Where does a picture come from?

• Result: image (raster)
• Input 2D/3D model of the world

• Rendering
  – term usually applied to whole scene
  – Implication of caring about quality
• Rasterization
  – term usually applied to individual primitives

Not just about fancy 3D!

• Rendering fonts
  – Really want it to look good
  – Have to do a lot of it
  – Complex shapes
  – Complex aliasing issues (since things are small)

Rendering/Rasterization

• Do the whole scene at once
  – Collect everything
• Do each primitive at a time

• Different algorithms and tradeoffs

When do we care?

• Rasterization
• Usually done by low-level
  – OS / Graphics Library / Hardware
  – Hardware implementations counter-intuitive
    • Modern hardware doesn’t work anything like what you’d expect

• High quality rendering
• Really high-quality 2D rendering
• Understanding of how to best use hardware

The simplest case: Points

• Not all that interesting – but good for bringing up aliasing issues
Drawing Points

• What is a point?
  – Position – without any extent
  – Can’t see it – since it has no extent, need to give it some
• Position requires co-ordinate system
  – Consider these in more depth later

• How does a point relate to a sampled world?
  – Points at samples?
  – Pick closest sample?
  – Give points finite extent and use little square model?
  – Use proper sampling

Sampling a point

• Point is a spike – need to LPF
  – Gives a circle w/roll-off
• Point sample this

• Or...
  – Samples look in circular (kernel shaped) regions around their position

• But, we can actually record a unique “splat” for any individual point

Anti-Aliasing

• Anti-Aliasing is about avoiding aliasing
  – once you’ve aliased, you’ve lost
• Draw in a way that is more precise
  – E.g. points spread out over regions

• Not always better
  – Lose contrast, might not look even if gamma is wrong, might need to go to binary display, …

Line drawing

• Was really important, now, not so important
• Let us replace expensive vector displays with cheap raster ones

• Modern hardware does it differently
  – Actually, doesn’t draw lines, draws small, filled polygons

• Historically significant algorithms
• Good for considering issues

Line Drawing (2)

• Consider the integer version
  – (x1,y1) -> (x2,y2) are integers
  – Not anti-aliased (binary decision on pixels)
• Naïve strawman version:
  – Y = mx + b

  For x = x1 to x2
  y = mx + b
  set(x,y)

• Problems:
  – Too much math (floating point)
  – gaps

Brezenham’s algorithm

(and variants)

• Consider only 1 octant (get others by symmetry)
  – 0 >= m >= 1
• Loop over x pixels
  – Guarantees 1 per column
• For each pixel, either move up 1 or not
  – If you plotted x,y then choose either x+1,y or x+1,y+1
  – Trick: how to decide which one easily
  – Same method works for circles (just need different test)
• Decision variable
  – Implicit equation for line (d=0 means on the line)
Midpoint method

\[ x_{k+1}, y_{k+1} \]

\[ d_1 = y - y_k \]
\[ d_2 = y_{k+1} - y \]

If \( d_1 < d_2 \) pick \( y_k \) otherwise pick \( y_{k+1} \)

If \( d_1 - d_2 < 0 \) pick \( y_k \)

\[ \Delta d = d_1 - d_2 \]
\[ \Delta d = (y - y_k) - (y_{k+1} - y) \]
\[ y = m(x_{k+1}) + b \]
\[ \Delta d = 2(m(x_{k+1}) + b) - 2y_k - 1 \]
\[ m = \frac{\Delta y}{\Delta x} \]

Multiply both sides by \( \Delta x \) (since we know its positive)
\[ \Delta d x = 2\Delta y x_k + 2\Delta y + 2b\Delta x - 2\Delta x y_k - \Delta x \]
\[ P_k = \Delta d x = 2\Delta y x_k + 2\Delta x y_k + c \]
\[ c = 2\Delta y + \Delta x(2b - 1) \]

(All the stuff that doesn’t depend on \( k \))

Derivation

Incremental Algorithm

- Suppose we know \( p_k \) – what is \( p_{k+1} \)?
- \( p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \)
  - Since \( x_{k+1} = x_k + 1 \)
- And \( y_{k+1} - y_k \) is either 1 or 0, depending on \( p_k \)

Why is this cool?

- No division!
- No floating point!
- No gaps!
- Extends to circles
- But...
  - Jaggies
  - Lines get thinner as they approach 45 degrees
  - Can’t do thick primitives

Brezenham’s Algorithm

- \( P_{x+k} = 2\Delta y + x \)
- \( Y = y_1 \)
- For \( X = x_1 \) to \( x_2 \)
  - Set \( X, Y \)
  - If \( P_x < 0 \)
    - \( Y += 1 \)
    - \( P_x += 2 \Delta y - 2 \Delta x \)
  - Else:\( P_x += 2 \Delta y \)

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

Modern Rasterization (in hardware)

- Generate pixel candidates
- Compute barycentric coords for each pixel
- Decide whether or not its inside triangle

- Why?
  - Easier for hardware
  - Parallel (each pixel somewhat independent)
  - Need barycentric coords for interpolation
  - Breaks work into even sized chunks
    - (regular tiles / groups of pixels)

What is scanline algorithm?

- Not scan conversion algorithm

- Keep all polygons – sort by Y value on screen, then by X
- Scan across each line of image – keep track of what polygons each pixel covers

- Why?
  - Image-space algorithm
  - One line Z-buffer (1 pixel Z-Buffer)
  - Makes anti-aliasing easier (know all polys that affect pixel)
  - Done by software renderers (that aren’t ray tracers)

The whole process: graphics pipeline

- Primitives (triangles) in – frame buffer writes out
  - Actually, any memory that we store images modified
- Software does the same steps

- Why pipeline?
  - Do step 1, then step 2, …
  - Can have one object in each step
  - Steps don’t depend on each other too much

- Parallelism mainly in hardware

Pipelining in conventional processors

- Start step 2 before step 1 completes
- Unless step 2 depends on step 1
- Pipe Stall

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

\[
\begin{align*}
C &= A \times B \\
F &= D \times C \\
J &= G \times H
\end{align*}
\]
Graphics Hardware / Interactive Rendering

- Key Idea: Set of basic abstractions
  - Z-buffer, texture, triangles, ...
- Implement these really well
- Let programmers figure out how to use it to do other things
- Expand abstractions based on what people figure out to do

History of Hardware

- 1980s – first workstation 3D hardware (SGI)
- 1990s – extension of abstraction set
  - Texture mapping, compositing, multi-buffering
- 1990s – first PC graphics hardware
  - Low end (Apple’s white magic project)
  - High end (3D solutions – expensive)
- 2000s – consumer graphics hardware
  - Driven by gaming market
  - Extensive use of the abstractions
- 2002++ - programmable graphics hardware
  - Better abstractions, generality, use as GP processor

Basic Graphics Pipeline

- Triangles
- Transform
- Clip
- Lighting
- Assembly
- Setup
- Rasterization (generate fragments)
- Per-Pixel Coloring
  - Shading, …
  - Pixel Tests
  - Memory Writes

What's a fragment

- It will be a pixel when it grows up
- Pixel = place on screen
- Fragment = makes up a pixel
  - Maybe won’t make it to the screen
  - Maybe combined to make a pixel (anti-aliasing)
- Position in the final image is known
  - (e.g. which pixel it contributes to)

Per-Pixel Coloring

- Interpolation (handled by rasterization)
- Texture lookup / blending
- Per-Pixel Lighting (if its allowed)
- Arbitrary programs (we’ll get to that later)
Per-Fragment operations

- Stencil test
- Window clipping
- Other things
- Z-testing
  - Note this is late (lots of work done and thrown away)
  - Could do Z-test earlier (maybe)

Memory writes

- Need to do read/modify (for z-buffer / stencil)
- Useful for color as well:
  - Alpha blending
  - Multi-pass operations

Basics of graphics performance

- Where is the bottleneck?
  - Getting triangles into the pipeline
  - Transforming the vertices
  - Rasterization
  - Doing the per-pixel operations
  - Getting texture for per-pixel operations
  - Reading/writing to memory
- Different systems have different bottlenecks
  - And the bottlenecks are moving

The Fixed-Function Pipeline

- We know what each block does
  - Vertex
    - Projection matrix, divide by Z, Phong lighting (per vertex)
    - Color, UV, Z, W, per vertex
  - Fragment
    - Interpolate colors
    - Look up textures – blend (and apply over interpolated lighting)
- Want more (per-pixel lighting, normal maps),…
  - But which one to put in hardware
  - All of them! (make it flexible and programmable)

Vertex Processor

- What comes in?
  - Vertex info (position, normal, assigned color&UV)
  - State information (matrices, lighting, …)
- What goes out
  - Vertex info – just now in screen space
- All the per-vertex operations do is change the values around!

Fragment Processor

- What comes in
  - Info about fragment
    - Interpolated from vertices
    - Position XY (can’t be changed, since it’s the “identity” of the fragment)
    - Other properties Z,W, UV, Color
  - State information (lighting, bound textures, …)
- What goes out
  - Info about fragment (mainly color, but also new Z/W)
- Can’t really change X/Y
Programming the pipeline

• Write "little" functions for each
• Remember what each can "do" (inputs and outputs)

• Each gets applied a lot
  – To every vertex
  – To every fragment
• But applied in parallel (so it can be fast)