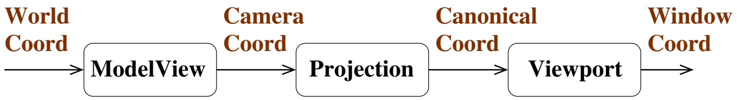
CSE251
Basics of Computer Graphics
Module: Rasterization Module

Avinash Sharma

Spring 2018

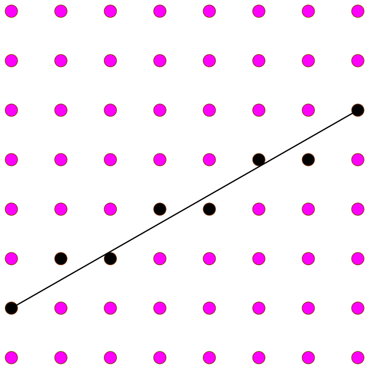
(Point) Pipeline in Action



- Points are transformed from Object to World to Canonical to Window coordinates.
- Each 3D point maps to a pixel (i, j) in the window space.
- Lines are made out of two points. Triangles and polygons are made out of 3 or more points.

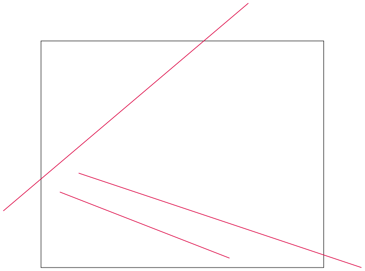
Lines in Action

- Lines are *rasterized* to the pixel grid of the window.
- Find pixels that lie closest to the line. Results in **aliasing**.
- Each pixel needs to be given a color and depth.



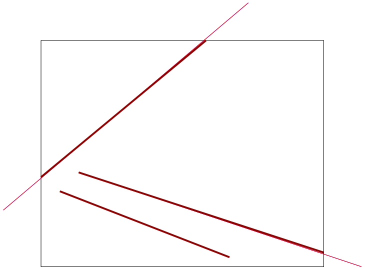
Clipping Lines

- End points map to window coordinates independently.
- World lines needn't map nicely onto points inside the window.



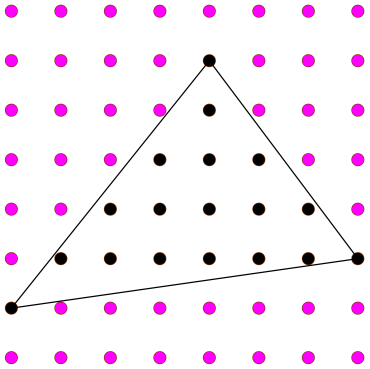
Clipping Lines

- End points map to window coordinates independently.
- World lines needn't map nicely onto points inside the window.
- **Clipping:** Finding part of the line that is *inside* the window.
- Clip first and then rasterize.



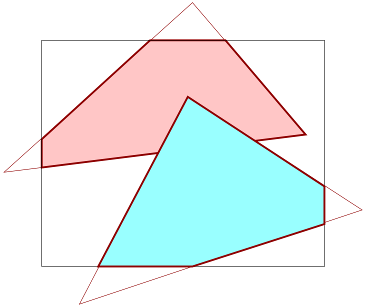
Triangles in Action

- Un-filled triangles are uninteresting. Filled ones represent surfaces.
- Triangles are **scan converted** or **rasterized** to include all pixels inside it.
- Each pixel needs to have a colour and depth.

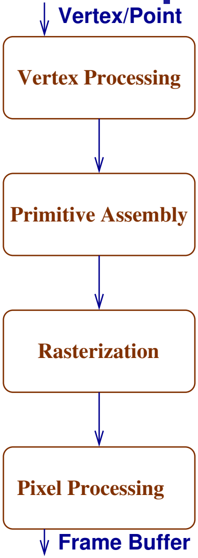


Clipping Triangles

- Only parts of the triangle may lie in the window.
- First clip a triangle to a (planar!) *polygon* that lies inside.
- Scan convert the polygon subsequently.



Primitive Pipeline



- From points, lines, triangles/polygons
- **Vertex** stage: process vertices independently
- Primitive stage: triangle assembly
- Rasterization: Clip & Determine the pixels inside the primitive
- **Pixel** stage: process each pixel independently

Linear Interpolation of Properties

- Each pixel needs: colour, depth, and texture coordinate.
- Assumption: Properties vary linearly across the plane.
- If we know the colour, texture coordinate, and depth at the vertices of the polygon (or line), these can be interpolated to pixels on the inside linearly!
- Colour: 3-vector, texture coord: 2-vector, depth: scalar.
- Rasterization step interpolates these values and gives to each pixel.
- Is the interpolation valid?

Vertex Processing

- Apply ModelView and Projection matrices to the vertex
- Find/send colour: either given or compute from physics!
- Find/send texture coordinates: usually given
- Find/send normals: usually given.
- Can process vertices of a primitive independently
- Modern GPUs: This stage is **programmable!** Can write own *vertex shader* to replace the standard processing.

Rasterization

- Apply viewport transformation.
- Clip primitive to the window or the viewport.
- Evaluate which pixels are part of the primitive.
- Interpolate values for each pixel and queue the pixels or *fragments* for further processing.
- This is computationally quite expensive and is usualy done by a dedicated hardware unit.
- A queue of fragments with associated data are built by this stage.

Pixel or Fragment Processing

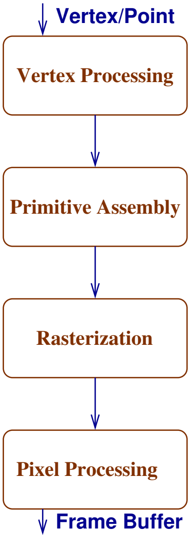
- The pixels generated by the rasterization stage are processed in arbitrary order by this stage.
- Depth value is available already. Can look up Z-buffer to keep or discard the fragment.
- Interpolated colour value can be sent to frame buffer.
- Texture image can be accessed using the texture coordinates. The final colour can be a combination of interpolated and texture colours.
- Modern GPUs: This stage is **programmable!** Can write own *fragment shader* to replace the standard processing.

Programmable GPUs

- Graphics Processing Units are *programmable* today
- Novel shading and lighting can be performed by writing appropriate vetex and pixel **shaders**, beyond OpenGL
- GPUs used parallel processing with 2-4 vertex and 32-64 pixel processing units, all working in parallel. Together, considerable computing power was in a GPU
- Clever idea: Use the power for other processing: matrix multiplication, FFT, sorting, image processing, etc.
- ***GPGPU***: General Processing on GPUs

Primitive Pipeline: Summary

- Basic primitives: Points, Lines, Triangles/Polygons.
- Each constructed fundamentally from points.
- Points map to pixels on screen. Primitives are assembled from points.
- Pipeline of operations on a primitive finds the pixels that are part of it. And performs a few operations on each pixel



Scan Conversion or Rasterization

- Primitives are defined using points, which have been mapped to the screen coordinates.
- In vector graphics, connect the points using a pen directly.
- In Raster Graphics, we create a discretized image of the whole screen onto the **frame buffer** first. The image is scanned automatically onto the display periodically.
- This step is called **Scan Conversion** or **Rasterization**.

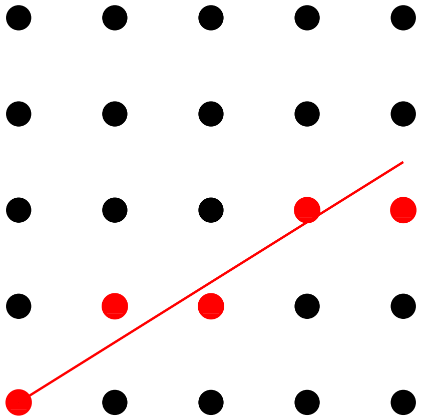
Scan Converting a Point

- The 3D point has been transformed to its screen coordinates (u, v).
- Round the coordinates to frame buffer array indices (i, j).
- Current colour is defined/known. Frame buffer array is initialized to the background colour.
- Perform: frameBuf[i, j] $\leftarrow$ currentColour
- Function WritePixel(i, j, colour) does the above.
- If *PointSize* > 1, assign the colour to a number of points in the neighbourhood!

Scan Converting a Line

- Identify the grid-points that lie on the line and colour them.
- Problem: Given two end-points on the grid, find the pixels on the line connecting them.
- Incremental algorithm or Digital Differential Analyzer (DDA) algorithm.
- Mid-Point Algorithm

Line on an Integer Grid



Incremental Algorithm

```
Function DrawLine($x_1, y_1, x_2, y_2$, colour)
    $\Delta x \leftarrow x_2 - x_1, \Delta y \leftarrow y_2 - y_1$, slope $\leftarrow \Delta y / \Delta x$
    $x \leftarrow x_1, y \leftarrow y_1$
    While ($x < x_2$)
        WritePixel ($x$, round($y$), colour)
        $x \leftarrow x + 1, \; y \leftarrow y +$ slope
    EndWhile
    WritePixel ($x_2, y_2$, colour)
EndFunction
```

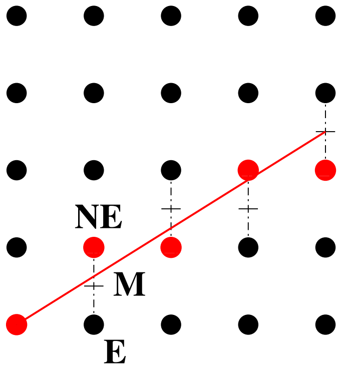
Incremental Algorithm With Integers

```
Function DrawLine($x_1, y_1, x_2, y_2$, colour)
    $\Delta x \leftarrow x_2 - x_1, \Delta y \leftarrow y_2 - y_1, sl \leftarrow 0, x \leftarrow x_1, y \leftarrow y_1$
    While ($x < x_2$)
        WritePixel ($x, y$, colour)
        $x \leftarrow x + 1$,  sl += $\Delta y$.
        if (sl $\geq \Delta x$) $\{y \leftarrow y + 1$, sl -= $\Delta x\}$
    EndWhile
    WritePixel ($x_2, y_2$, colour)
EndFunction
```

Points to Consider

- If abs(slope) > 1, step through y values, adding inverse slopes to x at each step.
- Simple algorithm, easy to implement.
- Floating point calculations were expensive once!
- Can we do with integer arithmetic only?
  Yes: **Bresenham's Algorithm, Mid-Point Line Algorithm**.

Two Options at Each Step!



Mid-Point Line Algorithm

- Line equation: $ax + by + c = 0, \quad a > 0$.
  Let $0 <$ slope $= \Delta y / \Delta x = -a/b < 1.0$
- $F(x, y) = ax + by + c > 0$ for below the line, < 0 for above.
- **NE** if $d = F(\mathbf{M}) > 0$; **E** if $d < 0$; else any!
- $d_{\mathbf{E}} = F(M_{\mathbf{E}}) = d + a, \qquad d_{\mathbf{NE}} = d + a + b$
- Therefore, $\Delta_{\mathbf{E}} = a, \qquad \Delta_{\mathbf{NE}} = a + b$
- Initial value: $d_0 = F(x_1 + 1, y_1 + \frac{1}{2}) = a + b\ /2$
- Similar analysis for other slopes. Eight cases in total.

Pseudocode

```
Function DrawLine ($x_1, y_1, x_2, y_2$, colour)
        $a \leftarrow (y_2 - y_1),\ b \leftarrow (x_1 - x_2),\ x \leftarrow x_1,\ y \leftarrow y_1$
        $d \leftarrow 2a + b,\ \Delta_E \leftarrow 2a,\ \Delta_{NE} \leftarrow 2(a + b)$
        While ($x < x_2$)
                WritePixel($x, y$, colour)
                if ($d < 0$)          // East
                        $d \leftarrow d + \Delta_E,\ x \leftarrow x + 1$
                else          // North-East
                        $d \leftarrow d + \Delta_{NE},\ x \leftarrow x + 1,\ y \leftarrow y + 1$
        EndWhile
        WritePixel($i, j$, colour)
EndFunction
```