Rasterization

MIT EECS 6.837
Frédo Durand and Barb Cutler

Final projects

- Rest of semester
  - Weekly meetings with TAs
  - Office hours on appointment
- This week, with TAs
  - Refine timeline
  - Define high-level architecture
- Project should be a whole, but subparts should be identified with regular merging of code

The Graphics Pipeline

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

Input:

Geometric model:
Description of all object, surface, and light source geometry and transformations
Lighting model:
Computational description of object and light properties, interaction (reflection)
Synthetic Viewpoint (or *Camera*):
Eye position and viewing frustum
Raster Viewport:
Pixel grid onto which image plane is mapped

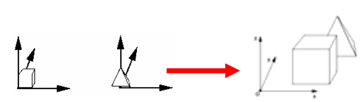
Output:

Colors/Intensities suitable for framebuffer display
(For example, 24-bit RGB value at each pixel)

Modeling Transformations

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- 3D models defined in their own coordinate system (object space)
- Modeling transforms orient the models within a common coordinate frame (world space)

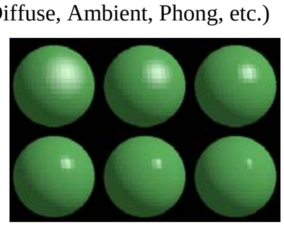


Object space World space

Illumination (Shading) (Lighting)

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

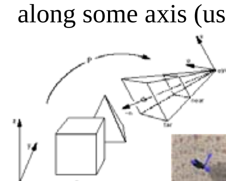
- Vertices lit (shaded) according to material properties, surface properties (normal) and light sources
- Local lighting model (Diffuse, Ambient, Phong, etc.)



Viewing Transformation

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Maps world space to eye space
- Viewing position is transformed to origin & direction is oriented along some axis (usually *z*)



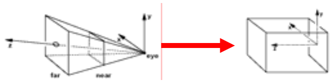
Eye space

World space

Clipping

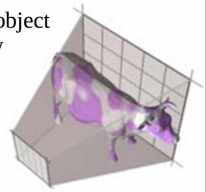
- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Transform to Normalized Device Coordinates (NDC)



Eye space NDC

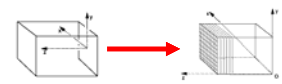
- Portions of the object outside the view volume (view frustum) are removed



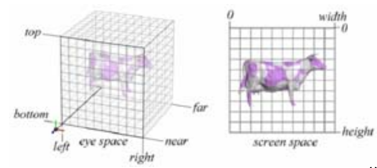
Projection

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- The objects are projected to the 2D image place (screen space)



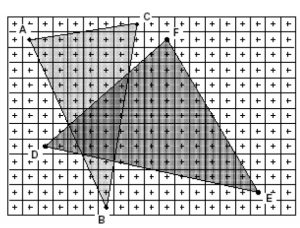
NDC Screen Space



Scan Conversion (Rasterization)

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Rasterizes objects into pixels
- Interpolate values as we go (color, depth, etc.)



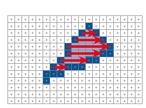
Visibility / Display

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

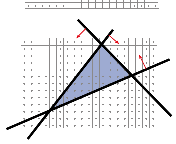
- Each pixel remembers the closest object (depth buffer)
- Almost every step in the graphics pipeline involves a change of coordinate system. Transformations are central to understanding 3D computer graphics.

Today

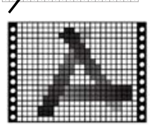
- Polygon scan conversion
  - smart



  - back to brute force

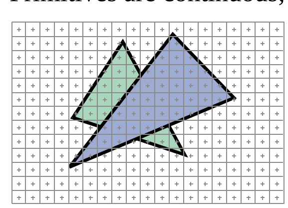


- Visibility



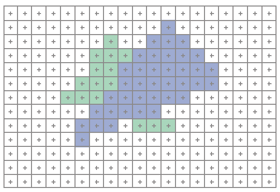
2D Scan Conversion

- Geometric primitive
  - 2D: point, line, polygon, circle...
  - 3D: point, line, polyhedron, sphere...
- Primitives are continuous; screen is discrete



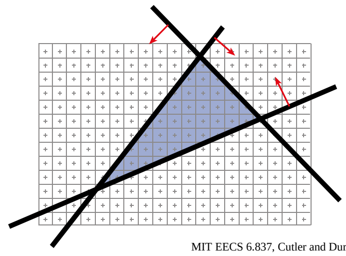
2D Scan Conversion

- Solution: compute discrete approximation
- Scan Conversion:
  algorithms for efficient generation of the samples comprising this approximation



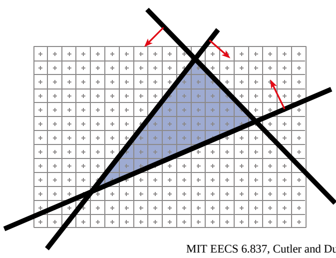
Brute force solution for triangles

- ?



Brute force solution for triangles

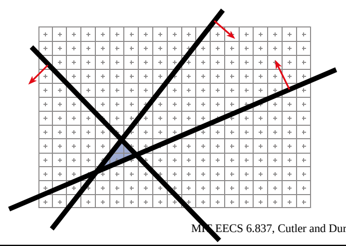
- For each pixel
  - Compute line equations at pixel center
  - “clip” against the triangle



Problem?

Brute force solution for triangles

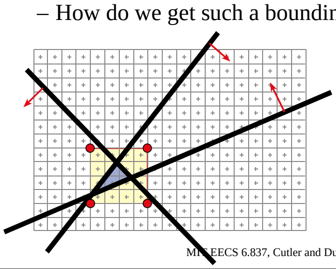
- For each pixel
  - Compute line equations at pixel center
  - “clip” against the triangle



Problem?
If the triangle is small, a lot of useless computation

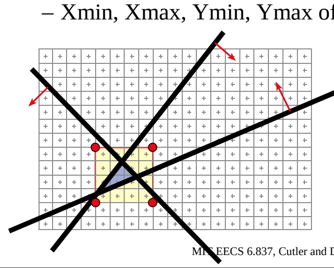
Brute force solution for triangles

- Improvement:
  - Compute only for the screen **bounding box** of the triangle
  - How do we get such a bounding box?



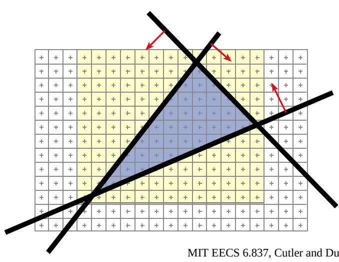
Brute force solution for triangles

- Improvement:
  - Compute only for the screen **bounding box** of the triangle
  - Xmin, Xmax, Ymin, Ymax of the triangle vertices



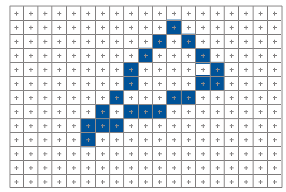
Can we do better? Kind of!

- We compute the line equation for many useless pixels
- What could we do?



Use line rasterization

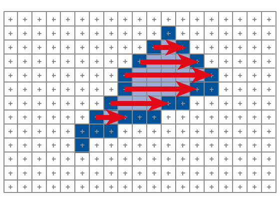
- Compute the boundary pixels



Shirley page 55

Scan-line Rasterization

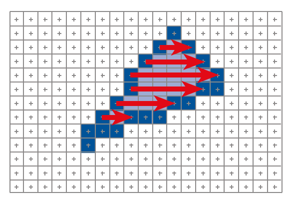
- Compute the boundary pixels
- Fill the spans



Shirley page 55

Scan-line Rasterization

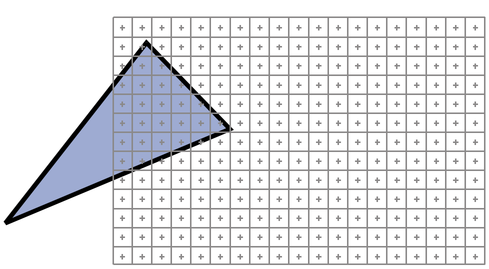
- Requires some initial setup to prepare



Shirley page 55

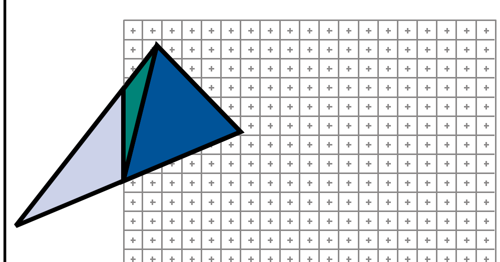
Clipping problem

- How do we clip parts outside window?



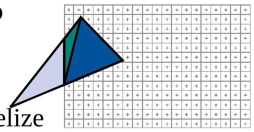
Clipping problem

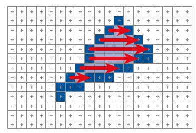
- How do we clip parts outside window?
- Create two triangles or more. Quite annoying.



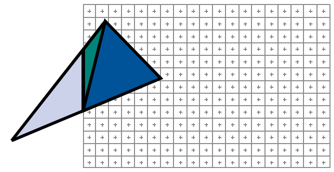
Old style graphics hardware

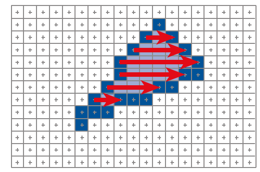
- Triangles were big
- Bresenham+interpolation is worth it
- Annoying clipping step
- + a couple of other things have changed
- Not that good to parallelize beyond triangle granularity





Questions?

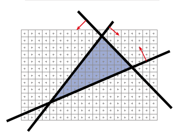




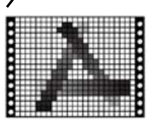
Today

- Polygon scan conversion

  - smart
  - back to brute force

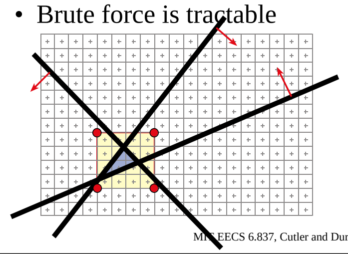


- Visibility



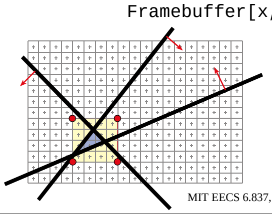
For modern graphics cards

- Triangles are usually very small
- Setup cost are becoming more troublesome
- Clipping is annoying
- Brute force is tractable



Modern rasterization

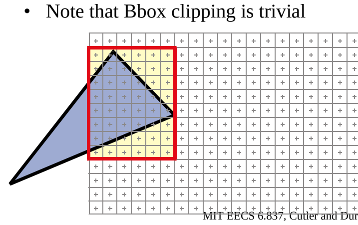
```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Compute line equations
      If all line equations>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```



Modern rasterization

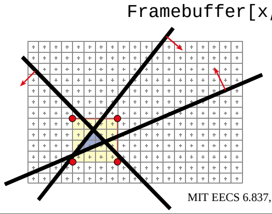
```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Compute line equations
      If all line equations>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```

- Note that Bbox clipping is trivial



Can we do better?

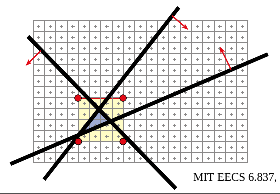
```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Compute line equations
      If all line equations>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```



Can we do better?

```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Compute line equations ax+by+c
      If all line equations>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```

- We don't need to recompute line equation from scratch



Can we do better?

```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   Setup line eq
      compute a_i dx, b_i dy for the 3 lines
      Initialize line eq, values for bbox corner
         L_i=a_i x0+b_i y+c_i
   For all scanline y in bbox
      For 3 lines, update Li
      For all x in bbox
         Increment line equations: Li+=adx
      If all Li>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```

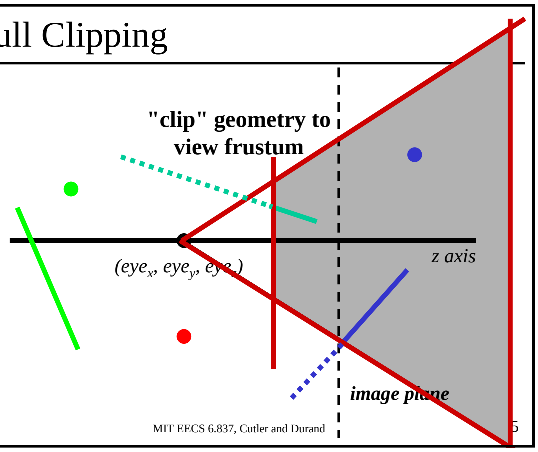
- We save one multiplication per pixel

The Graphics Pipeline

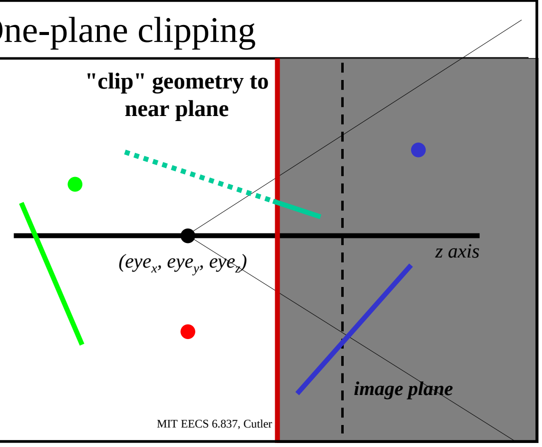
- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Modern hardware mostly avoids clipping
- Only with respect to plane z=0

Full Clipping

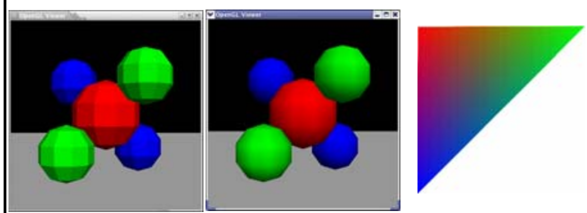


One-plane clipping



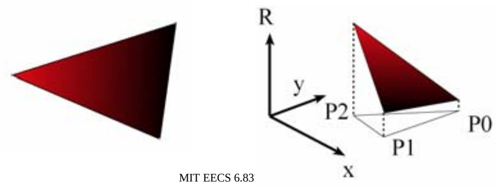
Adding Gouraud shading

- Interpolate colors of the 3 vertices
- Linear interpolation



Adding Gouraud shading

- Interpolate colors of the 3 vertices
- Linear interpolation, e.g. for R channel:
  - $R=a_Rx+b_Ry+c_R$
  - Such that R[x0,y0]=R0; R[x1, y1]=R1; R[x2,y2]=R2
  - Same as a plane equation in (x,y,R)

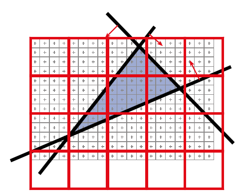


Adding Gouraud shading

```
Interpolate colors
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   Setup line eq
   Setup color equation
   For all pixels in bbox
      Increment line equations
      Increment color equation
      If all Li>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=interpolatedColor
```


In the modern hardware

- Edge eq. in homogeneous coordinates [x, y, w]
- Tiles to add a mid-level granularity
  - early rejection of tiles
  - Memory access coherence



Ref

- Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan Spach, John Austin, Frederick Brooks, Jr., John Eyles and John Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes", Proceedings of SIGGRAPH '85 (San Francisco, CA, July 22–26, 1985). In *Computer Graphics*, v19n3 (July 1985), ACM SIGGRAPH, New York, NY, 1985.
- Juan Pineda, "A Parallel Algorithm for Polygon Rasterization", Proceedings of SIGGRAPH '88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, v22n4 (August 1988), ACM SIGGRAPH, New York, NY, 1988. Figure 7: Image from the spinning teapot performance test.
- Triangle Scan Conversion using 2D Homogeneous Coordinates, Marc Olano Trey Greer http://www.cs.unc.edu/~olano/papers/2dh-tri/2dh-tri.pdf

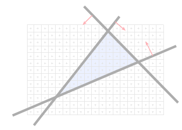
Take-home message

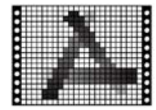
- The appropriate algorithm depends on
  - Balance between various resources (CPU, memory, bandwidth)
  - The input (size of triangles, etc.)
- Smart algorithms often have initial preprocess
  - Assess whether it is worth it
- To save time, identify redundant computation
  - Put outside the loop and interpolate if needed

Questions?

Today

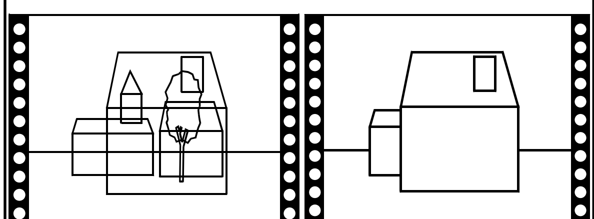
- Polygon scan conversion
  - smart
  - back to brute force
- Visibility





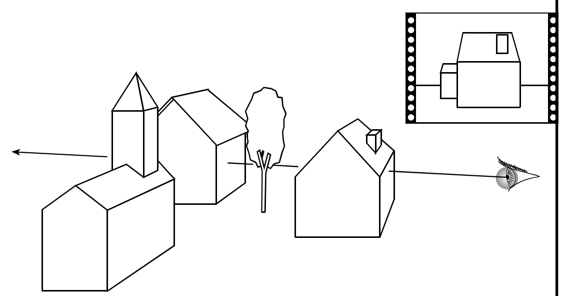
Visibility

- How do we know which parts are visible/in front?



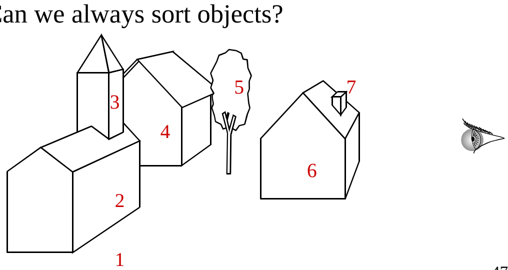
Ray Casting

- Maintain intersection with closest object



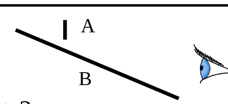
Painter's algorithm

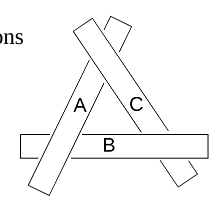
- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?



Painter's algorithm

- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?
  - No, there can be cycles
  - Requires to split polygons



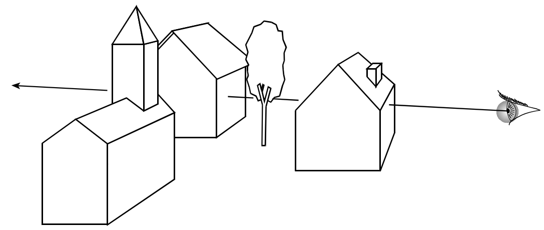


Painter's algorithm

- Old solution for hidden-surface removal
  - Good because ordering is useful for other operations (transparency, antialiasing)
- But
  - Ordering is tough
  - Cycles
  - Must be done by CPU
- Hardly used now
- But some sort of partial ordering is sometimes useful
  - Usuall front-to-back
  - To make sure foreground is rendered first

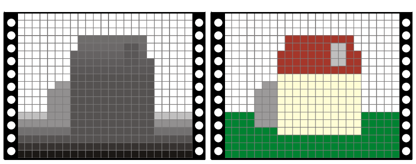
visibility

- In ray casting, use intersection with closest t
- Now we have swapped the loops (pixel, object)
- How do we do?



Z buffer

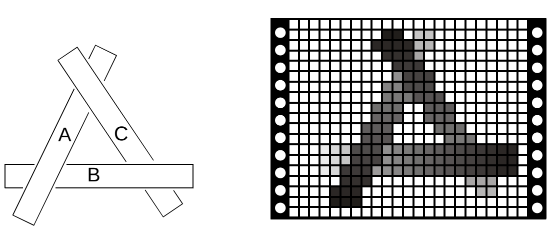
- In addition to frame buffer (R, G, B)
- Store distance to camera (z-buffer)
- Pixel is updated only if new z is closer than z-buffer value



Z-buffer pseudo code

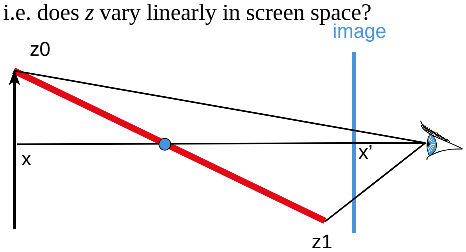
```
For every triangle
   Compute Projection, color at vertices
   Setup line equations
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Increment line equations
      Compute curentZ
      Increment currentColor
      If all line equations>0 //pixel [x,y] in triangle
         If currentZ<zBuffer[x,y] //pixel is visible
            Framebuffer[x,y]=currentColor
            zBuffer[x,y]=currentZ
```


Works for hard cases!



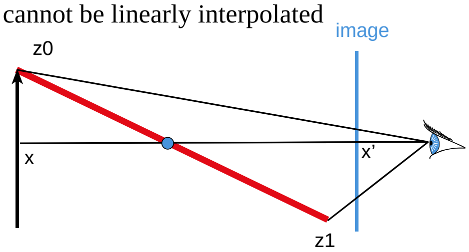
What exactly do we store

- Floating point distance
- Can we interpolate *z* in screen space?
  - i.e. does *z* vary linearly in screen space?



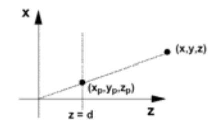
Z interpolation

- X'=x/z
- Hyperbolic variation
- Z cannot be linearly interpolated



Simple Perspective Projection

- Project all points along the *z* axis to the *z* = *d* plane, eyepoint at the origin

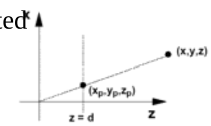


homogenize

$$\begin{pmatrix} x*d/z \\ y*d/z \\ d \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Yet another Perspective Projection

- Change the z component
- Compute d/z
- Can be linearly interpolated



homogenize

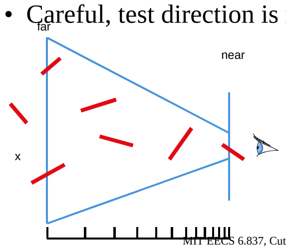
$$\begin{pmatrix} x*d/z \\ y*d/z \\ d/z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \\ z/d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Advantages of 1/z

- Can be interpolated linearly in screen space
- Puts more precision for close objects
- Useful when using integers
  - more precision where perceptible

Integer z-buffer

- Use 1/z to have more precision in the foreground
- Set a near and far plane
  - 1/z values linearly encoded between 1/near and 1/far
- Careful, test direction is reversed

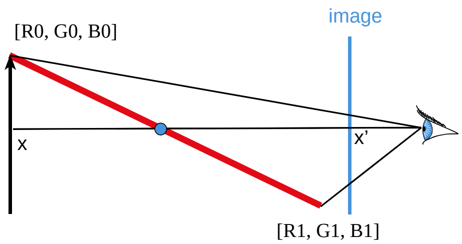


Integer Z-buffer pseudo code

```
For every triangle
   Compute Projection, color at vertices
   Setup line equations, depth equation
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Increment line equations
      Increment curent_1ovZ
      Increment currentColor
      If all line equations>0 //pixel [x,y] in triangle
         If current_1ovZ>1ovzBuffer[x,y]//pixel is visible
            Framebuffer[x,y]=currentColor
            1ovzBuffer[x,y]=current1ovZ
```

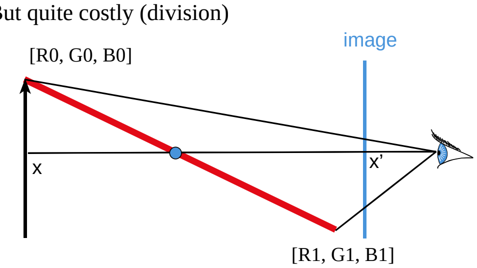
Gouraud interpolation

- Gouraud: interpolate color linearly in screen space
- Is it correct?



Gouraud interpolation

- Gouraud: interpolate color linearly in screen space
- Not correct. We should use hyperbolic interpolation
- But quite costly (division)



Next time

- Clipping
- Segment intersection & scanline algorithms