3D Drawing
Visibility & Rasterization

CS559 – Fall 2017
Lecture 9
October 4 2017
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)
Orthographic Projection

Projection = transformation that reduces dimension

Orthographic = flatten the world onto the film plane
Perspective Projection

Eye point
Film plane
Frustum

Simplification
Film plane centered with respect to eye
Sight down Z axis
• Can transform world to fit
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)

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

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

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

N = near distance, F = far distance
Z = n put on front plane, z=f put on far plane

\[
P = \begin{pmatrix}
  n & 0 & 0 & 0 & 0 \\
  0 & n & 0 & 0 & 0 \\
  0 & 0 & n+f & -fn & 0 \\
  0 & 0 & 1 & 0 & 0 \\
\end{pmatrix}
\]
Shirley’s Perspective Matrix

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

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

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

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

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

assuming height is 1

\([-\text{zNear}, -\text{zFar}]\) remapped to \([-1, +1]\)
Field of View

\[ \theta \]

\[ d \]
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

writeColor(@pixel) becomes:
  readZ(@pixel)
  test
  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!
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
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 me offline for detailed 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

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

\( N = \text{unit normal vector} \)
\( L = \text{unit light direction vector} \)
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
- Projection loses normals
- Camera space is OK

Normalized Device – [-1 1]
- Camera / Eye
- 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)
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)
(covers multiple pixels)
Fill the pixel
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
\[ 0 \leq \alpha, \beta, \gamma \leq 1 \]
Linear Interpolation

$1A+0B \quad \frac{1}{2} A+\frac{1}{2} B \quad 0A+1B$

Interpolative coordinate $(t)$

$0 \leq t \leq 1$ then in line segment
Dealing with Aliasing?

Simple:
Aliased (jaggies, …)
Crisp

Anti-Aliased:
Less aliased
Blurry
Lines
Triangles
Hardware Rasterization

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

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

Interpolative coordinate \( t \)

\[ 1A + 0B \quad \frac{1}{2} A + \frac{1}{2} B \quad 0A + 1B \quad -\frac{1}{2} A + 1\frac{1}{2} B \]

\[ 0 \leq t \leq 1 \text{ then on line segment} \]
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
Hardware Rasterization

Each center point in one triangle

If we choose consistently for “on-the-edge” cases

Over simplified version:
\[ 0 \leq a, b, c < 1 \]
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)
Pipelining in conventional processors

Start step 2 before step 1 completes

\[ C = A \times B \]
\[ F = D \times E \]
\[ J = G \times H \]

Unless step 2 depends on step 1

Pipe Stall

\[ C = A \times B \]
\[ F = D \times C \]
\[ J = G \times H \]
Triangles are independent! No stalls! (no complexity of handling stalls)

Start step 2 before step 1 completes

Unless step 2 depends on step 1
Pipe Stall

\[
\begin{align*}
C &= A \times B \\
F &= D \times E \\
J &= G \times H \\
C &= A \times B \\
F &= D \times C \\
J &= G \times H \\
C &= D \times C \\
J &= G \times H
\end{align*}
\]
A Pipeline

1. transf
2. light
3. project
4. raster
5. shade
6. write
A Pipeline

1. transf  light  project  raster  shade  write
2. transf  light  project  raster  shade  write
3. transf  light  project  raster  shade  write
Vertices are independent
Parallelize!
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)

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
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  

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

Frame Buffer

Render to texture
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: \((x_1, y_1, z_1), (r_1, g_1, b_1), (nx_1, ny_1, nz_1)\)

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

V3: \((x_3, y_3, z_3), (r_3, g_3, b_3), (nx_3, ny_3, nz_3)\)
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 $\rightarrow$ 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 $\rightarrow$ many pixels
Rasterizer:
Convert triangles to a list of pixels

**Input:** Triangle per-vertex info*3  
**Output:** Pixels interpolated info

---

**Rasterizer:**
Convert triangles to a list of pixels

- **Input:** Triangle per-vertex info*3
- **Output:** Pixels interpolated info
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”
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!