Today's Agenda

> Check your github repos

> Overview of rendering pipeline

> Vertex pipeline

> Coloring

> Basic transformations (translate, rotate, scale)

> Lighting / Normals / Materials

> Texturing

Rendering Pipeline

"Fixed Functionality"

    OpenGL's normal way of rendering

1. Vertex Transformation:
    projecting 3D geometry onto a 2D plane (ie, the screen)

2. The Fragment (or Pixel) Pipeline:
    changing the color of the pixels

Rendering Pipeline

theoretically...

transform the point in 3D into eye coordinates using the MODELVIEW matrix (the view from the camera).

transform the visible space (the "frustum") into the canonical view volume using the PROJECTION matrix, followed by a perspective division. The points are now in "clip coordinates" (this is a unit cube from -1 to 1 along each axis, where anything outside that range will not be visible on the screen).

transform the clip coordinates into actual pixels. The depth information, or "z" coordinate, is used for blending, occlusion tests, picking, etc.

Rendering Pipeline

```
practically...

1. calculate PROJECTION matrix (the lens of your camera)
2. calculate MODELVIEW matrix (positions the drawing "cursor")
3. set-up LIGHTING
4. set-up MATERIALS
5. bind TEXTURES
6. send information to the graphics card :
    for each vertex, you can send a color, a normal, and a texture coordinate

    glTexCoord2f(u, v); //between 0f and 1f, used for TEXTURING
    glNormal3f(x, y, z); //used for LIGHTING CALCULATIONS
    glColor3f(r, g, b); //between 0f and 1f
    glVertex3f(x, y, z);
```

the Projection matrix

The Projection matrix can be thought of as the "lens" of your camera.

It creates a view "frustum" (= a rectangular cone volume) which will contain everything that will be projected onto the screen.

The most common type of projection is a "perspective" view, which imitates the way our eye sees the world

```
gluPerspective(fovy, aspectRatio, nearPlane, farPlane);
```

fovy = our field of view along the y-axis
aspect ratio = the width/height of the screen
near plane = a distance near to the "camera" (usually a small number > 0, like .1f)
far plane = a distance far from the "camera" (usually a larger number, like 100f)

This is usually set up once at the beginning of the application, or when the window is resized.

```
gluPerspective(45.0, (float)w/(float)h, .1, 100.0);
```

the Modelview matrix

The Modelview matrix can be thought of as the "drawing cursor"

You move the cursor, and then set the geometry relative to that cursor.

The Modelview is a single construct which contains information about the "camera" and the position of particular elements.

Often you will set the "camera" by moving the cursor forward

+Z is facing toward the viewer,-Z toward the screen
a "right-handed" system (make an L with thumb and index facing toward you...)

```
glTranslatef(0.0,0.0,-5.0);
```

If you then draw a vertex at this point...

```
glBegin(GL_POINTS);
glVertex3f(0f,0f,0f);
glEnd();
```

...and your near plane is < 5f, then this point will be inside the frustum.

the Modelview matrix

Similiarly, if you draw a rectangle centered at this point...

```
glBegin(GL_QUADS);
glVertex3f(-1f,-1f,0f);
glVertex3f(1f,-1f,0f);
glVertex3f(1f,1f,0f);
glVertex3f(-1f,1f,0f);
glEnd();
```

...and your near plane is < 5f, then this rectangle will be inside the frustum.

the Modelview matrix

There are 3 main "affine" transformations which you can use to control your modelview matrix:

```
glTranslatef(x, y, z);

eg, glTranslate(3f, -1f, -2f); //move 3 to the right, 1 down, and 2 away from the viewer

glRotatef(ang-around-axis, x-axis, y-axis, z-axis);

most simply, this can be split into three separate commands, eg:

glRotatef(20f, 1f, 0f, 0f); //rotate 20 degrees around the x-axis (tilt forward...)
glRotatef(15f, 0f, 1f, 0f); //rotate 15 degrees around the y-axis (pivot...)
glRotatef(30f, 0f, 1f, 0f); //rotate 30 degrees around the z-axis (turn to the side...)

also these are sometimes call pitch, roll, and yaw.

glScalef(x-factor, y-factor, z-factor);

eg, glScalef(2f, .5f, 1f); //elongate the width by 100% and shrink the height by 50%
```

Drawing

Once you've positioned your drawing cursor (via the translate, rotate, and scale commands) you can start passing geometry to the graphics card.

There are different modes of sending geometry. Today we'll talk about "immediate" mode, and in the next class we'll talk about more optimal ways of sending the geometry from your application to the graphics card.

In immediate mode you signal that you are sending geometry by wrapping vertex commands with the glBegin and glEnd commands.

The primitive in the **glBegin(PRIMITIVE);** command can be any of the following:

```
GL_POINTS          //each vertex is a point
GL_LINES           //each two points is the start and end of a new line
GL_LINE_STRIP      //each point is the end of one line and the start of another
GL_LINE_LOOP       //as above, but a line is drawn between the last and first points
GL_TRIANGLES       //each three points defines a triangle
GL_TRIANGLE_STRIP  //draws a strip of connected triangles, re-using points to save space
GL_TRIANGLE_FAN    //a different way of drawing connected triangles
GL_QUADS           //every four points defines a quadrilateral
GL_QUAD_STRIP      //draws a strip of quadrilaterals
GL_POLYGON         //draws a convex polygon (need to use a Tesselation object for
                     more complex polygons, or build it out of triangle patches)
```

Drawing

```
glBegin(GL_QUAD_STRIP);

glVertex3f(-1f, 0f, 0f);    //V1
glVertex3f(-1f, 1f, 0f);    //V2
glVertex3f(0f, 0f, 0f);     //V3
glVertex3f(0f, 1f, 0f);     //V4
glVertex3f(1f, 0f, 0f);     //V5
glVertex3f(1f, 1f, 0f);     //V6

glEnd();



//draws:

V2 -- V4 -- V6
|     |      |
V1 -- V3 -- V5
```

```
see chapter 2 of the red book for examples of the various primitives
```

Drawing

You can specify a color for each of the points as well:

glColor3f(r, g, b);

also, if blending is enabled, you can specify an alpha channel:

glColor4f(r, g, b, a);

(Blending is enabled with the **glEnable(STATE)**; command)

```
glEnable(GL_BLEND);         //turn on blending
glBegin(GL_QUADS);          //switch to quad mode
glColor4f(1f,0f,0f,.5f);    //specify a color for the next two points
glVertex3f(-1f,-1f,0f);
glVertex3f(1f,-1f,0f);
glColor4f(0f,0f,1f,.5f);    //specify a color for the next two points
glVertex3f(1f,1f,0f);
glVertex3f(-1f,1f,0f);
glEnd();                    //turn off quad mode
glDisable(GL_BLEND);        //turn off blending
```

OpenGL will automatically interpolate colors if they change within a primitive.

Lighting

OpenGL has a built-in lighting model which lets you specify "per-vertex" ambient, diffuse, specular, and emmisive lighting.

(Nicer lights can be defined using a GLSL shader, which will cover in a few weeks.)

This model is a rough approximation of real light.

ambient = general lighting that lights the entire scene uniformly

diffuse = directional light that bounces of a surface uniformly

specular = directional light that bounces of a surface non-uniformly (used for "specular highlighting," or shininess)

emmissive = uniform lighting that emits from a particular surface

Lighting

For diffuse and specular lighting to work properly, you need to specify the normals of each vertex properly.

The normal is a vector (of length 1) that is perpendicular to a particular surface.

For example, if I define a simple rectangle as before, I can specify a single normal for the entire face:

```
glBegin(GL_QUADS);
glColorf3f(1f,1f,1f);
glNormal3f(0f, 0f, 1f);
glVertex3f(-1f,-1f,0f);
glVertex3f(1f,-1f,0f);
glVertex3f(1f,1f,0f);
glVertex3f(-1f,1f,0f);
glEnd();
```

Lighting

```
//DEFINE LIGHTS in some INIT function...
glEnable(GL_LIGHTING); //enable lighting

glEnable(GL_LIGHT0); //turn on one of the lights

//position the light
float pos[] = {1.0, 1.0, 0.0, 1.0}; //if the last value != 0, then light is "positional"

glLightfv(GL_LIGHT0, GL_POSITION, pos);

//define the lighting model
float ambient[] = {1.0, 0.0, 0.0, 1.0};
float diffuse[] = {0.0, 1.0, 0.0, 1.0};
float specular[] = {0.0, 0.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
```

Lighting

To calculate the normals by hand (which you normally have to do, unless you are using some of the built in shapes in the GLU or GLUT libraries) you can use the cross product of two vector built out of three vertices.

Lets say you have points P1, P2, P3, P4 defining a rectangle R.

```
P4 -- P3
|      |
P1 -- P2
```

You can define two vectors from V1 = P1→P2 and V2 = P1→P4

```
i.e., V1 = P2.x - P1.x, P2.y - P1.y, P2.z - P1.z, etc
```

Then you take the cross-product of V1 x V2, which creates a third vector, V3.

```
V3 = (V1.y*V2.z - V2.y*V1.z)*(V2.x*V1.z - V1.x*V2.z)*(V1.x*V2.y - V2.x*V1.y)
```

V3 points away from R at right angle. You then "normalize" V3 so that it is of unit length (where the length = 1)

```
Len = √(V3.x + V3.y + V3.z)
N3 = (V3.x/Len, V3.y/Len, V3.z/Len);
```

You can then use this normal N3 for your lighting model.

Lighting

By default, positional lights emit light in all directions. They also will not be blocked by any objects. (That is, a light at position 3f,3f,0f will equally light an object at 2f,2f,0f and 1f,1f,0f, even though the object at 2f,2f,0f is in between the light and the 1f,1f,0f object).

You can create a spotlight effect by specifying a direction and an angle for the cone of the spotlight:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 15f); //creates a spotlight with 15 degree radius

float[] direction = {-1f, -1f, 0f};
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, direction); //points the spotlight
```

Materials

You can also set the "material" properties of your objects using the "glMaterialf" commands.

if you use materials, you generally don't need to use glColor to set the colors.

Here are some common material commands:

```
//make the color of the object red
float[] objColor = { 1f, 0f, 0f, 1f };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, objColor);

//make the highlights white
float[] objHighlights = { 1f, 1f, 1f, 1f };
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, objHighlights);

//make the object super shiny (= more specular highlights)
float specularExponent = 20;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, specularExponent);
```

There is a lot more you can do with lighting and materials, but if you need more control it is much easier to do it in a GLSL fragment shader, which we'll cover in a few weeks.

Textures

Texturing allows you to place images on top of your geometry.

The general idea has the following steps:

```
A. loading and initializing the texture
1) load an image file (jpg, bmp, png, etc) into your program
2) generate space for this texture and give that space an ID
3) bind that texture to the GL context
4) set parameters for the texture
5) load the image into the texture

B. using the texture (within your display loop)
1) enable texturing
2) bind the pre-loaded texture to the GL context
3) begin drawing geometry
for each primitive:
   4) pass in texture coordinates
   5) pass in vertices
6) stop drawing geometry
7) disable texturing
```

Textures

This is a lot of steps... many platforms (JOGL, P5, OF) simplify the steps for you.

If you are using GLUT there are some good utility libraries that handle loading images for you.

Once you've loaded the image into the texture, it's sent directly to the graphics card, and using it thereafter is extremely fast. So in general, textures are loaded in the initialization phase of your application, but then used within the display loop.

Without going into too many details, here are commands for loading your image into a texture:

Textures

```
//load an image...
img = ...

//create space for one texture, which can be referred to by the int texID
int texID;
glGenTextures(1, &texID);

//bind the (empty) texture to the GL context
glBindTexture(GL_TEXTURE_2D, texID);

//define what happens if we go beyond the bounds of our texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

//define the filtering -- what happens if the geometry is bigger/smaller than the img
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_FILTER, GL_LINEAR);

//load the image into the texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, imgW, imgH, 0, GL_RGBA, GL_UNSIGNED_BYTE, img);
```

Textures

And here are the commands for applying the texture to geometry:

```
//turn on texturing
glEnable(GL_TEXTURE_2D);

//bind the (empty) texture to the GL context
glBindTexture(GL_TEXTURE_2D, texID);

//draw the geometry, indicating the texture coordinates
glBegin(GL_QUADS);
glTexCoord2f(0f, 0f);
glVertex3f(-1f, -1f, 0f);
glTexCoord2f(1f, 0f);
glVertex3f(1f, -1f, 0f);
glTexCoord2f(1f, 1f);
glVertex3f(1f, 1f, 0f);
glTexCoord2f(0f, 1f);
glVertex3f(-1f, 1f, 0f);
glEnd();

//turn off texturing
glDisable(GL_TEXTURE_2D);
```

Textures

Texture coordinates range from 0 to 1 in each direction.

That is, along the x-axis, 0f = 0% of the width, and 1f = 100% of the width, etc

If you go outside the bounds of the coordinates, it the texture will either clamp to the boundary, or repeat, depending on the parameter you've set when initially loading the image into the texture.

You don't necessarily need to include the full range of the texture, and you can potentially place multiple sub-images within a single image.

Textures are a common way in which to send data to the graphics card to be used by GLSL shader, which we'll explore in a few weeks.

GLU / GLUT geometry

GLU = OpenGL Utility library
GLUT = OpenGL Utility Toolkit

Both of these libraries have some common geometry which let you test out functionality quickly.

```
//GLU

//this
GLUquadricObj qObj = gluNewQuadric(); //load up new quadric
gluQuadricDrawStyle(qObj, GLU_FILL); //fill in the facets on the surface
gluQuadricNormals(qObj, GLU_SMOOTH); //make nice normals across the facets
//if lighting is enabled then the normals are already present
gluQuadricTexture(qObj, true); //automatically create texture coords for the object
//if you a bind a texture, then the texture coords are already present

gluSphere(qObj, 2f, 64, 64); //make a sphere with radius 2 and high resolution

//check out the glu library for other primitives
```

GLU / GLUT geometry

GLUT is simpler than GLU, but does not automatically create texture coordinates for you.

The "solid" version of the primitives will create normals, the "wire" version will not.

```
//GLUT
glutSolidSphere(2f, 64, 64); //make a sphere with radius 2 and high resolution

glutWireSphere(2f, 64, 64); //make a sphere with radius 2 and high resolution
```

check out the glut library for other primitives

Loading models from external programs

Instead of creating complicated models directly in OpenGL, you can use one of many existing 3D modeling programs, such as Maya3D, or Blender. Both of these programs (as well as lots of others) will let you save your model as an .OBJ file.

The simplest .OBJ files store your mesh as a set of vertices, normals, and texture coordinates.

More complicated ones will also have an associated .MTL file, which can also store material properties of different parts of your mesh.

There are lots of libraries out there which will load and display these OBJ files for you. Or you can write your own.