CSC 433/533 Computer Graphics

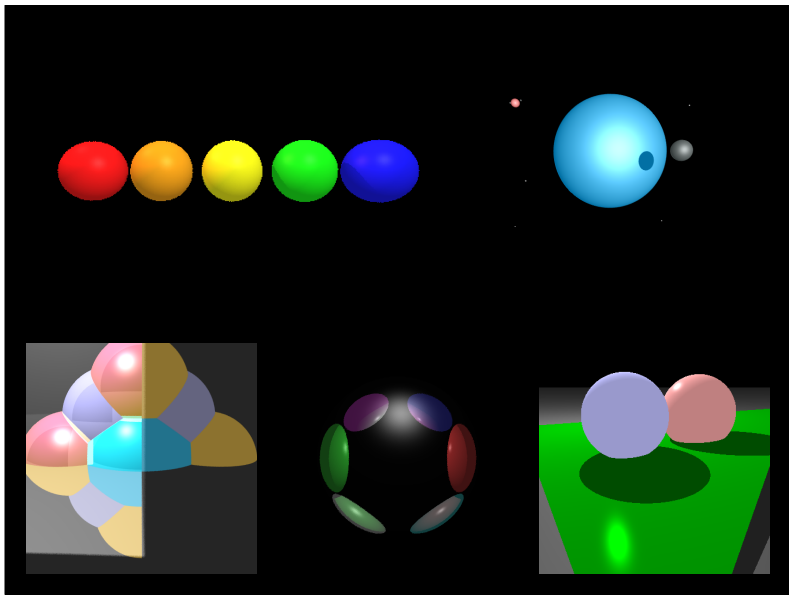
Joshua Levine
josh@email.arizona.edu

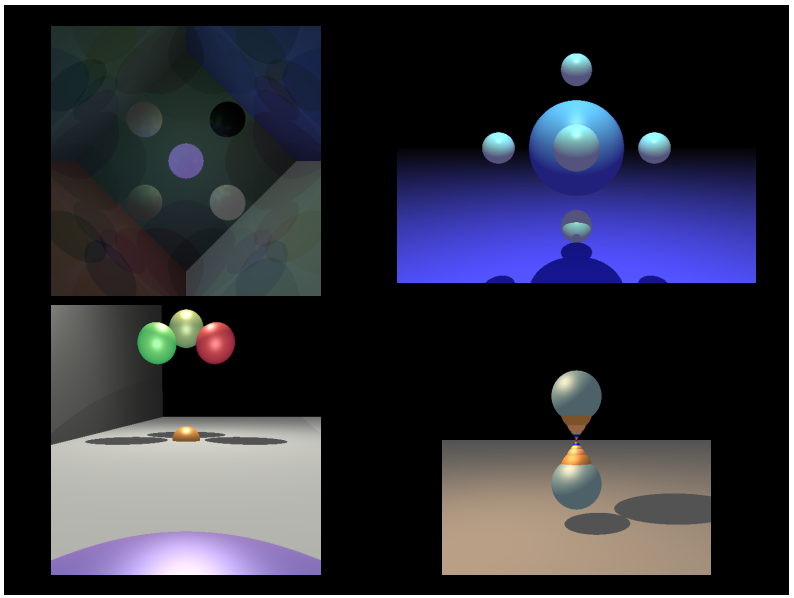
Lecture 20 Graphics Pipeline 1

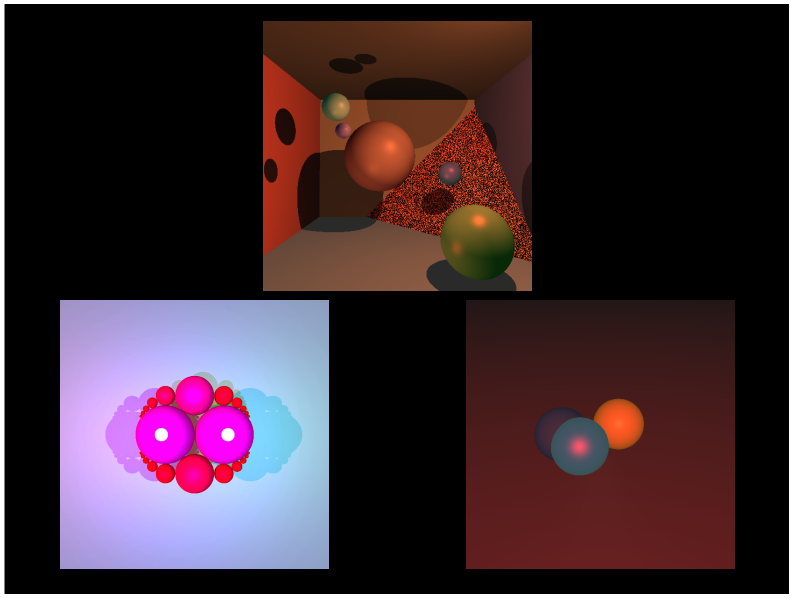
Today's Agenda

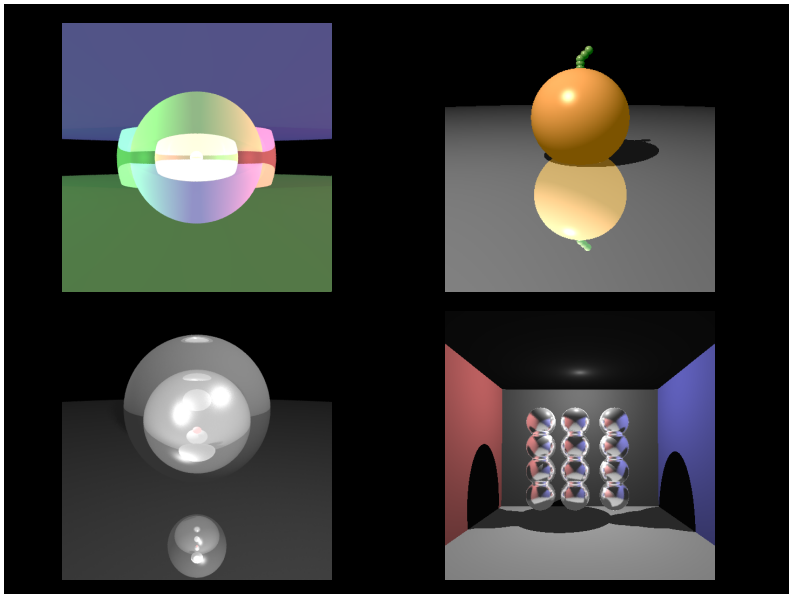
- Goals for today: discuss rasterization and clipping in the graphics pipeline

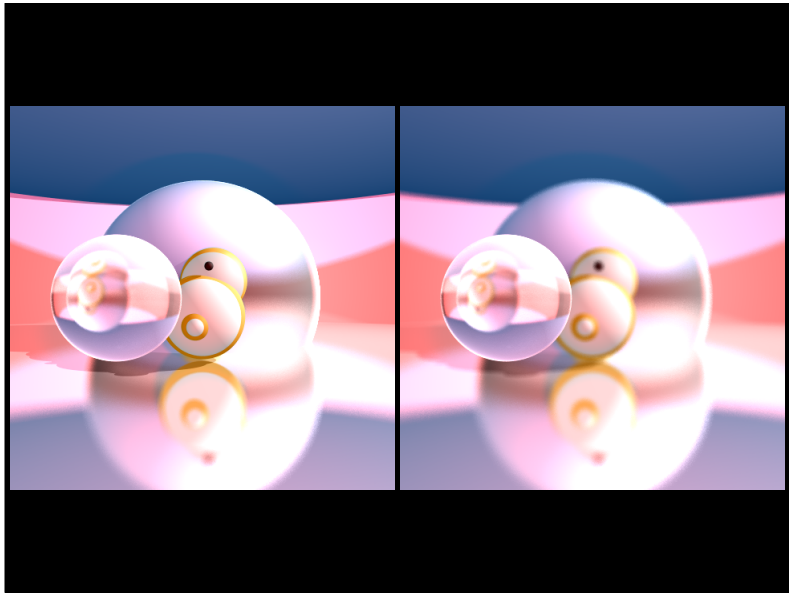
A04 Scenes

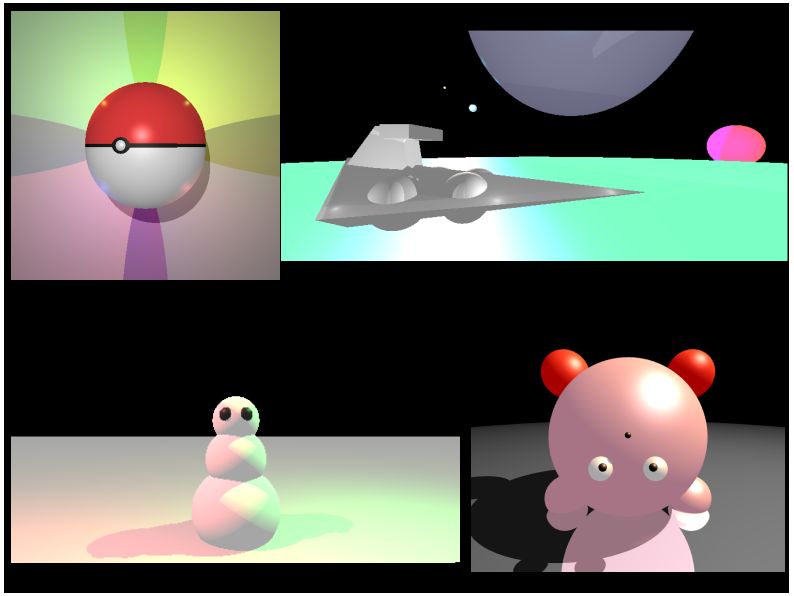






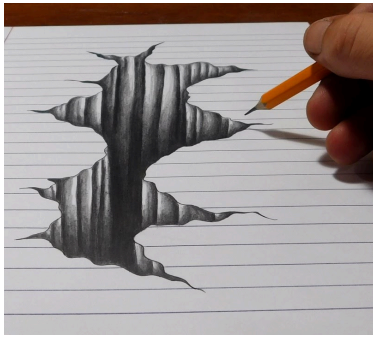






Recall: Two Ways to Think About How We Make Images

- Drawing



- Photography

Recall: Two Ways to Think About Rendering

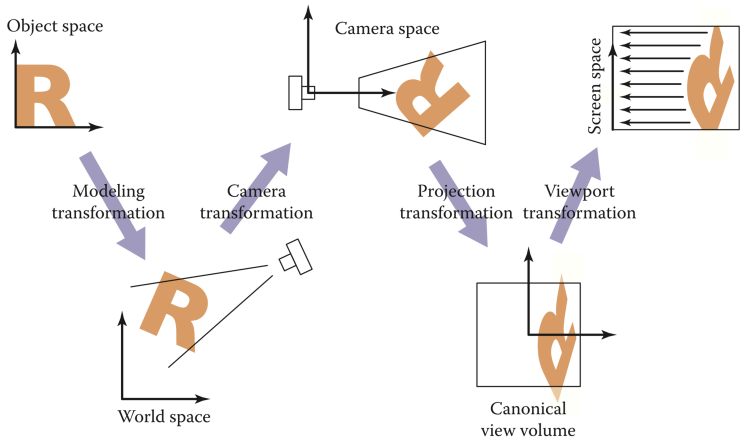
- Object-Ordered
- Decide, for every object in the scene, its contribution to the image

TODAY

- Image-Ordered
- Decide, for every pixel in the image, its contribution from every object

Recall: Step-by-Step Viewing Transformations

(Each arrow is a matrix)



Recall: Perspective Distortion

- Perspective matrix effect on coordinates is nonlinear distortion in z:

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- But it does, however, preserve order in the z-coordinate (which will become useful very soon)

Recall: Putting it all together

```
construct M_vp
construct M_per
construct M_cam
M = M_vp * M_per * M_cam
for each 3D object O {
  O_screen = M * O
  draw(O_screen)
}
```

Equivalently:

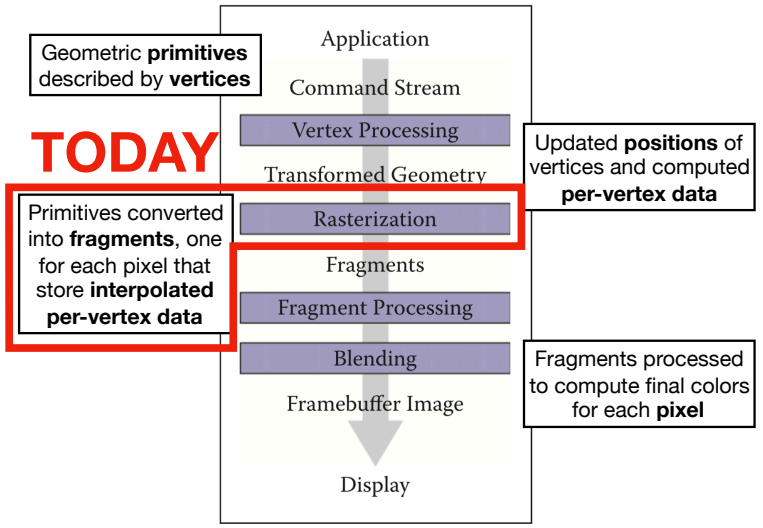
$$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{P}\mathbf{M}_{\text{cam}}$$

For a given vertex $\mathbf{a} = (x,y,z)$, $\mathbf{p} = \mathbf{M}\mathbf{a}$ should result in drawing $(x_\mathbf{p}/w_\mathbf{p}, y_\mathbf{p}/w_\mathbf{p}, z_\mathbf{p}/w_\mathbf{p})$ on the screen

The Graphics Pipeline

The Graphics Pipeline

- The sequence of operations that we'll perform to transform from objects in 3D to color pixels on the screen.
  - A "pipeline" because there are many stages
- All variants of object-ordered rendering follow a similar sequence to what we'll discuss
- Generally, there are plenty of opportunities for optimization in this and it can be much faster than raytracing with the right hardware



Geometric Primitives

- Points, lines, and triangles **only**.
- What about?
  - Curves/Text? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved surfaces? Approximate them with triangles
- Using minimal primitives keeps the pipeline simple and uniform for efficiency.

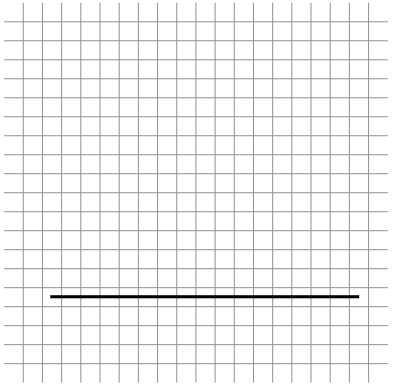
Rasterization

Two Jobs of a Rasterizer

- To *enumerate* the pixels that are covered by each primitive
- To *interpolate* the values/attributes across the primitives, e.g. normals, colors, texture coordinates
- Output: a set of **fragments**, one for each pixel.
  - Fragments live at a pixel and store the interpolated attributes.

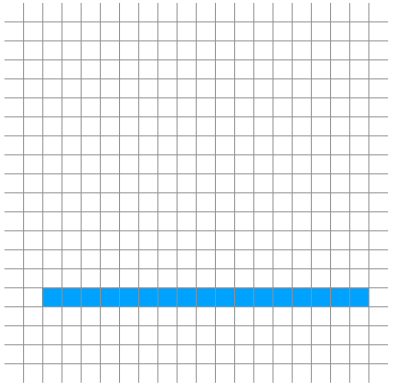
Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle



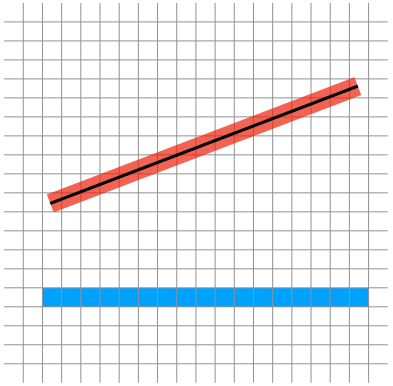
Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle



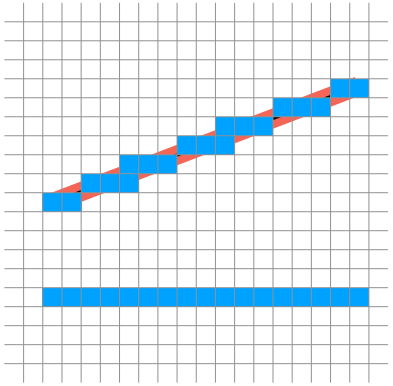
Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle
- Decide which pixels are turned on if their center is inside the line



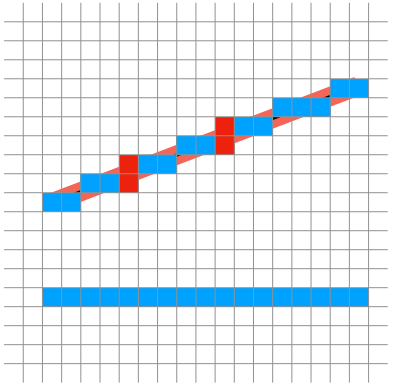
Rasterizing Lines to Fragments w/ Point Sampling

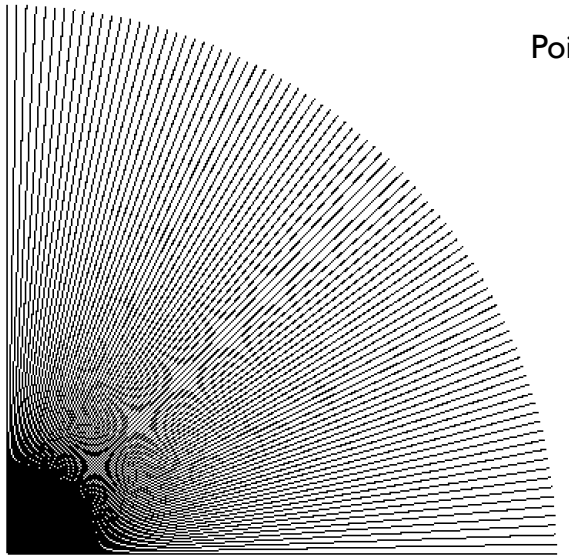
- Option 1: Fatten the line and decide which pixels hit the rectangle
- Decide which pixels are turned on if their center is inside the line



Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle
- Decide which pixels are turned on if their center is inside the line
- Problem: line width is not kept constant

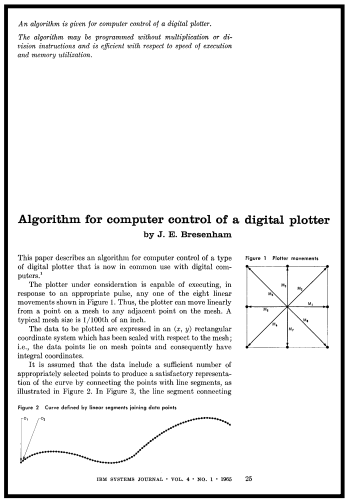




Point sampling in action

Bresenham's Algorithm

- Foundational algorithm in computer graphics while at IBM from
- Published in 1965



An algorithm is given for computer control of a digital plotter.

The algorithm may be programmed without multiplication or division instructions and is efficient with respect to speed of execution and memory utilization.

Algorithm for computer control of a digital plotter

by J. E. Bresenham

This paper describes an algorithm for computer control of a type of digital plotter that is now in common use with digital computers.[1]

The plotter under consideration is capable of executing, in response to an appropriate pulse, any one of the eight linear movements shown in Figure 1. Thus, the plotter can move linearly from a point on a mesh to any adjacent point on the mesh. A typical mesh size is 1/100th of an inch.

The data to be plotted are expressed in an (x, y) rectangular coordinate system which has been scaled with respect to the mesh; i.e., the data points lie on mesh points and consequently have integral coordinates.

It is assumed that the data include a sufficient number of appropriately selected points to produce a satisfactory representation of the curve by connecting the points with line segments, as illustrated in Figure 2. In Figure 3, the line segment connecting

Figure 1 Plotter movements

Figure 2 Curve defined by linear segments joining data points

IBM SYSTEMS JOURNAL • VOL. 4 • NO. 1 • 1965 25

Equations for a line

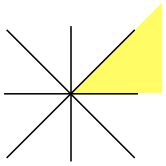
- Given points (x_0, y_0) and (x_1, y_1), the implicit equation for the line is:

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0 = 0$$

- We'll consider lines only defined in one octant: $x_0 \leq x_1$ (otherwise, swap the points) and that the slope of the line, $m \in (0,1]$ where

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

- But the same technique can be used for any octant

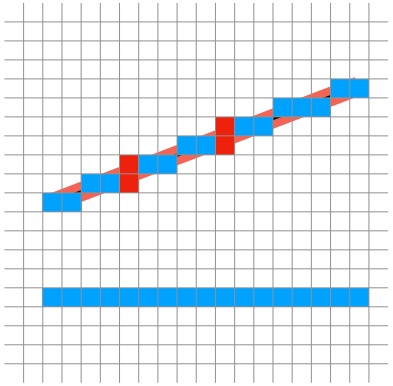


Bresenham's Approach

- Key idea: only turn on one pixel per column
- Multiple variants of this algorithm, we'll discuss the **midpoint algorithm**

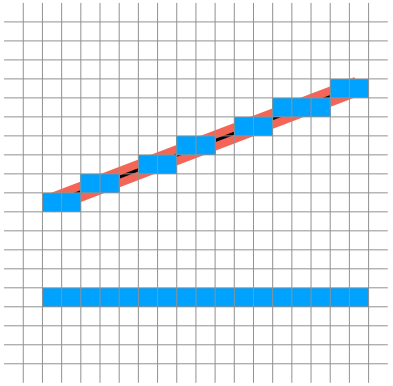
Resolving Sampling Issues with the Midpoint Algorithm

- Turn on only the pixel closest to the line in each column



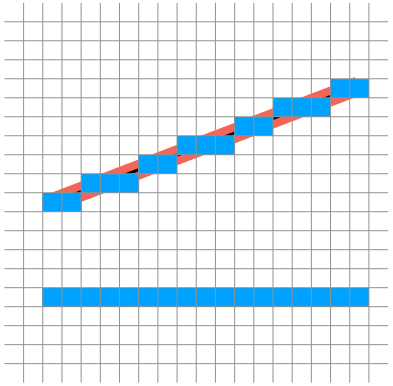
Resolving Sampling Issues with the Midpoint Algorithm

- Turn on only the pixel closest to the line in each column

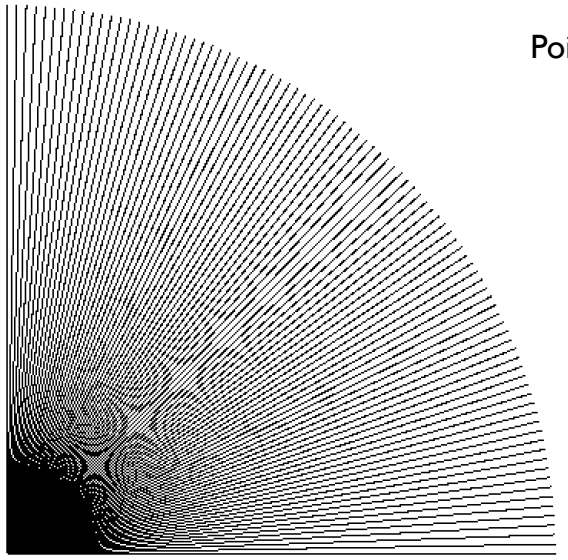


Resolving Sampling Issues with the Midpoint Algorithm

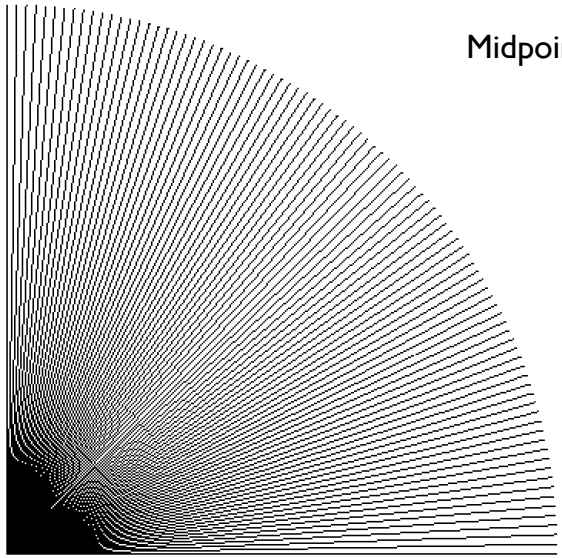
- Turn on only the pixel closest to the line in each column



Point sampling in action

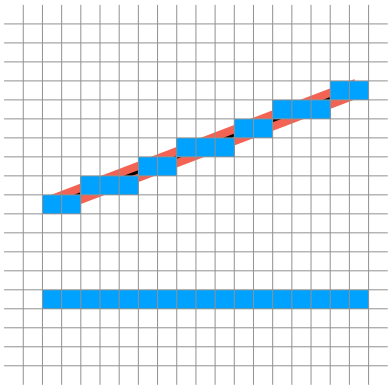


Midpoint algorithm in action

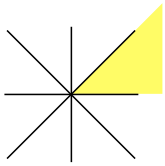


Resolving Sampling Issues with the Midpoint Algorithm

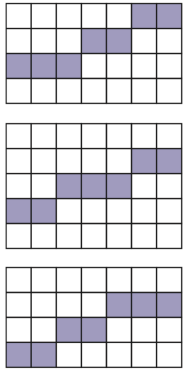
- Turn on only the pixel closest to the line in each column
- But then, why use the fattened rectangle at all?



The Midpoint Algorithm



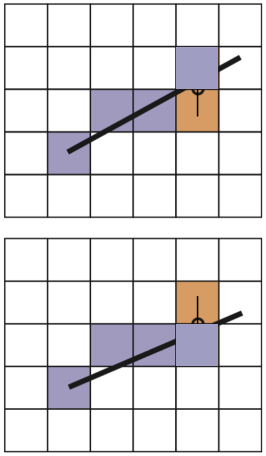
- For the case $m \in (0,1]$ there is more "run" (horizontal) than "rise" (vertical)
- We can draw any reasonable line that is 1 pixel thick by stepping horizontally for a while, and then deciding if we should move up to a new row
- Suggests that we should do:



```
y = y0;
for (x in [x0,x1]) {
  draw(x,y);
  if (some condition) {
    y = y+1;
  }
}
```

The Midpoint Algorithm

- Idea: Compare the midpoint of the line segment connecting the next two pixels to the line
  - If line is below the midpoint, then we have moved up a row
  - Otherwise, we stay on the same row:



```
y = y0;
for (x in [x0,x1]) {
  draw(x,y);
  if (f(x+1,y+0.5) < 0) {
    y = y+1;
  }
}
```


Incremental Midpoint

- Instead of repeated evaluating f(x,y), we can only evaluate f(x+1,y+0.5) once and then incrementally update by observing that:

$$f(x+1,y) = f(x,y) + (y_0 - y_1)$$
$$f(x+1,y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$$

```
y = y0;
d = f(x0+1,y0+0.5);
for (x in [x0,x1]) {
  draw(x,y);
  if (d < 0) {
    y = y+1;
    d = d + (x1-x0) + (y0-y1);
  } else {
    d = d + (y0-y1);
  }
}
```

Considerations w/ Rasterizing Triangles to Fragments

- Want to be able to interpolate values like colors, texture coordinates
- Have to handle the situation where two triangles share an edge
- A region-based approach based on deciding if pixel centers are inside triangles works well.

Rasterizing Triangles to Fragments

- How to determine if a pixel center is inside of the triangle?
- This is straightforward with barycentric coordinates

```
for all x {
  for all y {
  compute (α, β, γ) for (x,y)
  if (α ∈ [0,1] and β ∈ [0,1] and γ ∈ [0,1]) {
    draw(x,y);
  }
}
```

Recall: Barycentric Coordinates

- A coordinate system to write all points $\mathbf{p}$ as a weighted sum of the vertices

  $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$

  $\alpha + \beta + \gamma = 1$

- Equivalently, α, β, γ are the proportions of area of subtriangles relative total area, A

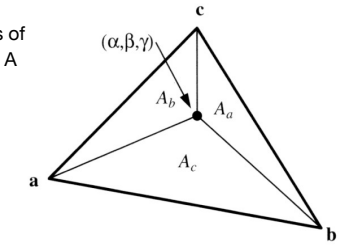
  $A_a / A = \alpha$

  $A_b / A = \beta$

  $A_c / A = \gamma$

- Triangle interior test:

  $\alpha > 0$, $\beta > 0$, and $\gamma > 0$



[Shirley 2000]

Computing Barycentric Coordinates for Pixels

- We can rely on the equations for each edge of the triangle. Given points (x_0,y_0), (x_1,y_1), and (x_2,y_2) on a triangle, we know:

$$f_{01}(x,y) = (y_o - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0$$
$$f_{12}(x,y) = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1$$
$$f_{20}(x,y) = (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2$$

- And we can write the barycentric coordinate as:

$$\alpha = f_{12}(x,y) \,/\, f_{12}(x_0,y_0)$$
$$\beta = f_{20}(x,y) \,/\, f_{20}(x_1,y_1)$$
$$\gamma = f_{01}(x,y) \,/\, f_{01}(x_2,y_2)$$

Acceleration: #1

- Only check x,y that are in the bounding box of the triangle

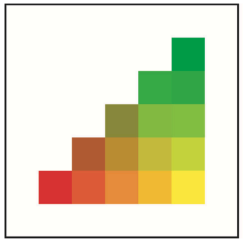
```
xmin = floor(min({x_0,x_1,x_2}))
xmax = ceiling(max({x_0,x_1,x_2}))
ymin = floor(min({y_0,y_1,y_2}))
ymax = ceiling(max({y_0,y_1,y_2}))

for all x in [xmin,xmax] {
  for all y in [ymin,ymax] {
  compute (α, β, γ) for (x,y)
  if (α ∈ [0,1] and β ∈ [0,1] and γ ∈ [0,1]) {
    draw(x,y);
  }
}
```

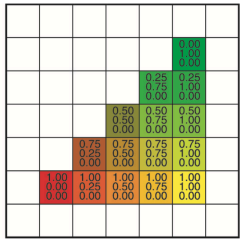
Interpolating Color w/ Gouraud Shading

- Having the barycentric coordinates means that we can interpolate color:

$$\mathbf{c}=\alpha\mathbf{c}_0+\beta\mathbf{c}_1+\gamma\mathbf{c}_2$$

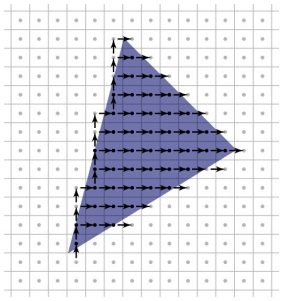
```
for all x in [xmin,xmax] {
  for all y in [ymin,ymax] {
  compute (α, β, γ) for (x,y)
  if (α,β,γ ∈ [0,1]) {
    c = α*c0 + β*c1 + γ*c2
    draw(x,y);
  }
}
```

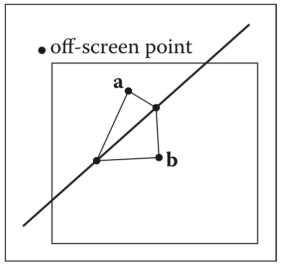




Some Additional Considerations

- Can we accelerate rendering using some variant of incremental rendering?
  - Yes: But we will still need to interpolate colors
- How to break the tie when a pixel center falls on a shared edge?
  - Lots of schemes, but one is use an offscreen point and prefer the triangle whose opposite corner is closer





Clipping

Should We Rasterize All Transformed Primitives?

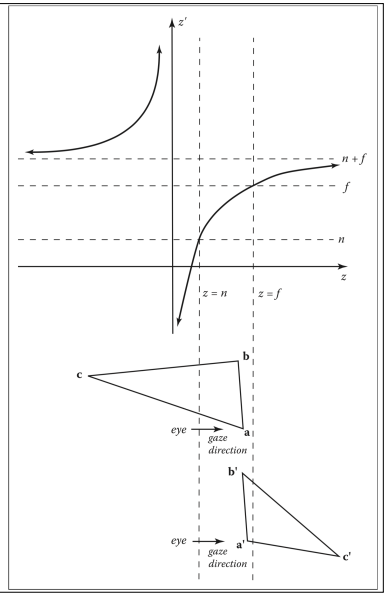
- Primitives outside of the view volume, particularly those behind the eye, will be projected by the projection transformation to nonsensical locations

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \sim \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix}$$

The homogeneous divide makes this a hyperbolic equation

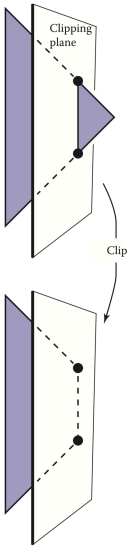
Nonsensical Transformations

- z values are transformed to z' values by projective transform
- Effect is to move vertex $\mathbf{c}$ from behind eye to vertex $\mathbf{c}'$ in front of eye



Solution: Clip Triangles Against View Volume

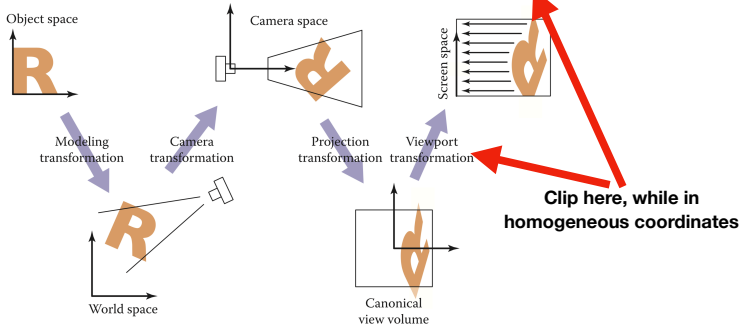
- Before rasterization, we explicitly check that the triangle is contained with the view volume
- View volume is defined by six planes, so we check for intersections with all of them, retaining the portion of the triangle that is inside



When to Clip?

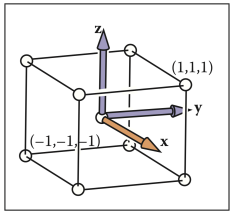
- Technically, any time before the homogeneous divide is reasonable, but it helps to use the simplest planes.
- Frequently, this is done with the canonical view volume

$$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{P}\mathbf{M}_{\text{cam}}$$



Clipping Planes

- Because we have not done the homogeneous divide, the view volume is actually a four dimensional object
- Math is a little bit tedious, but these equations use l = b = n = -1 and r = t = f = 1
- Each equation defines a side of the box in the form $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$, or equivalently: $f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + \mathbf{D} = 0$



$$\begin{aligned} -x + lw &= 0 \\ x - rw &= 0 \\ -y + bw &= 0 \\ y - tw &= 0 \\ -z + nw &= 0 \\ z - fw &= 0 \end{aligned}$$

Clipping a Triangle Against a Plane

- Basic idea: Check if any of three line segments cross the plane.
- How? Solve for intersection points:

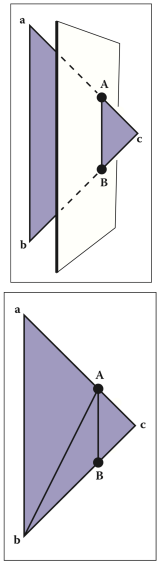
$$f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + \mathbf{D} = 0$$

- By plugging in:

$$\mathbf{p} = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$$

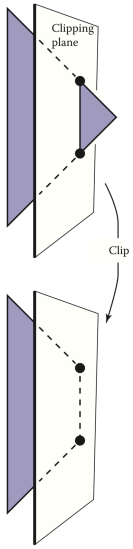
$$t = \frac{\mathbf{n} \cdot \mathbf{a} + D}{\mathbf{n} \cdot (\mathbf{a} - \mathbf{b})}$$

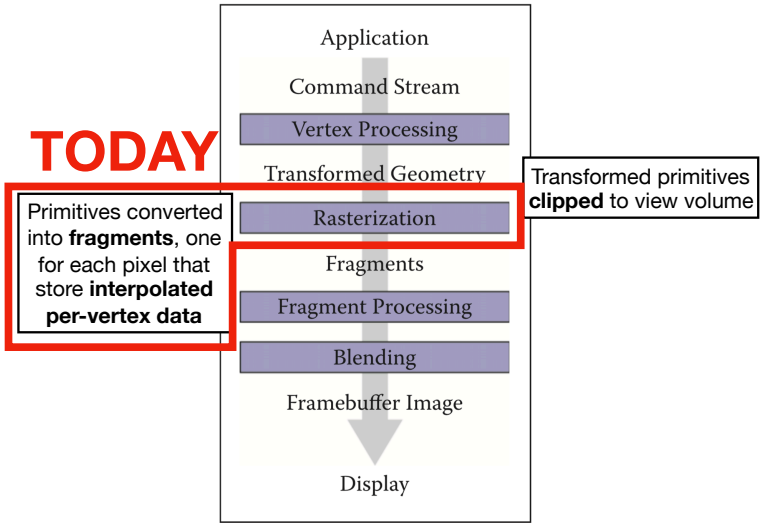
Note: $\mathbf{a}$, $\mathbf{b}$ are both 4D homogeneous points and $\mathbf{n}$ is a 4D vector



Clipping Algorithm

```
for (each triangle t) {
  for (each plane p of view volume) {
    if (t entirely outside of p) {
      discard t.
    } else if (t spans p) {
      q = clip(t).
      if (q is a quadrilateral) {
        replace t with two triangles.
      } else {
        replace t with q.
      }
    }
  }
}
```





Lec21 Required Reading

- FOCG, Ch. 8.2