

Assignment 1

CSE471: Computer Graphics

Fall 2022

Computer Science and Engineering

Instructor: Ilwoo Lyu

Objectives

- The purpose of this assignment is to learn:
 - OpenGL primitives
 - Coordinate systems (local \leftrightarrow world \leftrightarrow screen)
 - GLUT event handler
 - Basic geometric transformations
- You will create your own interactive tool
 - You will keep building up your tool for all the upcoming assignments

Introduction

OpenGL

- **Open Graphics Library**

- 2D, 3D computer graphics API
- Cross-platform (hardware-independent)
- Developed by Silicon Graphics Inc. in 1992
- Managed by a non-profit technology consortium, the Khronos Group
- Latest stable version: 4.6 (released in 2017)



OpenGL

- Graphic library for rendering primitives
 - Drawing geometric objects
 - Viewing transformation
 - Color / Lighting
 - Blending, Antialiasing, and Fog
 - Display List
 - Drawing pixels, Bitmaps, Fonts, and Images
 - Texture Mapping
 - Frame Buffer
 - Evaluators and NURBS
 - Selection and Feedback
- No high-level modeling commands

OpenGL-related Libraries

- GLU (OpenGL Utility Library)
 - additional high-level routines
- GLUT (OpenGL Utility Toolkit)
- GLUI (OpenGL User Interface Library)
- GLEE, GLEW ...

Important Concepts

- OpenGL Context
 - The context contains all of the information that will be used by the OpenGL system to render, when the system is given a rendering command. A context effectively is OpenGL, because OpenGL cannot be used without one.
- State
 - The OpenGL context contains information used by the rendering system. This information is called State, which has given rise to the saying that OpenGL is a "state machine".
 - Your computation (transformations, colors, etc.) is stored as "global" in a context
- See https://www.khronos.org/opengl/wiki/Portal:OpenGL_Concepts

A Simple OpenGL Program

- Example 1

```
glClearColor(0.0, 0.0, 0.0, 0.0) # clear background
glClear(GL_COLOR_BUFFER_BIT)      # clear color buffer
glBegin(GL_TRIANGLE)              # begin primitive
glColor3f(1.0,0.0,0.0)             # current state -> red
glVertex2f(0, 1)                   # vertex 0
glColor3f(0.0,1.0,0.0)             # current state -> green
glVertex2f(-1, 1)                  # vertex 1
glColor3f(0.0,0.0,1.0)             # current state -> blue
glVertex2f(1, -1)                  # vertex 2
glEnd()                           # end primitive
```

There are several ways to draw triangles without glBegin/glEnd as it is deprecated in modern OpenGL. We will see some other ways later but follow the above example at this moment for simplicity.

A Simple OpenGL Program

- Example

```
def draw():  
    glBegin(GL_TRIANGLE)  
    glColor3f(1.0,0.0,0.0)  
    glVertex2f(0, 1)  
    glColor3f(0.0,1.0,0.0)  
    glVertex2f(-1, 1)  
    glColor3f(0.0,0.0,1.0)  
    glVertex2f(1, -1)  
    glEnd()
```

```
draw() # original triangle appears  
mat = some transform  
glMultMatrixf(mat) # global state: mat  
draw() # a transformed triangle by mat  
draw() # a transformed triangle by mat  
glMultMatrixf(mat) # global state: mat * mat  
draw() # a transformed triangle by mat * mat  
glLoadIdentity() # global state: Identity  
draw() # original triangle appears
```

Command Syntax

- Function : gl~
- Constant : GL_~
- Type indicating suffix
 - b : signed char (GLbyte)
 - s : short (GLshort)
 - l : long (GLint, GLsizei)
 - f : float (GLfloat)
 - d : double (GLdouble)
 - ub : unsigned char (GLubyte, GLboolean)
 - us : unsigned short (GLushort)
 - ui : unsigned int (GLuint, GLenum, GLbitfield)
 - v : vector type

```
glColor3f(1.0, 0.0, 0.0)
color_array=[1.0, 0.0, 0.0]
glColor3fv(color_array)
```

GLUT

- The OpenGL Utility Toolkit (from its official site)
 - GLUT (pronounced like the glut in gluttony) is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs.
 - It implements a simple windowing application programming interface (API) for OpenGL.
 - GLUT makes it considerably easier to learn about and explore OpenGL programming.
 - GLUT provides a portable API so you can write a single OpenGL program that works on both Win32 PCs and X11 workstations.

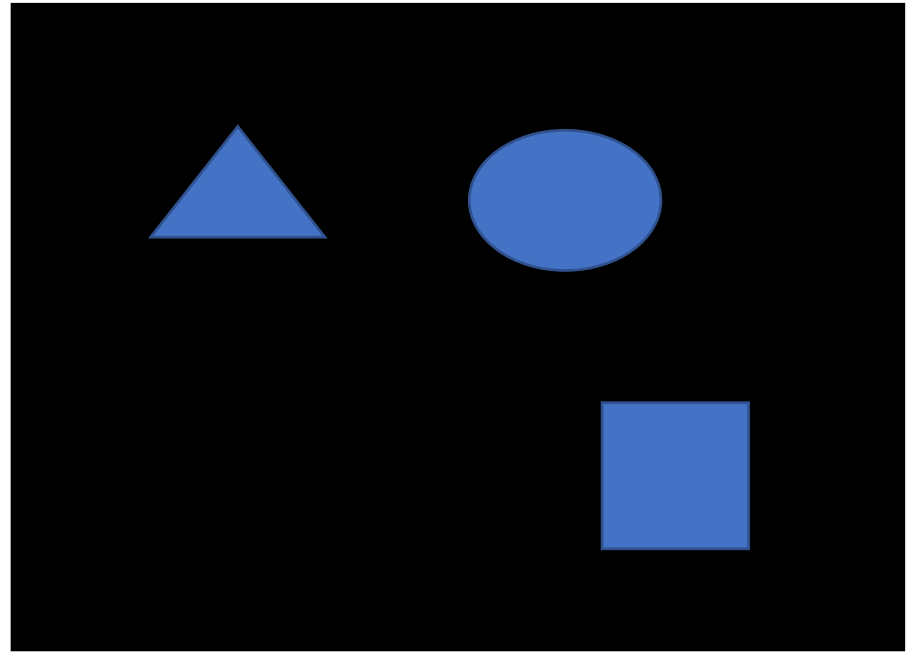
GLUT

- GLUT is based on call-back routines
 - You will bind your function to each built-in call-back function
 - Example: `glutDisplayFunc(display)` # you implement "display" function and GLUT will call this when events occur
- You can customize your interface window like
 - `glutInit()` # init OpenGL context
 - `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)` # display mode
 - `glutInitWindowSize(800, 600)` # window size
 - `glutInitWindowPosition(0, 0)` # window position
 - `glutCreateWindow("title goes here")` # window title

Assignment 1

Specifications

- Simple 2D interactive tool
 - You will add/remove predefined solid (filled) polygons (triangle, rectangle, and ellipse) as many as you want on your screen
 - You will manipulate global/local transformations




Specifications


- Polygons deployment

1. Choose a polygon type by pressing a key (1: triangle, 2: rectangle, 3: ellipse, esc: cancellation of polygon drawing mode)
2. Click a desirable location of your screen, where a polygon will be located
3. The center of the polygon should appear at the location you click
4. A user can add as many polygons as possible

Example



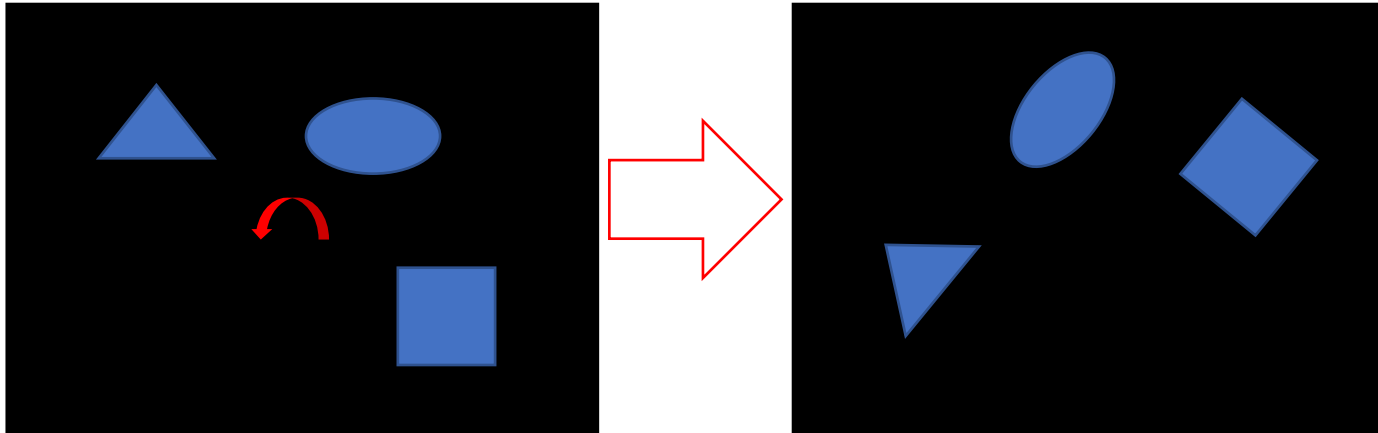
Press "1" and click here



A triangle appears on which you click
(you don't need to draw the center)

Specifications

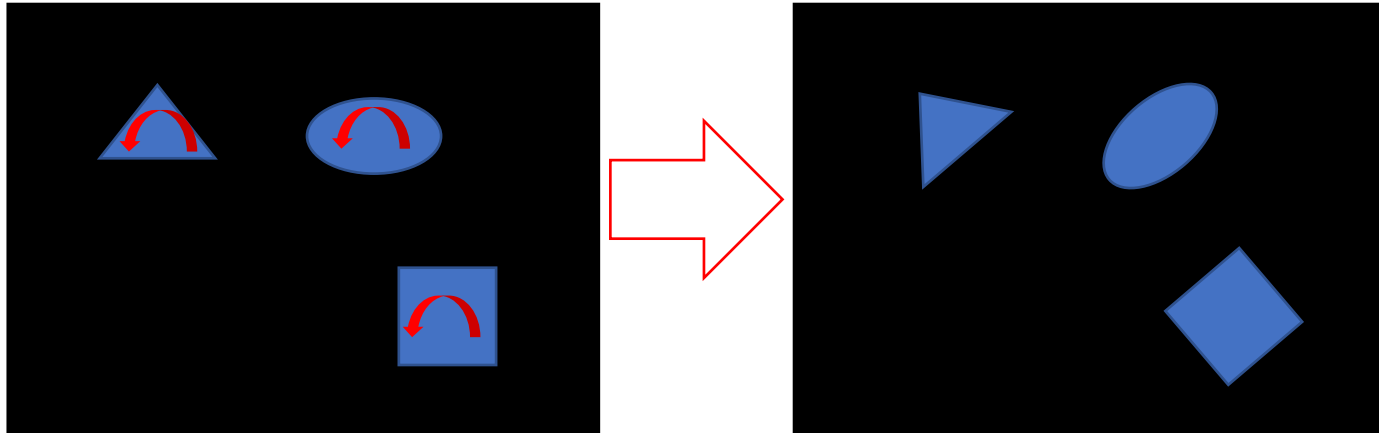
- Polygon manipulation
 - Global transformation occurs at the center of your screen



Example: counterclockwise global rotation (polygon center is changed)

Specifications

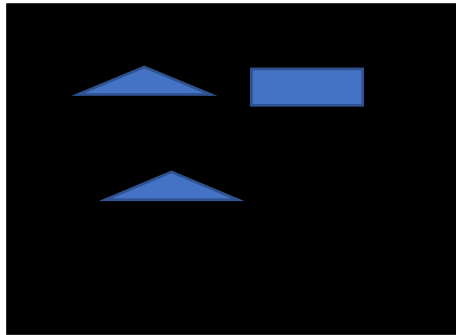
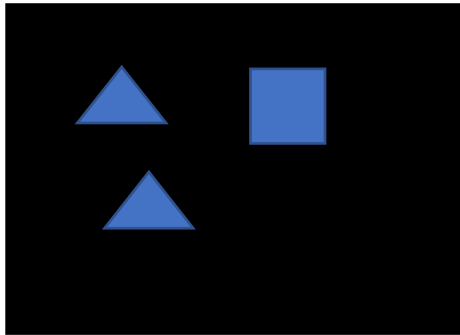
- Polygon manipulation
 - Local transformation occurs at the center of each polygon



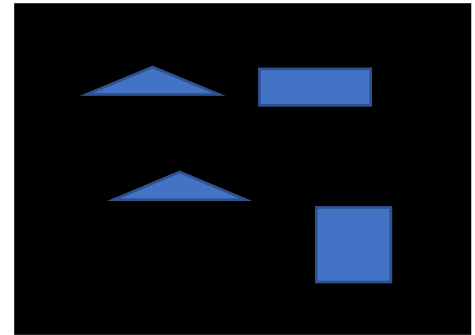
Example: counterclockwise local rotation (polygon center is not changed)

Specifications

- Transformations should be accumulated
 - For example, if you scale down your polygons and place a new polygon, the new polygon should have a default scale



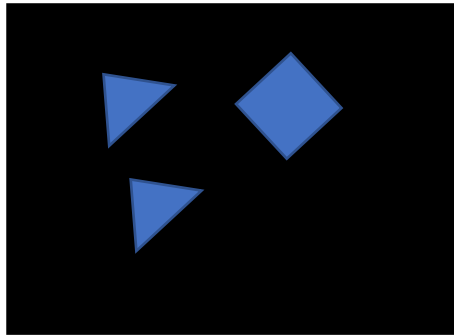
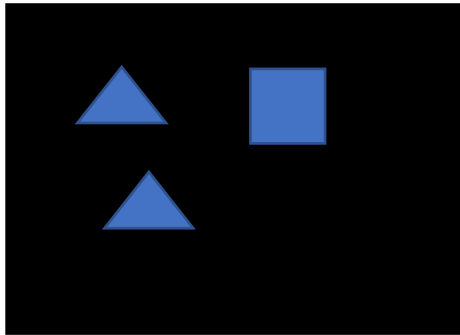
Scale down



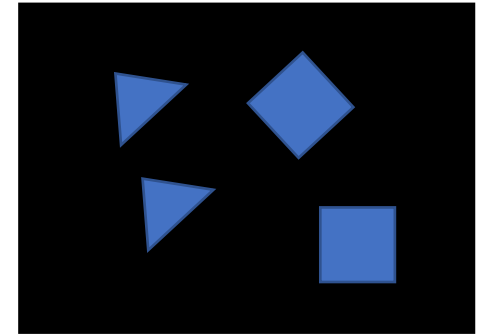
Place a new polygon

Specifications

- Transformations should be accumulated
 - For example, if you rotate your polygons and place a new polygon, the new polygon should have a default orientation



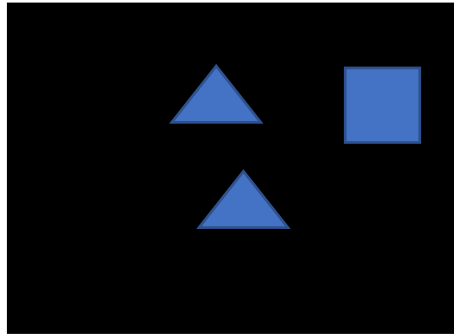
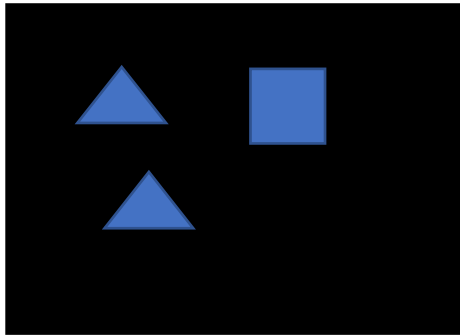
Rotation



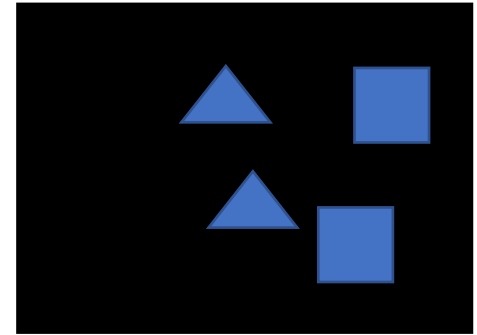
Place a new polygon

Specifications

- Transformations should be accumulated
 - For example, if you translate your polygons and place a new polygon, the new polygon should appear exactly where you click



Translation



Place a new polygon

Task 1

- Implement primitive polygons in your draw function
- In the skeleton code, create a subclass of "Polygon" and implement draw() function
 - Triangle: (0, 0.1, 0), (-0.1, -0.1, 0), (0.1, -0.1, 0)
 - Rectangle: (-0.1, 0.1, 0), (0.1, 0.1, 0), (0.1, -0.1, 0), (-0.1, -0.1, 0)
 - Ellipse: vertical radius=0.05, horizontal radius=0.1 (for ellipse drawing, you can sample as many points as you want)
 - You can assign whichever colors you like
- Use GL_TRIANGLES for triangles, GL_QUADS for rectangles, GL_TRIANGLE_FAN for ellipses

Task 2

- Implement keyboard/mouse event handlers
- You will need:
 - Conversion between window coordinates (i.e., screen size, origin=top-left corner) and world coordinates (i.e., $[-1,1] \times [-1,1]$, origin=window center)
 - Use this conversion when you compute the amount of each transformation
 - Example: window width = 800 px, 80 px difference becomes 1/10
 - Keyboard event handler
 - 1: triangle, 2: rectangle, 3: ellipse, esc: cancellation of polygon drawing mode

Task 3-1

- Implement basic transformation functions

- Scale

- `scale(sx, sy)`:

- Inputs: `sx`: double, `sy`: double

- Returns 4-by-4 matrix

- You will need to update only first two columns & rows

- Use numpy's `eye(4)` to make an identity matrix

- Use the coordinate conversion you implement in Task 2

- You may want to adjust "`sx`" & "`sy`" if transformation is too slow/fast

$$\begin{bmatrix} ? & ? & 0 & 0 \\ ? & ? & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 3-2

- Implement basic transformation functions
- Rotation
 - rotation(degree):
 - Inputs: degree: double
 - Returns 4-by-4 matrix
 - You will need to update only first two columns & rows
 - Use numpy's eye(4) to make an identity matrix
 - Use the coordinate conversion you implement in Task 2
 - You may want to adjust "degree" if transformation is too slow/fast

$$\begin{bmatrix} ? & ? & 0 & 0 \\ ? & ? & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 3-3

- Implement basic transformation functions

- Translation

- translation(dx, dy):

- Inputs: dx: double, dy: double

- Returns: 4-by-4 matrix

- You will need to update only last column's first two rows

- Use numpy's eye(4) to make an identity matrix

- Use the coordinate conversion you implement in Task 2

- Note: we will learn why translation has the above matrix form later

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 3-1, 3-2, 3-3

- Hints

- Use @ operator for matrix multiplication in numpy (do not be confused with *, which is for element-wise multiplication)
- OpenGL uses a column-major format for matrix whereas a row-major format in numpy by default. You will thus need to transpose your numpy matrix to feed transformations to OpenGL (Use .T for transpose in numpy)

Task 4

You can bind different keys/mouse behaviors, but if so, you must describe how to operate your tool

- Implement polygon manipulation (see event handlers in the skeleton code)
 - Press "g" to toggle global/local transformations
 - Rotation: drag mouse
 - left→right or down→up: increase rotation degree (counterclockwise)
 - right→left or up→down: decrease rotation degree (clockwise)
 - Scaling: ctrl+drag mouse
 - left→right / right→left: increase/decrease scale along local/world x-axis
 - down→up / up→down: increase/decrease scale along local/world y-axis
 - Translation: arrow keys
 - Move along local/world axes

Local axes: polygon's coordinates (origin: polygon center)

Task 4

- Implement polygon manipulation (see event handlers in the skeleton code)
 - Implement `draw()` and add relevant functions in your polygon classes to handle transformations
- Hints
 - You may need to call `glLoadIdentity()` followed by `glMultMatrixf()` to place your polygons properly – remember OpenGL is a “state” language
 - Or you can call `glLoadMatrixf()` to load a matrix from an array directly

Task 5

- Implement visualization of your polygons
- Rendering
 - Implement `display()` of Viewer and `draw()` of your polygon classes
 - You may need to maintain a list of polygon instances internally as they are added on the fly during runtime

Be Aware

- Please DO (you will otherwise lose your credits):
 - The default coordinates of the primitive polygons (i.e., contents of `glVertex`) never change during runtime as the purpose of this assignment is to learn transformations (if modified, no credits)
 - Instead, your polygon's coordinates should be changed via `glMultMatrixf()` or `glLoadMatrixf()`
 - Try to reduce calling `glMultMatrixf()` / `glLoadMatrixf()` (i.e., make your transformations composite)
 - Provide sufficient comments on what you implement; also, put Task # in your code
 - Provide user instructions to run your program
 - Do not make multiple files for your implementation – a single source file will be enough
 - Make window size 800-by-800

Submission

- What to submit
 - Your implementation and user manual
- Where to submit
 - Blackboard
- When to submit by
 - September 30th by midnight, 10 days from now (no extension – start as soon as possible)

Questions?

- Use Blackboard (Discussions → Course Board)
- Email/call the instructor
- Useful resources:
 - <https://www.opengl.org/>
 - <http://www.songho.ca/opengl/index.html>
 - <http://pyopengl.sourceforge.net/context/tutorials/index.html>