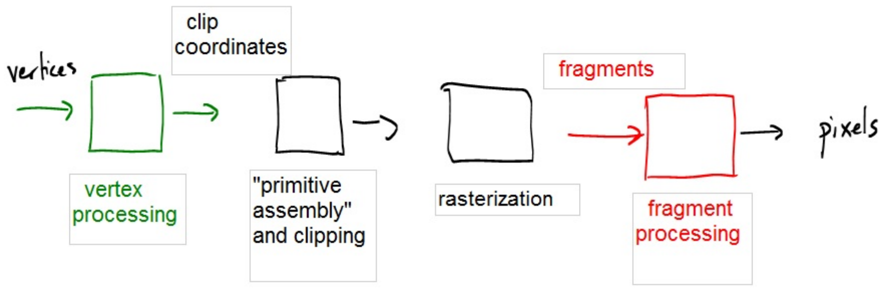
Lecture 7

At the begin of this lecture I discuss the standard graphics pipeline. This finishes up Part 1 of the course, covered in lectures 1-6, and also provides a segue to Part 2 which concerns issues of visibility and scene and object modelling.

Graphics pipeline

We have spent the last several lectures discussing how to map positions of object points to positions of pixels in the image. This mapping involved a sequence of matrix transformations, as well as other operations such as clipping, rasterization, depth testing, and many more. Here I will sketch out the basic stages of the pipeline.



Vertex processing

Here each vertex is processed individually. For now, you should think of the vertex processor as transforming the coordinates of the vertex from object to clip coordinates, since this is what we have been mostly talking about the past several lectures. It turns out there is more to it that that, for example, the color of the vertex and other properties of a vertex can be determined here as well.

In classical OpenGL, there are a limited (fixed) set of functions for the programmer to use at this stage – basically the transformations covered in lectures 1-6. In modern OpenGL, there is much more flexibility. The programmer implements this stage explicitly by writing programs called *vertex shaders*. This includes explicitly writing the transformations we have seen, as well as other interesting transformations on vertices and surface normals.

For example, suppose you wanted to make an animation of a water surface with complicated waves. Or suppose you wanted to animate a person's face as the person speaks. In classic OpenGL, the changes in the mesh would be done on the CPU. For example, you would define data structures for representing your surface object (say a "mesh" of triangles), and you would write code for the CPU to manipulate the triangles of this object so that they have a certain motion over time (and maybe changing colors).

In modern OpenGL, these vertex manipulations would be done on the graphics card (the GPU). You would write instructions for manipulating the surface. THis would be a program called a vertex shader. The advantage of using the GPU is that it has many parallel processors. They all run the same instructions (the code of the vertex shader) in parallel over multiple vertices. The CPU is then free to do other things.

"Primitive Assembly" and Clipping

In order to clip a line segment or triangle or polygon, these primitive must be created. Loosely, think of an object of type 'line segment' or type 'triangle'. The object would have pointers to its constituent vertices.

After the vertices have been transformed individually to clip coordinates by the vertex processor (shader), the "geometric primitives" – e.g. line segment objects, or triangle objects – must then be assembled. Think Cohen-Sutherland here: you can't run the clipping algorithm unless the two vertices are considered as a line segment. (Note that a different clipping algorithm is used to clip a line vs a triangle vs a general polygon. I omitted those details in the clipping lecture.)

As a concrete example, think of the following. The `glBegin` specifies what the 'primitive assembler' need to assemble (a line) and which vertices to use. The `glVertex()` calls send the vertices to the vertex shader who maps them to clip coordinates.

```
glBegin(GL_LINES)
   glVertex()
   glVertex()
glEnd()
```

Both in classical OpenGL and modern OpenGL, these two stages – primitive assembly and clipping – are done "under the hood", and indeed are sometimes considered as grouped into one stage. As a programmer, you don't have to think about it. You define the geometric primitives and you let the graphics card take care of assembling them at this stage.

Rasterization

Here we say the processing is done by a "rasterizer". For each primitive (line, triangle, etc), the rasterizer defines a set of potential pixels that this primitive might contribute to. (I say "potential" because the primitive is not necessarily visible in the image.) These potential pixels generally have more information associated with them than just (x,y) values, for example, they may have depth and color information. Such potential pixels are called *fragments*. So the output of the rasterizer is a set of fragments.

Both in classical OpenGL and modern OpenGL, this stage is also hidden. The programmer has no access to the rasterizer.

Fragment Processing

The next stage is to process the fragments. The job the "fragment processor" is a big one. Given all the fragments, the fragment processor computes the image RGB value at each pixel. This involves not just deciding on the intensities. It also involves deciding which surfaces are visible at each pixel.

In classical OpenGL, there are a limited (fixed) number of functions for the programmer to use at this stage. In modern OpenGL, there is much more flexibility and the programmer implements this stage explicitly by writing programs called *fragment shaders*. Much of the second half of the course will deal with this stage. We won't be writing fragment shaders, but we will be discussing the problems that these shaders solve and the classical algorithms for solving them.

We are now ready to continue to part 2 of the course which concerns visibility and geometry modelling. We begin with the hidden surface removal problem.

Hidden surface removal

Suppose we have a view volume that contains a set of polygons, for example, triangles, that are pieced together to make up an object. We would like to know which points on these polygons are visible in the image. We will examine several methods. Some of these methods examine each of the objects of the scene and decide, *for each object*, at which pixels are the objects visible. These are called "object order" methods. Others examine, *for each pixel*, which of the objects can be visible at that pixel. These are called "image order" methods.

Let's begin by looking at a few object order methods that are used by OpenGL. I'll then take a brief step back and discuss how these methods fit into the "OpenGL pipeline". After that, I'll return to the hidden surface problem and discuss some methods that are *not* used by OpenGL but that are heavily used in other graphics systems. These can be object or image order.

Back face culling

In many cases a polygon is part of a surface that bounds a solid object, e.g. the square faces that bound a cube. We only want to draw the "front" part of the object, namely the part facing towards the camera. Thus for each polygon we would like to test if it faces toward the eye or away from the eye and, in the latter case, we discard the polygon rather than draw it. Discarding back faces is called *back face culling*[1].

In the above description, we relied on having a solid object to define front and back faces. But back face culling is more general than this. For any polygon, we can define the front and back face of the polygon. Back face culling just means that we only draw the polygon if the viewer is on the side of the polygon that corresponds to the front face. This is a more general notion since it applies to polygons that don't belong to solid objects.

In OpenGL, the default for the front face is that the vertices are ordered in a counter-clockwise direction as seen by the viewer. That is, given a polygon and a viewer position, the viewer sees the front face if the viewer sees the vertices ordered counter clockwise and the viewer sees the back face if the viewer sees the vertices ordered clockwise.

[ASIDE: OpenGL has commands that can reverse this definition. The default front face is counterclockwise, but `glFrontFace(GL_CW)` and `glFrontFace(GL_CCW)` can be used to toggle this definition. If you want both sides of a polygon to be rendered, then there is a state variable `GL_FRONT_AND BACK` that you can set.]

Conceptually, the front face has an outward facing normal and the back face has an inward facing normal. However, these concepts of outward and inward being associated with front and back faces, respectively, is only in your head. It is not part of OpenGL. In OpenGL, one can define surface normals explicitly (and we will do so later in the course when we discuss shading) but these explicitly defined surface normals are not constrained to correspond to front faces, and the explicit surface normals have nothing to do with how back face culling is implemented. Back face culling is

[1]to "cull" basically means to "remove"

implemented only by examining whether a polygon's vertices are counterclockwise, as seen by the viewer.

Finally, we don't *have to* do back face culling to draw the scene correctly – at least, the solid objects. As long as we do hidden surface removal (see above), back faces of a solid object automatically will not appear in the image but rather will be hidden by front faces which are closer to the camera. The reason for doing back face culling is that it is a cheap way to speed up the computation.

Back face culling is a general technique and there is no single stage of the graphics pipeline where it has to occur. As shown in the slides, it can be done in eye coordinates: in this case the decision to cull or not cull a polygon depends on the sign of the dot product of two vectors: the vector from the eye to a vertex on the polygon, and the normal pointing out of the front face. If that dot product is positive then the polygon is back facing and can be culled; otherwise, it is kept (not culled).

In OpenGL, back face culling is done at the rasterization phase, that is, after the polygon has passed the clipping stage and so we know the polygon lies in the view frustum. The vertices of the polygon are now represented in normalized view coordinates, not eye coordinates. To decide if a surface polygon is a front or back face, the rasterizer checks if the outward normal vector of the polygon *in normalized device coordinates* has a negative or positive z component, respectively. (Recall that viewing in normalized device coordinates is orthographic. To get the sign of the test correct, recall that normalized device coordinates are left handed coordinates i.e. $\hat{\mathbf{z}}$ points away from the camera.) If the polygon is culled, this means that the rasterizer doesn't generate any fragments for that polygon.

Depth buffer method

Let's now turn to a second object order method for deciding which points on a polygon are visible in the image. Here we need to introduce the idea of a *buffer*. In computer graphics, a buffer is just a 2D matrix that holds values. The buffer holding the RGB values of the image[2] is referred to as the *frame buffer* or *image buffer*. Many graphics systems also have a buffer that holds the z values (depth) of the surface point that is visible at a pixel. This is known as the *z buffer* or *depth buffer*. In OpenGL, the z buffer holds values in $[0, 1]$ where 0 corresponds to the near plane and 1 corresponds to the far plane.

Recall I mentioned last lecture that there is a window to viewport mapping that takes the xy in $[-1, 1] \times [-1, 1]$ of normalized device coordinates and maps it to viewport coordinates. We can tack on the depth values to this transformation and map z from $[-1, 1]$ to some range of values. In OpenGL, the z mapping is from $[-1, 1]$ to $[0, 1]$. So in screen coordinates, depth has values in $[0, 1]$. This mapping is achieved in the expected way, namely a translation and a scaling. (You now should see why we used f_0 and f_1 back in lecture 5.) We referred to the xy coordinates as viewport coordinates last lecture. We also sometimes call them *screen coordinates.* With the depth buffer in mind, we can think of screen coordinates as having a depth too.

Here is the depth buffer algorithm for calculating visible surfaces. Note that it is an object order method: the polygons are in the outer loop and pixels are in the inner loop.

[2]Or RGBA – see later in the course

```
for each pixel (x,y)   // initialization
   z(x,y) = 1
   RGB(x,y) = background color
for each polygon
   for each pixel in the image projection of the polygon
        z := Ax + By + C
         // equation of polygon's plane *in screen coordinates*
        if z < z(x,y)
            compute RGB(x,y)
            z(x,y) :=  z
```

A few points to note. First, you might expect that floats are used for z values in order to make precise decisions about which surface is in front of which. This is not the case, however. Rather OpenGL uses fixed point depth representation, typically 24 bits (but sometimes 32). This can lead to errors when multiple objects fall into the same depth bin. However, 2^{24} is alot of bins so this is unlikely to happen (though see below).

To elaborate of why floating point isn't used: floating point allows you to represent extremely small numbers and extremely large numbers, with the bin size being roughly proportional to the size of the number. But that last condition makes no sense for representing depth values in graphics where the objects can be anywhere between the near and far plane. Moreover, you don't typically deal with extremely tiny or extremly large depths. So floating point makes little sense for depth. [And besides, fixed point arithmetic is generally faster.]

A second point to note is that we can now see why the choice of `near` and `far` is important. You might have been thinking up to now that it is best to choose your `near` to be very close to 0 and your `far` to be infinity. The problem with doing so is that you only get 2^{24} depth bins. If near is too close and far is too far, then "aliasing" can easily occur – namely two objects can mapped to the same depth bin for some (x,y). In that case, when the algorithm compares the depth of a polygon at that pixel to the value stored in the depth buffer, it can potentially make errors in deciding which surface is closest.

Finally, where in the pipeline is the depth buffer algorithm run? The simplest way to think about it for now is that it is done by the rasterizer, rather than the fragment shader. For each polygon, the rasterizer generates a set of fragments and compares the depth of each fragment to the current depth value at the associated pixel in the depth buffer. If the depth of the fragment is greater than the closest depth at that position, then the fragment is discarded.

Painter's algorithm (Depth sort)

Here I give an overview of a second object order method, which draws polygons in terms of *decreasing* distance i.e. draw farthest ones first. This is called the painter's algorithm because it is similar to what many painters do, namely paint closer objects *on top of* farther ones. The method goes like this:

1. Sort polygons according to each polygon's *furthest* z coordinate.

   However, in the case that two polygons have overlapping z range, we need to be careful about how we place them in the ordered list. This is discussed below and you should consult the slides for figures.

2. Draw the polygons in the order defined by the list, namely from back to front (farthest to nearest).

Suppose we have two polygons P and Q such that furthest vertex of Q is further than the further vertex of P. It could still happen that Q could cover up part of P or even all of P in the image – and in that case we do not want to draw Q before P. One thing to try is to swap the ordering of P and Q. This doesn't necessarily solve the problem, though since it could create problems with other polygons.

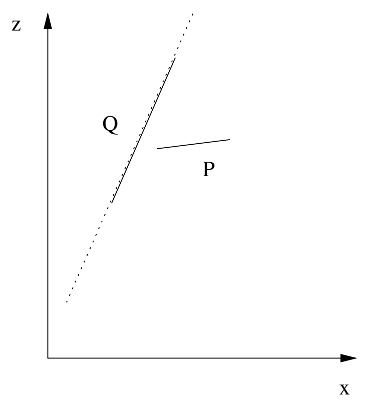
I am not presenting the full algorithm here, since there are alot of details and, in the end, the Painter's algorithm has flaws that we will address next lecture (when we discuss BST trees). But I still want to give you a flavour of the issues here, so let me say a bit more.

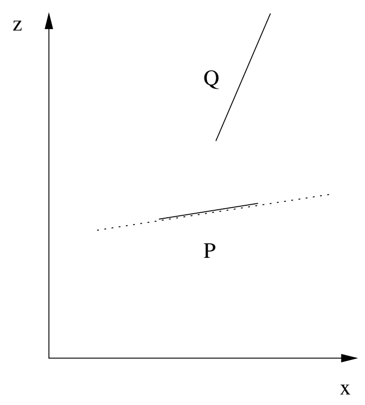
What are some conditions to check in order we can be sure its ok to draw Q before P ? If any of the following conditions is true, then we are ok.

- the z range of P and Q do not overlap, i.e. all of Q's vertices are further than all of P's vertices
- the x range of P and Q do not overlap
- the y range of P and Q do not overlap
- all vertices of Q are on the far side of P's plane (requires checking each vertex of Q against P's plane)
- all vertices of P are on the near side of Q's plane (requires checking each vertex of P against Q's plane)

The last two conditions are subtle. For both, we need to check whether a set of points on one polygon lies in front of or behind another polygon, in the sense of being in front of or behind the plane defined by that polygon. It is a simple exercise to do this, using basic linear algebra.

Note that these last two conditions are not the same. For example, below I show a 2D version of the conditions. On the left, all the vertices of P lie in front of Q's plane, whereas it is not true that all vertices of Q lie behind P's plane. On the right, all vertices of Q lie behind P's plane, whereas it is not the case that all vertices of P lie in front of Q's plane.





What if none of the five conditions holds? In class, I said that the solution is to cut (say) P's plane using Q. However, this is not guarenteed to work. For example, it doesn't necessarily solve the 'highway over bridge' problem of slide 33 in the lecture. e.g. Planes P and Q in that slide could be parallel in 3D and hence not intersect.

Ray casting method

Let's now turn to image order methods. The first image order method, called *ray casting*, is similar to the depth buffer method but it inverts the order of the two loops. It considers each pixel in the image window and shoots (casts) a corresponding ray in the z direction. It then computes the intersection of the ray with all the polygons in the scene and figures out which polygon is closest along the ray.

```
for each pixel (x,y)
   z_min = 1  #  initialized to max value i.e. max[-1,1]
   RGB(x,y) = backgroundRGB
   for each polygon
      if (x,y) lies in image projection of polygon // see item below
        z =  Ax + By + C
            // equation of polygon's plane in screen coordinates
        if z < z_min
           z_min := z
           pixel(x,y).poly = polygon  //  point to polygon
   compute RGB(x,y)   // more later
```

A few observations:

- The ray casting method doesn't require a z buffer, since we are dealing with one pixel at a time. The scalar variable `z_min` plays the role of the z buffer for that single pixel.
- We choose the color of a pixel (x, y) only once in this method. We do so by maintaining a pointer to the polygon that is closest. We only choose the color once we have examined all polygons.
- We need a method to check if a pixel (x,y) lies with the image projection of a polygon. This is known as the "point in polygon" test. I am not planning to present algorithms for solving this problem. If you are interested, please see `http://erich.realtimerendering.com/ptinpoly/`.

The ray casting method implicitly assumed that the scene either was being viewed under orthographic projection, or it already had been transformed into screen coordinates (via normalized device coordinates and a transform to viewport). That is, we assumed z was a function of image coordinates (x, y) of the form $z = Ax + By + C$. In particular, we were assumed we were ray casting in the z direction.

The idea of ray casting is more general than this. For example, suppose we have a polygon that lies in a plane

$$ax + by + cz + d = 0.$$

Consider two points (x_0, y_0, z_0) and (x_1, y_1, z_1). Does the ray that starts at (x_0, y_0, z_0) and that passes through (x_1, y_1, z_1) intersect that polygon? For example, (x_0, y_0, z_0) might be a viewer position and (x_1, y_1, z_1) might be a point in the projection plane.

We do this ray cast in two steps. The first step is to intersect the ray with the plane. We substitute three parametric equations of the ray

$$(x(t), y(t), z(t)) = (x_0, y_0, z_0) + ((x_1, y_1, z_1) - (x_0, y_0, z_0))\ t$$

into the equation of the plane and solve for t. This gives us the point of intersection of the ray with the plane. (Note if $t < 0$ then the ray doesn't intersect the plane, and if the solution gives a division by zero then the ray is parallel to the plane and either doesn't intersect it or lies within it.)

The second step is to decide if this intersection point lies within the polygon. At first glance, this seems a bit awkward to decide since we might be tempted to write the vertices and intersection points in terms of a 2D coordinate system for this plane. A much simpler approach is to project the intersection point and polygon orthographically onto one of the three canonical planes i.e. set either $x = 0$, or $y = 0$, or $z = 0$, and then solve the "point in polygon" problem within that canonical plane. Note that the point is in the polygon in the original plane if and only if its projection is in the projection of the polygon in the canonical plane. The only warning is that you need to ensure that the projection doesn't collapse the polygon into a line.[3]

We can do ray casting with quadric surfaces, just as we did with polygons. We substitute the three parametric equations of the ray $\{(x(t), y(t), z(t)) : t > 0\}$ into the equation for the quadric surface $\mathbf{x^T Q x = 0}$. This yields a second order "quadratic equation" in the variable t, namely

$$\alpha t^2 + \beta t + \gamma = 0.$$

This equation yields two solutions for t, which may be either real or complex. If this equation has no real solution, then the ray does not intersect the quadric. If it has two identical solutions, then the ray is tangent to the surface. If the equation has two real solutions but only one is positive[4], then this one positive solution defines the point on the quadric that is visible along the ray. If the equation has two positive real roots, then the smaller one is used. For example, the quadric might be a sphere, and the ray might pass twice through the sphere – once on the way in, and once on the way out. We would choose the first of these two solutions since we are interested in the "front face".

[3]For example, if the original polygon lay in the $z = 5$ plane, and we project in the x or y direction, then the projected polygon would collapse to a line. This is no good. So, instead for this polygon we must project in z direction.

[4]If a solution is negative, then this solution does not lie on the ray, since the ray goes in one direction away from the point in the positive t direction.