Geometry: GLUT Basics and 2-D Transformations

Course web page:

http://goo.gl/EB3aA

Outline

- Quick peek at compiling in Ubuntu
- OpenGL & GLUT basics
  - Setting up a program
  - 2-D drawing
  - 2-D transformations

My coding environment

- Ubuntu
  - Here I've installed it to a virtual machine using VirtualBox
  - Add compiler (g++), GLUT library (freeglut3-dev) with "sudo apt-get install …"
  - I also use xemacs as text editor; added with "apt-get install xemacs21" (plus syntax highlighting)
- Makefile
  - Specifies .cpp files that are part of "project", plus libraries and any path info needed
  - Just type "make" in that directory and you get an executable

OpenGL – What is It?

- **GL** (**G**raphics **L**ibrary): Library of 2-D, 3-D drawing primitives and operations
  - API for 3-D hardware acceleration
- **GLU** (**GL U**tilities): Miscellaneous functions dealing with camera set-up and higher-level shape descriptions
- **GLUT** (**GL U**tility **T**oolkit): Window-system independent toolkit with numerous utility functions, mostly dealing with user interface

- Course web page has links to online function references (functions from each library start with library prefix—i.e., **`gl*`**, **`glu*`**, **`glut*`**)

Event-driven GLUT program structure

1. Configure and open window
2. Initialize OpenGL state, program variables
3. Register callback functions
   - Display (where rendering occurs)
   - Resize
   - User input: keyboard, mouse clicks, motion, etc. (on Thursday)
4. Enter event processing loop

Portions of some slides adapted from "An Interactive Introduction to OpenGL Programming", D. Shreiner, E. Angel, V. Shreiner, SIGGRAPH 2001 course

Simple OpenGL program (no animation)

```
#include <stdio.h>
#include <GL/glut.h>

void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutInitWindowSize(100, 100);
    glutCreateWindow("hello");
    init();                          // set OpenGL states, variables

    glutDisplayFunc(display);        // register callback routines

    glutMainLoop();                  // enter event-driven loop
}
```

Simple OpenGL program (no animation)

```
#include <stdio.h>
#include <GL/glut.h>

void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
    glutInitWindowSize(100, 100);
    glutCreateWindow("hello");
    init();                          // set OpenGL states, variables

    glutDisplayFunc(display);        // register callback routines

    glutMainLoop();                  // enter event-driven loop
}
```

Something to know about OpenGL:
It uses global state a lot

adapted from E. Angel

Initialization

- **`glutInit:`** Pass command-line flags on to GLUT
- **`glutInitDisplayMode:`** OR together bit masks to set modes on pixel type (indexed vs. true color), single- vs. double-buffering, etc.
- **`glutInitWindowSize, glutCreateWindow:`** Set drawing window attributes, then make it
- **`init()`** (my function): Set OpenGL state, program variables
  - Use GL types/typedefs **`GLfloat`**, **`GLint`**, **`GL_TRUE`**, **`GL_FALSE`**, etc. for cross-platform compatibility
  - Most notable function here: **`glClearColor(0, 0, 0, 0);`**
  - **`glOrtho()`** explained in a few slides; other functions will wait a few lectures

Let's see how to look these up…

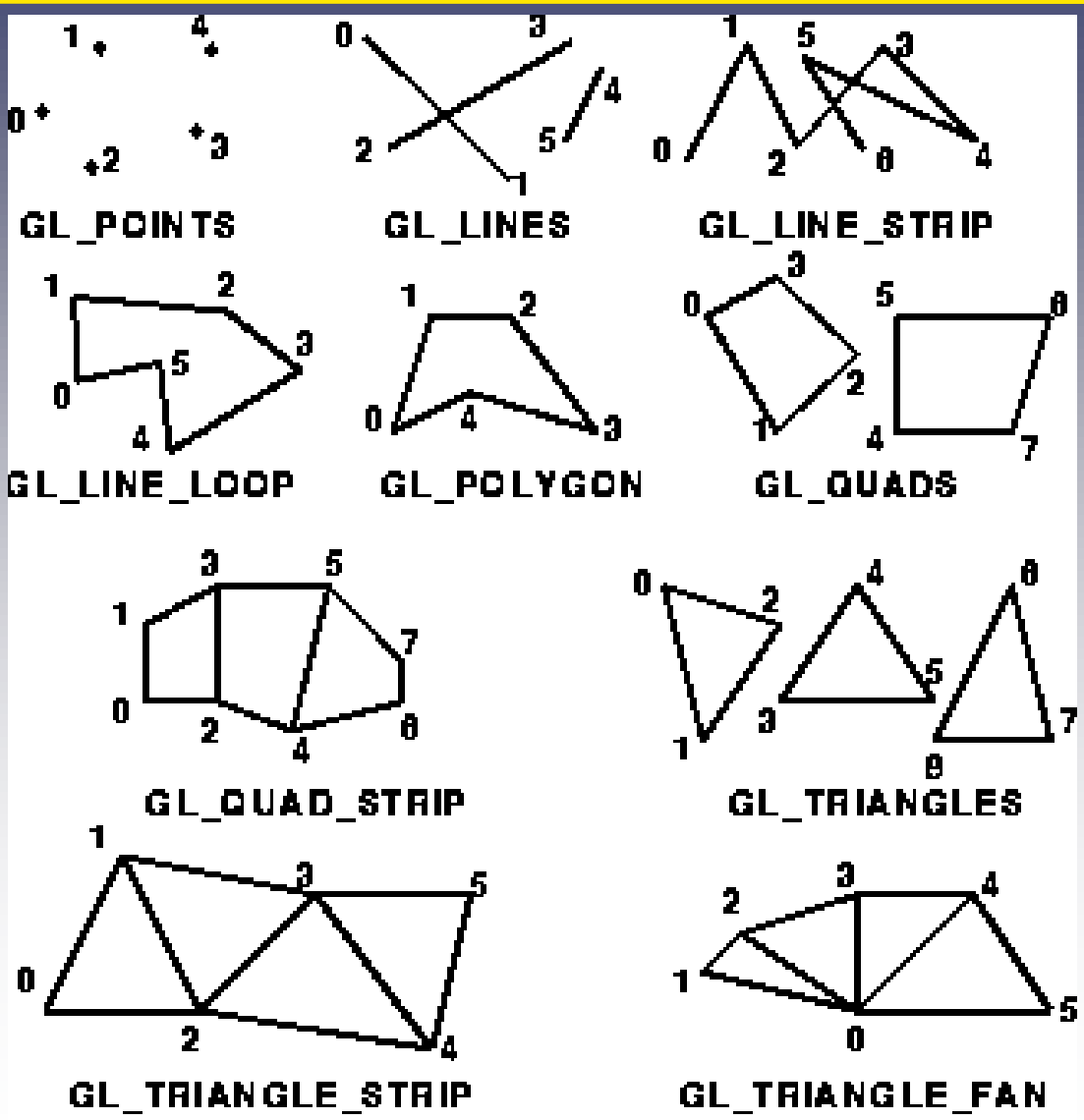
Simple OpenGL program

```
#include <stdio.h>
#include <GL/glut.h>

void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
    glutInitWindowSize(100, 100);
    glutCreateWindow("hello");
    init();                          // set OpenGL states, variables

    glutDisplayFunc(display);        // register callback routines

    glutMainLoop();                  // enter event-driven loop
}
```

Rendering Steps (no animation)

- **`glutDisplayFunc(display)`** registers a function called "display" (could have any name though) as a "callback" when it's time to draw the window
  - When might this happen?
- What does **`display()`** do?
  1. Clear window: **`glClear(GL_COLOR_BUFFER_BIT)`**
  2. Draw shapes
     - Set colors, patterns, point/line sizes
     - Specify type of geometric primitive(s) and list vertices
  3. Commands aren't necessarily executed immediately, so force completion with **`glFlush()`**

OpenGL Geometric Primitives



Specifying Geometric Primitives

- Primitives are specified using

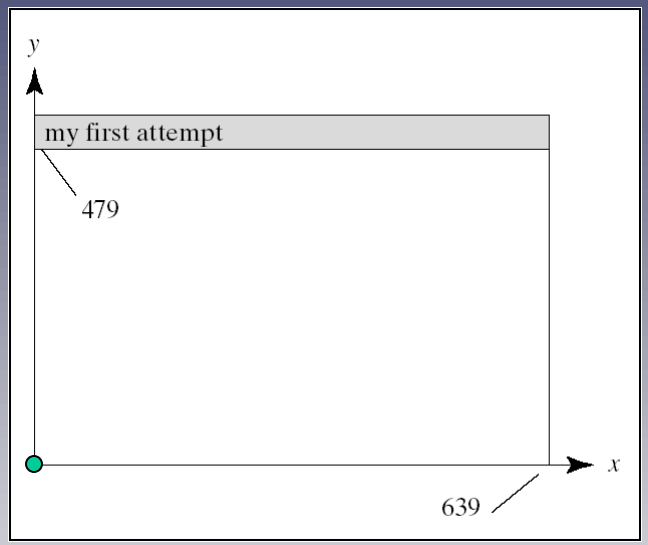
```
glBegin(primType);
...
glEnd();
```

  - *primType* determines how vertices are combined

```
GLfloat red, green, blue;
GLfloat x, y;

glBegin(primType);
for (i = 0; i < nVerts; i++) {
 // set color, coord. values
 glColor3f(red, green, blue);
 glVertex2f(x, y);
}
glEnd();
```

OpenGL screen coordinates

- Bottom left corner is origin

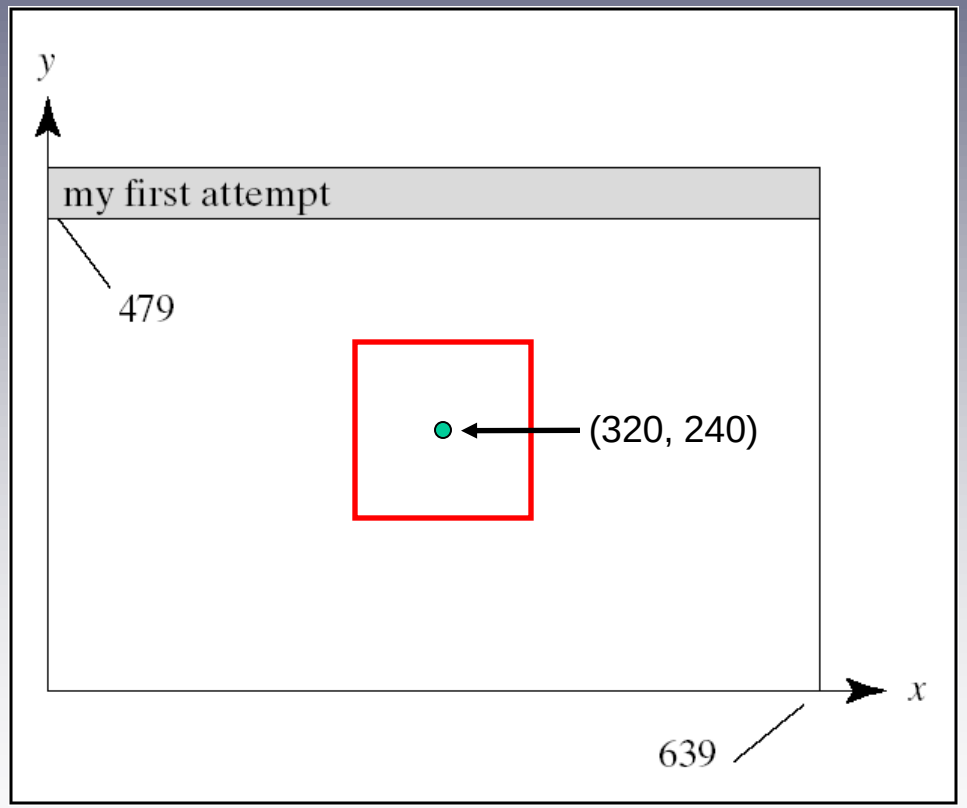


from Hill

- **`gluOrtho2D()`** sets the units of the screen coordinate system (just a wrapper around **`glOrtho()`**)
  - **`gluOrtho2D(0, w, 0, h)`** means the coordinates are in units of pixels
  - **`gluOrtho2D(0, 1, 0, 1)`** means the coordinates are in units of "fractions of window size" (regardless of actual window size)

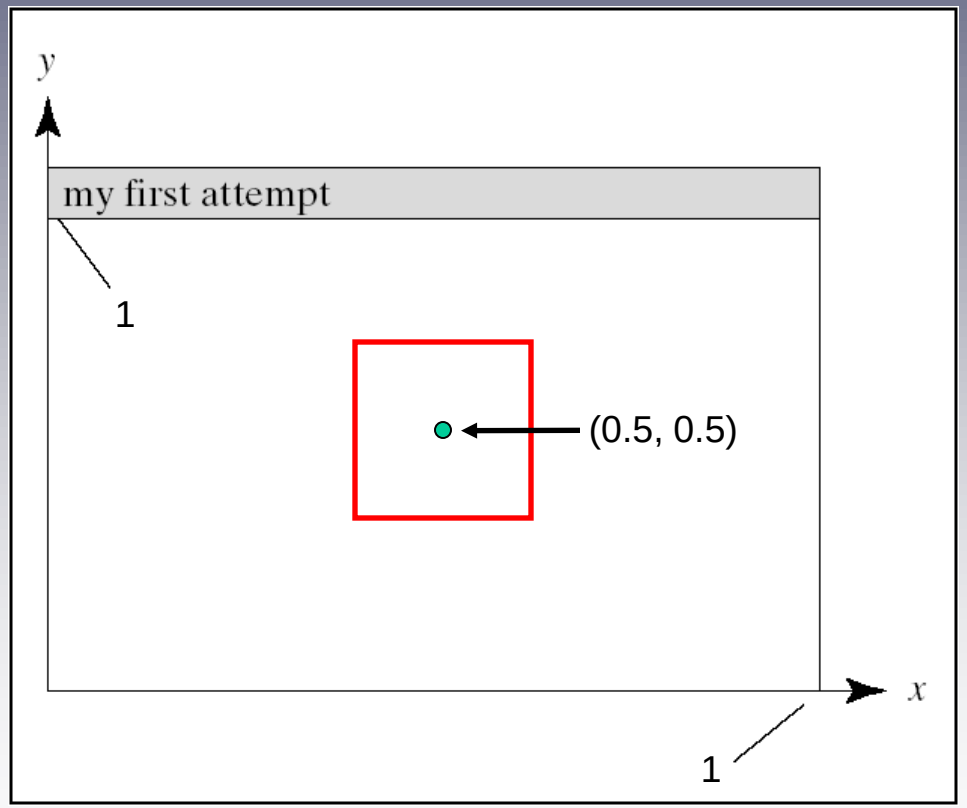
Example: Specifying the center of a square

```
gluOrtho2D(0, 640, 0, 480)
```



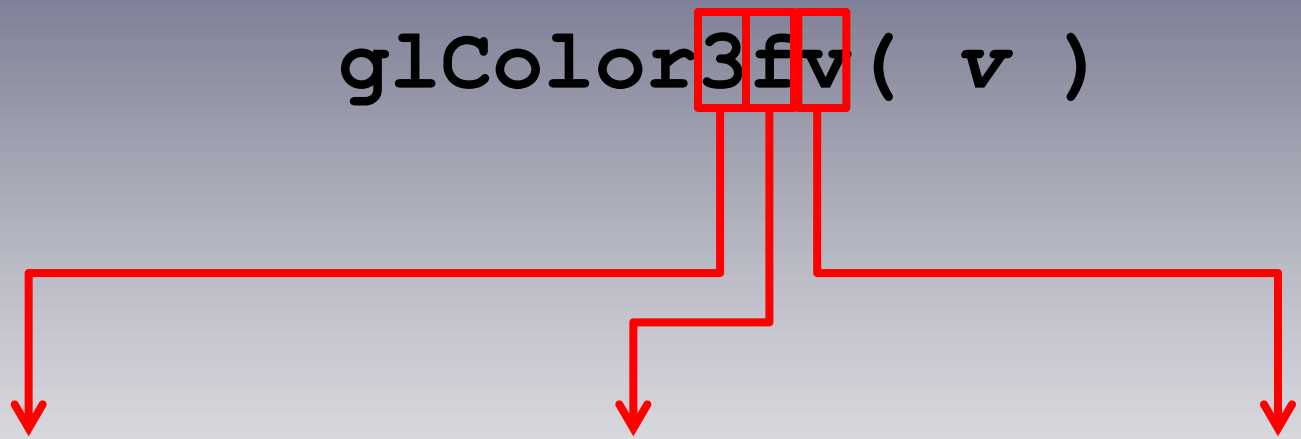
Example: Specifying the center of a square

```
gluOrtho2D(0, 1, 0, 1)
```



OpenGL Command Formats

```
glVertex3fv( v )
glColor3fv( v )
```



Number of components

```
2 - (x,y)
3 - (x,y,z),
    (r,g,b)
4 - (x,y,z,w),
    (r,g,b,a)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

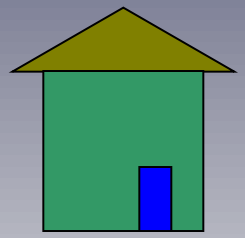
omit "v" for scalar form–
e.g.,
glVertex2f(x, y)
glColor3f(r, g, b)

Drawing: Miscellaneous

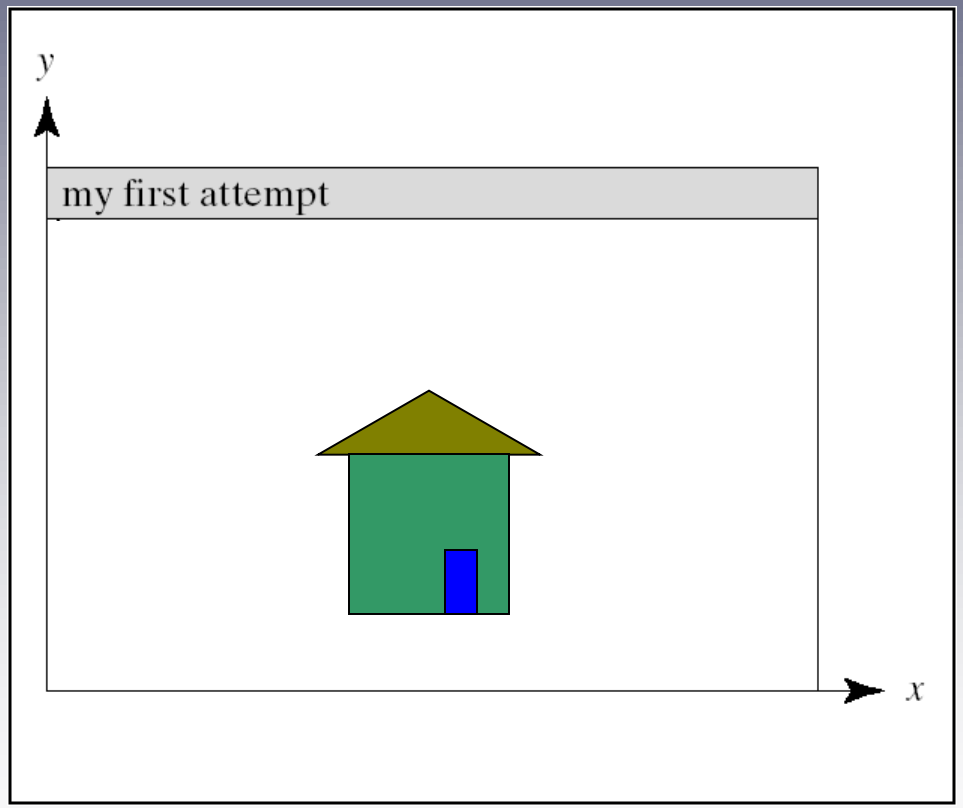
- **`glColor()`** range
  - [0, 1] for each color channel for **`glColor3f()`**
  - [0, 255] for **`glColor3ub()`**
- Can set persistent "pen size" outside of **`glBegin()/ glEnd()`**
  - **`glPointSize(GLfloat size)`**
  - **`glLineWidth(GLfloat width)`**
- **`glRect(x1, y1, x2, y2)`** specifying opposite corners of rectangle is equivalent to GL_POLYGON with four vertices listed (i.e., filled)

Why We Need Transformations

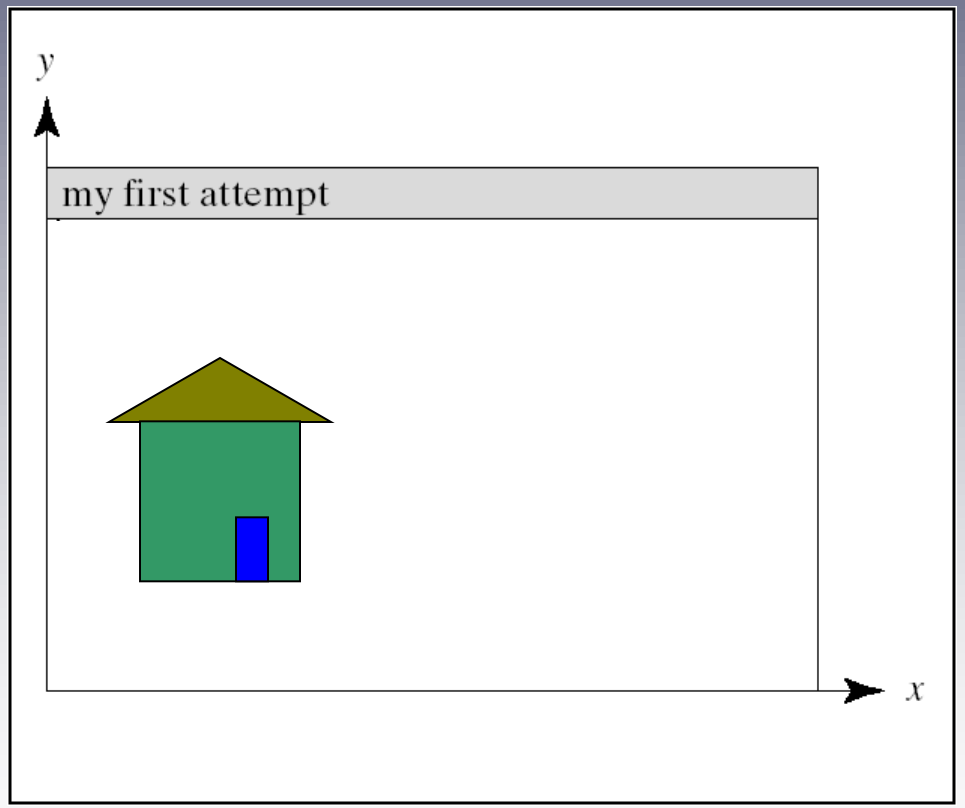
- What can we do so far?
  - Draw 2-D shapes by exhaustively listing their vertices
- Something missing:
  - The ability to specify the intrinsic shape of an object independently of its location, scale, or orientation



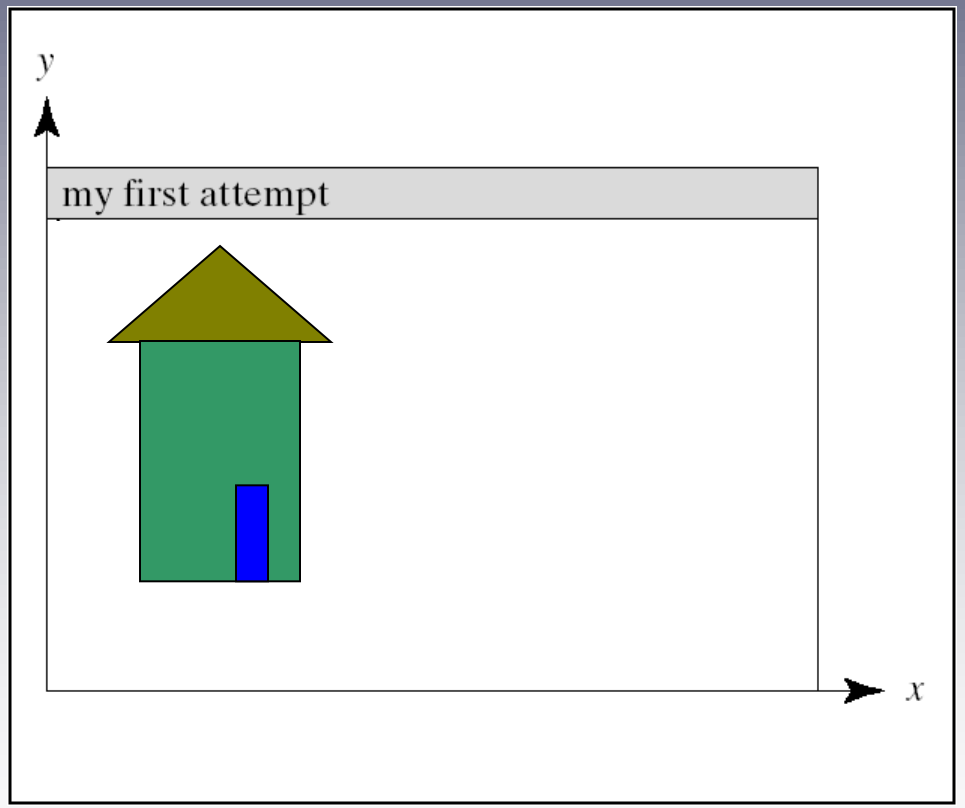
Example: Shape vs. Viewing Issues



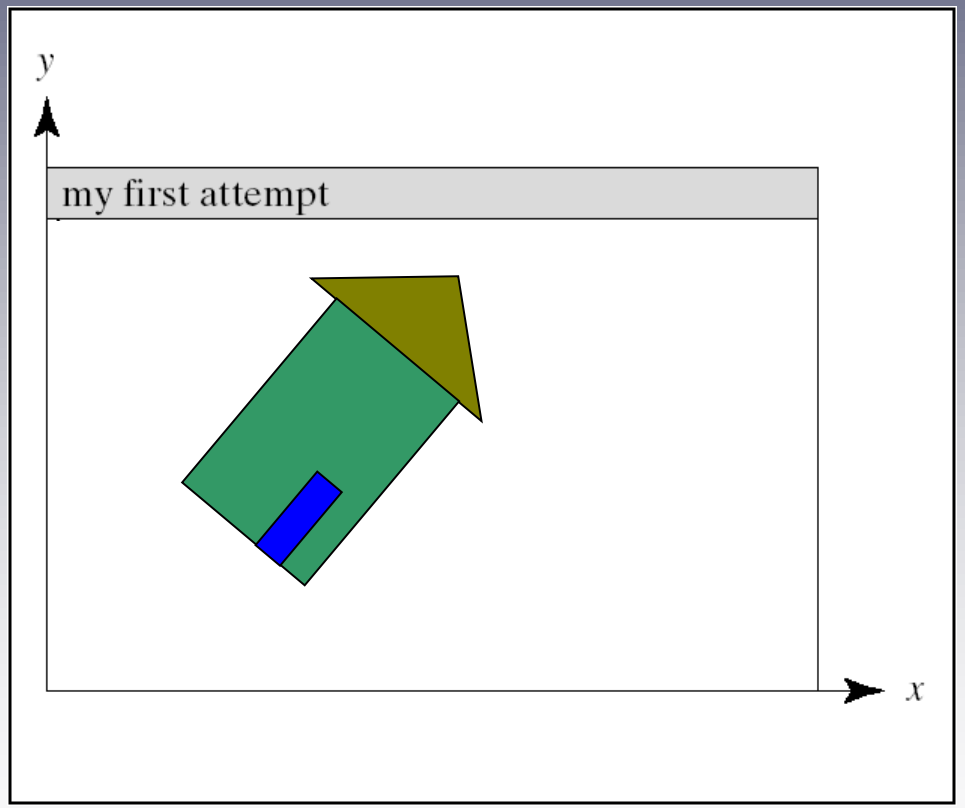
Example: Shape vs. Viewing Issues



Example: Shape vs. Viewing Issues



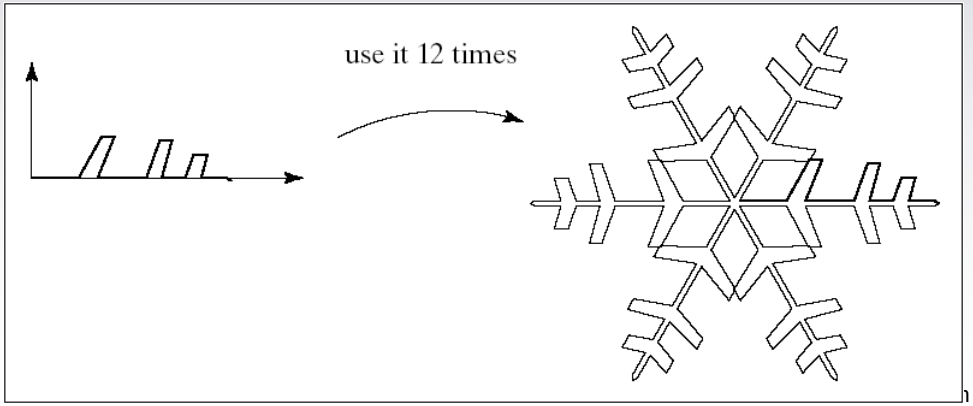
Example: Shape vs. Viewing Issues



Transformations for modeling, viewing

1. Make object model in canonical coordinate frame
2. Transform object with appropriate translation, scaling, rotation as needed

- Also useful for building complex objects from simpler parts

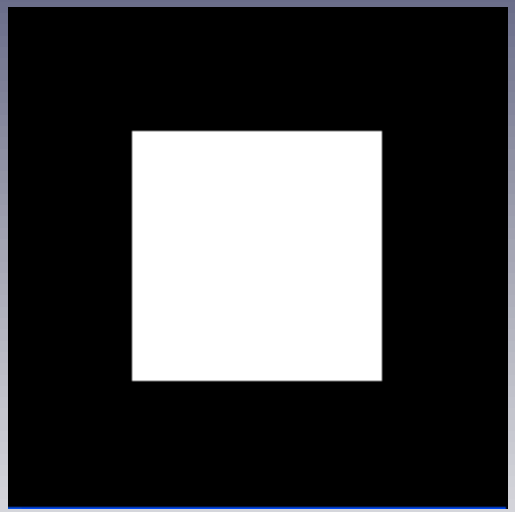


2-D Transformations: OpenGL

- 2-D transformation functions*
  - **`glTranslatef(x, y, 0)`**
  - **`glScalef(sx, sy, 0)`**
    - Negative scaling is reflection
  - **`glRotatef(theta, 0, 0, 1)`** (angle in degrees; direction is counterclockwise)
- Notes
  - Set **`glMatrixMode(GL_MODELVIEW)`** first
  - Transformations should be specified **before** drawing commands to be affected
  - Multiple transformations are applied in **reverse** order

*Technically, these are 3-D—we're just ignoring 1 dim.

Example: 2-D Translation in OpenGL



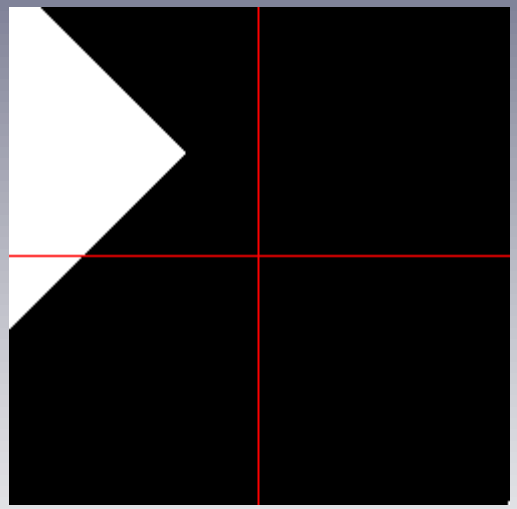
Two ways to do this:

```
glRectf(.25,.25,.75,.75);
```

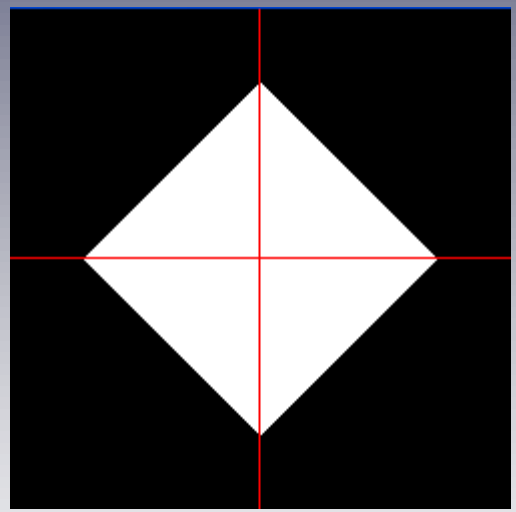
```
glTranslatef(.5,.5,0);
glRectf(-.25,-.25,.25,.25);
```

Assuming `gluOrtho2D(0, 1, 0, 1)`

Suppose we want to draw a diamond



```
glRotatef(45,0,0,1);
glTranslatef(.5,.5,0);
glRectf(-.25,-.25,.25,.25);
```



```
glTranslatef(.5,.5,0);
glRotatef(45,0,0,1);
glRectf(-.25,-.25,.25,.25);
```

How can we get this?

Remember: Order of application is **backwards** from drawing commands

Why does the order of transformations seem backwards?

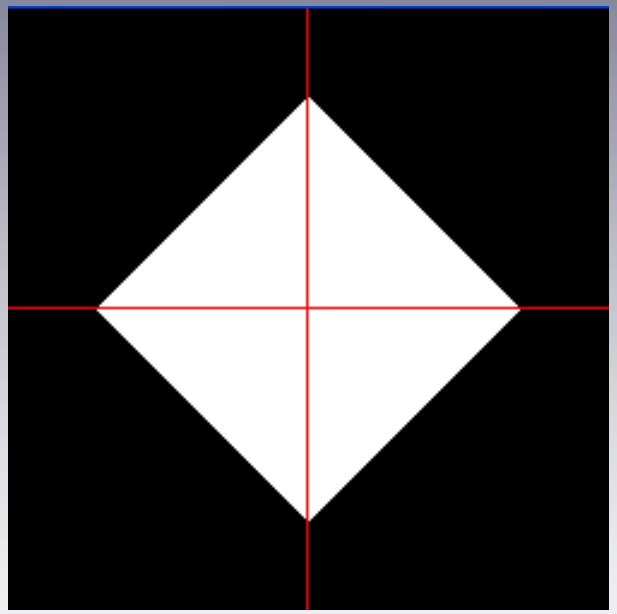
- Because (as we will learn), transformations are implemented with matrix multiplications
- Thus the rules of linear algebra dictate that **T** x **R** x **p** means "rotate **p**, then translate it"
- Can also think of it functionally: **T**(**R**(**p**))

Limiting "Scope" of Transformations

- Transformations are ordinarily applied to **all** subsequent draw commands
- Useful to think of a *transformation stack*
- To limit effects, use push/pop functions:

```
glPushMatrix();
// transform
// draw affected by transform
glPopMatrix();
// draw unaffected by transform
```

Example: Pushing, popping transformations



```
glPushMatrix();
glTranslatef(.5,.5,0);
glRotatef(45,0,0,1);
glRectf(-.25,-.25,.25,.25);
glPopMatrix();

glPushMatrix();
// draw axis lines
glPopMatrix();
```

3-D Analogies

- GL_MODELVIEW transformation is about moving scene objects (MODEL) or virtual camera (VIEW) relative to each other
- GL_PROJECTION transformation is about lens properties of camera
  - gluOrtho2D() is one wrapper for this
- Both described by matrices, each has separate stack