-buffer (is it in front?)
Alpha / color / stencil - tests
Color blending
Then (and only then) can we write
Each pixel requires a read/write cycle

Potential memory bandwidth bottleneck (cache writes)
There are worse memory bottlenecks
We’ve made it!

Now the frame buffer gets sent to the screen. (at the appropriate time)
What if we didn’t make it...

Suppose the triangle’s pixels are occluded
Removed by the z-buffer
Normal Z-Test

Happens at the end
We wasted all that work!