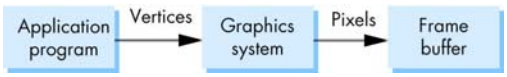
Universität Bielefeld

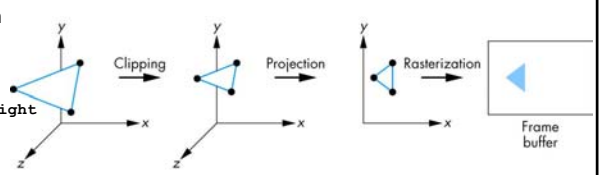
Realtime 3D Computer Graphics Virtual Reality

From Vertices to Fragments

Overview



- Overall goal recapitulation:
  - Input: World description, e.g., set of vertices and states for objects, attributes, camera,…
  - Output: Array of colored pixels in framebuffer.
- After transformation and projection (with possible normalization) we have to
  1. decide visibility of objects:
     - which objects will be seen due to frustum and output window:
       - **Culling:** in object or world space
       - **Clipping:** in canonical view or image space
     - which objects will be in front of others (depth-sorting):
       - **Occlusion culling**, **depth sort:** in object or world space
       - **z-buffer algorithm, w-buffer algorithm:** in canonical view or image space
  2. draw the remaining primitives; map from continuous space to discrete space
     - **Scan conversion, rasterization**

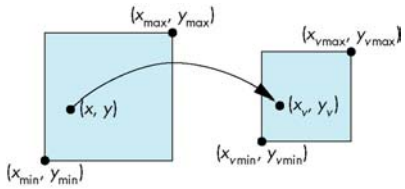


- Two different possibilities:

```
for(each_object) render(object); //see right
for(each_pixel) assign_a_color(pixel);
```

Tasks

- Modelling
- Geometry processing
  - Which objects appear on the screen
  - Assign color shades to vertices
  - Primitive assembly and clipping
  - Perspective division
- Rasterization
  - Maps from NDC* to window coords (see right):
  - Lines: Which fragments should be used to approximate
  - Polygons: Which pixels are inside
  - Assign color to fragments
- Fragment processing
  - Combine with image path (texturing)
  - Blending
  - Anti-aliasing



$$x_v = x_{v\min} + \frac{x+1.0}{2.0}(x_{v\max} - x_{v\min})$$

$$y_v = y_{v\min} + \frac{y+1.0}{2.0}(y_{v\max} - y_{v\min})$$

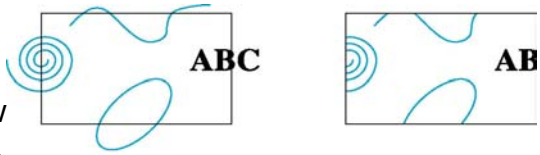
$$z_v = z_{v\min} + \frac{z+1.0}{2.0}(z_{v\max} - z_{v\min})$$

*NDC: All vertices lie in the cube with $-w \le a \le w, a \in \{x, y, z\}$

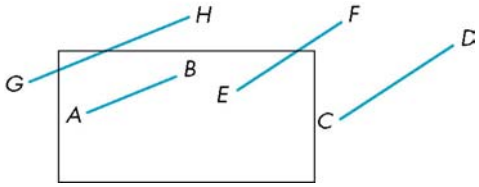
Realtime 3D Computer Graphics
Virtual Reality

Clipping

Clipping



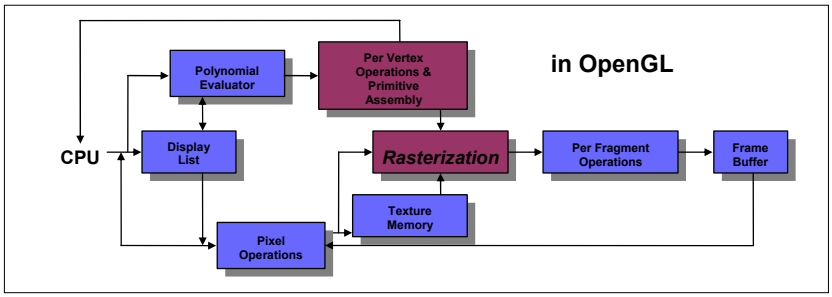
- 2D against clipping window
- 3D against clipping volume
- Easy for line segments/polygons
- Hard for curves and text
  - Convert to lines and polygons first



- Clipping 2D Line Segments:
  - Brute force approach:
    - compute intersections with all sides of clipping window
    - Inefficient: one division per intersection

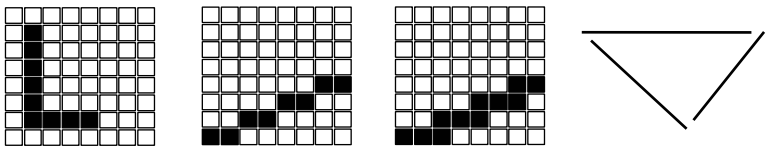
Line processing

- Lines:
  - Most common 2D primitive - done 100s or 1000s of times each frame, even 3D wireframes are eventually 2D lines!
  - Lines are *compatible* with vector displays but nowadays most displays are raster displays. Any render stage before viz might need discretization.
  - Optimized algorithms contain numerous tricks/techniques that help in designing more advanced algorithms for line processing.



Line Requirements

- Must compute integer coordinates of pixels which lie on or near a line or circle.
- Pixel level algorithms are invoked hundreds or thousands of times when an image is created or modified – must be fast!
- Lines must create visually satisfactory images.
  - Lines should appear straight
  - Lines should terminate accurately
  - Lines should have constant density
- Line algorithm should always be defined.

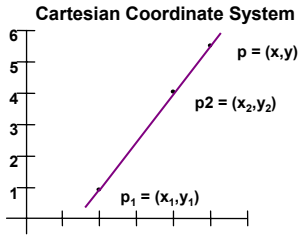


Basic Math Review

- **Point-slope formula for a Line:**
  - Given two points (x_1,y_1), (x_2, y_2)
  - Consider a third point on the line:
    - $p = (x,y)$
    - Slope $= (y_2 - y_1)/(x_2 - x_1)$
      $= (y - y_1)/(x - x_1)$
  - Solving For y
    - $y = [(y_2-y_1)/(x_2-x_1)]*(x-x_1)+ y_1$
  - or, plug in the point (0, b) to get the **Slope-intercept form:**
    - **$y = mx + b$**
- **Length of line segment between p_1 and p_2:**
  - $l = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- **Midpoint of a line segment between p_1 and p_3:**
  - $p_2 = ((x_1+x_3)/2 , (y_1+y_3)/2)$
- Two lines are **perpendicular** iff
  - 1) $m_1 = -1/m_2$
  - 2) Cosine of the angle between them is 0.

- **Parametric form**
  - Given points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
    - $x = x_1 + \alpha(x_2-x_1)$, $y = y_1 + \alpha(y_2-y_1)$
    - $p(\alpha)=(1- \alpha)p_1+ \alpha p_2$
      α is called the parameter. When $\alpha = 0$ we get (x_1,y_1), $\alpha = 1$ we get (x_2,y_2)
  - As $0 < \alpha < 1$ we get all the other points on the line segment between (x_1,y_1) and (x_2,y_2).

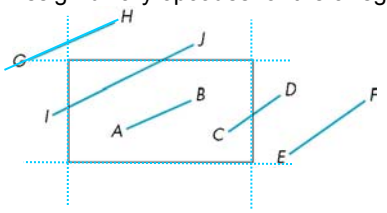
Cartesian Coordinate System



$$\text{slope} = \frac{\text{rise}}{\text{run}} = \frac{y_2-y_1}{x_2-x_1}$$

Cohen-Sutherland Algorithm

- **Idea:**
  - Eliminate as many cases as possible without computing intersections
  - Start with four lines that determine the sides of the clipping window
  - Assign binary opcodes for the 9 regions for fast processing: $b_0b_1b_2b_3$



1001	1000	1010
0001	0000	0010
0101	0100	0110

$y = y_{max}$

$y = y_{min}$

$x = x_{min}$ $x = x_{max}$

- Computation of outcode requires at most 4 subtractions:

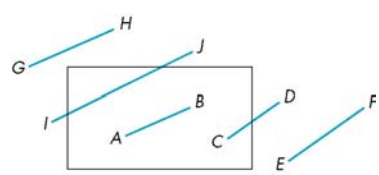
  b_0 = 1 if $y > y_{max}$, 0 otherwise
  b_1 = 1 if $y < y_{min}$, 0 otherwise
  b_2 = 1 if $x > x_{max}$, 0 otherwise
  b_3 = 1 if $x < x_{min}$, 0 otherwise

```
GH = [0001,1000]
IJ = [0001,1000]
AB = [0000,0000]
CD = [0000,0010]
EF = [0010,0010]
```


Cohen-Sutherland Algorithm

- Compute opcodes for each line segment s.
- Given: line segment by two opcodes $o_1(s)$, $o_2(s)$.

```
procedure cs-clip(segment s){
   if (o1(s) OR o2(s) = 0)
    raterize(s);
   else if (o1(s) AND o2(s) != 0)
    reject(s);
   else
    cs-clip( clipseg (s) );
}
function clipseg (s){
   // Line can be determined by opcode
   line l = find_line_by_opcode (s);
   return shortenseg (s,l);
}
```



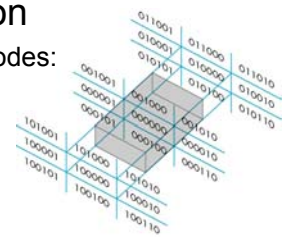

```
GH = [0001,1000]
IJ = [0001,1000]
AB = [0000,0000]
CD = [0000,0010]
EF = [0010,0010]
```

Cohen-Sutherland

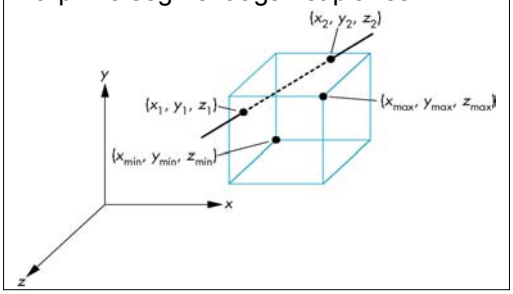
- Efficiency:
  - Often, clipping window is small relative to the size of the entire data base
  - Most segments are outside one or more window sides and can be eliminated based on their outcodes
  - Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

3D-extension

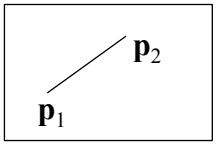
- Use 6-bit outcodes:



- When needed, clip line segment against planes:



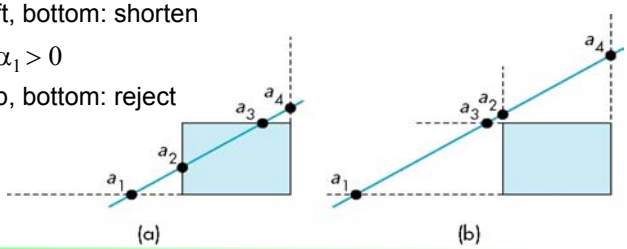
Liang-Barsky Clipping



- Consider the parametric form of a line segment

$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2 \quad 1 \geq \alpha \geq 0$$

- Distinguish between cases
  - based on ordering of the values of α
  - where the line determined by the line segment crosses the lines that determine the window
- In (a): $1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1 > 0$
  - Intersect right, top, left, bottom: shorten
- In (b): $1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1 > 0$
  - Intersect right, left, top, bottom: reject



Liang-Barsky Clipping

1. Avoid computing intersection as long as possible.
2. Rejection can be done on ordering.
3. Using parametric form, it follows: $\alpha = \frac{y_{\max} - y_1}{y_2 - y_1}$
   (similar for the other 3 sides)
   - This can be rewritten as: $\alpha = (y_2 - y_1) = \alpha\Delta y = y_{\max} - y_1 = \Delta y_{\max}$
   - Restate decisions in terms of $\Delta y_{\max}, \Delta y$
   - avoid floating-point execution during decision until intersection is actually required.

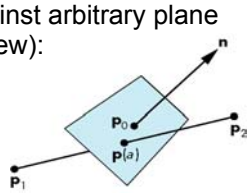
Advantages:

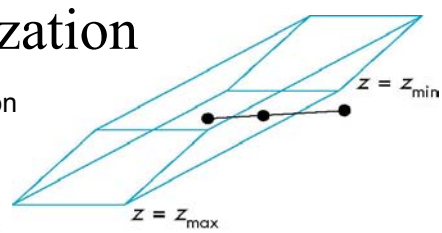
- Can accept/reject as easily as with Cohen-Sutherland
- Using values of α, we do not have to use algorithm recursively as with C S
- Extends to 3D

Clipping and Normalization

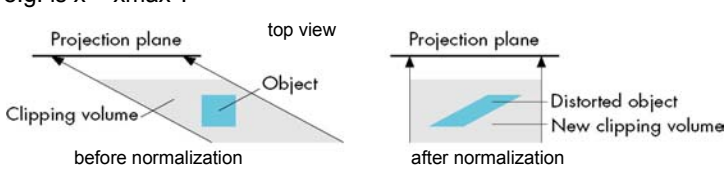
- General clipping in 3D requires intersection of line segments against arbitrary plane (Example: oblique view):

$$a = \frac{n \bullet (p_o - p_1)}{n \bullet (p_2 - p_1)}$$

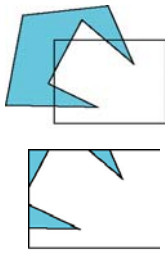




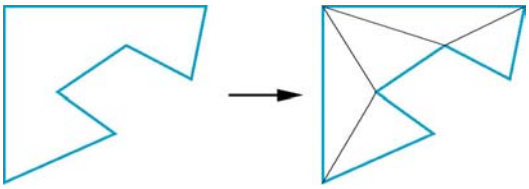
- Normalization is part of viewing (pre clipping)but after normalization, we clip against sides of right parallelepiped
  - Typical intersection calculation now requires only a floating point subtraction, e.g. is x > xmax ?



Polygon Clipping

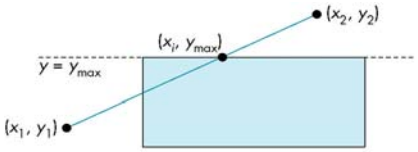


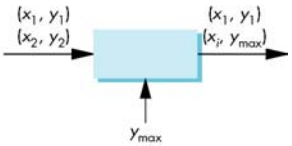
- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons
  - However, clipping a convex polygon can yield at most one other polygon
- One strategy is to replace nonconvex (concave) polygons with a set of triangular polygons (a tessellation)
- Also makes fill easier (e.g., tessellation code available in GLU library)



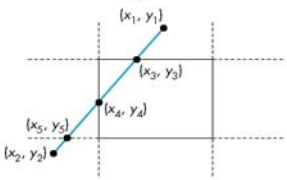
Clipping pipeline

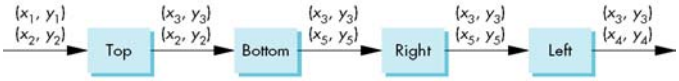
- Consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment:



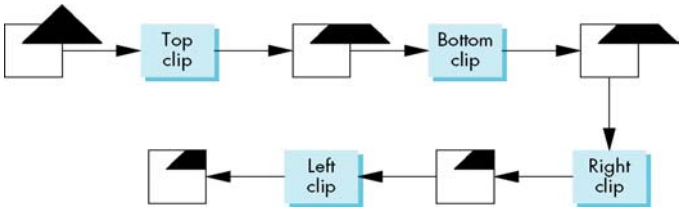


- Clipping against window sides is independent of other sides
- Place independent clippers in a pipeline:





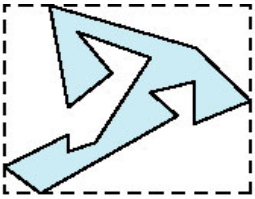
Pipeline Clipping of Polygons



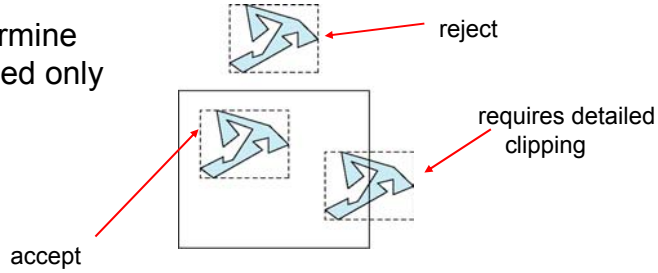
- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

Clipping Bounding Boxes

- Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*
  - Smallest rectangle aligned with axes that encloses the polygon
  - Simple to compute: max and min of x and y



- Can usually determine accept/reject based only on bounding box



Realtime 3D Computer Graphics Virtual Reality

Rasterization – Scan conversion

Rasterization

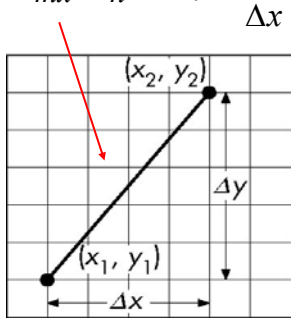
- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties

Scan Conversion of Line Segments

- Start with segment in window coordinates with integer values for endpoints
- Assume implementation has a **`write_pixel`** function
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line $y=mx+h$ satisfies differential equation

    $dy/dx = m = \Delta y/\Delta x = y_2-y_1/x_2-x_1$
- Along scan line $\Delta x = 1$

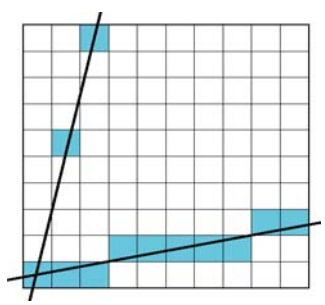
$$y = mx + h \qquad m = \frac{\Delta y}{\Delta x}$$



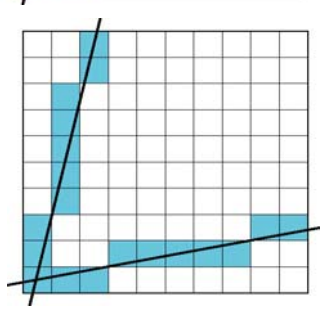

```
For(x=x1; x<=x2,ix++) {
   y+=m;
  write_pixel(x, round(y), line_color)
}
```


Problem

- DDA = for each x plot pixel at closest y
  - Problems for steep lines



- Solution: Using symmetry
  - Use for $1 \geq m \geq 0$
  - For $m > 1$, swap role of x and y
    - For each y, plot closest x



Simple DDA Line Algorithm

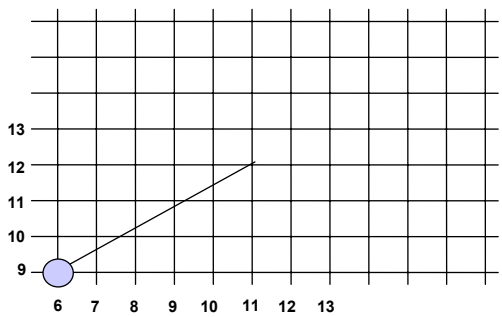
```
void DDA(int X1,Y1,X2,Y2)
{
   int    Length, I;
   float X,Y,Xinc,Yinc;

   Length = ABS(X2 - X1);
   if (ABS(Y2 - Y1)  > Length)
        Length = ABS(Y2-Y1);
   Xinc = (X2 - X1)/Length;
   Yinc = (Y2 - Y1)/Length;
   X = X1;
   Y = Y1;
   while(X<X2){

   write_pixel(Round(X),Round(Y);
                line_color);
      X = X + Xinc;
      Y = Y + Yinc;
   }
}
```

DDA creates good lines but it is too time consuming due to the round function and long operations on real values.

DDA Example

Compute which pixels should be turned on to represent the line from (6,9) to (11,12).

Length = ?
Xinc = ?
Yinc = ?

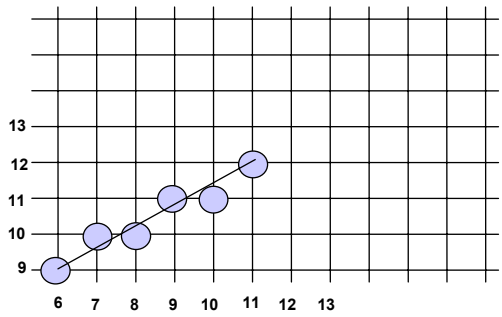


DDA Example

Line from (6,9) to (11,12).

Length := Max of (ABS(11-6), ABS(12-9)) = 5
Xinc := 1
Yinc := 0.6

Values computed are:
(6, 9)
(7, 9.6)
(8, 10.2)
(9, 10.8)
(10, 11.4)
(11, 12)

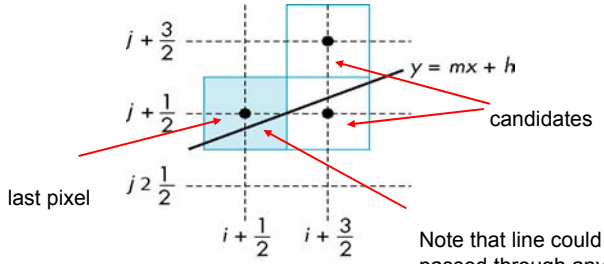


Bresenham's Algorithm

- DDA requires one floating point addition per step
- We can eliminate all fp through Bresenham's algorithm
- Consider only $1 \geq m \geq 0$
  - Other cases by symmetry
- Assume pixel centers are at half integers
- If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer

Idea

$1 \geq m \geq 0$



Note that line could have passed through any part of this pixel

Decision variable: $d = \Delta x(a-b)$

d is an integer
d < 0 use upper pixel
d > 0 use lower pixel