Computer Graphics

- Rasterization -

Philipp Slusallek
Rasterization

**Definition**
- Given some 2D geometry (point, line, circle, triangle, polygon,…), specify which pixels of a raster display each primitive covers
  - Often also called “scan-conversion”
- Anti-aliasing: instead of only fully-covered pixels (single sample), specify what parts of a pixel are covered (multi/super-sampling)

**Perspectives**
- OpenGL lecture: from an application programmer’s point of view
- This lecture: from a graphics package implementer’s point of view
- Looking at rasterization of (i) lines and (ii) polygons (areas)

**Usages of rasterization in practice**
- 2D-raster graphics, e.g. Postscript, PDF, SVG, …
- 3D-raster graphics, e.g. SW rasterizers (Mesa, OpenSWR), HW
- 3D volume modeling and rendering
- Volume operations (CSG operations, collision detection)
- Space subdivision (spatial indices): construction and traversal
**Rasterization**

**Assumptions**
- Pixels are sample **points** on a 2D integer grid
  - OpenGL: at cell bottom-left, integer-coordinate
  - X11, Foley: at the cell center (we will use this)
- Simple raster operations
  - Just setting pixel values or not (binary decision)
  - More complex operations later: compositing/anti-aliasing
- Endpoints snapped to (sub-)pixel integer coordinates
  - Simple and consistent computations with fixed-point arithmetic
- Limiting to lines with gradient/slope $|m| \leq 1$ (mostly horizontal)
  - Separate handling of horizontal and vertical lines
  - For mostly vertical, swap $x$ and $y$ ($|1/m| \leq 1$), rasterize, swap back
    - Special cases in SW, trivial in HW :-)
- Line width is one pixel
  - $|m| \leq 1$: 1 pixel per column (X-driving axis)
  - $|m| > 1$: 1 pixel per row (Y-driving axis)
Lines: As Functions

• **Specification**
  – Initial and end points: \((x_b, y_b), (x_e, y_e), (dx, dy) = (x_e - x_b, y_e - y_b)\)
  – Functional form: \(y = mx + B\)
  – End points with integer coordinates \(\Rightarrow\) rational slope \(m = \frac{dy}{dx}\)

• **Goal**
  – Find that pixel per column whose distance to the line is smallest

• **Brute-force algorithm**
  – Assume that +X is the driving axis \(\rightarrow\) set pixel in every column
    
    \[
    \text{for } x_i = x_b \text{ to } x_e \\
    y_i = m \times x_i + B \\
    \text{setPixel}(x_i, \text{Round}(y_i)) \quad // \text{Round}(y_i) = \text{Floor}(y_i + 0.5)
    \]

• **Comments**
  – Variables \(m\) and thus \(y_i\) need to be calculated in floating-point
  – Not well suited for direct HW implementation
    • A floating-point ALU is significantly larger in HW than integer
**Lines: DDA**

- **DDA: Digital Differential Analyzer**
  - Origin of incremental solvers for simple differential equations
    - The Euler method
    - Per time-step: \( x' = x + \frac{dx}{dt}, \ y' = y + \frac{dy}{dt} \)

- **Incremental algorithm**
  - Choose \( dt=dx \), then per pixel
    - \( x_{i+1} = x_i + 1 \)
    - \( y_{i+1} = m \times x_{i+1} + B = m(x_i + 1) + B = (m \times x_i + B) + m = y_i + m \)
    - setPixel\((x_{i+1}, \text{Round}(y_{i+1}))\)

- **Remark**
  - Utilization of coherence through incremental calculation
    - Avoids the “costly” multiplication
  - Accumulates error over length of the line
    - Up to 4k additions on UHD!
  - Floating point calculations may be moved to fixed point
    - Must control accuracy of fixed point representation
    - Enough extra bits to hide accumulated error (>>12 bits for UHD)
**Lines: Bresenham (1963)**

- **DDA analysis**
  - Critical point: decision whether we need rounding up or down

- **Idea**
  - Integer-based decision through implicit functions
  - Implicit line equation
    - \( F(x, y) = ax + by + c = 0 \)
    - Here with \( y = mx + B = \frac{dy}{dx} x + B \) \( \Rightarrow 0 = dy x - dx y + B dx \)
    - \( a = dy, \quad b = -dx, \quad c = Bdx \)
  - Results in
    - \( F(x, y) = dy x - dx y + dx B = 0 \)

\[ F(x, y) = 0 \]
\[ F(x, y) < 0 \]
\[ F(x, y) > 0 \]
Lines: Bresenham

- **Decision variable $d$ (the midpoint formulation)**
  - Assume we are at $x=i$, calculating next step at $x=i+1$
  - Measures the vertical distance of midpoint from line:
    \[
    d_{i+1} = F(M_{i+1}) = F(x_i + 1, y_i + 1/2) = a(x_i + 1) + b(y_i + 1/2) + c
    \]

- **Preparations for the next pixel**
  
  IF $(d_{i+1} \leq 0)$ // Increment in $x$ only
  
  $d_{i+2} = d_{i+1} + a = d_{i+1} + dy$ // Incremental calculation
  
  ELSE // Increment in $x$ and $y$
  
  $d_{i+2} = d_{i+1} + a + b = d_{i+1} + dy - dx$
  
  $y = y + 1$
  
  ENDIF
  
  $x = x + 1$
Lines: Integer Bresenham

- **Initialization**
  
  $d_1 = F\left(x_b + 1, y_b + \frac{1}{2}\right) = a(x_b + 1) + b\left(y_b + \frac{1}{2}\right) + c$
  
  $= ax_b + by_b + c + a + \frac{b}{2} = F(x_b, y_b) + a + \frac{b}{2} = a + \frac{b}{2}$
  
  - Because $F(x_b, y_b)$ is zero by definition (line goes through $(x_b, y_b)$)
    
    • Pixel is always set (but check consistency rules → later)

- **Elimination of fractions**
  
  - Any positive scale factor maintains the sign of $F(x,y)$
    
    • $2F(x_b, y_b) = 2(ax_b + by_b + c) \rightarrow d_{start} = 2a + b$

- **Observation:**
  
  - When the start and end points have integer coordinates then
    $b = -dx$ and $a = dy$ are also integers
      
      • Floating point computation can be eliminated
  
  - No accumulated error!!
Lines: Arbitrary Directions

- **8 different cases**
  - Driving (active) axis: ±X or ±Y
  - Increment/decrement of y or x, respectively
Thick Lines

• **Pixel replication**
  – Problems with even-numbered widths
  – Varying intensity of a line as a function of slope

• **The moving pen**
  – For some pen footprints the thickness of a line might change as a function of its slope
  – Should be as “round” as possible

• **Real Solution: Draw 2D area**
  – Allows for anti-aliasing and fractional width
  – Main approach these days!
Handling Start and End Points

• End points handling (not available in current OpenGL)
  – Joining: handling of joints between lines
    • Bevel: connect outer edges by straight line
    • Miter: join by extending outer edges to intersection
    • Round: join with radius of half the line width
  – Capping: handling of end point
    • Butt: end line orthogonally at end point
    • Square: end line with oriented square
    • Round: end line with radius of half the line width
  – Avoid overdraw when lines join
Bresenham: Circle

- Eight different cases, here +X, y--

  Initialization: \(x = 0, y = R\)
  \(F(x,y) = x^2 + y^2 - R^2\)
  \(d = F(x+1, y-1/2)\)
  IF \(d < 0\)
    \(d = F(x+2, y-1/2)\)
  ELSE IF \(d > 0\)
    \(d = F(x+2, y-3/2)\)
    \(y = y - 1\)
  ENDIF
  \(x = x + 1\)

  - Works because |slope| is smaller than 1

- Eight-way symmetry: only one 45° segment is needed to determine all pixels in a full circle
Reminder: Polygons

- **Types**
  - Triangles
  - Trapezoids
  - Rectangles
  - Convex polygons
  - Concave polygons
  - Arbitrary polygons
    - Holes
    - Overlapping

- **Two approaches**
  - Polygon tessellation into triangles
    - Only option for OpenGL
    - Must mark internal edges so they are not drawn for outlines
  - Direct scan-conversion
    - Mostly in early SW algorithms
Inside-Outside Tests

• What is the interior of a polygon?
  – Jordan curve theorem
    • „Any continuous simple closed curve in the plane, separates the plane into two disjoint regions, the inside and the outside, one of which is bounded.“

• What to do with non-simple polygons?
  – Even-odd rule (odd parity rule)
    • Counting the number of edge crossings with a ray starting at the queried point \( P \) till infinity
    • Inside, if the number of crossings is odd
  – (Non-zero) winding number rule
    • Counts \# times polygon wraps around \( P \)
      – Signed intersections with a ray
    • Inside, if the number is not equal to zero
  – Differences only in the case of non-simple curves (e.g. self-intersection)
Triangle Rasterization

Raster3_box(vertex v[3])
{
    int x, y;
    bbox b;
    bound3(v, &b);
    for (y = b.ymin; y < b.ymax; y++)
        for (x = b.xmin; x < b.xmax; x++)
            if (inside(v, x, y)) // upcoming
                fragment(x, y);
}

• **Brute-force algorithm**
  – Iterate over all pixels within bounding box

• **Possible approaches for dealing with scissoring**
  – Scissoring: Only draw on AA-Box of the screen (region of interest)
    • Test triangle for overlap with scissor box, otherwise discard
    • Use intersection of scissor and bounding box, otherwise as above
    • Important if clipping only against enlarged region! (→ see later)
Rasterization w/ Edge Functions

- **Approach (Pineda, `88)**
  - Implicit edge functions for every edge
    \[ F_i(x, y) = ax + by + c \]
  - Point is *inside* triangle, if every
    \[ F_i(x, y) \] has the same sign
  - Perfect for parallel evaluation at many points
    - Particularly with wide SIMD machines (GPUs, SIMD CPU instructions)
  - Requires “triangle setup”: Computation of 3 edge functions \((a, b, c)\)
  - Evaluation can also be done in homogeneous coordinates

- **Hierarchical approach**
  - Can be used to efficiently check large rectangular blocks of pixels
    - Divide screen into tiles/bins (possibly at several levels)
    - Evaluate \(F\) at tile corners
    - Recurse only where necessary, possibly until subpixel level
Gap and T-Vertices

• **Observations**
  – Pixels set can be non-connected
  – May have overlap and gaps at T-edges

Non-connected pixels: OK
Not OK: Model must be changed
Problem on Edges

- **Consistency**: edge singularity (shared by 2 triangles)
  - What if term \( d = ax+by+c = 0 \) (pixel centers lies exactly on the line)
  - For \( d \leq 0 \): pixels would get set twice
    - Problem with some algorithms
    - Transparency, XOR, CSG, ...
  - Missing pixels for \( d < 0 \) (set by no tri.)
- **Solution**: “shadow” test
  - Pixels are not drawn on the right and bottom edges
  - Pixels are drawn on the left and upper edges
    - Evaluated via derivatives \( a \) and \( b \)
    - Testing for all edges also solves problem at vertices

```c
inside(value d, value a, value b)
{
    // ax + by + c = 0
    return (d < 0) || (d == 0 && !shadow(a, b));
}
shadow(value a, value b)
{
    return (a > 0) || (a == 0 && b > 0);
}
```
Ray Tracing vs. Rasterization

• In-Triangle test (for common origin)
  – Rasterization:
    • Project to 2D, clip
    • Set up 2D edge functions, evaluate for each sample (using 2D point)
  – Ray tracing:
    • Set up 3D edge functions, evaluate for each sample (using direction)
  – The ray tracing test can also be used for rasterization in 3D
    • Avoids projection & clipping

• Enumerating scene primitives
  – Rasterization (simple):
    • Sequentially enumerate them all in any order
  – Rasterization (advanced):
    • Build (coarse) spatial index (typically on application side)
    • Traverse with view frustum (large)
      – Possibly one frustum for every image tile separately, when using *tiled rendering*
  – Ray Tracing:
    • Build (detailed) spatial index
    • Traverse with (infinitely thin) ray or with some (typically small) frustum
  – Both approaches can benefit greatly from spatial index!
Ray Tracing vs. Rasterization (II)

- **Binning (finding relevant pixels in a large frustum)**
  - Test to (hierarchically) find pixels likely covered by a primitive
  - Rasterization:
    - Great speedup due to very large view frustum (many pixels)
  - Ray tracing (frustum tracing)
    - Can speed up, depending on frustum size [Benthin'09]
- Ray Tracing (single/few rays)
  - Not needed

- **Conclusion**
  - Both algorithms can use the same in-triangle test
    - In 3D, requires floating point, but boils down to 2D computation
  - Both algorithms can benefit from spatial index
    - Benefit depends on relative cost of in-triangle test (HW vs. SW)
  - Both algorithms can benefit from 2D binning to find relevant samples
    - Benefit depends on ratio of covered/uncovered samples per frustum

- **Both approaches are very similar**
  - Different organization (size of frustum, binning)
  - There is no reason RT needs to be slower for primary rays (exc. FP)
Imagination-Grafikchip: 5 Mal schneller als GeForce GTX 980 Ti beim Raytracing

Fünf Mal schneller als eine GeForce GTX 980 Ti soll die Mobil-GPU PowerVR GR6500 sein, allerdings nur bei bestimmten Raytracing-Anwendungen.

Die Mobil-Grafikeinheit PowerVR GR6500 soll fünf Mal schneller arbeiten als Nvidias GeForce GTX 980 Ti bei nur einem Zehntel der Leistungsaufnahme; allerdings nur bei bestimmten Raytracing-Anwendungen.
AMD unveils three Radeon 6000 graphics cards with ray tracing and RTX-beating performance
The RX 6800, 6800 XT and 6900 XT are coming soon.

It’s time for BIG NAVI, as AMD has unveiled their new Radeon graphics cards: the $579 RX 6800, $649 RX 6800 XT and $999 RX 6900 XT. AMD claims that the cards should meet or beat Nvidia’s flagship RTX 30-series graphics cards, all the way up to the $1499 RTX 3090, often at lower price and while consuming less power. The 6000-series cards are also the first desktop AMD GPUs to support real-time ray tracing, variable rate shading and other DirectX 12 Ultimate features. All in all, it’s an exciting package for AMD fans - and would-be Nvidia users that might have become frustrated with poor RTX 30-series availability.

Intel’s new Xe GPU will have hardware-accelerated ray tracing

August 13, 2020 Intel has now confirmed that the Xe-HPG microarchitecture exists and that it will have ray tracing.