Model View Projection

Release 0.0.1

William Emerison Six

Jan 24, 2025

CONTENTS

1 Introduction 1

2 Opening a Window - Demo 01 5

3 Draw A Rectangle - Demo 02 19

4 Window Resizing and Proportionality - Demo 03 31

5 Moving the Paddles - Keyboard Input - Demo 04 39

6 Add Translate Method to Vertex - Demo 05 45

7 Modelspace - Demo 06 51

8 Rotations - Demo 07 67

9 Rotation Fix Attempt 1 - Demo 08 85

10 Rotation Fixed - Sequence of Transformations - Demo 09 91

11 Camera Space - Demo 10 97

12 Relative Objects - Demo 11 105

13 Rotate the Square - Demo 12 111

14 Rotate the Square Around Paddle 1 - Demo 13 115

15 Adding Depth - Z axis Demo 14 119

16 Adding Depth - Enable Depth Buffer - Demo 15 129

17 Moving Camera in 3D Space - Demo 16 133

18 3D Perspective - Demo 17 141

19 Lambda Stack - Demo 18 151

20 Matrix Stacks - Demo 19 163

21 Shaders - Demo 20 169

22 OpenGL3.3 Core Profile - Demo 21 175

23 Standard Perspective Matrix 191

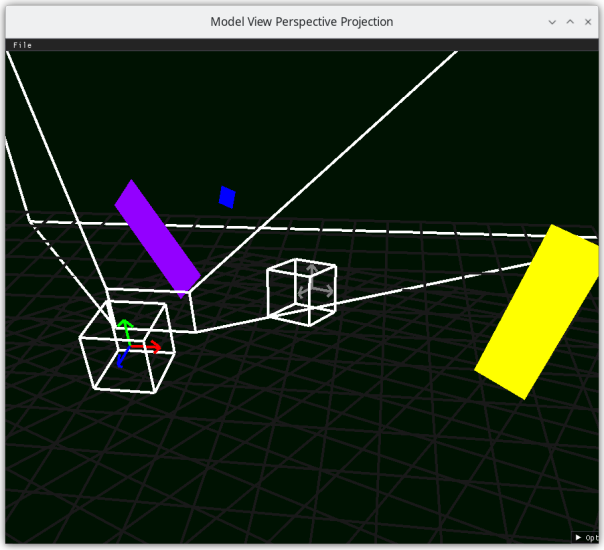
24 Indices and tables 201

CHAPTER

ONE

INTRODUCTION

Learn how to program in 3D computer graphics in Python!



1.1 Source code

This book references source code, which is at https://github.com/billsix/modelviewprojection

1.2 Approach

This book takes a "mistake-driven development" approach: instead of handing you polished math formulae, it walks you through building complex graphics applications step by step. Along the way, we'll make mistakes—and then fix them—so you can see how solutions emerge in real-world programming.

You'll learn how to place objects in space, draw them relative to others, add a camera that moves over time based on user input, and transform all those objects into the 2D pixel coordinates of your screen. By the end, you'll understand the foundations of creating first-person and third-person applications or games. The goal? To empower you to build the graphics programs you want, using math you mostly already know.

This book keeps things intentionally simple. The applications we create won't be particularly pretty or realistic-looking. For more advanced topics, you'll want to dive into references like the OpenGL "Red Book " and "Blue Book," or explore some of the tutorials listed at the end.

1. LearnOpenGL
2. OpenGLTutorial

While this book fills a huge gap by focusing on the basics in a hands -on way, those other books are fantastic references for diving into advanced topics.

1.3 Pre-requisities

1. Basic programming concepts in Python.
    1. YouTube videos
        1. Learn Python with Socratica
        2. Microsoft Python Tutorials
    2. Books
        1. https://learnbyexample.github.io/100_page_python_intro/preface.html
        2. https://diveintopython3.problemsolving.io/
2. High school trigonometry
3. Linear Algebra (optional)
    1. 3Blue1Brown - Linear Transformations
    2. 3Blue1Brown - Matrix Multiplication as Composition

1.4 Required Software

You will need to install Python. https://realpython.com/installing-python/

Before running this code, you need a virtual environment, with dependencies installed. https://docs.python.org/3/tutorial/venv.html

1.4.1 Windows

On Windows, if you use the Developer command prompt, to set up the environment run

```
python -m venv venv
cd venv\Scripts
activate.bat
cd ..\..\
python -m pip install --upgrade pip setuptools
python -m pip install -r requirements.txt
```

To run the Spyder IDE to execute the code in the book, open Spyder on the developer command prompt

```
cd venv\Scripts
activate.bat
cd ..\..\
spyder
```

1.4.2 MacOS or Linux

On MacOS or Linux, on a terminal, to set up the environment, run

```
python3 -m venv venv
source venv/bin/activate
python3 -m pip install --upgrade pip setuptools
python3 -m pip install -r requirements.txt
```

To run the Spyder IDE to execute the code in the book, open Spyder on the developer command prompt

```
source venv/bin/activate
spyder
```

1.4.3 Linux

Install Python3, glfw via a package manager. Use pip and virtualenv to install dependencies

1.4.4 Mac

Python Python3 (via anaconda, homebrew, macports, whatever), and use pip and virtualenv to install dependencies.

CHAPTER

TWO

OPENING A WINDOW - DEMO 01

2.1 Purpose

Learn how to open a window, make a black screen, and close the window.

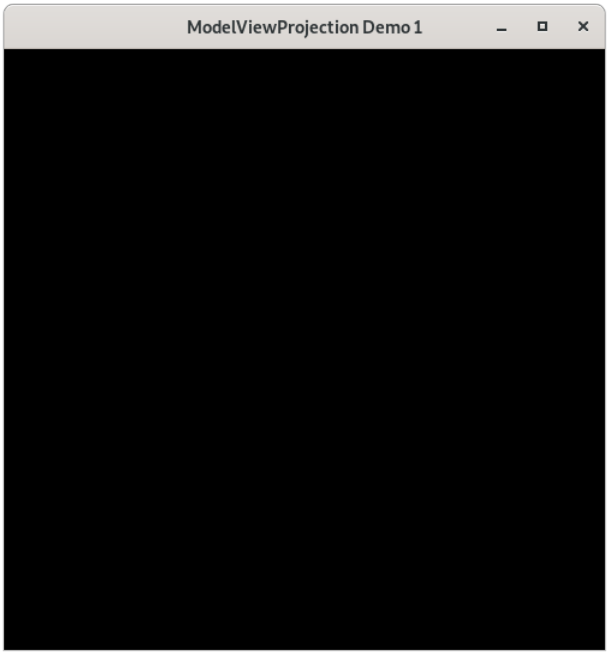


Fig. 1: Demo 01

2.2 How to Execute

Load src/demo01/demo.py in Spyder and hit the play button

2.3 Terminology

The device connected to a computer that displays information to the user is called a *monitor*.

Fig. 2: Computer Monitor

A monitor consists of a two-dimensional grid of tiny light-emitting elements, called *pixel*s. Each pixel typically has three components: red, green, and blue.

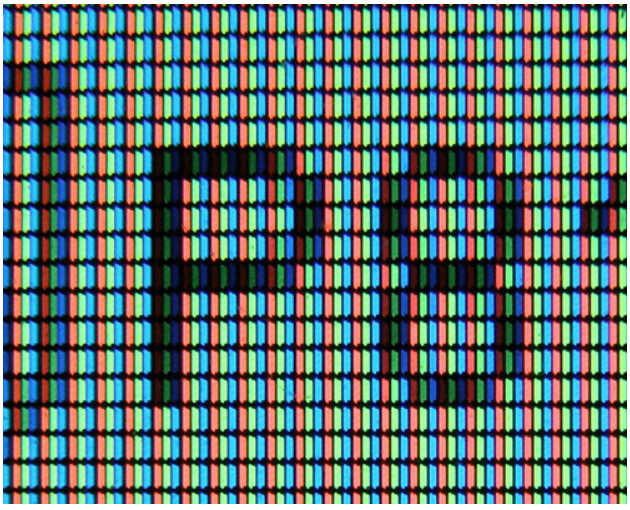


Fig. 3: Pixels

At any given moment, the computer instructs each pixel to display a specific color, which is represented as a number.

The combination of all these pixel colors at a single point in time is called a frame, and this frame forms a picture that conveys meaning to the user. In OpenGL, pixel coordinates start at (0, 0) in the bottom -left corner of the window. The top-right corner is at (window_width, window_height).

Frames are generated by the computer and sent to the monitor at a constant rate, called the *frame rate*, measured in *Hertz* (Hz). By updating these frames rapidly and consistently, the computer creates the illusion of motion for the user.

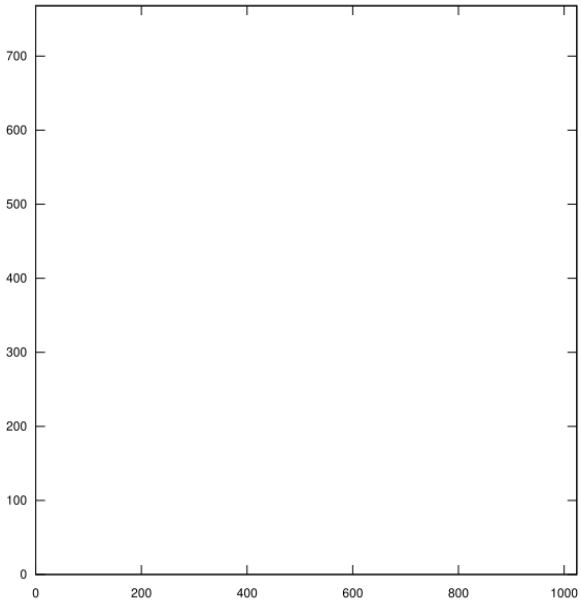


Fig. 4: 1024x768 monitor

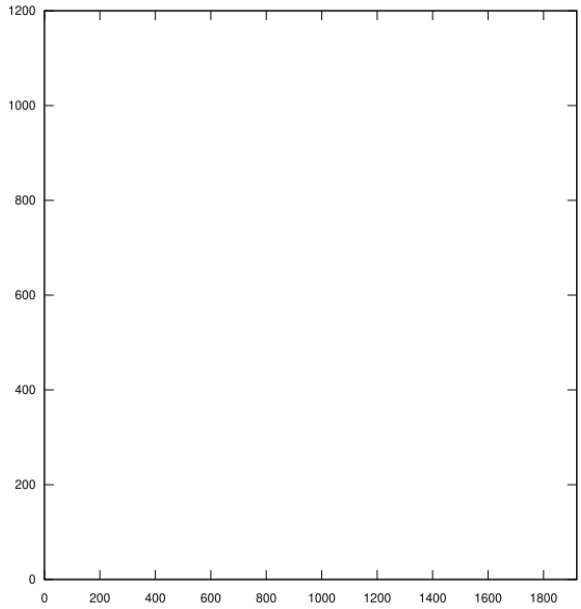
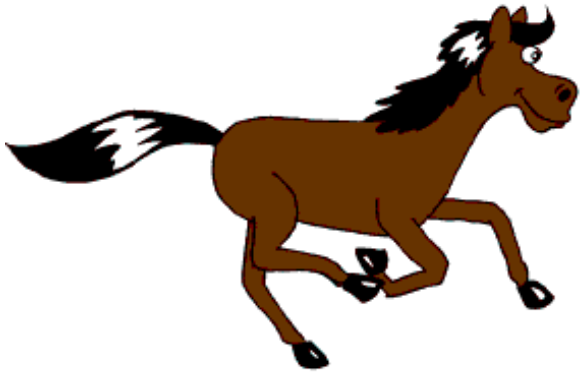
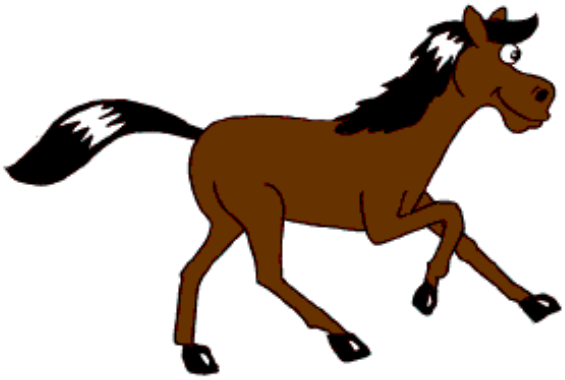
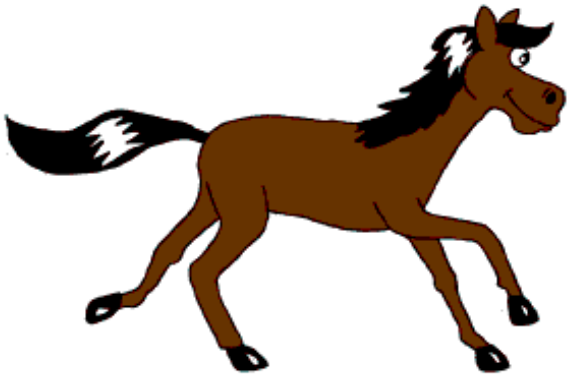


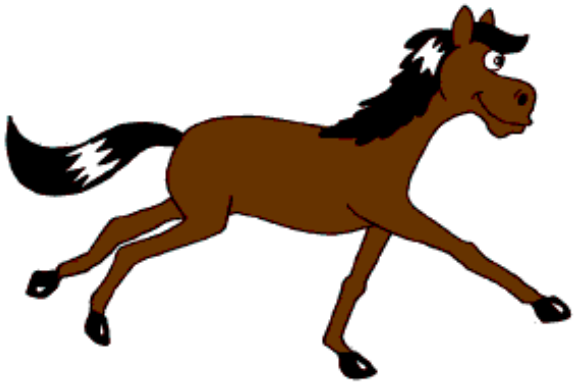
Fig. 5: 1920x1200 monitor

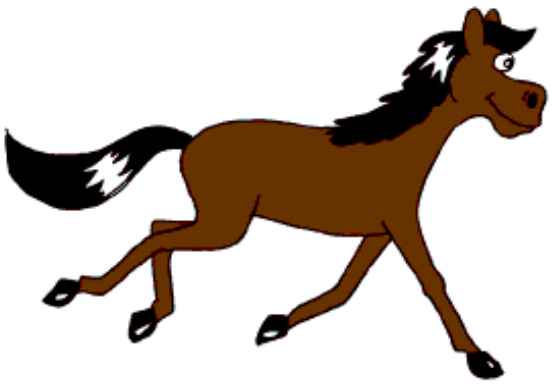
Fig. 6: 14 Hertz Motion











2.4 Code

That's enough terms for now, let's get on to a working program!

2.4.1 Importing Libraries

Import Python modules, which are Python programmer's main form of libraries.

Listing 1: src/demo01/demo.py

```
22 import sys
23
24 # doc-region-begin import glfw
25 import glfw
26
27 # doc-region-begin import individual functions without needing module name
28 from OpenGL.GL import (
29     GL_COLOR_BUFFER_BIT,
30     GL_DEPTH_BUFFER_BIT,
31     GL_MODELVIEW,
32     GL_PROJECTION,
33     glClear,
34     glClearColor,
35     glLoadIdentity,
36     glMatrixMode,
37     glViewport,
38 )
39
```

- The "sys" module is imported. To call functions from this module, the syntax is sys.function

Listing 2: src/demo01/demo.py

```
28 from OpenGL.GL import (
29     GL_COLOR_BUFFER_BIT,
30     GL_DEPTH_BUFFER_BIT,
31     GL_MODELVIEW,
32     GL_PROJECTION,
33     glClear,
34     glClearColor,
35     glLoadIdentity,
36     glMatrixMode,
37     glViewport,
38 )
39
40 # doc-region-end import first module
41
42 # doc-region-end import glfw
43
```

- GL is a submodule of the OpenGL module. By directly importing specific functions from GL into our current module, we avoid having to write OpenGL.GL.function every time. This keeps our code cleaner and easier to read, especially since we'll be using these functions frequently.

Listing 3: src/demo01/demo.py

```
import glfw

# doc-region-begin import individual functions without needing module name
from OpenGL.GL import (
    GL_COLOR_BUFFER_BIT,
    GL_DEPTH_BUFFER_BIT,
    GL_MODELVIEW,
    GL_PROJECTION,
    glClear,
    glClearColor,
    glLoadIdentity,
    glMatrixMode,
    glViewport,
)

# doc-region-end import first module

```

- GLFW is a library that lets us create windows and handle input from both the keyboard and mouse. It works seamlessly across Linux, Windows, and macOS, making it a versatile tool for cross-platform development.

In a Python prompt, you can use tab completion to see which functions are available in a module. You can also type help(modulename) to get detailed information about the module (press q to exit the help pager). The help() function works on any object in Python, including modules, classes, and functions, making it a handy tool for exploring and learning.

2.4.2 Opening A Window

Desktop operating systems allow users to run multiple programs simultaneously. Each program displays its output within a dedicated area of the monitor, called a window.

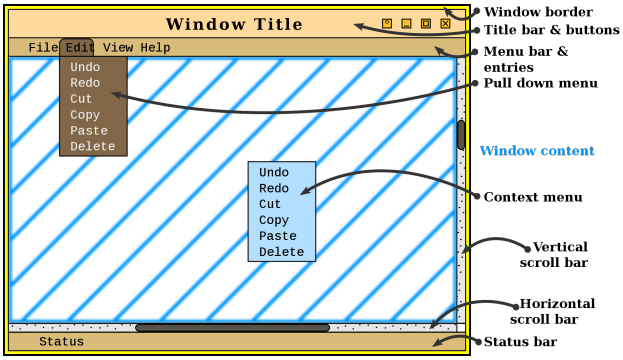


Fig. 7: Window

To create and open a window in a cross-platform way, this book uses functions provided by the widely supported GLFW library, which works on Windows, macOS, and Linux. In addition to window management, GLFW also provides functions for handling input from keyboards and game controllers.

2.4.3 GLFW/OpenGL Initialization

Initialize GLFW.

Listing 4: src/demo01/demo.py

```
48 if not glfw.init():
49     sys.exit()
```

Initialize GLFW. If initialization fails, the program should terminate. What does the initialization do? Doesn't matter. Think of it like a constructor for a class; it initializes some state that it needs for later function calls.

Set the version of OpenGL

OpenGL has been around a long time, and has multiple, possibly incompatible versions. For this demo, we use OpenGL 1.4. By the end of this book, we will be using OpenGL 3.3.

Listing 5: src/demo01/demo.py

```
53 glfw.window_hint(glfw.CONTEXT_VERSION_MAJOR, 1)
54 glfw.window_hint(glfw.CONTEXT_VERSION_MINOR, 4)
```

Create a Window

Listing 6: src/demo01/demo.py

```
58 window = glfw.create_window(500, 500, "ModelViewProjection Demo 1", None, None)
```

Listing 7: src/demo01/demo.py

```
62 if not window:
63     glfw.terminate()
64     sys.exit()
```

- If GLFW cannot open the window, quit. Unlike MC Hammer , we are quite legit, yet still able to quit.

Listing 8: src/demo01/demo.py

```
68 glfw.make_context_current(window)
```

- Make the window's context current. The details of this do not matter for this book.

Listing 9: src/demo01/demo.py

```
73 def on_key(win, key, scancode, action, mods):
74     if key == glfw.KEY_ESCAPE and action == glfw.PRESS:
75         glfw.set_window_should_close(win, 1)
76
77
```

- Define an register a key handler.

If the user presses the escape key while the program is running, inform GLFW that the user wants to quit. We will handle this later in the event loop.

Functions are first class values in Python, and are objects just like anything else. The can be passed as arguments, stored in variables, and applied later zero, 1, or more times.

```
>>> def doubler(x):
...     return x * 2
...
>>> def add_five_to_result_of(f, x):
...     return 5 + f(x)
...
>>> add_five_to_result_of(doubler, 3)
11
```

Listing 10: src/demo01/demo.py

```
83 glClearColor(0.0289, 0.071875, 0.0972, 1.0)
```

- Before a frame is drawn, it is first turned into a blank slate, where the color of each pixel is set to some value representing a color. We are not clearing the frame-buffer right now, but setting what color will be used for a later clear. Calling "glClearColor" "0,0,0,1", means black "0,0,0", without transparency (the "1").

Listing 11: src/demo01/demo.py

```
87 glMatrixMode(GL_PROJECTION)
88 glLoadIdentity()
89 glMatrixMode(GL_MODELVIEW)
90 glLoadIdentity()
```

- Don't worry about the 4 lines here. Although they are necessary, we will cover them in depth later. After all, this book is called ModelViewProjection. :-)

2.4.4 The Event Loop

When you pause a movie, motion stops and you see one picture. Movies are composed of sequence of pictures, when rendered in quick succession, provide the illusion of motion. Interactive computer graphics are rendered the same way, one "frame" at a time.

Render a frame, flush the complete frame-buffer to the monitor. Repeat indefinitely until the user closes the window, or the program needs to terminate.

Listing 12: src/demo01/demo.py

```
95  while not glfw.window_should_close(window):
96      glfw.poll_events()
97
98      width, height = glfw.get_framebuffer_size(window)
99      glViewport(0, 0, width, height)
100     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
101     glfw.swap_buffers(window)
```

- Poll the operating system for any events, such as mouse movements, keyboard input, etc. This does not handle them, just registers them as having happened.

- Get the size of the frame-buffer. The *frame-buffer* contains data representing all the pixels in a complete video frame. Python allows the returning of multiple values in the form of a tuple. Assigning to the variables this way is a form of "destructuring"
- Tell OpenGL that we wish to draw in the entire frame-buffer, from the bottom left corner to the upper right corner.
- Make the frame-buffer a blank slate by setting all of the pixels to have the same color. The color of each pixel will be the clear color. If we hadn't cleared the frame-buffer, then frame number n+1 would be drawing on top of whatever was drawn on frame number n. Programming in OpenGL is a bit different than normal programming in a normal language, in that individual function calls do not complete self-contained tasks, as subroutines typically do. Instead, the procedure calls to OpenGL functions only make sense based off of the context in which they are evaluated, and the sequence of OpenGL calls to complete a task.
- We have colored every pixel to be black, so flush the frame-buffer to the monitor, and swap the back and front buffers.

2.5 Exercise

- Run the program, close it by hitting Escape.
- Before the call to glClear, enter two new lines. On the first, type "import pdb". On the second type "pdb.set_trace()". Now run the program again and observe what is different. (pdb.set_trace() sets a breakpoint, meaning that the program pauses execution, although the GLFW window is still on screen over time)

One frame is created incrementally at a time on the CPU, but the frame is sent to the monitor only when frame is completely drawn, and each pixel has a color. The act of sending the frame to the monitor is called *flushing* the frame.

OpenGL has two *frame-buffers* (regions of memory which will eventually contain the full data for a frame), only one of which is "active", or writable, at a given time. "glfwSwapBuffers" initiates the flushing the current buffer, and which switches the current writable frame-buffer to the other one.

2.6 Black Screen

Type "python src/demo01/demo.py", or "python3 src/demo01/demo.py" to run.

The first demo is the least interesting graphical program possible.

1. Sets the color at every pixel black. (A constant color is better than whatever color happened to be the previous time it was drawn.)
2. If the user resized the window, reset OpenGL's mappings from *normalized-device-coordinates* to *screen-coordinates*.
3. Cleared the color buffer and the depth buffer (don't worry about this for now).

When this code returns, the event loop flushes (i.e) sends the frame to the monitor. Since no geometry was drawn, the color value for each pixel is still black.

Each color is represented by a number, so the frame is something like this, where 'b' represents black

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

(continues on next page)

(continued from previous page)

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

The event loop then calls this code over and over again, and since we retain no state and we draw nothing, a black screen will be displayed every frame until the user closes the window, and says to himself, "why did I buy Doom 3"?

2.7 Questions

- In the following ASCII-art diagram of the frame-buffer, where 'b' represents black, 'Y' represents yellow, and "G" represents green, what is width and height of the frame-buffer?

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

- In the above ASCII-art diagram of the frame-buffer, where 'b' represents black, 'Y' represents yellow, and "G" represents green, what is the color at pixel (1,1)?
- In the below ASCII-art diagram of the frame-buffer, where 'b' represents black, 'Y' represents yellow, and "G" represents green, what is the color at pixel (2,3)? At (36,2)?

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
GGGGGbbbbbbbbbbbbbbbbbbbbbbbbbYYYYYYY
GGGGGbbbbbbbbbbbbbbbbbbbbbbbbbYYYYYYY
GGGGGbbbbbbbbbbbbbbbbbbbbbbbbbYYYYYYY
GGGGGbbbbbbbbbbbbbbbbbbbbbbbbbYYYYYYY
GGGGGbbbbbbbbbbbbbbbbbbbbbbbbbYYYYYYY
GGGGGbbbbbbbbbbbbbbbbbbbbbbbbbYYYYYYY
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbYYYYYYY
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

-

2.8 Answers

- width - 37, column 0-36. Height - 12, rows 0 - 11
- ‘b’, for the color black
- ‘G’ for Green, ‘Y’ for Yellow

CHAPTER

THREE

DRAW A RECTANGLE - DEMO 02

3.1 Purpose

Learn how to plot a rectangle. Learn about OpenGL's coordinate system, normalized-device coordinates, which are from -1.0 to 1.0, in the X, Y, and Z directions. Anything drawn entirely outside of this region will not be displayed on the screen.

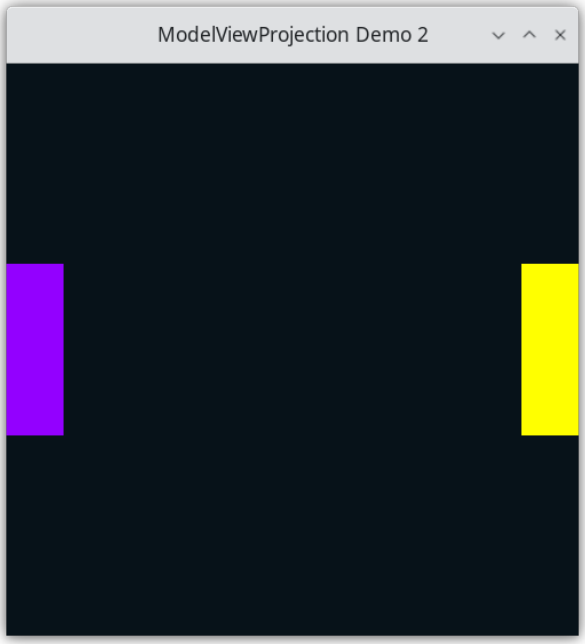


Fig. 1: Demo 02

3.2 How to Execute

Load src/demo02/demo.py in Spyder and hit the play button

3.3 Code

3.4 GLFW/OpenGL Initialization

The setup code is the same. Initialize GLFW. Set the OpenGL version. Create the window. Set a key handler for closing. Set the background to be black. Execute the event/drawing loop.

3.5 The Event Loop

Within the event loop, demo02/demo.py draws 2 rectangles, as one might see in a game of Pong.

Listing 1: src/demo02/demo.py

```
70 while not glfw.window_should_close(window):
```

```
...
```

3.5.1 Draw Paddles

A black screen is not particularly interesting, so let's draw something, say, two rectangles.

We need to figure out what colors to use, and the positions of the rectangles, defined below.

To set the color, we will use "glColor3f". "glColor3f" sets a global variable, which makes it the color to be used for the subsequently-drawn graphical shape(s). Given that the background will be black, lets make the first paddle purple, and a second paddle yellow.

To specify the corners of the rectangle, "glBegin(GL_QUADS)" tells OpenGL that we will soon specify 4 *vertices*, (i.e. points) which define the quadrilateral. The vertices will be specified by calling "glVertex2f" 4 times.

"glEnd()" tells OpenGL that we have finished providing vertices for the begun quadrilateral.

3.6 Draw Paddle 1

Listing 2: src/demo02/demo.py

```
79     glColor3f(0.578123, 0.0, 1.0)
80     glBegin(GL_QUADS)
81     glVertex2f(-1.0, -0.3)
82     glVertex2f(-0.8, -0.3)
83     glVertex2f(-0.8, 0.3)
84     glVertex2f(-1.0, 0.3)
85     glEnd()
```

The paddle looks like this relative to normalized device coordinates (NDC):

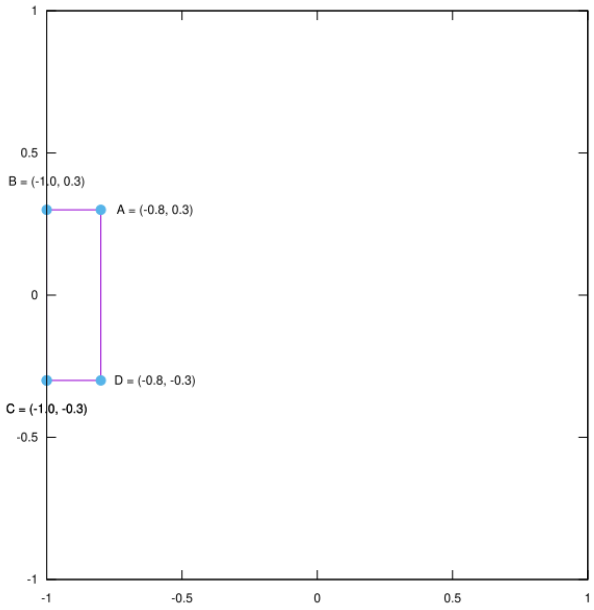


Fig. 2: Rectangle

3.7 Draw Paddle 2

Listing 3: src/demo02/demo.py

```
89      glColor3f(1.0, 1.0, 0.0)
90      glBegin(GL_QUADS)
91      glVertex2f(0.8, -0.3)
92      glVertex2f(1.0, -0.3)
93      glVertex2f(1.0, 0.3)
94      glVertex2f(0.8, 0.3)
95      glEnd()
```

The 2 paddles looks like this relative to normalized device coordinates (NDC):

Listing 4: src/demo02/demo.py

```
99      glfw.swap_buffers(window)
```

- done with frame, flush the current buffer to the monitor
- Swap front and back buffers

The frame sent to the monitor is a set of values like this

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
PPPPPbbbbbbbbbbbbbbbbbbbbbbbbbbbbbRRRRR
```

(continues on next page)

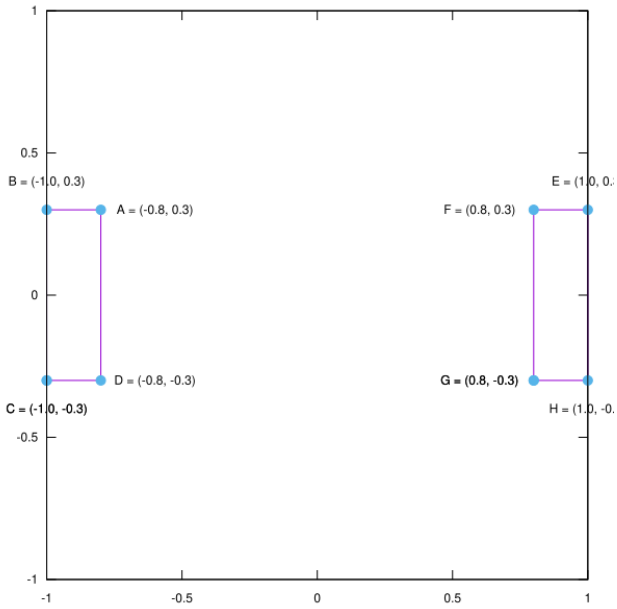


Fig. 3: Rectangle

(continued from previous page)

```
PPPPPbbbbbbbbbbbbbbbbbbbbbbbbbbbbbRRRRR
PPPPPbbbbbbbbbbbbbbbbbbbbbbbbbbbbbRRRRR
PPPPPbbbbbbbbbbbbbbbbbbbbbbbbbbbbbRRRRR
PPPPPbbbbbbbbbbbbbbbbbbbbbbbbbbbbbRRRRR
PPPPPbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

What do we have to do to convert from normalized-device-coordinates (i.e. (1.0, 0.8)) into pixel coordinates (i.e. pixel (10,15))? Nothing, OpenGL does that for us automatically; therefore we never have to think in terms of pixels coordinates, only in terms of vertices of shapes, specified by normalized-device-coordinates. OpenGL also automatically colors all of the pixels which are inside of the quadrilateral.

Why do we use normalized-device coordinates instead of pixel coordinates?

3.8 Normalized-Device-Coordinates

The author owns two monitors, one which has 1024x768 pixels, and one which has 1920x1200 pixels. When he purchases a game from Steam, he expects that his game will run correctly on either monitor, in full-screen mode. If a graphics programmer had to explicitly set the color of individual pixels using the pixel's coordinates, the the programmer would have to program using "screen-space" (Any space means a system of numbers which you're using. Screen-space means you're specifically using pixel coordinates, i.e, set pixel (5,10) to be yellow).

What looks alright in screen-space on a large monitor…

Isn't the same picture on a smaller monitor.

Like any good program or library, OpenGL creates an abstraction. In this case, it abstracts over screen-space, thus freeing the programmer from caring about screen size. If a programmer does not want to program in discrete (dis-

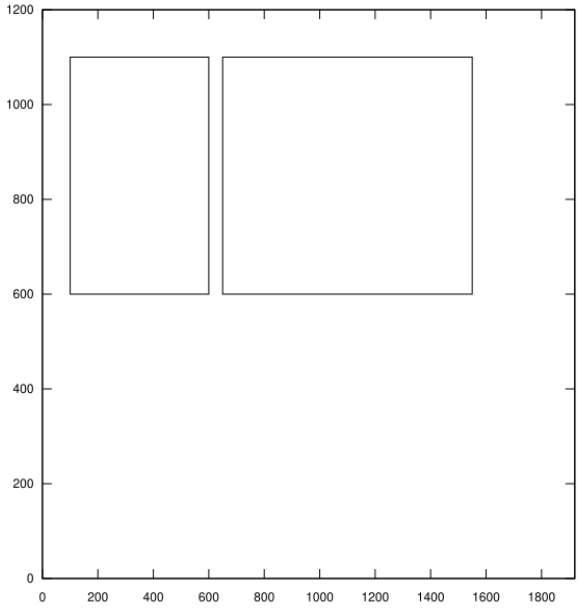


Fig. 4: Screenspace

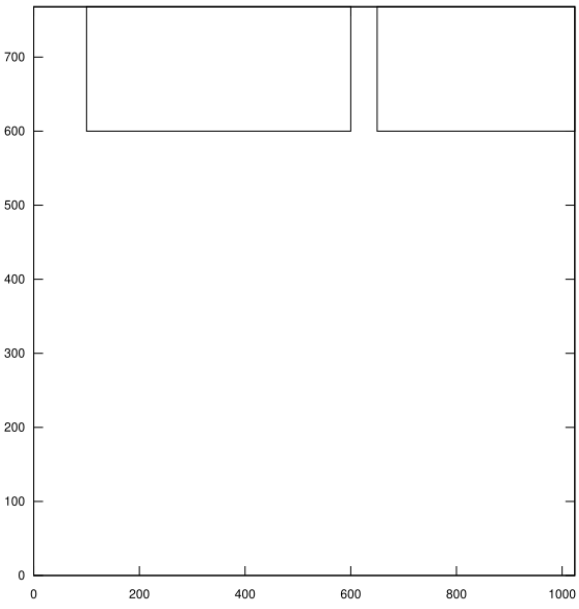


Fig. 5: Screenspace

crete means integer values, not continuous) screen-space, what type of numbers should he use? Firstly, it should be a continuous space, meaning that it should be in decimal numbers. Because if a real-world object is 10.3 meters long, a programmer should be able to enter "float foo = 10.3". Secondly, it should be a fixed range vertically and an fixed range horizontally. OpenGL will have to convert points from some space to screen-space, and since this is done in hardware (i.e. you can't pragmatically change how the conversion happens), it should be a fixed size.

OpenGL uses what's called *normalized-device-coordinates*, which is a continuous space from -1.0 to 1.0 horizontally, -1.0 to 1.0 vertically, and from -1.0 to 1.0 depth-ally. (Is there an actual word for that???)

The programmer specifies geometry using normalized-device-coordinates, and OpenGL will convert from a continuous, -1.0 to 1.0 space, to discrete pixel-space, and the programmer cannot change this.

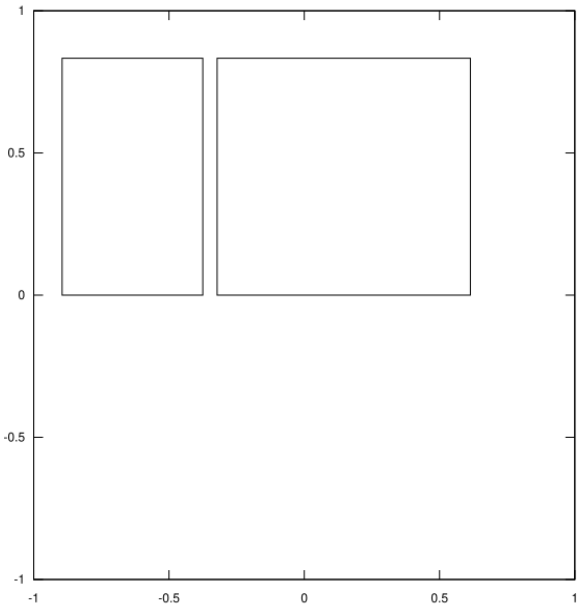


Fig. 6: NDC space

Whether we own a small monitor

Or a large monitor.

3.9 Graph of Spaces

The following is a graph, specifically a Cayley Graph, of the two spaces shown so far. The nodes represent a coordinate system (i.e. origin and axes), and the directed edge represents a function that converts coordinates from one coordinate system to another. If this isn't clear to the reader, look above at the 3 pictures of the 2 quadrilaterals. They look the same, but if you label each vertex on all three graphs, and look at the axes to find their plotted values, you will find that they differ. Changing between coordinate systems means to take vertices from one coordinate system and changing them to another, like converting from Celsius to Fahrenheit, meters to feet, etc.

The function that converts from NDC to Screen space is arbitrarily named f, sub-scripted by "NDC", super-scripted by "S". This is a common notation when creating functions that change basis, i.e. coordinate conversion. A definition of this function is not provided right now, because we can't change it in software. But we must recognize that it exists.

The Wikipedia article for Cayley graphs (https://en.wikipedia.org/wiki/Cayley_graph) is intimidating, but for our purposes, its use is very simple.

An uni-variate example use of Cayley graphs is exchanging coins with a bank.

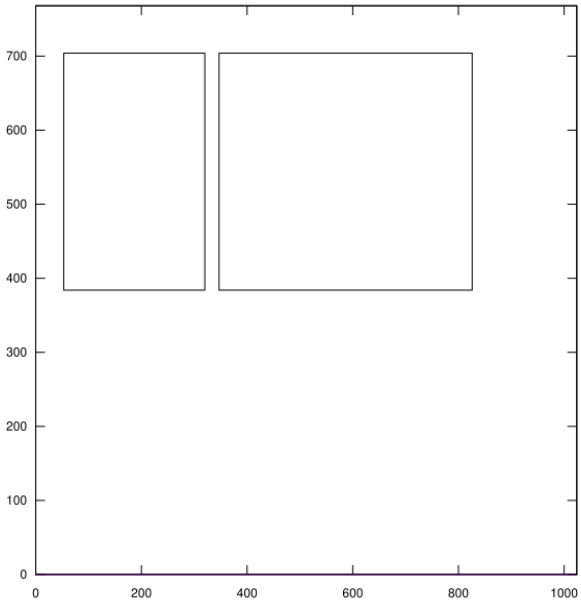


Fig. 7: NDC space

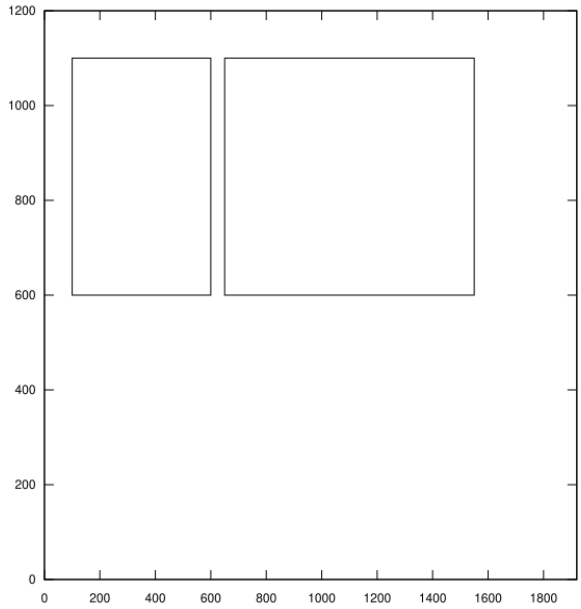


Fig. 8: NDC space

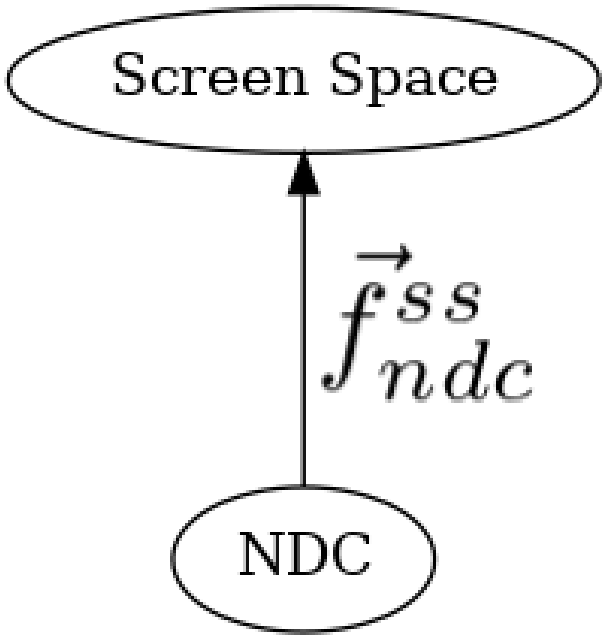


Fig. 9: Demo 02

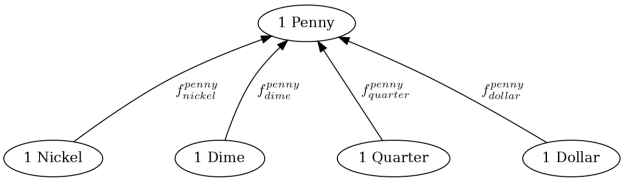


Fig. 10: Cayley graph of uni variate coordinate conversion for money.

$$f_{nickel}^{penny}(x) = 5 * x$$

$$f_{dime}^{penny}(x) = 10 * x$$

$$f_{quarter}^{penny}(x) = 25 * x$$

$$f_{dollar}^{penny}(x) = 100 * x$$

The directed edges in the Cayley graph show the direction that the function, i.e. transformation, applies.

To convert 20 nickels into pennies, start at nickel, move to penny, while applying the appropriate function along the way.

$$f_{nickel}^{penny}(x) = 5 * x$$

$$\begin{aligned} f_{nickel}^{penny}(20) &= 5 * 20 \\ &= 100 \end{aligned}$$

To convert those 100 penny into quarters, move from penny to quarter, but since we are moving in the opposite direction of the edge, we must apply the multiplicative inverse of that function. These functions are invertible by taking the reciprocal of the coefficient

$$f_{quarter}^{penny}(x) = 25 * x$$

$$\begin{aligned} f_{penny}^{quarter}(x) &= {f_{quarter}^{penny}}^{-1}(x) \\ &= 1/25 * x \end{aligned}$$

$$\begin{aligned} f_{penny}^{quarter}(100) &= 1/25 * x \\ &= 4 \end{aligned}$$

By taking the inverse of the coefficient, we satisfy a definition of an inverse

$$(f \circ f^{-1})(x) = x$$

To convert between any denomination, say dimes to dollars, just compose the functions, remembering to take the inverse of any directed edge that you against.

$$\begin{aligned} f_{dime}^{dollar}(x) &= f_{penny}^{dollar}(f_{dime}^{penny}(x)) \\ &= {f_{dollar}^{penny}}^{-1}(f_{dime}^{penny}(x)) \end{aligned}$$

$$f_{dime}^{dollar} = {f_{dollar}^{penny}}^{-1} \circ f_{dime}^{penny}$$

Notice in the last equation that we defined the function via function composition, and didn't specify any arguments. We just focus on the types of units in and the units out, but the details of those functions are not relevant when traversing the Cayley graph.

A big part of being able to understand graphics well is being able to figure out what to ignore. So how would we convert coordinates from space D to space B (defined below)? What do those names mean? It doesn't matter. What is the definition of all of the functions? It doesn't matter. All that matters

- is that the function exists
- the definition of the function is provided to us
- the function is invertible
- and that we can call a procedure to invert a function.

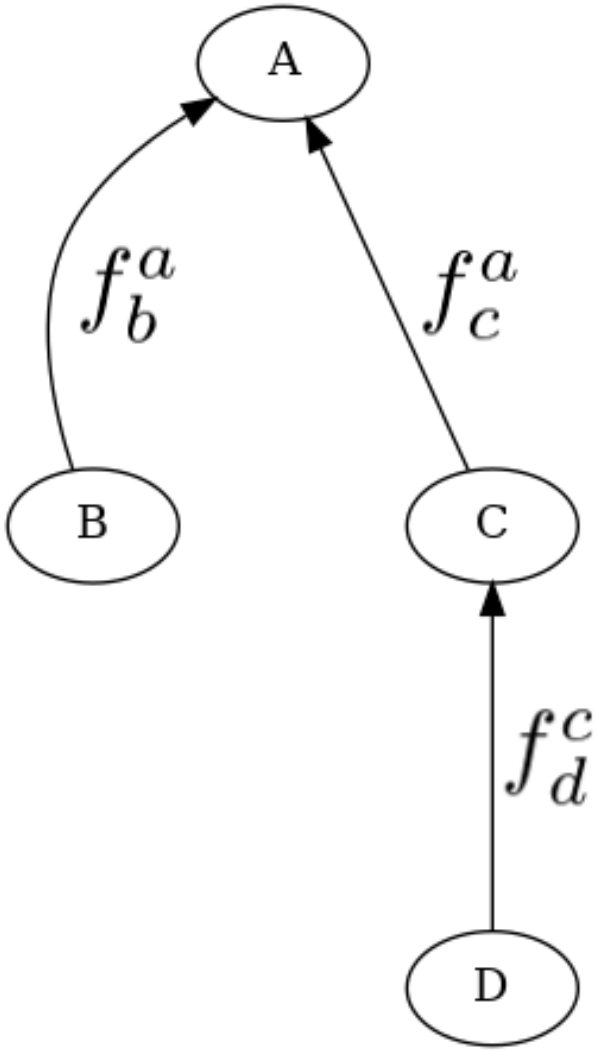


Fig. 11: Generic Cayley Graph

Well to convert data frame space D to space B, we know that D is defined relative to C, C is defined relative to A, and B is defined relative to A. Because of the arrows, we know that we are given a function from D to C, a function from C to A, and a function from B to A. We are not directly given their inverses, but we can calculate them easily enough, or have a computer do it for us.

In tracing out the graph, we are going with the first two directed edges, and against the last one. So we compose the functions and take the appropriate inverse(s).

$$\begin{aligned}\vec{f}_d^b(\vec{x}) &= \vec{f}_a^b(\vec{f}_c^a(\vec{f}_d^c(\vec{x}))) \\ &= \vec{f}_b^{a\,-1}(\vec{f}_c^a(\vec{f}_d^c(\vec{x})))\end{aligned}$$

A function from A to B is not provided to us, but we wrote it in the equation as an idea of a function that we wish to have, even though we don't currently have it. However, we can create this function by invoking inverse on the provide function from A to B.

Since we're dealing with function composition mainly, we don't even need to specify the argument

$$\vec{f}_d^b = \vec{f}_b^{a\,-1} \circ \vec{f}_c^a \circ \vec{f}_d^c$$

If this seems to abstract for now, don't worry. By the end of the course, it should be clear. The goal of this book is to make it clear, and then, obvious.

3.10 Exercise

- Run Demo 2. Resize the window using the GUI controls provided by the Operating system. first make it skinny, and then wide. Observe at what happens to the rectangles.

CHAPTER

FOUR

WINDOW RESIZING AND PROPORTIONALITY - DEMO 03

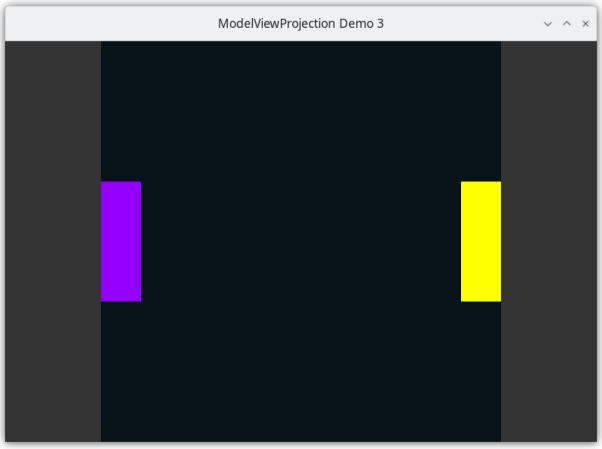


Fig. 1: Demo 03

4.1 Problem With Previous Demo

When running Demo02, if the user resizes the windows, then the paddles lose their proportionality, as NDC no longer is mapped to a square screen-space.

4.2 How to Execute

Load src/demo03/demo.py in Spyder and hit the play button

4.3 Purpose

Modify the previous demo, so that if the user resizes the window of the OpenGL program, that the picture does not become distorted.

Create procedure to ensure proportionality.

4.4 Keeping the Paddles Proportional

In the previous demo, if the user resized the window, the paddles appear distorted, as they were shrunk in one direction if the window became too thin or too fat.

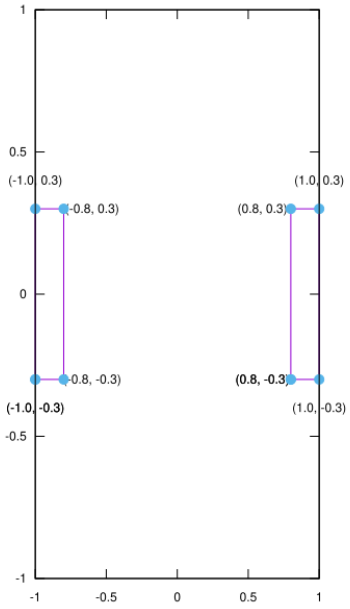


Fig. 2: Yuck

Assume that this is a problem for the application we are making, how could we solve it and keep proportionality regardless of the dimensions of the window? Ideally, we would like to draw our paddles with a black background within a square region in the center of the window, regardless of the dimensions of the window.

OpenGL has a solution for us. The *viewport* is a rectangular region contained within the window into which OpenGL will render. By specifying a viewport, OpenGL will convert the normalized-device-coordinates to the sub-window space of the viewport, instead of the whole window.

Because we will only draw in a subset of the window, and because all subsequent chapters will use this functionality, I have created a procedure for use in all chapters named “draw_in_square_viewport”.

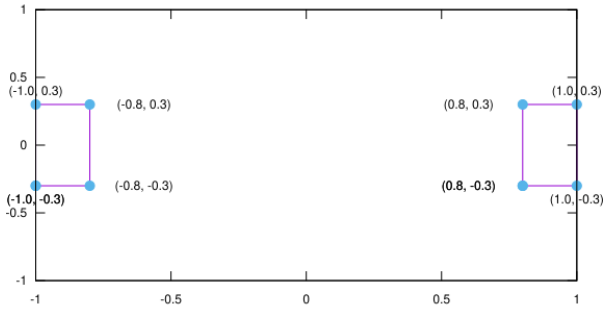


Fig. 3: Yuck

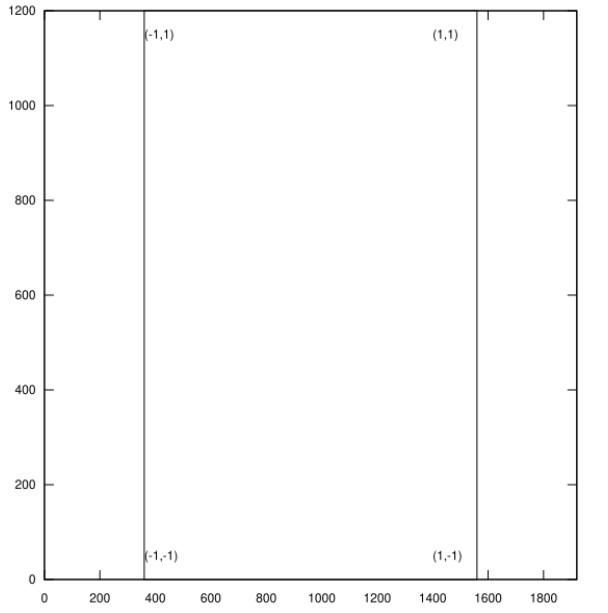


Fig. 4: Nice

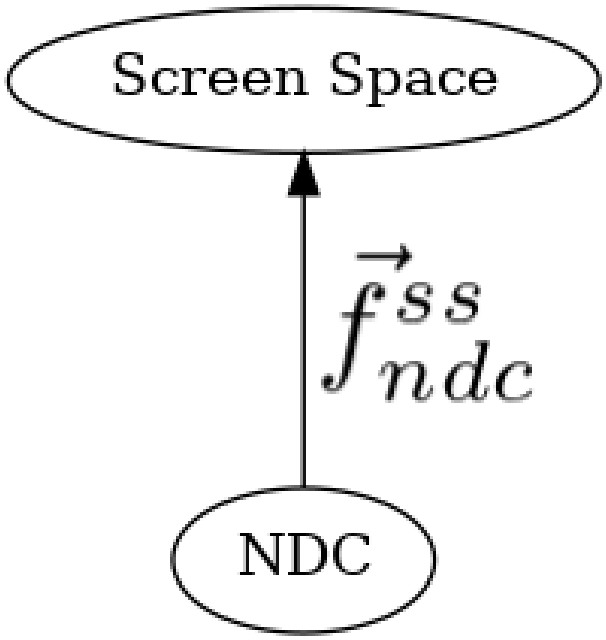


Fig. 5: Demo 03

4.5 Code

4.5.1 GLFW/OpenGL Initialization

The setup code is the same as the previous demo's setup. Initialize GLFW. Set the OpenGL version. Create the window. Set a key handler for closing. Execute the event/drawing loop. The only code showed in this book will be the relevant parts. Consult the python source for the full, working code.

Set to Draw in Square Subsection Of Window

Listing 1: src/demo03/demo.py

```
75 def draw_in_square_viewport() -> None:
```

- declare a function to configure OpenGL to draw only in a square subset of the monitor, i.e. the viewport

Listing 2: src/demo03/demo.py

```
79     glClearColor(0.2, 0.2, 0.2, 1.0)
80     glClear(GL_COLOR_BUFFER_BIT)
```

- set the clear color to be gray.
- glClear clear the color of every pixel in the whole frame buffer, independent of viewport. So now the entire frame-buffer is gray.

Listing 3: src/demo03/demo.py

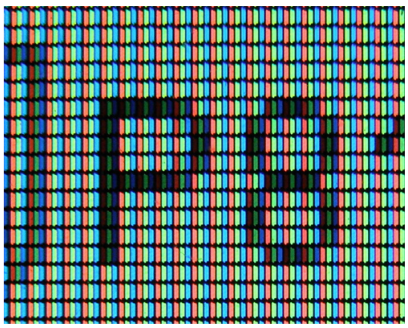
```
84      w, h = glfw.get_framebuffer_size(window)
85
86      square_size = w if w < h else h
```

- figure out the minimum dimension of the window. In the image above, the "square_size" is 1200, as the monitor's vertical screen-space is only 1200 pixels tall.
- To make a square sub-region, we need a number for the distance between vertices of the square. By using the minimum of the width and height, we can at least fill up the screen in one dimension.

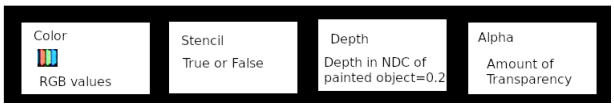
Listing 4: src/demo03/demo.py

```
90      glEnable(GL_SCISSOR_TEST)
91      glScissor(
92          int((w - square_size) / 2.0),  # bottom left x_screenspace
93          int((h - square_size) / 2.0),  # bottom left y_screenspace
94          square_size,  # x width, screenspace
95          square_size,  # y height, screenspace
96      )
```

- Enable the scissor test. Internally, OpenGL drivers likely have global variables that we set by calling functions. Every OpenGL feature isn't used by every OpenGL program. For instance, we are not using lighting to add realism. We aren't using texturing. We are using the scissor test, so we must enable it. We only enable the features that we need so that the OpenGL driver doesn't waste time doing unnecessary computations.
- the scissor test allows us to specify a region of the frame-buffer into which the OpenGL operations will apply. In this case, the color in every pixel in the frame-buffer is currently gray because of the existing class to glClearColor. By calling glScissor, we are setting a value in each fragment (i.e., pixel) on a square region of pixels to be true (and false everywhere else) which means "only do the OpenGL call on these fragments, ignore the others". As we will learn later, OpenGL stores much more information per fragment (i.e. pixel) than just its current color.



Fragment



- Look at the image above of NDC superimposed on Screen Space. From this, the arguments sent to glScissor should be clear.

Listing 5: src/demo03/demo.py

```
100     glClearColor(0.0289, 0.071875, 0.0972, 1.0)
101     glClear(GL_COLOR_BUFFER_BIT)
```

- glClear will only update the square to black values.

Listing 6: src/demo03/demo.py

```
104     glDisable(GL_SCISSOR_TEST)
```

- disable the scissor test, so now any OpenGL calls will happen as usual.

So we've drawn black into a square, and disabled the scissor test, so any subsequent OpenGL calls will still be drawn into the full frame-buffer. But, we only want to draw within the black square, and the scissor test does not modify the NDC to screen-space transformations. To modify the NDC to screen-space transformations, we set the viewport again, so that the NDC coordinates will be mapped the the region of screen coordinates that we care about, which is the black square.

Listing 7: src/demo03/demo.py

```
107     glViewport(
108         int(0.0 + (w - square_size) / 2.0),
109         int(0.0 + (h - square_size) / 2.0),
110         square_size,
111         square_size,
112     )
```

The Event Loop

This demo's event loop is just like the previous demo, but this time we call the procedure to ensure that we only draw in a square subset of the window.

Listing 8: src/demo03/demo.py

```
117 while not glfw.window_should_close(window):
118     glfw.poll_events()
119 
120     width, height = glfw.get_framebuffer_size(window)
121     glViewport(0, 0, width, height)
122     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

Listing 9: src/demo03/demo.py

```
126     draw_in_square_viewport()
```

- The event loop is the same as the previous demo, except that we call draw_in_square_viewport every frame at the beginning.

Listing 10: src/demo03/demo.py

```
129     glColor3f(0.578123, 0.0, 1.0)
130     glBegin(GL_QUADS)
```

(continues on next page)

(continued from previous page)

```
    glVertex2f(-1.0, -0.3)
    glVertex2f(-0.8, -0.3)
    glVertex2f(-0.8, 0.3)
    glVertex2f(-1.0, 0.3)
    glEnd()

    glColor3f(1.0, 1.0, 0.0)
    glBegin(GL_QUADS)

    glVertex2f(0.8, -0.3)
    glVertex2f(1.0, -0.3)
    glVertex2f(1.0, 0.3)
    glVertex2f(0.8, 0.3)
    glEnd()

    glfw.swap_buffers(window)
```

CHAPTER

FIVE

MOVING THE PADDLES - KEYBOARD INPUT - DEMO 04

5.1 Purpose

Add movement to the paddles using keyboard input.

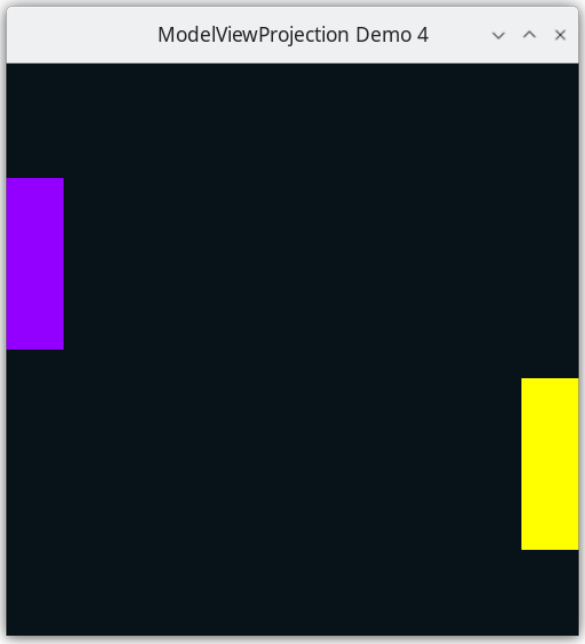


Fig. 1: Demo 04

5.2 How to Execute

Load src/demo04/demo.py in Spyder and hit the play button

5.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up

Paddles which don't move are quite boring. Let's make them move up or down by getting keyboard input.

And while we are at it, let's go ahead and create data structures for a Vertex, and for the collection of vertices that make up a Paddle.

5.4 Code

5.4.1 Data Structures

Here we use dataclasses, which automatically creates on the class a constructor, accessor methods, and pretty-printer. This saves a lot of boiler plate code.

Listing 1: src/demo04/demo.py

```
104 @dataclass
105 class Vertex:
106     x: float
107     y: float
108
109
```

Listing 2: src/demo04/demo.py

```
114 @dataclass
115 class Paddle:
116     vertices: list[Vertex]
117     r: float
118     g: float
119     b: float
120
121
```

Although Python is a dynamically-typed language, we can add type information as helpful hints to the reader, and for use with static type-checking tools for Python, such as mypy.

Listing 3: src/demo04/demo.py

```
125 paddle1 = Paddle(
126     vertices=[
127         Vertex(x=-1.0, y=-0.3),
128         Vertex(x=-0.8, y=-0.3),
129         Vertex(x=-0.8, y=0.3),
130         Vertex(x=-1.0, y=0.3),
```

(continues on next page)

(continued from previous page)

```
    ],
    r=0.578123,
    g=0.0,
    b=1.0,
)

paddle2 = Paddle(
    vertices=[Vertex(0.8, -0.3), Vertex(1.0, -0.3), Vertex(1.0, 0.3), Vertex(0.8, 0.3)],
    r=1.0,
    g=1.0,
    b=0.0,
)
```

- Create two instances of a Paddle.

I make heavy use of keyword arguments in Python.

Notice that I am nesting the constructors. I could have instead have written the construction of paddle1 like this:

```
x = -0.8
y = 0.3
vertex_a = Vertex(x, y)
x = -1.0
y = 0.3
vertex_b = Vertex(x, y)
x = -1.0
y = -0.3
vertex_c = Vertex(x, y)
x = -0.8
y = -0.3
vertex_d = Vertex(x, y)
vertex_list = list(vertex_a, vertex_b, vertex_c, vertex_d)
r = 0.57
g = 0.0
b = 1.0
paddle1 = Paddle(vertex_list, r, g, b)
```

But then I would have many local variables, some of whose values change frequently over time, and most of which are single use variables. By nesting the constructors as the author has done above, the author minimizes those issues at the expense of requiring a degree on non-linear reading of the code, which gets easy with practice.

5.4.2 Query User Input and Use It To Animate

$$\vec{x'} = \vec{t}(\vec{x}; \vec{c}) = \vec{x} + \vec{c}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \vec{t}(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}; \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

Listing 4: src/demo04/demo.py

```
def handle_movement_of_paddles() -> None:
    global paddle1, paddle2
    if glfw.get_key(window, glfw.KEY_S) == glfw.PRESS:
```

(continues on next page)

(continued from previous page)

```
        for v in paddle1.vertices:
            v.x += 0.0
            v.y -= 0.1
    if glfw.get_key(window, glfw.KEY_W) == glfw.PRESS:
        for v in paddle1.vertices:
            v.x += 0.0
            v.y += 0.1
    if glfw.get_key(window, glfw.KEY_K) == glfw.PRESS:
        for v in paddle2.vertices:
            v.x += 0.0
            v.y -= 0.1
    if glfw.get_key(window, glfw.KEY_I) == glfw.PRESS:
        for v in paddle2.vertices:
            v.x += 0.0
            v.y += 0.1


```

- If the user presses 's' this frame, subtract 0.1 from the y component of each of the vertices in the paddle. If the key continues to be held down over time, this value will continue to decrease.
- If the user presses 'w' this frame, add 0.1 more to the y component of each of the vertices in the paddle
- If the user presses 'k' this frame, subtract .1.
- If the user presses 'i' this frame, add .1 more.
- when writing to global variables within a nested scope, you need to declare their scope as global at the top of the nested scope. (technically it is not a global variable, it is local to the current python module, but the point remains)

5.4.3 The Event Loop

Monitors can have variable frame-rates, and in order to ensure that movement is consistent across different monitors, we choose to only flush the screen at 60 hertz (frames per second).

Listing 5: src/demo04/demo.py

```
TARGET_FRAMERATE: int = 60

time_at_beginning_of_previous_frame: float = glfw.get_time()
```

Listing 6: src/demo04/demo.py

```
while not glfw.window_should_close(window):
    while (
        glfw.get_time() < time_at_beginning_of_previous_frame + 1.0 / TARGET_FRAMERATE
    ):
        pass

    time_at_beginning_of_previous_frame = glfw.get_time()
```

Listing 7: src/demo04/demo.py

```
186     glfw.poll_events()
187
188     width, height = glfw.get_framebuffer_size(window)
189     glViewport(0, 0, width, height)
190     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

Listing 8: src/demo04/demo.py

```
194     draw_in_square_viewport()
```

Listing 9: src/demo04/demo.py

```
198     handle_movement_of_paddles()
```

- We're still near the beginning of the event loop, and we haven't drawn the paddles yet. So we call the function to query the user input, which will also modify the vertices' values if there was input.

Listing 10: src/demo04/demo.py

```
202     glColor3f(paddle1.r, paddle1.g, paddle1.b)
203
204     glBegin(GL_QUADS)
205     for vertex in paddle1.vertices:
206         glVertex2f(vertex.x, vertex.y)
207     glEnd()
```

- While rendering, we now loop over the vertices of the paddle. The paddles may be displaced from their original position that was hard-coded, as the callback may have updated the values based off of the user input.
- When glVertex is now called, we are not directly passing numbers into it, but instead we are getting the numbers from the data structures of Paddle and its associated vertices.

Listing 11: src/demo04/demo.py

```
211     glColor3f(paddle2.r, paddle2.g, paddle2.b)
212
213     glBegin(GL_QUADS)
214     for vertex in paddle2.vertices:
215         glVertex2f(vertex.x, vertex.y)
216     glEnd()
```

Listing 12: src/demo04/demo.py

```
220     glfw.swap_buffers(window)
```

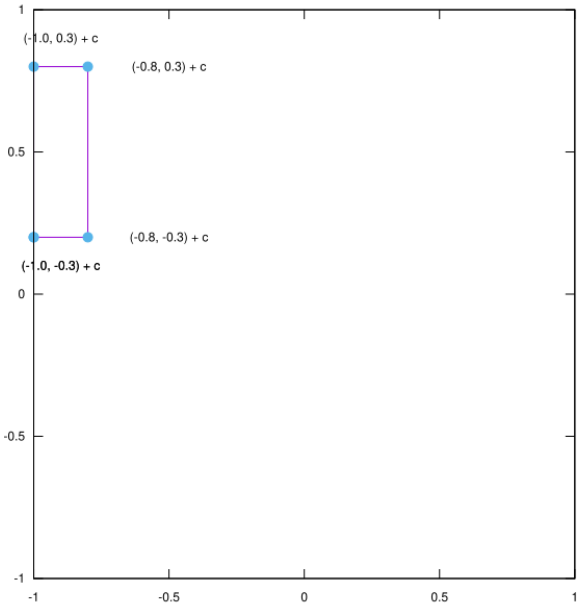


Fig. 2: Adding input offset to Paddle 1

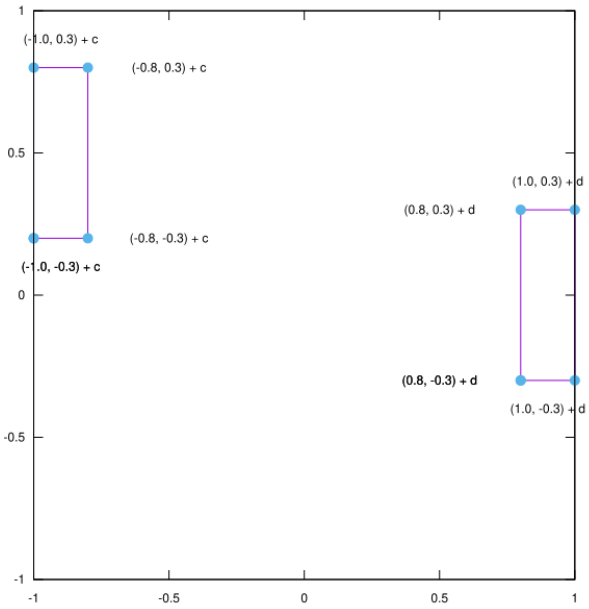


Fig. 3: Adding input offset to Paddle 2

CHAPTER

SIX

ADD TRANSLATE METHOD TO VERTEX - DEMO 05

6.1 Purpose

Restructure the code towards the model view projection pipeline.

Transforming vertices, such as translating, is one of the core concept of computer graphics.

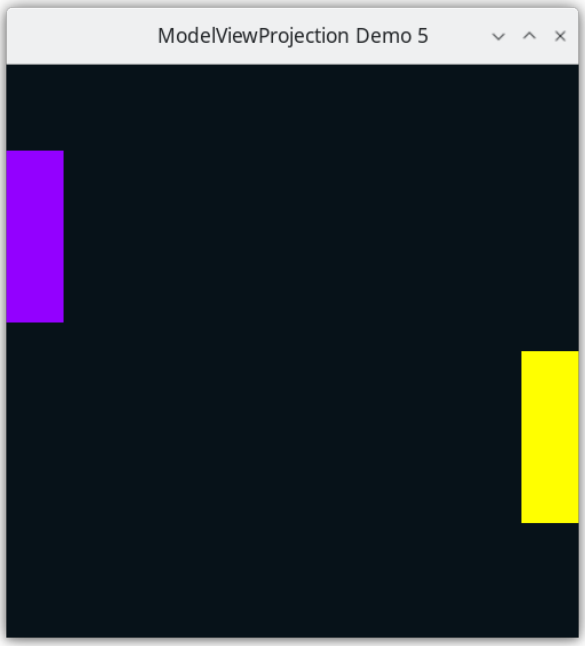


Fig. 1: Demo 05

6.2 How to Execute

Load src/demo05/demo.py in Spyder and hit the play button

6.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up

6.4 Translation

Dealing with the two Paddles the way we did before is not ideal. Both Paddles have the same size, although they are placed in different spots of the screen. We should be able to a set of vertices for the Paddle, relative to the paddle's center, that is independent of its placement in NDC.

Rather than using values for each vertex relative to NDC, in the Paddle data structure, each vertex will be an offset from the center of the Paddle. The center of the paddle will be considered x=0, y=0. Before rendering, each Paddle's vertices will need to be translated to its center relative to NDC.

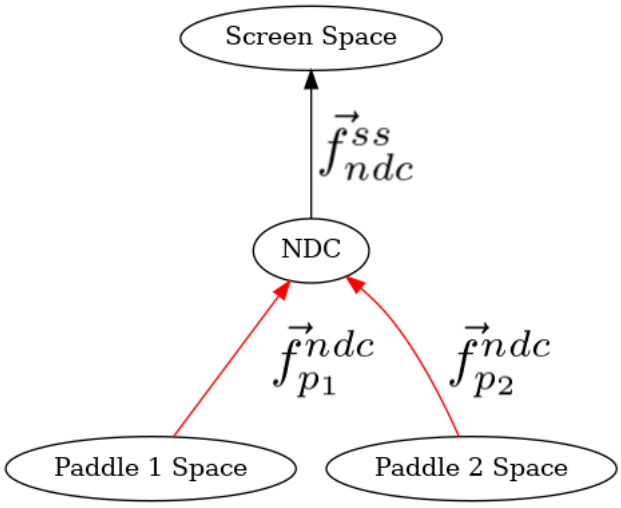


Fig. 2: Paddle space

All methods on vertices will be returning new vertices, rather than mutating the instance variables. The author does this on purpose to enable method-chaining the Python methods, which will be useful later on.

Method-chaining is the equivalent of function composition in math.

6.5 Code

6.5.1 Data Structures

Listing 1: src/demo05/demo.py

```
@dataclass
class Vertex:
    x: float
    y: float

    def translate(self: Vertex, rhs: Vertex) -> Vertex:
        return Vertex(x=self.x + rhs.x, y=self.y + rhs.y)


```

We added a translate method to the Vertex class. Given a translation amount, the vertex will be shifted by that amount. This is a primitive that we will be using to transform from one space to another.

If the reader wishes to use the data structures to test them out, import them and try the methods

```
>>> import src.demo05.demo as demo
>>> a = demo.Vertex(x=1,y=2)
>>> a.translate(demo.Vertex(x=3,y=4))
Vertex(x=4, y=6)
```

Note the use of "keyword arguments". Without using keyword arguments, the code might look like this:

```
>>> import src.demo05.demo as demo
>>> a = demo.Vertex(1,2)
>>> a.translate(demo.Vertex(x=3,y=4))
Vertex(x=4, y=6)
```

Keyword arguments allow the reader to understand the purpose of the parameters are, at the call-site of the function.

Listing 2: src/demo05/demo.py

```
@dataclass
class Paddle:
    vertices: list[Vertex]
    r: float
    g: float
    b: float
    position: Vertex


```

Add a position instance variable to the Paddle class. This position is the center of the paddle, defined relative to NDC. The vertices of the paddle will be defined relative to the center of the paddle.

6.5.2 Instantiation of the Paddles

Listing 3: src/demo05/demo.py

```
paddle1: Paddle = Paddle(
    vertices=[
        Vertex(x=-0.1, y=-0.3),
        Vertex(x=0.1, y=-0.3),
        Vertex(x=0.1, y=0.3),
        Vertex(x=-0.1, y=0.3),
    ],
    r=0.578123,
    g=0.0,
    b=1.0,
    position=Vertex(-0.9, 0.0),
)

paddle2: Paddle = Paddle(
    vertices=[
        Vertex(-0.1, -0.3),
        Vertex(0.1, -0.3),
        Vertex(0.1, 0.3),
        Vertex(-0.1, 0.3),
    ],
    r=1.0,
    g=1.0,
    b=0.0,
    position=Vertex(0.9, 0.0),
)
```

- The vertices are now defined as relative distances from the center of the paddle. The centers of each paddle are placed in positions relative to NDC that preserve the positions of the paddles, as they were in the previous demo.

6.5.3 Handling User Input

Listing 4: src/demo05/demo.py

```
def handle_movement_of_paddles() -> None:
    global paddle1, paddle2

    if glfw.get_key(window, glfw.KEY_S) == glfw.PRESS:
        paddle1.position.y -= 0.1
    if glfw.get_key(window, glfw.KEY_W) == glfw.PRESS:
        paddle1.position.y += 0.1
    if glfw.get_key(window, glfw.KEY_K) == glfw.PRESS:
        paddle2.position.y -= 0.1
    if glfw.get_key(window, glfw.KEY_I) == glfw.PRESS:
        paddle2.position.y += 0.1


```

- We put the transformation on the center of the paddle, instead of directly on each vertex. This is because the vertices are defined relative to the center of the paddle.

6.5.4 The Event Loop

Listing 5: src/demo05/demo.py

```
while not glfw.window_should_close(window):
    while (
        glfw.get_time() < time_at_beginning_of_previous_frame + 1.0 / TARGET_FRAMERATE
    ):
        pass

    time_at_beginning_of_previous_frame = glfw.get_time()

    glfw.poll_events()

    width, height = glfw.get_framebuffer_size(window)
    glViewport(0, 0, width, height)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    draw_in_square_viewport()
    handle_movement_of_paddles()
```

Listing 6: src/demo05/demo.py

```
    glColor3f(paddle1.r, paddle1.g, paddle1.b)

    glBegin(GL_QUADS)
    for paddle1_vertex_ms in paddle1.vertices:
        paddle1_vertex_ndc: Vertex = paddle1_vertex_ms.translate(paddle1.position)
        glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
    glEnd()
```

Here each of paddle 1's vertices, which are in their "model-space", are converted to NDC by calling the translate

method on the vertex. This function corresponds to the Cayley graph below, the function from Paddle 1 space to NDC.

Listing 7: src/demo05/demo.py

```
209     glColor3f(paddle2.r, paddle2.g, paddle2.b)
210
211     glBegin(GL_QUADS)
212     for paddle2_vertex_ms in paddle2.vertices:
213         paddle2_vertex_ndc: Vertex = paddle2_vertex_ms.translate(paddle2.position)
214         glVertex2f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y)
215     glEnd()
```

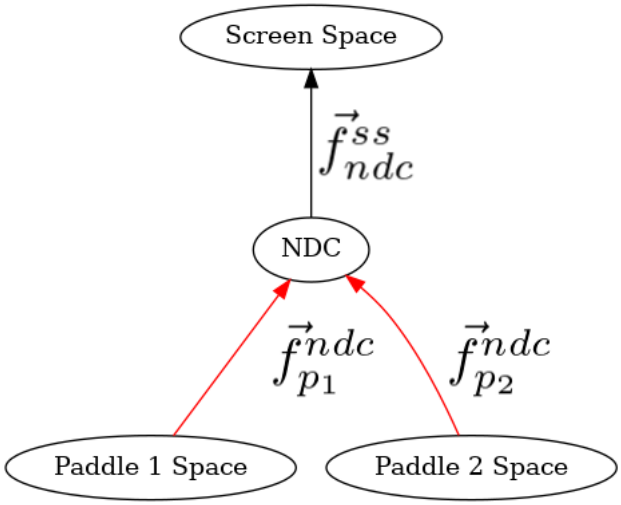


Fig. 3: Paddle space

The only part of the diagram that we need to think about right now is the function that converts from paddle1's space to NDC, and from paddle2's space to NDC.

These functions in the Python code are the translation of the paddle's center (i.e. paddle1.position) by the vertex's offset from the center.

N.B. In the code, I name the vertices by their space. I.e. "modelSpace" instead of "vertex_relative_to_modelspace". I do this to emphasize that you should view the transformation as happening to the "graph paper", instead of to each of the points. This will be explained more clearly later.

CHAPTER

SEVEN

MODELSPACE - DEMO 06

7.1 Purpose

Learn about modelspace.

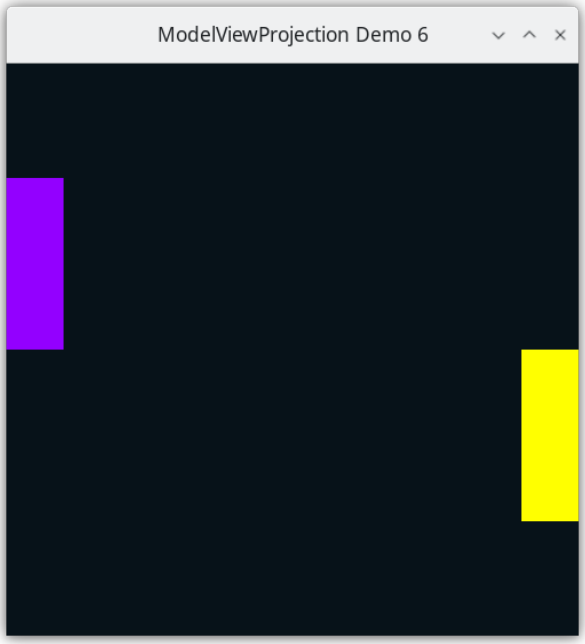


Fig. 1: Demo 06

7.2 How to Execute

Load src/demo06/demo.py in Spyder and hit the play button

7.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up

7.4 Modelspace

Normalized-device-coordinates are not a natural system of numbers for use by humans. Imagine that the paddles in the previous chapters exist in real life, and are 2 meters wide and 6 meters tall. The graphics programmer should be able to use those numbers directly; they shouldn't have to manually transform the distances into normalized-device-coordinates.

Whatever a convenient numbering system is (i.e. coordinate system) for modeling objects is called "model-space". Since a paddle has four corners, which corner should be at the origin (0,0)? If you don't already know what you want at the origin, then none of the corners should be; instead put the center of the object at the origin (Because by putting the center of the object at the origin, scaling and rotating the object are trivial, as shown in later chapters).

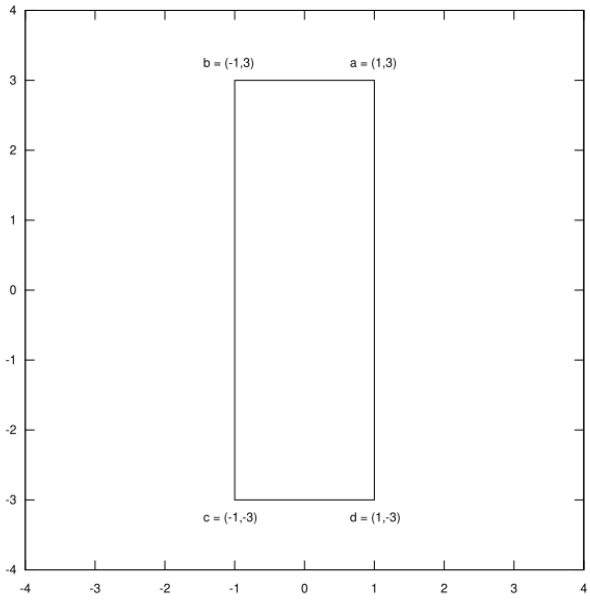


Fig. 2: Representing a Paddle using Modelspace

Modelspace - the coordinate system (origin plus axes), in which some object's vertices are defined.

7.5 WorldSpace

WorldSpace is a top-level space, independent of NDC, that we choose to use. It is arbitrary. If you were to model a racetrack for a racing game, the origin of WorldSpace may be the center of that racetrack. If you were modeling our solar system, the center of the sun could be the origin of "WorldSpace". I personally would put the center of our flat earth at the origin, but reasonable people can disagree.

For our demo with paddles, the author arbitrarily defines the WorldSpace to be 20 units wide, 20 units tall, with the origin at the center.

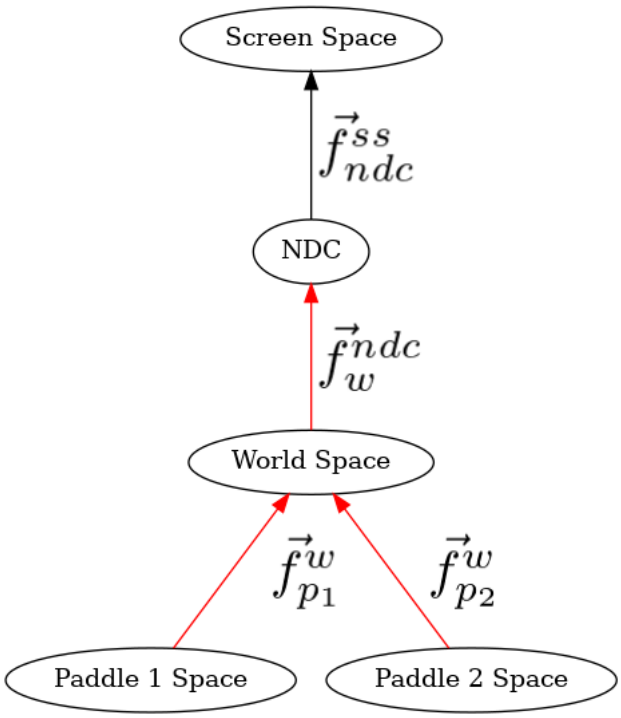


Fig. 3: Demo 06

7.6 Modelspace to WorldSpace

The author prefers to view transformations as changes to the graph paper, as compared to view transformations as changes to points.

As such, for placing paddle1, we can view the translation as a change to the graph paper relative to world space coordinates (only incidentally bringing the vertices along with it) and then resetting the graph paper (i.e. both origin and axes) back to its original position and orientation. Although we will think of the paddle's vertices as relative to its own space (i.e. -1 to 1 horizontally, -3 to 3 vertically), we will not look at the numbers of what they are in world space coordinates, as doing so

- Will not give us any insight
- Will distract us from thinking clearly about what's happening
- As an example, figure out the world space coordinate of the upper rights corner of the paddle after it has been translated, and ask yourself what that means and what insight did you gain?

The animation above shows multiple steps, shown now without animation.

7.6.1 Modelspace of Paddle 1

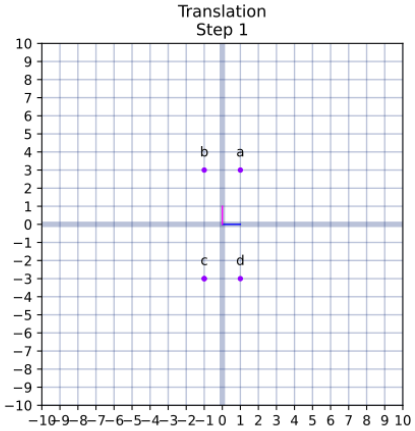


Fig. 4: Paddle 1's Modelspace

Vertex	Coordinates
a	(1,3)
b	(-1,3)
c	(-1,-3)
d	(1,-3)

7.6.2 Modelspace of Paddle 1 Superimposed on Worldspace after the translation

Paddle 1's graph paper gets translated -9 units in the x direction, and some number of units in the y direction, 0 during the first frame, based off of user input. The origin is translated, and the graph paper comes with it, onto which you can plot the vertices. Notice that the coordinate system labels below the plot and to the left of the plot is unchanged. That is world space, which has not changed.

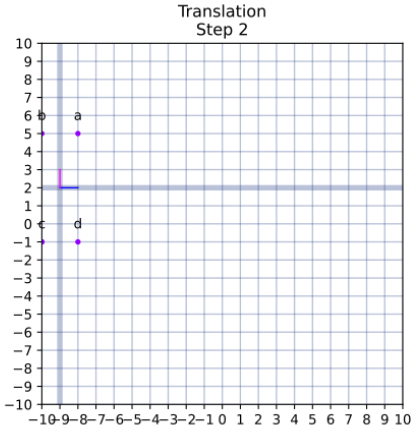


Fig. 5: Paddle 1's Modelspace Superimposed on World Space

Vertex	Coordinates (modelspace)	Coordinates (worldspace)
a	(1,3)	(1,3) + (9,3) = (-8,5)
b	(-1,3)	(-1,3) + (9,3) = (-10,5)
c	(-1,-3)	(-1,-3) + (9,3) = (-10,-1)
d	(1,-3)	(1,-3) + (9,3) = (-8,-1)

7.6.3 Paddle 1's vertices in WorldSpace Coordinates

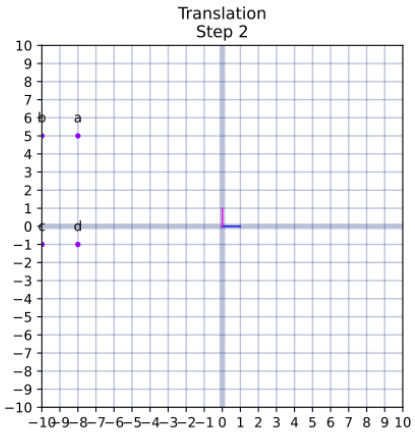


Fig. 6: Paddle 1's Vertices in World Space.

Vertex	Coordinates (worldspace)
a	(-8,5)
b	(-10,5)
c	(-10,-1)
d	(-8,-1)

Now that the transformation has happened, the vertices are all in world space. You could calculate their values in world space, but that will not give you any insight. The only numbers that matter for insight as that the entire graph paper of modelspace, which originally was the same as world space, has changed, bringing the vertices along with it.

Same goes for Paddle 2's modelspace, relative to its translation, which are different values.

7.6.4 Modelspace of Paddle 2

Vertex	Coordinates
a	(1,3)
b	(-1,3)
c	(-1,-3)
d	(1,-3)

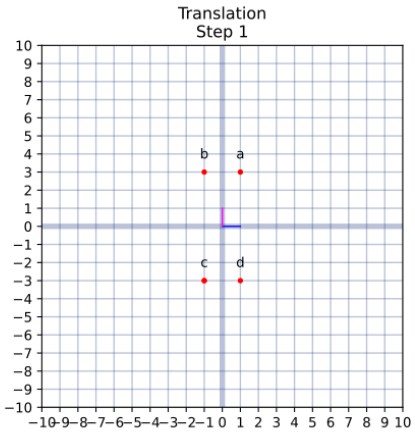


Fig. 7: Paddle 2's Modelspace

7.6.5 Modelspace of Paddle 2 Superimposed on Worldspace after the translation

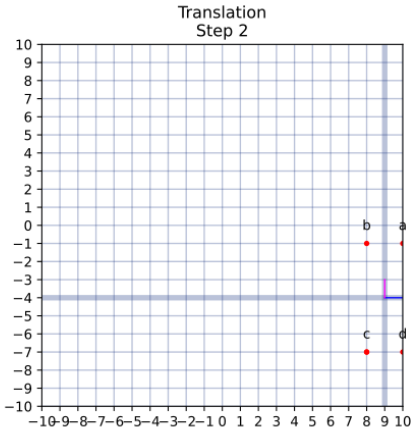


Fig. 8: Paddle 2's Modelspace Superimposed on World Space

Vertex	Coordinates (modelspace)	Coordinates (worldspace)
a	(1,3)	(1,3) + (9,-4) = (10,-1)
b	(-1,3)	(-1,3) + (9,-4) = (8,-1)
c	(-1,-3)	(-1,-3) + (9,-4) = (8,-7)
d	(1,-3)	(1,-3) + (9,-4) = (10,-7)

7.6.6 Paddle 2's vertices in WorldSpace Coordinates

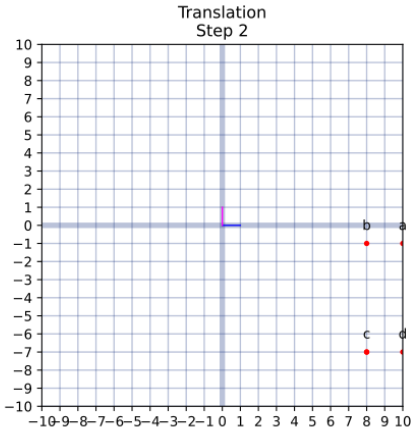


Fig. 9: Paddle 2's Vertices in World Space.

Vertex	Coordinates (worldspace)
a	(10,-1)
b	(8,-1)
c	(8,-7)
d	(10,-7)

7.6.7 Scaling

Our paddles are now well outside of NDC, and as such, they would not be displayed, as they would be *clipped* out. Their values are outside of -1.0 to 1.0. All we will need to do to convert them from world space to NDC is divide each component, x and y, by 10.

As a demonstration of how scaling works, let's make an object's width twice as large, and height three times as large. (The author tried doing the actual scaling of 1/10 in an animated gif, and it looked awful, therefore a different scaling gif is showed here, but the concept is the same).

We can expand or shrink the size of an object by "scale"ing each component of the vertices by some coefficient.

Our global space is -10 to 10 in both dimensions, and to get it into NDC, we need to scale by dividing by 10

$$\begin{bmatrix} x_w \\ y_w \end{bmatrix} = \vec{f}^{w}_{p1}(\begin{bmatrix} x_{p1} \\ y_{p1} \end{bmatrix}) = \begin{bmatrix} x_{p1} \\ y_{p1} \end{bmatrix} + \begin{bmatrix} p1_x \\ p1_y \end{bmatrix}$$

where x_p1, y_p1 are the modelspace coordinates of the paddle's vertices, and where p1_center_x_worldspace, p1_center_y_worldspace, are the offset from the world space's origin to the center of the paddle, i.e. the translation.

$$\begin{bmatrix} x_w \\ y_w \end{bmatrix} = \vec{f}^{w}_{p2}(\begin{bmatrix} x_{p2} \\ y_{p2} \end{bmatrix}) = \begin{bmatrix} x_{p2} \\ y_{p2} \end{bmatrix} + \begin{bmatrix} p2_x \\ p2_y \end{bmatrix}$$

Now, the coordinates for paddle 1 and for paddle 2 are in world space, and we need the match to take any world space coordinates and convert them to NDC.

$$\begin{bmatrix} x_{ndc} \\ y_{ndc} \end{bmatrix} = \vec{f}^{ndc}_{w}(\begin{bmatrix} x_w \\ y_w \end{bmatrix}) = 1/10 * \begin{bmatrix} x_w \\ y_w \end{bmatrix}$$

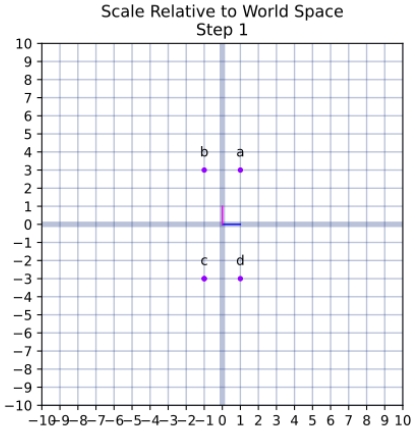


Fig. 10: Modelspace

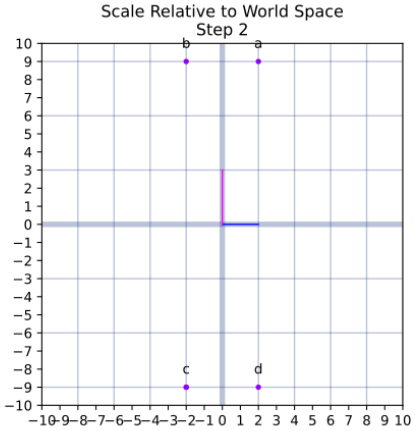


Fig. 11: Modelspace Superimposed on World Space

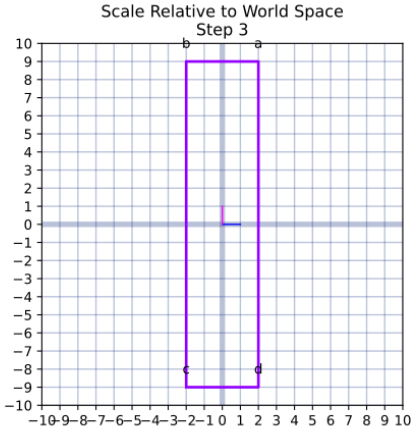


Fig. 12: Worldspace. Don't concern yourself with what the numbers are.

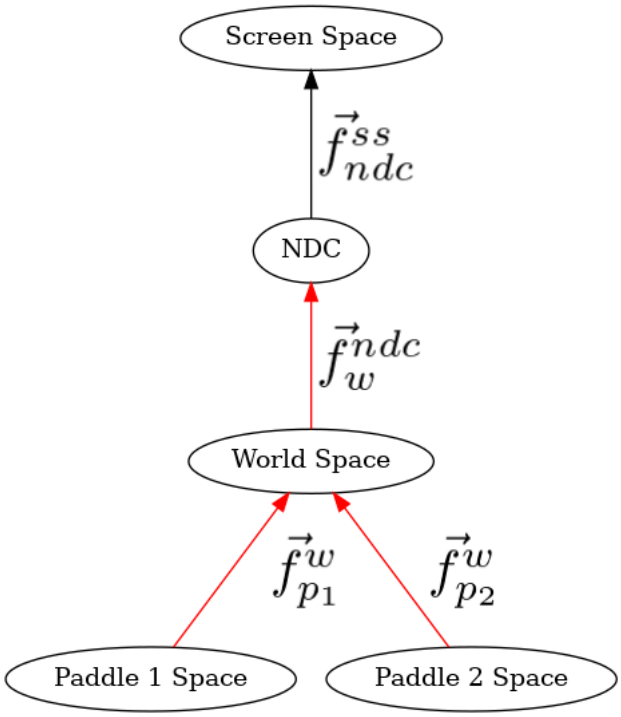


Fig. 13: Demo 06

Listing 1: src/demo06/demo.py

```
@dataclass
class Vertex:
    x: float
    y: float

    def translate(self: Vertex, rhs: Vertex) -> Vertex:
        return Vertex(x=(self.x + rhs.x), y=(self.y + rhs.y))

```

Listing 2: src/demo06/demo.py

```
    def uniform_scale(self: Vertex, scale: float) -> Vertex:
        return Vertex(x=(self.x * scale), y=(self.y * scale))

```

- NEW – Add the ability to scale a vertex, to stretch or to shrink

Listing 3: src/demo06/demo.py

```
paddle1: Paddle = Paddle(
    vertices=[
        Vertex(x=-1.0, y=-3.0),
        Vertex(x=1.0, y=-3.0),
        Vertex(x=1.0, y=3.0),
        Vertex(x=-1.0, y=3.0),
```

(continues on next page)

(continued from previous page)

```
    ],
    r=0.578123,
    g=0.0,
    b=1.0,
    position=Vertex(-9.0, 0.0),
)

paddle2: Paddle = Paddle(
    vertices=[
        Vertex(x=-1.0, y=-3.0),
        Vertex(x=1.0, y=-3.0),
        Vertex(x=1.0, y=3.0),
        Vertex(x=-1.0, y=3.0),
    ],
    r=1.0,
    g=1.0,
    b=0.0,
    position=Vertex(9.0, 0.0),
)
```

- paddles are using modelspace coordinates instead of NDC

Listing 4: src/demo06/demo.py

```
def handle_movement_of_paddles() -> None:
    global paddle1, paddle2

    if glfw.get_key(window, glfw.KEY_S) == glfw.PRESS:
        paddle1.position.y -= 1.0
    if glfw.get_key(window, glfw.KEY_W) == glfw.PRESS:
        paddle1.position.y += 1.0
    if glfw.get_key(window, glfw.KEY_K) == glfw.PRESS:
        paddle2.position.y -= 1.0
    if glfw.get_key(window, glfw.KEY_I) == glfw.PRESS:
        paddle2.position.y += 1.0


```

- Movement code needs to happen in Modelspace's units.

7.7 Code

7.7.1 The Event Loop

Listing 5: src/demo06/demo.py

```
while not glfw.window_should_close(window):
    while (
        glfw.get_time() < time_at_beginning_of_previous_frame + 1.0 / TARGET_FRAMERATE
    ):
```

(continues on next page)

(continued from previous page)

```
        pass
    time_at_beginning_of_previous_frame = glfw.get_time()

    glfw.poll_events()

    width, height = glfw.get_framebuffer_size(window)
    glViewport(0, 0, width, height)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    draw_in_square_viewport()
    handle_movement_of_paddles()
```

Rendering Paddle 1

Listing 6: src/demo06/demo.py

```
    glColor3f(paddle1.r, paddle1.g, paddle1.b)

    glBegin(GL_QUADS)
    for paddle1_vertex_ms in paddle1.vertices:
```

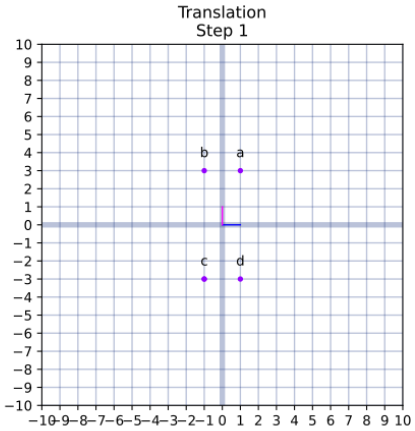


Fig. 14: Paddle 1's Modelspace

Listing 7: src/demo06/demo.py

```
        paddle1_vertex_ws: Vertex = paddle1_vertex_ms.translate(paddle1.position)
```

The coordinate system now resets back to the coordinate system specified on the left and below. Now, we must scale everything by 1/10. I have not included a picture of that here. Scaling happens relative to the origin, so the picture would be the same, just with different labels on the bottom and on the left.

Listing 8: src/demo06/demo.py

```
        paddle1_vertex_ndc: Vertex = paddle1_vertex_ws.uniform_scale(1.0 / 10.0)
```

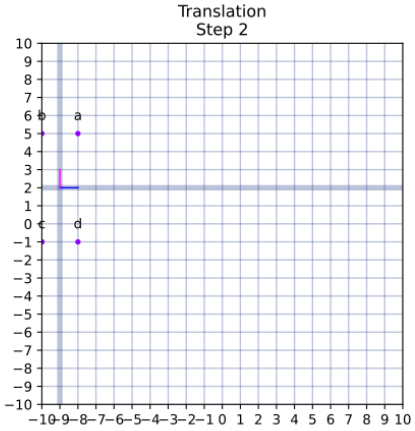


Fig. 15: Paddle 1's Modelspace Superimposed on World Space

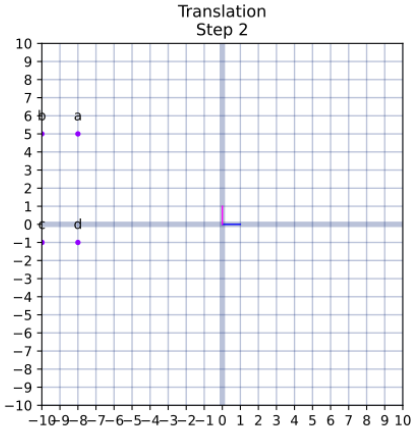


Fig. 16: Reset coordinate system.

Listing 9: src/demo06/demo.py

```
212         glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
213
214     glEnd()
```

Rendering Paddle 2

Listing 10: src/demo06/demo.py

```
218     glColor3f(paddle2.r, paddle2.g, paddle2.b)
219
220     glBegin(GL_QUADS)
221     for paddle2_vertex_ms in paddle2.vertices:
```

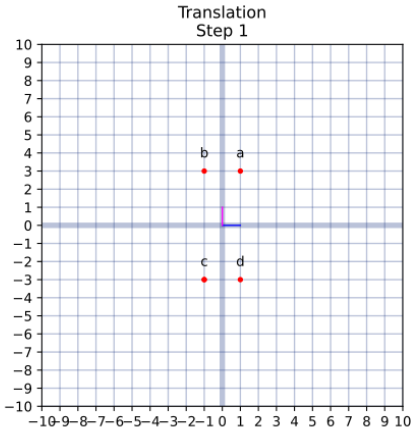


Fig. 17: Paddle 2's Modelspace

Listing 11: src/demo06/demo.py

```
224         paddle2_vertex_ws: Vertex = paddle2_vertex_ms.translate(paddle2.position)
```

Listing 12: src/demo06/demo.py

```
227         paddle2_vertex_ndc: Vertex = paddle2_vertex_ws.uniform_scale(1.0 / 10.0)
```

Listing 13: src/demo06/demo.py

```
231         glVertex2f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y)
232     glEnd()
```

The coordinate system is reset. Now scale everything by 1/10. I have not included a picture of that here. Scaling happens relative to the origin, so the picture would be the same, just with different labels on the bottom and on the left.

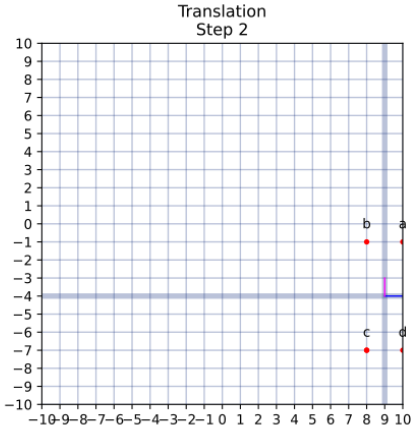


Fig. 18: Paddle 2's Modelspace Superimposed on World Space

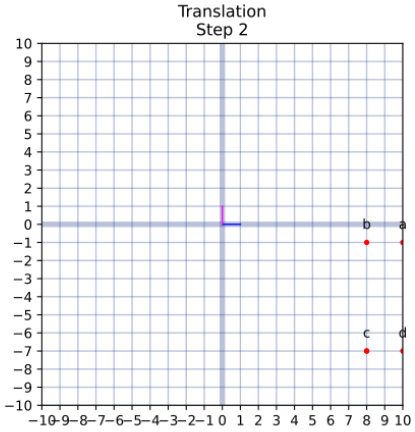


Fig. 19: Reset coordinate system.

Listing 14: src/demo06/demo.py

```
236     glfw.swap_buffers(window)
```

CHAPTER

EIGHT

ROTATIONS - DEMO 07

8.1 Purpose

Attempt to rotate the paddles around their center. Learn about rotations. This demo does not work correctly, because of a misunderstanding of how to interpret a sequence of transformations.

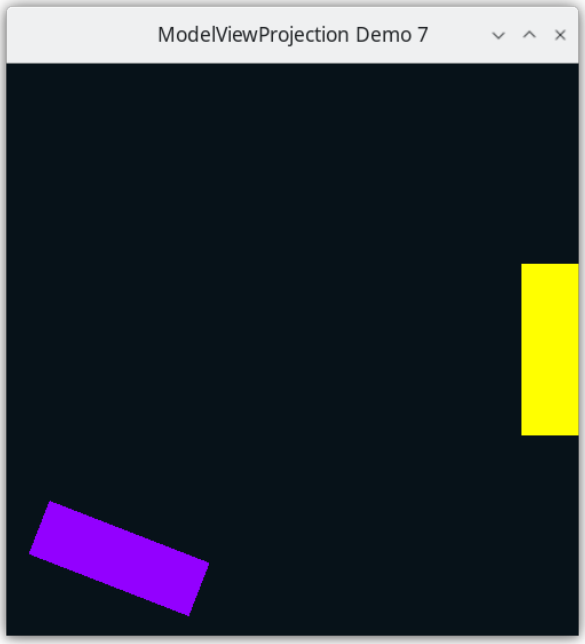


Fig. 1: Demo 07

8.2 How to Execute

Load src/demo07/demo.py in Spyder and hit the play button

8.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation

For another person's explanation of the trigonometry of rotating in 2D, see

8.4 Rotate the Paddles About their Center

Besides translate and scale, the third main operation in computer graphics is to rotate an object.

8.5 Rotation Around Origin (0,0)

We can rotate an object around (0,0) by rotating all of the object's vertices around (0,0).

In high school math, you will have learned about sin, cos, and tangent. Typically the angles are described on the unit circle, where a rotation starts from the positive x axis.

We can expand on this knowledge, allowing us to rotate a given vertex, wherever it is, around the origin (0,0). This is done by separating the x and y value, rotating each component separately, and then adding the results together.

That might not have been fully clear. Let me try again. The vertex (0.5,0.4) can be separated into two vertices, (0.5,0) and (0,0.4).

These vertices can be added together to create the original vertex. But, before we do that, let's rotate each of the vertices.

(0.5,0) is on the x-axis, so rotating it by "angle" degrees, results in vertex (0.5*cos(angle), 0.5*sin(angle)). This is high school geometry, and won't be explained here in detail.

But what may not be obvious, is what happens to the y component? Turns out, it is easy. Just rotate your point of view, and it is the same thing, with one minor difference, in that the x value is negated.

(0,0.4) is on the y-axis, so rotating it by "angle" degrees, results in vertex (0.4*-sin(angle), 0.4*cos(angle)).

Wait. Why is negative sin applied to the angle to make the x value, and cos applied to angle to make the y value? It is because positive 1 unit on the x axis goes downwards. The top of the plot is in units of -1.

After the rotations have been applied, sum the results to get your vertex rotated around the origin!

(0.5*cos(angle), 0.5*sin(angle)) + (0.4*-sin(angle), 0.4*cos(angle)) = (0.5*cos(angle) + 0.4*-sin(angle), 0.5*sin(angle) + 0.4*cos(angle))

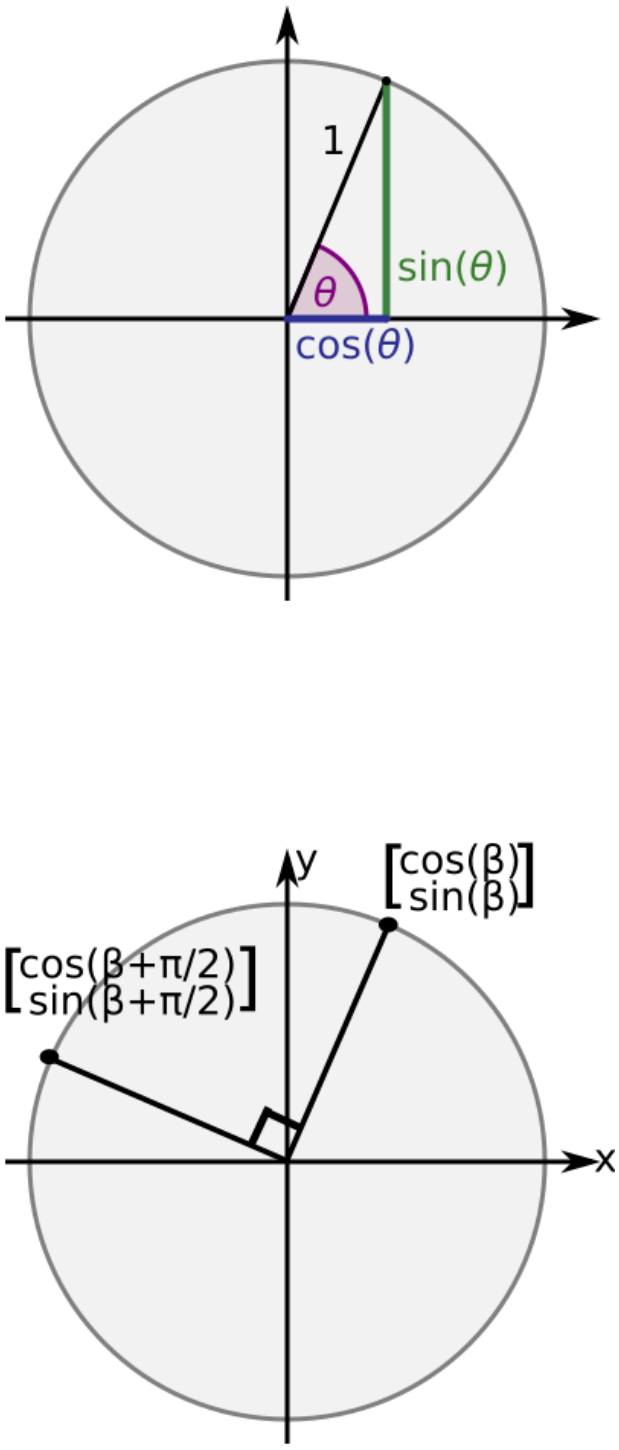


Fig. 2: Rotate

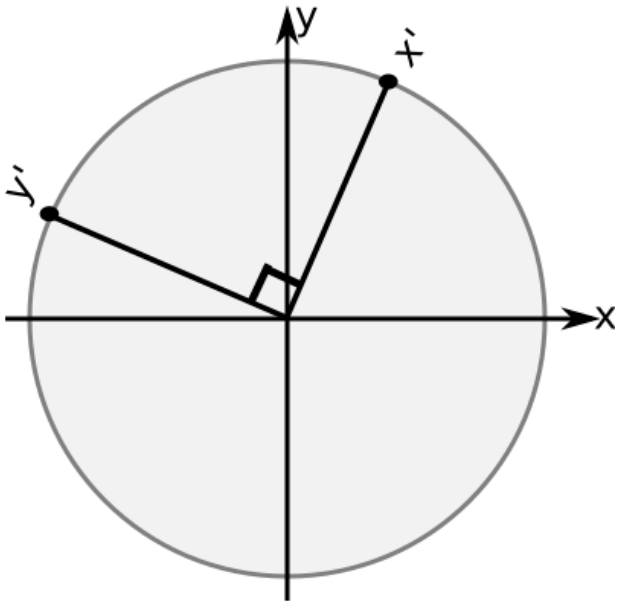


Fig. 3: Rotate

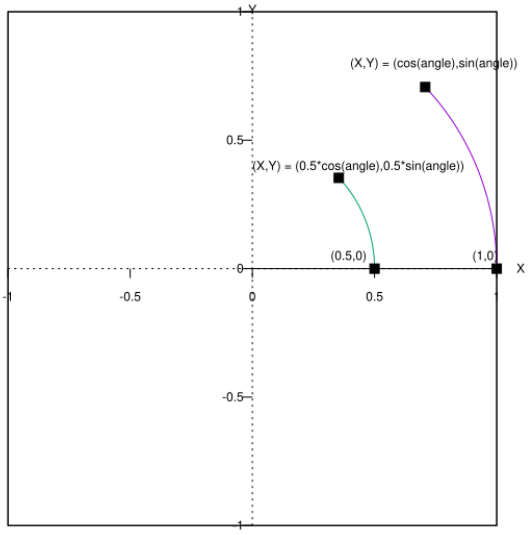


Fig. 4: Rotate the x component.

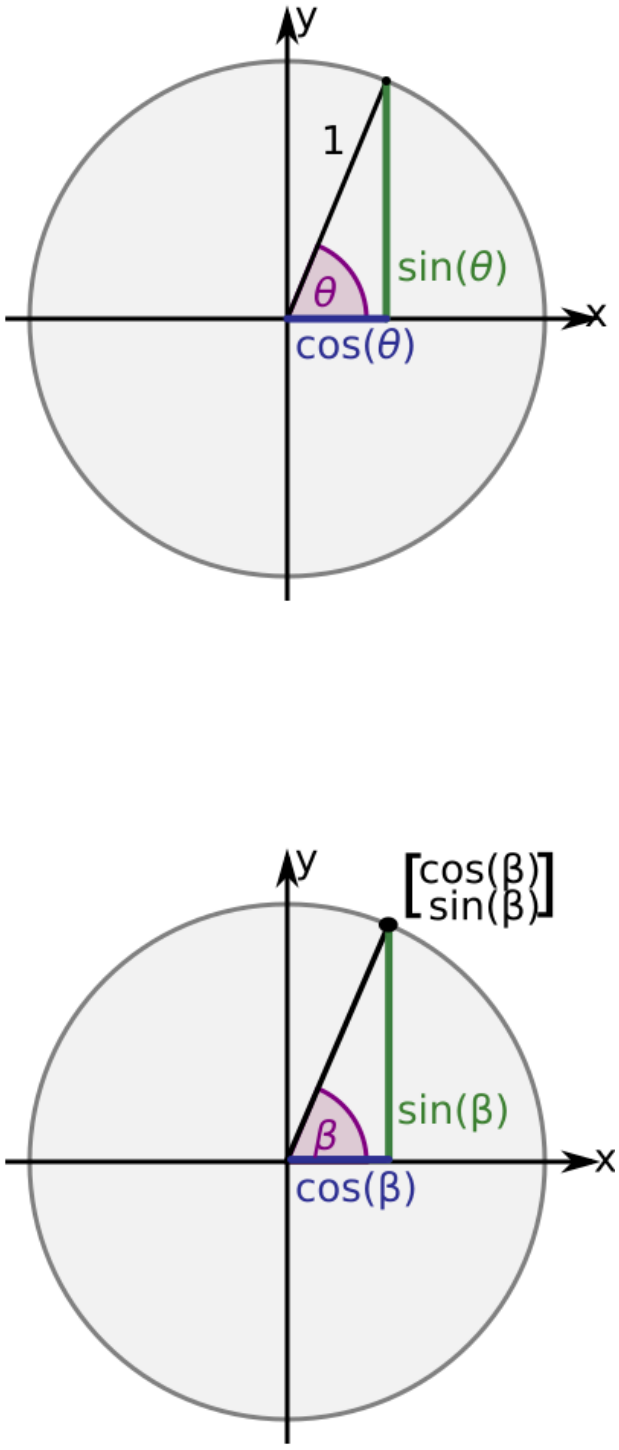
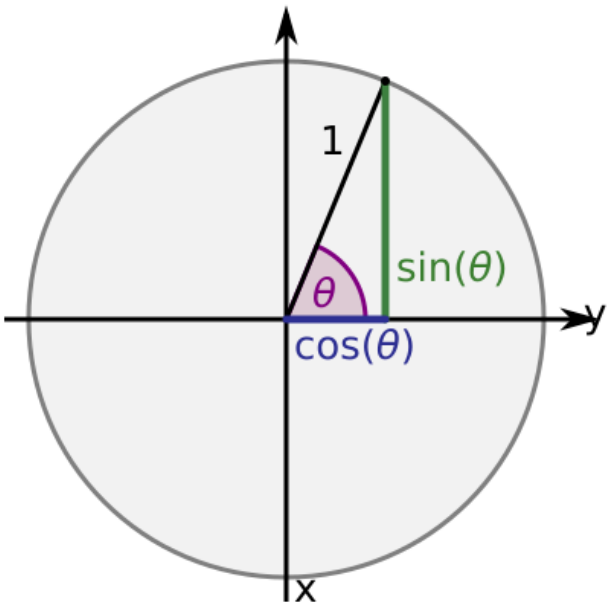


Fig. 5: Rotate the y component



Listing 1: src/demo07/demo.py

```
@dataclass
class Vertex:
    x: float
    y: float

    def __add__(self, rhs: Vertex) -> Vertex:
        return Vertex(x=self.x + rhs.x, y=self.y + rhs.y)

    def translate(self: Vertex, translate_amount: Vertex) -> Vertex:
        return self + translate_amount

    def __mul__(self, scalar: float) -> Vertex:
        return Vertex(x=self.x * scalar, y=self.y * scalar)

    def __rmul__(self, scalar: float) -> Vertex:
        return self * scalar

    def uniform_scale(self: Vertex, scalar: float) -> Vertex:
        return self * scalar

    def scale(self: Vertex, scale_x: float, scale_y: float) -> Vertex:
        return Vertex(x=self.x * scale_x, y=self.y * scale_y)

    def __neg__(self):
        return -1.0 * self

    def rotate_90_degrees(self: Vertex):
        return Vertex(x=-self.y, y=self.x)

```

(continues on next page)

(continued from previous page)

```
    # fmt: off
    def rotate(self: Vertex, angle_in_radians: float) -> Vertex:
        return math.cos(angle_in_radians) * self + math.sin(angle_in_radians) * self.
↪rotate_90_degrees()
    # fmt: on

```

- Note the definition of rotate, from the description above. cos and sin are defined in the math module.

Listing 2: src/demo07/demo.py

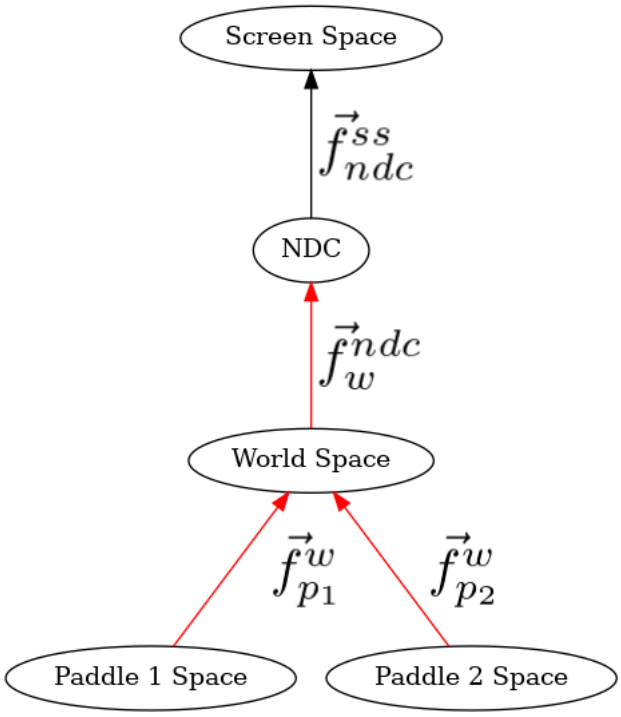
```
@dataclass
class Paddle:
    vertices: list[Vertex]
    r: float
    g: float
    b: float
    position: Vertex
    rotation: float = 0.0
```

- a rotation instance variable is defined, with a default value of 0

Listing 3: src/demo07/demo.py

```
def handle_movement_of_paddles() -> None:
    global paddle1, paddle2

    if glfw.get_key(window, glfw.KEY_S) == glfw.PRESS:
        paddle1.position.y -= 1.0
    if glfw.get_key(window, glfw.KEY_W) == glfw.PRESS:
        paddle1.position.y += 1.0
    if glfw.get_key(window, glfw.KEY_K) == glfw.PRESS:
        paddle2.position.y -= 1.0
    if glfw.get_key(window, glfw.KEY_I) == glfw.PRESS:
        paddle2.position.y += 1.0

    if glfw.get_key(window, glfw.KEY_A) == glfw.PRESS:
        paddle1.rotation += 0.1
    if glfw.get_key(window, glfw.KEY_D) == glfw.PRESS:
        paddle1.rotation -= 0.1
    if glfw.get_key(window, glfw.KEY_J) == glfw.PRESS:
        paddle2.rotation += 0.1
    if glfw.get_key(window, glfw.KEY_L) == glfw.PRESS:
        paddle2.rotation -= 0.1
```

8.6 Cayley Graph



8.7 Code

8.7.1 The Event Loop

Listing 4: src/demo07/demo.py

```
212 while not glfw.window_should_close(window):
```

So to rotate paddle 1 about its center, we should translate to its position, and then rotate around the paddle's center.

Listing 5: src/demo07/demo.py

```
230     glColor3f(paddle1.r, paddle1.g, paddle1.b)
231
232     glBegin(GL_QUADS)
233     for paddle1_vertex_ms in paddle1.vertices:
```

$$\vec{f}^{w}_{p1}$$

Listing 6: src/demo07/demo.py

```
236         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.translate(
237             paddle1.position
238         ).rotate(paddle1.rotation)
```

$$\vec{f}_{w}^{ndc}$$

Listing 7: src/demo07/demo.py

```
241         paddle1_vertex_ndc: Vertex = paddle1_vertex_ws.uniform_scale(1.0 / 10.0)
```

Listing 8: src/demo07/demo.py

```
244         glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
245     glEnd()
```

```
...
```

Likewise, to rotate paddle 2 about its center, we should translate to its position, and then rotate around the paddle's center.

Listing 9: src/demo07/demo.py

```
251     glBegin(GL_QUADS)
252     for paddle2_vertex_ms in paddle2.vertices:
```

$$\vec{f}_{p2}^{w}$$

Listing 10: src/demo07/demo.py

```
255         paddle2_vertex_ws: Vertex = paddle2_vertex_ms.translate(
256             paddle2.position
257         ).rotate(paddle2.rotation)
```

$$\vec{f}_{w}^{ndc}$$

Listing 11: src/demo07/demo.py

```
260         paddle2_vertex_ndc: Vertex = paddle2_vertex_ws.uniform_scale(1.0 / 10.0)
```

Listing 12: src/demo07/demo.py

```
263         glVertex2f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y)
264     glEnd()
```

8.8 Why it is Wrong

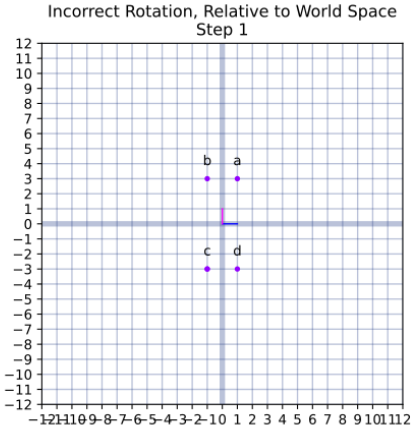
Turns out, our program doesn't work as predicted, even though translate, scale, and rotate are all defined correctly. The paddles are not rotating around their center.

Let's take a look in detail about what our paddle-space to world space transformations are doing.

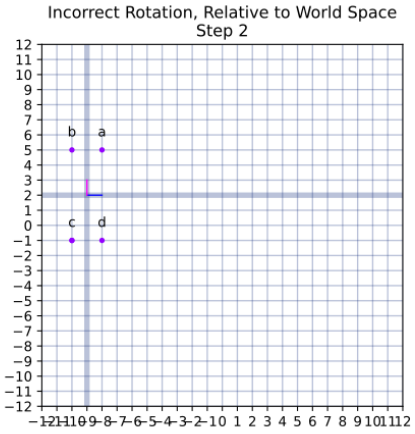
Listing 13: src/demo07/demo.py

```
236         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.translate(
237             paddle1.position
238         ).rotate(paddle1.rotation)
```

- Modelspace vertices



- Translate



- Reset the coordinate system
- Rotate around World Spaces's origin
- Reset the coordinate system
- Final world space coordinates

So then what the heck are we supposed to do in order to rotate around an object's center? The next section provides a solution.

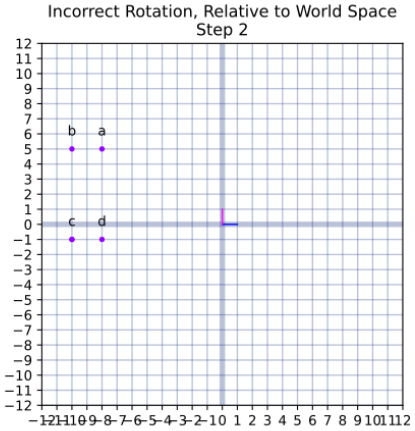


Fig. 6: Modelspace

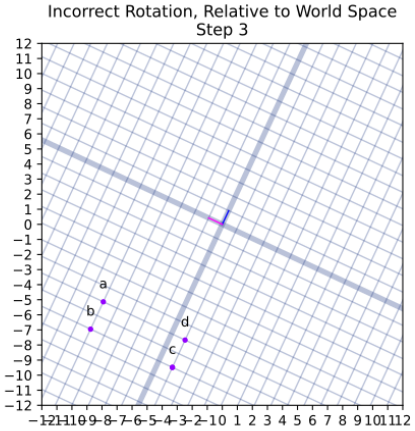


Fig. 7: Modelspace

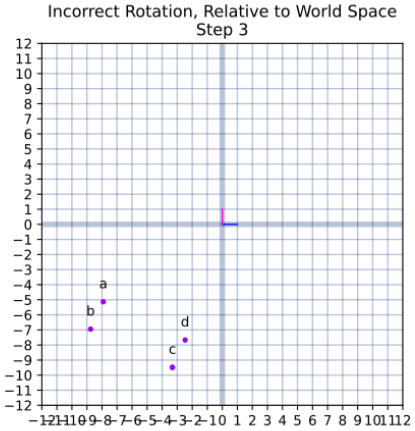


Fig. 8: Modelspace

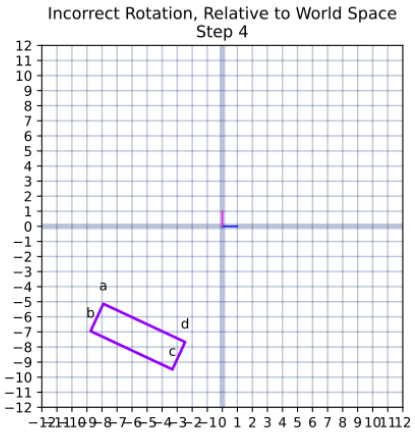


Fig. 9: Modelspace

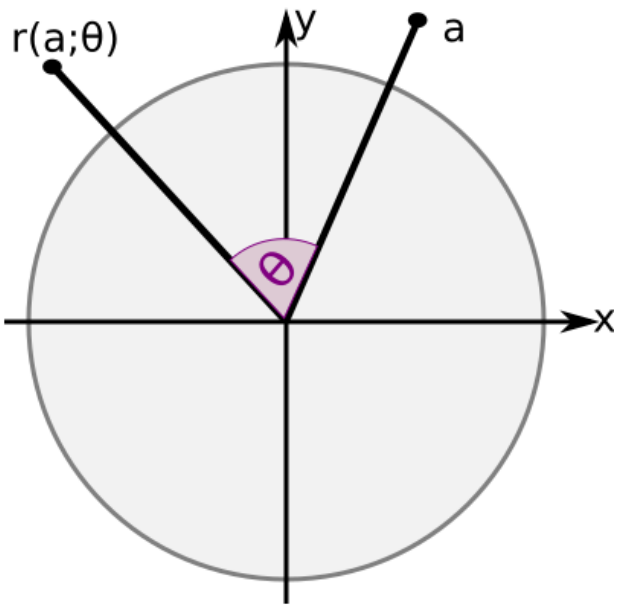
8.9 Updated explanation

We can expand on this knowledge, allowing us to rotate all vertices, wherever they are, around the origin (0,0), by some angle theta.

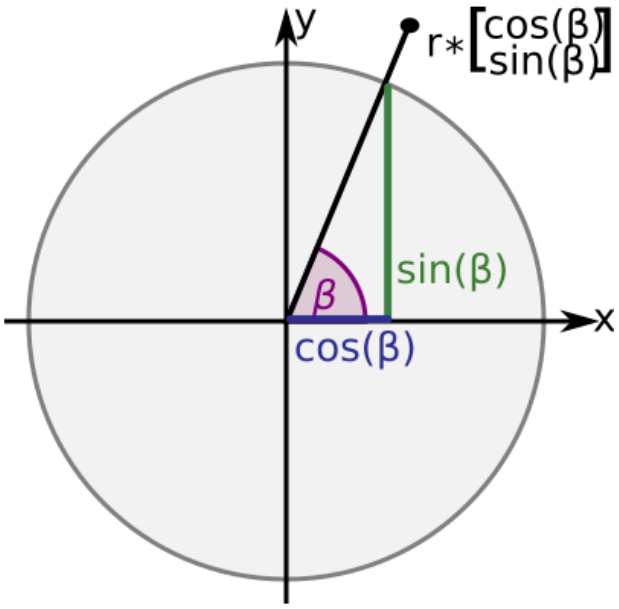
Let's take any arbitrary point

$$\vec{a}$$

$$\vec{r}(\vec{a};\theta)$$

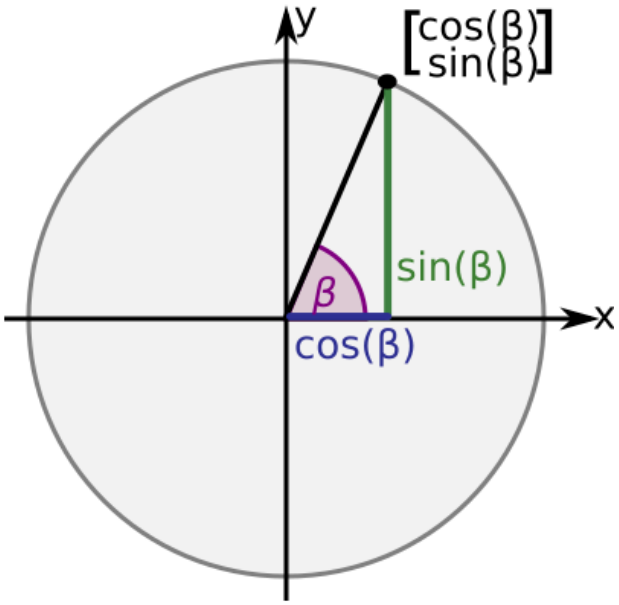


From high school geometry, remember that we can describe a Cartesian (x,y) point by its length r, and its cosine and sine of its angle theta.



Also remember that when calculating sine and cosine, because right triangles are proportional, that sine and cosine are preserved for a right triangle, even if the length sides are scaled up or down.

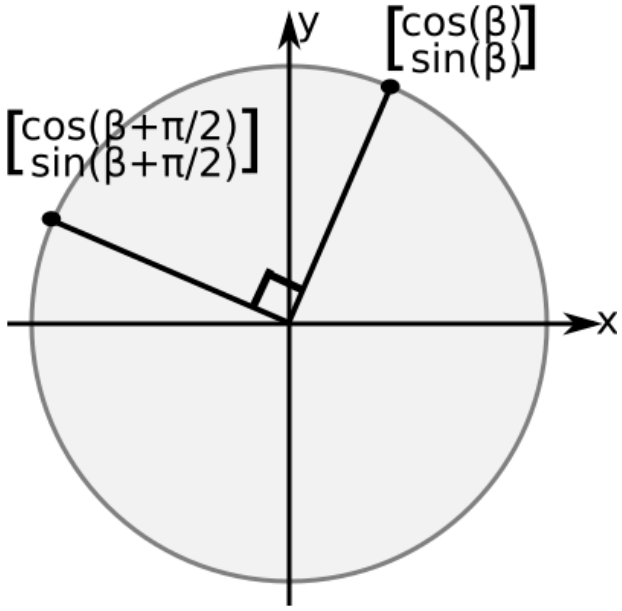
So we'll make a right triangle on the unit circle, but we will remember the length of (x,y), which we'll call "r", and we'll call the angle of (x,y) to be "beta". As a reminder, we want to rotate by a different angle, called "theta".

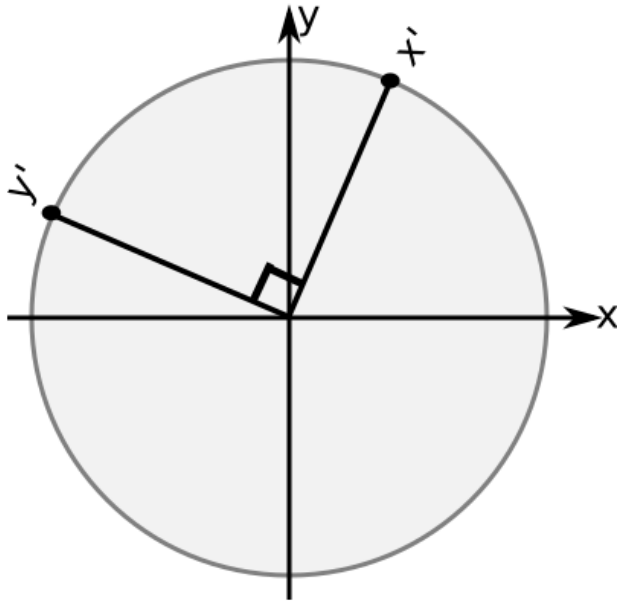


Before we can rotate by "theta", first we need to be able to rotate by 90 degrees, or pi/2. So to rotate (cos(beta), sin(beta)) by pi/2, we get (cos(beta+pi/2), sin(beta+pi/2)).

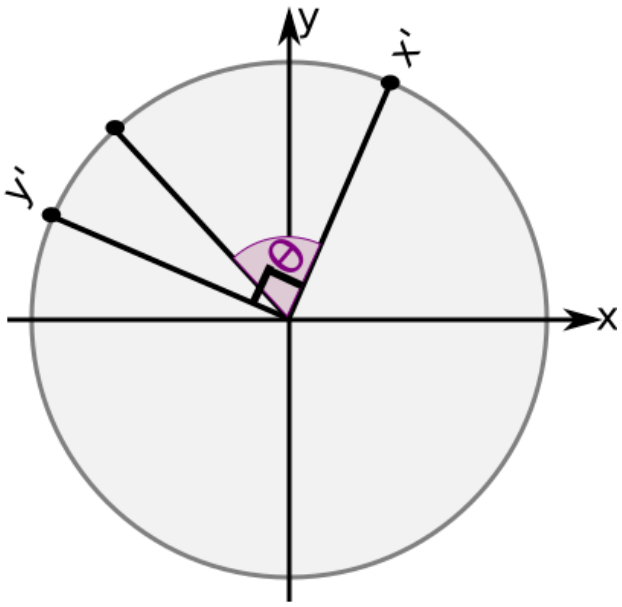
Now let's give each of those vertices new names, x' and y', for the purpose of ignoring details for now that we'll return to later, just let we did for length "r" above.

Now forget about "beta", and remember that our goal is to rotate by angle "theta". Look at the picture below, while turning your head slightly to the left. x' and y' look just like our normal Cartesian plane and unit circle, combined with

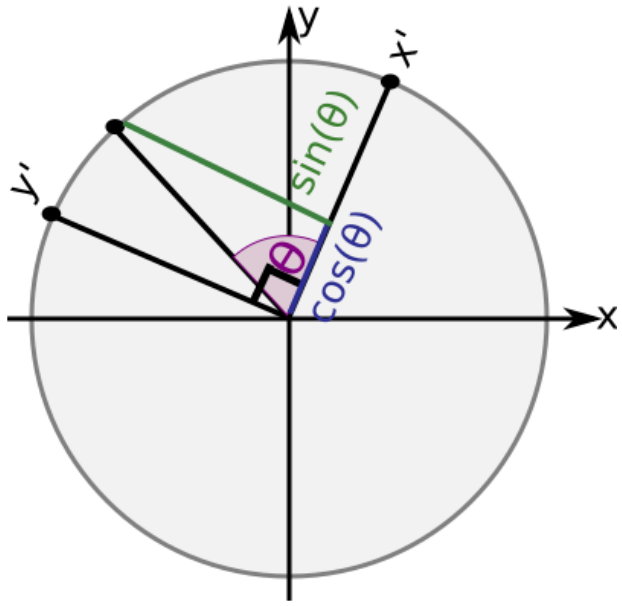

y
[cos(β)
sin(β)]
[cos(β+π/2)
sin(β+π/2)]
x


y
x'
y'
x

the “theta”; it looks like what we already know from high school geometry.



So with this new frame of reference, we can rotate x’ by “theta”, and draw a right triangle on the unit circle using this new frame of reference.

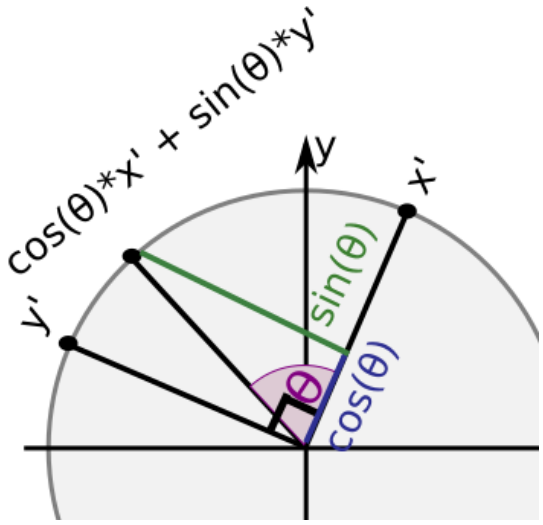


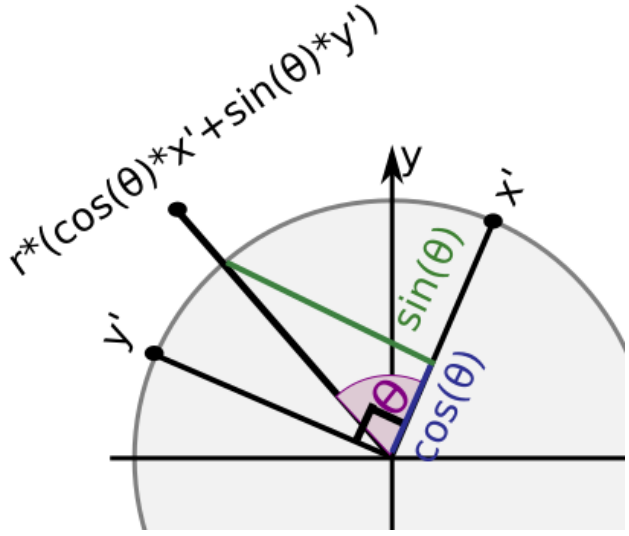
So the rotated point can be constructed by the following

$$\vec{r}(\vec{a}; theta) = cos(\theta) * \vec{x'} + sin(\theta) * \vec{y'}$$

Now that we’ve found the direction on the unit circle, we remember to make it length “r”.

Ok, we are now going to stop thinking about geometry, and we will only be thinking about algebra. Please don’t try to look at the formula and try to draw any diagrams.


cos(θ)*x' + sin(θ)*y'
y
x'
y'
Sin(θ)
θ
cos(θ)


r*(cos(θ)*x'+sin(θ)*y')
y
x'
y'
Sin(θ)
θ
cos(θ)

Ok, now it is time to remember what the values that x' and y' are defined as.

$$\begin{aligned}\vec{r}(\vec{a};\theta) &= r * (cos(\theta) * \vec{x'} + sin(\theta) * \vec{y'}) \\ &= r * (cos(\theta) * \begin{bmatrix} cos(\beta) \\ sin(\beta) \end{bmatrix} + sin(\theta) * \begin{bmatrix} cos(\beta + \pi/2) \\ sin(\beta + \pi/2) \end{bmatrix})\end{aligned}$$

A problem we have now is how to calculate cosine and sine of beta + pi/2, because we haven't actually calculated beta; we've calculated sine and cosine of beta by dividing the x value by the magnitude of the a, and the sine of beta by dividing the y value by the magnitude of a.

$$\begin{aligned}cos(\theta) &= \vec{a}_x/r \\ sine(\theta) &= \vec{a}_y/r\end{aligned}$$

We could try to take the inverse sine or inverse cosine of theta, but there is no need given properties of trigonometry.

$$\begin{aligned}cos(\theta + \pi/2) &= -sin(\theta) \\ sin(\theta + \pi/2) &= cos(\theta)\end{aligned}$$

Therefore

$$\begin{aligned}cos(\theta) &= \vec{a}_x/r \\ sin(\theta) &= \vec{a}_y/r\end{aligned}$$

$$\begin{aligned}\vec{r}(\vec{a};\theta) &= r * (cos(\theta) * \vec{x'} + sin(\theta) * \vec{y'}) \\ &= r * (cos(\theta) * \begin{bmatrix} cos(\beta) \\ sin(\beta) \end{bmatrix} + sin(\theta) * \begin{bmatrix} cos(\beta + \pi/2) \\ sin(\beta + \pi/2) \end{bmatrix}) \\ &= r * (cos(\theta) * \begin{bmatrix} \vec{a}_x/r \\ \vec{a}_y/r \end{bmatrix} + sin(\theta) * \begin{bmatrix} -\vec{a}_y/r \\ \vec{a}_x/r \end{bmatrix}) \\ &= (cos(\theta) * \begin{bmatrix} \vec{a}_x \\ \vec{a}_y \end{bmatrix} + sin(\theta) * \begin{bmatrix} -\vec{a}_y \\ \vec{a}_x \end{bmatrix}) \\ &= (cos(\theta) * \vec{a}_x + sin(\theta) * \begin{bmatrix} -\vec{a}_y \\ \vec{a}_x \end{bmatrix}) \\ \vec{r}(\vec{a};\theta) &= \begin{bmatrix} -\vec{a}_y \\ \vec{a}_x \end{bmatrix} \text{ if } (\theta = \pi/2) \\ &= (cos(\theta) * \vec{a} + sin(\theta) * \vec{r}(\vec{a};\pi/2) \text{ if } (\theta \neq \pi/2)\end{aligned}$$

CHAPTER

NINE

ROTATION FIX ATTEMPT 1 - DEMO 08

9.1 Purpose

Fix the rotation problem from the previous demo in a seemingly intuitive way, but do it inelegantly.

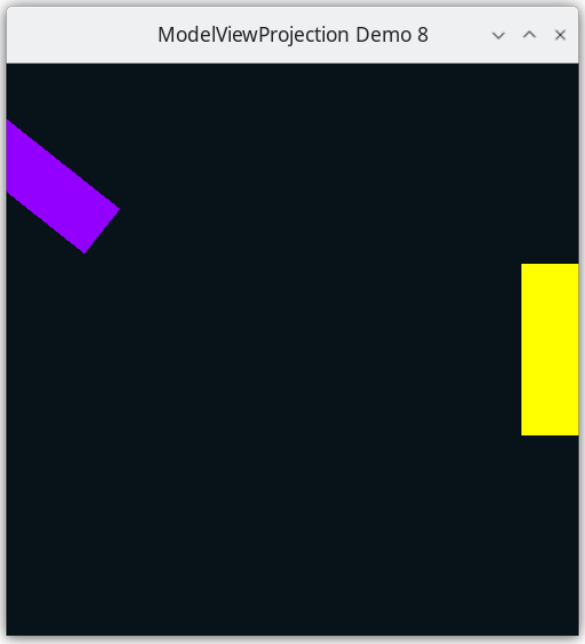


Fig. 1: Demo 08

9.2 How to Execute

Load src/demo08/demo.py in Spyder and hit the play button

9.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation

9.4 Description

The problem in the last demo is that all rotations happen relative to World Space's (0,0) and axes. By translating our paddles to their position before rotating, they are rotated around World Space's origin, instead of being rotated around their modelspace's center.

In this demo, we try to solve the problem by making a method to rotate around a given point in world space, in this case, the paddle's center.

Listing 1: src/demo08/demo.py

```
109 class Vertex:
```

Listing 2: src/demo08/demo.py

```
145     def rotate_around(self: Vertex, angle_in_radians: float, center: Vertex) -> Vertex:
146         translate_to_center: Vertex = self.translate(-center)
147         rotated_around_origin: Vertex = translate_to_center.rotate(angle_in_radians)
148         back_to_position: Vertex = rotated_around_origin.translate(center)
149         return back_to_position
150
```

Within the event loop, this seems quite reasonable

Listing 3: src/demo08/demo.py

```
218 while not glfw.window_should_close(window):
```

Listing 4: src/demo08/demo.py

```
238     glColor3f(paddle1.r, paddle1.g, paddle1.b)
239
240     glBegin(GL_QUADS)
241     rotatePoint: Vertex = paddle1.position
242     for paddle1_vertex_ms in paddle1.vertices:
243         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.translate(paddle1.position)
244         paddle1_vertex_ws: Vertex = paddle1_vertex_ws.rotate_around(
245             paddle1.rotation, rotatePoint
246         )
```

(continues on next page)

(continued from previous page)

```
        paddle1_vertex_ndc: Vertex = paddle1_vertex_ws.uniform_scale(scalar=1.0 / 10.0)
        glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
```

Listing 5: src/demo08/demo.py

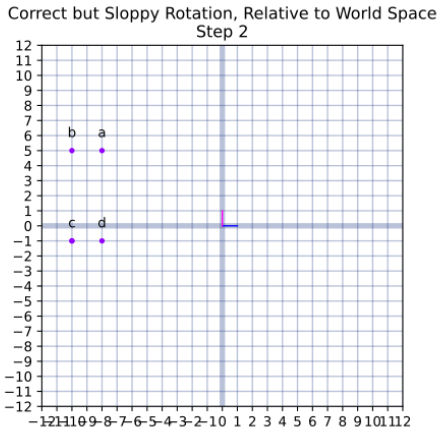
```
    # draw paddle 2
    glColor3f(paddle2.r, paddle2.g, paddle2.b)

    glBegin(GL_QUADS)
    rotatePoint: Vertex = paddle2.position
    for paddle2_vertex_ms in paddle2.vertices:
        paddle2_vertex_ws: Vertex = paddle2_vertex_ms.translate(paddle2.position)
        paddle2_vertex_ws: Vertex = paddle2_vertex_ws.rotate_around(
            paddle2.rotation, rotatePoint
        )
        paddle2_vertex_ndc: Vertex = paddle2_vertex_ws.uniform_scale(scalar=1.0 / 10.0)
        glVertex2f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y)
    glEnd()
```

All we did was add a rotate around method, and call it, with the paddle's center as the rotate point.

Although this works for now and looks like decent code, this is extremely sloppy, and not thought out well at all. We apply a transformation from paddle space to world space, then do the inverse, then rotate, and then do the first transformation from paddle space to world space again.

The images of the transformation sequence below should show how brain-dead it is, and the Cayley graph is gross.

But from this we will learn something important.



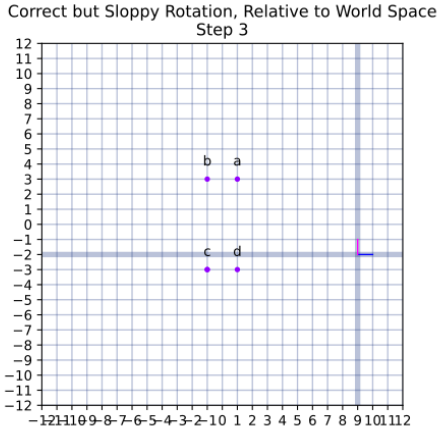
translating back to the origin

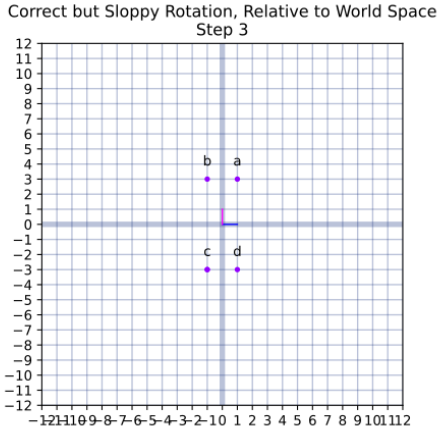
resetting the coordinate system

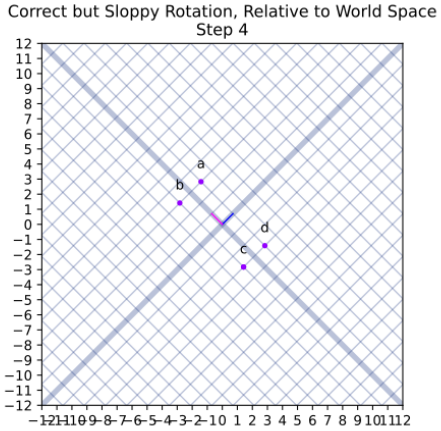
rotating

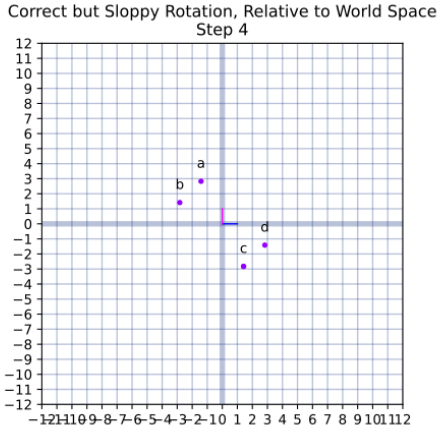
resetting the coordinate system

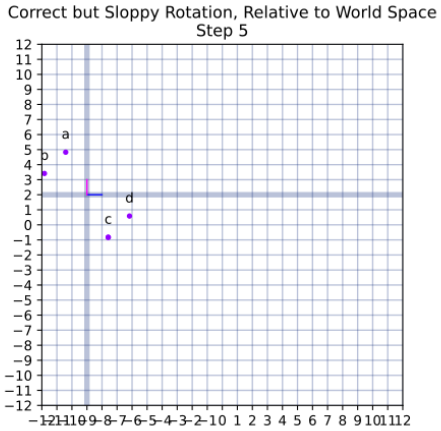
and them translating them back to the paddle space origin


Correct but Sloppy Rotation, Relative to World Space
Step 3
b
a
c
d


Correct but Sloppy Rotation, Relative to World Space
Step 3
b
a
c
d

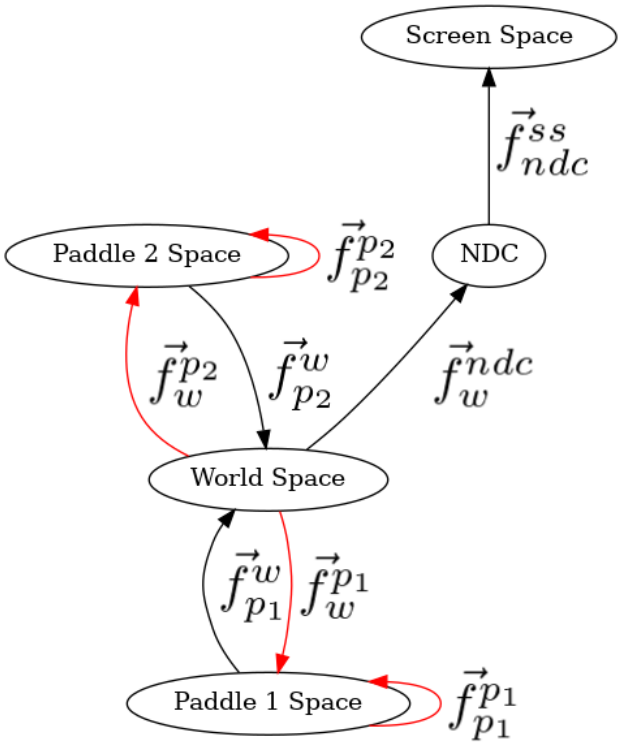

Correct but Sloppy Rotation, Relative to World Space
Step 4
a
b
d
c


Correct but Sloppy Rotation, Relative to World Space
Step 4
a
b
c
d


Correct but Sloppy Rotation, Relative to World Space
Step 5
a
b
c
d

9.5 Cayley Graph

Note, this is gross, and the edge from the paddlespace to itself doesn't even make any sense, but the author did not know how else to represent this code.



CHAPTER

TEN

ROTATION FIXED - SEQUENCE OF TRANSFORMATIONS - DEMO 09

10.1 Purpose

Make the rotations work correctly by thinking about the problem more clearly.

We do this by the following:

- Read sequence of transformations from modelspace to worldspace in the reverse order, from bottom to top.
- Each subsequent transformation is relative to the current local coordinate system.

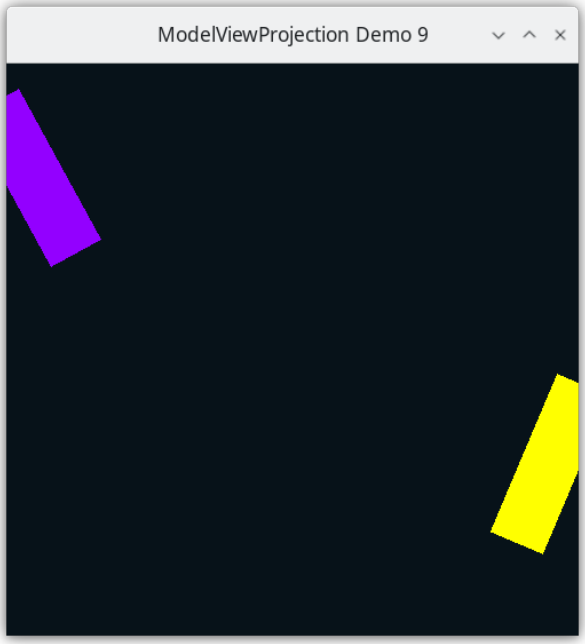


Fig. 1: Demo 09

10.2 How to Execute

Load src/demo09/demo.py in Spyder and hit the play button

10.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation

10.4 Description

In Demo 07 we tried to represent the translation and then rotation around the paddle's center by code that worked as follows

$$\vec{f}_p^w = \vec{r} \circ \vec{t}$$

where t and r are the translate and rotate functions for a paddle, and we read the functions from right to left. (In code, we were reading the method chaining from top-down, as that is the order in which the function calls are evaluated.)

But that caused erratic behavior of the paddle over time, as it was not rotating around its own origin. It was not working.

So in Demo 08 we fixed it by writing a "rotate_around" method, in which we translated the paddle to its position, then in order to rotate around its center, we re-centered the paddle, did the rotation, and then re-translated to the paddle's position.

$$\begin{aligned}
\vec{f}_p^w &= (\vec{t} \circ \vec{r} \circ \vec{t}^{-1}) \circ \vec{t} \\
&= \vec{t} \circ \vec{r} \circ \vec{t}^{-1} \circ \vec{t} \\
&= \vec{t} \circ \vec{r} \circ (\vec{t}^{-1} \circ \vec{t}) \\
&= \vec{t} \circ \vec{r}
\end{aligned}$$

So translate and inverse translate cancel out, and demonstrate to us that the rotate function actually needs to be applied first, then the translate function. Which means that we were actually reading the function applications in the wrong order! Intuition can lead us astray!

To understand why the code in this demo works, you can think about it in one of two ways, forwards or backwards.

10.4.1 Reading transformations top-down

When reading a composition of function calls top down, all the transformations happen relative to the global origin and axes, i.e. the natural *basis*. After each function call, the basis is set back to the origin basis.

Listing 1: src/demo09/demo.py

```
232         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate(
233             paddle1.rotation
234         ).translate(paddle1.position)
```

10.4.2 Reading transformations bottom-up

Alternatively, you can read the transformations backwards, where the operations all cumulatively modify the current axes and origin, i.e. basis, to a new basis. All subsequent functions move that relative basis to a new relative basis.

Listing 2: src/demo09/demo.py

```
232         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate(
233             paddle1.rotation
234         ).translate(paddle1.position)
```

10.4.3 Suggested Order

The author has a strong opinion on the matter for computer graphics.

Start at world space node. As a reminder, this is the top-level coordinate system that we choose to use. It is whatever you want it to be.

Think of world space's origin, x and y axes. On the Cayley graph, move towards each modelspace node, which is against the direction of the edge, Look at that edge's corresponding sequence of function calls in the graphics code. Read the transformations from the bottom up, imagining a transformed local coordinate system against which each function in the sequence operates.

The data you plot that is in its own modelspace will actually be plotted relative to that transformed coordinate system.

For the linear-algebra inclined reader, we've used

$$L * (M * \vec{x}) = (L * M) * \vec{x}$$

to understand a complicated sequence of matrix multiplications, modifying L by a right-multiplication of M.

Listing 3: src/demo09/demo.py

```
232         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate(
233             paddle1.rotation
234         ).translate(paddle1.position)
```

Now go back to world space. Think of all modelspace data as having already been "plotted". So now follow the edges to screen space, reading the transformations from top down, and resetting the coordinate system after every transformation.

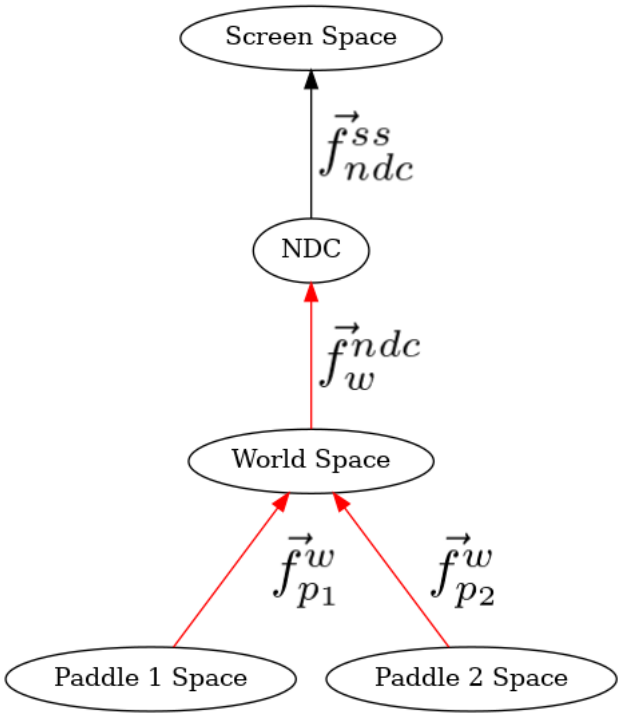


Fig. 2: Demo 06

Listing 4: src/demo09/demo.py

```
237         paddle1_vertex_ndc: Vertex = paddle1_vertex_ws.uniform_scale(scalar=1.0 / 10.0)
```

As a side note, the computer has no notion of coordinate systems. It just has a sequence of procedures to apply on a sequence of numbers. For us humans to make sense of what we are modeling, and to implement it correctly, we need to be able to reason and compose our thoughts clearly.

10.5 Code

10.5.1 The Event Loop

Listing 5: src/demo09/demo.py

```
207 while not glfw.window_should_close(window):
```

Listing 6: src/demo09/demo.py

```
227     glColor3f(paddle1.r, paddle1.g, paddle1.b)
228
229     glBegin(GL_QUADS)
230     for paddle1_vertex_ms in paddle1.vertices:
231         # doc-region-begin paddle 1 transformations
232         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate(
```

(continues on next page)

(continued from previous page)

```
            paddle1.rotation
        ).translate(paddle1.position)
        # doc-region-end paddle 1 transformations
        # doc-region-begin paddle 1 scale
        paddle1_vertex_ndc: Vertex = paddle1_vertex_ws.uniform_scale(scalar=1.0 / 10.0)
        # doc-region-end paddle 1 scale
        glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
    glEnd()
```

- Read the modelspace to world space, starting from the bottom, up. Translate, then rotate
- Reset the coordinate system
- Read the worldspace to NDC

Listing 7: src/demo09/demo.py

```
    glColor3f(paddle2.r, paddle2.g, paddle2.b)

    glBegin(GL_QUADS)
    for paddle2_vertex_ms in paddle2.vertices:
        paddle2_vertex_ws: Vertex = paddle2_vertex_ms.rotate(
            paddle2.rotation
        ).translate(paddle2.position)
        paddle2_vertex_ndc: Vertex = paddle2_vertex_ws.uniform_scale(scalar=1.0 / 10.0)
        glVertex2f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y)
    glEnd()
```

- Read the modelspace to world space, starting from the bottom, up. Translate, then rotate
- Reset the coordinate system
- Read the worldspace to NDC

CHAPTER

ELEVEN

CAMERA SPACE - DEMO 10

11.1 Purpose

Graphical programs in which the viewer never “moves” are boring. Model a virtual “camera”, and let the user move the camera around in the scene. In this picture, notice that the purple paddle is no longer on the left side of the screen, but towards the right, horizontally.

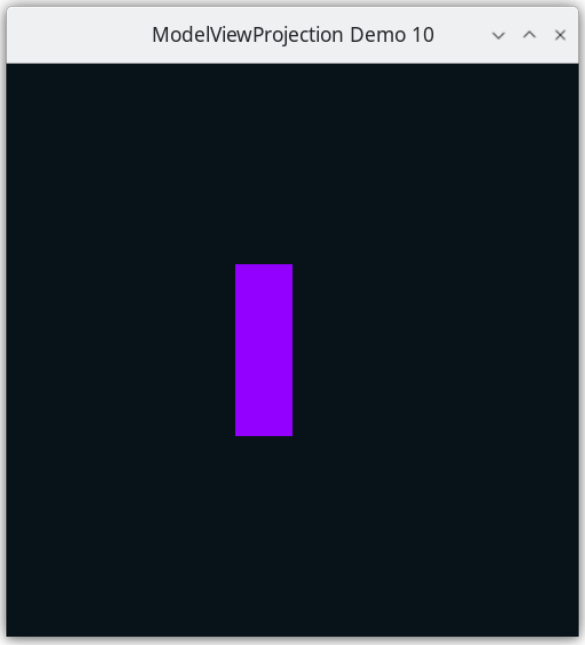


Fig. 1: Demo 10

11.2 How to Execute

Load src/demo10/demo.py in Spyder and hit the play button

11.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left

11.4 Description

"Camera space" means the coordinates of everything relative to a camera, not relative to world space.

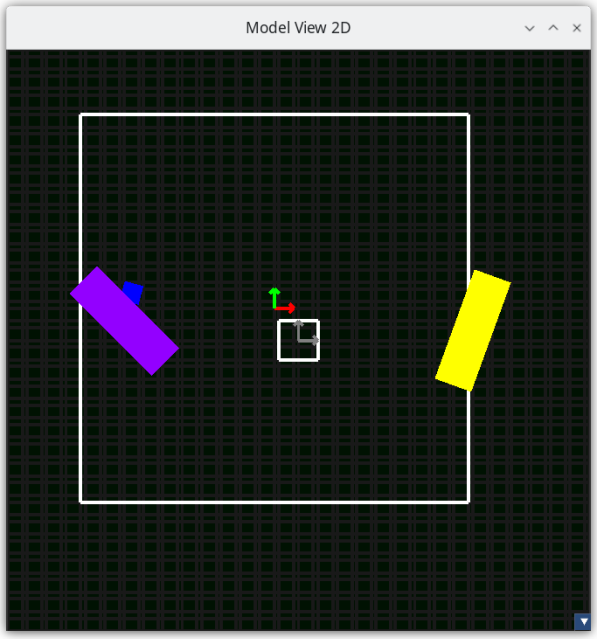


Fig. 2: Camera space

In the picture above, NDC is the square in the center of the screen, and its coordinate system is defined by its two axes X and Y. The square to the to the left and up is the position of the camera; which has its own X axis and Y axis.

The coordinates of the paddle can be described in world space, or in camera space, and that we can create functions to convert coordinates between the two spaces.

Towards that, in the Cayley graph, camera space will be between world space and NDC, but which way should the direction of the edge be?

It could be

- From world space to camera space.

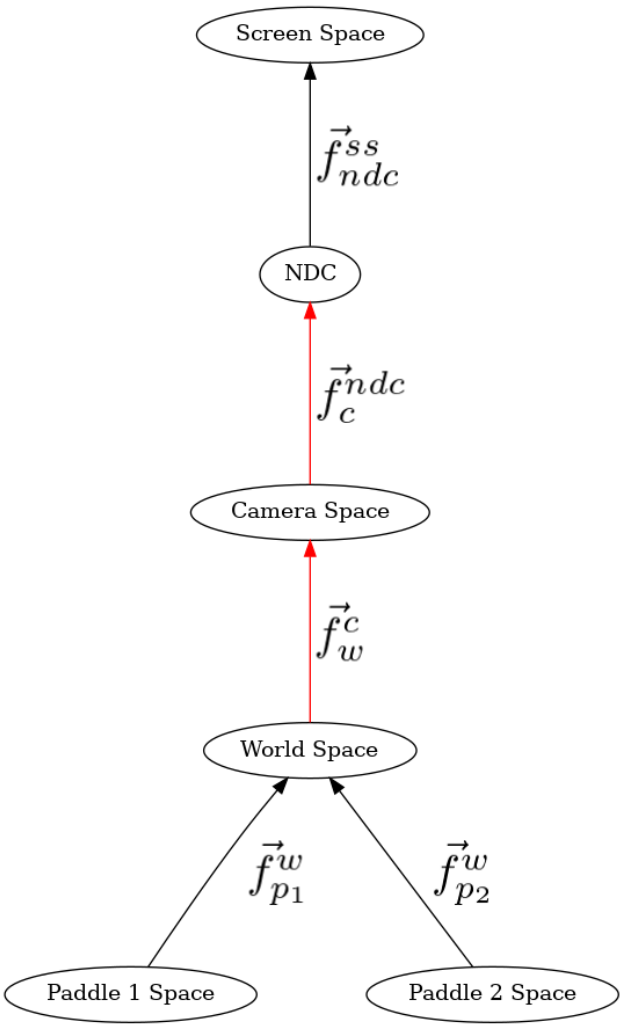


Fig. 3: Demo 10

- From camera space to world space.

Since the camera's position will be described relative to world space in world space coordinates, just like the paddles are, it makes more sense to use the latter, in which the directed edge goes from camera space to world space.

Think about it this way. Imagine you a driving. Are you staying in a constant position, and the wheels of your car rotate the entire earth and solar system around yourself? That's one way to look at it, and the earth revolves around me too, but the mathematical descriptions of everything else that's not you breaks down.

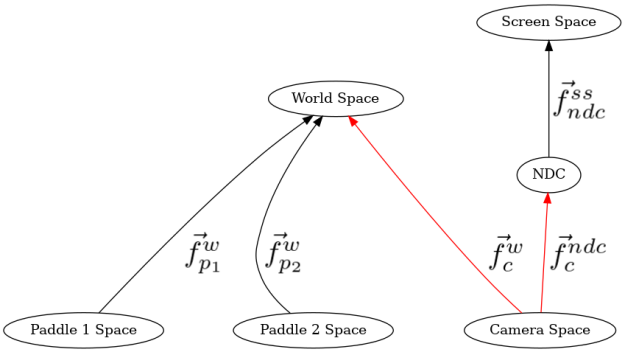


Fig. 4: Demo 10

The math is easier in the aggregate if you describe your position as moving relative to earth.

But this introduces a new problem. Follow from the paddle's modelspace to screen space. Up until this demo, we've always been following the the direction of each edges. In our new Cayley graph, when going from world space to camera space, we're moving in the opposite direction of the edge.

Going against the direction of an edge in a Cayley Graph means that we don't apply the function itself; instead we apply the inverse of the function. This concept comes from Group Theory in Abstract Algebra, the details of which won't be discussed here, but we will use it to help us reason about the transformations.

11.4.1 Inverses

Inverse Of Translate

The inverse of

```
>>> v.translate(x,y)
```

is

```
>>> v.translate(-x,-y)
```

Inverse Of Rotate

The inverse of

```
>>> v.rotate(theta)
```

is

```
>>> v.rotate(-theta)
```

Inverse Of Scale

The inverse of

```
>>> v.scale(x,y)
```

is

```
>>> v.scale(1.0/x,1.0/y)
```

Inverse Of Sequence Of Functions

The inverse of a sequence of functions is the inverse of each function, applied in reverse order.

For the linear-algebra inclined reader,

$$(A * B)^{-1} = B^{-1} * A^{-1}$$

The inverse of

```
>>> v.scale(x,y).translate(Vertex(x,y))
```

is

```
>>> v.translate(-Vertex(x,y).scale(1.0/x,1.0/y)
```

Think of the inverses this way. Stand up. Make sure that nothing is close around you that you could trip on or that you would walk into. Look forward. Keeping your eyes straight ahead, sidestep a few steps to the left. From your perspective, the objects in your room all moved to the right. Now, rotate your head to your right, keeping your eyes looking directly in front of your head. Which way did the room move? Towards the left.

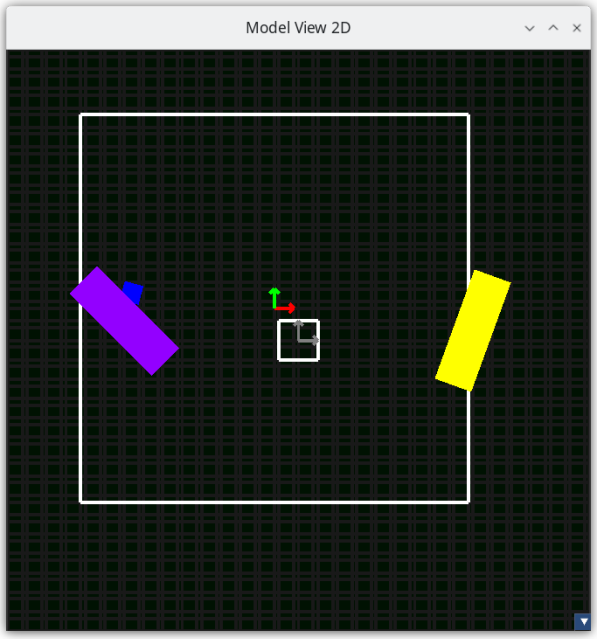


Fig. 5: Camera is 3D space

Looking at the graph paper on the ground there, that's world space. The camera gets placed in world space just like any other modelspace data. In order to render from the camera's point of view, we need to move and orient the -1.0 to 1.0 box relative to the camera to match the -1.0 to 1.0 box in world space.

Run "python mvpVisualization/modelviewperspectiveprojection/modelviewperspectiveprojection.py", and follow along with the Cayley graph. The camera will be placed like any other modelspace data, but then the inverse of the transformations will be applied to all of the vertices.

11.4.2 For the attentive reader

The purpose of watching that animation was to see how the camera was placed just like the modelspace data was placed, while not modifying the modelspace data, but then the inverse was applied, moving the modelspace data.

The attentive reader may notice: "Bill, you said the the inverse causes the operations to be run in opposite order, yet I saw them run in the same order. For camera placement, I saw translate, then rotate sideways, then rotate up. For the inverse, I saw inverse translate, then inverse rotate sideways, then inverse rotate up. What gives?"

Well, the answer, which we talk about later in more detail, is that we change the order in which we read the sequence of transformations.

Let's say we read the following from left to right

$$\vec{f} = \vec{a} \circ \vec{b} \circ \vec{c}$$

So we imagine the a transformation, then the b transformation, then the c.

To take the inverse, we reverse the order of the transformations, and invert them.

$$\vec{f}^{-1} = \vec{c}^{-1} \circ \vec{b}^{-1} \circ \vec{a}^{-1}$$

But now we read the transformations from right to left.

So we imagine the a inverse transformation, then the b inverse transformation, then the c inverse.

The author understands that this may be confusing to the reader, and seems illogical. Camera placement is, in the author's opinion, the most difficult concept for a novice to grasp, and the author is unaware of a better explanation. Bear with the author, by the end of this book, it should make more sense to the reader.

11.5 Code

Listing 1: src/demo10/demo.py

```
180
181
182 @dataclass
183 class Camera:
184     position_ws: Vertex = field(default_factory=lambda: Vertex(x=0.0, y=0.0))
```

Listing 2: src/demo10/demo.py

```
192 def handle_inputs() -> None:
193     global camera
194
195     if glfw.get_key(window, glfw.KEY_UP) == glfw.PRESS:
```

(continues on next page)

(continued from previous page)

```
196         camera.position_ws.y += 1.0
197     if glfw.get_key(window, glfw.KEY_DOWN) == glfw.PRESS:
198         camera.position_ws.y -= 1.0
199     if glfw.get_key(window, glfw.KEY_LEFT) == glfw.PRESS:
200         camera.position_ws.x -= 1.0
201     if glfw.get_key(window, glfw.KEY_RIGHT) == glfw.PRESS:
202         camera.position_ws.x += 1.0
```

```
...
```

11.5.1 The Event Loop

Listing 3: src/demo10/demo.py

```
230 while not glfw.window_should_close(window):
```

Listing 4: src/demo10/demo.py

```
249     glColor3f(paddle1.r, paddle1.g, paddle1.b)
250
251     glBegin(GL_QUADS)
252     for paddle1_vertex_ms in paddle1.vertices:
253         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate(
254             paddle1.rotation
255         ).translate(translate_amount=paddle1.position)
```

The camera's position is at camera.x, camera.y. So apply the inverse of the camera's transformations to paddle 1's vertices, i.e. translate with the negative values.

Listing 5: src/demo10/demo.py

```
258         paddle1_vertex_cs: Vertex = paddle1_vertex_ws.translate(
259             translate_amount=-camera.position_ws
260         )
```

Listing 6: src/demo10/demo.py

```
263         paddle1_vertex_ndc: Vertex = paddle1_vertex_cs.uniform_scale(scalar=1.0 / 10.0)
264         glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
265     glEnd()
```

Listing 7: src/demo10/demo.py

```
269     glColor3f(paddle2.r, paddle2.g, paddle2.b)
270
271     glBegin(GL_QUADS)
272     for paddle2_vertex_ms in paddle2.vertices:
273         paddle2_vertex_ws: Vertex = paddle2_vertex_ms.rotate(
274             paddle2.rotation
275         ).translate(paddle2.position)
```

The camera's position is at camera.x, camera.y. So apply the inverse of the camera's transformations to paddle 2's vertices, i.e. translate with the negative values.

Listing 8: src/demo10/demo.py

```
278         paddle2_vertex_cs: Vertex = paddle2_vertex_ws.translate(-camera.position_ws)
```

Listing 9: src/demo10/demo.py

```
282         glVertex2f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y)
283     glEnd()
```

CHAPTER

TWELVE

RELATIVE OBJECTS - DEMO 11

12.1 Purpose

Introduce relative objects, by making a small blue square that is defined relative to the left paddle, but offset some in the x direction. When the paddle on the left moves or rotates, the blue square moves with it, because it is defined relative to it.

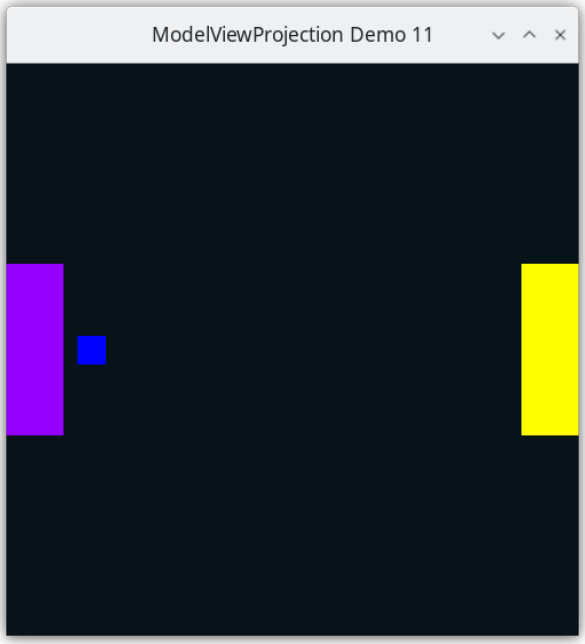


Fig. 1: Demo 11

12.2 How to Execute

Load src/demo11/demo.py in Spyder and hit the play button

12.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left

12.4 Description

12.5 Cayley Graph

In the graph below, all we have added is "Square space", relative to paddle 1 space.

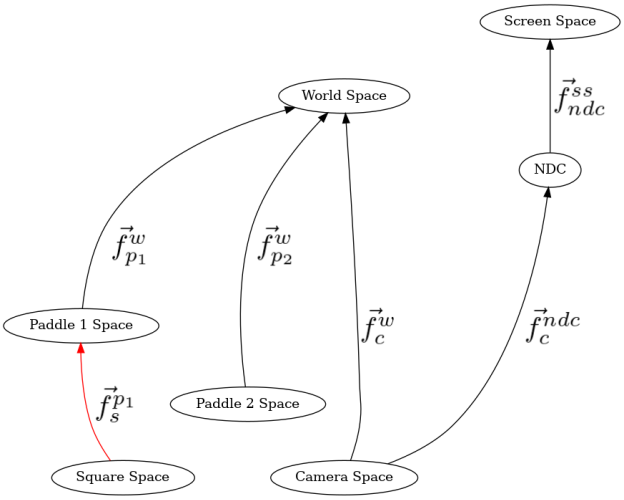


Fig. 2: Demo 11

In the picture below, in 3D space, we see that the square has its own modelspace (as evidenced by the 3 arrows), and we are going to define its position and orientation relative to paddle 1.

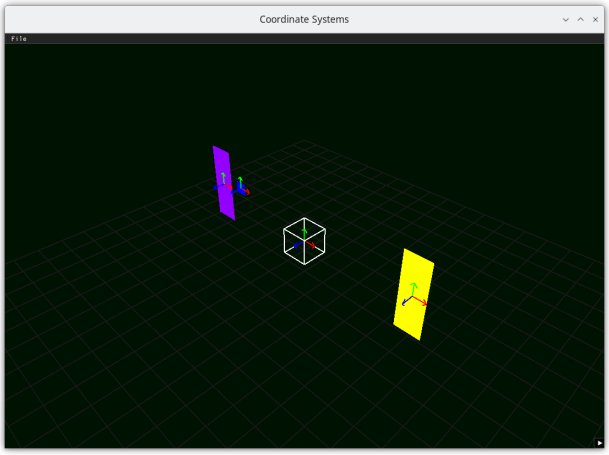


Fig. 3: Coordinate Frames

12.6 Code

Define the geometry of the square in its own modelspace.

Listing 1: src/demo11/demo.py

```
187 square: list[Vertex] = [
188     Vertex(x=-0.5, y=-0.5),
189     Vertex(x=0.5, y=-0.5),
190     Vertex(x=0.5, y=0.5),
191     Vertex(x=-0.5, y=0.5),
192 ]
```

12.6.1 Event Loop

Listing 2: src/demo11/demo.py

```
235 while not glfw.window_should_close(window):
```

```
    ...
```

Draw paddle 1, just as before.

Listing 3: src/demo11/demo.py

```
255     glColor3f(paddle1.r, paddle1.g, paddle1.b)
256
257     glBegin(GL_QUADS)
258     for paddle1_vertex_ms in paddle1.vertices:
259         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate(
260             paddle1.rotation
261         ).translate(paddle1.position)
262         paddle1_vertex_cs: Vertex = paddle1_vertex_ws.translate(-camera.position_ws)
```

(continues on next page)

(continued from previous page)

```
        paddle1_vertex_ndc: Vertex = paddle1_vertex_cs.uniform_scale(scalar=1.0 / 10.0)
        glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
    glEnd()
```

As a refresher, the author recommends reading the code from modelspace to worldspace from the bottom up, and from worldspace to NDC from top down.

- Read from modelspace to world space, bottom up
- Reset the coordinate system
- Read from world space to camera space, knowing that camera transformations are implemented as the inverse of placing the camera space in world space.
- Reset the coordinate system
- Read camera-space to NDC

New part! Draw the square relative to the first paddle! Translate the square to the right by 2 units. We are dealing with a -1 to 1 world space, which later gets scaled down to NDC.

Listing 4: src/demo11/demo.py

```
    glColor3f(0.0, 0.0, 1.0)
    glBegin(GL_QUADS)
    for model_space in square:
        paddle1space: Vertex = model_space.translate(Vertex(x=2.0, y=0.0))
        world_space: Vertex = paddle1space.rotate(paddle1.rotation).translate(
            paddle1.position
        )
        camera_space: Vertex = world_space.translate(-camera.position_ws)
        ndc: Vertex = camera_space.uniform_scale(scalar=1.0 / 10.0)
        glVertex2f(ndc.x, ndc.y)
    glEnd()
```

Towards that, we need to do all of the transformations to the square that we would to the paddle, and then do any extra transformations afterwards.

As such, read

- Read paddle1space to world space, from bottom up

If we were to plot the square now, it would be in paddle 1's space. We don't want that, we want in to be moved in the X direction some units. Therefore

- Read modelspace to paddle1space, from bottom up
- Reset the coordinate system.

Now the square's geometry will be in its own space!

- Read from worldspace to camera-space, knowing that camera transformations are implemented as the inverse of placing the camera space in world space.
- Reset the coordinate system
- Read camera-space to NDC

Draw paddle 2 just like before.

Listing 5: src/demo11/demo.py

```
283     glColor3f(paddle2.r, paddle2.g, paddle2.b)
284
285     glBegin(GL_QUADS)
286     for paddle2_vertex_ms in paddle2.vertices:
287         paddle2_vertex_ws: Vertex = paddle2_vertex_ms.rotate(
288             paddle2.rotation
289         ).translate(paddle2.position)
290         paddle2_vertex_cs: Vertex = paddle2_vertex_ws.translate(-camera.position_ws)
291         paddle2_vertex_ndc: Vertex = paddle2_vertex_cs.uniform_scale(1.0 / 10.0)
292         glVertex2f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y)
293     glEnd()
```

CHAPTER

THIRTEEN

ROTATE THE SQUARE - DEMO 12

13.1 Purpose

Rotate the square around its origin.

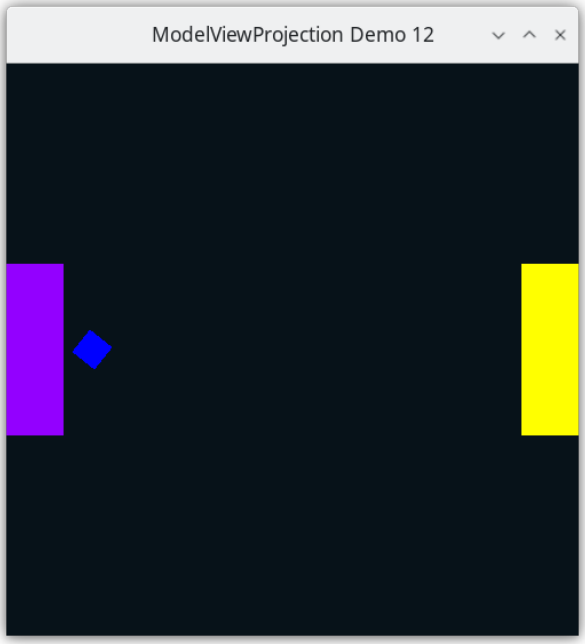


Fig. 1: Demo 12

13.2 How to Execute

Load src/demo12/demo.py in Spyder and hit the play button

13.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center

13.4 Description

13.5 Cayley Graph

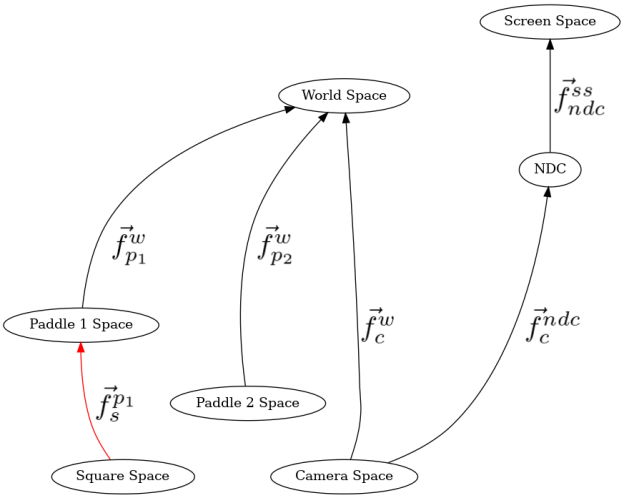


Fig. 2: Demo 12

13.6 Code

Make a variable to determine the angle that the square will be rotated.

Listing 1: src/demo12/demo.py

```
196 square_rotation: float = 0.0
```

When 'q' is pressed, increase the angle.

Listing 2: src/demo12/demo.py

```
201 def handle_inputs() -> None:
202     global square_rotation
203     if glfw.get_key(window, glfw.KEY_Q) == glfw.PRESS:
204         square_rotation += 0.1
```

```
...
```

13.6.1 Event Loop

In the previous chapter, this was the rendering code for the square.

Listing 3: src/demo11/demo.py

```
269     glColor3f(0.0, 0.0, 1.0)
270     glBegin(GL_QUADS)
271     for model_space in square:
272         paddle1space: Vertex = model_space.translate(Vertex(x=2.0, y=0.0))
273         world_space: Vertex = paddle1space.rotate(paddle1.rotation).translate(
274             paddle1.position
275         )
276         camera_space: Vertex = world_space.translate(-camera.position_ws)
277         ndc: Vertex = camera_space.uniform_scale(scalar=1.0 / 10.0)
278         glVertex2f(ndc.x, ndc.y)
279     glEnd()
```

Since we just want to add one rotation at the end of the sequence of transformations from paddle 1 space to square space, just add a rotate call at the top.

Listing 4: src/demo12/demo.py

```
272     glColor3f(0.0, 0.0, 1.0)
273     glBegin(GL_QUADS)
274     for square_vertex_ms in square:
275         paddle_1_space: Vertex = square_vertex_ms.rotate(square_rotation).translate(
276             Vertex(x=2.0, y=0.0)
277         )
278         world_space: Vertex = paddle_1_space.rotate(paddle1.rotation).translate(
279             paddle1.position
280         )
281         camera_space: Vertex = world_space.translate(-camera.position_ws)
```

(continues on next page)

(continued from previous page)

```
282         ndc: Vertex = camera_space.uniform_scale(scalar=1.0 / 10.0)
283         glVertex2f(ndc.x, ndc.y)
284     glEnd()
```

The author is getting really tired of having to look at all the different transformation functions repeatedly defined for each object being drawn.

CHAPTER

FOURTEEN

ROTATE THE SQUARE AROUND PADDLE 1 - DEMO 13

14.1 Purpose

Rotate the square around paddle1's center.

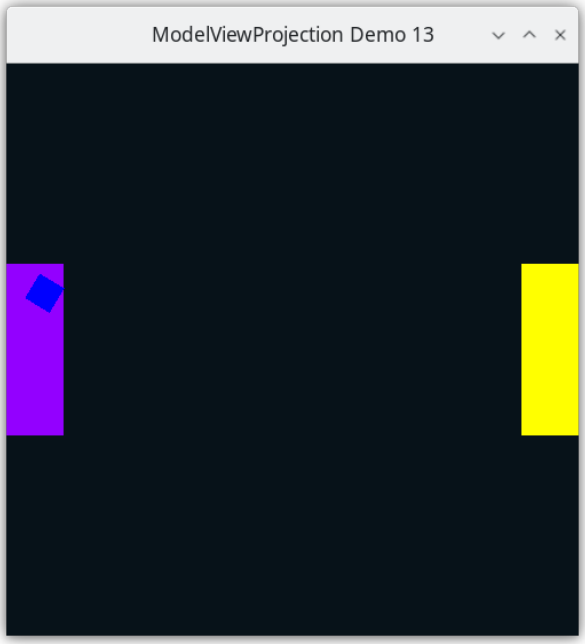


Fig. 1: Demo 13

14.2 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

14.3 Description

14.4 Cayley Graph

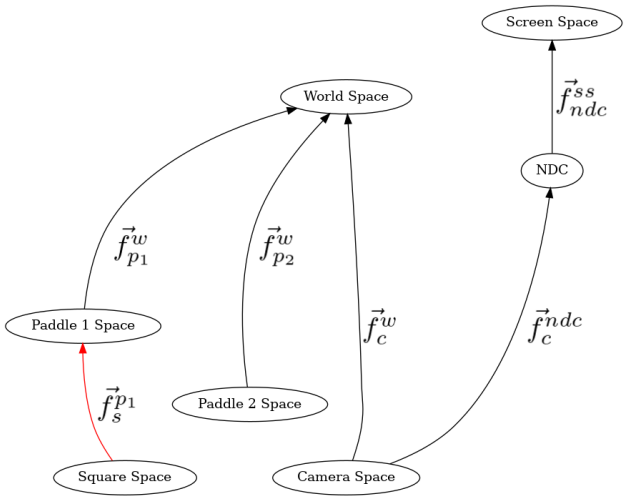


Fig. 2: Demo 13

Listing 1: src/demo13/demo.py

```
196 rotation_around_paddle1: float = 0.0
```

14.5 Event Loop

Listing 2: src/demo13/demo.py

```
246 while not glfw.window_should_close(window):
```

```
    ...
```

Listing 3: src/demo13/demo.py

```
277     glColor3f(0.0, 0.0, 1.0)
278     glBegin(GL_QUADS)
279     for square_vertex_ms in square:
280         paddle_1_space: Vertex = (
281             square_vertex_ms.rotate(square_rotation)
282             .translate(Vertex(x=2.0, y=0.0))
283             .rotate(rotation_around_paddle1)
284         )
285         world_space: Vertex = paddle_1_space.rotate(paddle1.rotation).translate(
286             paddle1.position
287         )
288         camera_space: Vertex = world_space.translate(-camera.position_ws)
289         ndc: Vertex = camera_space.uniform_scale(scalar=1.0 / 10.0)
290         glVertex2f(ndc.x, ndc.y)
291     glEnd()
```

CHAPTER

FIFTEEN

ADDING DEPTH - Z AXIS DEMO 14

15.1 Purpose

Do the same stuff as the previous demo, but use 3D coordinates, where the negative z axis goes into the screen (because of the right hand rule). Positive z comes out of the monitor towards your face.

Things that this demo doesn't end up doing correctly:

- The blue square is always drawn, even when its z-coordinate in world space is less than the paddle's. The solution will be z-buffering https://en.wikipedia.org/wiki/Z-buffering, and it is implemented in the next demo.

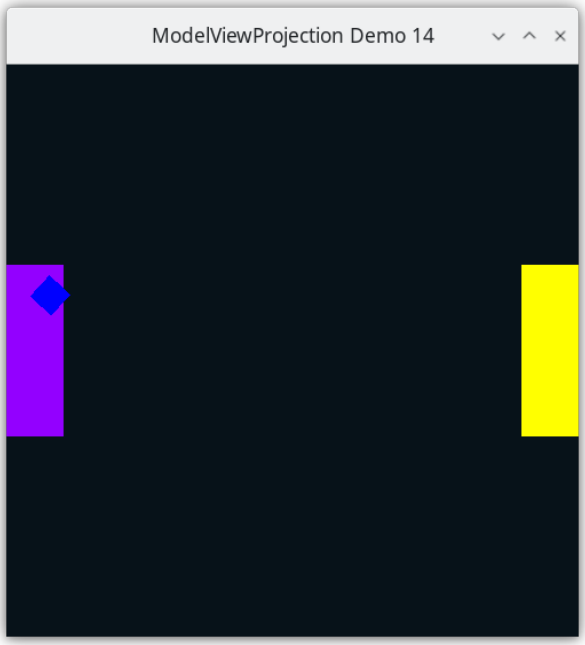


Fig. 1: Demo 14

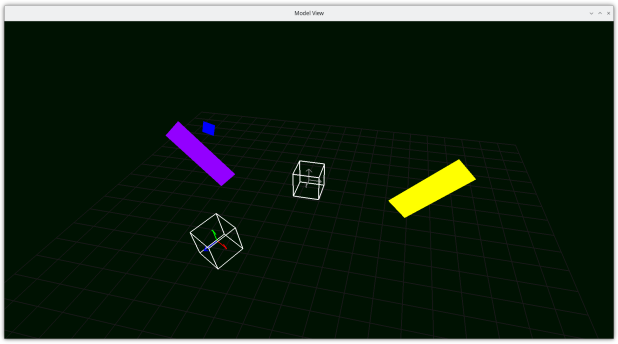


Fig. 2: Camera Space

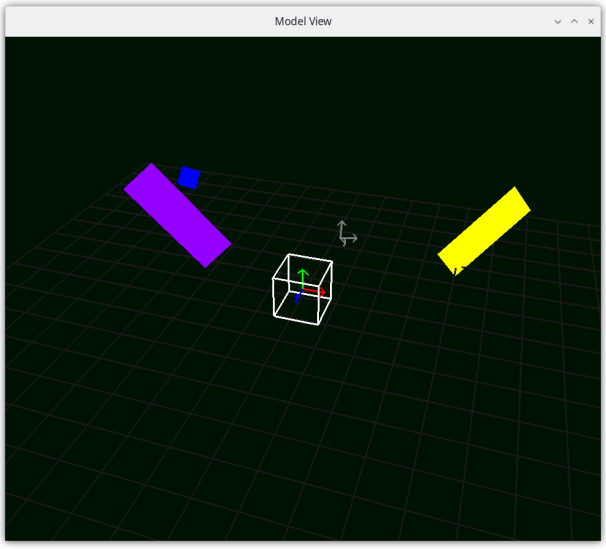


Fig. 3: Camera Space

15.2 How to Execute

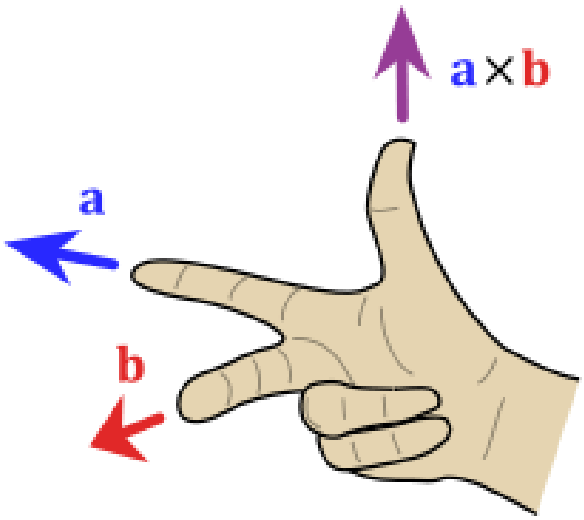
Load src/demo14/demo.py in Spyder and hit the play button

15.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

15.4 Description

- Vertex data will now have an X, Y, and Z component.
- Rotations around an angle in 3D space follow the right hand rule. Here's a link to them in matrix form, which we have not yet covered.



- With open palm, fingers on the x axis, rotating the fingers to y axis, means that the positive z axis is in the direction of the thumb. Positive Theta moves in the direction that your fingers did.

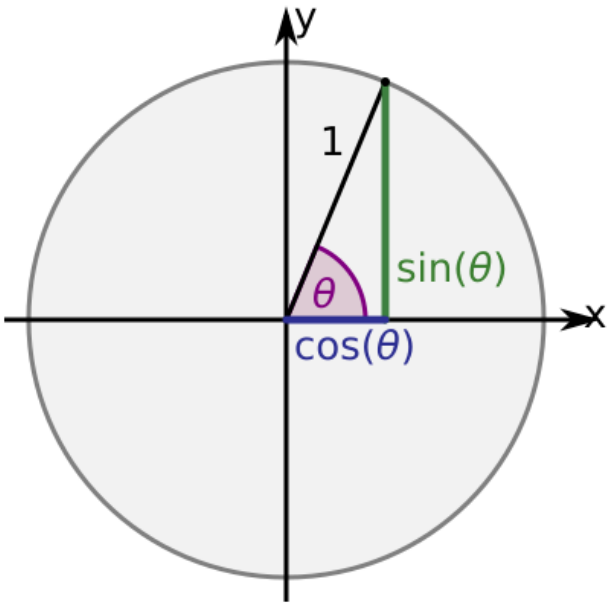
- starting on the y axis, rotating to z axis, thumb is on the positive x axis.
- starting on the z axis, rotating to x axis, thumb is on the positive y axis.

Listing 1: src/demo14/demo.py

```
@dataclass
class Vertex2D:
    x: float
    y: float

    def __add__(self, rhs: Vertex2D) -> Vertex2D:
        return Vertex2D(x=self.x + rhs.x, y=self.y + rhs.y)

    def translate(self: Vertex2D, translate_amount: Vertex2D) -> Vertex2D:
        return self + translate_amount

    def __mul__(self, scalar: float) -> Vertex2D:
        return Vertex2D(x=self.x * scalar, y=self.y * scalar)

    def __rmul__(self, scalar: float) -> Vertex2D:
        return self * scalar

    def uniform_scale(self: Vertex2D, scalar: float) -> Vertex2D:
        return self * scalar

    def scale(self: Vertex2D, scale_x: float, scale_y: float) -> Vertex2D:
        return Vertex2D(x=self.x * scale_x, y=self.y * scale_y)

    def __neg__(self):
        return -1.0 * self

    def rotate_90_degrees(self: Vertex2D):
        return Vertex2D(x=-self.y, y=self.x)

    def rotate(self: Vertex2D, angle_in_radians: float) -> Vertex2D:
        return (
            math.cos(angle_in_radians) * self
            + math.sin(angle_in_radians) * self.rotate_90_degrees()
        )


@dataclass
class Vertex:
    x: float
    y: float
    z: float

    def __add__(self, rhs: Vertex) -> Vertex:
        return Vertex(x=self.x + rhs.x, y=self.y + rhs.y, z=self.z + rhs.z)

    def translate(self: Vertex, translate_amount: Vertex) -> Vertex:
        return self + translate_amount

```

15.4.1 Rotate Z

Rotate Z is the same rotate that we've used so far, but doesn't affect the z component at all.



Listing 2: src/demo14/demo.py

```
174     def rotate_z(self: Vertex, angle_in_radians: float) -> Vertex:
175         xy_on_xy: Vertex2D = Vertex2D(x=self.x, y=self.y).rotate(angle_in_radians)
176         return Vertex(x=xy_on_xy.x, y=xy_on_xy.y, z=self.z)
177
```

15.4.2 Rotate X

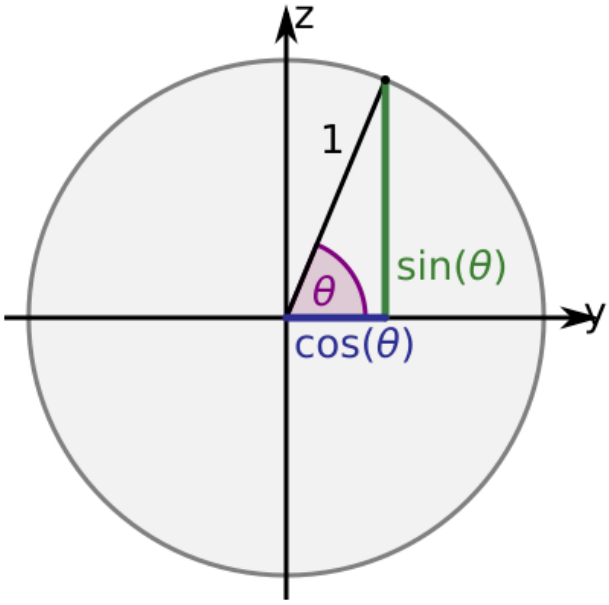
Listing 3: src/demo14/demo.py

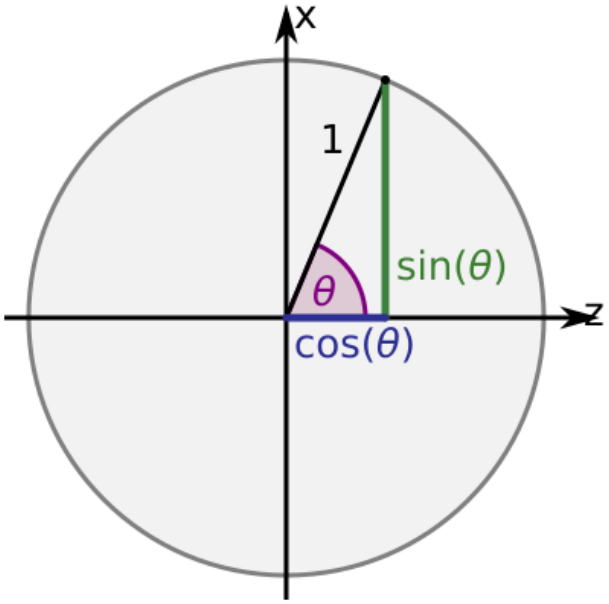
```
160     def rotate_x(self: Vertex, angle_in_radians: float) -> Vertex:
161         yz_on_xy: Vertex2D = Vertex2D(x=self.y, y=self.z).rotate(angle_in_radians)
162         return Vertex(x=self.x, y=yz_on_xy.x, z=yz_on_xy.y)
163
```

15.4.3 Rotate Y

Listing 4: src/demo14/demo.py

```
167     def rotate_y(self: Vertex, angle_in_radians: float) -> Vertex:
168         zx_on_xy: Vertex2D = Vertex2D(x=self.z, y=self.x).rotate(angle_in_radians)
169         return Vertex(x=zx_on_xy.y, y=self.y, z=zx_on_xy.y)
170
```


z
1
sin(θ)
θ
y
cos(θ)


x
1
sin(θ)
θ
z
cos(θ)

15.4.4 Scale

Listing 5: src/demo14/demo.py

```
181
182     def __mul__(self, scalar: float) -> Vertex:
183         return Vertex(x=self.x * scalar, y=self.y * scalar, z=self.z * scalar)
184
185     def __rmul__(self, scalar: float) -> Vertex:
186         return self * scalar
187
188     def uniform_scale(self: Vertex, scalar: float) -> Vertex:
189         return self * scalar
190
191     def scale(self: Vertex, scale_x: float, scale_y: float, scale_z: float) -> Vertex:
192         return Vertex(x=self.x * scale_x, y=self.y * scale_y, z=self.z * scale_z)
193
194     def __neg__(self):
195         return -1.0 * self
196
```

15.5 Code

The only new aspect of the code below is that the paddles have a z-coordinate of 0 in their modelspace.

Listing 6: src/demo14/demo.py

```
211 paddle1: Paddle = Paddle(
212     vertices=[
213         Vertex(x=-1.0, y=-3.0, z=0.0),
214         Vertex(x=1.0, y=-3.0, z=0.0),
215         Vertex(x=1.0, y=3.0, z=0.0),
216         Vertex(x=-1.0, y=3.0, z=0.0),
217     ],
218     r=0.578123,
219     g=0.0,
220     b=1.0,
221     position=Vertex(x=-9.0, y=0.0, z=0.0),
222 )
223
224 paddle2: Paddle = Paddle(
225     vertices=[
226         Vertex(x=-1.0, y=-3.0, z=0.0),
227         Vertex(x=1.0, y=-3.0, z=0.0),
228         Vertex(x=1.0, y=3.0, z=0.0),
229         Vertex(x=-1.0, y=3.0, z=0.0),
230     ],
231     r=1.0,
232     g=1.0,
233     b=0.0,
234     position=Vertex(x=9.0, y=0.0, z=0.0),
235 )
```

The only new aspect of the square below is that the paddles have a z-coordinate of 0 in their modelspace. N.B that since we do a sequence transformations to the modelspace data to get to world-space coordinates, the X, Y, and Z coordinates are subject to be different.

Listing 7: src/demo14/demo.py

```
240 @dataclass
241 class Camera:
242     position_ws: Vertex = field(default_factory=lambda: Vertex(x=0.0, y=0.0, z=0.0))
243
244
245 camera: Camera = Camera()
```

The camera now has a z-coordinate of 0 also.

Listing 8: src/demo14/demo.py

```
249 square: list[Vertex] = [
250     Vertex(x=-0.5, y=-0.5, z=0.0),
251     Vertex(x=0.5, y=-0.5, z=0.0),
252     Vertex(x=0.5, y=0.5, z=0.0),
253     Vertex(x=-0.5, y=0.5, z=0.0),
254 ]
```

15.5.1 Event Loop

Listing 9: src/demo14/demo.py

```
307 while not glfw.window_should_close(window):
```

```
    ...
```

- Draw Paddle 1

Listing 10: src/demo14/demo.py

```
326     glColor3f(paddle1.r, paddle1.g, paddle1.b)
327     glBegin(GL_QUADS)
328     for paddle1_vertex_ms in paddle1.vertices:
329         paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate_z(
330             paddle1.rotation
331         ).translate(paddle1.position)
332         paddle1_vertex_cs: Vertex = paddle1_vertex_ws.translate(-camera.position_ws)
333         paddle1_vertex_ndc: Vertex = paddle1_vertex_cs.uniform_scale(scalar=1.0 / 10.0)
334         glVertex2f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y)
335     glEnd()
```

The square should not be visible when hidden behind the paddle1, as we do a translate by -1. But in running the demo, you see that the square is always drawn over the paddle.

- Draw the Square

Listing 11: src/demo14/demo.py

```
    # draw square
    glColor3f(0.0, 0.0, 1.0)
    glBegin(GL_QUADS)
    for model_space in square:
        paddle_1_space: Vertex = (
            model_space.rotate_z(square_rotation)
            .translate(Vertex(x=2.0, y=0.0, z=0.0))
            .rotate_z(rotation_around_paddle1)
            .translate(Vertex(x=0.0, y=0.0, z=-1.0))
        )
        world_space: Vertex = paddle_1_space.rotate_z(paddle1.rotation).translate(
            paddle1.position
        )
        camera_space: Vertex = world_space.translate(-camera.position_ws)
        ndc: Vertex = camera_space.uniform_scale(scalar=1.0 / 10.0)
        glVertex3f(ndc.x, ndc.y, ndc.z)
    glEnd()
```

This is because without depth buffering, the object drawn last clobbers the color of any previously drawn object at the pixel. Try moving the square drawing code to the beginning, and you will see that the square can be hidden behind the paddle.

- Draw Paddle 2

Listing 12: src/demo14/demo.py

```
    # draw paddle 2
    glColor3f(paddle2.r, paddle2.g, paddle2.b)
    glBegin(GL_QUADS)
    for paddle2_vertex_ms in paddle2.vertices:
        paddle2_vertex_ws: Vertex = paddle2_vertex_ms.rotate_z(
            paddle2.rotation
        ).translate(paddle2.position)
        paddle2_vertex_cs: Vertex = paddle2_vertex_ws.translate(-camera.position_ws)
        paddle2_vertex_ndc: Vertex = paddle2_vertex_cs.uniform_scale(scalar=1.0 / 10.0)
        glVertex3f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y, paddle2_vertex_ndc.z)
    glEnd()
```

Added translate in 3D. Added scale in 3D. These are just like the 2D versions, just with the same process applied to the z axis.

They direction of the rotation is defined by the right hand rule.

https://en.wikipedia.org/wiki/Right-hand_rule

CHAPTER

SIXTEEN

ADDING DEPTH - ENABLE DEPTH BUFFER - DEMO 15

16.1 Purpose

Fix the issue from the last demo, in which the square was drawn over Paddle 1, even though the square is further away from the camera.

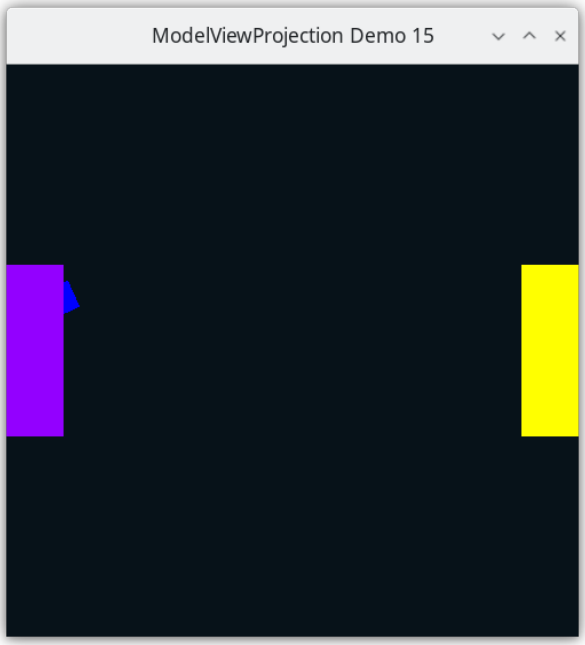


Fig. 1: Demo 15

16.2 How to Execute

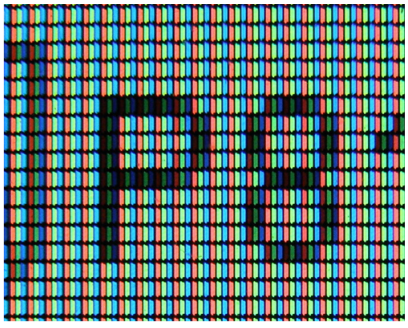
Load src/demo15/demo.py in Spyder and hit the play button

16.3 Move the Paddles using the Keyboard

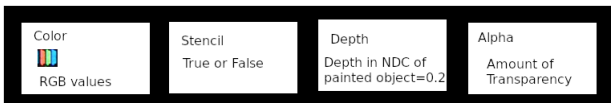
Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

16.4 Description

Early in this book we used the stencil buffer, but didn't go into much detail about it. Each pixel in the frame-buffer is a fragment, which is more information than just the color to be drawn.

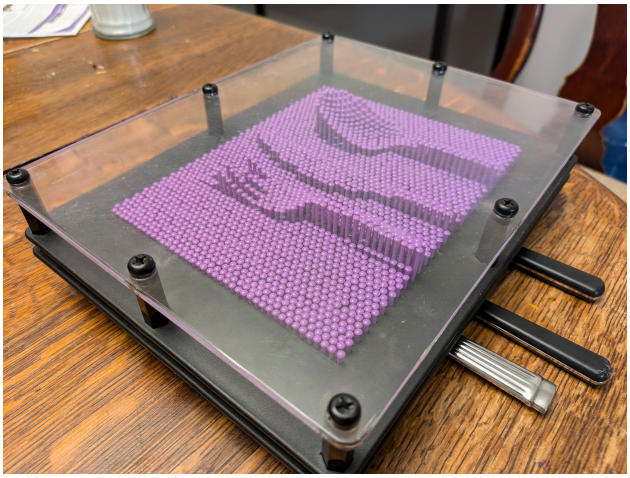


Fragment



Besides the color, there is also

- Stencil buffer, being true or false, to specify whether subsequent OpenGL calls should affect this fragment or leave it alone



- Depth - When an object in NDC is drawn, the color is updated to the new value, but the z-component of the NDC is placed into the depth buffer. When a new object is drawn at this pixel later, its Z value will be compared to the existing Z value, to determine which object is in front. If the new object is farther away from the camera than the already-draw object, then the fragment will not be updated.
- Alpha, for transparency, which we have not yet covered.

For a history of what 3D programming for game systems was like without having a z-buffer, take a look at some history of the Nintendo 64, and comments on hacker news about it.

Use the depth buffer to make further objects hidden if nearer objects are drawn in front

1. Set the clear depth to -1 (just like clearcolor, it is the default depth on a given fragment (pixel).
2. Set the depth func, i.e. the test to see if the newly drawn object should overwrite the color in the current fragment or not.
3. Enable the depth test.

Listing 1: src/demo15/demo.py

```
79 glClearDepth(-1.0)
80 glDepthFunc(GL_GREATER)
81 glEnable(GL_DEPTH_TEST)
```

the square should not be visible when hidden behind the paddle1, as we did a translate by -1. this is because without depth buffering, the object drawn last clobbers the color of any previously drawn object at the pixel. Try moving the square drawing code to the beginning, and you will see that the square can be hidden behind the paddle.

CHAPTER

SEVENTEEN

MOVING CAMERA IN 3D SPACE - DEMO 16

17.1 Purpose

Make a moving camera in 3D space. Use Ortho to transform a rectangular prism, defined relative to camera space, into NDC.

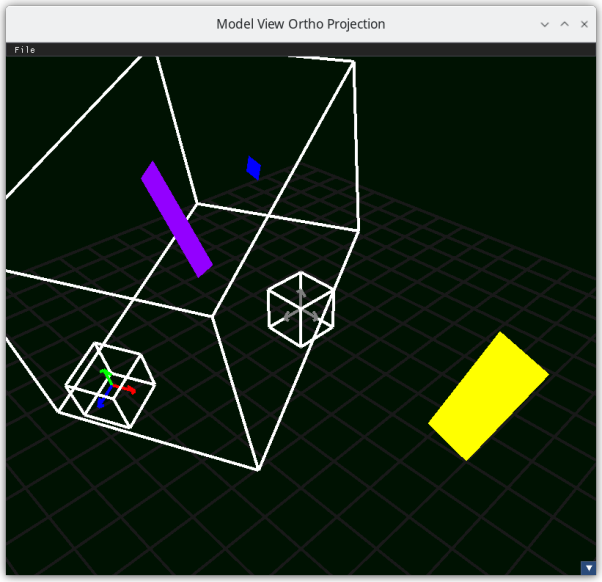


Fig. 1: Camera space with ortho volume

17.1.1 Problem purposefully put in

When running this demo and moving the viewer, parts of the geometry will disappear. This is because it gets "clipped out", as the geometry will be outside of NDC, (-1 to 1 on all three axis). We could fix this by making a bigger ortho rectangular prism, but that won't solve the fundamental problem.

This doesn't look like a 3D application should, where objects further away from the viewer would appear smaller. This will be fixed in demo17.

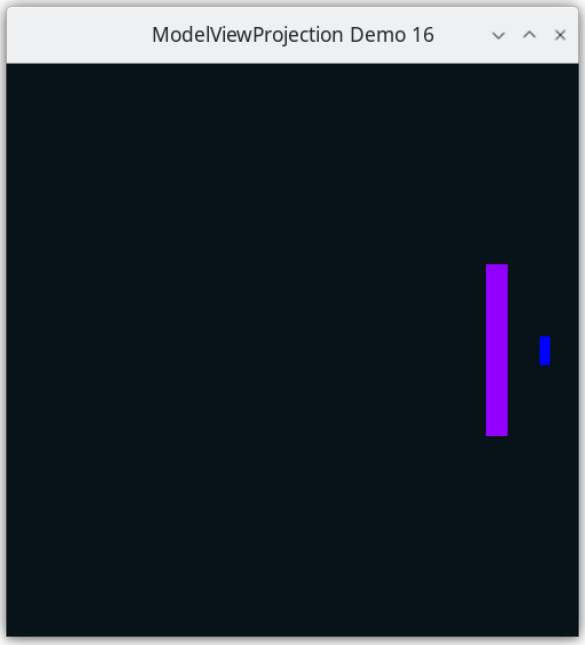


Fig. 2: Demo 16, which looks like trash

17.2 How to Execute

Load src/demo16/demo.py in Spyder and hit the play button

17.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

17.4 Description

Before starting this demo, run mvpVisualization/modelvieworthoprojection/modelvieworthoprojection.py, as it will show graphically all of the steps in this demo. In the GUI, take a look at the camera options buttons, and once the camera is placed and oriented in world space, use the buttons to change the camera's position and orientation. This will demonstrate what we have to do for moving the camera in a 3D scene.

There are new keyboard inputs to control the moving camera. As you would expect to see in a first person game, up moves the camera forward (-z), down moves the camera backwards (z), left rotates the camera as would happen if you rotated your body to the left, and likewise for right. Page UP and Page DOWN rotate the camera to look up or to look down.

To enable this, the camera is modeled with a data structure, having a position in x,y,z relative to world space, and two rotations (one around the camera's x axis, and one around the camera's y axis).

To position the camera you would

1. translate to the camera's position, using the actual position values of camera position in world space coordinates.
2. rotate around the local y axis
3. rotate around the local x axis

To visualize this, run "python mvpVisualization/modelvieworthoprojection/modelvieworthoprojection.py"

The ordering of 1) before 2) and 3) should be clear, as we are imagining a coordinate system that moves, just like we do for the model-space to world space transformations. The ordering of 2) before 3) is very important, as two rotations around different axes are not commutative, meaning that you can't change the order and still expect the same results https://en.wikipedia.org/wiki/Commutative_property.

Try this. Rotate your head to the right a little more that 45 degrees. Now rotate your head back a little more than 45 degrees.

Now, reset your head (glPopMatrix, which we have not yet covered). Try rotating your head back 45 degrees. Once it is there, rotate your head (not your neck), 45 degrees. It is different, and quite uncomfortable!

We rotate the camera by the y axis first, then by the relative x axis, for the same reason.

Listing 1: src/demo16/demo.py

```
375         # paddle1_vertex_ws: Vertex = paddle1_vertex_cs.rotate_x(camera.rot_x) \
376         #                                              .rotate_y(camera.rot_y) \
377         #                                              .translate(camera.position_ws)
```

(Remember, read bottom up, just like the previous demos for model-space to world-space data)

Back to the point, we are envisioning the camera relative to the world space by making a moving coordinate system (composed of an origin, 1 unit in the "x" axis, 1 unit in the "y" axis, and 1 unit in the "z" axis), where each subsequent transformation is relative to the previous coordinate system. (This system is beneficial btw because it allows us to think of only one coordinate system at a time, and allows us to forget how we got there, (similar to a Markov process, https://en.wikipedia.org/wiki/Markov_chain))

But this system of thinking works only when we are placing the camera into its position/orientation relative to world space, which is not what we need to actually do. We don't need to place the camera. We need to move every already-plotted object in world space towards the origin and orientation of NDC. Looking at the following graph,

We want to take the model-space geometry from, say Paddle1 space, to world space, and then to camera space (which is going in the opposite direction of the arrow, therefore requires an inverse operation, because to plot data we go from model-space to screen space on the graph.

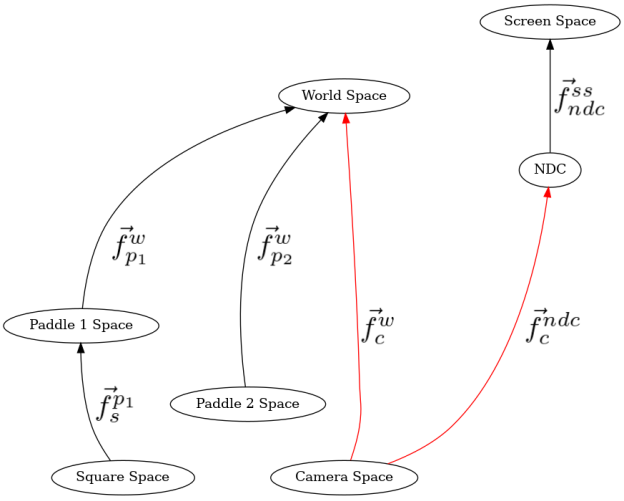


Fig. 3: Demo 16

Given that the inverse of a sequence of transformations is the sequence backwards, with each transformations inverted, we must do that to get from world space to camera space.

The inverted form is

Listing 2: src/demo16/demo.py

```
380         paddle1_vertex_cs: Vertex = (
381             paddle1_vertex_ws.translate(-camera.position_ws)
382             .rotate_y(-camera.rot_y)
383             .rotate_x(-camera.rot_x)
384         )
```

Other things added Added rotations around the x axis, y axis, and z axis. https://en.wikipedia.org/wiki/Rotation_matrix

17.5 Code

The camera now has two angles as instance variables.

Listing 3: src/demo16/demo.py

```
267
268
269 @dataclass
270 class Camera:
271     position_ws: Vertex = field(default_factory=lambda: Vertex(x=0.0, y=0.0, z=15.0))
272     rot_y: float = 0.0
273     rot_x: float = 0.0
```

Since we want the user to be able to control the camera, we need to read the input.

Listing 4: src/demo16/demo.py

```
291 def handle_inputs() -> None:
```

```
...
```

Left and right rotate the viewer's horizontal angle, page up and page down the vertical angle.

Listing 5: src/demo16/demo.py

```
304     if glfw.get_key(window, glfw.KEY_RIGHT) == glfw.PRESS:
305         camera.rot_y -= 0.03
306     if glfw.get_key(window, glfw.KEY_LEFT) == glfw.PRESS:
307         camera.rot_y += 0.03
308     if glfw.get_key(window, glfw.KEY_PAGE_UP) == glfw.PRESS:
309         camera.rot_x += 0.03
310     if glfw.get_key(window, glfw.KEY_PAGE_DOWN) == glfw.PRESS:
311         camera.rot_x -= 0.03
```

The up arrow and down arrow make the user move forwards and backwards. Unlike the camera space to world space transformation, here for movement code, we don't do the rotate around the x axis. This is because users expect to simulate walking on the ground, not flying through the sky. I.e, we want forward/backwards movement to happen relative to the XZ plane at the camera's position, not forward/backwards movement relative to camera space.

Listing 6: src/demo16/demo.py

```
314     if glfw.get_key(window, glfw.KEY_UP) == glfw.PRESS:
315         forwards_cs = Vertex(x=0.0, y=0.0, z=-1.0)
316         forward_ws = forwards_cs.rotate_y(camera.rot_y).translate(camera.position_ws)
317         camera.position_ws = forward_ws
318     if glfw.get_key(window, glfw.KEY_DOWN) == glfw.PRESS:
319         forwards_cs = Vertex(x=0.0, y=0.0, z=1.0)
320         forward_ws = forwards_cs.rotate_y(camera.rot_y).translate(camera.position_ws)
321         camera.position_ws = forward_ws
```

Ortho is the function call that shrinks the viewable region relative to camera space down to NDC, by moving the center of the rectangular prism to the origin, and scaling by the inverse of the width, height, and depth of the viewable region.

Listing 7: src/demo16/demo.py

```
192     def ortho(self: Vertex,
193               left: float,
194               right: float,
195               bottom: float,
196               top: float,
197               near: float,
198               far: float,
199               ) -> Vertex:
200         midpoint = Vertex(
201             x=(left + right) / 2.0,
202             y=(bottom + top) / 2.0,
203             z=(near + far) / 2.0
204         )
```

(continues on next page)

(continued from previous page)

```
        length_x: float
        length_y: float
        length_z: float
        length_x, length_y, length_z = right - left, top - bottom, far - near
        return self.translate(-midpoint) \
                   .scale(2.0 / length_x,
                          2.0 / length_y,
                          2.0 / (-length_z))
```

We will make a wrapper function camera_space_to_ndc_space_fn which calls ortho, setting the size of the rectangular prism.

Listing 8: src/demo16/demo.py

```
    def camera_space_to_ndc_fn(self: Vertex) -> Vertex:
        return self.ortho(left=-10.0,
                          right=10.0,
                          bottom=-10.0,
                          top=10.0,
                          near=-0.1,
                          far=-30.0)
```

17.5.1 Event Loop

The amount of repetition in the code below in starting to get brutal, as there's too much detail to think about and retype out for every object being drawn, and we're only dealing with 3 objects. The author put this repetition into the book on purpose, so that when we start using matrices later, the reader will fully appreciate what matrices solve for us.

Listing 9: src/demo16/demo.py

```
while not glfw.window_should_close(window):
```

```
...
```

Paddle 1

Listing 10: src/demo16/demo.py

```
    glColor3f(paddle1.r, paddle1.g, paddle1.b)
    glBegin(GL_QUADS)
    for paddle1_vertex_ms in paddle1.vertices:
        paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate_z(
            paddle1.rotation
        ).translate(paddle1.position)
        # doc-region-begin commented out camera placement
        # paddle1_vertex_ws: Vertex = paddle1_vertex_cs.rotate_x(camera.rot_x) \
        #                                              .rotate_y(camera.rot_y) \
        #                                              .translate(camera.position_ws)
        # doc-region-end commented out camera placement
        # doc-region-begin inverted transformation to go from world space to camera space
        paddle1_vertex_cs: Vertex = (
```

(continues on next page)

(continued from previous page)

```
            paddle1_vertex_ws.translate(-camera.position_ws)
            .rotate_y(-camera.rot_y)
            .rotate_x(-camera.rot_x)
        )
        # doc-region-end inverted transformation to go from world space to camera space
        paddle1_vertex_ndc: Vertex = paddle1_vertex_cs.camera_space_to_ndc_fn()
        glVertex3f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y, paddle1_vertex_ndc.z)
    glEnd()
```

Square

the square should not be visible when hidden behind the paddle1, as we did a translate by -10 in the z direction.

Listing 11: src/demo16/demo.py

```
    glColor3f(0.0, 0.0, 1.0)
    glBegin(GL_QUADS)
    for model_space in square:
        paddle_1_space: Vertex = (
            model_space.rotate_z(square_rotation)
            .translate(Vertex(x=2.0, y=0.0, z=0.0))
            .rotate_z(rotation_around_paddle1)
            .translate(Vertex(x=0.0, y=0.0, z=-1.0))
        )
        world_space: Vertex = paddle_1_space.rotate_z(paddle1.rotation).translate(
            paddle1.position
        )
        # world_space: Vertex = camera_space.rotate_x(camera.rot_x) \
        #                                   .rotate_y(camera.rot_y) \
        #                                   .translate(camera.position_ws)
        camera_space: Vertex = (
            world_space.translate(-camera.position_ws)
            .rotate_y(-camera.rot_y)
            .rotate_x(-camera.rot_x)
        )
        ndc: Vertex = camera_space.camera_space_to_ndc_fn()
        glVertex3f(ndc.x, ndc.y, ndc.z)
    glEnd()
```

Paddle 2

Listing 12: src/demo16/demo.py

```
    glColor3f(paddle2.r, paddle2.g, paddle2.b)
    glBegin(GL_QUADS)
    for paddle2_vertex_ms in paddle2.vertices:
        paddle2_vertex_ws: Vertex = paddle2_vertex_ms.rotate_z(
            paddle2.rotation
        ).translate(paddle2.position)
        # paddle2_vertex_ws: Vertex = paddle2_vertex_cs.rotate_x(camera.rot_x) \
        #                                                 .rotate_y(camera.rot_y)␣
↪\
        #                                                 .translate(camera.
```

(continues on next page)

(continued from previous page)

```
↪position_ws)

        paddle2_vertex_cs: Vertex = (
            paddle2_vertex_ws.translate(-camera.position_ws)
            .rotate_y(-camera.rot_y)
            .rotate_x(-camera.rot_x)
        )

        paddle2_vertex_ndc: Vertex = paddle2_vertex_cs.camera_space_to_ndc_fn()
        glVertex3f(paddle2_vertex_ndc.x, paddle2_vertex_ndc.y, paddle2_vertex_ndc.z)
    glEnd()
```

CHAPTER

EIGHTEEN

3D PERSPECTIVE - DEMO 17

18.1 Purpose

Implement a perspective projection so that objects further away are smaller than the would be if they were close by

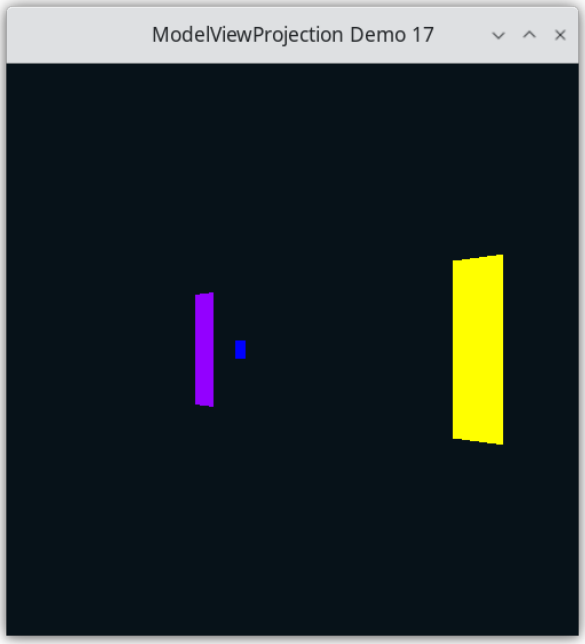


Fig. 1: Demo 17

18.2 How to Execute

Load src/demo17/demo.py in Spyder and hit the play button

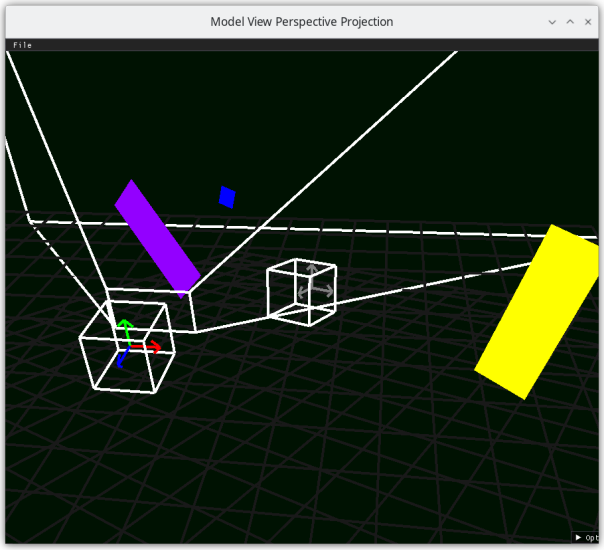


Fig. 2: Frustum 1

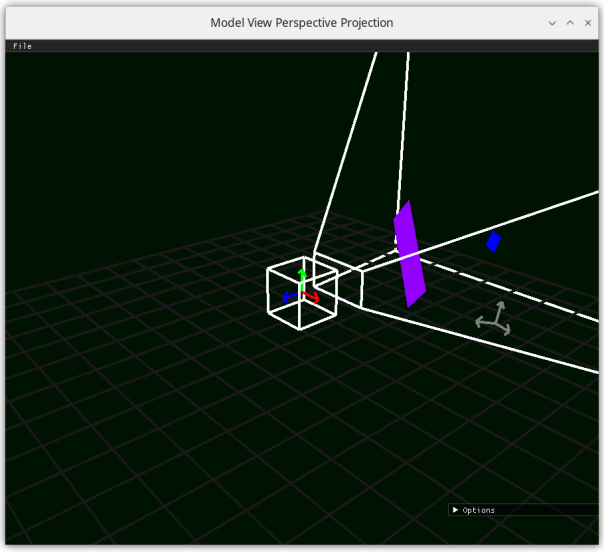


Fig. 3: Frustum 2

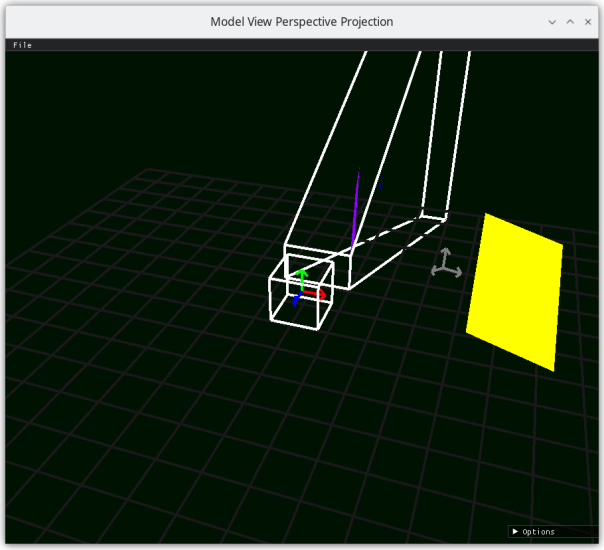


Fig. 4: Frustum 3

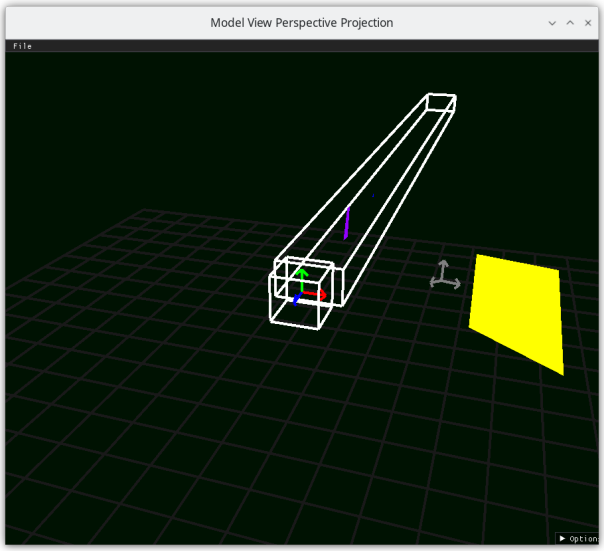


Fig. 5: Frustum 4

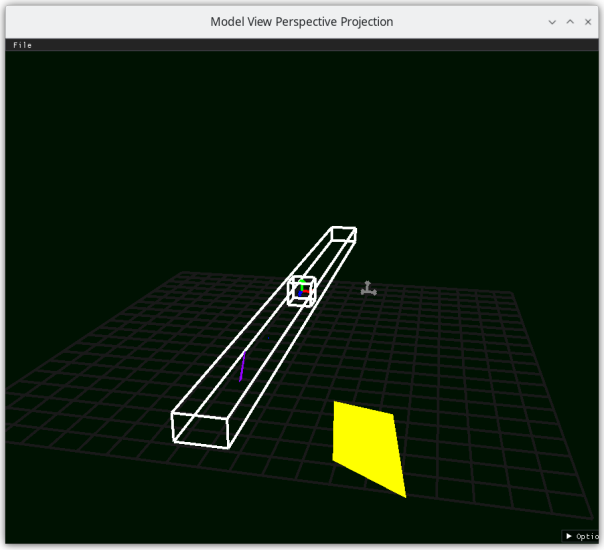


Fig. 6: Frustum 5

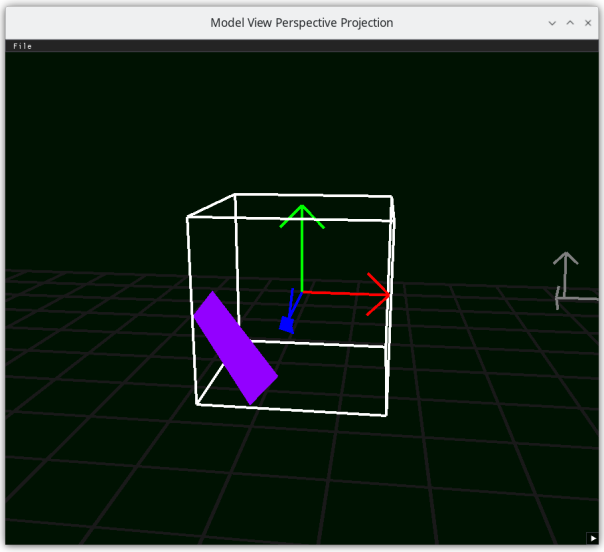
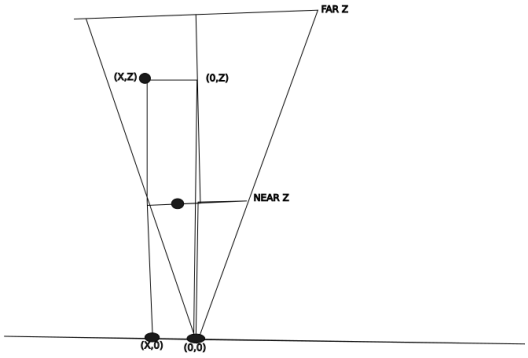


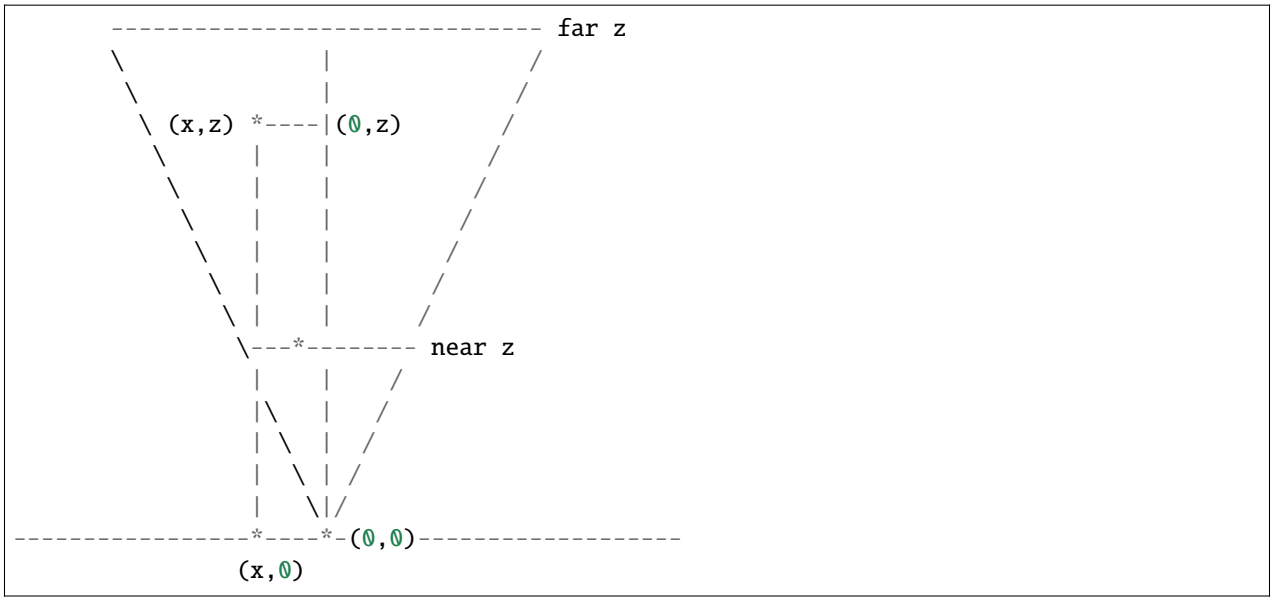
Fig. 7: Frustum 6

18.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

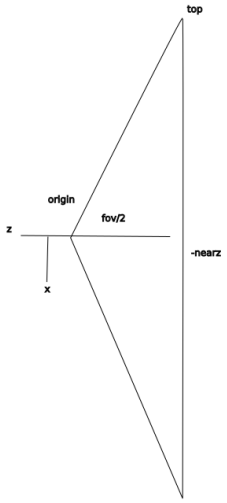
18.4 Description



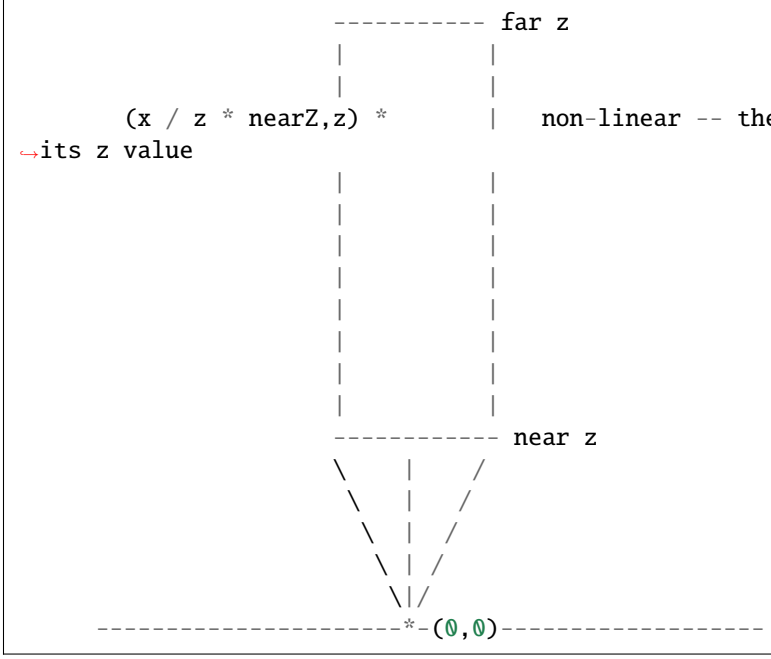


If we draw a straight line between (x,z) and (0,0), we will have a right triangle with vertices (0,0), (0,z), and (x,z).

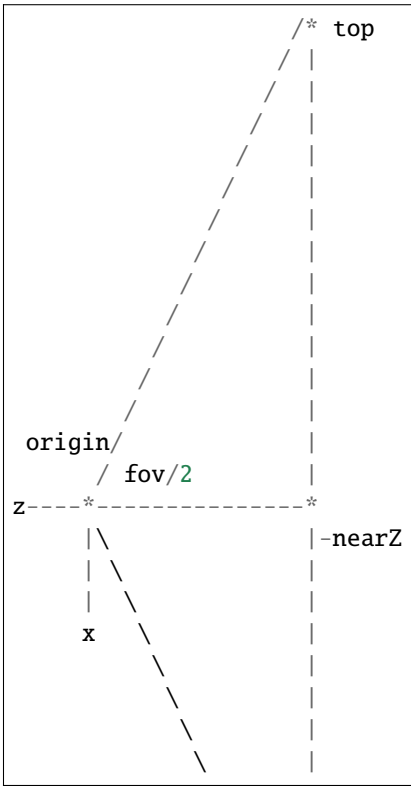
There also will be a similar right triangle with vertices (0,0), (0,nearZ), and whatever point the line above intersects the line at z=nearZ. Let's call that point (projX, nearZ)



```
because right angle and tan(theta) = tan(theta)
x / z = projX / nearZ
projX = x / z * nearZ

So use projX as the transformed x value, and keep the distance z.
```



Top calculation based off of vertical field of view



(continues on next page)

(continued from previous page)

```
                \      |
                 \     |
                  \    |
                   \   |
                    \  |
                     \ |
                      \|
```

Right calculation based off of Top and aspect ration

```
                        top
-------------------------------------------------------
|                                                     |
|                         y                           |
|                         |                           |
|                         |                           |
|                         *----x                      | right =
|                            origin                   |   top * aspectRatio
|                                                     |
|                                                     |  aspect ratio should be the␣
↪viewport's
|                                                     |    width/height, not necessarily␣
↪the
-------------------------------------------------------    window's
```

Listing 1: src/demo17/demo.py

```
@dataclass
class Vertex:
```

```
...
```

Listing 2: src/demo17/demo.py

```
    def perspective(self: Vertex,
                    field_of_view: float,
                    aspect_ratio: float,
                    near_z: float,
                    far_z: float) -> Vertex:
        # field_of_view, field of view, is angle of y
        # aspect_ratio is x_width / y_width

        top: float = -near_z * math.tan(math.radians(field_of_view) / 2.0)
        right: float = top * aspect_ratio

        scaled_x: float = self.x / self.z * near_z
        scaled_y: float = self.y / self.z * near_z
        rectangular_prism: Vertex = Vertex(scaled_x,
                                           scaled_y,
                                           self.z)

        return rectangular_prism.ortho(left=-right,
```

(continues on next page)

(continued from previous page)

```
                                    right=right,
                                    bottom=-top,
                                    top=top,
                                    near=near_z,
                                    far=far_z)

    def camera_space_to_ndc_fn(self: Vertex) -> Vertex:
        return self.perspective(field_of_view=45.0,
                                aspect_ratio=1.0,
                                near_z=-.1,
                                far_z=-1000.0)
```

Listing 3: src/demo17/demo.py

```
while not glfw.window_should_close(window):
```

```
...
```

Listing 4: src/demo17/demo.py

```
    # draw paddle 1
    glColor3f(paddle1.r, paddle1.g, paddle1.b)

    glBegin(GL_QUADS)
    for paddle1_vertex_ms in paddle1.vertices:
        paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate_z(
            paddle1.rotation
        ).translate(paddle1.position)
        # paddle1_vertex_ws: Vertex = paddle1_vertex_cs.rotate_x(camera.rot_x) \
        #                                              .rotate_y(camera.rot_y) \
        #                                              .translate(camera.position_ws)
        paddle1_vertex_cs: Vertex = (
            paddle1_vertex_ws.translate(-camera.position_ws)
            .rotate_y(-camera.rot_y)
            .rotate_x(-camera.rot_x)
        )
        paddle1_vertex_ndc: Vertex = paddle1_vertex_cs.camera_space_to_ndc_fn()
        glVertex3f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y, paddle1_vertex_ndc.z)
    glEnd()
```

CHAPTER

NINETEEN

LAMBDA STACK - DEMO 18

19.1 Purpose

Remove repetition in the coordinate transformations, as previous demos had very similar transformations, especially from camera space to NDC space. Each edge of the graph of objects should only be specified once per frame.

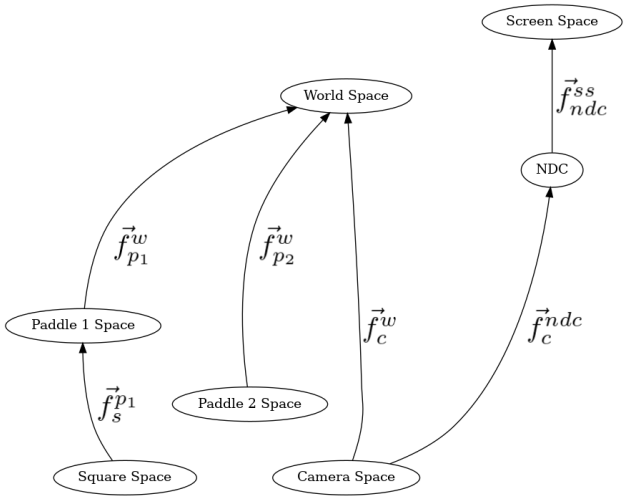


Fig. 1: Full Cayley graph.

Noticing in the previous demos that the lower parts of the transformations have a common pattern, we can create a stack of functions for later application. Before drawing geometry, we add any functions to the top of the stack, apply all of our functions in the stack to our model-space data to get NDC data, and before we return to the parent node, we pop the functions we added off of the stack, to ensure that we return the stack to the state that the parent node gave us.

To explain in more detail —

What's the difference between drawing paddle 1 and the square?

Here is paddle 1 code

Listing 1: src/demo17/demo.py

```
    # draw paddle 1
    glColor3f(paddle1.r, paddle1.g, paddle1.b)

    glBegin(GL_QUADS)
    for paddle1_vertex_ms in paddle1.vertices:
```

(continues on next page)

(continued from previous page)

```
        paddle1_vertex_ws: Vertex = paddle1_vertex_ms.rotate_z(
            paddle1.rotation
        ).translate(paddle1.position)
        # paddle1_vertex_ws: Vertex = paddle1_vertex_cs.rotate_x(camera.rot_x) \
        #                                              .rotate_y(camera.rot_y) \
        #                                              .translate(camera.position_ws)
        paddle1_vertex_cs: Vertex = (
            paddle1_vertex_ws.translate(-camera.position_ws)
            .rotate_y(-camera.rot_y)
            .rotate_x(-camera.rot_x)
        )
        paddle1_vertex_ndc: Vertex = paddle1_vertex_cs.camera_space_to_ndc_fn()
        glVertex3f(paddle1_vertex_ndc.x, paddle1_vertex_ndc.y, paddle1_vertex_ndc.z)
    glEnd()
```

Here is the square's code:

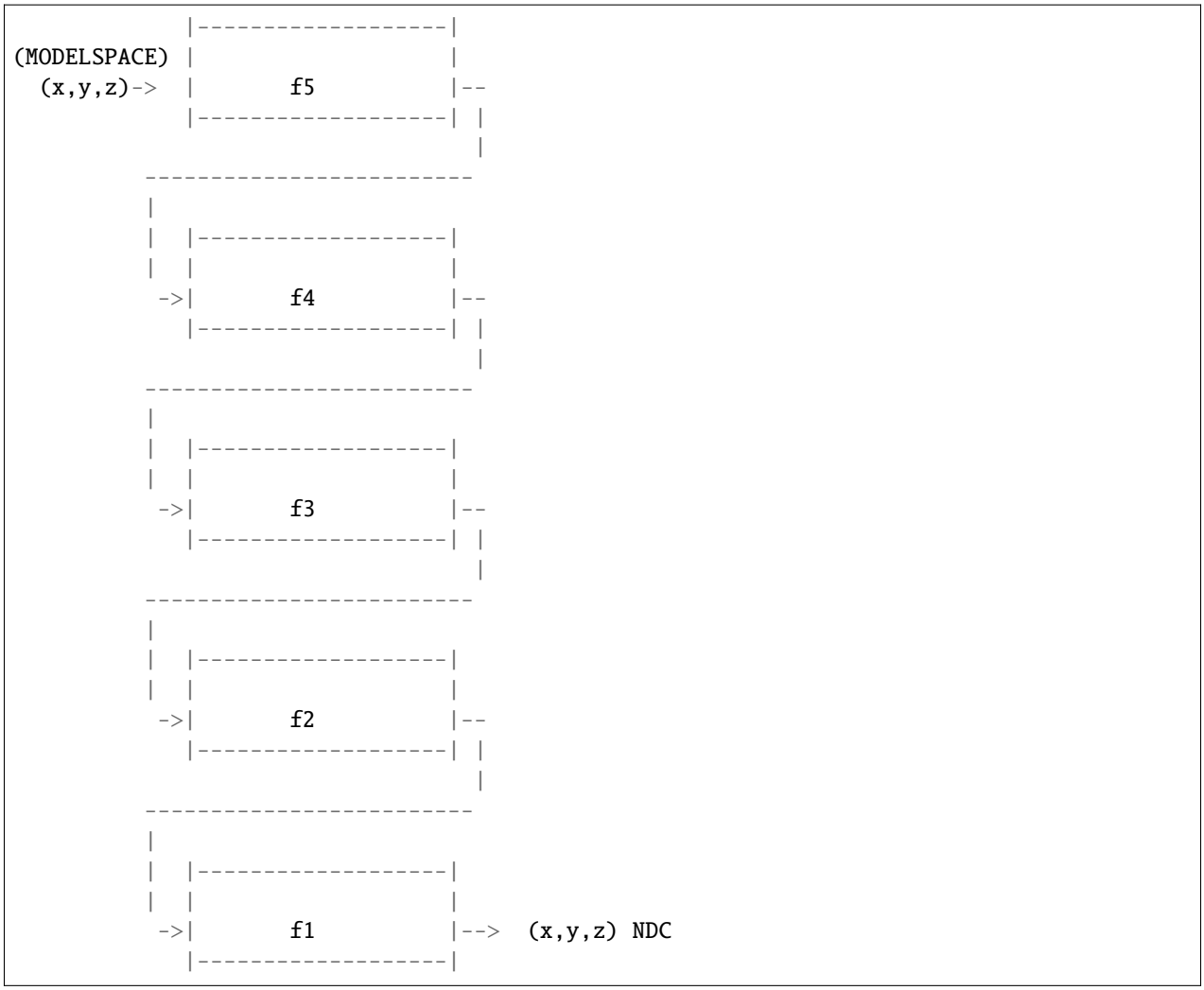
Listing 2: src/demo17/demo.py

```
    glColor3f(0.0, 0.0, 1.0)
    glBegin(GL_QUADS)
    for model_space in square:
        paddle_1_space: Vertex = (
            model_space.rotate_z(square_rotation)
            .translate(Vertex(x=2.0, y=0.0, z=0.0))
            .rotate_z(rotation_around_paddle1)
            .translate(Vertex(x=0.0, y=0.0, z=-1.0))
        )
        world_space: Vertex = paddle_1_space.rotate_z(paddle1.rotation).translate(
            paddle1.position
        )
        camera_space: Vertex = (
            world_space.translate(-camera.position_ws)
            .rotate_y(-camera.rot_y)
            .rotate_x(-camera.rot_x)
        )
        ndc: Vertex = camera_space.camera_space_to_ndc_fn()
        glVertex3f(ndc.x, ndc.y, ndc.z)
    glEnd()
```

The only difference is the square's model-space to paddle1 space. Everything else is exactly the same. In a graphics program, because the scene is a hierarchy of relative objects, it is unwise to put this much repetition in the transformation sequence. Especially if we might change how the camera operates, or from perspective to ortho. It would required a lot of code changes. And I don't like reading from the bottom of the code up. Code doesn't execute that way. I want to read from top to bottom.

When reading the transformation sequences in the previous demos from top down the transformation at the top is applied first, the transformation at the bottom is applied last, with the intermediate results method-chained together. (look up above for a reminder)

With a function stack, the function at the top of the stack (f5) is applied first, the result of this is then given as input to f4 (second on the stack), all the way down to f1, which was the first fn to be placed on the stack, and as such, the last to be applied. (Last In First Applied - LIFA)



So, in order to ensure that the functions in a stack will execute in the same order as all of the previous demos, they need to be pushed onto the stack in reverse order.

This means that from model space to world space, we can now read the transformations FROM TOP TO BOTTOM!!!! SUCCESS!

Then, to draw the square relative to paddle one, those six transformations will already be on the stack, therefore only push the differences, and then apply the stack to the paddle's model space data.

19.2 How to Execute

Load src/demo18/demo.py in Spyder and hit the play button

19.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

19.4 Description

Function stack. Internally it has a list, where index 0 is the bottom of the stack. In python we can store any object as a variable, and we will be storing functions which transform a vertex to another vertex, through the "modelspace_to_ndc" method.

Listing 3: src/demo18/demo.py

```
@dataclass
class FunctionStack:
    stack: List[Callable[Vertex, Vertex]] = field(default_factory=lambda: [])

    def push(self, o: object):
        self.stack.append(o)

    def pop(self):
        return self.stack.pop()

    def clear(self):
        self.stack.clear()

    def modelspace_to_ndc(self, vertex: Vertex) -> Vertex:
        v = vertex
        for fn in reversed(self.stack):
            v = fn(v)
        return v
```

(continues on next page)

(continued from previous page)

```


fn_stack = FunctionStack()
```

There is an example at the bottom of src/demo18/demo.py

Listing 4: src/demo18/demo.py

```
def identity(x):
    return x


def add_one(x):
    return x + 1


def multiply_by_2(x):
    return x * 2


def add_5(x):
    return x + 5


```

Define four functions, which we will compose on the stack.

Push identity onto the stack, which will will never pop off of the stack.

Listing 5: src/demo18/demo.py

```
fn_stack.push(identity)
print(fn_stack)
print(fn_stack.modelspace_to_ndc(1))  # x = 1
```

Listing 6: src/demo18/demo.py

```
fn_stack.push(add_one)
print(fn_stack)
print(fn_stack.modelspace_to_ndc(1))  # x + 1 = 2
```

Listing 7: src/demo18/demo.py

```
fn_stack.push(multiply_by_2)  # (x * 2) + 1 = 3
print(fn_stack)
print(fn_stack.modelspace_to_ndc(1))
```

Listing 8: src/demo18/demo.py

```
fn_stack.push(add_5)  # ((x + 5) * 2) + 1 = 13
print(fn_stack)
print(fn_stack.modelspace_to_ndc(1))
```

Listing 9: src/demo18/demo.py

```
554 fn_stack.pop()
555 print(fn_stack)
556 print(fn_stack.modelspace_to_ndc(1))  # (x * 2) + 1 = 3
```

Listing 10: src/demo18/demo.py

```
560 fn_stack.pop()
561 print(fn_stack)
562 print(fn_stack.modelspace_to_ndc(1))  # x + 1 = 2
```

Listing 11: src/demo18/demo.py

```
566 fn_stack.pop()
567 print(fn_stack)
568 print(fn_stack.modelspace_to_ndc(1))  # x = 1
```

19.5 Event Loop

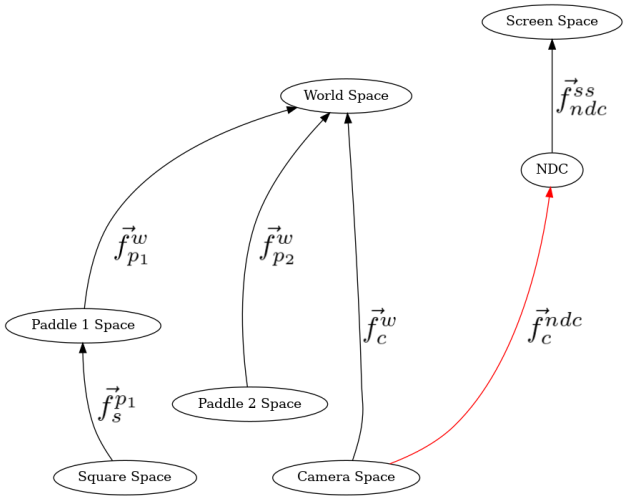
Listing 12: src/demo18/demo.py

```
388 while not glfw.window_should_close(window):
```

```
...
```

In previous demos, camera_space_to_ndc_space_fn was always the last function called in the method chained pipeline. Put it on the bottom of the stack, by pushing it first, so that "modelspace_to_ndc" calls this function last. Each subsequent push will add a new function to the top of the stack.

$$\vec{f}_c^{ndc}$$



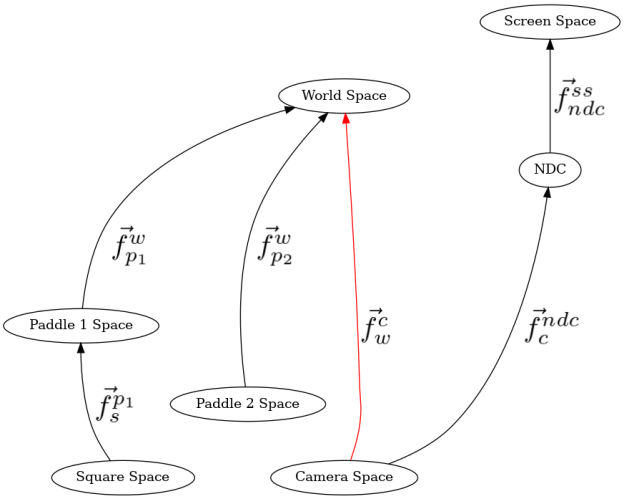
Listing 13: src/demo18/demo.py

```
422     fn_stack.push(lambda v: v.camera_space_to_ndc_space_fn())  # (1)
```

Unlike in previous demos in which we read the transformations from model space to world space backwards; this time because the transformations are on a stack, the fns on the model stack can be read forwards, where each operation translates/rotates/scales the current space

The camera's position and orientation are defined relative to world space like so, read top to bottom:

$$\vec{f}_c^w$$



Listing 14: src/demo18/demo.py

```
426     # fn_stack.push(
427     #     lambda v: v.translate(camera.position_ws)
428     # fn_stack.push(lambda v: v.rotate_y(camera.rot_y))
429     # fn_stack.push(lambda v: v.rotate_x(camera.rot_x))
```

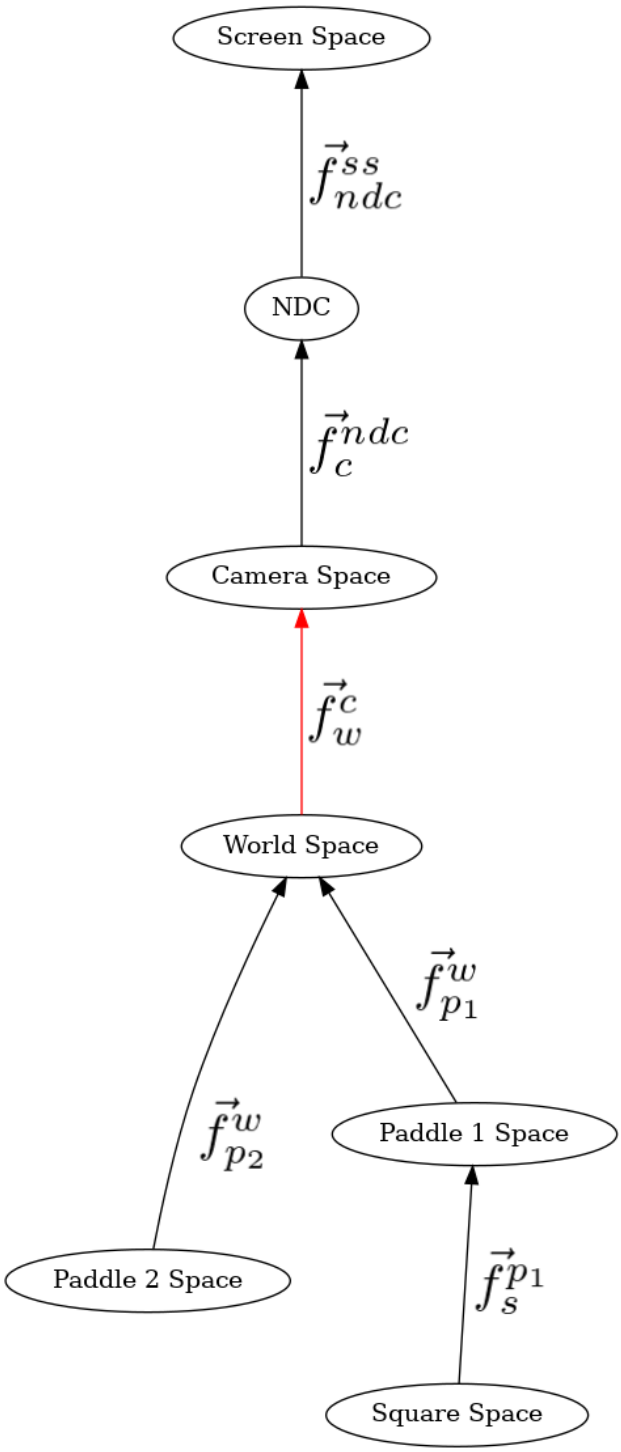
But, since we need to transform world-space to camera space, they must be inverted by reversing the order, and negating the arguments

Therefore the transformations to put the world space into camera space are.

$$\vec{f}_w^c$$

Listing 15: src/demo18/demo.py

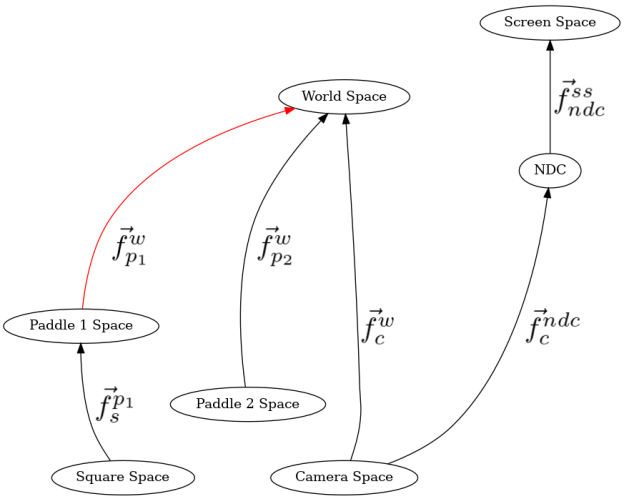
```
433     fn_stack.push(lambda v: v.rotate_x(-camera.rot_x))  # (2)
434     fn_stack.push(lambda v: v.rotate_y(-camera.rot_y))  # (3)
435     fn_stack.push(lambda v: v.translate(-camera.position_ws))  # (4)
```


Screen Space
$\vec{f}^{ss}_{ndc}$
NDC
$\vec{f}^{ndc}_{c}$
Camera Space
$\vec{f}^{c}_{w}$
World Space
$\vec{f}^{w}_{p_1}$
$\vec{f}^{w}_{p_2}$
Paddle 1 Space
Paddle 2 Space
$\vec{f}^{p_1}_{s}$
Square Space

19.5.1 draw paddle 1

Unlike in previous demos in which we read the transformations from model space to world space backwards; because the transformations are on a stack, the fns on the model stack can be read forwards, where each operation translates/rotates/scales the current space

$$\vec{f}^{w}_{p1}$$



Listing 16: src/demo18/demo.py

```
    fn_stack.push(
        lambda v: v.translate(paddle1.position)
    )  # (5) translate the local origin
    fn_stack.push(
        lambda v: v.rotate_z(paddle1.rotation)
    )  # (6) (rotate around the local z axis
```

for each of the modelspace coordinates, apply all of the procedures on the stack from top to bottom this results in coordinate data in NDC space, which we can pass to glVertex3f

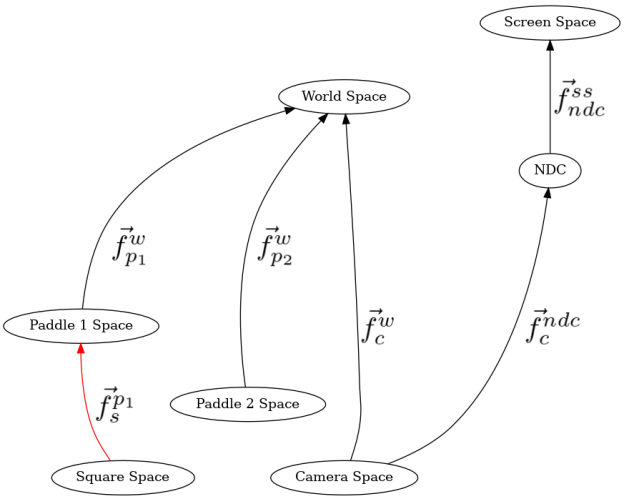
Listing 17: src/demo18/demo.py

```
    glColor3f(paddle1.r, paddle1.g, paddle1.b)

    glBegin(GL_QUADS)
    for paddle1_vertex_ms in paddle1.vertices:
        paddle1_vertex_ndc = fn_stack.modelspace_to_ndc(paddle1_vertex_ms)
        glVertex3f(
            paddle1_vertex_ndc.x,
            paddle1_vertex_ndc.y,
            paddle1_vertex_ndc.z,
        )
    glEnd()
```

19.5.2 draw the square

since the modelstack is already in paddle1's space, and since the blue square is defined relative to paddle1, just add the transformations relative to it before the blue square is drawn. Draw the square, and then remove these 4 transformations from the stack (done below)

$$\vec{f}_s^{p1}$$



Listing 18: src/demo18/demo.py

```
462     glColor3f(0.0, 0.0, 1.0)
463
464     fn_stack.push(lambda v: v.translate(Vertex(x=0.0, y=0.0, z=-1.0)))  # (7)
465     fn_stack.push(lambda v: v.rotate_z(rotation_around_paddle1))  # (8)
466     fn_stack.push(lambda v: v.translate(Vertex(x=2.0, y=0.0, z=0.0)))  # (9)
467     fn_stack.push(lambda v: v.rotate_z(square_rotation))  # (10)
468
469     glBegin(GL_QUADS)
470     for model_space in square:
471         ndc = fn_stack.modelspace_to_ndc(model_space)
472         glVertex3f(ndc.x, ndc.y, ndc.z)
473     glEnd()
```

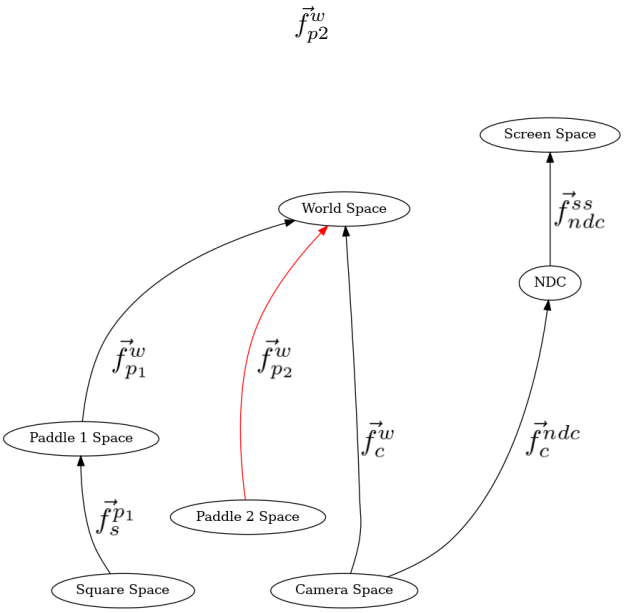
Now we need to remove fns from the stack so that the lambda stack will convert from world space to NDC. This will allow us to just add the transformations from world space to paddle2 space on the stack.

Listing 19: src/demo18/demo.py

```
477     fn_stack.pop()  # pop off (10)
478     fn_stack.pop()  # pop off (9)
479     fn_stack.pop()  # pop off (8)
480     fn_stack.pop()  # pop off (7)
481     fn_stack.pop()  # pop off (6)
482     fn_stack.pop()  # pop off (5)
```

since paddle2's model_space is independent of paddle 1's space, only leave the view and projection fns (1) - (4)

19.5.3 draw paddle 2



Listing 20: src/demo18/demo.py

```
    glColor3f(0.0, 0.0, 1.0)

    fn_stack.push(lambda v: v.translate(Vertex(x=0.0, y=0.0, z=-1.0)))  # (7)
    fn_stack.push(lambda v: v.rotate_z(rotation_around_paddle1))  # (8)
    fn_stack.push(lambda v: v.translate(Vertex(x=2.0, y=0.0, z=0.0)))  # (9)
    fn_stack.push(lambda v: v.rotate_z(square_rotation))  # (10)

    glBegin(GL_QUADS)
    for model_space in square:
        ndc = fn_stack.modelspace_to_ndc(model_space)
        glVertex3f(ndc.x, ndc.y, ndc.z)
    glEnd()
```

remove all fns from the function stack, as the next frame will set them clear makes the list empty, as the list (stack) will be repopulated the next iteration of the event loop.

Listing 21: src/demo18/demo.py

```
    fn_stack.clear()  # done rendering everything, just go ahead and clean 1-6 off of␣
↪the stack
```

Swap buffers and execute another iteration of the event loop

Listing 22: src/demo18/demo.py

```
    glfw.swap_buffers(window)
```

Notice in the above code, adding functions to the stack is creating a shared context for transformations, and before we call "glVertex3f", we always call "modelspace_to_ndc" on the modelspace vertex. In Demo 19, we will be using OpenGL 2.1 matrix stacks. Although we don't have the code for the OpenGL driver, given that you'll see that we pass

modelspace data directly to “glVertex3f”, it should be clear that the OpenGL implementation must fetch the modelspace to NDC transformations from the ModelView and Projection matrix stacks.

CHAPTER

TWENTY

MATRIX STACKS - DEMO 19

20.1 Purpose

Replace lambda stacks with OpenGL 2.1 matrix stacks, provided by the driver for your graphics card. This is how preshader opengl worked.

The function stack allowed us to aggregate the entirety of the transformations from modelspace to NDC, by creating a context of transformations, and a function to do the conversion. We then needed to push or pop functions from the stack, depending on what space transition we were currently dealing with.

A given matrix in OpenGL2.1 is the equivalent of a function stack; given one matrix, it can perform a sequence of transformations from one space to another, with one matrix multiplication.

OpenGL 2.1 deals with two different types of matrices: 1) the projection matrix, which effectively is the function from camera space to NDC (clip space) and 2) The model-view matrix, which deals with the transformations from model space to camera space.

Given that a matrix can perform a sequence of transformations across multiple spaces in the Cayley graph, it may appear that we no longer need any notion of a stack, as we had in the function stacks. But that is not true. For the Model-View matrix, we still need a stack of matrices, so that we can return to a previous transformation sequence. For instance, if we are at world space, the Model-View matrix at the top of the matrix stack will convert from world space to camera space. But we need to draw two relative child spaces to world space, paddle 1 space and paddle 2 space. So before we do the transformations to paddle 1 space, we "push" a copy of the current model-view matrix to the top of the model-view matrix stack, so that after we draw paddle 1 and the square, we can "pop" that matrix off, leaving us the matrix at the top of the model-view matrix stack that represents world space, so that we can then begin to transform to paddle 2 space.

The concepts behind the function stack, in which the first function added to the stack is the last function applied, hold true for matrices as well. But matrices are a much more efficient representation computationally than the function stack, and instead of adding fns and later having to remove them, we can save onto the current frame of reference with a "glPushStack", and restore the saved state by "glPopStack".

Use glPushMatrix and glPopMatrix to save/restore a local coordinate system, that way a tree of objects can be drawn without one child destroying the relative coordinate system of the parent node.

In mvpVisualization/pushmatrix/pushmatrix.py, the grayed out coordinate system is one that has been pushed onto the stack, and it regains its color when it is reactivated by "glPopMatrix"

20.2 How to Execute

Load src/demo19/demo.py in Spyder and hit the play button

20.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

20.4 Description

First thing to note is that we are now using OpenGL 2.1's official transformation procedures, and in the projection transformation, they flip the z axis, making it a left hand coordinate system. The reason for this is long, and I have begun discusses in the "Standard Perspective Matrix" section, but it is an incomplete section for now. But for now, all it means that we have to change how the depth test will be configured, as after the projection transformation, the far z value is 1.0, and the near z value is -1

The clear depth that is set for each fragment each frame is now 1.0, and the test for a given fragment to overwrite the color in the color buffer is changed to be less than or equal to.

20.5 Code

Listing 1: src/demo19/demo.py

```
84 glEnable(GL_DEPTH_TEST)
85 glClearDepth(1.0)
86 glDepthFunc(GL_LEQUAL)
```

20.5.1 The Event Loop

Set the model, view, and projection matrix to the identity matrix. This just means that the functions (currently) will not transform data. In uni-variate terms, f(x) = x

Listing 2: src/demo19/demo.py

```
252     glMatrixMode(GL_PROJECTION)
253     glLoadIdentity()
254     glMatrixMode(GL_MODELVIEW)
255     glLoadIdentity()
```

change the projection matrix to convert the frustum to clip space set the projection matrix to be perspective. Since the viewport is always square, set the aspect ratio to be 1.0. We are now going to clip space instead of to NDC, which we be discussed in the next chapter.

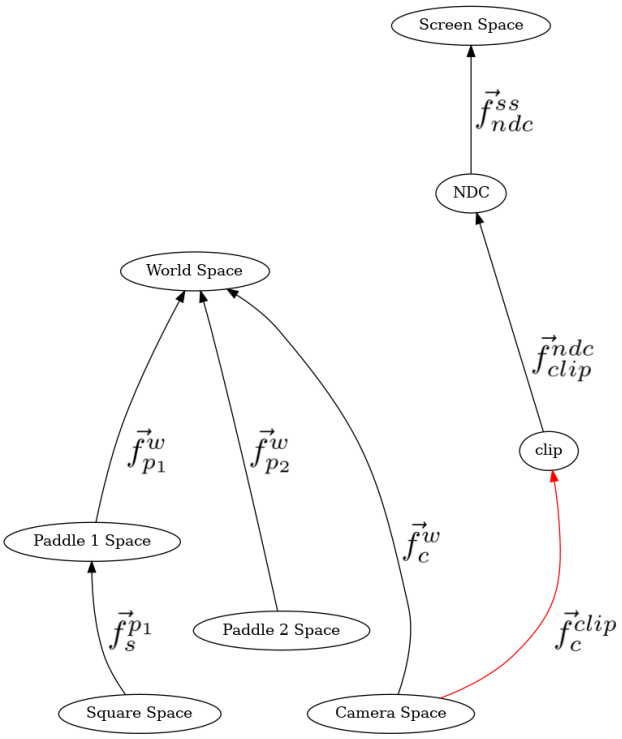


Fig. 1: Turn our NDC into Clip Space

Listing 3: src/demo19/demo.py

```
260     glMatrixMode(GL_PROJECTION)
261     gluPerspective(
```

(continues on next page)

(continued from previous page)

```
        45.0,  # field_of_view
        1.0,  # aspect_ration
        0.1,  # near_z
        1000.0,  # far_z
    )
```

"glMatrixMode" tells the computer which matrix stack should be the active one, against which subsequent matrix operations which affect. In this case, we set the current matrix stack to be the projection one. We then call "gluPerspective" to set the projection transformation, which we covered in previous sections, and we will ignore the implementation of it; it is now a black box.

Now onto camera space!

The camera's position could be described relative to world space by the following sequence of transformations.

```
# glTranslate(camera.x, camera.y, camera.z)
# glRotatef(math.degrees(camera.rot_y), 0.0, 1.0, 0.0)
# glRotatef(math.degrees(camera.rot_x), 1.0, 0.0, 0.0)
```

Therefore, to take the object's world space coordinates and transform them into camera space, we need to do the inverse operations to the view stack.

Listing 4: src/demo19/demo.py

```
    glMatrixMode(GL_MODELVIEW)

    glRotatef(math.degrees(-camera.rot_x), 1.0, 0.0, 0.0)
    glRotatef(math.degrees(-camera.rot_y), 0.0, 1.0, 0.0)
    glTranslate(-camera.x, -camera.y, -camera.z)
```

First thing we did was set the current matrix to be the model-view matrix, instead of the projection matrix. Most of our work will be with the model-view matrix, as looking at the Cayley graph, there's only one function from camera space to NDC. N.B., OpenGL 2.1 transformations use degrees, not radians, so we need to convert to degrees.

Unlike in the lambda stack demo, in which a new function was added to the top of the stack, without modifying any functions below it on the stack, with OpenGL matrices, each transformation, such as translate, rotate, and scale, actually destructively modifies the matrix at the top of the stack, as matrices can be premultiplied together for efficiency.

In linear algebra terms, the matrix multiplication takes place, but then the resulting values of the matrix replace the values of the matrix at the top of the stack.

```
|a b|     |e f|         |ae+bg  af+bh|
|c d|  *  |g h|  =      |ce+dg  cf+dh|
```

This means that rotate_x, rotate_y, translate, etc are destructive operations to the matrix on the top of the stack. Instead of creating new matrix to the top of the stack of matrices, these operations aggregate the transformations, but add no new matrices to the stack, and as such are destructive operations to the current matrix.

But many times we need to hold onto a transformation stack (matrix), so that we can do something else now, and return to this context later, so we have a stack composed of matrices.

This is what glPushMatrix, and glPopMatrix do.

"PushMatrix" describes what the function does, but its purpose is to save onto the current coordinate system for later drawing modelspace data.

The model-view matrix stack is currently the transformation from world space into camera space. Since we now have

to draw paddle 1, the square, and paddle 2, save onto the current model-view stack, to hold onto world space, so that after we draw paddle 1 and the square, we can restore the world space, so that paddle 2 can be drawn relative to it.

Listing 5: src/demo19/demo.py

```
288     glPushMatrix()
```

20.5.2 draw paddle 1

Unlike in previous demos before the lambda stack, because the transformations are now on a stack, the functions on the model stack can be read forwards, where each operation translates/rotates/scales the current space.

glVertex data is specified in modelspace coordinates, but since we loaded the projection matrix and the modelview matrix into OpenGL, glVertex3f will apply those transformations for us[1]!

Listing 6: src/demo19/demo.py

```
292     glColor3f(paddle1.r, paddle1.g, paddle1.b)
293
294     glTranslate(
295         paddle1.position[0],
296         paddle1.position[1],
297         0.0,
298     )
299     glRotatef(math.degrees(paddle1.rotation), 0.0, 0.0, 1.0)
300
301     glBegin(GL_QUADS)
302     for paddle1_vertex_ms in paddle1.vertices:
303         glVertex3f(
304             paddle1_vertex_ms[0],
305             paddle1_vertex_ms[1],
306             paddle1_vertex_ms[2],
307         )
308     glEnd()
```

draw the square relative to paddle 1

Since the modelstack is already in paddle1's space just add the transformations relative to it before paddle 2 is drawn, we need to remove the square's 3 model_space transformations

Listing 7: src/demo19/demo.py

```
314     # draw the square
315     # given that no nodes are defined relative to the square, we do not need
```

(continues on next page)

[1] To note. The computer has no notion of these relative coordinate systems. From the computer's point of view, there is one coordinate system, (x,y,z,w). So now let's look at what the OpenGL drawing calls are in more detail, and think about what they must do. "glBegin(GL_QUADS)", 4 calls to "glVertex", and "glEnd". "glVertex", given modelspace data, must pull 2 matrices from OpenGL's matrix stacks, the modelview and the projection. It then does two transformations to turn the modelspace data in NDC (clip-space, where the x y and z are each divided by the w to get NDC). From this point of view, there are no relative coordinate systems, just numbers in and numbers out. So then what do glBegin and glEnd do? This will be a subject for after the midterm, but the 4 vertices will be converted to screen space, given the matrices, and when "glEnd" is called, the graphics driver will need to determine what pixels are within the quadrilateral or not. For the fragments within the quadrilateral, the driver will also need to know the color of the vertex. So glVertex is doing a lot behind the scenes, grabbing the modelview and the projection matrices from the top of the stack, grabbing the current color, set by "glColor3f".

(continued from previous page)

```
    # to push a matrix.  Here we will do so anyway, just to clarify what is
    # happening
    glPushMatrix()
    # the current model matrix will be copied and then the copy will be
    # pushed onto the model stack
    glColor3f(0.0, 0.0, 1.0)

    # these functions change the current model matrix
    glTranslate(0.0, 0.0, -1.0)
    glRotatef(math.degrees(rotation_around_paddle1), 0.0, 0.0, 1.0)
    glTranslate(2.0, 0.0, 0.0)
    glRotatef(math.degrees(square_rotation), 0.0, 0.0, 1.0)

    glBegin(GL_QUADS)
    for model_space in square_vertices:
        glVertex3f(model_space[0], model_space[1], model_space[2])
    glEnd()
    glPopMatrix()
    # the mode matrix that was on the model stack before the square
    # was drawn will be restored
    glPopMatrix()
```

20.5.3 draw paddle 2

No need to push matrix here, as this is the last object that we are drawing, and upon the next iteration of the event loop, all 3 matrices will be reset to the identity

Listing 8: src/demo19/demo.py

```
    # draw paddle 2.  Nothing is defined relative to paddle to, so we don't
    # need to push matrix, and on the next iteration of the event loop,
    # all matrices will be cleared to identity, so who cares if we
    # mutate the values for now.
    glColor3f(paddle2.r, paddle2.g, paddle2.b)

    glTranslate(
        paddle2.position[0],
        paddle2.position[1],
        0.0,
    )
    glRotatef(math.degrees(paddle2.rotation), 0.0, 0.0, 1.0)

    glBegin(GL_QUADS)
    for paddle2_vertex_ms in paddle2.vertices:
        glVertex3f(
            paddle2_vertex_ms[0],
            paddle2_vertex_ms[1],
            paddle2_vertex_ms[2],
        )
    glEnd()
```

CHAPTER

TWENTYONE

SHADERS - DEMO 20

21.1 Purpose

Learn what "shaders" are, with a brief introduction to how to use them in OpenGL 2.1.

21.2 How to Execute

Load src/demo20/demo.py in Spyder and hit the play button

21.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

21.4 Description

In the previous demo, in the footnote, we found that "glVertex" did a lot behind the scenes, sneakily grabbing data from the matrix stacks and the current color (and lots of other things we haven't covered in this book, like texturing and lighting).

What did the graphics driver do with the data? This older style of graphics programming is called "fixed-function" graphics programming, in that we can tweak some values, but the graphics card and its driver are going to do whatever they were programmed to, and we can't control it. Kind of like having a graphing calculator, but not having the ability to program it - OpenGL is in control at this point, and we can just parameterize it.

Programmers wanted more control, first over how lighting works (which we don't cover in this book). Rather than having 3 or so lighting models to choose from, the programmers wanted to be able to specify the math of how their custom lighting worked. So OpenGL 2.1 allowed the programmer to create "shaders", which I believe were called such because they allow the programmer to change the "shade" of a fragment[1].

In this demo, we keep most of the code from the previous demo the same, we just add in programmable shaders. So the calls to "glVertex" are no longer a black box, in which something happens of the graphics card; instead, before sending modelspace data to the graphics card, we program a sequence of transformations that happen on the graphics card per vertex, which a driver could implement in parallel.

We have some new imports, "glUseProgram", "GL_VERTEX_SHADER", "GL_FRAGMENT_SHADER"

Listing 1: src/demo20/demo.py

```
34 from OpenGL.GL import (
35     GL_COLOR_BUFFER_BIT,
36     GL_DEPTH_BUFFER_BIT,
37     GL_DEPTH_TEST,
38     GL_FRAGMENT_SHADER,
39     GL_LEQUAL,
40     GL_MODELVIEW,
41     GL_PROJECTION,
42     GL_QUADS,
43     GL_SCISSOR_TEST,
44     GL_VERTEX_SHADER,
45     glBegin,
46     glClear,
47     glClearColor,
48     glClearDepth,
49     glColor3f,
50     glDepthFunc,
51     glDisable,
52     glEnable,
53     glEnd,
54     glLoadIdentity,
55     glMatrixMode,
56     glPopMatrix,
57     glPushMatrix,
58     glRotatef,
59     glScissor,
60     glTranslate,
```

(continues on next page)

[1] To note. The word "shader" now just means a small function the executes on the GPU. Taylor Swift would not write a song saying "A Shader's gonna shade shade shade shade", because shaders do all sorts of things now, like transformations of vertices from modelspace to clip-space, or adding new geometry in the middle of the graphic transformation pipeline.

(continued from previous page)

```
    glUseProgram,
    glVertex3f,
    glViewport,
)
from OpenGL.GLU import gluPerspective

```

We now ask the OpenGL driver to accept OpenGL 2.1 function calls.

Listing 2: src/demo20/demo.py

```
glfw.window_hint(glfw.CONTEXT_VERSION_MAJOR, 2)
glfw.window_hint(glfw.CONTEXT_VERSION_MINOR, 1)
```

Before the event loop starts, we send two mini programs to the GPU, the vertex shader and the fragment shader. The purpose of the vertex shader is to transform the modelspace data to clip-space (x,y,z,w), where to get NDC you would divide by the w; i.e. (x/w,y/w,z/w,1)

I like to think of a matrix multiplication to a vector as a "function-application", in the same way we did in the lambda stack.

Listing 3: src/demo20/demo.py

```
# compile shaders

# initialize shaders
pwd = os.path.dirname(os.path.abspath(__file__))

with open(os.path.join(pwd, "triangle.vert"), "r") as f:
    vs = shaders.compileShader(f.read(), GL_VERTEX_SHADER)

with open(os.path.join(pwd, "triangle.frag"), "r") as f:
    fs = shaders.compileShader(f.read(), GL_FRAGMENT_SHADER)

shader = shaders.compileProgram(vs, fs)
glUseProgram(shader)
```

In the vertex shader, there are a lot of predefined variable names with values provided to us. "gl_Modelview_matrix" is the matrix at the top of the modelview matrix stack at the time "glVertex" is called, "glProjectionMatrix" is the top of the projection matrix stack (although there is typically only one). "glColor" is the color that was defined for this vertex by calling "glColor3f".

In OpenGL 2.1, the output of the vertex shader that we use here is "glFrontColor", which is a variable name that we must use to get the data to the fragment shader, which we haven't covered. (Actually there are a bunch of other predefined variable outputs, including "glBackColor", for the case that we are looking at the back face of the geometry.

```
//Copyright (c) 2018-2024 William Emerison Six
//
//Permission is hereby granted, free of charge, to any person obtaining a copy
//of this software and associated documentation files (the "Software"), to deal
//in the Software without restriction, including without limitation the rights
//to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//copies of the Software, and to permit persons to whom the Software is
//furnished to do so, subject to the following conditions:
```

(continues on next page)

(continued from previous page)

```
//
//The above copyright notice and this permission notice shall be included in all
//copies or substantial portions of the Software.
//
//THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
//IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
//AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//SOFTWARE.

#version 120

void main(void)
{
    // gl_Vertex is modelspace data

    vec4 camera_space = gl_ModelViewMatrix * gl_Vertex;
    // in camera space, the frustum is on the negative z axis
    vec4 clip_space = gl_ProjectionMatrix * camera_space;
    vec3 ndc_space = clip_space.xyz / clip_space.w;
    // in ndc space, x y and z need to be divided by w

    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Copy the primary color
    gl_FrontColor = gl_Color;
}
```

This is the fragment shader. Unlike the vertex shader, which took an vertex in (x,y,z) and outputted clip-space (NDC), the fragment shader for a given pixel is determining the color[2]. Is "glColor" in the fragment shader the same "glColor" from the vertex shader, or is it the value of the vertex shader's output "glFrontColor"? Honestly, the author doesn't know, but it is set to the output of the fragment shader, and such questions are a moot point once we get to OpenGL 3.3 Core profile in the next section.

```
//Copyright (c) 2018-2024 William Emerison Six
//
//Permission is hereby granted, free of charge, to any person obtaining a copy
//of this software and associated documentation files (the "Software"), to deal
//in the Software without restriction, including without limitation the rights
//to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//copies of the Software, and to permit persons to whom the Software is
//furnished to do so, subject to the following conditions:
//
//The above copyright notice and this permission notice shall be included in all
//copies or substantial portions of the Software.
//
//THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
```

(continues on next page)

[2] We are not yet concerned about how the rasterization process works, and the linear interpolation of color, normals, etc. Online resources like "LearnOpenGL" and "PortableGL" cover this.

(continued from previous page)

```
//IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
//AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//SOFTWARE.

#version 120

void main(void)
{
    gl_FragColor = gl_Color;
}
```

The explicit mapping of variable names to values in OpenGL 3.3 Core Profile will make the flow of data from the CPU program, to the GPU vertex shader, to the fragment shader much more clear, albeit at the expense of verbosity.

CHAPTER

TWENTYTWO

OPENGL3.3 CORE PROFILE - DEMO 21

22.1 Purpose

In OpenGL 3.3 Core Profile, shaders are no longer optional, they are mandatory.

22.2 How to Execute

Load src/demo21/demo.py in Spyder and hit the play button

22.3 Move the Paddles using the Keyboard

Keyboard Input	Action
w	Move Left Paddle Up
s	Move Left Paddle Down
k	Move Right Paddle Down
i	Move Right Paddle Up
d	Increase Left Paddle's Rotation
a	Decrease Left Paddle's Rotation
l	Increase Right Paddle's Rotation
j	Decrease Right Paddle's Rotation
UP	Move the camera up, moving the objects down
DOWN	Move the camera down, moving the objects up
LEFT	Move the camera left, moving the objects right
RIGHT	Move the camera right, moving the objects left
q	Rotate the square around its center
e	Rotate the square around paddle 1's center

22.4 Description

```
# Copyright (c) 2018-2024 William Emerison Six
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

from __future__ import annotations  # to appease Python 3.7-3.9

import ctypes
import math
import os
import sys
from dataclasses import dataclass, field

import glfw
import imgui
import numpy as np

# new - SHADERS
import OpenGL.GL.shaders as shaders
import pyMatrixStack as ms
import staticlocal
from imgui.integrations.glfw import GlfwRenderer
from OpenGL.GL import (
    GL_ARRAY_BUFFER,
    GL_BLEND,
    GL_COLOR_BUFFER_BIT,
    GL_DEPTH_BUFFER_BIT,
    GL_DEPTH_TEST,
    GL_FLOAT,
    GL_FRAGMENT_SHADER,
    GL_LEQUAL,
    GL_LINES,
    GL_ONE_MINUS_SRC_ALPHA,
    GL_SRC_ALPHA,
    GL_STATIC_DRAW,
```

(continues on next page)

(continued from previous page)

```
    GL_TRIANGLES,
    GL_TRUE,
    GL_VERTEX_SHADER,
    glBindBuffer,
    glBindVertexArray,
    glBlendFunc,
    glBufferData,
    glClear,
    glClearColor,
    glClearDepth,
    glDeleteBuffers,
    glDeleteProgram,
    glDeleteVertexArrays,
    glDepthFunc,
    glDisable,
    glDrawArrays,
    glEnable,
    glEnableVertexAttribArray,
    glGenBuffers,
    glGenVertexArrays,
    glGetAttribLocation,
    glGetUniformLocation,
    glUniformMatrix4fv,
    glUseProgram,
    glVertexAttribPointer,
    glViewport,
)

if not glfw.init():
    sys.exit()

# NEW - for shader location
pwd = os.path.dirname(os.path.abspath(__file__))

# NEW - for shaders
glfloat_size = 4
floatsPerVertex = 3
floatsPerColor = 4


glfw.window_hint(glfw.CONTEXT_VERSION_MAJOR, 3)
glfw.window_hint(glfw.CONTEXT_VERSION_MINOR, 3)
# CORE profile means no fixed functions.
# compatibility profile would mean access to legacy fixed functions
# compatibility mode isn't supported by every graphics driver,
# particulary on laptops which switch between integrated graphics
# and a discrete card over time based off of usage.
glfw.window_hint(glfw.OPENGL_PROFILE, glfw.OPENGL_CORE_PROFILE)
# for osx
glfw.window_hint(glfw.OPENGL_FORWARD_COMPAT, GL_TRUE)

imgui.create_context()
```

(continues on next page)

(continued from previous page)

```
window = glfw.create_window(500, 500, "ModelViewProjection Demo 21 ", None, None)
if not window:
    glfw.terminate()
    sys.exit()

# Make the window's context current
glfw.make_context_current(window)


impl = GlfwRenderer(window)

# Install a key handler


def on_key(win, key, scancode, action, mods):
    if key == glfw.KEY_ESCAPE and action == glfw.PRESS:
        glfw.set_window_should_close(win, 1)


glfw.set_key_callback(window, on_key)

glClearColor(0.0289, 0.071875, 0.0972, 1.0)


glClearDepth(1.0)
glDepthFunc(GL_LEQUAL)
glEnable(GL_DEPTH_TEST)

__enable_blend__ = True
if __enable_blend__:
    glEnable(GL_BLEND)
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)


@dataclass
class Paddle:
    r: float
    g: float
    b: float
    position: any
    rotation: float = 0.0
    vertices: np.array = field(
        default_factory=lambda: np.array(
            [
                [-1.0, -3.0, 0.0],
                [1.0, -3.0, 0.0],
                [1.0, 3.0, 0.0],
                [1.0, 3.0, 0.0],
                [-1.0, 3.0, 0.0],
                [-1.0, -3.0, 0.0],
            ],
            dtype=np.float32,
```

(continues on next page)

(continued from previous page)

```
        )
    )

    vao: int = 0
    vbo: int = 0
    shader: int = 0

    def prepare_to_render(self):
        # GL_QUADS aren't available anymore, only triangles
        # need 6 vertices instead of 4
        vertices = self.vertices
        self.number_of_vertices = np.size(vertices) // floatsPerVertex
        # fmt: off
        color = np.array(
            [
                self.r, self.g, self.b, .75,
                self.r, self.g, self.b, .75,
                self.r, self.g, self.b, .75,
                self.r, self.g, self.b, .75,
                self.r, self.g, self.b, .75,
                self.r, self.g, self.b, .75,
            ],
            dtype=np.float32,
        )
        # fmt: on

        self.number_of_colors = np.size(color) // floatsPerColor

        self.vao = glGenVertexArrays(1)
        glBindVertexArray(self.vao)

        # initialize shaders

        with open(os.path.join(pwd, "triangle.vert"), "r") as f:
            vs = shaders.compileShader(f.read(), GL_VERTEX_SHADER)

        with open(os.path.join(pwd, "triangle.frag"), "r") as f:
            fs = shaders.compileShader(f.read(), GL_FRAGMENT_SHADER)

        self.shader = shaders.compileProgram(vs, fs)

        # send the modelspace data to the GPU
        self.vbo = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.vbo)

        position = glGetAttribLocation(self.shader, "position")
        glEnableVertexAttribArray(position)

        glVertexAttribPointer(
            position, floatsPerVertex, GL_FLOAT, False, 0, ctypes.c_void_p(0)
        )
```

(continues on next page)

(continued from previous page)

```
        glBufferData(
            GL_ARRAY_BUFFER, glfloat_size * np.size(vertices), vertices, GL_STATIC_DRAW
        )

        # send the modelspace data to the GPU
        vbo_color = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, vbo_color)

        color_attrib_loc = glGetAttribLocation(self.shader, "color_in")
        glEnableVertexAttribArray(color_attrib_loc)
        glVertexAttribPointer(
            color_attrib_loc, floatsPerColor, GL_FLOAT, False, 0, ctypes.c_void_p(0)
        )

        glBufferData(
            GL_ARRAY_BUFFER, glfloat_size * np.size(color), color, GL_STATIC_DRAW
        )

        # reset VAO/VBO to default
        glBindVertexArray(0)
        glBindBuffer(GL_ARRAY_BUFFER, 0)

    # destructor
    def __del__(self):
        glDeleteVertexArrays(1, [self.vao])
        glDeleteBuffers(1, [self.vbo])
        glDeleteProgram(self.shader)

    def render(self):
        glUseProgram(self.shader)
        glBindVertexArray(self.vao)

        mvp_matrix_loc = glGetUniformLocation(self.shader, "mvpMatrix")
        # ascontiguousarray puts the array in column major order
        glUniformMatrix4fv(
            mvp_matrix_loc,
            1,
            GL_TRUE,
            np.ascontiguousarray(
                ms.get_current_matrix(ms.MatrixStack.modelviewprojection),
                dtype=np.float32,
            ),
        )
        glDrawArrays(GL_TRIANGLES, 0, self.number_of_vertices)
        glBindVertexArray(0)


paddle1 = Paddle(r=0.578123, g=0.0, b=1.0, position=np.array([-9.0, 0.0, 0.0]))
paddle1.prepare_to_render()
paddle2 = Paddle(r=1.0, g=1.0, b=0.0, position=np.array([9.0, 0.0, 0.0]))
paddle2.prepare_to_render()
```

(continues on next page)

(continued from previous page)

```
@dataclass
class Square(Paddle):
    rotation_around_paddle1: float = 0.0
    vertices: np.array = field(
        default_factory=lambda: np.array(
            [
                [-0.5, -0.5, 0.0],
                [0.5, -0.5, 0.0],
                [0.5, 0.5, 0.0],
                [0.5, 0.5, 0.0],
                [-0.5, 0.5, 0.0],
                [-0.5, -0.5, 0.0],
            ],
            dtype=np.float32,
        )
    )


square = Square(r=0.0, g=0.0, b=1.0, position=[0.0, 0.0, 0.0])

square.prepare_to_render()

number_of_controllers = glfw.joystick_present(glfw.JOYSTICK_1)


test_bool = False
test_float = 0.0


@dataclass
class Camera:
    x: float = 0.0
    y: float = 0.0
    z: float = 0.0
    rot_y: float = 0.0
    rot_x: float = 0.0


camera = Camera(x=0.0, y=0.0, z=40.0, rot_y=0.0, rot_x=0.0)


class Ground:
    def __init__(self):
        pass

    def vertices(self):
        # glColor3f(0.1,.1,.1)
        verts = []
        for x in range(-600, 601, 20):
            for z in range(-600, 601, 20):
                verts.append(float(-x))
```

(continues on next page)

(continued from previous page)

```
                verts.append(float(-5.0))
                verts.append(float(z))
                verts.append(float(x))
                verts.append(float(-5.0))
                verts.append(float(z))
                verts.append(float(x))
                verts.append(float(-5.0))
                verts.append(float(-z))
                verts.append(float(x))
                verts.append(float(-5.0))
                verts.append(float(z))

        return np.array(verts, dtype=np.float32)

    def prepare_to_render(self):
        # GL_QUADS aren't available anymore, only triangles
        # need 6 vertices instead of 4
        vertices = self.vertices()
        self.number_of_vertices = np.size(vertices) // floatsPerVertex

        self.vao = glGenVertexArrays(1)
        glBindVertexArray(self.vao)

        # initialize shaders

        with open(os.path.join(pwd, "ground.vert"), "r") as f:
            vs = shaders.compileShader(f.read(), GL_VERTEX_SHADER)

        with open(os.path.join(pwd, "ground.frag"), "r") as f:
            fs = shaders.compileShader(f.read(), GL_FRAGMENT_SHADER)

        self.shader = shaders.compileProgram(vs, fs)

        # send the modelspace data to the GPU
        self.vbo = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.vbo)

        position = glGetAttribLocation(self.shader, "position")
        glEnableVertexAttribArray(position)

        glVertexAttribPointer(
            position, floatsPerVertex, GL_FLOAT, False, 0, ctypes.c_void_p(0)
        )

        glBufferData(
            GL_ARRAY_BUFFER, glfloat_size * np.size(vertices), vertices, GL_STATIC_DRAW
        )

        # send the modelspace data to the GPU
        # TODO, send color to the shader

        # reset VAO/VBO to default
```

(continues on next page)

(continued from previous page)

```
        glBindVertexArray(0)
        glBindBuffer(GL_ARRAY_BUFFER, 0)

    # destructor
    def __del__(self):
        glDeleteVertexArrays(1, [self.vao])
        glDeleteBuffers(1, [self.vbo])
        glDeleteProgram(self.shader)

    def render(self):
        glUseProgram(self.shader)
        glBindVertexArray(self.vao)

        # pass projection parameters to the shader
        mvp_matrix_loc = glGetUniformLocation(self.shader, "mvpMatrix")
        # ascontiguousarray puts the array in column major order
        glUniformMatrix4fv(
            mvp_matrix_loc,
            1,
            GL_TRUE,
            np.ascontiguousarray(
                ms.get_current_matrix(ms.MatrixStack.modelviewprojection),
                dtype=np.float32,
            ),
        )
        glDrawArrays(GL_LINES, 0, self.number_of_vertices)
        glBindVertexArray(0)


ground = Ground()
ground.prepare_to_render()


def handle_inputs():
    if glfw.get_key(window, glfw.KEY_E) == glfw.PRESS:
        square.rotation_around_paddle1 += 0.1

    if glfw.get_key(window, glfw.KEY_Q) == glfw.PRESS:
        square.rotation += 0.1

    global camera

    move_multiple = 1.0
    if glfw.get_key(window, glfw.KEY_RIGHT) == glfw.PRESS:
        camera.rot_y -= 0.03
    if glfw.get_key(window, glfw.KEY_LEFT) == glfw.PRESS:
        camera.rot_y += 0.03
    if glfw.get_key(window, glfw.KEY_PAGE_UP) == glfw.PRESS:
        camera.rot_x += 0.03
    if glfw.get_key(window, glfw.KEY_PAGE_DOWN) == glfw.PRESS:
        camera.rot_x -= 0.03
    # //TODO -  explain movement on XZ-plane
```

(continues on next page)

(continued from previous page)

```
    # //TODO -  show camera movement in graphviz
    if glfw.get_key(window, glfw.KEY_UP) == glfw.PRESS:
        camera.x -= move_multiple * math.sin(camera.rot_y)
        camera.z -= move_multiple * math.cos(camera.rot_y)
    if glfw.get_key(window, glfw.KEY_DOWN) == glfw.PRESS:
        camera.x += move_multiple * math.sin(camera.rot_y)
        camera.z += move_multiple * math.cos(camera.rot_y)

    global paddle1, paddle2

    if glfw.get_key(window, glfw.KEY_S) == glfw.PRESS:
        paddle1.position[1] -= 1.0
    if glfw.get_key(window, glfw.KEY_W) == glfw.PRESS:
        paddle1.position[1] += 1.0
    if glfw.get_key(window, glfw.KEY_K) == glfw.PRESS:
        paddle2.position[1] -= 1.0
    if glfw.get_key(window, glfw.KEY_I) == glfw.PRESS:
        paddle2.position[1] += 1.0

    if glfw.get_key(window, glfw.KEY_A) == glfw.PRESS:
        paddle1.rotation += 0.1
    if glfw.get_key(window, glfw.KEY_D) == glfw.PRESS:
        paddle1.rotation -= 0.1
    if glfw.get_key(window, glfw.KEY_J) == glfw.PRESS:
        paddle2.rotation += 0.1
    if glfw.get_key(window, glfw.KEY_L) == glfw.PRESS:
        paddle2.rotation -= 0.1


# fmt: off
# square_vertices = np.array(
#     [[-5.0, -5.0, 0.0],
#      [5.0, -5.0, 0.0],
#      [5.0, 5.0, 0.0],
#      [-5.0, 5.0, 0.0]],
#     dtype=np.float32,
# )
# fmt: on


TARGET_FRAMERATE = 60  # fps

# to try to standardize on 60 fps, compare times between frames
time_at_beginning_of_previous_frame = glfw.get_time()

# Loop until the user closes the window
while not glfw.window_should_close(window):
    # poll the time to try to get a constant framerate
    while (
        glfw.get_time() < time_at_beginning_of_previous_frame + 1.0 / TARGET_FRAMERATE
    ):
        pass
```

(continues on next page)

(continued from previous page)

```
    # set for comparison on the next frame
    time_at_beginning_of_previous_frame = glfw.get_time()

    # Poll for and process events
    glfw.poll_events()
    impl.process_inputs()

    imgui.new_frame()

    if imgui.begin_main_menu_bar():
        if imgui.begin_menu("File", True):
            clicked_quit, selected_quit = imgui.menu_item("Quit", "Cmd+Q", False, True)

            if clicked_quit:
                exit(1)

            imgui.end_menu()
        imgui.end_main_menu_bar()

    imgui.begin("Custom window", True)

    changed, __enable_blend__ = imgui.checkbox(label="Blend", state=__enable_blend__)

    if changed:
        if __enable_blend__:
            glEnable(GL_BLEND)
        else:
            glDisable(GL_BLEND)

    imgui.text("Bar")
    imgui.text_colored("Eggs", 0.2, 1.0, 0.0)

    # use static local instead of try: except
    # normally you would pass the present function name to staticlocal.var
    # , but since we are not in a function, pass the current module
    staticlocal.var(sys.modules[__name__], test_bool=True, test_float=1.0)
    clicked_test_bool, test_bool = imgui.checkbox("test_bool", test_bool)
    clicked_test_float, test_float = imgui.slider_float("float", test_float, 0.0, 1.0)

    imgui.end()

    width, height = glfw.get_framebuffer_size(window)
    glViewport(0, 0, width, height)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    ms.set_to_identity_matrix(ms.MatrixStack.model)
    ms.set_to_identity_matrix(ms.MatrixStack.view)
    ms.set_to_identity_matrix(ms.MatrixStack.projection)

    # set the projection matrix to be perspective
    ms.perspective(
        field_of_view=45.0,
```

(continues on next page)

(continued from previous page)

```
        aspect_ratio=float(width) / float(height),
        near_z=0.1,
        far_z=1000.0,
    )

    # render scene
    width, height = glfw.get_framebuffer_size(window)
    glViewport(0, 0, width, height)
    glClearColor(0.0289, 0.071875, 0.0972, 1.0)  # r  # g  # b  # a
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    handle_inputs()

    axes_list = glfw.get_joystick_axes(glfw.JOYSTICK_1)
    if len(axes_list) >= 1 and axes_list[0]:
        if math.fabs(float(axes_list[0][0])) > 0.1:
            camera.x += 1.0 * axes_list[0][0] * math.cos(camera.rot_y)
            camera.z -= 1.0 * axes_list[0][0] * math.sin(camera.rot_y)
        if math.fabs(float(axes_list[0][1])) > 0.1:
            camera.x += 1.0 * axes_list[0][1] * math.sin(camera.rot_y)
            camera.z += 1.0 * axes_list[0][1] * math.cos(camera.rot_y)

        # print(axes_list[0][4])
        if math.fabs(axes_list[0][3]) > 0.10:
            camera.rot_x -= 3.0 * axes_list[0][3] * 0.01
        if math.fabs(axes_list[0][2]) > 0.10:
            camera.rot_y -= 3.0 * axes_list[0][2] * 0.01

    # note - opengl matrices use degrees
    ms.rotate_x(ms.MatrixStack.view, -camera.rot_x)
    ms.rotate_y(ms.MatrixStack.view, -camera.rot_y)
    ms.translate(ms.MatrixStack.view, -camera.x, -camera.y, -camera.z)

    ground.render()

    with ms.push_matrix(ms.MatrixStack.model):
        # draw paddle 1
        # Unlike in previous demos, because the transformations
        # are on a stack, the fns on the model stack can
        # be read forwards, where each operation translates/rotates/scales
        # the current space
        ms.translate(
            ms.MatrixStack.model,
            paddle1.position[0],
            paddle1.position[1],
            0.0,
        )
        ms.rotate_z(ms.MatrixStack.model, paddle1.rotation)
        paddle1.render()

        with ms.push_matrix(ms.MatrixStack.model):
            # # draw the square
```

(continues on next page)

(continued from previous page)

```
            # since the modelstack is already in paddle1's space
            # just add the transformations relative to it
            # before paddle 2 is drawn, we need to remove
            # the square's 3 model_space transformations

            ms.translate(ms.MatrixStack.model, 0.0, 0.0, -1.0)
            ms.rotate_z(ms.MatrixStack.model, square.rotation_around_paddle1)

            ms.translate(ms.MatrixStack.model, 2.0, 0.0, 0.0)
            ms.rotate_z(ms.MatrixStack.model, square.rotation)

            square.render()
        # back to padde 1 space
    # get back to center of global space

    with ms.push_matrix(ms.MatrixStack.model):
        # draw paddle 2

        ms.translate(
            ms.MatrixStack.model,
            paddle2.position[0],
            paddle2.position[1],
            0.0,
        )
        ms.rotate_z(ms.MatrixStack.model, paddle2.rotation)
        paddle2.render()

    imgui.render()
    impl.render(imgui.get_draw_data())
    # done with frame, flush and swap buffers
    # Swap front and back buffers
    glfw.swap_buffers(window)


glfw.terminate()
```

```
//Copyright (c) 2018-2024 William Emerison Six
//
//Permission is hereby granted, free of charge, to any person obtaining a copy
//of this software and associated documentation files (the "Software"), to deal
//in the Software without restriction, including without limitation the rights
//to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//copies of the Software, and to permit persons to whom the Software is
//furnished to do so, subject to the following conditions:
//
//The above copyright notice and this permission notice shall be included in all
//copies or substantial portions of the Software.
//
//THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
//IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
```

(continues on next page)

(continued from previous page)

```
//AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//SOFTWARE.

#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec4 color_in;

uniform mat4 mvpMatrix;

out VS_OUT {
  vec4 color;
} vs_out;


void main()
{
   gl_Position = mvpMatrix * vec4(position,1.0);
   vs_out.color = color_in;
}
```

```
//Copyright (c) 2018-2024 William Emerison Six
//
//Permission is hereby granted, free of charge, to any person obtaining a copy
//of this software and associated documentation files (the "Software"), to deal
//in the Software without restriction, including without limitation the rights
//to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//copies of the Software, and to permit persons to whom the Software is
//furnished to do so, subject to the following conditions:
//
//The above copyright notice and this permission notice shall be included in all
//copies or substantial portions of the Software.
//
//THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
//IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
//AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//SOFTWARE.

#version 330 core

out vec4 color;

in VS_OUT {
  vec4 color;
} fs_in;

void main()
```

(continues on next page)

(continued from previous page)

```
{
   color = fs_in.color;
}
```

```
//Copyright (c) 2018-2024 William Emerison Six
//
//Permission is hereby granted, free of charge, to any person obtaining a copy
//of this software and associated documentation files (the "Software"), to deal
//in the Software without restriction, including without limitation the rights
//to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//copies of the Software, and to permit persons to whom the Software is
//furnished to do so, subject to the following conditions:
//
//The above copyright notice and this permission notice shall be included in all
//copies or substantial portions of the Software.
//
//THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
//IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
//AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//SOFTWARE.

#version 330 core

layout (location = 0) in vec3 position;

uniform mat4 mvpMatrix;

out VS_OUT {
  vec4 color;
} vs_out;

void main()
{
   gl_Position = mvpMatrix * vec4(position,1.0);
   vs_out.color = vec4(.5,.5,.5,1.0);
}
```

```
//Copyright (c) 2018-2024 William Emerison Six
//
//Permission is hereby granted, free of charge, to any person obtaining a copy
//of this software and associated documentation files (the "Software"), to deal
//in the Software without restriction, including without limitation the rights
//to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//copies of the Software, and to permit persons to whom the Software is
//furnished to do so, subject to the following conditions:
//
//The above copyright notice and this permission notice shall be included in all
//copies or substantial portions of the Software.
```

(continues on next page)

(continued from previous page)

```
//
//THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
//IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
//AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//SOFTWARE.


#version 330 core

out vec4 color;

in VS_OUT {
  vec4 color;
} fs_in;

void main()
{
   color = fs_in.color;
}
```

22.5 Code

22.5.1 The Event Loop

CHAPTER

TWENTYTHREE

STANDARD PERSPECTIVE MATRIX

23.1 Purpose

Derive the standard perspective matrix that OpenGL expects.

23.2 Description

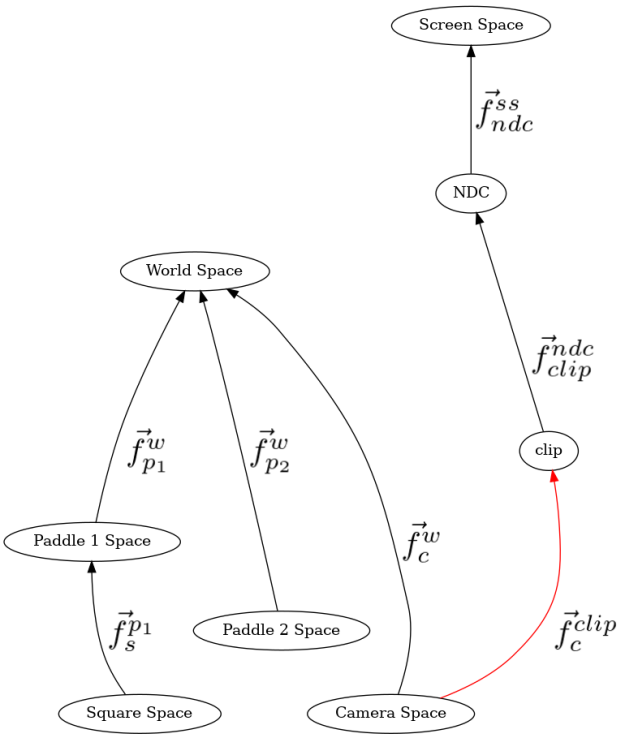
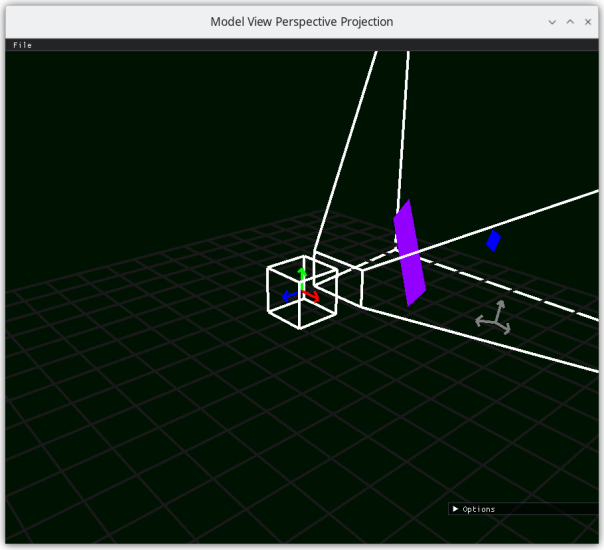


Fig. 1: Turn our NDC into Clip Space

23.2.1 Matrix form of perspective projection



Scale Camera-space x by Camera-space z

$$\vec{f_1}(\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}; nearZ_c) = \begin{bmatrix} \frac{nearZ_c}{z_c} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0, & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

resulting in

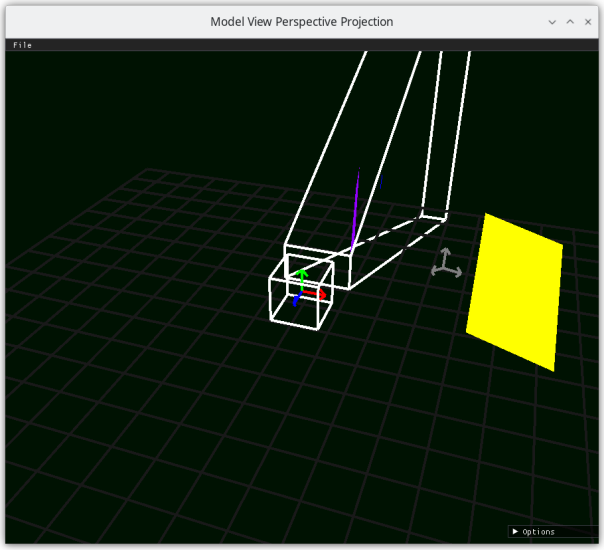


Fig. 2: Scale the camera space x by the camera space z

Scale Camera-space y by Camera-space z

$$(\vec{f_2}; nearZ_c)(\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{nearZ_c}{z_c} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

resulting in

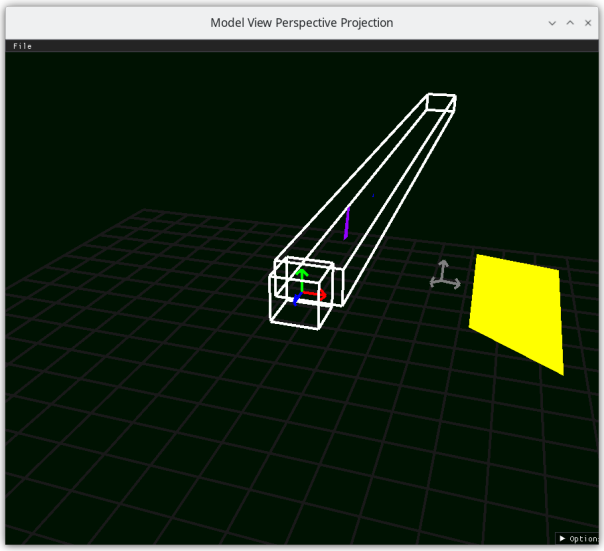


Fig. 3: Scale the camera space y by the cameraspace z

Translate Rectangular Prism's Center to Center

$x_{midpoint} = 0$ // centered on x

$y_{midpoint} = 0$ // centered on y

$z_{midpoint} = \frac{farZ_c + nearZ_c}{2}$;

$$(\vec{f_3}; nearZ_c)(\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{farZ_c + nearZ_c}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

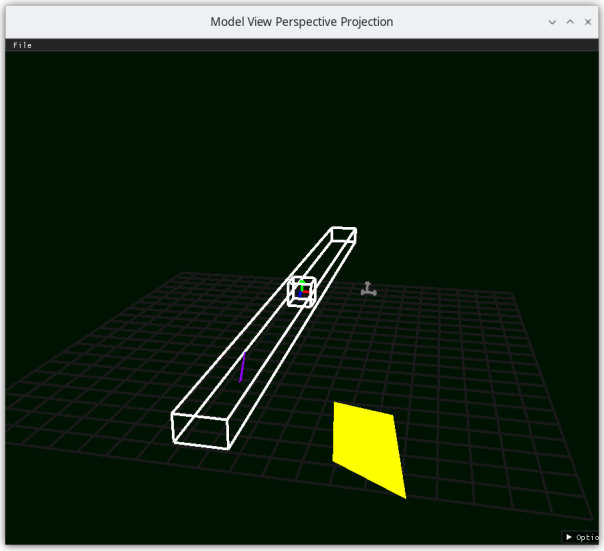
Scale by inverse of the dimensions of the Rectangular Prism

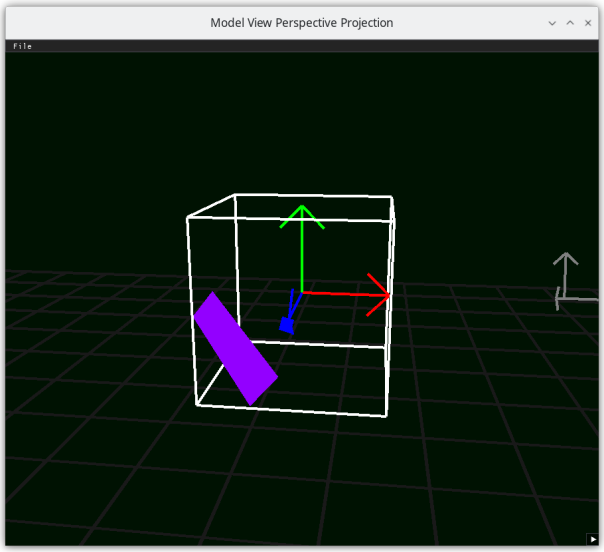
$x_{length} = right * 2$;

$y_{length} = top * 2$;

$z_{length} = nearZ_c - farZ_c$;

$$(\vec{f_4}; nearZ_c, farZ_c)(\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}) = \begin{bmatrix} \frac{1}{right} & 0 & 0 & 0 \\ 0 & \frac{1}{top} & 0 & 0 \\ 0 & 0 & \frac{2}{nearZ_c - farZ_c} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$





Pre-multiply the matricies

$$\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} = 1 \end{bmatrix} = (\vec{f_4} \circ \vec{f_3} \circ \vec{f_2} \circ \vec{f_1}; farZ_c, nearZ_c, top, right) \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

Multiply them all together to get the following. The elements of this premultiplied matrix have no geometric meaning to the author, and that's ok. The matricies above all of geometric meaning, and we premultiply them together for computational efficiency, as well as being able to do the next step in clip space, which we couldn't do without having

the premultiplied matrix.

$$\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} = 1 \end{bmatrix} = \vec{f}_c^{ndc}(\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix} ; farZ_c, nearZ_c, top, right) = \begin{bmatrix} \frac{nearZ_c}{right * z_c} & 0 & 0 & 0 \\ 0 & \frac{nearZ_c}{top * z_c} & 0 & 0 \\ 0 & 0 & \frac{2}{nearZ_c - farZ_c} & \frac{-(farZ_c + nearZ_c)}{nearZ_c - farZ_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

As a quick smoke test to ensure that the aggregate matrix works correctly, let's test the bounds of the frustum and make sure that they map to the NDC cube.

Given that $nearZ_c$ is negative, assuming z_c is equal to $nearZ_c$, $right$ goes to 1, $left$ which is $-right$ goes to -1.

Given that $nearZ_c$ is negative, assuming z_c is equal to $nearZ_c$, top goes to 1, $bottom$ which is $-top$ goes to -1.

Given that w_c is 1, if $z_c = nearZ_c$, $z_{ndc} = 1$. Given that w_c is 1, if $z_c = farZ_c$, $z_{ndc} = -1$.

23.2.2 Clip Space

convert the data from NDC to clip-space.

We have never used clip-space in the class, only NDC, because 4D space is confusing geometrically, nevermind the fact that (NDCx NDCy NDCz) = (Clipx/Clipw, Clipy/Clipy, Clipz/Clipz)

The purpose of going to clip space is that eventually we will be able to remove the camera space's z coordinate from the matrix. This will allow us to use one perspective projection matrix for all vertices, independent of the z coordinate of each input vertex.

I assume, without any evidence to support me, that this was done for efficiency reasons when using OpenGL's fixed function pipeline. (Side note, the standard perspective projection matrix, which we will get to by demo 25, does not linearly position the $nearZ_c$ to $farZ_c$ data into NDC. Everything we've done so far in the class does. The standard perspective matrix ends up having less Z-fighting close to $nearZ_c$, and more problems with Z-fighting near farZ_c)

OpenGL will automatically convert from clip space to NDC such as follows.

$$\vec{f}_{clip}^{ndc}(\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} \end{bmatrix}) = \begin{bmatrix} \frac{1}{w_{clip}} & 0 & 0 & 0 \\ 0 & \frac{1}{w_{clip}} & 0 & 0 \\ 0 & 0 & \frac{1}{w_{clip}} & 0 \\ 0 & 0 & 0 & \frac{1}{w_{clip}} \end{bmatrix} * \begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix}$$

So to put our NDC data into clip space, knowing what OpenGL is going to do in the equation above, we need to decide what we want our clip space value, w to be, and do the inverse of the equation above

$$\vec{f}_{ndc}^{clip}(\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} \end{bmatrix} ; w) = \begin{bmatrix} w & 0 & 0 & 0 \\ 0 & w & 0 & 0 \\ 0 & 0 & w & 0 \\ 0 & 0 & 0 & w \end{bmatrix} * \begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} \end{bmatrix}$$

$$\vec{f}_{clip}^{clip}(\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix}) = (\vec{f}_{clip}^{ndc} \circ \vec{f}_{ndc}^{clip}) \begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w \end{bmatrix}$$

Since we want to get the z_c relative to camera space out of the premultiplied matrix above, we choose the following

$$\vec{f}_{ndc}^{clip}(\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} = 1 \end{bmatrix} ; z_c) = \begin{bmatrix} z_c & 0 & 0 & 0 \\ 0 & z_c & 0 & 0 \\ 0 & 0 & z_c & 0 \\ 0 & 0 & 0 & z_c \end{bmatrix} * \begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} \end{bmatrix}$$

because multiplying by this matrix will remove the z_c out of the upper left quadrant.

Remove Z of Camera Space from Part of the Matrix

To get camera z out of the matrix, where it's currently in two denominators, we can use knowledge of clip space, wherein we put cameraspace's z into W. because cameraSpace's z coordinate is negative, we want to scale all dimensions without reflecting over the origin, hence the negative sign in $-z_c$.

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \vec{f}_c^{clip}(\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix} ; farZ_c, nearZ_c, top, right)$$

$$= (\vec{f}_{ndc}^{clip} \circ \vec{f}_c^{ndc}) * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

$$= \begin{bmatrix} z_c & 0 & 0 & 0 \\ 0 & z_c & 0 & 0 \\ 0 & 0 & z_c & 0 \\ 0 & 0 & 0 & z_c \end{bmatrix} * \begin{bmatrix} \frac{nearZ_c}{right * z_c} & 0 & 0 & 0 \\ 0 & \frac{nearZ_c}{top * z_c} & 0 & 0 \\ 0 & 0 & \frac{2}{nearZ_c - farZ_c} & \frac{-(farZ_c + nearZ_c)}{nearZ_c - farZ_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{nearZ_c}{right} & 0 & 0 & 0 \\ 0 & \frac{nearZ_c}{top} & 0 & 0 \\ 0 & 0 & z_c * \frac{2}{nearZ_c - farZ_c} & z_c * \frac{-(farZ_c + nearZ_c)}{nearZ_c - farZ_c} \\ 0 & 0 & 0 & z_c \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

The result of this is in clip space, where for the first time, our w component is not 1, but z_c.

Turning clip space back into NDC

$$\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} \end{bmatrix} = \begin{bmatrix} x_{clip}/z_{clip} \\ y_{clip}/z_{clip} \\ z_{clip}/z_{clip} \\ w_{clip}/z_{clip} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{z_c} & 0 & 0 & 0 \\ 0 & \frac{1}{z_c} & 0 & 0 \\ 0 & 0 & \frac{1}{z_c} & 0 \\ 0 & 0 & 0 & \frac{1}{z_c} \end{bmatrix} * \begin{bmatrix} \frac{nearZ_c}{right} * x_c \\ \frac{nearZ_c}{top} * y_c \\ {z_c}^2 * \frac{2}{nearZ_c - farZ_c} + z_c * \frac{-(farZ_c + nearZ_c)}{nearZ_c - farZ_c} \\ z_c \end{bmatrix}$$

To test a corner of the frustum as a smoke test, say

$$
\begin{aligned}
\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} = 1 \end{bmatrix} &= \vec{f}_c^{ndc}(\begin{bmatrix} x_c = right \\ y_c = top \\ z_c = nearZ_c \\ w_c = 1 \end{bmatrix} ; farZ_c, nearZ_c, top, right) \\
&= \begin{bmatrix} \frac{1}{z_c} & 0 & 0 & 0 \\ 0 & \frac{1}{z_c} & 0 & 0 \\ 0 & 0 & \frac{1}{z_c} & 0 \\ 0 & 0 & 0 & \frac{1}{z_c} \end{bmatrix} * \begin{bmatrix} \frac{nearZ_c}{right} * x_c \\ \frac{nearZ_c}{top} * y_c \\ z_c{}^2 * \frac{2}{nearZ_c - farZ_c} + z_c * \frac{-(farZ_c + nearZ_c)}{nearZ_c - farZ_c} \\ z_c \end{bmatrix} \\
&= \begin{bmatrix} \frac{nearZ_c}{right} * x_c \\ \frac{nearZ_c}{top} * y_c \\ z_c * \frac{2}{nearZ_c - farZ_c} + \frac{-(farZ_c + nearZ_c)}{nearZ_c - farZ_c} \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\end{aligned}
$$

And that's what we'd expect and the top right corner of the near plane of the frustum should go to the upper right corner with a z value of 1, as -1 is where the back plane must go.

If we had used $x_c = farZ_c$, then $z_{ndc} = -1$, which is what we want, as negative z axis goes into the monitor.

Remove Z of Camera Space from the Rest of the Matrix

We successfully moved z_c out of the upper left quadrant, but in doing so, we moved it down to the lower right. Can we get rid of it there too? Turn out, we can.

Since the vector multiplied by this matrix will provide z_c as it's third element, we can put $-z_c$ into the w by taking the

explicit version of it out of the fourth column, and put -1 into the third column's w.

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \vec{f}_c^{clip}(\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}; farZ_c, nearZ_c, top, right)$$

$$= (\vec{f}_{ndc}^{clip} \circ \vec{f}_c^{ndc}) * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

$$= \frac{1}{z_c} * \begin{bmatrix} \frac{nearZ_c}{right} & 0 & 0 & 0 \\ 0 & \frac{nearZ_c}{top} & 0 & 0 \\ 0 & 0 & z_c * \frac{2}{nearZ_c - farZ_c} & z_c * \frac{-farZ_c + nearZ_c}{nearZ_c - farZ_c} \\ 0 & 0 & 0 & z_c \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

$$= \frac{1}{z_c} * \begin{bmatrix} \frac{nearZ_c}{right} & 0 & 0 & 0 \\ 0 & \frac{nearZ_c}{top} & 0 & 0 \\ 0 & 0 & z_c * \frac{2}{nearZ_c - farZ_c} & z_c * \frac{-farZ_c + nearZ_c}{nearZ_c - farZ_c} \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c = 1 \end{bmatrix}$$

$$= \frac{1}{z_c} * \begin{bmatrix} \frac{nearZ_c}{right} * x_c \\ \frac{nearZ_c}{top} * y_c \\ {z_c}^2 * \frac{2}{nearZ_c - farZ_c} + z_c * \frac{-(farZ_c + nearZ_c)}{nearZ_c - farZ_c} \\ z_c \end{bmatrix}$$

To remove z_c from the matrix, all that to do is remove it from row 3, somehow. We're about to ride dirty.

If we were to change row three, it would not be the same transformation. But if we ensure the following two properties of our changes, everything will be alright

We need the

- $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \cdot \vec{f}_c^{clip}(\begin{bmatrix} x_c \\ y_c \\ nearZ_c \\ w_c = 1 \end{bmatrix}) = -1.0$
- $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \cdot \vec{f}_c^{clip}(\begin{bmatrix} x_c \\ y_c \\ farZ_c \\ w_c = 1 \end{bmatrix}) = 1.0$
- Ordering is preserved after the function is applied, i.e. monotonicity. if $z_1 < z_2$, then $(\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \cdot \vec{f}_c^{clip}(\begin{bmatrix} 0 \\ 0 \\ z_1 \\ 0 \end{bmatrix})) < (\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \cdot \vec{f}_c^{clip}(\begin{bmatrix} 0 \\ 0 \\ z_2 \\ 0 \end{bmatrix}))$.

If we can make a function, that like the third row of the matrix, has those properties, we can replace the third row and remove camera space's z, z_c, from the matrix. This is desirable because, if it were to exist, would would not need per vertex to create a custom perspective matrix.

Towards that, let's look at these jibronies.

$..//clipSpace.z = A * c.z + B * 1.0(thefirstcolumnandthesecondcolumnarezerobecausezisindependentofxandy)//fornear$

CHAPTER

TWENTYFOUR

INDICES AND TABLES

- genindex
- modindex
- search