First Program

- Retrieve the file "square.cpp" from Moodle
- Review the code
- Note the following lines of code in the drawScene() function

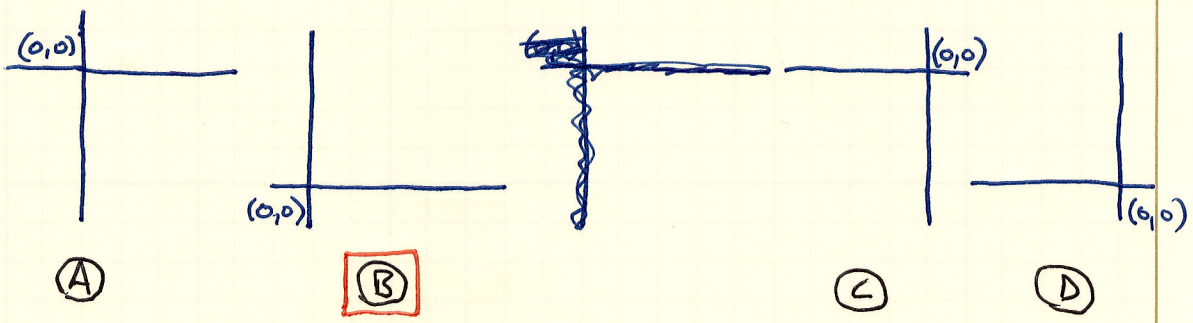
```
glBegin(GL_POLYGON);
  glVertex3f(20.0, 20.0, 0.0);
  glVertex3f(80.0, 20.0, 0.0);
  glVertex3f(80.0, 80.0, 0.0);
  glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

Note: This uses what we will call the classical approach, rather than the more modern approach which uses shaders.

- Let's run the code → it should draw a square inside a window

Coordinate Systems

- Q: How do the vertices correspond to actual coordinates, that is which of the following is the coordinate system used in OpenGL?



- Note: There is a z-component, which
  - Allows 3D representation, on a 2D view

```
glVertex3f(x, y, z)
```

Ortho Projection

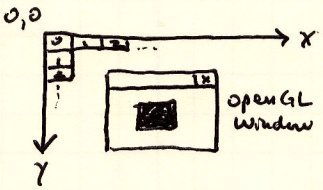
Q: What are the underlying units used in the coordinates?

A: There are none, this is left to the programmer's interpretation

- Thus they could be
  - mm
  - in
  - m
  - km, etc.

Note: Your selection of a base unit must be applied wholly across the system. Meaning all coordinates and later transforms must be relative to the base unit.

Windows



- Let us take a look at the main(...) function.
- Here we see the following two commands which control the display of the window:
  - glutInitWindowSize(w, h) – sets the initial size of the window to be w pixels in width and h pixels in height, in our case:

```
glutInitWindowSize(500, 500);
```

  - glutInitWindowPosition(x, y) – sets the initial position of the window to be at screen coordinate (x, y) for the upper left corner of the window. In our case we set the window as follows

```
glutInitWindowPosition(100, 100);
```

Ex: What happens to the square when we resize the window with glutInitWindowSize(...)?

- Try resizing to 300 x 300
- Try resizing to 500 x 250

Ex: What happens to the square when we resize the window at runtime?

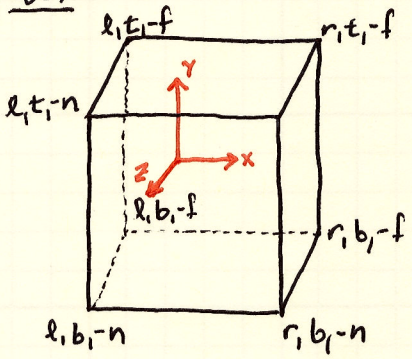
Q: Why does this happen?

A: This occurs because we are adjusting the aspect ratio which in turn distorts our view in Ortho Projections.

glOrtho and the Viewing Box

- The glOrtho(l, r, b, t, n, f) command defines a viewing box and a set of ~~boundaries~~ boundaries on what can be displayed

- The viewing box:



- Where:
  - l = Left boundary
  - r = right boundary
  - t = top boundary
  - b = bottom boundary
  - n = near boundary
  - f = far boundary

- We note that this is set in the resize(..) function
- Note: In OpenGL, near and far are flipped in sign
- The viewing box sits in 3D space (3space, $\mathbb{R}^3$, or World Space)
- This is, world space, where we create all our entities.

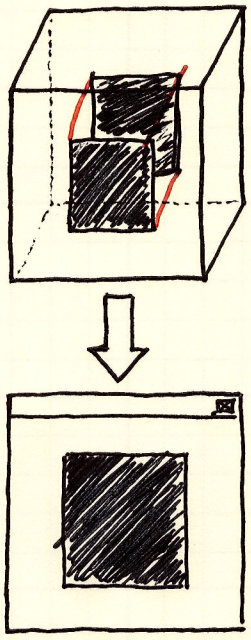
glOrtho Rendering Process

- glOrtho use a two step rendering process:

① Shoot ≡ Objects are projected perpendicularly onto the front face of the viewing box

Z = -near plane

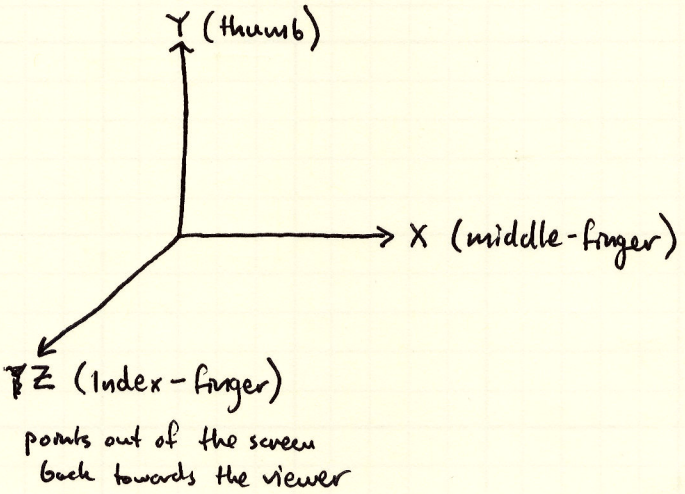
② Print ≡ Viewing box front face is proportionally scaled to fit the rectangular OpenGL window.



- The size and location of rendering is dependent on the ratio of the original object's size to that of viewing box in the direction of the viewing face.

glOrtho Rendering Process (cont'd)

- glOrtho Rendering assumes the following
  - The coordinate system is rectangular
  - The coordinate system is right-handed



Fixed World System

- Setup a right-handed rectangular coordinate system w/ axes calibrated identically
- Units do not matter, rather treat each axes as identical
- Imagine this system anywhere you wish, but it remains fixed
- This creates the world coordinate system, in which the following exist
  - The viewing box
  - All objects we create

- ~~As noted before~~

- This is also called, object space
- OpenGL then projects this into our view in the window as a rectangle
  - This rectangle is called screen space
  - Note we define this via: glViewport(x, y, w, h)
    - w, h: width and height of the view port w/in its containing window.
    - x: x-coordinate in window of view port upper-left corner
    - y: y-coordinate in window of view port upper-left corner

Fixed World Coordinates (cont'd)

- Currently, our object space is constrained by the command

```
glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, -1.0)
```

Ex: How does our view change if we use

```
glOrtho(0.0, 200.0, 0.0, 200.0, -1.0, -1.0);
```

?

```
glOrtho(20.0, 80.0, 20.0, 80.0, -1.0, -1.0);
```

?

Note: when setting the parameters of glOrtho, it is wise to maintain the following constraints:

- left < right
- bottom < top
- near < far

Note: To minimize distortions the aspect ratio of the viewport/window should be set to 1:1, 4:3, etc. by controlling the width and height parameters of

- glutInitWindowSize(...)
- glViewport(...)

Exp: Let us add an additional square, just after the existing square in the drawScene(...) function. Use the following code:

```
glColor3f(0.0, 1.0, 0.0);
glBegin(GL_POLYGON);
    glVertex3f(40.0, 40.0, 0.0);
    glVertex3f(60.0, 40.0, 0.0);
    glVertex3f(60.0, 60.0, 0.0);
    glVertex3f(40.0, 60.0, 0.0);
glEnd();
```

What do you see?

Why did this happen?

Now remove the last vertex of the first square.

- What do you see now?
- What does this tell us?

Color and Interpolation

- In OpenGL there a 3 basic color functions

`glColor3f(r,g,b)` - sets the current drawing/foreground color

r - parameter which sets the intensity of the red channel of the color. Hard bounded to [0,1]
g - parameter which sets the intensity of the green channel of the color. Hard bounded to [0,1]
b - parameter which sets the intensity of the blue channel of the color. Hard bounded to [0,1]

r, b, and g values must be between 0.0 and 1.0, values less than 0.0 will be set to 0.0 and values greater than 1.0 will be set to 1.0.

`glClearColor(r,g,b,a)` - sets the background color

r,g,b - same as for glColor3f

a - The intensity of the color's alpha (transparency) channel. Hard bounded to [0,1].

`glClear(GL_COLOR_BUFFER_BIT)` - clears the color buffer and sets the value of all projected pixels to the color set by glClearColor(...).

- Common Colors

R,G,B Triple	Color Name	
1.0, 0.0, 0.0	Red	Reducing intensity equally for each color reduces the brightness of the produced color thus:
0.0, 1.0, 0.0	Green	
0.0, 0.0, 1.0	Blue	1.0, 1.0, 0.0 → Brightest Yellow
1.0, 1.0, 0.0	Yellow	0.5, 0.5, 0.0 → Less bright Yellow
1.0, 0.0, 1.0	Magenta	
0.0, 1.0, 1.0	Cyan	
1.0, 1.0, 1.0	White	→ Reducing intensities equally creates various grays
0.0, 0.0, 0.0	Black	

For Next Time

1. Read Ch. 2 sections 2.4, 2.6-2.11
2. Review Programming Assignment/Homework 1
3. Make sure you can compile and execute the example code

Additional Notes