Scan Conversion

Wolfgang Heidrich

Course News

Assignment 2

- Due March 2

Homework 3

- Discussed in labs next week

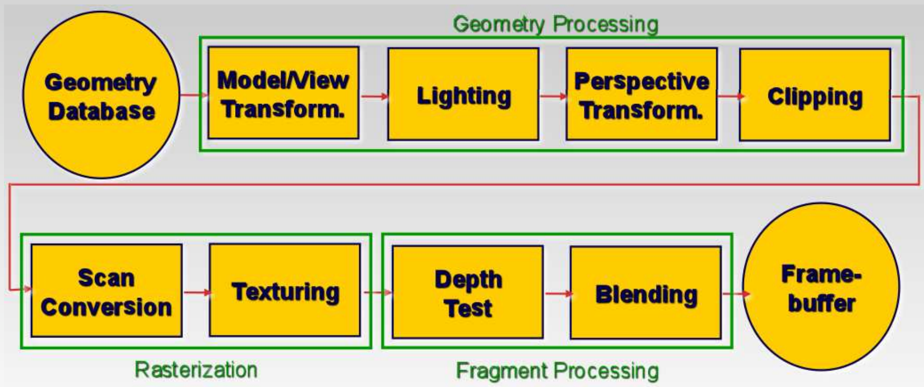
Reading (this week)

- Chapter 3

Reading (next week)

- Chapter 8

The Rendering Pipeline



Wolfgang Heidrich

Scan Conversion - Rasterization

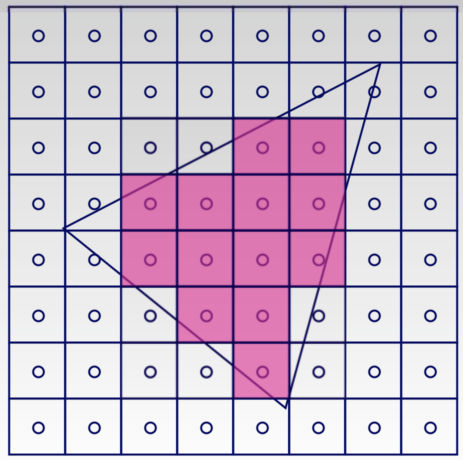
Convert continuous rendering primitives into discrete fragments/pixels

- Lines
  - *Midpoint/Bresenham*
- Triangles
  - *Flood fill*
  - *Scanline*
  - *Implicit formulation*
- Interpolation

Wolfgang Heidrich

Scan Conversion of Polygons

One possible scan conversion

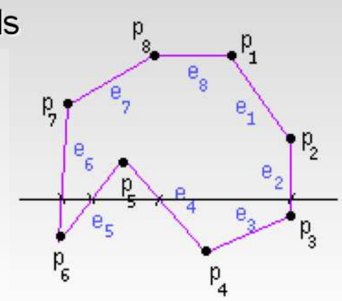


Wolfgang Heidrich

Scan Conversion of Polygons

A General Algorithm

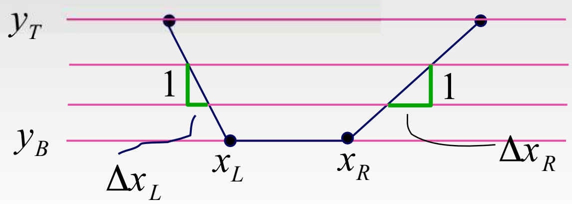
- Intersect each scanline with all edges
- Sort intersections in x
- Calculate parity to determine in/out
- Fill the 'in' pixels



Wolfgang Heidrich

Edge Walking

```
for (y=yB; y<=yT; y++) {
    for (x=xL; x<=xR; x++)
     setPixel(x,y);
    xL += DxL;
    xR += DxR;
}
```



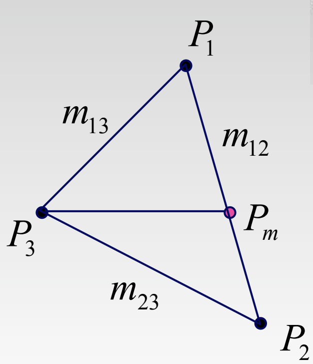
Wolfgang Heidrich

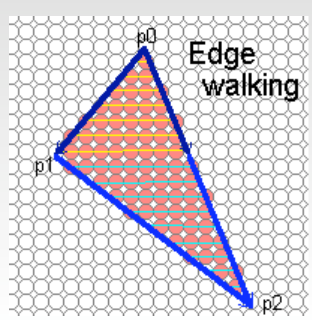
Edge Walking Triangles

- Split triangles into two regions with continuous left and right edges

scanTrapezoid($x_3, x_m, y_3, y_1, \frac{1}{m_{13}}, \frac{1}{m_{12}}$ **)**

scanTrapezoid($x_2, x_2, y_2, y_3, \frac{1}{m_{23}}, \frac{1}{m_{12}}$ **)**

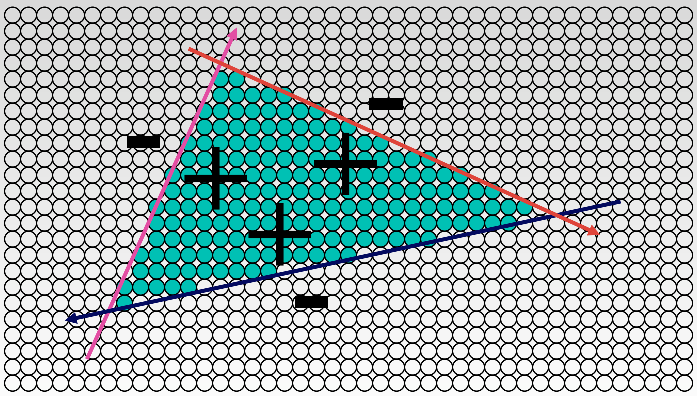




Wolfgang Heidrich

Modern Rasterization: Edge Equations

Define a triangle as follows:

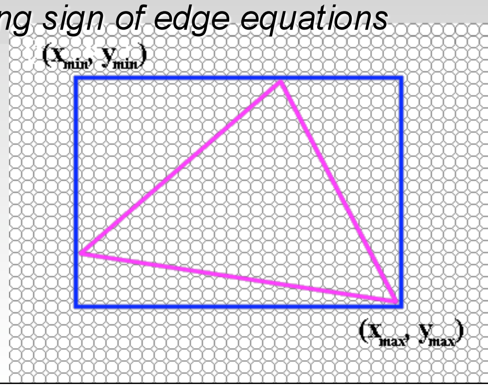


Wolfgang Heidrich

Using Edge Equations

Usage:

- Go over each pixel in bounding rectangle
- Check if pixel is inside/outside of triangle
  - *Using sign of edge equations*



Wolfgang Heidrich

Edge Equations

Counter-Clockwise Triangles

- The equation L(x,y) as specified above is *negative inside, positive outside*
  - *Flip sign:*

$$L(x,y) = -(y_e - y_s)(x - x_s) + (y - y_s)(x_e - x_s) = 0$$

Clockwise triangles

- *Use original formula*

$$L(x,y) = (y_e - y_s)(x - x_s) - (y - y_s)(x_e - x_s) = 0$$

Wolfgang Heidrich

Discussion of Polygon Scan Conversion Algorithms

On old hardware:

- Use first scan-conversion algorithm
  - *Scan-convert edges, then fill in scanlines*
  - *Compute interpolated values by interpolating along edges, then scanlines*
- Requires clipping of polygons against viewing volume
- Faster if you have a few, large polygons
- Possibly faster in software

Wolfgang Heidrich

Discussion of Polygon Scan Conversion Algorithms

Modern GPUs:

- Use edge equations
  - *And plane equations for attribute interpolation*
  - *No clipping of primitives required*
- Faster with many small triangles

Additional advantage:

- Can control the order in which pixels are processed
- Allows for more memory-coherent traversal orders
  - *E.g. tiles or space-filling curve rather than scanlines*

Wolfgang Heidrich

Triangle Rasterization Issues (Independent of Algorithm)

Exactly which pixels should be lit?

- A: Those pixels inside the triangle edge (of course)

But what about pixels exactly on the edge?

- Draw them: order of triangles matters (it shouldn't)
- Don't draw them: gaps possible between triangles

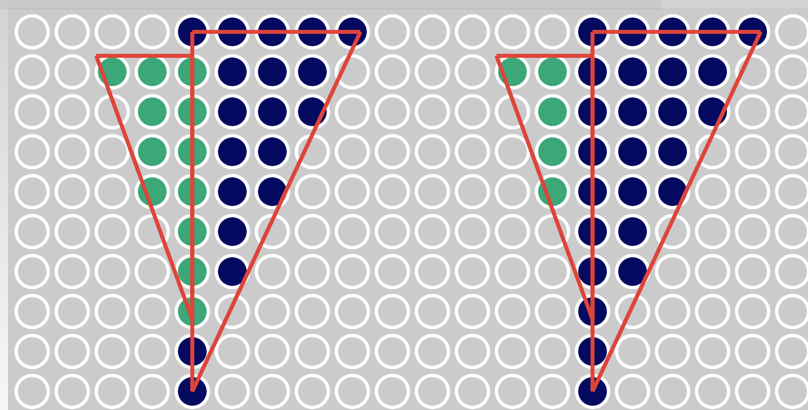
We need a consistent (if arbitrary) rule

- Example: draw pixels on left or top edge, but not on right or bottom edge

Wolfgang Heidrich

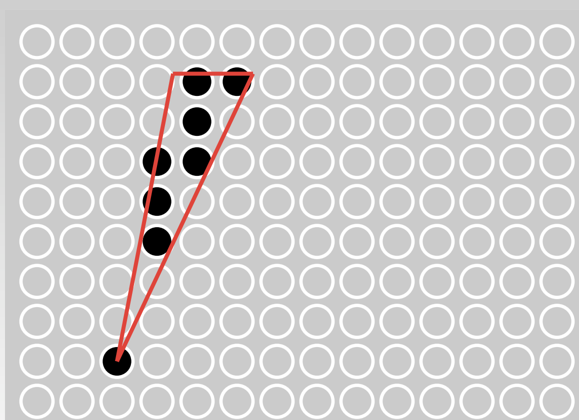
Triangle Rasterization Issues

Shared Edge Ordering



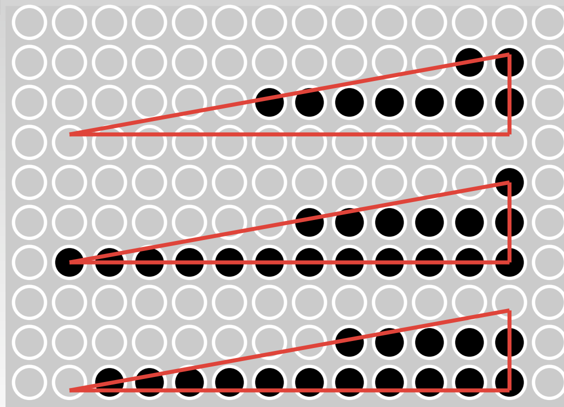
Triangle Rasterization Issues

Sliver



Triangle Rasterization Issues

Moving Slivers



Triangle Rasterization Issues

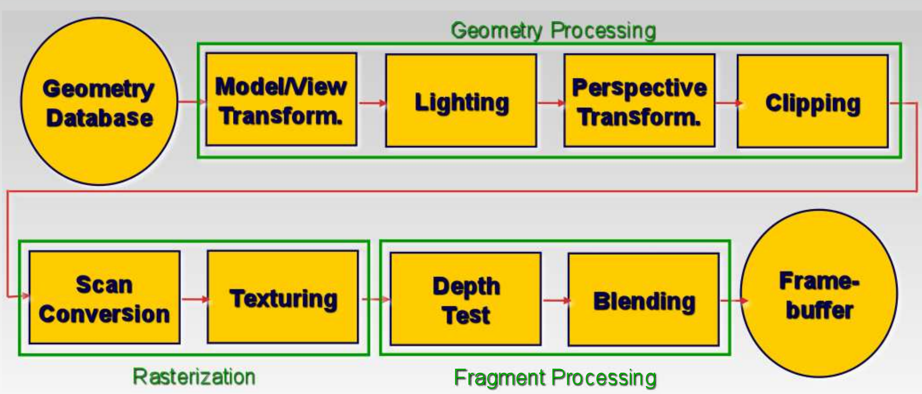
These are ALIASING Problems

- Problems associated with representing continuous functions (triangles) with finite resolution (pixels)
- More on this problem when we talk about sampling…

Shading

Wolfgang Heidrich

The Rendering Pipeline



Shading

Input to Scan Conversion:

- Vertices of triangles (lines, quadrilaterals…)
- Color (per vertex)
  - *Specified with glColor*
  - *Or: computed with lighting*
- World-space normal (per vertex)
  - *Left over from lighting stage*

Shading Task:

- Determine color of every pixel in the triangle

Shading

How can we assign pixel colors using this information?

- Easiest: flat shading
  - *Whole triangle gets one color (color of 1st vertex)*
- Better: Gouraud shading
  - *Linearly interpolate color across triangle*
- Even better:
  - *Linearly interpolate the normal vector*
  - *Compute lighting for every pixel*
  - *Note: not supported by rendering pipeline as discussed so far*

Flat Shading

- Simplest approach calculates illumination at a single point for each polygon

- Obviously inaccurate for smooth surfaces

Wolfgang Heidrich

Flat Shading Approximations

If an object really is faceted, is this accurate?

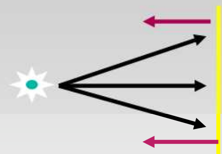
Wolfgang Heidrich

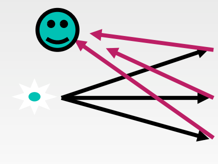
Flat Shading Approximations

If an object really is faceted, is this accurate?

no!

- For point sources, the direction to light varies across the facet





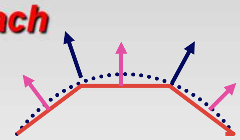
- For specular reflectance, direction to eye varies across the facet

Improving Flat Shading

What if evaluate Phong lighting model at each pixel of the polygon?

- Better, but result still clearly faceted

For smoother-looking surfaces we introduce vertex normals at each vertex

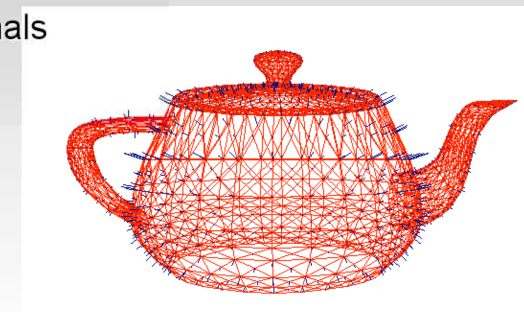


- Usually different from facet normal
- Used ***only*** for shading
- Think of as a better approximation of the ***real*** surface that the polygons approximate

Vertex Normals

Vertex normals may be

- Provided with the model
- Computed from first principles
- Approximated by averaging the normals of the facets that share the vertex

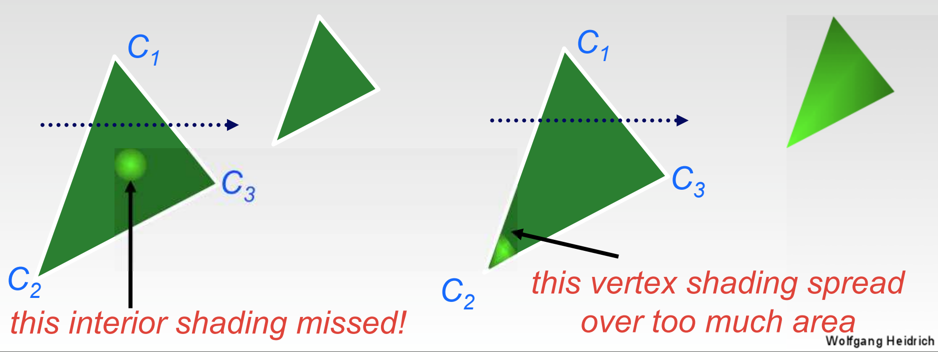


Gouraud Shading Artifacts

often appears dull, chalky

lacks accurate specular component

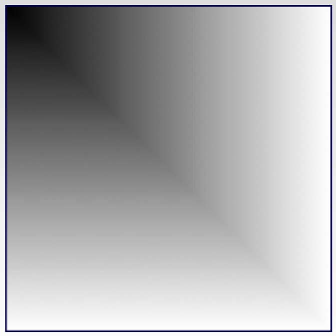
- if included, will be averaged over entire polygon



Gouraud Shading Artifacts

Mach bands

- Eye enhances discontinuity in first derivative
- Very disturbing, especially for highlights



Wolfgang Heidrich

Phong Shading

linearly interpolating surface normal across the facet, applying Phong lighting model at every pixel

- Same input as Gouraud shading
- Pro: much smoother results
- Con: considerably more expensive

Not the same as Phong lighting

- Common confusion
- **Phong lighting:** empirical model to calculate illumination at a point on a surface

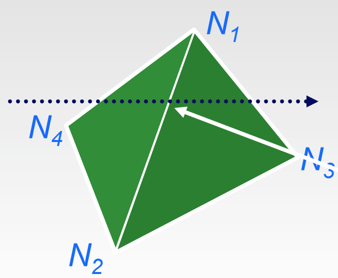
Wolfgang Heidrich

Phong Shading

Linearly interpolate the vertex normals

- Compute lighting equations at each pixel
- Can use specular component

$$I_{total} = k_a I_{ambient} + \sum_{i=1}^{\# lights} I_i \left(k_d (\mathbf{n} \cdot \mathbf{l_i}) + k_s (\mathbf{v} \cdot \mathbf{r_i})^{n_{shiny}} \right)$$



remember: normals used in diffuse and specular terms

discontinuity in normal's rate of change harder to detect

Wolfgang Heidrich

Phong Shading Difficulties

Computationally expensive

- Per-pixel vector normalization and lighting computation!
- Floating point operations required

Lighting after perspective projection

- Messes up the angles between vectors
- Have to keep eye-space vectors around

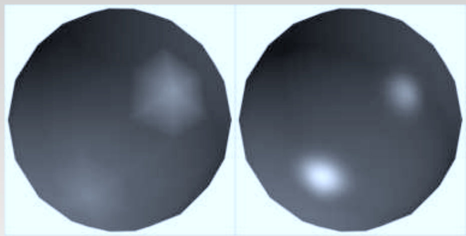
No direct support in standard rendering pipeline

- But can be simulated with texture mapping, procedural shading hardware (see later)

Wolfgang Heidrich

Shading Artifacts: Silhouettes

Polygonal silhouettes remain



Gouraud *Phong*

Wolfgang Heidrich

How to Interpolate?

Need to propagate vertex attributes to pixels

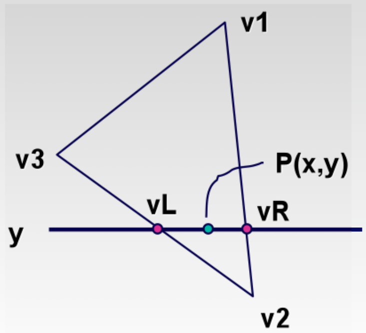
- Interpolate between vertices:
  - *z (depth)*
  - *r,g,b color components*
  - N_x, N_y, N_z *surface normals*
  - *u,v texture coordinates (talk about these later)*
- Three equivalent ways of viewing this (for triangles)
  1. *Linear interpolation*
  2. *Barycentric coordinates*
  3. *Plane Equation*

Wolfgang Heidrich

1. Linear Interpolation

Interpolate quantity along L and R edges

- (as a function of y)
- Then interpolate quantity as a function of x



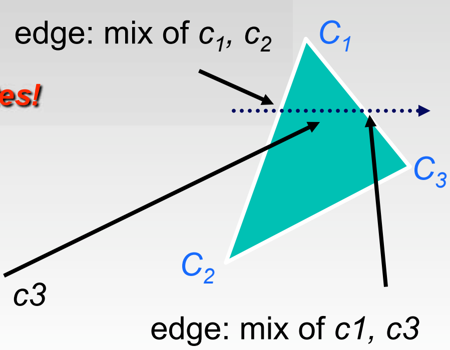
Wolfgang Heidrich

Linear Interpolation

Most common approach, and what OpenGL does

- Perform Phong lighting at the vertices
- Linearly interpolate the resulting colors over faces
  - *Along edges*
  - *Along scanlines*

Same as Barycentric Coordinates!



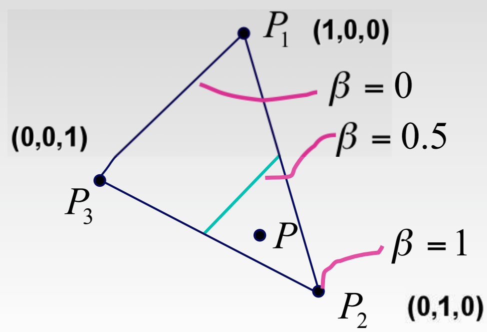
Wolfgang Heidrich

2. Barycentric Coordinates

Have seen this before

- Barycentric Coordinates: weighted combination of vertices, with weights summing to 1

$$P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3$$
$$\alpha + \beta + \gamma = 1$$
$$0 \le \alpha, \beta, \gamma \le 1$$



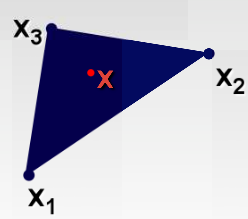
Wolfgang Heidrich

Barycentric Coordinates

- Convex combination of 3 points

$$\mathbf{x} = \alpha \cdot \mathbf{x}_1 + \beta \cdot \mathbf{x}_2 + \gamma \cdot \mathbf{x}_3$$
$$\text{with } \alpha + \beta + \gamma = 1, \ 0 \le \alpha, \beta, \gamma \le 1$$

- α, β, and γ are called *barycentric coordinates*



Wolfgang Heidrich

Barycentric Coordinates

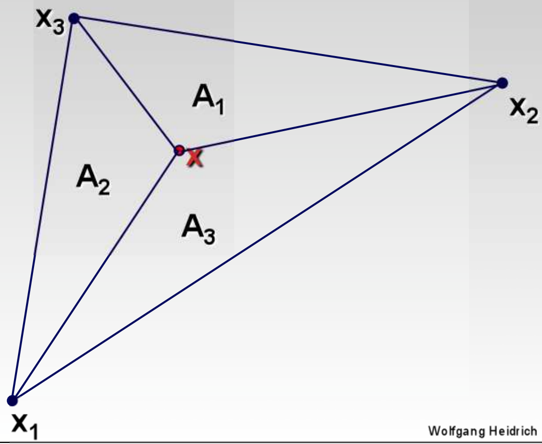
One way to compute them:

$$\mathbf{x} = \alpha\mathbf{x}_1 + \beta\mathbf{x}_2 + \gamma\mathbf{x}_3 \quad \text{with}$$

$$\alpha = A_1 / A$$

$$\beta = A_2 / A$$

$$\gamma = A_3 / A$$



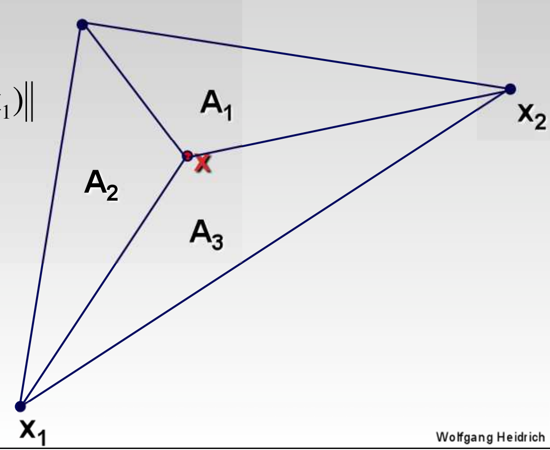
Wolfgang Heidrich

Barycentric Coordinates

How to compute areas?

- Cross products!
- e.g:

$$A_1 = \frac{1}{2}\left\|(\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x} - \mathbf{x}_1)\right\|$$



Wolfgang Heidrich

3. Plane Equation

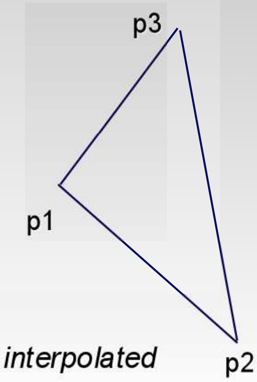
Observation: Quantities vary linearly across image plane

- E.g.: r = Ax + By + C
  - *r= red channel of the color*
  - *Same for g, b, Nx, Ny, Nz, z…*
- From info at vertices we know:

$$r_1 = Ax_1 + By_1 + C$$

$$r_2 = Ax_2 + By_2 + C$$

$$r_3 = Ax_3 + By_3 + C$$



  - *Solve for A, B, C*
  - *One-time set-up cost per triangle and interpolated quantity*

Wolfgang Heidrich

Coming Up:

Next week

- Clipping, hidden surface removal

Wolfgang Heidrich