



University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2010

Tamara Munzner

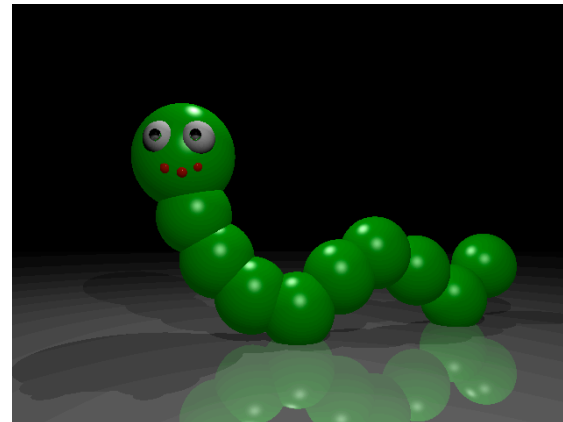
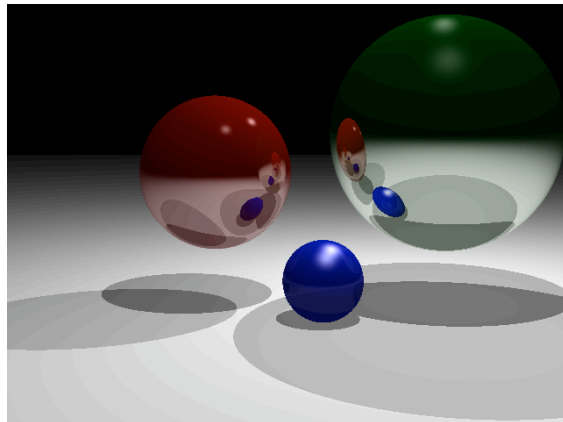
Advanced Rendering II, Clipping I

Week 8, Wed Mar 10

<http://www.ugrad.cs.ubc.ca/~cs314/Vjan2010>

News

- Project 3 out
 - due Fri Mar 26, 5pm
- raytracer
 - template code has significant functionality
 - clearly marked places where you need to fill in required code



News

- Project 2 F2F grading done
 - if you have not signed up, do so immediately with `glj3 AT cs.ubc.ca`
 - penalty already for being late
 - bigger penalty if we have to hunt you down

Reading for Advanced Rendering

- FCG Sec 8.2.7 Shading Frequency
- FCG Chap 4 Ray Tracing
- FCG Sec 13.1 Transparency and Refraction
 - (10.1-10.7 2nd ed)
- Optional - FCG Chap 24: Global Illumination

Review: Specifying Normals

- OpenGL state machine
 - uses last normal specified
 - if no normals specified, assumes all identical
- per-vertex normals

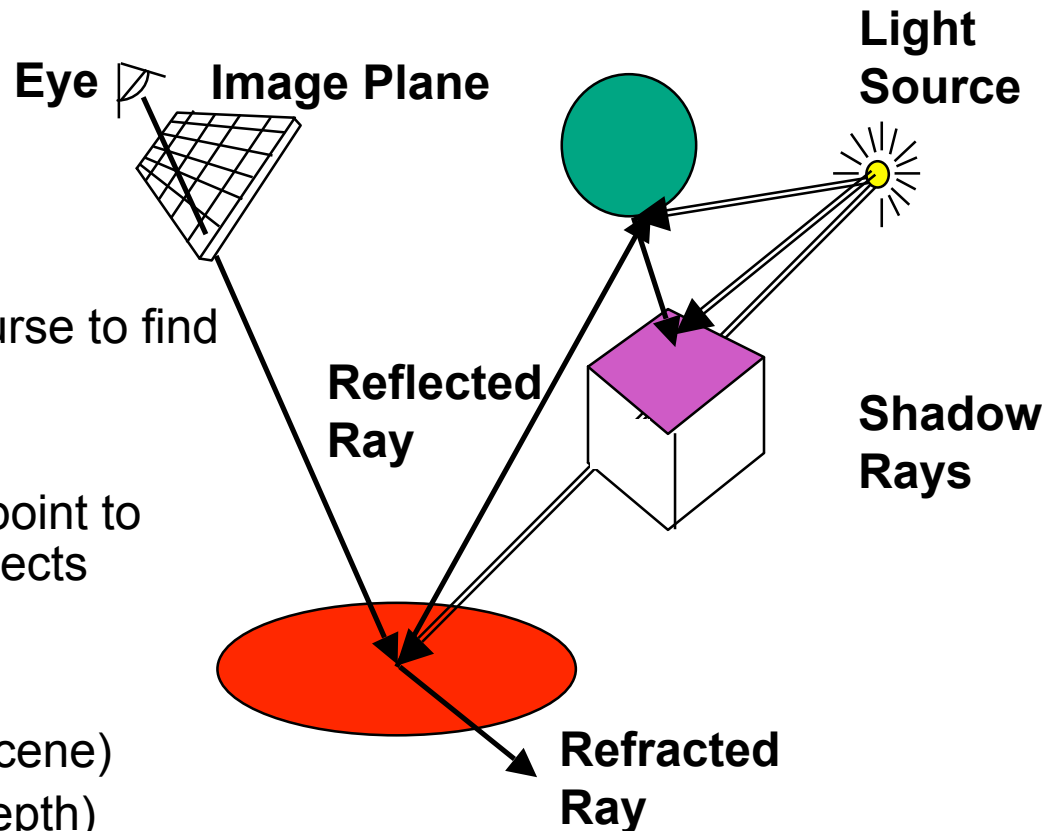
```
glNormal3f(1,1,1);
glVertex3f(3,4,5);
glNormal3f(1,1,0);
glVertex3f(10,5,2);
```
- per-face normals

```
glNormal3f(1,1,1);
glVertex3f(3,4,5);
glVertex3f(10,5,2);
```
- normal interpreted as direction from vertex location
- can automatically normalize (computational cost)

```
glEnable(GL_NORMALIZE);
```

Review: Recursive Ray Tracing

- ray tracing can handle
 - reflection (chrome/mirror)
 - refraction (glass)
 - shadows
- one primary ray per pixel
- spawn secondary rays
 - reflection, refraction
 - if another object is hit, recurse to find its color
 - shadow
 - cast ray from intersection point to light source, check if intersects another object
 - termination criteria
 - no intersection (ray exits scene)
 - max bounces (recursion depth)
 - attenuated below threshold

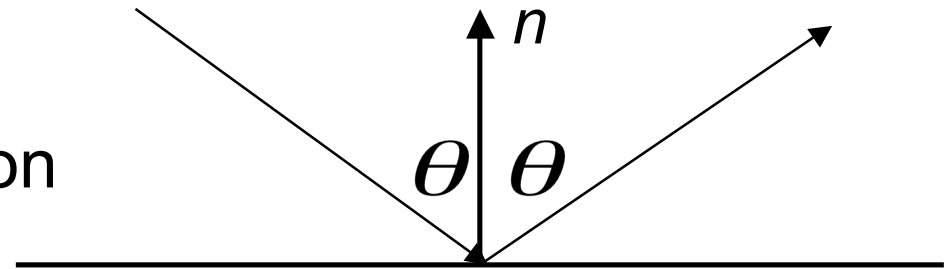


Review/Correction: Recursive Ray Tracing

```
RayTrace(r,scene)
obj := FirstIntersection(r,scene)
if (no obj) return BackgroundColor;
else begin
  if ( Reflect(obj) ) then
    reflect_color := RayTrace(ReflectRay(r,obj));
  else
    reflect_color := Black;
  if ( Transparent(obj) ) then
    refract_color := RayTrace(RefractRay(r,obj));
  else
    refract_color := Black;
  return Shade(reflect_color,refract_color,obj);
end;
```

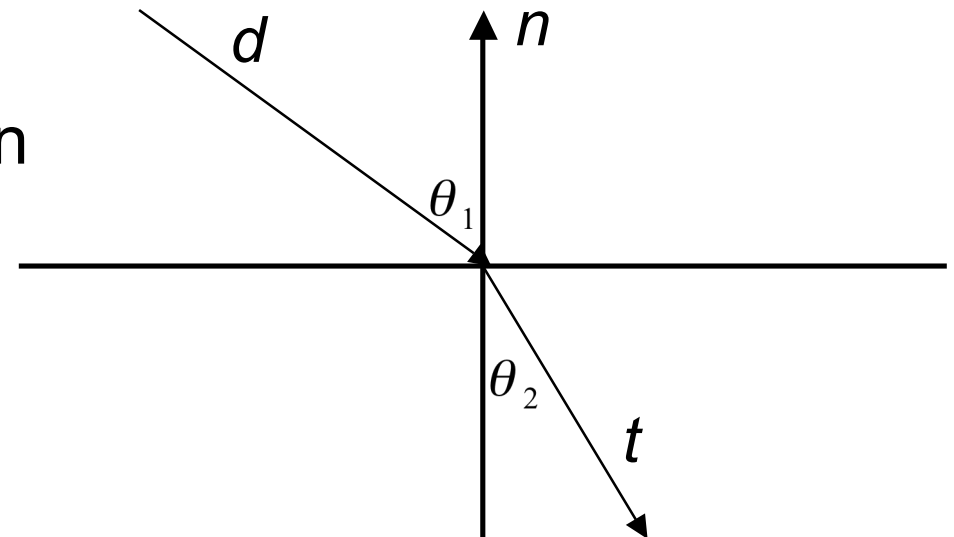
Review: Reflection and Refraction

- refraction: mirror effects
 - perfect specular reflection



- refraction: at boundary
- Snell's Law
 - light ray bends based on refractive indices c_1, c_2

$$c_1 \sin \theta_1 = c_2 \sin \theta_2$$

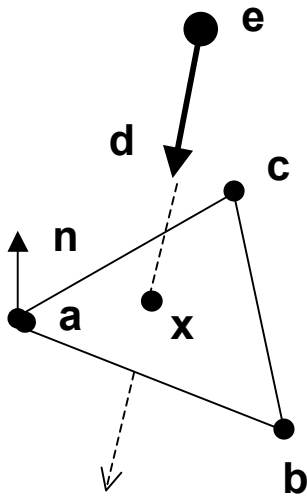


Review: Ray Tracing

- issues:
 - generation of rays
 - intersection of rays with geometric primitives
 - geometric transformations
 - lighting and shading
 - efficient data structures so we don't have to test intersection with *every* object

Ray-Triangle Intersection

- method in book is elegant but a bit complex
- easier approach: triangle is just a plane
 - intersect ray with plane



$$\text{normal : } \mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$

$$\text{ray : } \mathbf{x} = \mathbf{e} + t\mathbf{d}$$

$$\text{plane : } (\mathbf{p} - \mathbf{x}) \cdot \mathbf{n} = 0 \Rightarrow \mathbf{x} = \frac{\mathbf{p} \cdot \mathbf{n}}{\mathbf{n}}$$

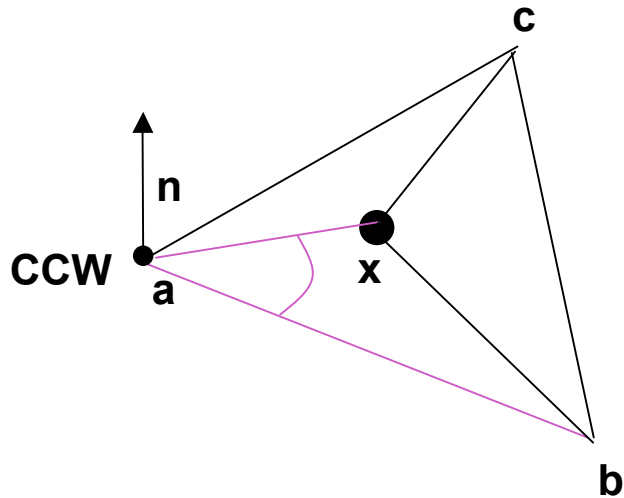
$$\frac{\mathbf{p} \cdot \mathbf{n}}{\mathbf{n}} = \mathbf{e} + t\mathbf{d} \Rightarrow t = -\frac{(\mathbf{e} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

\mathbf{p} is \mathbf{a} or \mathbf{b} or \mathbf{c}

- check if ray inside triangle

Ray-Triangle Intersection

- check if ray inside triangle
 - check if point counterclockwise from each edge (to its left)
 - check if cross product points in same direction as normal (i.e. if dot is positive)



$$(\mathbf{b} - \mathbf{a}) \times (\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} \geq 0$$

$$(\mathbf{c} - \mathbf{b}) \times (\mathbf{x} - \mathbf{b}) \cdot \mathbf{n} \geq 0$$

$$(\mathbf{a} - \mathbf{c}) \times (\mathbf{x} - \mathbf{c}) \cdot \mathbf{n} \geq 0$$

- more details at

<http://www.cs.cornell.edu/courses/cs465/2003fa/homeworks/raytri.pdf>

Ray Tracing

- issues:
 - generation of rays
 - intersection of rays with geometric primitives
 - **geometric transformations**
 - lighting and shading
 - efficient data structures so we don't have to test intersection with *every* object

Geometric Transformations

- similar goal as in rendering pipeline:
 - modeling scenes more convenient using different coordinate systems for individual objects
- problem
 - not all object representations are easy to transform
 - problem is fixed in rendering pipeline by restriction to polygons, which are affine invariant
 - ray tracing has different solution
 - ray itself is always affine invariant
 - thus: transform ray into object coordinates!

Geometric Transformations

- ray transformation
 - for intersection test, it is only important that ray is in same coordinate system as object representation
 - transform all rays into object coordinates
 - transform camera point and ray direction by inverse of model/view matrix
 - shading has to be done in world coordinates (where light sources are given)
 - transform object space intersection point to world coordinates
 - thus have to keep both world and object-space ray

Ray Tracing

- issues:
 - generation of rays
 - intersection of rays with geometric primitives
 - geometric transformations
 - **lighting and shading**
 - efficient data structures so we don't have to test intersection with *every* object

Local Lighting

- local surface information (normal...)
 - for implicit surfaces $F(x,y,z)=0$: normal $\mathbf{n}(x,y,z)$ can be easily computed at every intersection point using the gradient

$$\mathbf{n}(x, y, z) = \begin{pmatrix} \partial F(x, y, z) / \partial x \\ \partial F(x, y, z) / \partial y \\ \partial F(x, y, z) / \partial z \end{pmatrix}$$

- example: $F(x, y, z) = x^2 + y^2 + z^2 - r^2$

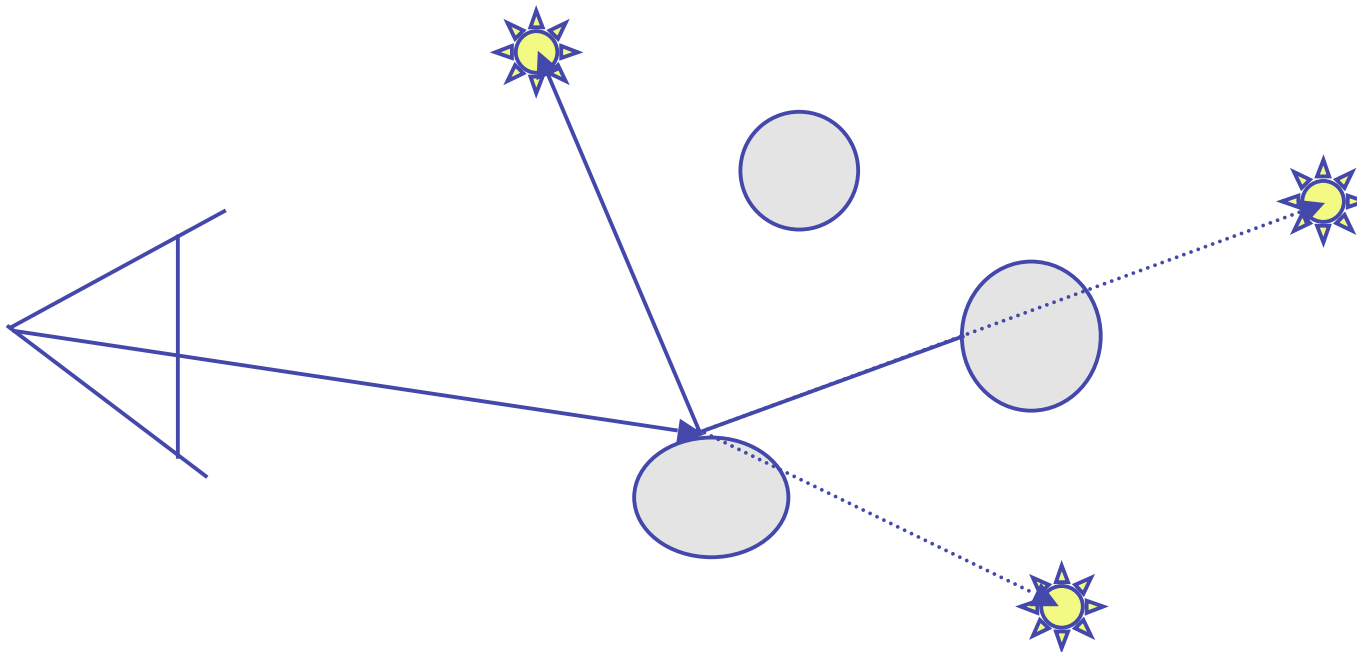
$$\mathbf{n}(x, y, z) = \begin{pmatrix} 2x \\ 2y \\ 2z \end{pmatrix} \quad \text{needs to be normalized!}$$

Local Lighting

- local surface information
 - alternatively: can interpolate per-vertex information for triangles/meshes as in rendering pipeline
 - now easy to use Phong shading!
 - as discussed for rendering pipeline
 - difference with rendering pipeline:
 - interpolation cannot be done incrementally
 - have to compute barycentric coordinates for every intersection point (e.g plane equation for triangles)

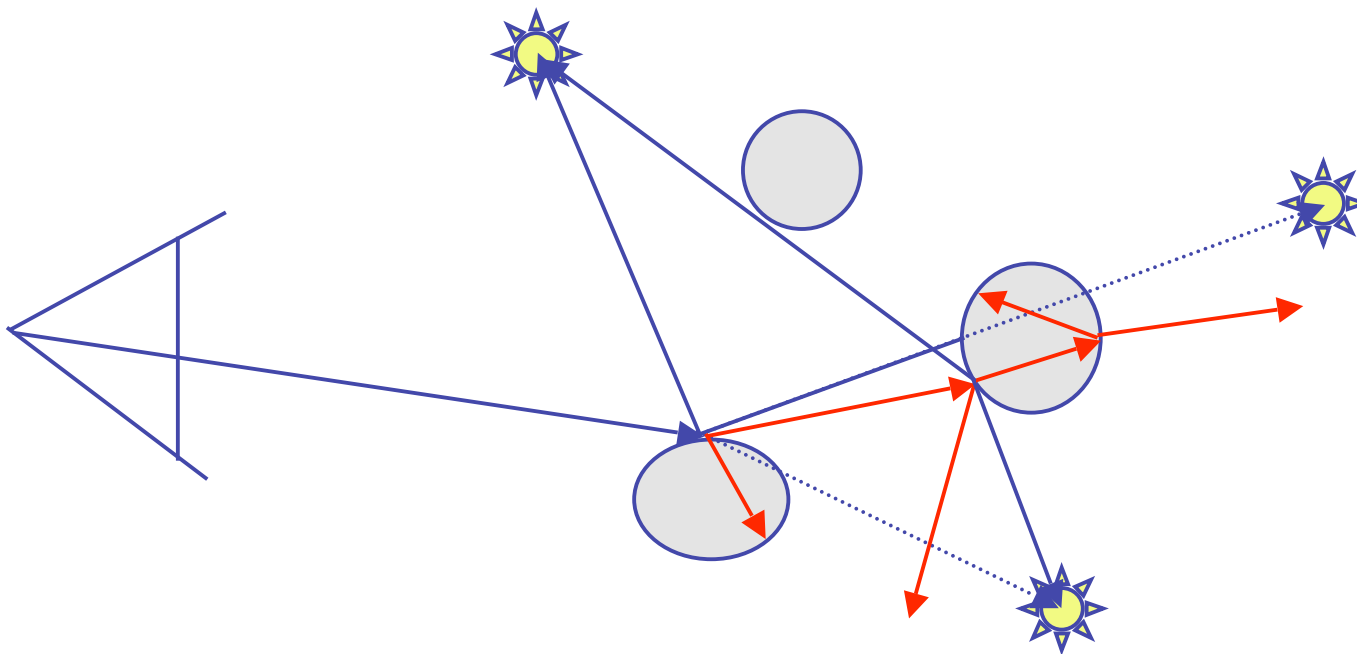
Global Shadows

- approach
 - to test whether point is in shadow, send out **shadow rays** to all light sources
 - if ray hits another object, the point lies in shadow



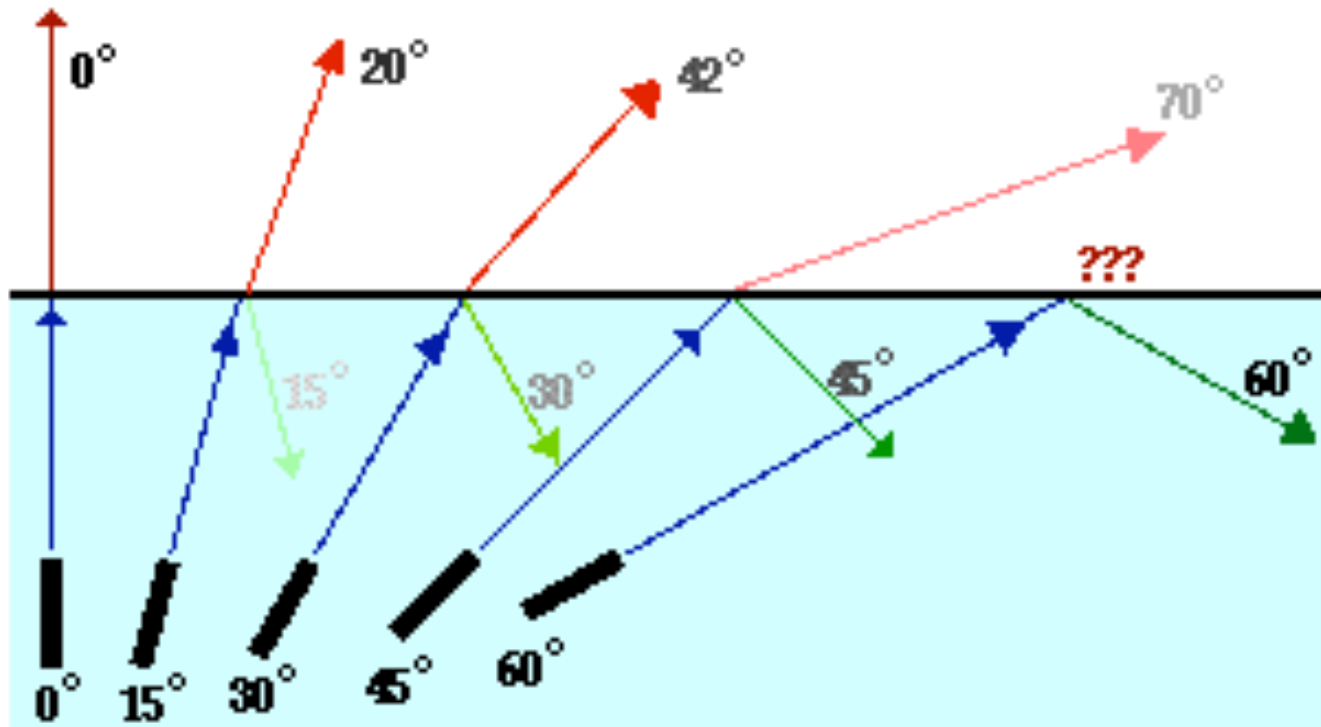
Global Reflections/Refractions

- approach
 - send rays out in reflected and refracted direction to gather incoming light
 - that light is multiplied by local surface color and added to result of local shading



Total Internal Reflection

As the angle of incidence increases from 0 to greater angles ...



- ...the refracted ray becomes dimmer (there is less refraction)
- ...the reflected ray becomes brighter (there is more reflection)
- ...the angle of refraction approaches 90 degrees until finally a refracted ray can no longer be seen.

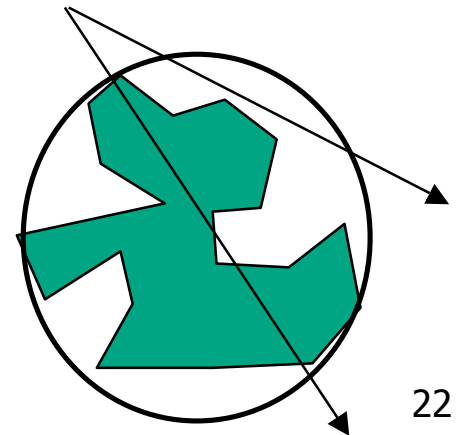
<http://www.physicsclassroom.com/Class/refrn/U14L3b.html>

Ray Tracing

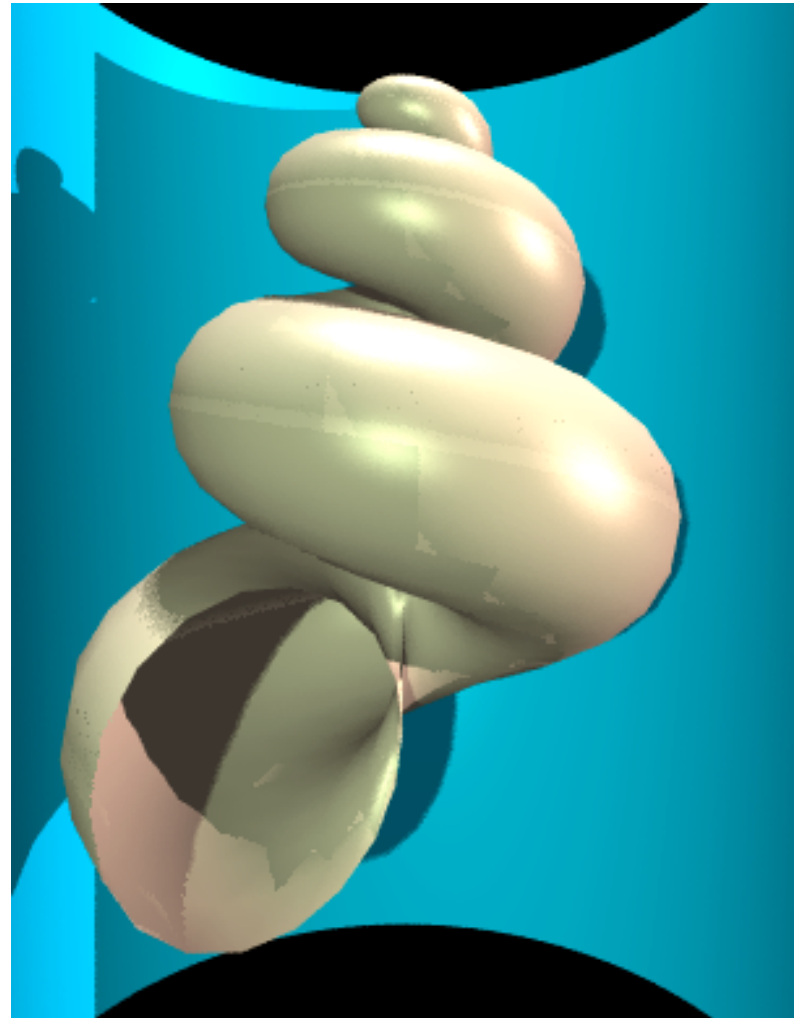
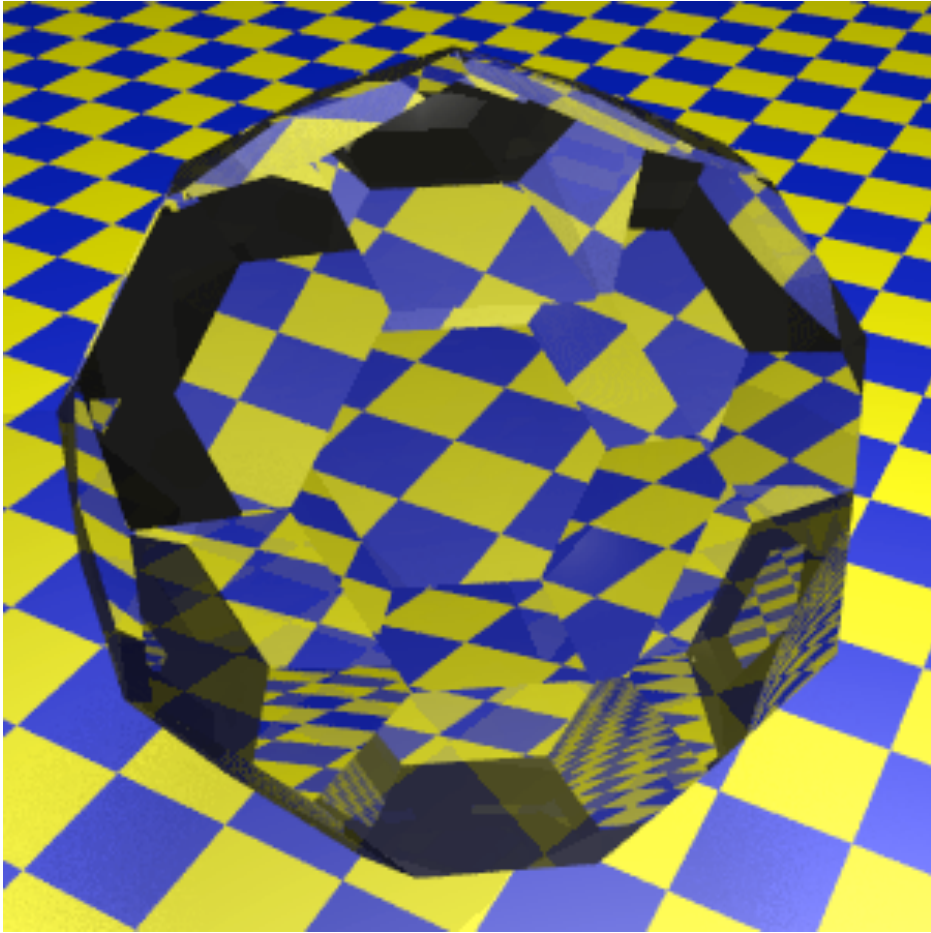
- issues:
 - generation of rays
 - intersection of rays with geometric primitives
 - geometric transformations
 - lighting and shading
 - efficient data structures so we don't have to test intersection with every object

Optimized Ray-Tracing

- basic algorithm simple but **very** expensive
- optimize by reducing:
 - number of rays traced
 - number of ray-object intersection calculations
- methods
 - bounding volumes: boxes, spheres
 - spatial subdivision
 - uniform
 - BSP trees
- (more on this later with collision)



Example Images



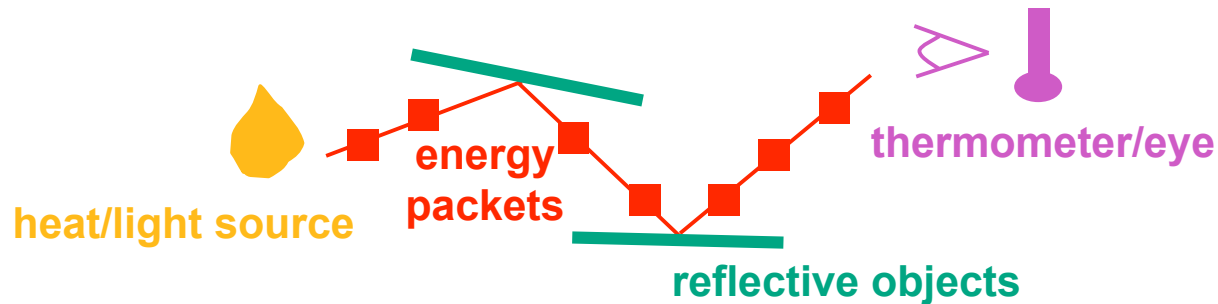
Radiosity

- radiosity definition
 - rate at which energy emitted or reflected by a surface
- radiosity methods
 - capture diffuse-diffuse bouncing of light
 - indirect effects difficult to handle with raytracing



Radiosity

- illumination as radiative heat transfer



- conserve light energy in a volume
 - model light transport as packet flow until convergence
 - solution captures diffuse-diffuse bouncing of light
-
- view-independent technique
 - calculate solution for entire scene offline
 - browse from any viewpoint in realtime

Radiosity

- divide surfaces into small patches
- loop: check for light exchange between all pairs
 - form factor: orientation of one patch wrt other patch ($n \times n$ matrix)



escience.anu.edu.au/lecture/cg/GlobalIllumination/Image/discrete.jpg

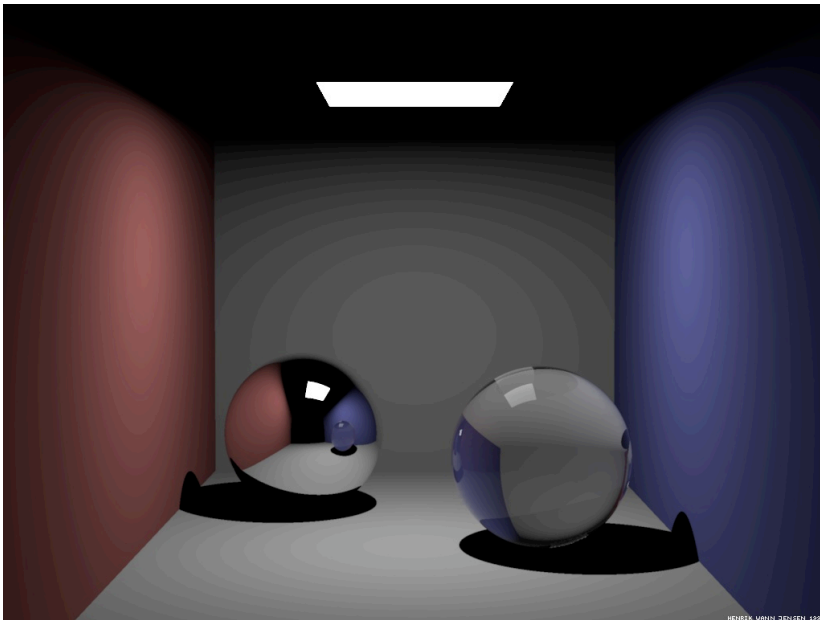


escience.anu.edu.au/lecture/cg/GlobalIllumination/Image/continuous.jpg

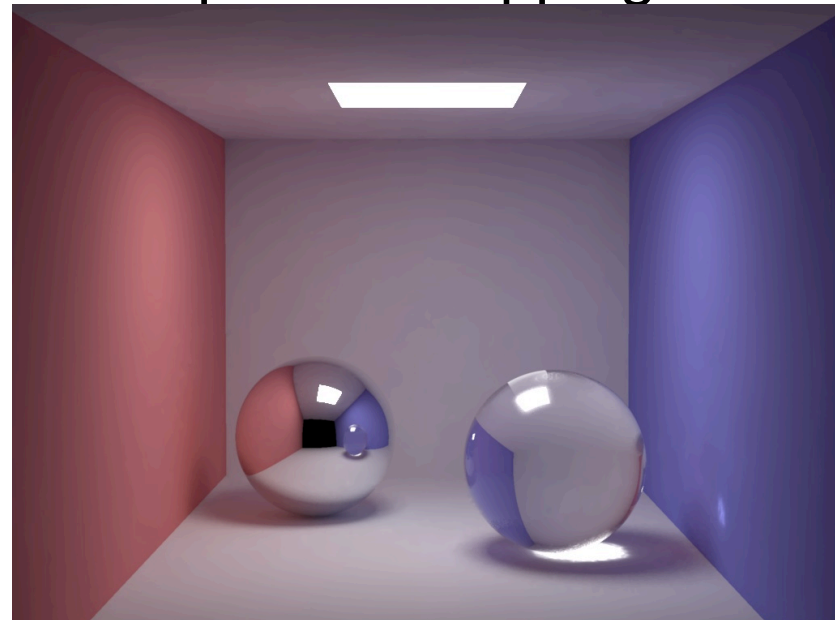
Better Global Illumination

- ray-tracing: great specular, approx. diffuse
 - view dependent
- radiosity: great diffuse, specular ignored
 - view independent, mostly-enclosed volumes
- photon mapping: superset of raytracing and radiosity
 - view dependent, handles both diffuse and specular well

raytracing

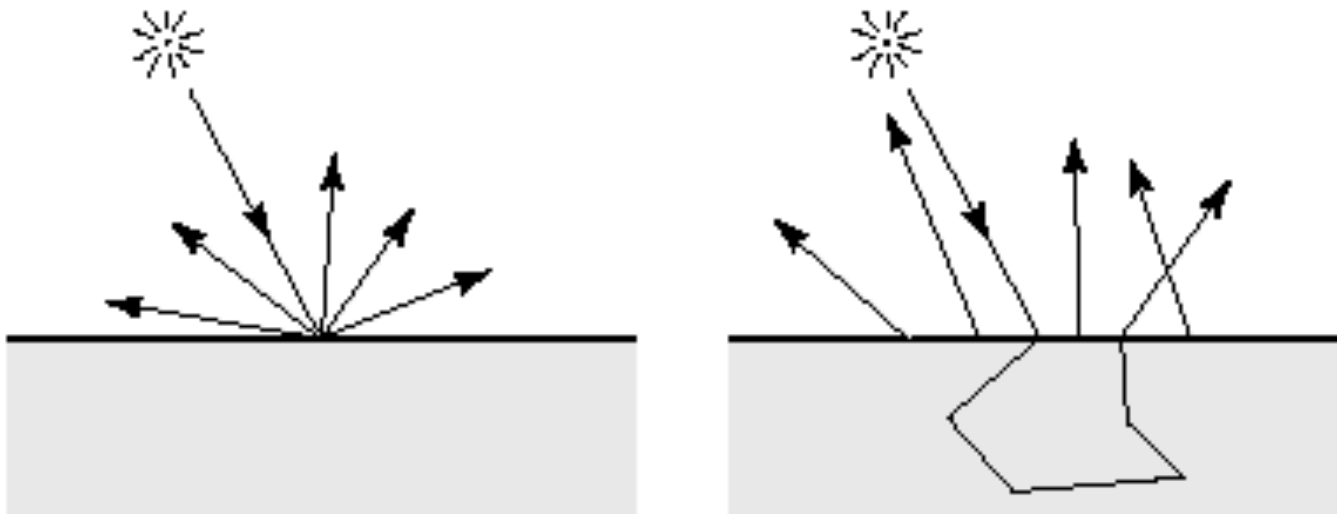


photon mapping



Subsurface Scattering: Translucency

- light enters and leaves at *different* locations on the surface
 - bounces around inside
- technical Academy Award, 2003
 - Jensen, Marschner, Hanrahan



Subsurface Scattering: Marble



Subsurface Scattering: Milk vs. Paint



RENDERED USING DALI - HENRIK WANN JENSEN 2001

Subsurface Scattering: Skin



RENDERED BY HENRIK WANN JENSEN - 2001

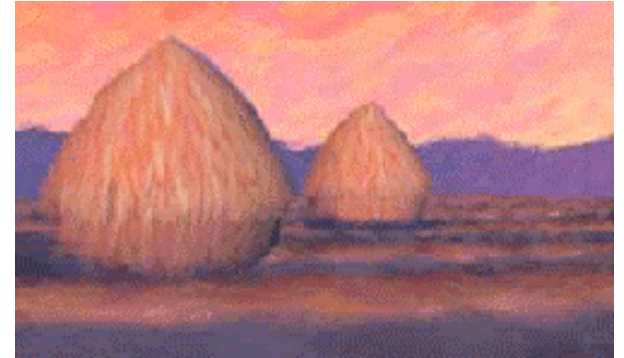
Subsurface Scattering: Skin



RENDERED BY HENRIK WANN JENSEN - 2001

Non-Photorealistic Rendering

- simulate look of hand-drawn sketches or paintings, using digital models



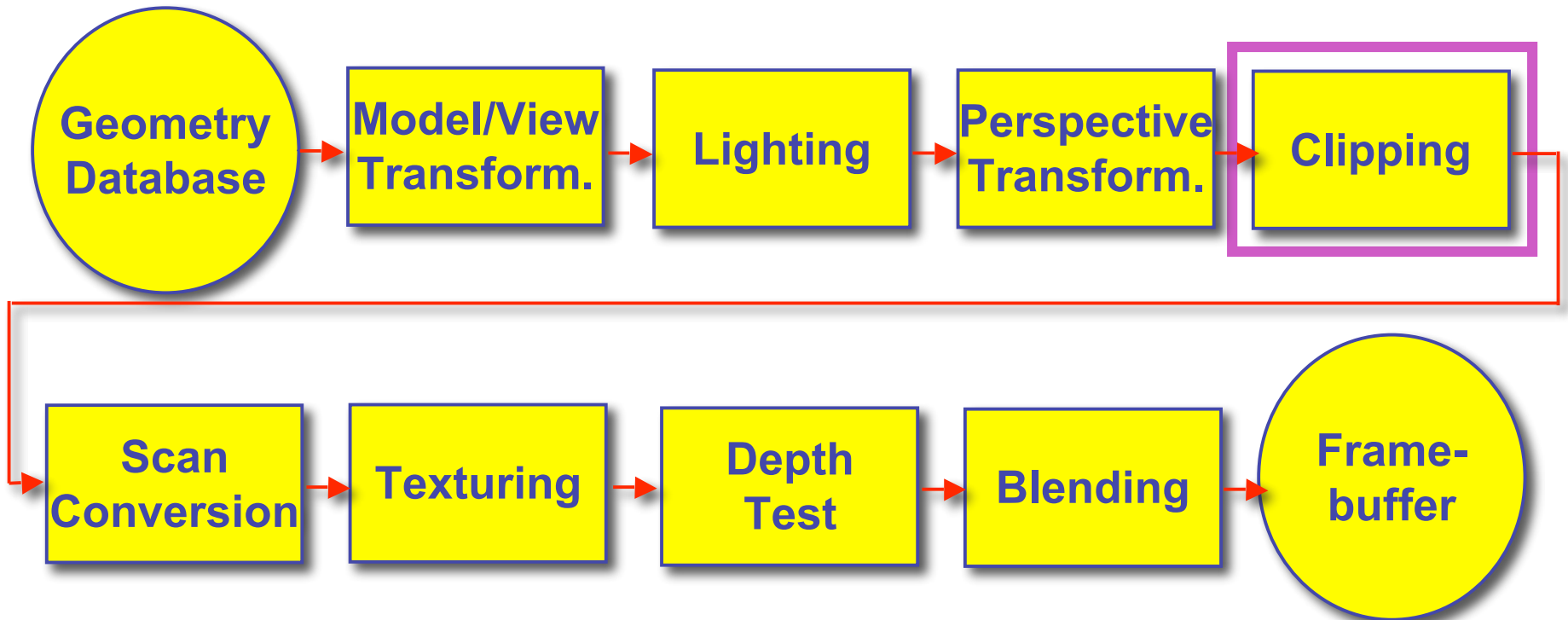
www.red3d.com/cwr/npr/

Clipping

Reading for Clipping

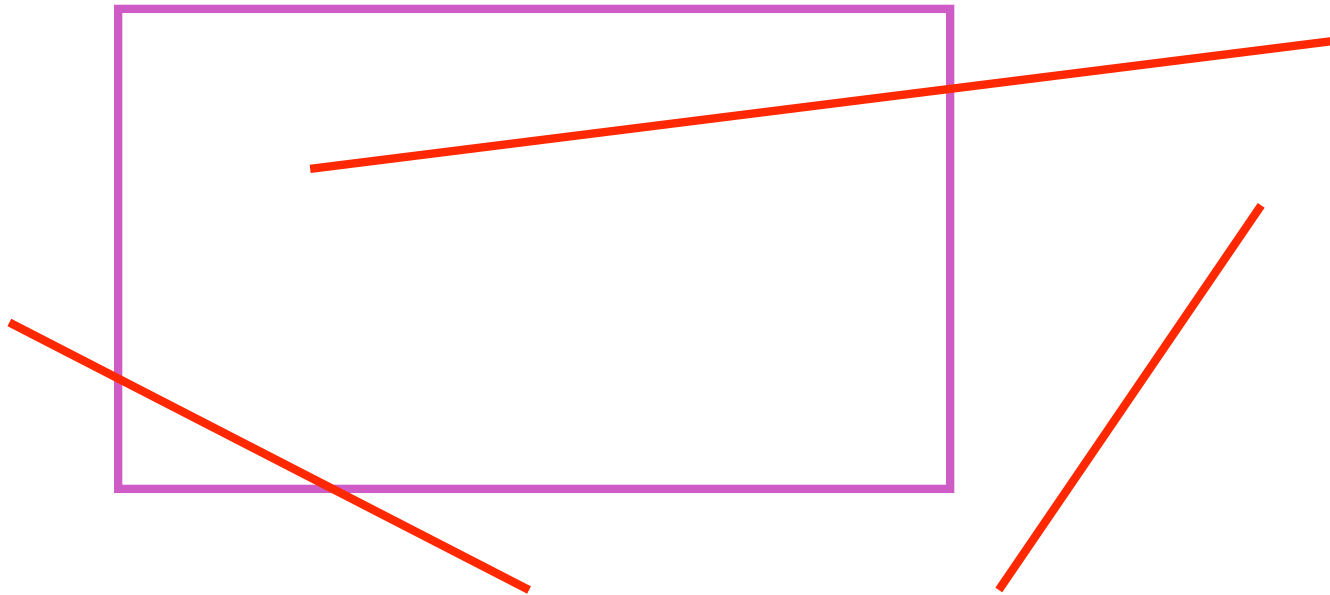
- FCG Sec 8.1.3-8.1.6 Clipping
- FCG Sec 8.4 Culling
 - (12.1-12.4 2nd ed)

Rendering Pipeline



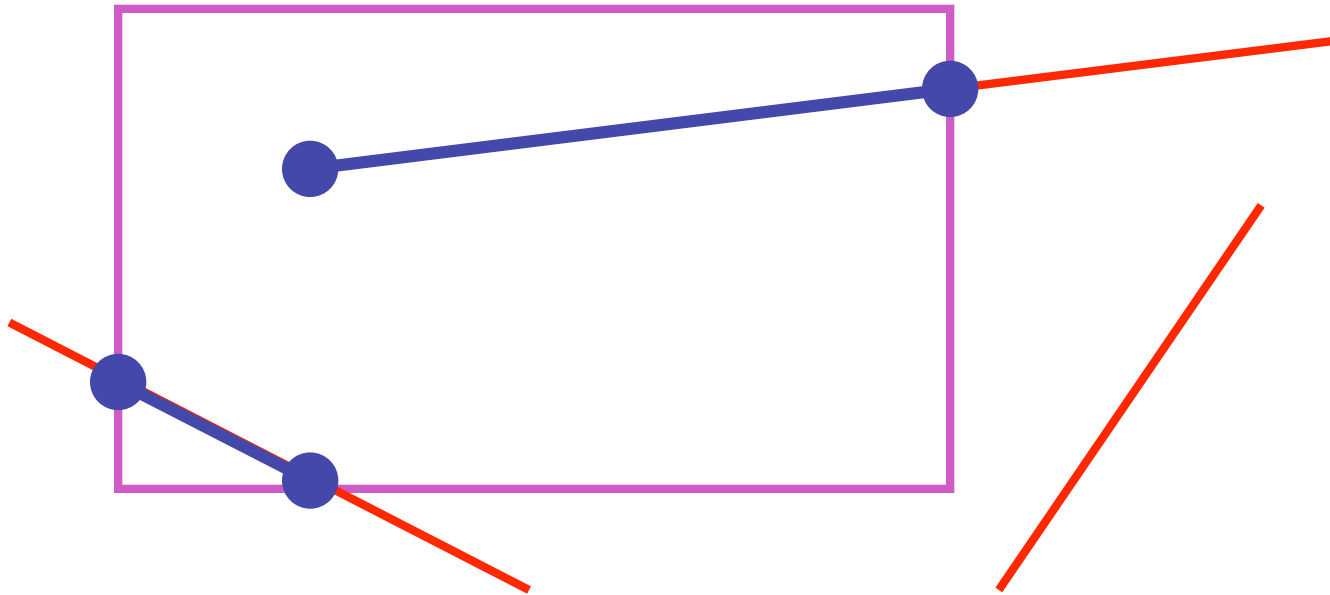
Next Topic: Clipping

- we've been assuming that all primitives (lines, triangles, polygons) lie entirely within the *viewport*
 - in general, this assumption will not hold:



Clipping

- analytically calculating the portions of primitives within the viewport



Why Clip?

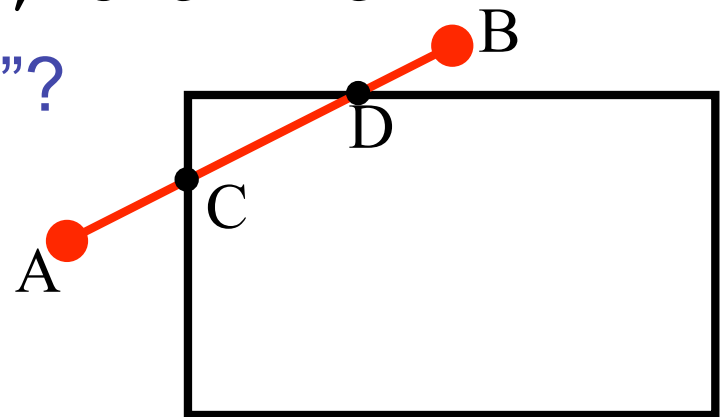
- bad idea to rasterize outside of framebuffer bounds
- also, don't waste time scan converting pixels outside window
 - could be billions of pixels for very close objects!

Line Clipping

- 2D
 - determine portion of line inside an axis-aligned rectangle (screen or window)
- 3D
 - determine portion of line inside axis-aligned parallelepiped (viewing frustum in NDC)
 - simple extension to 2D algorithms

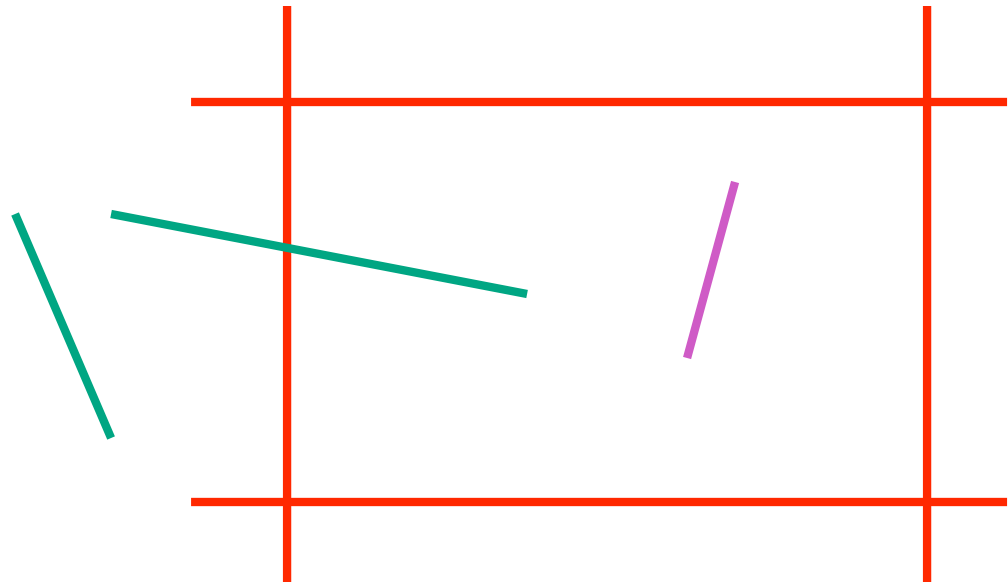
Clipping

- naïve approach to clipping lines:
 - for each line segment
 - for each edge of viewport
 - find intersection point
 - pick “nearest” point
 - if anything is left, draw it
- what do we mean by “nearest”?
- how can we optimize this?



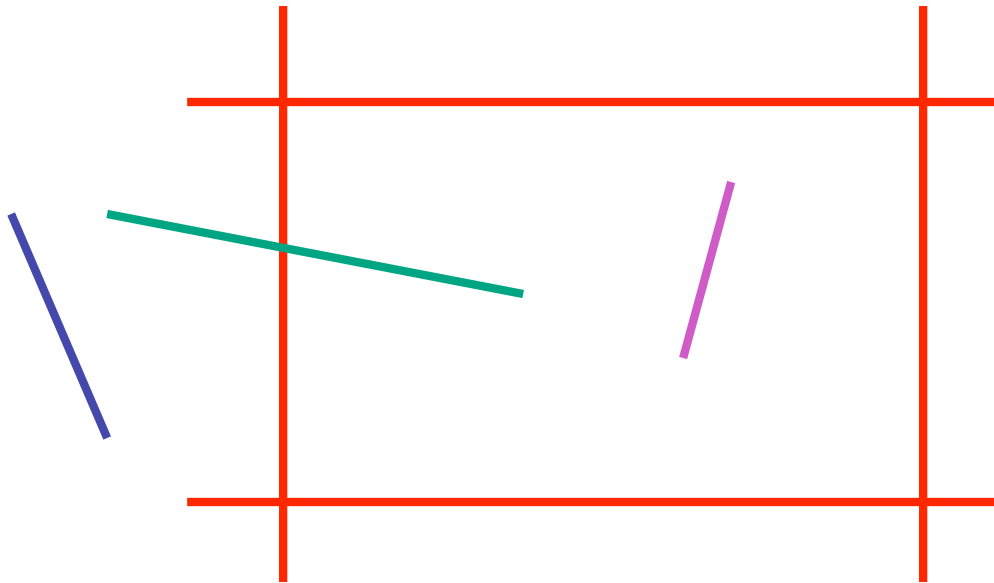
Trivial Accepts

- big optimization: trivial accept/rejects
 - Q: how can we quickly determine whether a line segment is entirely inside the viewport?
 - A: test both endpoints



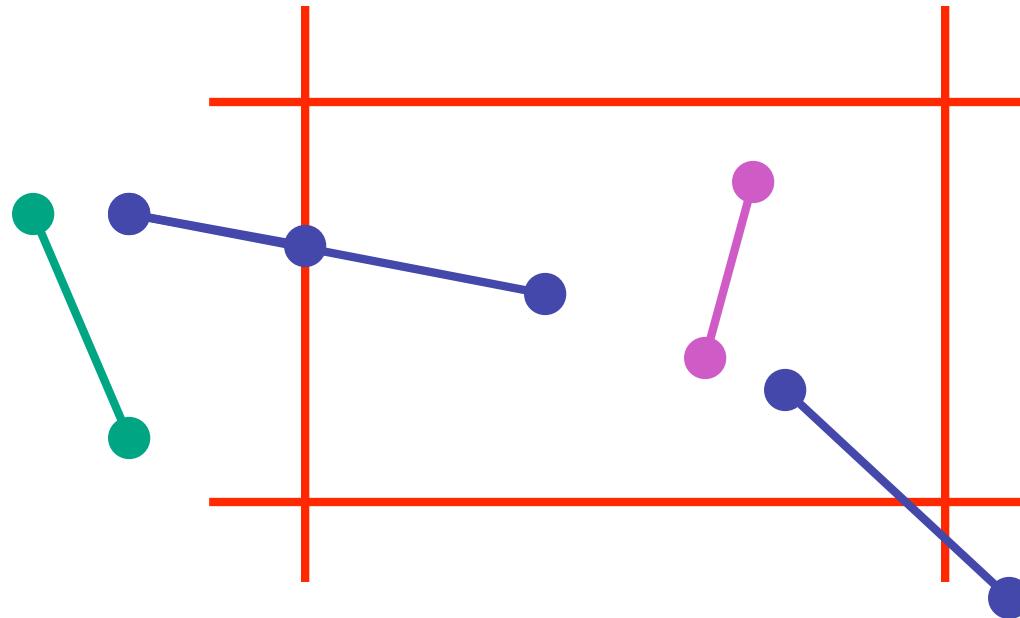
Trivial Rejects

- Q: how can we know a line is outside viewport?
- A: if both endpoints on wrong side of **same** edge, can trivially reject line



Clipping Lines To Viewport

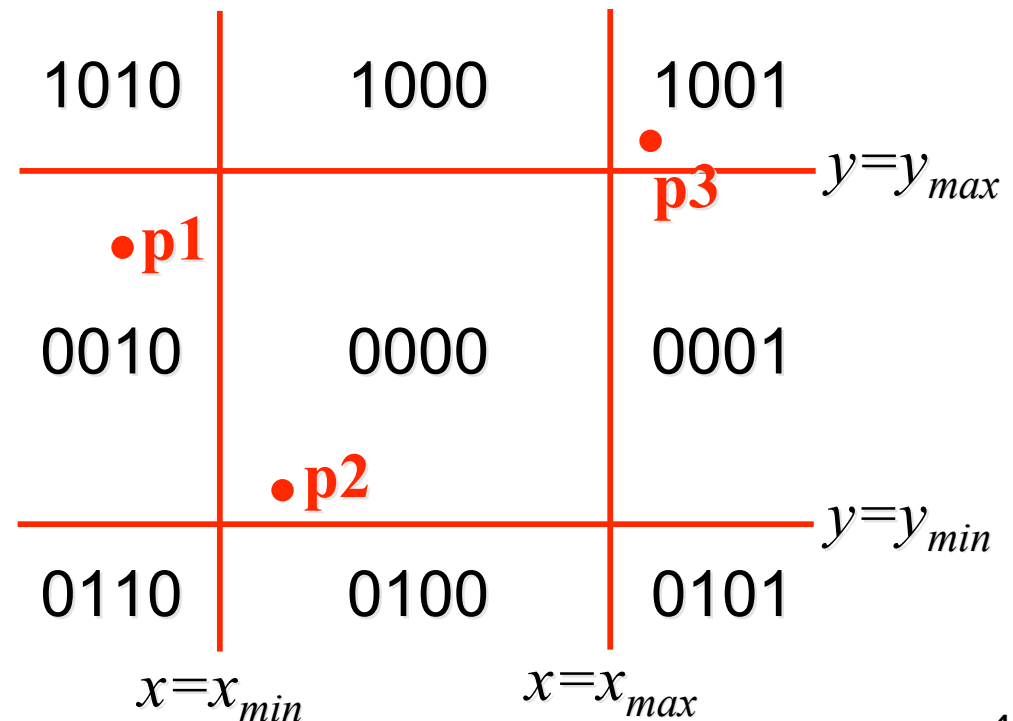
- combining trivial accepts/rejects
 - trivially **accept** lines with both endpoints **inside all edges of the viewport**
 - trivially **reject** lines with both endpoints **outside the same edge of the viewport**
 - otherwise, reduce to trivial cases **by splitting into two segments**



Cohen-Sutherland Line Clipping

- outcodes
 - 4 flags encoding position of a point relative to top, bottom, left, and right boundary

- $OC(p1)=0010$
- $OC(p2)=0000$
- $OC(p3)=1001$



Cohen-Sutherland Line Clipping

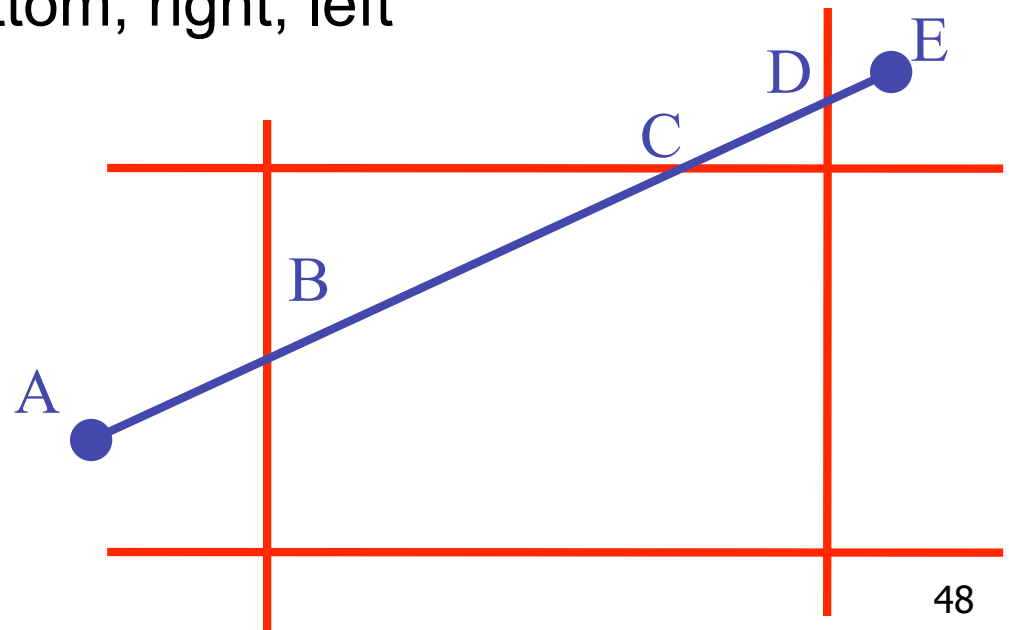
- assign outcode to each vertex of line to test
 - line segment: $(p1, p2)$
- trivial cases
 - $OC(p1) == 0 \ \&\& \ OC(p2) == 0$
 - both points inside window, thus line segment completely visible (trivial accept)
 - $(OC(p1) \ \& \ OC(p2)) \neq 0$
 - there is (at least) one boundary for which both points are outside (same flag set in both outcodes)
 - thus line segment completely outside window (trivial reject)

Cohen-Sutherland Line Clipping

- if line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
- pick an edge that the line crosses (*how?*)
- intersect line with edge (*how?*)
- discard portion on wrong side of edge and assign outcode to new vertex
- apply trivial accept/reject tests; repeat if necessary

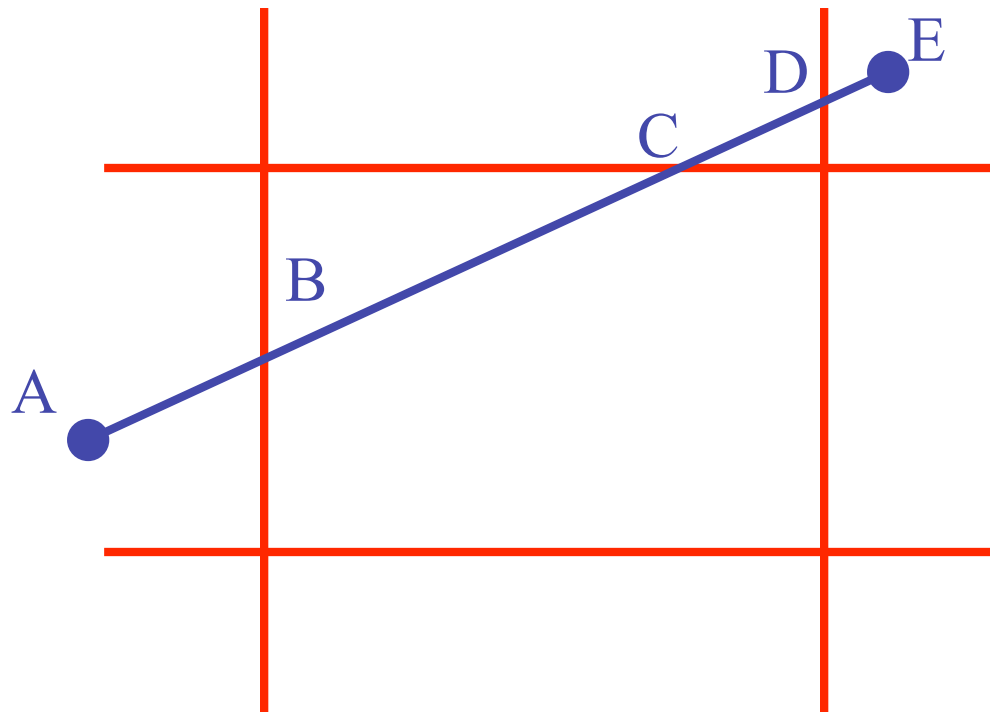
Cohen-Sutherland Line Clipping

- if line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
- pick an edge that the line crosses
 - check against edges in same order each time
 - for example: top, bottom, right, left



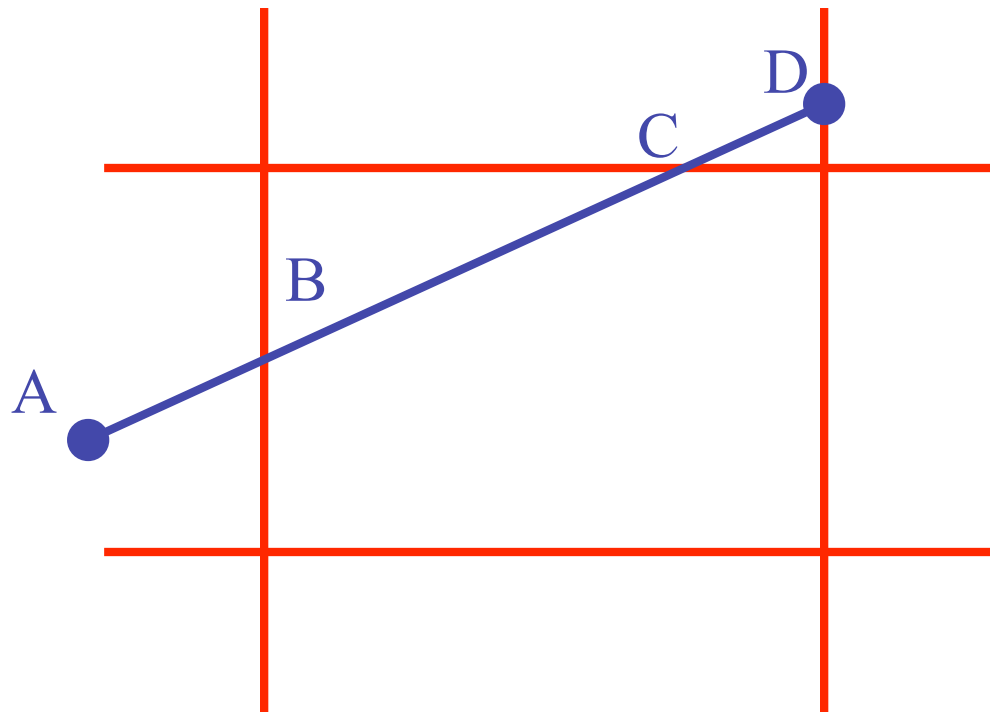
Cohen-Sutherland Line Clipping

- intersect line with edge



Cohen-Sutherland Line Clipping

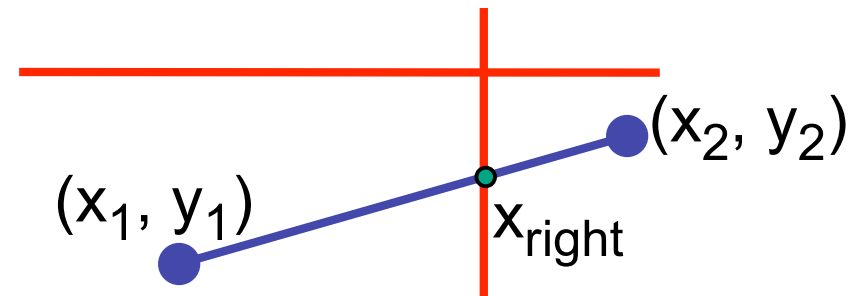
- discard portion on wrong side of edge and assign outcode to new vertex



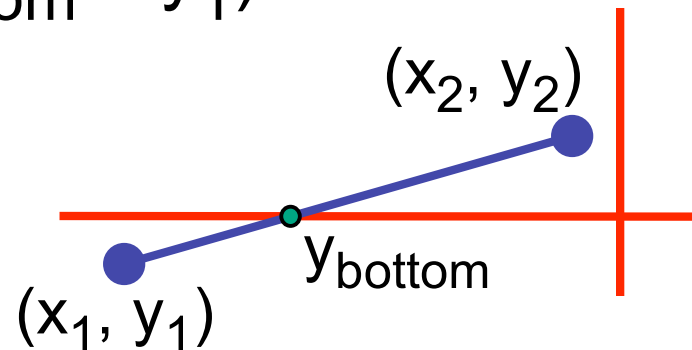
- apply trivial accept/reject tests and repeat if necessary

Viewport Intersection Code

- $(x_1, y_1), (x_2, y_2)$ intersect vertical edge at x_{right}
 - $y_{\text{intersect}} = y_1 + m(x_{\text{right}} - x_1)$
 - $m = (y_2 - y_1) / (x_2 - x_1)$



- $(x_1, y_1), (x_2, y_2)$ intersect horiz edge at y_{bottom}
 - $x_{\text{intersect}} = x_1 + (y_{\text{bottom}} - y_1) / m$
 - $m = (y_2 - y_1) / (x_2 - x_1)$



Cohen-Sutherland Discussion

- key concepts
 - use opcodes to quickly eliminate/include lines
 - best algorithm when trivial accepts/rejects are common
 - must compute viewport clipping of remaining lines
 - non-trivial clipping cost
 - redundant clipping of some lines
- basic idea, more efficient algorithms exist

Line Clipping in 3D

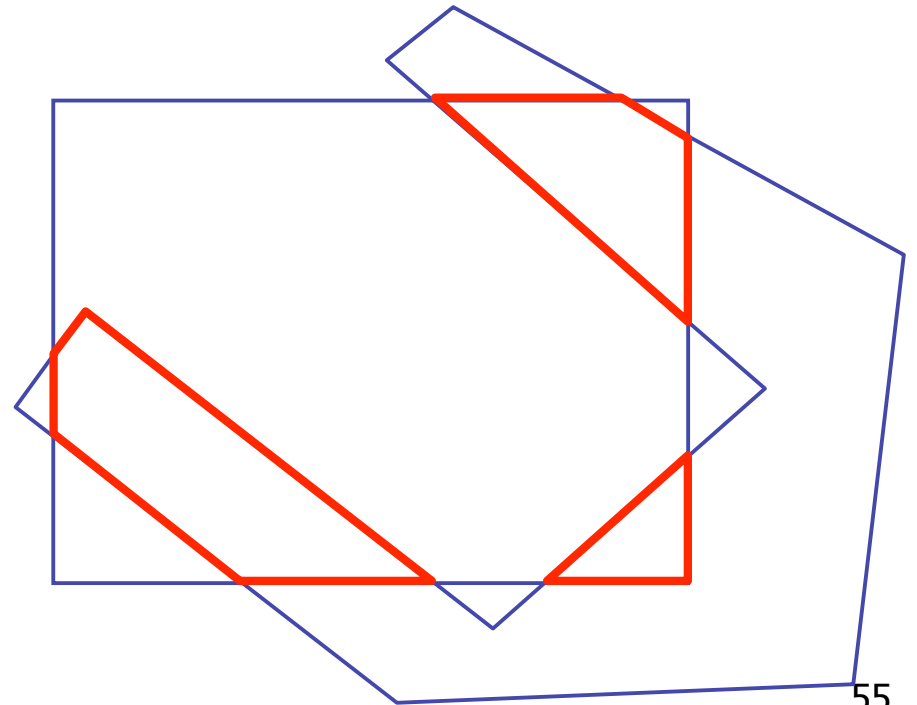
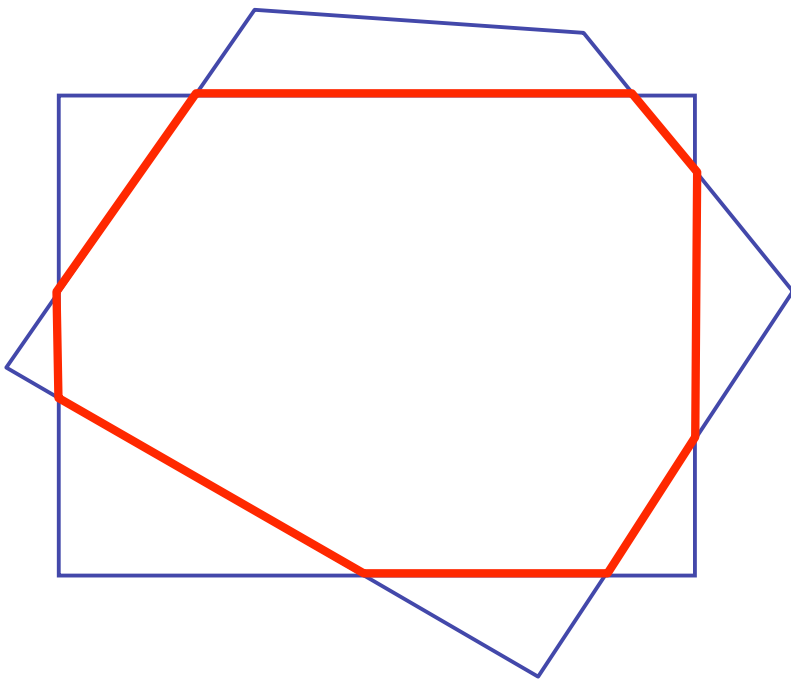
- approach
 - clip against parallelepiped in NDC
 - after perspective transform
 - means that clipping volume always the same
 - $x_{\min}=y_{\min}= -1$, $x_{\max}=y_{\max}= 1$ in OpenGL
- boundary lines become boundary planes
 - but outcodes still work the same way
 - additional front and back clipping plane
 - $z_{\min} = -1$, $z_{\max} = 1$ in OpenGL

Polygon Clipping

- objective
 - 2D: clip polygon against rectangular window
 - or general convex polygons
 - extensions for non-convex or general polygons
 - 3D: clip polygon against parallelepiped

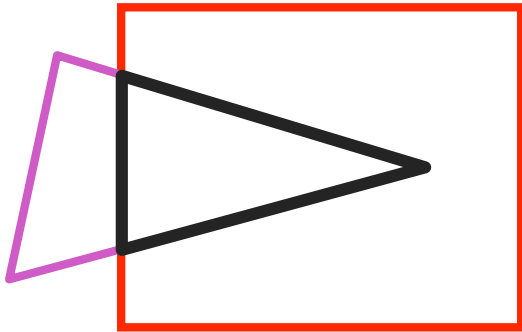
Polygon Clipping

- not just clipping all boundary lines
 - may have to introduce new line segments

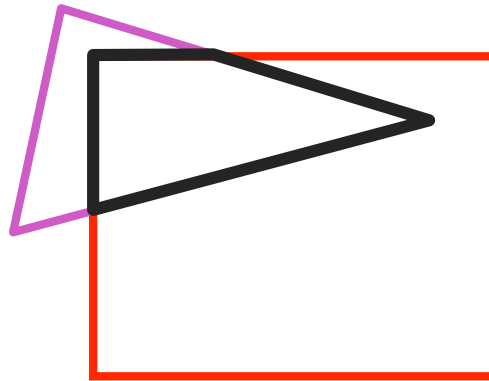


Why Is Clipping Hard?

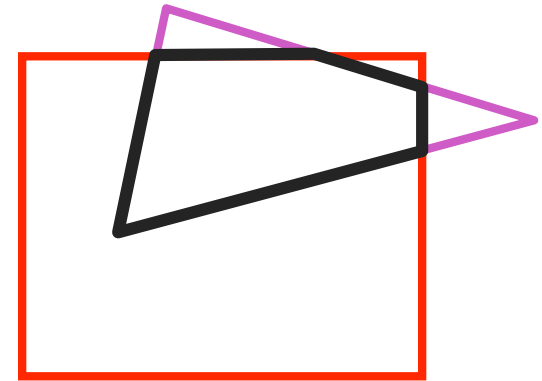
- what happens to a triangle during clipping?
 - some possible outcomes:



triangle to triangle

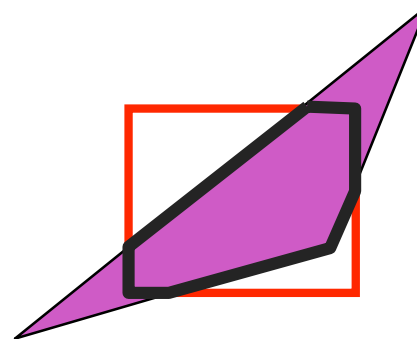


triangle to quad



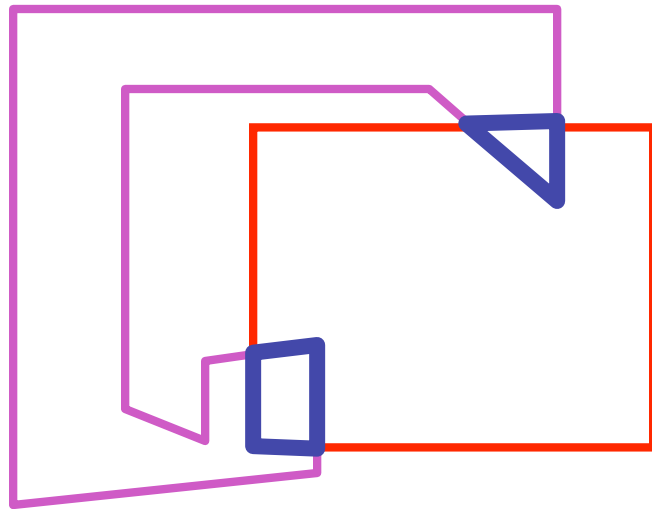
triangle to 5-gon

- how many sides can result from a triangle?
 - seven



Why Is Clipping Hard?

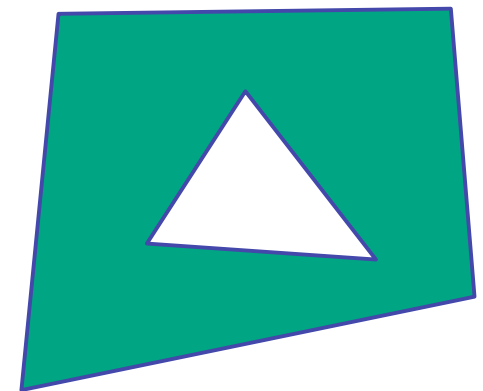
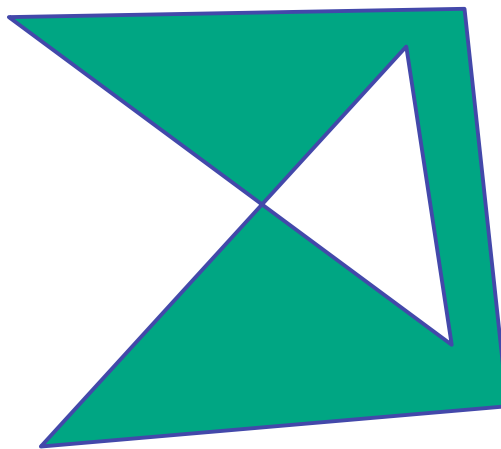
- a really tough case:



concave polygon to multiple polygons

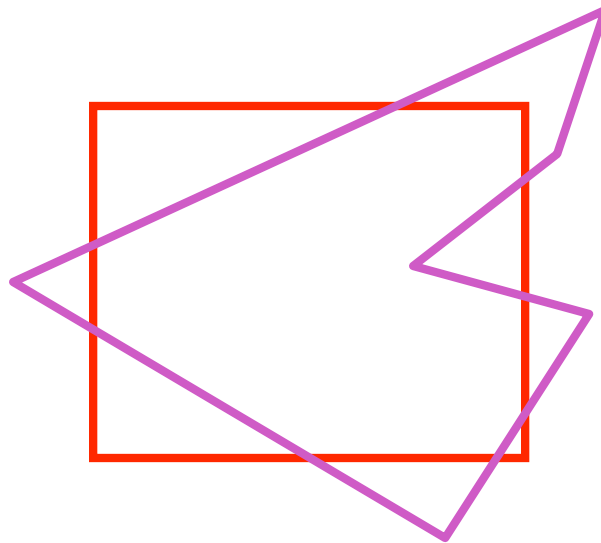
Polygon Clipping

- classes of polygons
 - triangles
 - convex
 - concave
 - holes and self-intersection



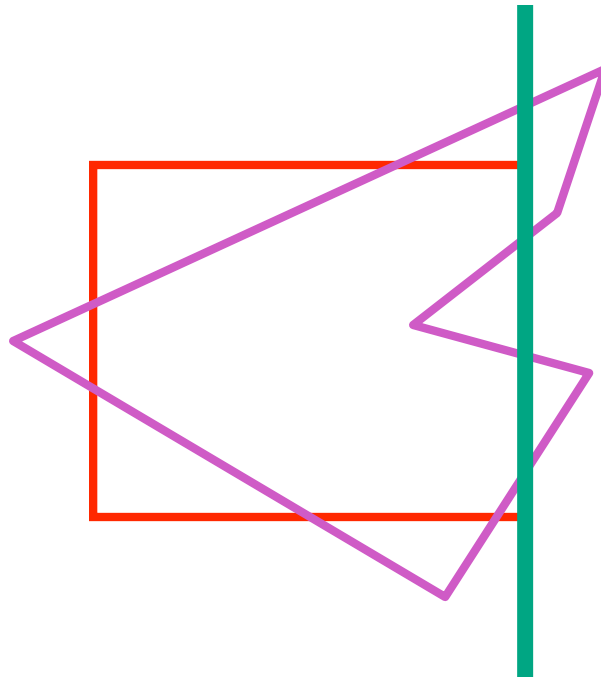
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



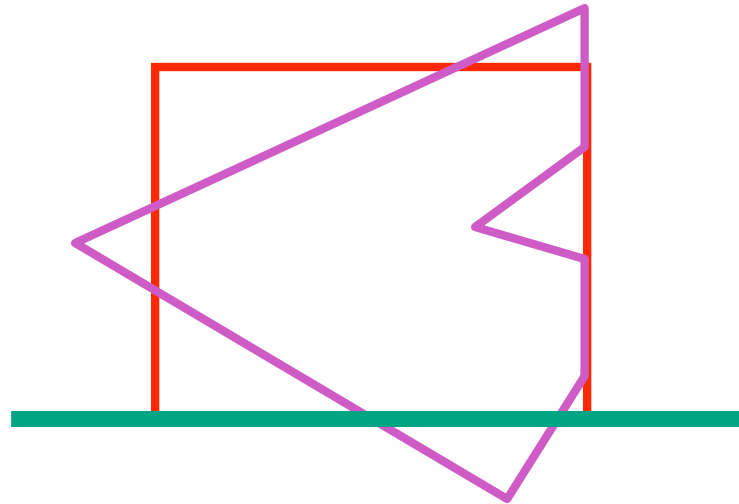
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



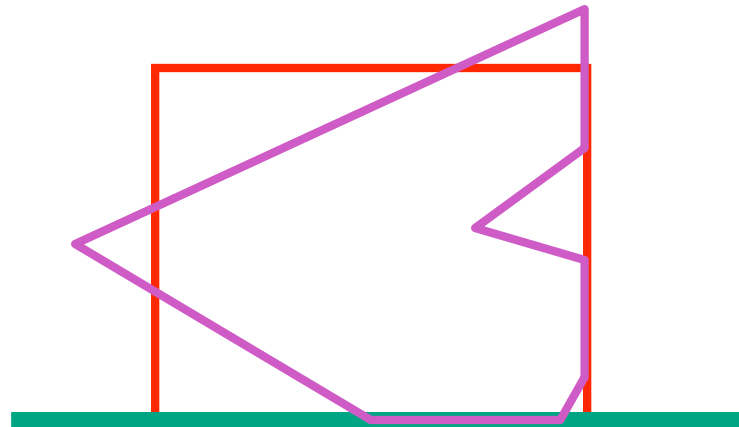
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



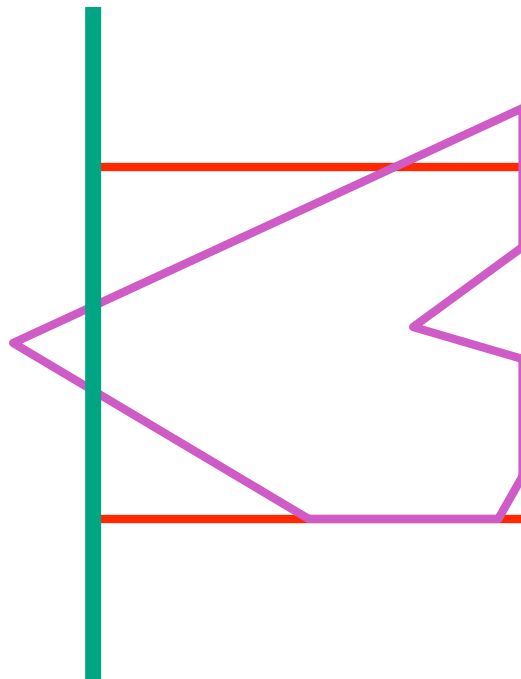
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



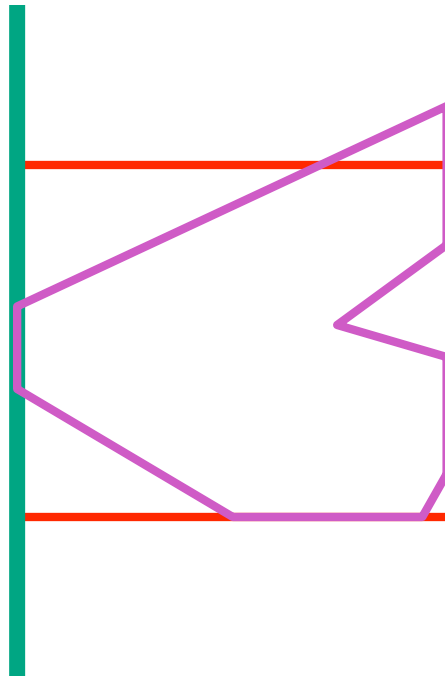
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



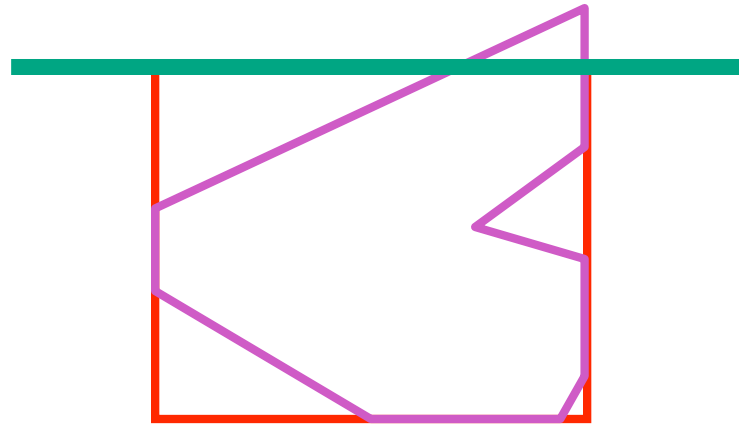
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



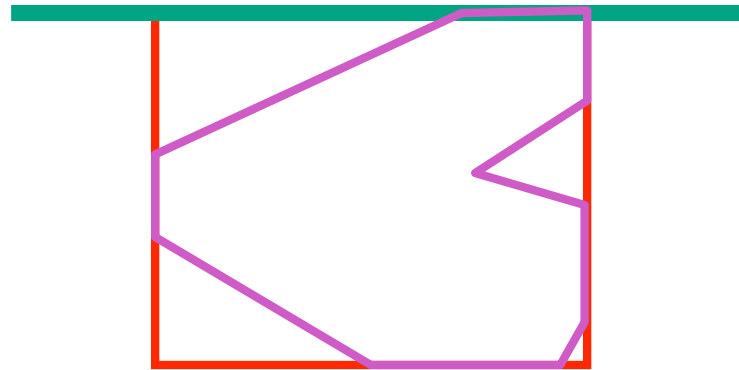
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



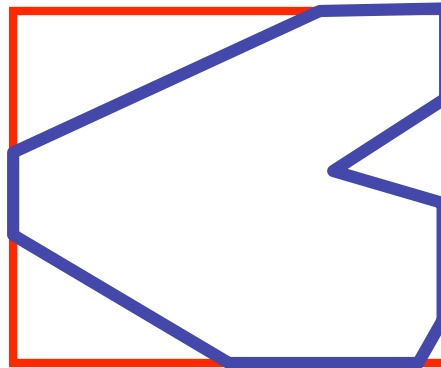
Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped



Sutherland-Hodgeman Clipping

- basic idea:
 - consider each edge of the viewport individually
 - clip the polygon against the edge equation
 - after doing all edges, the polygon is fully clipped

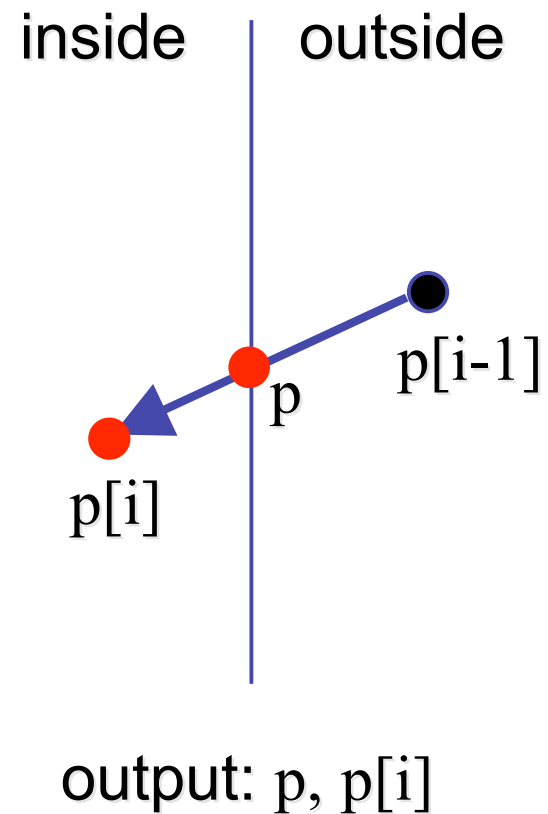
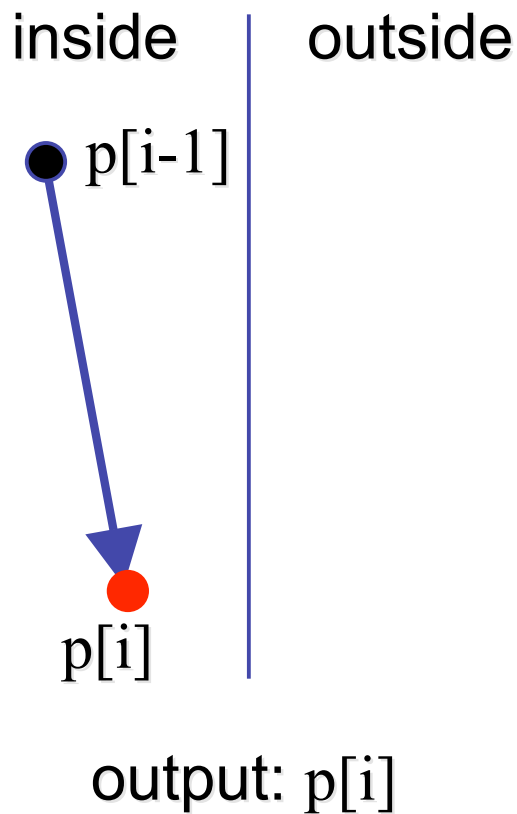


Sutherland-Hodgeman Algorithm

- input/output for whole algorithm
 - input: list of polygon vertices in order
 - output: list of clipped polygon vertices consisting of old vertices (maybe) and new vertices (maybe)
- input/output for each step
 - input: list of vertices
 - output: list of vertices, possibly with changes
- basic routine
 - go around polygon one vertex at a time
 - decide what to do based on 4 possibilities
 - is vertex inside or outside?
 - is previous vertex inside or outside?

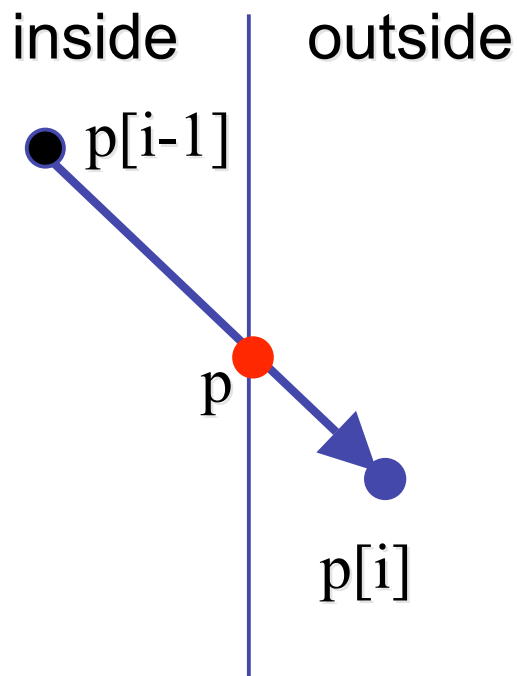
Clipping Against One Edge

- $p[i]$ inside: 2 cases

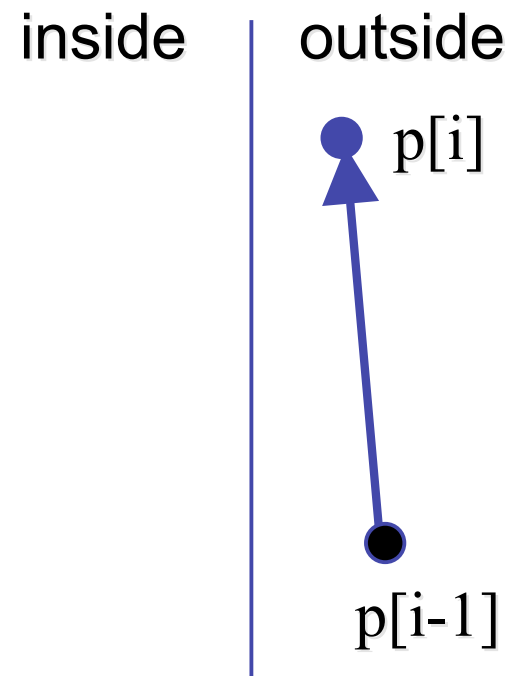


Clipping Against One Edge

- $p[i]$ outside: 2 cases



output: p

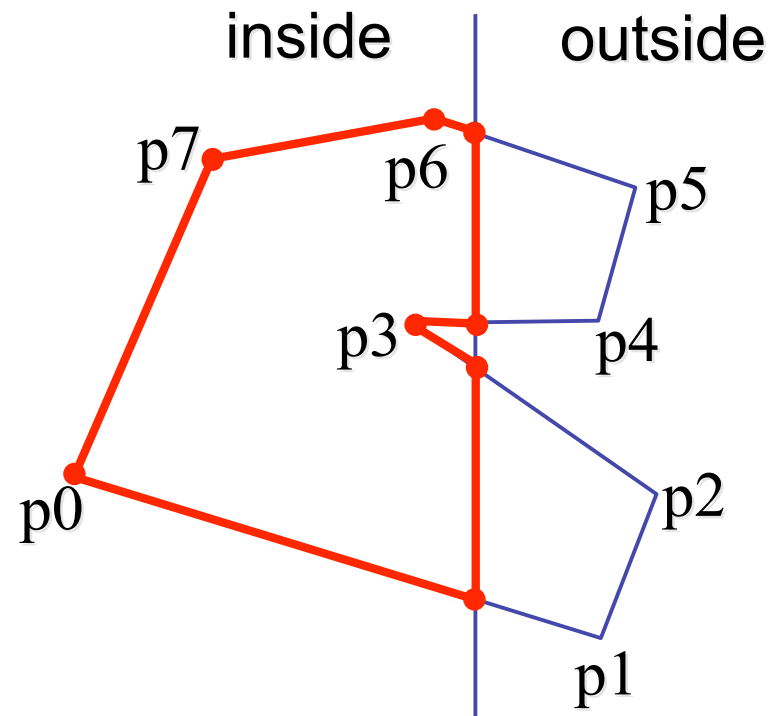


output: nothing

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 ) output p[i];    // p[-1]= p[n-1]  
            else {  
                p= intersect( p[i-1], p[i], edge ); output p, p[i];  
            }  
        } else {                                     // p[i] is outside edge  
            if( p[i-1] inside edge ) {  
                p= intersect(p[i-1], p[i], edge ); output p;  
            }  
        }  
    }  
}
```

Sutherland-Hodgeman Example



Sutherland-Hodgeman Discussion

- similar to Cohen/Sutherland line clipping
 - inside/outside tests: outcodes
 - intersection of line segment with edge: window-edge coordinates
- clipping against individual edges independent
 - great for hardware (pipelining)
 - all vertices required in memory at same time
 - not so good, but unavoidable
 - another reason for using triangles only in hardware rendering