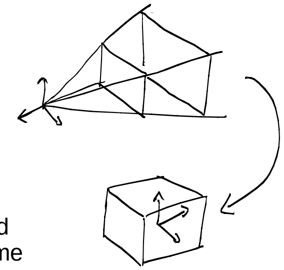
Lecture 6

- clipping
- windowing and viewport
- scan conversion/ rasterization

Last class



normalized
view volume

projective
transform
followed by
normalization

<u>Last lecture (clip coordinates):</u>

A vertex (w x, w y, w z, w)

is in the

normalized view volume if:

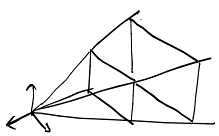
w > 0

- w <= w x <= w

- w <= w y <= w

- w <= w z <= w

Terminology: clipping vs. culling

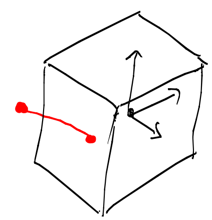


Any object that lies entirely outside the view volume doesn't need to be drawn. Such objects can "culled".

Any object that lies *partly* outside the view volume needs to be "clipped".

Today, "clipping" refers to both of these.

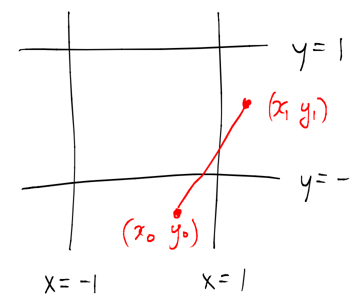
3D <u>Line</u> Clipping



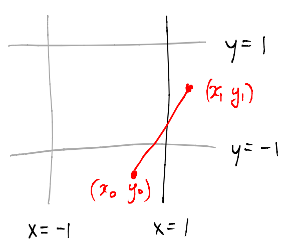
Q: Given endpoints (x0, y0, z0), (x1, y1, z0), how to check if the line segment needs to be clipped ?

i.e. either discarded, or modified to lie in volume

2D Line Clipping (simpler to discuss)



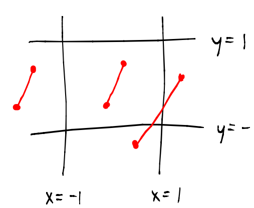
Q: Given endpoints (x0, y0), (x1, y1), how to check if the line segment needs to be clipped ?



To check if a line segment intersects a boundary e.g. x=1, solve for t:

t (x0, y0) + (1 - t) (x1, y1) = 1

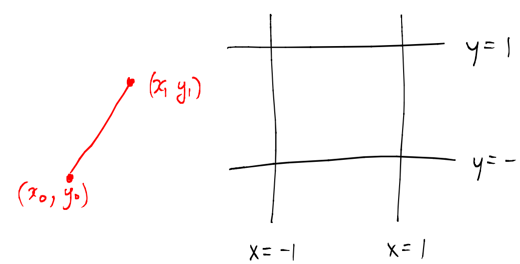
and check if 0 <= t <= 1.



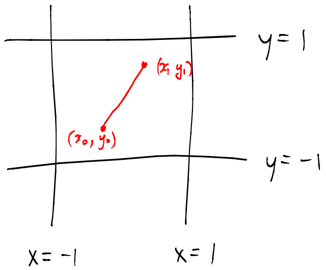
3 cases of interest: the line may be....

- entirely outside of view volume
- entirely in view volume
- partly in view volume

Q: Given endpoints (x0, y0), (x1, y1), how to check if the line segment needs to be clipped ?



This line can be "trivially rejected" since the endpoint x values are both less than -1.



This line can be "trivially accepted" since the endpoint x and y values are all between -1 and 1.

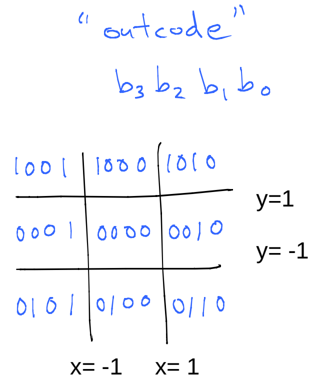
Cohen-Sutherland (1965) encoded the above rules :

$b_3 = y > 1$

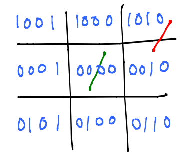
$b_2 = y < -1$

$b_1 = x > 1$

$b_0 = x < -1$



For each vertex, compute the outcode.



Trivially reject a line segment if

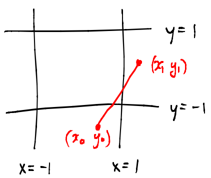
bitwiseAND (_____ , _____) contains a 1.

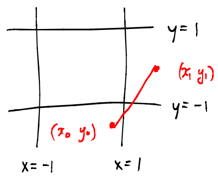
Trivially accept a line segment if

bitwiseOR (_____ , ______) == 0000.

In both cases below, we can *neither* trivially accept nor reject.

Outcodes are the same in the two cases.

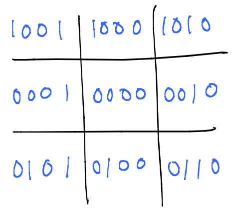


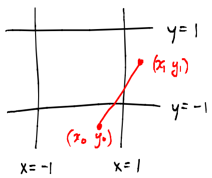


clipping required
(line modification)

reject
(non-trivial)

What if we cannot trivially accept or reject ?

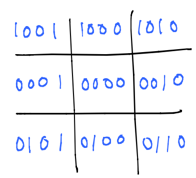


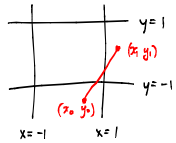


Q: what is the logic condition for this general case ?

A: bitwiseXOR(____ , ____) is

If we cannot trivially accept or reject, then the line must cross one of x=1, x=-1, y=1, or y = -1.





Cohen-Sutherland: consider the bits b3, b2, b1, b0 such that XOR(b, b') = 1.

Modify/clip the line segment to remove the offending part.

Example:

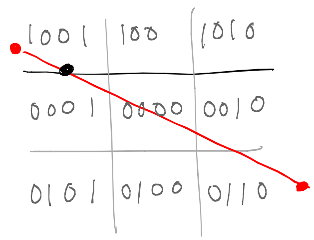
b3 b2 b1 b0

$b_3 = y > 1$

$b_2 = y < -1$

$b_1 = x > 1$

$b_0 = x < -1$



First clip line segment so that b3 = 0 for both outcodes.

---

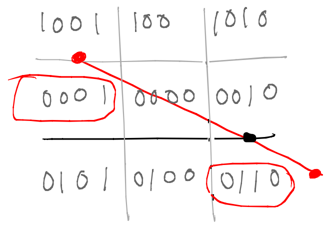
b3 b2 b1 b0

$b_3 = y > 1$

$b_2 = y < -1$

$b_1 = x > 1$

$b_0 = x < -1$



Then, clip line segment so that b2= 0 for both outcodes.

---

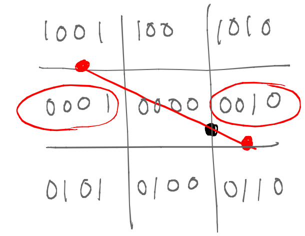
b3 b2 b1 b0

$b_3 = y > 1$

$b_2 = y < -1$

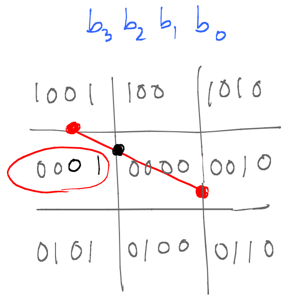
$b_1 = x > 1$

$b_0 = x < -1$



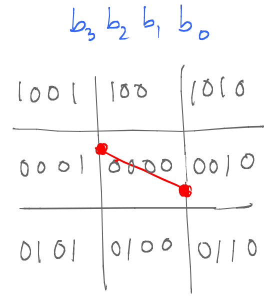
Then, clip line segment so that b1= 0 for both outcodes.

$b_3 = y > 1$
$b_2 = y < -1$
$b_1 = x > 1$
$b_0 = x < -1$



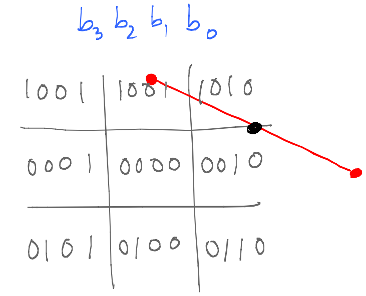
Then, clip line segment so that b0 = 0 for both outcodes.

$b_3 = y > 1$
$b_2 = y < -1$
$b_1 = x > 1$
$b_0 = x < -1$



And we're done.... trivial accept !

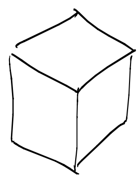
Typically we don't need to do all four clips before trivially rejecting.



Cohen-Sutherland line clipping in 3D:

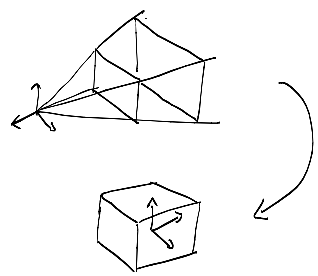
(exactly the same idea but the outcodes have 6 bits)

$b_5 = z > 1$
$b_4 = z < -1$
$b_3 = y > 1$
$b_2 = y < -1$
$b_1 = x > 1$
$b_0 = x < -1$

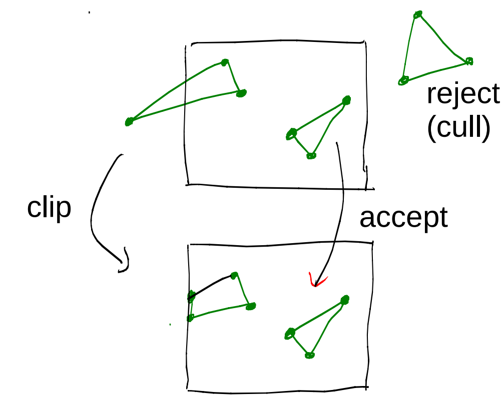


By the way.....

If we didn't do a projective transformation and map to normalized view volume, we could still compute outcodes and do line clipping, but it wouldn't be as easy.



Algorithms for clipping polygons (SKIP !)



Recall:

OpenGL clips in (4D) 'clip coordinates'

(w x, w y, w z, w)

<u>not</u> in (3D) 'normalized device coordinates'

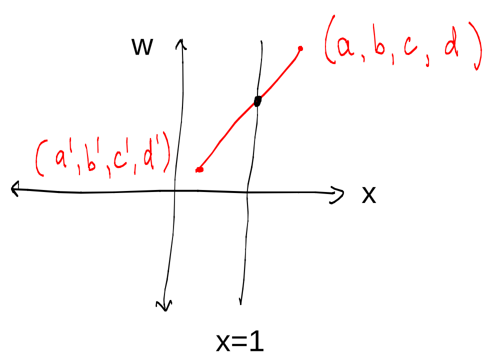
(x, y, z).

We can compute outcodes in clip coordinates easily.

But the line clipping is *tricky* in clip coordinates.
Why?

Exercise (surprising):

Clipping based on 4D interpolation works !



Recall from lecture 2:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} + \begin{bmatrix} a' \\ b' \\ c' \\ d' \end{bmatrix} \neq \begin{bmatrix} a + a' \\ b + b' \\ c + c' \\ d + d' \end{bmatrix}$$

The above was an abuse of notation.
It was meant to express that:

$$\begin{bmatrix} a/d \\ b/d \\ c/d \end{bmatrix} + \begin{bmatrix} a'/d' \\ b'/d' \\ c'/d' \end{bmatrix} \neq \begin{bmatrix} (a+a')/(d+d') \\ (b+b')/(d+d') \\ (c+c')/(d+d') \end{bmatrix}$$

The issue for clipping is whether the following interpolation scheme can be used.

$$t\begin{bmatrix}a\\b\\c\\d\end{bmatrix} + (1-t)\begin{bmatrix}a'\\b'\\c'\\d'\end{bmatrix}$$

The answer is yes, but it requires some thought to see why.

Lecture 6

clipping

windowing and viewport

scan conversion / rasterization

What is a "window" ?

Two meanings:

- region of display screen (pixels) that you can drag and resize. Also known as "display window".

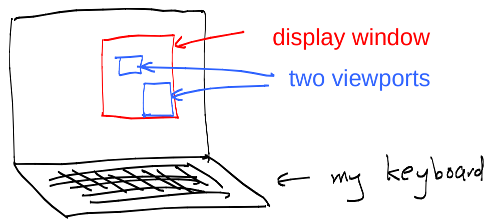
- region of the near plane in camera coordinates. Also known as "viewing window".

```
glutCreateWindow('COMP557 A1')

glutInitWindowSize(int width, int height)

glutInitWindowPosition(int x, int y)

glutReshapeWindow(int width, int height)

glutPositionWindow(int x, int y)
```

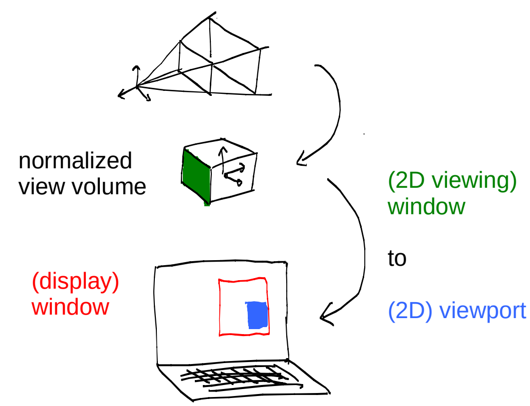
What is a "viewport" ?

glViewport(int x, int y, int width, int height)

A viewport is a region within a display window.
(The default viewport is the whole window.)



"window to viewport" transformation



We've finally arrived at pixels!

How do we convert our floating point (continuous) primitives into integer locations (pixels) ?

Lecture 6

clipping

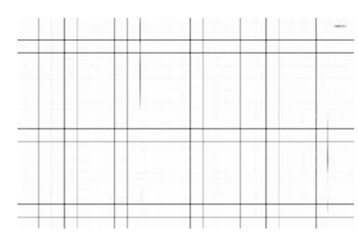
windowing and viewport

scan conversion / rasterization

What is a pixel ?

Sometimes it is a point (intersection of grid lines).

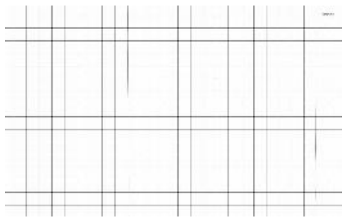
Sometimes it is a little square.



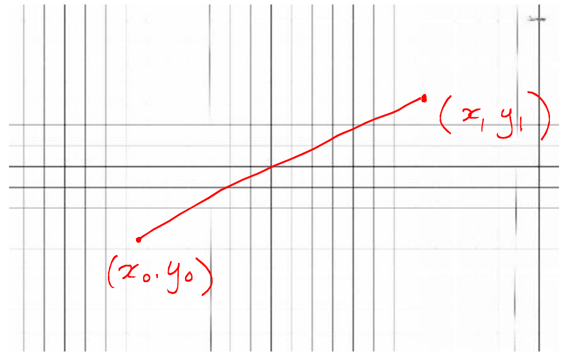
"Scan Conversion" ("Rasterization")

- convert a continuous representation of an object such as a point, line segment, curve, triangle, etc into a discrete (pixel) representation on a pixel grid

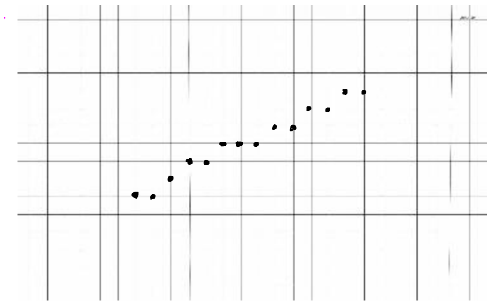
- why "scan" ?



e.g. Scan Converting a Line Segment ?



The endpoints of the line segment may be floats.



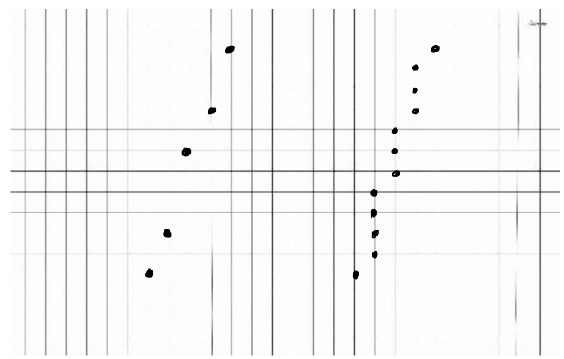
In this illustration, pixels are intersections of grid lines (not little squares).

Algorithm:

scan convert a line segment
from (x0, y0) to (x1, y1)

```
m = (y1 - y0) / (x1 - x0)        // slope of line
y = y0

for x = round(x0) to round(x1)

     writepixel(x, Round(y), rgbValue)
     y = y + m
```

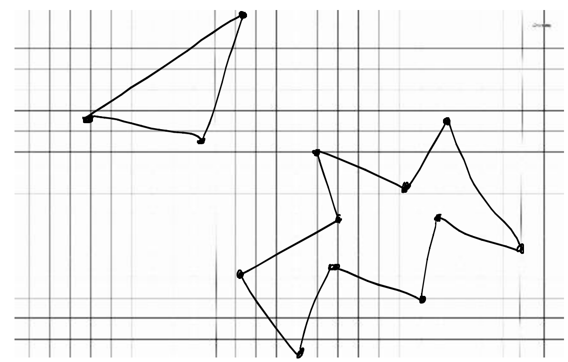
What if slope |m| is greater than 1 ?



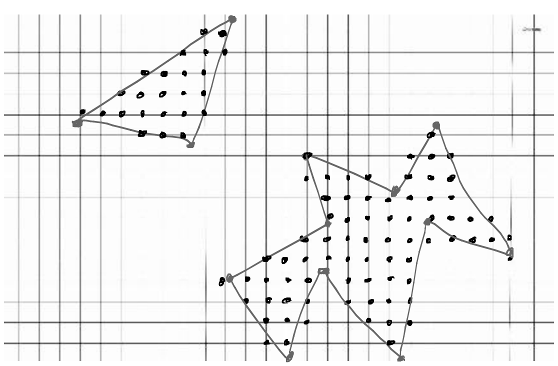
Iterating over x leaves gaps (bad)

Iterating over y fills gaps (good)

Scan converting (filling) a Polygon



Scan converting (filling) a Polygon



Scan converting a polygon <u>(Sketch only)</u>

ymin = round(min of y values of vertices)
ymax = round(max of y values of vertices)

for y = ymin to ymax

compute intersection of polygon edges with row y

fill in pixels between adjacent pairs of edges
i.e. (x, y) to (x', y'), (x'', y) to (x''', y), ...
where x < x' < x'' < x''' < ...