Shading
Clipping

Wolfgang Heidrich

Course News

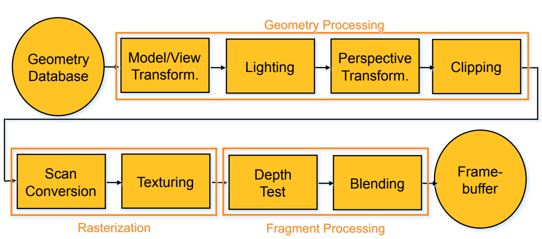
Assignment 2
- Due Monday, Feb 28

Homework 3
- Discussed in labs this week

Homework 4

Reading
- Chapters 8, 9
- Hidden surface removal, shading

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

Flat Shading Approximations

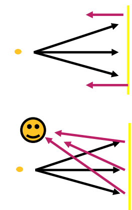
If an object really is faceted, is this accurate?

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

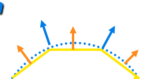


Improving Flat Shading

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

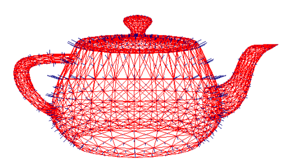


- Usually different from facet normal
- Used *only* for shading
- Think of as a better approximation of the *real* surface that the polygons approximate

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

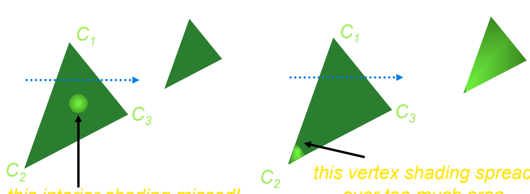


Gouraud Shading Artifacts

often appears dull, chalky

lacks accurate specular component

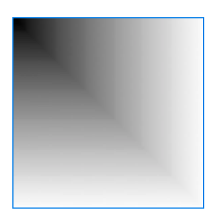
- if included, will be averaged over entire polygon



Gouraud Shading Artifacts

Mach bands

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



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

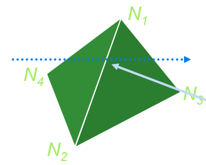
- Common confusion
- Phong lighting: empirical model to calculate illumination at a point on a surface

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

Phong Shading Difficulties

Computationally expensive

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

Lighting after perspective projection

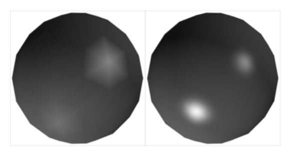
- Messes up the angles between vectors
- Have to keep eye-space vectors around

No direct support in standard rendering pipeline

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

Shading Artifacts: Silhouettes

Polygonal silhouettes remain



Gouraud *Phong*

How to Interpolate?

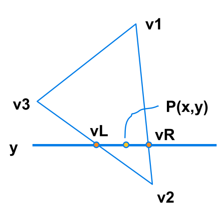
Need to propagate vertex attributes to pixels

- 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*

1. Linear Interpolation

Interpolate quantity along L and R edges

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

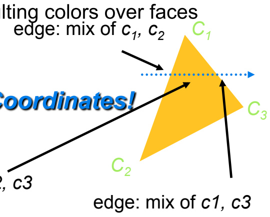


Linear Interpolation

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

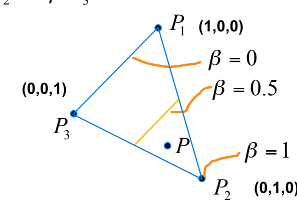
Have seen this before

- 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

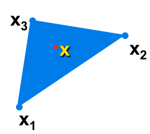
Barycentric Coordinates

- 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

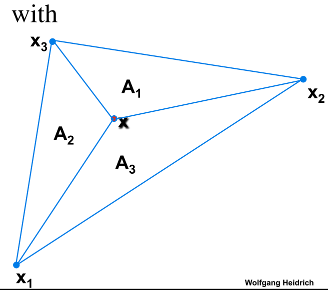
One way to compute them:

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

$$\alpha = A_1 / A$$

$$\beta = A_2 / A$$

$$\gamma = A_3 / A$$



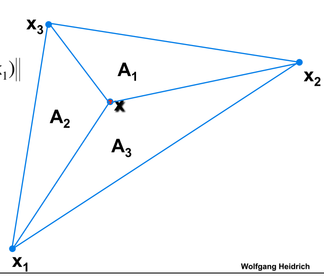
Wolfgang Heidrich

Barycentric Coordinates

How to compute areas?

- Cross products!
- e.g:

$$A_3 = \frac{1}{2} \left\| (\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x} - \mathbf{x}_1) \right\|$$



Wolfgang Heidrich

3. Plane Equation

Observation: Quantities vary linearly across image plane

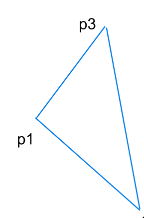
- E.g.: r = Ax + By + C
  - *r= red channel of the color*
  - *Same for g, b, Nx, Ny, Nz, z…*
- From info at vertices we know:

$$r_1 = Ax_1 + By_1 + C$$

$$r_2 = Ax_2 + By_2 + C$$

$$r_3 = Ax_3 + By_3 + C$$

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



Wolfgang Heidrich

Discussion

Which algorithm to use when?

- Scanline interpolation
  - *Together with trapezoid scan conversion*
- Plane equations
  - *Together with edge equation scan conversion*
- Barycentric coordinates
  - *Not useful in the current context*
  - *But: method of choice for ray-tracing*
    - Whenever you only need to compute the value for a <u>single</u> pixel

Clipping

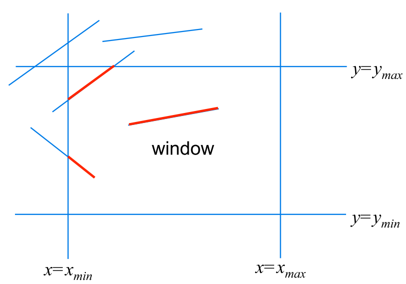
Wolfgang Heidrich

Line Clipping

Purpose

- Originally: 2D
  - *Determine portion of line inside an axis-aligned rectangle (screen or window)*
- 3D
  - *Determine portion of line inside axis-ligned parallelpiped (viewing frustum in NDC)*
  - *Simple extension to the 2D algorithms*

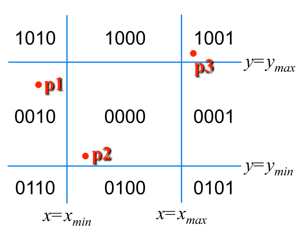
Line Clipping



Line Clipping

Outcodes (Cohen, Sutherland '74)

- 4 flags encoding position of a point relative to top, bottom, left, and right boundary
- E.g.:
  - OC(**p1**)=0010
  - OC(**p2**)=0000
  - OC(**p3**)=1001



Line Clipping

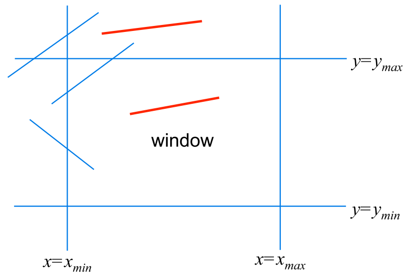
Line segment:

- (**p1**,**p2**)

Trivial cases:

- OC(**p1**)== 0 && OC(**p2**)==0
  - *Both points inside window, thus line segment completely visible (trivial accept)*
- (OC(**p1**) & OC(**p2**))!= 0 (i.e. **bitwise** "and"!)
  - *There is (at least) one boundary for which both points are outside (same flag set in both outcodes)*
  - *Thus line segment completely outside window (trivial reject)*

Line Clipping



Line Clipping

α-Clipping

- Handling of all the non-trivial cases
- Improvement of earlier algorithms (Cohen/Sutherland, Cyrus/Beck, Liang/Barsky)
- Define *window-edge-coordinates* of a point $\mathbf{p}=(x,y)^T$
  - $WEC_L(\mathbf{p})= x-x_{min}$
  - $WEC_R(\mathbf{p})= x_{max}-x$
  - $WEC_B(\mathbf{p})= y-y_{min}$
  - $WEC_T(\mathbf{p})= y_{max}-y$

Negative if outside!

Line Clipping

α-Clipping

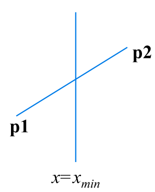
- Line segment defined as: p1+ α(p2-p1)
- Intersection point with one of the borders (say, left):

$$x_1 + \alpha(x_2 - x_1) = x_{min} \Leftrightarrow$$

$$\alpha = \frac{x_{min} - x_1}{x_2 - x_1}$$

$$= \frac{x_{min} - x_1}{(x_2 - x_{min}) - (x_1 - x_{min})}$$

$$= \frac{\mathrm{WEC}_L(x_1)}{\mathrm{WEC}_L(x_1) - \mathrm{WEC}_L(x_2)}$$



Line Clipping

α-Clipping: algorithm

```
alphaClip( p1, p2, window ) {
  Determine window-edge-coordinates of p1, p2
  Determine outcodes OC(p1), OC(p2)

  Handle trivial accept and reject

  α1= 0; // line parameter for first point
  α2= 1; // line parameter for second point
  …
```


Line Clipping

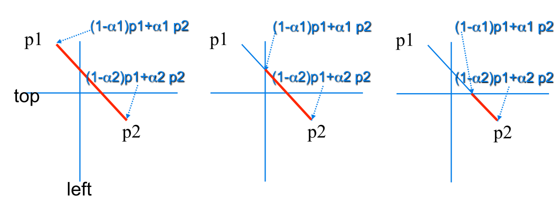
α-Clipping: algorithm (cont.)

```
  …
  // now clip point p1 against all edges
  if( OC(p1) & LEFT_FLAG ) {
    α= WEC_L(p1)/(WEC_L (p1) - WEC_L (p2));
    α1= max(α1, α );
  }

  Similarly clip p1 against other edges
  …
```


Line Clipping

α-Clipping: example for clipping p1



Start configuration — After clipping to left — After clipping to top

Line Clipping

α-Clipping: algorithm (cont.)

```
…
// now clip point p2 against all edges
if( OC(p2) & LEFT_FLAG ) {
   α= WEC_L(p2)/(WEC_L (p1) - WEC_L (p2));
   α2= min(α2, α );
}

Similarly clip p1 against other edges
…
```

Wolfgang Heidrich

Line Clipping

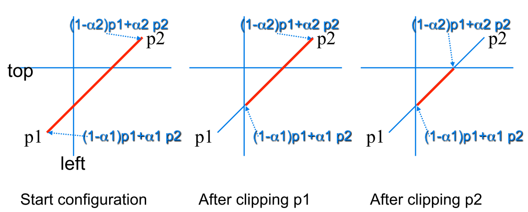
α-Clipping: algorithm (cont.)

```
...
// wrap-up
if(α1 > α2 )
   no output;
else
   output line from p1+α1(p2-p1) to p1+α2(p2-p1)
} // end of algorithm
```

Wolfgang Heidrich

Line Clipping

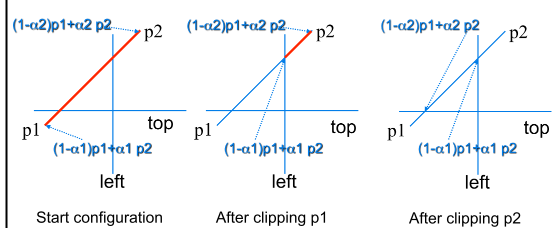
Example



Wolfgang Heidrich

Line Clipping

Another Example



Wolfgang Heidrich

Line Clipping in 3D

Approach:

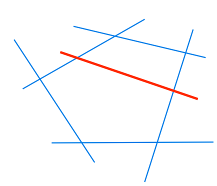
- Clip against parallelpiped in NDC (*after* perspective transform)
- Means that the clipping volume is always the same!
  - OpenGL: $x_{min}=y_{min}= -1$, $x_{max}=y_{max}= 1$
- Boundary lines become boundary planes
  - *But outcodes and WECs still work the same way*
  - *Additional front and back clipping plane*
    - $z_{min}=0$, $z_{max}=1$ in OpenGL

Wolfgang Heidrich

Line Clipping

Extensions

- Algorithm can be extended to clipping lines against
  - *Arbitrary convex polygons (2D)*
  - *Arbitrary convex polytopes (3D)*

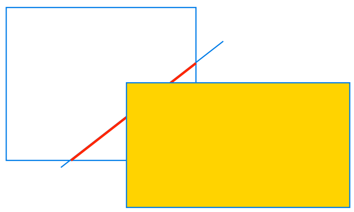


Wolfgang Heidrich

Line Clipping

Non-convex clipping regions

- E.g.: windows in a window system!

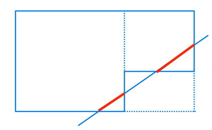


Wolfgang Heidrich

Line Clipping

Non-convex clipping regions

- Problem: arbitrary number of visible line segments
- Different approaches:
  - *Break down polygon into convex parts*
  - *Scan convert for full window, and discard hidden pixels*



Wolfgang Heidrich

Polygon Clipping

Objective

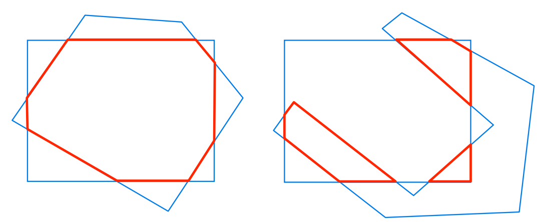
- 2D: clip polygon against rectangular window
  - *Or general convex polygons*
  - *Extensions for non-convex or general polygons*
- 3D: clip polygon against parallelpiped

Wolfgang Heidrich

Polygon Clipping

Not just clipping all boundary lines

- May have to introduce new line segments

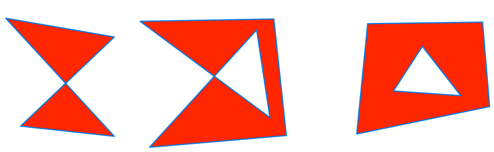


Wolfgang Heidrich

Polygon Clipping

Classes of Polygons

- Triangles
- Convex
- Concave
- Holes and self-intersection

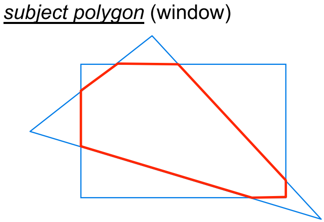


Wolfgang Heidrich

Polygon Clipping

Sutherland/Hodgeman Algorithm ('74)

- Arbitrary convex or concave <u>object polygon</u>
  - *Restriction to triangles does not simplify things*
- Convex <u>subject polygon</u> (window)

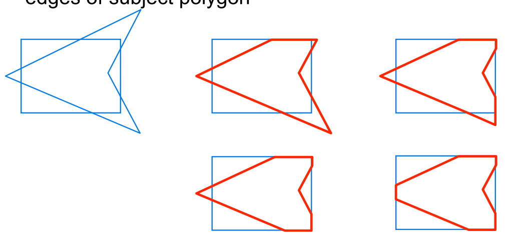


Wolfgang Heidrich

Polygon Clipping

Sutherland/Hodgeman Algorithm ('74)

- Approach: clip object polygon independently against all edges of subject polygon



Polygon Clipping

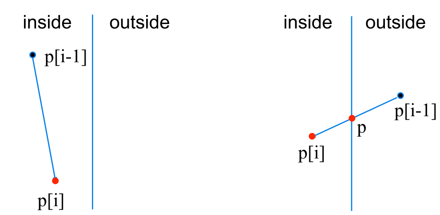
Clipping against one edge:

```
clipPolygonToEdge( p[n], edge ) {
  for( i= 0 ; i< n ; i++ ) {
    if( p[i] inside edge ) {
      if( p[i-1] inside edge ) // p[-1]= p[n-1]
        output p[i];
      else {
        p= intersect( p[i-1], p[i], edge );
        output p, p[i];
      }
    } else…
```


Polygon Clipping

Clipping against one edge (cont)

- p[i] inside: 2 cases



Output: p[i]　　Output: p, p[i]

Polygon Clipping

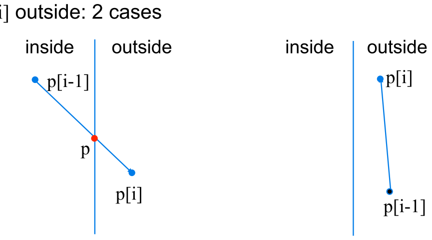
Clipping against one edge (cont)

```
  …
  else {            // p[i] is outside edge
    if( p[i-1] inside edge ) {
      p= intersect(p[i-1], p[I], edge );
      output p;
    }
  } // end of algorithm
```


Polygon Clipping

Clipping against one edge (cont)

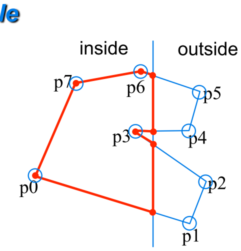
- p[i] outside: 2 cases



Output: p　　Output:nothing

Polygon Clipping

Example



Polygon Clipping

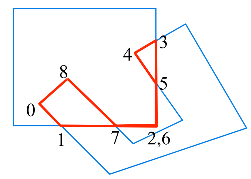
Sutherland/Hodgeman Algorithm

- Inside/outside tests: outcodes
- Intersection of line segment with edge: window-edge coordinates
- Similar to Cohen/Sutherland algorithm for line clipping

Polygon Clipping

Sutherland/Hodgeman Algorithm

- Discussion:
  - *Works for concave polygons*
  - *But generates degenerate cases*



Polygon Clipping

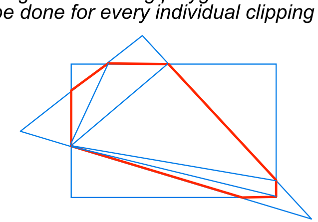
Sutherland/Hodgeman Algorithm

- Discussion:
  - *Clipping against individual edges independent*
    - Great for hardware (pipelining)
  - *All vertices required in memory at the same time*
    - Not so good, but unavoidable
    - Another reason for using triangles only in hardware rendering

Polygon Clipping

Sutherland/Hodgeman Algorithm

- For Rendering Pipeline:
  - *Re-triangulate resulting polygon (can be done for every individual clipping edge)*



Polygon Clipping

Other Polygon Clipping Algorithms

- Weiler/Aetherton '77:
  - *Arbitrary concave polygons with holes both as subject and as object polygon*
- Vatti '92:
  - *Self intersection allowed as well*
- … many more
  - *Improved handling of degenerate cases*
  - *But not often used in practice due to high complexity*

Coming Up:

Friday

- More clipping, hidden surface removal