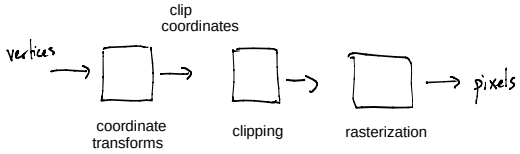
lecture 7

- graphics pipeline (overview)

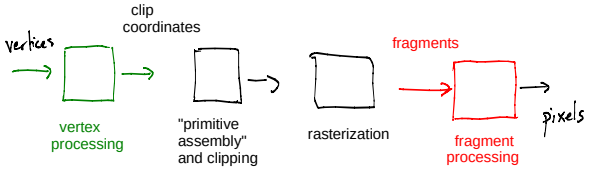
- hidden surface removal

  - object vs image order
  - back face culling
  - depth buffer (z buffer)
  - painters algorithm
  - ray casting

graphics pipeline (lectures 1-6)

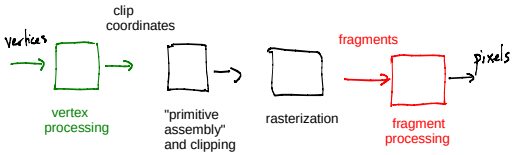


OpenGL pipeline **(on the graphics card)**



A "fragment" is a potential pixel and the data needed to color it, including depth, surface normal.

classic vs. modern OpenGL



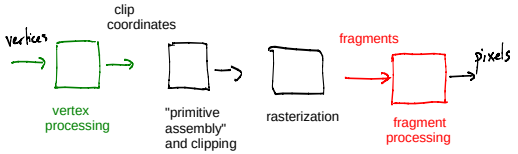
OpenGL 1.0	**"fixed function"**	hidden	hidden	**"fixed function"**
Modern OpenGL	**"vertex shader" (programmable)**	hidden	hidden	**"fragment shader" (programmable)**

Vertex Processing

Suppose you want to make a water wave animation. The surface is a set of triangles, made from vertices

{ (x, height(x,z,t)), z }

height() is a little program -- a "vertex shader"
e.g. a sine wave.



In classic OpenGL, the CPU calculates height(x, z, t) for each x, z and then calls glVertex().

In modern OpenGL, height(x, z, t) is computed using a "vertex shader", on the graphics card and in parallel for different (x, z) and t, .

"Particle systems" (vertex processing)

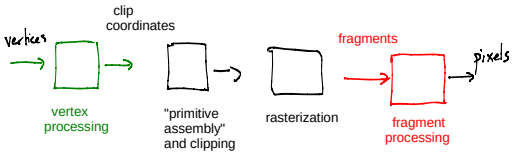
e.g. Fire, explosions, smoke, fog, ...

Calculate geometric transforms on vertices/primitives.
Calculate (time varying) "color" of vertices, too!

Careful: no pixels yet !

"Primitive Assembly" and Clipping

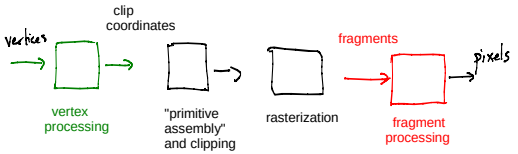
```
glBegin(GL_LINES)   //  line primitive (data structure)
  glVertex(...)     //  is assembled after its vertices
  glVertex(...)     //   are mapped to clip coordinates
glEnd()
```



Fragment processing

A fragment is a potential pixel. It has an (x,y) coordinate, and information about depth, color, ...

We will discuss fragment processing later in the course.



Part 2 of the course starts here.

Visibility, geometry modelling

7. hidden surface removal
   back face culling, depth buffer, painter, ray casting
8. efficient ray casting
   bounding volumes, octtrees, BSP tree
9. object hierarchies 1
   OpenGL transformation stack, scene graph
10. object hierarchies 2
    fractals, L-systems, plant models
11. smooth curves and surfaces
    cubics: Hermite curves & Catmull-Rom splines, bicubic surfaces
12. meshes
    level of detail, edge collapse

lecture 7

- graphics pipeline (overview)

- hidden surface removal:

the problem of deciding
which polygon/object is visible at each pixel.

- object vs image order
- back face culling
- depth buffer (z buffer)
- painters algorithm
- ray casting

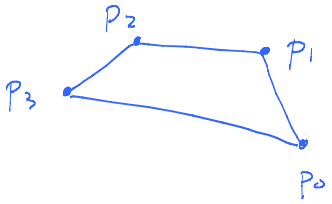
"object order" methods

```
for each object
    for each pixel
        decide if object is visible at that pixel
```

"image order" methods

```
for each pixel
    for each object
        decide if object is visible at that pixel
```

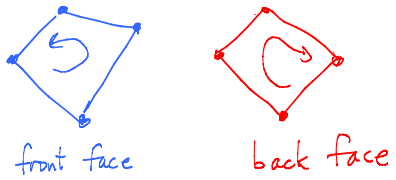
Back face culling (object order)



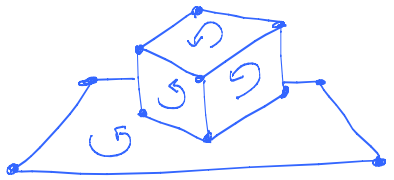
A polygon is defined by a sequence of vertices.
What is the significance of the ordering?

In OpenGL, the **front face** of a polygon is defined (by default) as the side where vertices would be ordered counter clockwise, i.e. if a viewer were on that side.

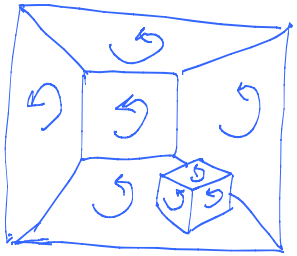
glFrontFace(GL_CCW) // default



glFrontFace(GL_CW) // override default

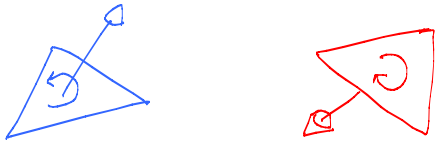


Choose ordering of vertices for each face as shown so that front faces are seen by a viewer.



In this example, the vertices of each wall of the room have opposite ordering as on previous slide. This allows a viewer that is inside the room to see the walls.

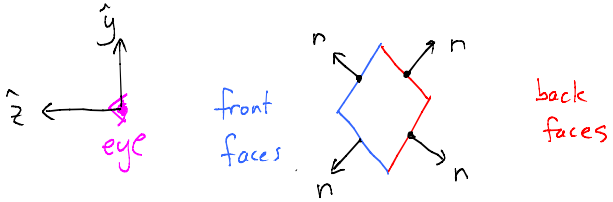
The normal comes out of the front face.
(The front face has an "outward pointing normal" and a back face has an "inward pointing normal".)



[WARNING: In OpenGL, surface normals also can be explicitly defined defined at vertices. (We will see later why.) But these normals have nothing to do with front and back faces as just defined.]

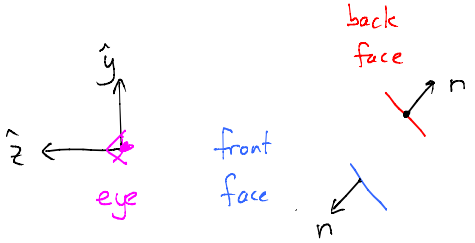
"Back Face Culling"

= don't draw the back faces!



For a solid object, back faces shouldn't be visible because they are hidden by the front faces.

glEnable(GL_CULL_FACE)

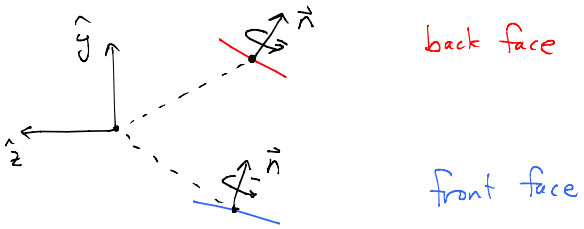


If back face culling is "enabled", then the back face is not drawn.

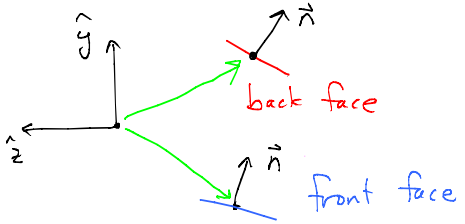
In A1, we used glDisable(GL_CULL_FACE).

When is a polygon back vs. front face ?

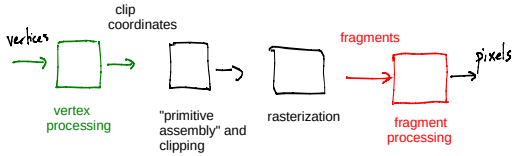
In camera coordinates, it is subtle. You can't just look at the sign of the z coordinate.



Use the sign of the dot product of the outward facing normal and the vector from the viewer to any vertex on the polygon.

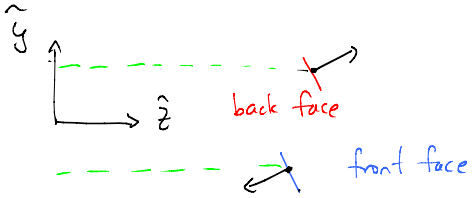


Q: Where is back face culling done ?



A: In OpenGL it is done by rasterizer. But in principle, it can be done before entering pipeline!

In OpenGL, back face culling is done by the rasterizer, hence it is done in *normalized device coordinates*. Check the sign of the normal's z coordinate.



Rule: $\vec{n} \cdot \vec{z} > 0$ back face

$\vec{n} \cdot \vec{z} < 0$ front face

Depth Buffer (z buffer)

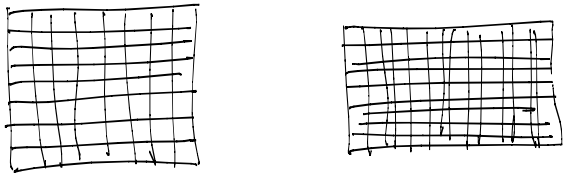


image buffer/ frame buffer

(not same as monitor screen)

z buffer

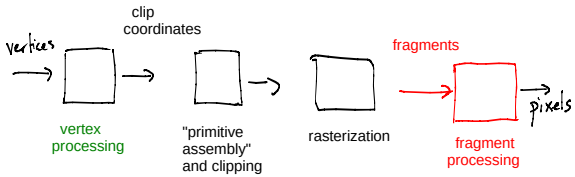
glEnable(GL_DEPTH_TEST)

Hidden surface removal algorithm (depth buffer) Catmull 1974

```
for each polygon P:
    for each pixel (x,y) in  P's image projection

        if  P's depth at (x,y)  is less than zbuffer(x,y)

        then

            update zbuffer(x,y)
            compute color of P at (x,y) and replace
                the current color with the new color
```

Pseudocode...

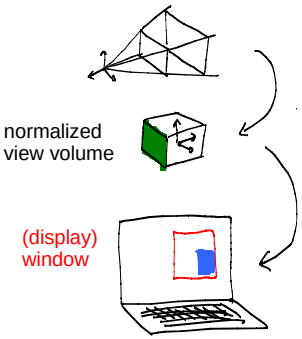
```
for each pixel (x,y):     // initialization
    zbuffer(x,y) = 1
    RGB(x,y) = background color

for each polygon:
   for each pixel (x,y) in the image projection of
       polygon     z := Ax + By + C

    // equation of polygon's plane in SCREEN coordinates

      if z < zbuffer(x,y) :
         zbuffer(x,y) :=  z
         compute RGB(x,y)
```



Q: Where is the depth buffer algorithm here ?

A: It is done by the rasterizer. If polygon fails the depth test at (x,y), then fragment never gets generated.

Recall last lecture: "window to viewport" transformation



normalized view volume

(x,y) in [-1, 1] x [-1, 1]

(2D viewing) window

to

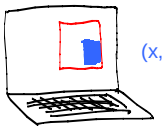
(display) window

(2D) viewport

(x,y) in [0, width] x [0, height]

There is also a depth component to the mapping.

(x,y, z) in [-1, 1] x [-1, 1] x [-1, 1]



(x,y, z) in [0, width] x [0, height] x [0, 1]

Depth buffer typically holds fixed precision (e.g. 24 bits),
not float.

i.e. z in [0, 1] partitioned into bins of size 1 / 2^24.

Q: Why ?

A: Floating point is useful for representing tiny and huge numbers, but that's in appropriate here.

Recall glFrustum(__ , __ , __ , __ , near, far)

where near and far are float or double.

Q: Why not set:

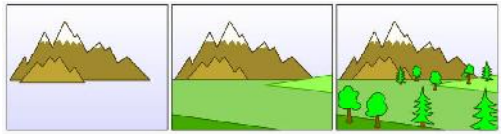
near = .00000...1

far = 2^ 1023 ?

A: You would divide the huge [near, far] interval into 2^24 depth intervals/bins. Most points will fall in one of these intervals --> useless because they all have same (quantized/rounded) depth

Painter's Algorithm [Newell et al 1972]

(older than depth buffer)



http://en.wikipedia.org/wiki/Painter's_algorithm

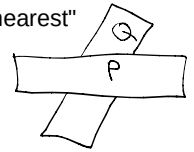
Draw farthest polygons first.

Painter's Algorithm

Sort polygons in depth. (Arbitrarily) use the farthest vertex in each polygon as the sorting key.

e.g 26 polygons [A, B, C, ... P, Q, ... , Z]

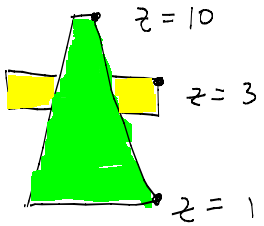
Then draw polygons from "farthest" to "nearest" (back to front).



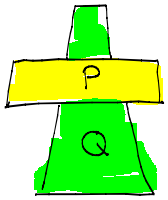
However, that doesn't always work. Why not ?

Typical failure of method on previous slide:

desired

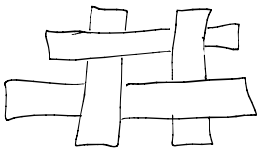


(naive) Painter



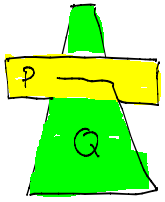
Solution ? Swap order of P and Q in the list ?
(That can create other problems.)

Classic failure case (cardboard box):

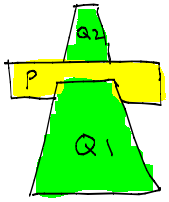


For this example, there does not exist a 'correct' ordering for drawing the polygons.

A related common example of failure

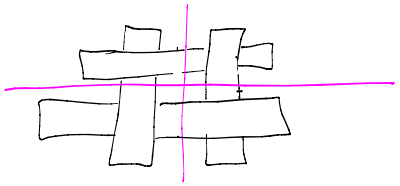


Problem



Solution
(cut Q into Q1, Q2)

Classic failure case (cardboard box):



Same solution can be applied here.
(Cut the polygons.)

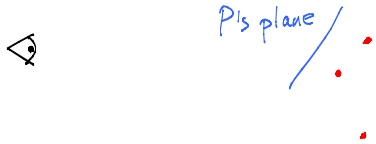
Q: When is it safe to draw Q before P ?

A: when *Q does not occlude P.*

Any of the following:

- all Q's vertices are farther than all P's vertices
- the x-range of P and Q do not overlap
- the y-range of P and Q do not overlap
- ... (previous slides problems not covered yet)

- all Q's vertices are on the far side of P's plane



- all P's vertices are on the near side of Q's plane

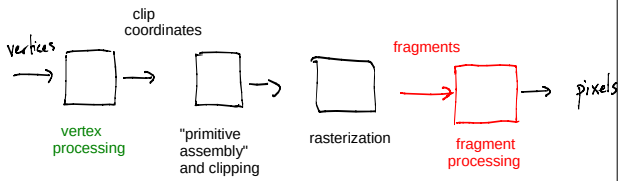
SLIDE ADDED (morning after)

I have not given the full Painter's Algorithm.

I have not shown how to deal with every case.

The full algorithm has more details that I want to cover.

But I think you get the main ideas. And its enough for me to motivate BST trees next lecture.



Q: Where is the painter's algorithm here ?

A: The decision to draw (or cut) a polygon is made by the CPU prior to putting vertices into the graphics pipeline.

lecture 7

- graphics pipeline (overview)
- hidden surface removal
  - object vs image order
  - back face culling (object order)
  - depth buffer (z buffer) (object order)
  - painter's algorithm (object order)
  - ray casting (image order)

Ray Casting (an image order method)

```
for each pixel (x,y)  {

    for each polygon{

        check if the pixel lies in the image
        of that polygon  and  if the depth is
        smallest seen so far at that pixel.
    }
    draw the color of the nearest polygon
    at that pixel
}
```

Ray Casting

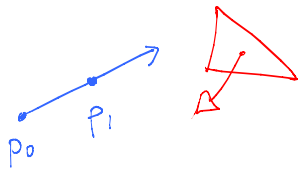
```
for each pixel (x,y){                 // screen coordinates
    zMin = 1                          // initialized to max value

    for each polygon{
        if (x,y) lies in image projection of this polygon{
            z =  Ax + By + C    //     screen coordinates
            if  z < zMin
                zMin := z
                pixel(x,y).poly = polygon
        }
    }
    RGB(x,y)  =  ...?...          // only draw once per pixel
}
```

No z-buffer needed.

For each pixel, only choose its color once.

Need to determine if a pixel lies in a polygon.
(Details omitted.)

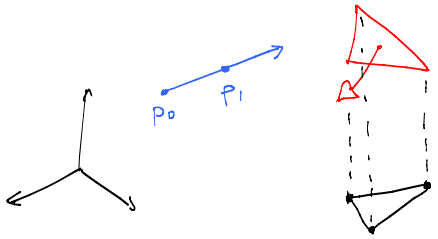
More general ray casting (used next lecture)



Does ray from p0 through p1 intersect the polygon?

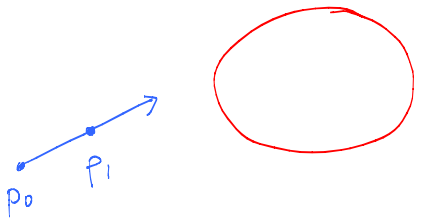
Typically p0 is the camera position. But not necessarily...

How to decide if the intersection point lies in the polygon?



Trick: project the polygon and the intersection point with plane orthographically into a canonical plane and use the 2D "point in polygon" solution.

Does ray intersect this quadric ?



[x, y, z, 1] Q [x, y, z, 1]^T = 0

(x(t), y(t), z(t)) = p0 + (p1 - p0) t

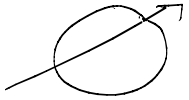
Substitute the ray into the quadric gives a second order equation:

$\alpha\, t^2 + \beta\, t + c = 0$

Solve for t.

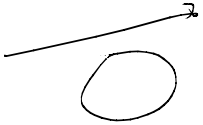
Gives two solutions. What are the possibilities ?

Two real roots.
If at least one is positive, then take the smaller positive one.



Two real roots (identical).

Both roots are complex.



Announcement

A1 was posted on Friday. It is due next Monday.

Example of solution (executable):
http://cim.mcgill.ca/~fmannan/comp557/index.html

TA office hours this week.

DISCUSSION BOARD SETTINGS.
Please uncheck: "Include original post in reply"