

**ECE/CS 412**

# **Introduction to Computer Graphics**

Class 14

Pradeep Sen  
Advanced Graphics Lab



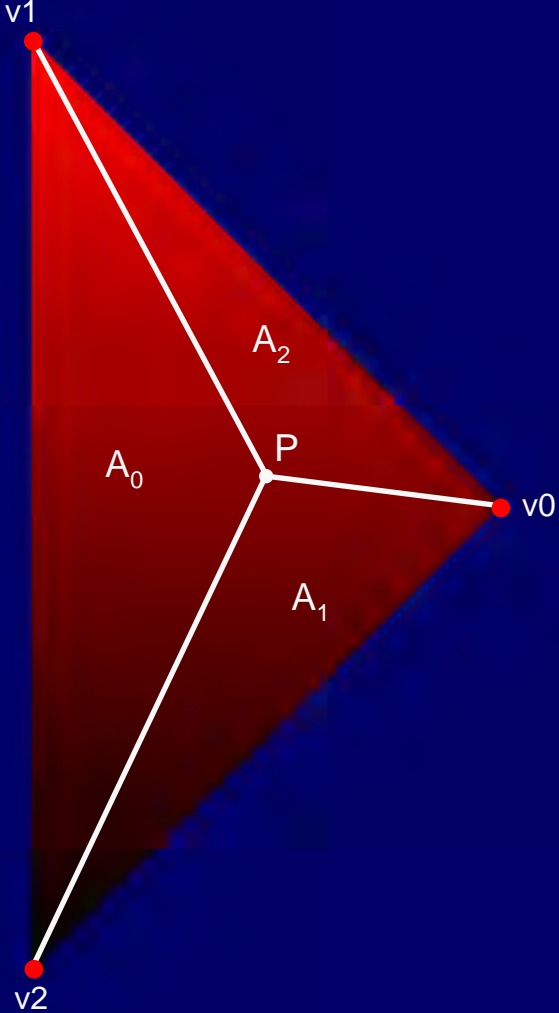
# Announcements

---

- HW4 will be posted soon.



# Barycentric coordinates



$P = (A_0/A) v_0 + (A_1/A) v_1 + (A_2/A) v_2$

$P = \alpha_0 v_0 + \alpha_1 v_1 + \alpha_2 v_2$

$\alpha_0 + \alpha_1 + \alpha_2 = 1$

$(A_0/A) + (A_1/A) + (A_2/A) = 1$

Remember: this is done in screen space!

# Interpolants

---

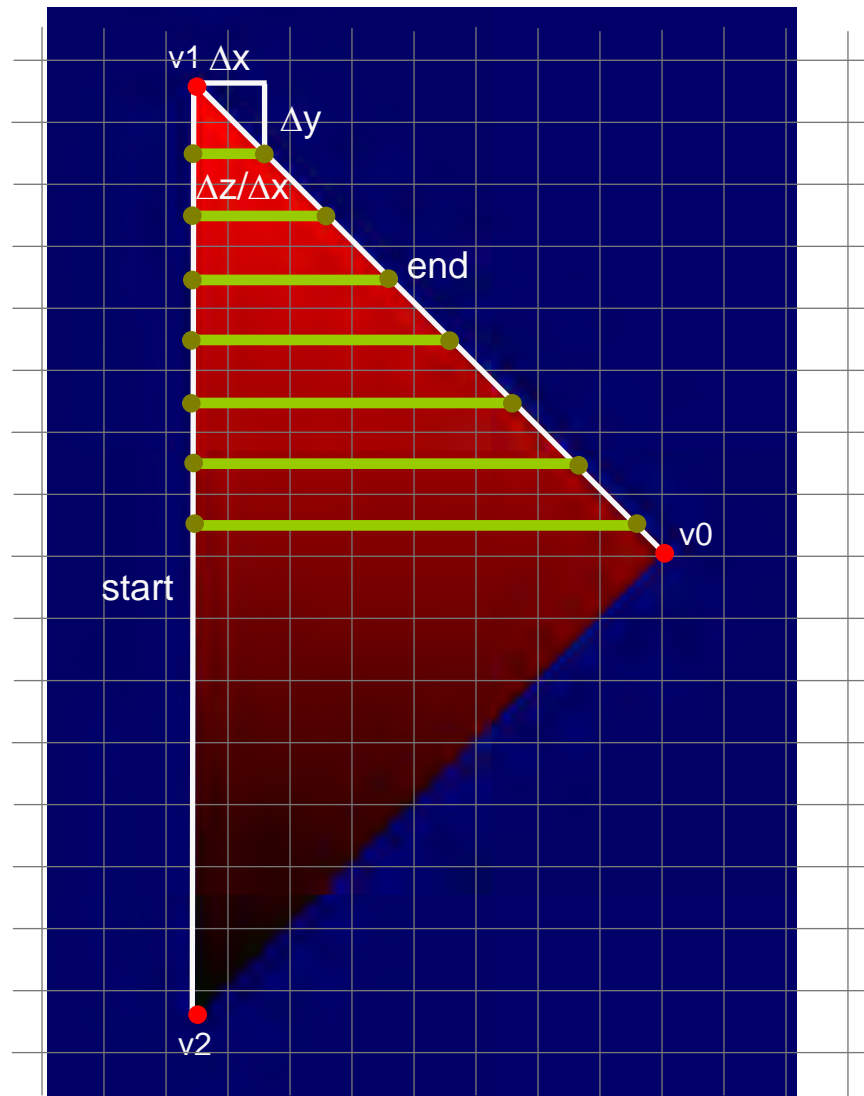
- You can interpolate:
  - Colors (3)
  - Depth (1)
  - Normals (3)
  - Texture coordinates (1,2,3)
  - etc.

# Status

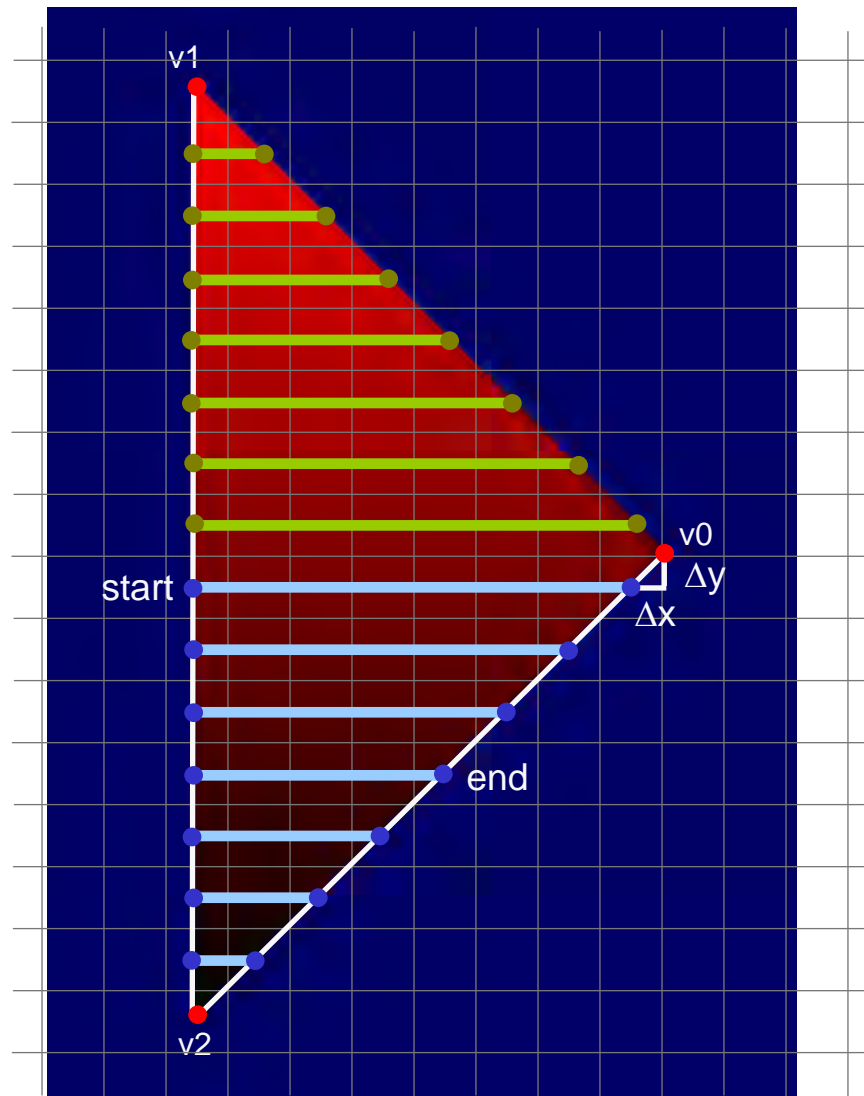
---

- So now we have an algorithm that goes through the bounding box and determines which pixels should be rasterized by the triangle
- Then we must work out the interpolated value separately for each pixel
- Very inefficient...

# Edge-walking rasterization



# Edge-walking rasterization



# Triangle rasterization algorithm

---

```
void rasterizeTriangle(void) {
    // sort vertices in order of descending y
    // identify scanlines of first half of the triangle
    // compute screen-space derivatives along edges

    for each scanline {
        // find span by interpolating vertex coordinates
        // find start and end values for interpolants
        // (color, depth, texture coordinates)
        // rasterize span

        for each pixel in span {
            // interpolate color, depth, texture coords
            // perform depth buffering
            // shade pixel
        }
    }

    // repeat for the second half of the triangle
}
```



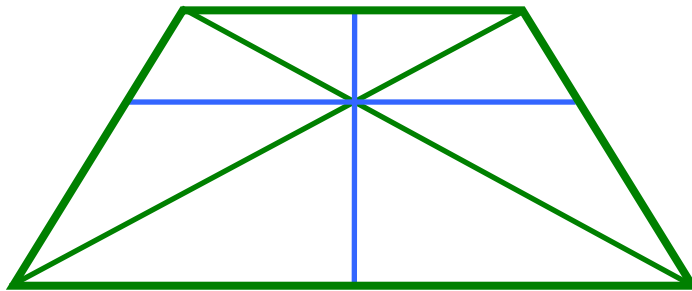
# Acceleration

---

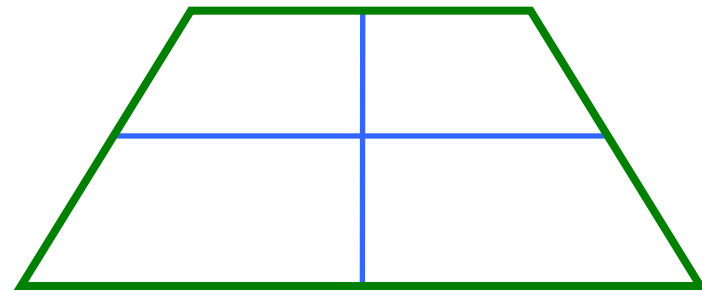
- Digital Differential Analyzer (DDA)
  - Represent slope and accumulation with fixed point
  - Have to be careful to avoid overflow
- Homogeneous recursive descent
  - Used in modern GPU's

# Perspective-correct interpolation

---



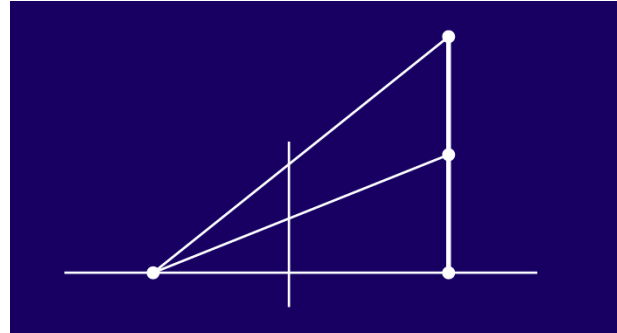
perspective correct



perspective incorrect

# Perspective-correct interpolation

---



# Solution

---

- Project all interpolants first by dividing by  $w$
- Interpolate interpolants as well as  $1/w$
- Get original value back by multiplying by interpolated  $w$

# Once fragments are rasterized

---

- They can be shaded (colored)
- Texture mapping
- Fragment program
- Etc.

# Framebuffer operations

---

- Framebuffer is essentially a large memory array to store the color, depth, stencil data
- Initial framebuffers (circa 1975) were very small (e.g. Evans and Sutherland)
- Made possible by large arrays of DRAM

# Color buffer

---

- Essentially a large array of color values
- Read out for display to the screen
- Allows for blend mode operations, e.g.:

$$f = \alpha C_{\text{src}} + (1-\alpha) C_{\text{dst}}$$

# Depth buffer

---

- Solution to the hidden surface problem
- Generate depth for every fragment rasterized
- Compare to the value currently in the buffer, replace if smaller

- Essentially:

```
void depthBuffer1D(void) {
    z = Z_MAX;
    for each fragment rasterized f {
        if (f.z < z) {
            z = f.z;
            write f into framebuffer
        }
    }
}
```

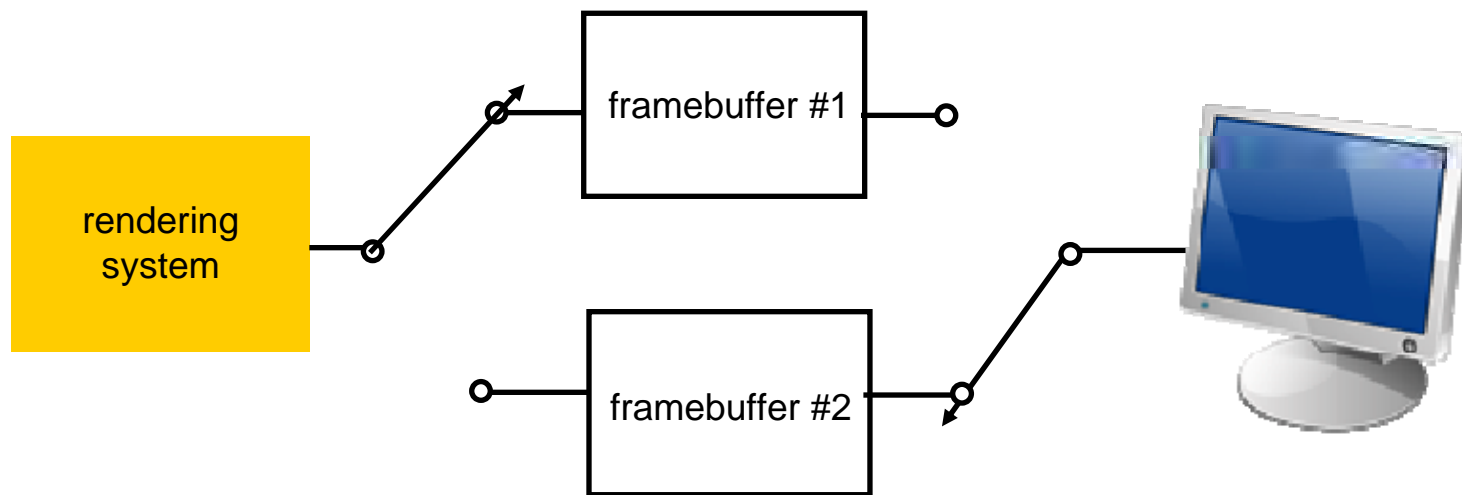
# Stencil buffer

---

- Essentially a state counter at each pixel
- Can keep track of how many times each pixel was rasterized
- Can be used for various algorithms, such as stencil shadows

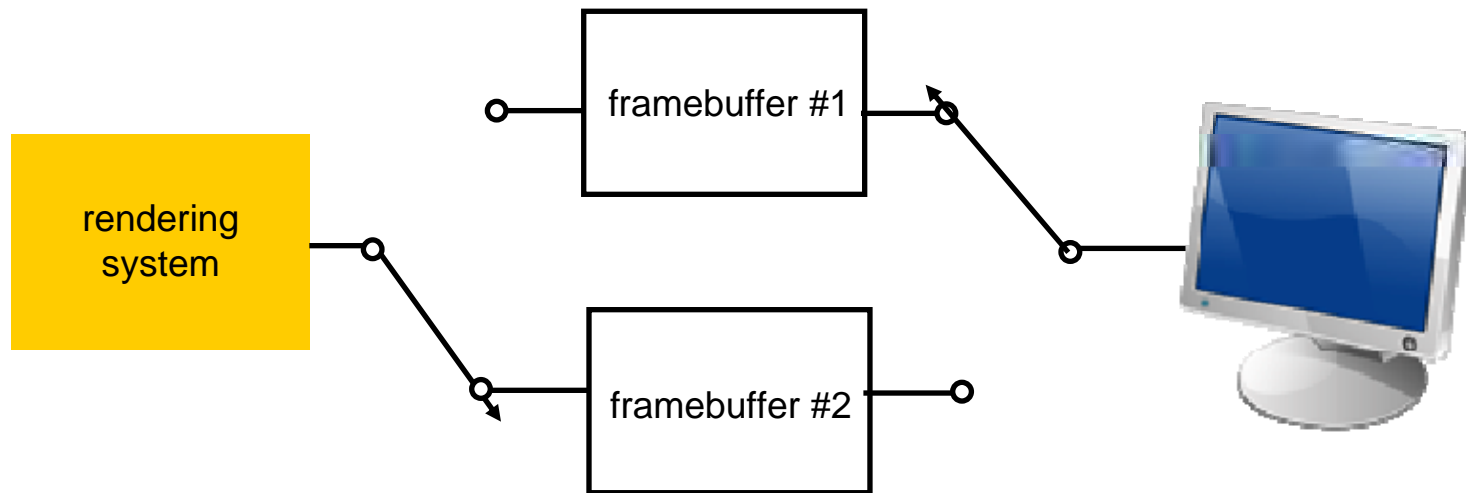
# Double buffering

---



# Double buffering

---



# That's the pipeline!

---

- We started off with triangles and ended up with pixels on the screen!



# Graphics pipeline

---

