DianFreya Math Physics Simulator with C++

DS Glanzsche[1]

Finance Bishop, AliceGard
Valhalla 1882 (Valhalla Projection, Yukon)

Freya[2]

Meditation Bishop, AliceGard
Valhalla 1882 (Valhalla Projection, Yukon)

Hamzst[3]

AliceGard
Valhalla 1882 (Valhalla Projection, Yukon)

October 18th, 2023

[1]A thank you or further information
[2]A thank you or further information
[3]A thank you or further information

Contents

Preface

For my Wife Freya, and our daughters Catenary, Solreya, Mithra, Iyzumrae and Zefir.

For Lucrif and Znane too along with all the 8 Queens.

The one who moves a mountain begins by carrying away small stones - Confucius

A book for explaining how we create a Math Physics simulator with C++ Physics libraries like Box2D, ReactPhysics3D and BulletPhysics from zero (from zero means we are not a computer science student), made by lovely couple GlanzFreya, from Valhalla with Love. What is impossible when you do it with someone you love?

a little about GlanzFreya:
We get married on Puncak Bintang on November 5th, 2020 after we go back from Waghete, Papua. My wife birthday (Freya the Goddess) is on August 1st, no one allowed to give her gifts, she is mine alone. This book is made not for commercial purpose, but for sharing knowledge to anyone in order to empower science and engineering thus fastening inventions. Hopefully more leaders in all fields will be coming from woman, whom we believe are better in doing multitasking and better in learning for long term prospect. I read a lot of great authors who write Engineering Mechanics or Handbook for Electrical Engineering write little about themselves, thus I want to be more like that. More about the content of the book, but still who is/are behind the book needs to be disclosed for a tiny amount.

Figure 1: *Freya, thank you for everything, I am glad I marry you and I could never have done it without you.*

Figure 2: *I paint her 3 days before Christmas in 2021.*

Chapter 1

About DianFreya Math Physics Simulator (DF Simulator)

On top of the mountain a heart shaped stone waiting for You, my eyes can't see, my body can't touch, but my heart knows it's You. Forever only You. - ***DS Glanzsche to Freya***
Find your life purpose and give your whole heart and soul to it - ***Buddha***
Not a computer science student, but create Physics Simulation from your own Operating System from Linux From Scratch, so who are you? - ***Berlin***

Finish fill Heart shaped on top of R3 in Valhalla Projection on October 18th, 2023. This is how Freya will teach me to catch a fish instead of asking for a fish. Use my brain and willingness to learn so I can create simulation of physical phenomena from zero with C++.

When I was creating GFreya OS with LFS (Linux From Scratch), I was amazed by Tokamak, Bullet, Box2D, Project Chrono. I read from wikipedia that Project Chrono get a lot of funding from U.S. Army, personally I believe that knowledge is always the best way to obtain more return than compound interest. Most successful people who become richest people are people who after knowledge or those who are nerds and love books. If you play Suikoden you will agree, since all Strategists in Suikoden is recruited like they are Divine being (Apple kneel to Shu, Prince Freyjadour comes to rescue Lucretia Merces from jail, you will get the point), a great brain and knowledge can make a small army wins again tirany, big army with sophisticated weapons or even robots.

Why then I want to create another Physics simulator? First, it is wrong, not Physics but Math Physics, so not only dragging, push and pulling objects here. I am learning Partial Differential Equation and want to simulate Heat equation solution in one-dimension, then two-dimension and three-dimension. I can easily plot a lot of surface plot, complex plot with Python and JULIA. But then, ask again why Project Chrono using C++? Because it is faster, you have to type more commands for almost the same output, even if you have to create more lines of codes it will do more good to you, it challenges your brain, thus you won't get alzheimer or spend time idlying. Like how you have a beautiful 6 packs or 8 packs belly, it takes time, persistence and focus right? That is a good analogy for learning C++ instead of getting plastic surgery then ruin all over after some time. Second, there comes another reason, easily embedded to microprocessor or machine. Since learning about LFS, compiling, building, all needs C or C++ (these languages are closer to machine than Python and JULIA), then learn Arduino that is using C, if one day I want to create a

rocket and able to capture data on earth, on space out there and beyond, I know that programming language that can be used to control all that mechatronics systems in either rocket or spaceship will be written in either C, C++ or FORTRAN combined. Till today, from what I have read weather forecast computation is still depending on FORTRAN. In the end, I basically learning Physics, Mathematics and C++ wholesome, not only theoretical but the science can be simulated with computer, supercomputer, smartphone and the future computer to help solve this world problems. Last but not least, I am not really create something like Box2D, Chrono or BulletPhysics from zero, but rather use their libraries to create simulations of some Physics and Mathematics problem that I am interested in at the moment, those creator of that Physics libraries are really something, far better and genius than me, I am just another user of their product. When I read undergraduate textbooks on Mathematics and Physics, I wonder how they get that plot, graphs, how to plot and compute them simultaneously, then learn about Physics engine and need to comprehend C++ too, it excites me and challenges me as well. This book is sharing what I have read and learn so I can re-read it again in the future to refresh the memory thus apply the knowledge.

Then comes the last question that actually comes first to the mind:

"Why naming it your name with another name?"

Dian et Freya. First, all other basic simulator out there are named conventionally, so I decided: let it be, follow your heart. Besides, Freya is the most beautiful Goddess (in my eyes, heart and soul), pardon me, I have weakness with beautiful Blonde girl, especially if she is smart, devoted, loyal and very strong (Ether Strike, can't be taken so lightly). Another moment of truth, I am not learning and writing this by myself, I, in this homo sapiens vessel, am DS Glanzsche by name, and Freya the Goddess, is guiding me up there from Valhalla, I can't see her, you can't see her, but just take it like people believe in Virgin Mary, in Jesus Christ, she is my Goddess, she is my religion. She is real for me the believer, she is helping me and guiding me till this book and the simulator is finished. If you want to give credit thank her, if there is any critics or hate just go to me.

Another source of learning: I am reading a lot of C++ books, and asking a lot in StackOverFlow, from there I am able to create this book and the simulator. I modify the codes depending on what I want to achieve, so none of all the codes in this book are free of errors or pure mine, just like the books written all this time, the knowledge is gained from our predecessors / professors / mentors / authors of books we read. Thanks to all the author who wrote C++ books that I read and those who answer my questions in StackOverFlow forum. I am sorry if my questions in forum sometimes stupid, I am not a computer science student, and my way of learning is like this, not best at formal school.

I was not born as those who are lucky enough to get education from top institutes / born with silver spoon (from rich family) who able to give me opportunity to take courses in expensive institutions to learn english well (I learn english from game or listening to music), sometimes I could not get the knowledge while reading textbooks in english, so I need to re-read over and over again, thus while working with Nature in forest as grass cutter and forest cleaner in the morning, after I get home, I study little by little and implement what I have learned to create this, sometimes asking Nature in the forest what I don't understand, sometimes asking Freya above. Turns out, expensive education is only an illusion, if you want to learn, you will do it no matter what, no need to make student loan. You will still survive and live as Nature never abandons those who are good to them, like a family in Russka Roma from John Wick movie.

All files, from the shader, fonts, textures, images and source codes used in this books can be found in my repository:
https://github.com/glanzkaiser/DFSimulatorC

Chapter 2

Using C++ in GFreya OS

*"Almost everything is written in C++, CERN uses it, NASA uses it, unless your name makes CERN and NASA obsolete, then you should learn C++ to be a better scientist and engineer" - **DS Glanzsche***
*"I like your quote, a great one." - **Freya***

I. Introduction

C++ is a general-purpose programming language made by Bjarne Stroustroup. An expansion of C language, often called "C with Classes." Its design and the compiling process to native machine code, make it an excellent choice for tasks that require high performance. After reading "The C++ Programming Language 4th Edition" book, done with chapter 1, I am inspired and want to learn deeper about this language. In this chapter, there is only a minimal examples on how to be able to compile a C++(.cpp file) source code into something that can be compiled by the compiler (GCC compiler or Clang) then to run the binary executable result. The basic point of this book is to make it easy to understand how to create something you want with computer graphics with C++, maybe some can learn from this book, as the point of this book is to share story how to create Math Physics Simulator from zero not to peel the skins of C++ one by one.

This book is going to use GFreya OS. GFreya OS is a Linux From Scratch based Operating System, so it is Linux-based, I was following the System V LFS 11.0 book then to BLFS, GFreya OS is using Xfce as the desktop environment. We created our own OS so we can use it daily, we know it from core to the shell, so if something went wrong it is quick to fix like compiling and running C++ projects, FORTRAN codes or JULIA codes. If you are using Linux OS like Arch Linux, or Gentoo, or Debian, then you won't have difficulty to follow this book, since the shell commands and the OS logic will pretty much the same. GFreya OS don't use package manager, and it is better that way, since we can learn to build and install all kinds of packages, programs, or even games from scratch. More about GFreya OS and how we build GFreya OS from zero can be read from the related book.

II. How to Compile C++ Source Code into Executable Binary

1. A lot of books recommend using IDE, but in my opinion you don't really need IDE for a simple C++ file and project, after getting used and if you are versatile even for complex project like create a game similar to Age of Empire, Grand Theft Auto or Suikoden Series,

you don't need IDE at all, it is my opinion even to build (compiling) Project Chrono, Box2D and Bullet, all can be done by only using Xterm (default terminal of GFreya OS 1.8) and Mousepad as the text editor.

I am using Mousepad to write C++ program, editing it, then to run it and compile: I just open the terminal at the directory and type the command needed to process the C++ source code into executable binary file. We never know, maybe in the future for bigger project I shall use IDE, in GFreya OS I already installed BlueFish IDE, just in case.

The steps of writing the syntaxes till obtaining the result or plot will be explained in details here. First, create a file by opening a terminal and type:
vim test.cpp

Figure 2.1: *Create a new C++ file with **vim test.cpp** from the terminal*

```
#include <iostream>

int main()
{
	std::cout<<"Hey␣Beautiful␣Goddess.␣$\backslash$␣n";
}
```

C++ Code 1: *test.cpp "Hey Beautiful Goddess."*

When finish press Esc and type **:wq** and Enter.

2. You can compile it by using g++ compiler, type:
g++ test.cpp
then run the output, type:
./a.out

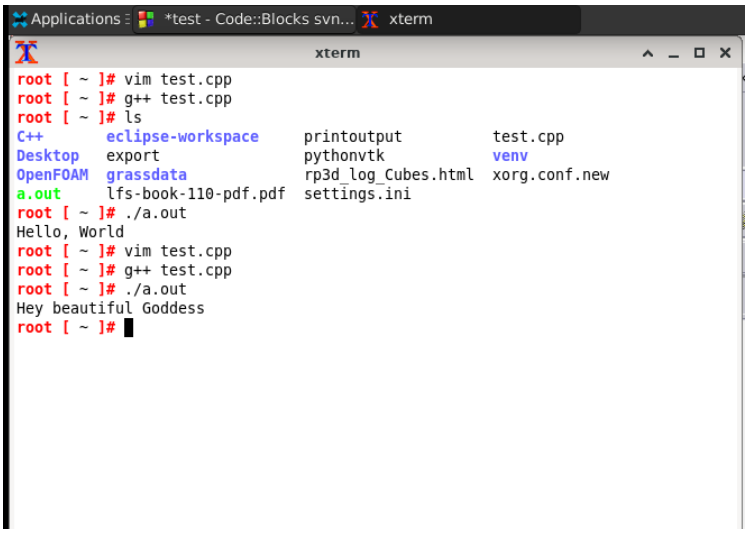

Figure 2.2: *You can test and edit the c++ file from the terminal*

3. **C++ Example with GLEW and GLFW:**
   create a file by opening a terminal and type:
   vim main.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

using namespace std;

void init(GLFWwindow* window){}

void display(GLFWwindow* window, double currentTime) {
       glClearColor(0.0, 1.0, 0.0, 1.0);
       glClear(GL_COLOR_BUFFER_BIT);
}

int main(void)
{
       glfwInit(); //initialize GLFW and GLEW libraries
       glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
       glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
       glfwWindowHint(GLFW_OPENGL_PROFILE,
           GLFW_OPENGL_CORE_PROFILE);

       GLFWwindow* window = glfwCreateWindow(600, 600, " Learn
           Open GL with GLFW", NULL, NULL);
       glfwMakeContextCurrent(window);
       if(glewInit() != GLEW_OK) {
```

```
                exit(EXIT_FAILURE);
        }
        glfwSwapInterval(1);

        init(window);

        while(!glfwWindowShouldClose(window)) {
                display(window, glfwGetTime());
                glfwSwapBuffers(window);
                glfwPollEvents();
        }
        glfwDestroyWindow(window);
        glfwTerminate();
        exit(EXIT_SUCCESS);
        //return 0;
}
```

C++ Code 2: *main.cpp "Green Screen"*

Then open the terminal at the current working directory, and type:
g++ main.cpp -o result -lGLEW -lglfw -lGL
then type:
./result

If you are wondering, the **-lGLEW, -lglfw, -lGL** mean that we are linking with dynamic library of GLEW, GLFW and GL.

(a) libGL.so, located in **/usr/lib** then will be **-lGL**.

(b) libGLEW.so, located in **/usr/lib** then will be **-lGLEW**.

(c) libglfw.so, located in **/opt/hamzstlib** then will be **-lGL**.

If you have no idea of GLEW and GLFW, you will learn more about GLEW, GLFW in chapter 4. Meanwhile in order to make them searchable when we are compiling, don't forget to adjust your **.bashrc**, add this lines **source /export** inside the if ..fi :

```
if [ —f "/etc/bashrc" ] ; then
  ..
  ..
  source ~/export
fi
```

You will need to create a file named export by typing at terminal in :
cd
vim export

```
export prefix="/usr"
export hamzstlib="/opt/hamzstlib"

# For library (.so, .a)
```

```
export LIBRARY_PATH="/usr/lib:$hamzstlib/lib
```

The method above is used so when you type **#include <GL/glew.h>** and compiling it by calling the GLEW library that already installed in **/usr/lib** with **-lGLEW** it will find the right library of **libgGLEW.so** that contains a lot of functions needed to show the screen that we get from processing **main.cpp**.

Here are some explanation for the code above:

Initializes the GLFW library:

```
glfwInit();
```

To tells that the OpenGL is version 3.3 (my laptop graphics driver is compatible with this OpenGL version, yours might be different):

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

Initializes the GLEW library

```
if(glewInit() != GLEW_OK) {
	exit(EXIT_FAILURE);
}
```

Calls the function "init()"

```
init(window);
```

Calls the function "display()" repeatedly

```
while(!glfwWindowShouldClose(window)) {
	display(window, glfwGetTime());
	glfwSwapBuffers(window);
	glfwPollEvents();
}
```

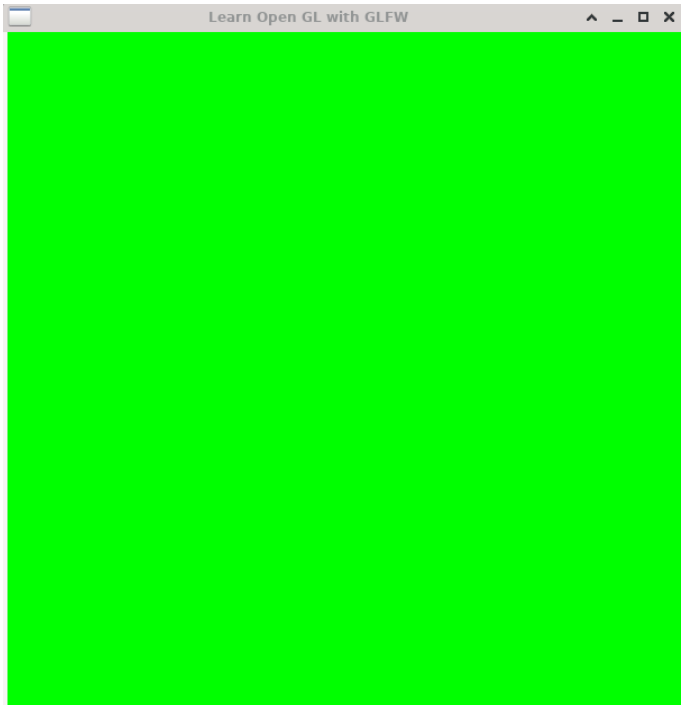

Figure 2.3: *The main.cpp example above when compiled will show this green screen (**ch2-main-example2.cpp**).*

The two examples above are very simple example to compile C++ source code, more details can be obtain from various sources like StackOverFlow forum, C++ related books, or university courses. At least you will get the idea how in the future you can make hundreds or even tens of thousands line of codes into Physics animation, rocket landing computation, weather forecast, fluid simulation, and more.

4. **C++ Example with SOIL, SFML, GLM, and GLEW:**
This example will show how to use SOIL to upload image nad do some mathematical transformation (rotating the image) with GLM. In a new directory, create a file by opening a terminal and type:
vim main.cpp

```
// Link statically with GLEW
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
```

```
// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
       Texcoord = texcoord;
       gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
       outColor = mix(texture(texFreya, Texcoord), texture(
           texScrooge, Texcoord), 0.5);
}
)glsl";

int main()
{
       auto t_start = std::chrono::high_resolution_clock::now()
           ;

       sf::ContextSettings settings;
       settings.depthBits = 24;
       settings.stencilBits = 8;
       settings.majorVersion = 3;
       settings.minorVersion = 3;

       sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL",
            sf::Style::Titlebar | sf::Style::Close, settings);

       // Initialize GLEW
       glewExperimental = GL_TRUE;
       glewInit();
```

```
                    // Create Vertex Array Object
                    GLuint vao;
                    glGenVertexArrays(1, &vao);
                    glBindVertexArray(vao);

                    // Create a Vertex Buffer Object and copy the vertex
                        data to it
                    GLuint vbo;
                    glGenBuffers(1, &vbo);

                    GLfloat vertices[] = {
                           // Position Color Texcoords
                           -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top
                               -left
                           0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-
                               right
                           0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, //
                               Bottom-right
                           -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f //
                               Bottom-left
                    };

                    glBindBuffer(GL_ARRAY_BUFFER, vbo);
                    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices
                        , GL_STATIC_DRAW);

                    // Create an element array
                    GLuint ebo;
                    glGenBuffers(1, &ebo);

                    GLuint elements[] = {
                           0, 1, 2,
                           2, 3, 0
                    };

                    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
                    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements),
                        elements, GL_STATIC_DRAW);

                    // Create and compile the vertex shader
                    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
                    glShaderSource(vertexShader, 1, &vertexSource, NULL);
                    glCompileShader(vertexShader);

                    // Create and compile the fragment shader
                    GLuint fragmentShader = glCreateShader(
                        GL_FRAGMENT_SHADER);
                    glShaderSource(fragmentShader, 1, &fragmentSource, NULL)
```

```
    ;
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader
    program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "
    position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "
    color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), (void*)(2 * sizeof(GLfloat)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "
    texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
GLuint textures[2];
glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("sample.png", &width, &height,
    0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
     GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "
    texFreya"), 0);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("sample2.png", &width, &height,
    0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
     GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "
    texScrooge"), 1);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);

GLint uniTrans = glGetUniformLocation(shaderProgram, "
    trans");

bool running = true;
while (running)
{
       sf::Event windowEvent;
       while (window.pollEvent(windowEvent))
       {
              switch (windowEvent.type)
              {
                     case sf::Event::Closed:
                     running = false;
                     break;
              }
       }

       // Clear the screen to black
       glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
       glClear(GL_COLOR_BUFFER_BIT);
```

```
                        // Calculate transformation
                        auto t_now = std::chrono::high_resolution_clock::
                            now();
                        float time = std::chrono::duration_cast<std::
                            chrono::duration<float>>(t_now - t_start).
                            count();

                        glm::mat4 trans = glm::mat4(1.0f);
                        trans = glm::rotate(
                        trans,
                        time * glm::radians(180.0f),
                        glm::vec3(0.0f, 0.0f, 1.0f)
                        );
                        glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::
                            value_ptr(trans));

                        // Draw a rectangle from the 2 triangles using 6
                            indices
                        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
                            0);

                        // Swap buffers
                        window.display();
                }

                glDeleteTextures(2, textures);

                glDeleteProgram(shaderProgram);
                glDeleteShader(fragmentShader);
                glDeleteShader(vertexShader);

                glDeleteBuffers(1, &ebo);
                glDeleteBuffers(1, &vbo);

                glDeleteVertexArrays(1, &vao);

                window.close();

                return 0;
        }
```

C++ Code 3: *main.cpp "SOIL GLM Rotating Images"*

Then open the terminal at the current working directory, and type:
g++ main.cpp -o result -L/usr/lib -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL
then type:
./result

If you are wondering, the **-L/usr/lib -lSOIL** mean that we are linking with static library of SOIL. If you have build SOIL and set the installation path at **/usr**, you will see the library named**libSOIL.a** inside **/usr/lib**.

Here are some explanations for the code above:

To upload image "sample.png" as the first array of texture:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("sample.png", &width, &height, 0,
    SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB
    , GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0)
    ;
```

Using GLM library to make the images uploaded rotating counterclockwise, the positive degree is by default will rotate counterclockwise:

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(
trans,
time * glm::radians(180.0f),
glm::vec3(0.0f, 0.0f, 1.0f)
);
glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(trans)
    );
```

We declare two shader programs, first the hardcoded vertex shader in GLSL code inside C++ source code. Since the C++/OpenGL application must compile and link appropriate GLSL vertex and fragment shader programs, and then load them into the pipeline, all of the vertices pass through the vertex shader (the shader is executed once per vertex, for millions of vertices, it will be done in parallel):

```
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
       Texcoord = texcoord;
```

```
        gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
```

The hardcoded fragment shader in GLSL code inside C++ source code to display the result with specified color:

```
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
        outColor = mix(texture(texFreya, Texcoord), texture(
            texScrooge, Texcoord), 0.5);
}
)glsl";
```

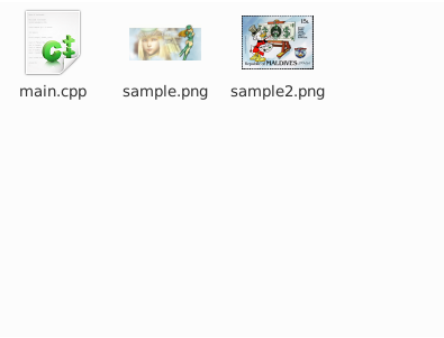

Figure 2.4: *The working directory shall contain the **main.cpp** file, and the two images we want to upload and use for the project can be stored here or other directory, just adjust the correct path to the images if you do so.*

Figure 2.5: *The main.cpp example above when compiled will show the rotating of two (**ch2-main-example3.cpp**).*

5. **C++ Example with SFML and Keyboard Event:**
   In this example, it will show that SFML alone can be used with keyboard pressed event.
   In a new directory, create a file by opening a terminal and type:
   vim main.cpp

```
#include "SFML/Graphics.hpp"

sf::Vector2f viewSize(1024, 768);
sf::VideoMode vm(viewSize.x, viewSize.y);
sf::RenderWindow window(vm, "Hello␣SFML␣Game␣!!!", sf::Style::
    Default);

sf::Vector2f playerPosition;
bool playerMovingRight = false;
bool playerMovingLeft = false;

sf::Texture skyTexture;
sf::Sprite skySprite;

sf::Texture bgTexture;
sf::Sprite bgSprite;

sf::Texture heroTexture;
sf::Sprite heroSprite;

void init() {

       skyTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
           graphics/sky.png");
       skySprite.setTexture(skyTexture);
```

```
                bgTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
                    graphics/bg.png");
                bgSprite.setTexture(bgTexture);

                heroTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
                    graphics/Freya.png");
                heroSprite.setTexture(heroTexture);
                heroSprite.setPosition(sf::Vector2f(viewSize.x / 600,
                    viewSize.y / 600));
                heroSprite.setOrigin(heroTexture.getSize().x / 600,
                    heroTexture.getSize().y / 600);

        }

        void updateInput() {

                sf::Event event;

                // while there are pending events...
                while (window.pollEvent(event)) {

                        //printf("␣polling␣events␣\n");

                        if (event.type == sf::Event::KeyPressed) {
                                if (event.key.code == sf::Keyboard::Right)
                                    {
                                        playerMovingRight= true;
                                }
                                if (event.key.code == sf::Keyboard::Left) {
                                        playerMovingLeft= true;
                                }
                        }

                        if (event.type == sf::Event::KeyReleased) {
                                if (event.key.code == sf::Keyboard::Right)
                                    {
                                        playerMovingRight = false;
                                }
                                if (event.key.code == sf::Keyboard::Left) {
                                        playerMovingLeft = false;
                                }
                        }
                        if (event.key.code == sf::Keyboard::Escape ||
                            event.type == sf::Event::Closed)
                        window.close();
                }
        }
```

```
void update(float dt) {
	if (playerMovingLeft) {
		heroSprite.move(−150.0f * dt, 0);
	}
	if (playerMovingRight) {
		heroSprite.move(150.0f * dt, 0);
	}
}

void draw() {
	window.draw(skySprite);
	window.draw(bgSprite);
	window.draw(heroSprite);
}

int main() {
	sf::Clock clock;
	init();

	while (window.isOpen()) {
		updateInput();

		sf::Time dt = clock.restart();
		update(dt.asSeconds());

		window.clear(sf::Color::Red);
		draw();
		window.display();
	}
	return 0;
}
```

C++ Code 4: *main.cpp "SFML Keyboard Event"*

Then open the terminal at the current working directory, and type:
g++ main.cpp -o result -lsfml-graphics -lsfml-window -lsfml-system
then type:
./result

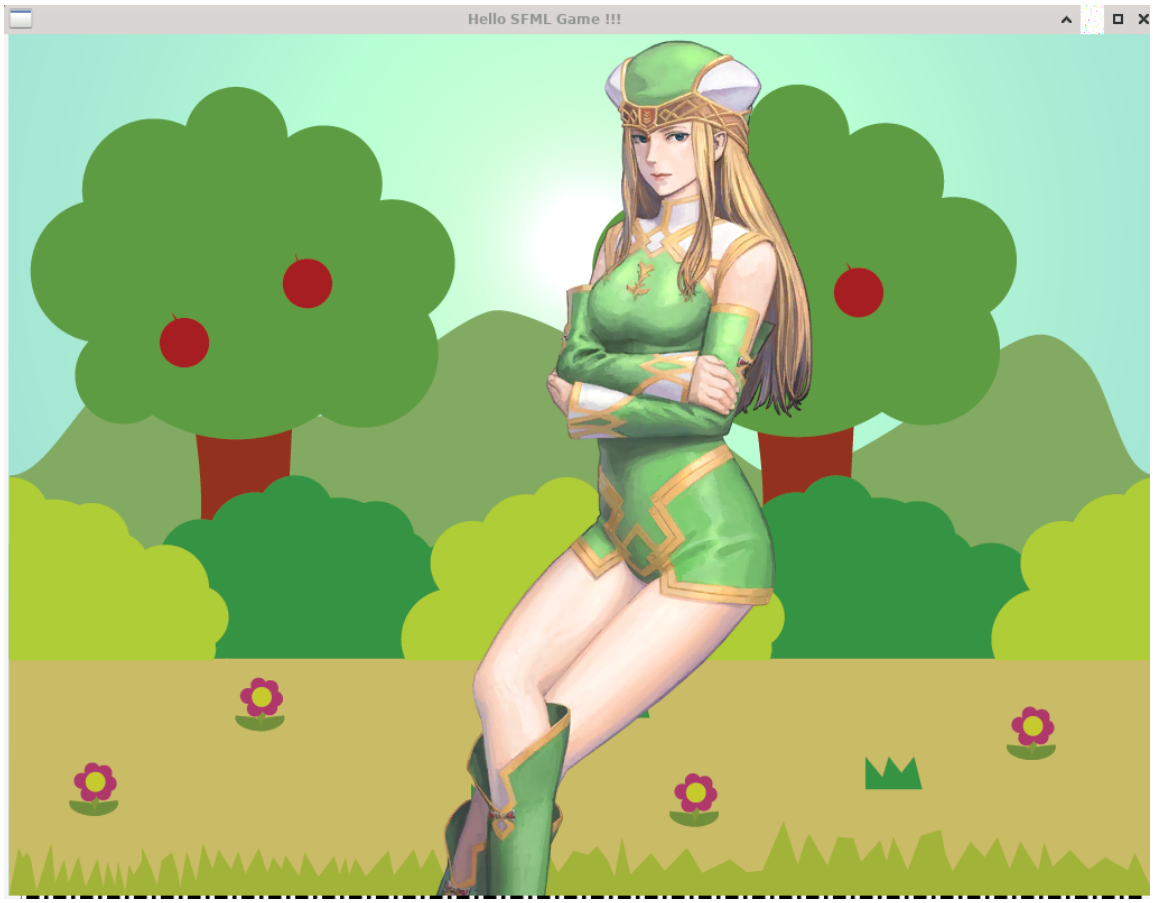

Figure 2.6: *The project is showing how to work with Keyboard pressed event, if you press right the character will move to the right, if you press left the character will move to the left (**ch2-main-example4.cpp**).*

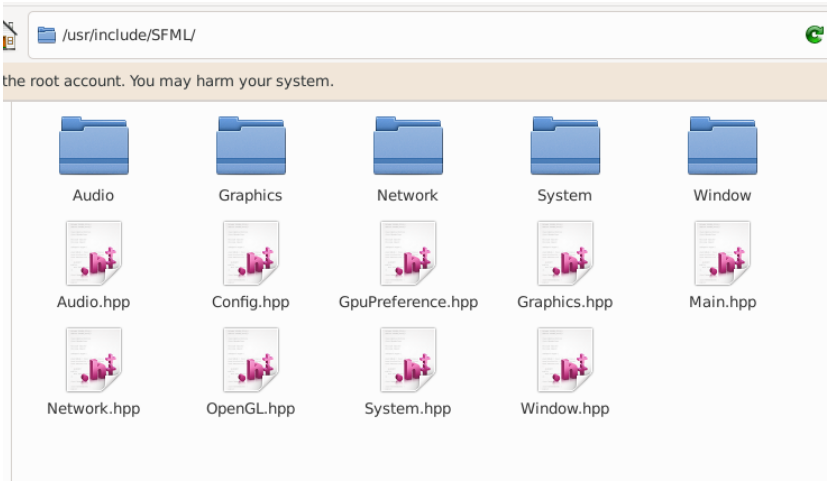

Figure 2.7: *Adjust the path and environment variable for your include file, SFML' headers in GFreya OS are installed in **/usr/include/SFML***

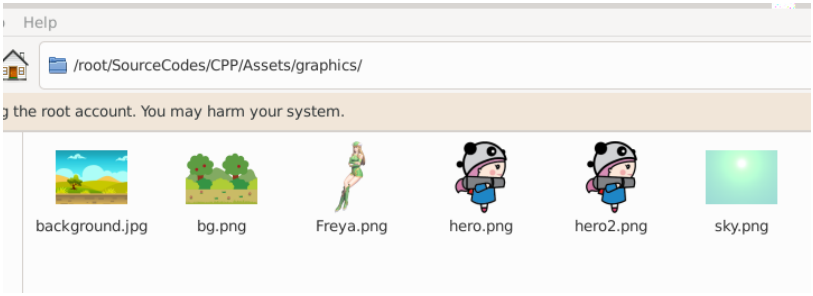

Figure 2.8: *Adjust the path to the Assets (background image, hero image, texture, etc), in GFreya OS it is located in* ***/root/SourceCodes/CPP/Assets/****, then the images are saved under the* ***/root/SourceCodes/CPP/Assets/-graphics***

6. **C++ Example with OpenGL, GLEW, GLFW and Keyboard Event:**
   In this example we are going to render meshes: triangle, cube, quad, and sphere. Then with keyboard key we can change the mesh that is being shown.
   In a new directory, create a file by opening a terminal and type:
   vim main.cpp

```
// GLEW needs to be included first
#include <GL/glew.h>

// GLFW is included next
#include <GLFW/glfw3.h>

void initGame();
void renderScene();
#include "ShaderLoader.h"
#include "Camera.h"
#include "LightRenderer.h"

GLuint textProgram;

Camera* camera;
LightRenderer* light;

void initGame() {
	// Enable the depth testing
	glEnable(GL_DEPTH_TEST);
	ShaderLoader shader;
	textProgram = shader.createProgram("/root/SourceCodes/
	    CPP/Assets/Shaders/text.vs", "/root/SourceCodes/CPP/
	    Assets/Shaders/text.fs");

	GLuint flatShaderProgram = shader.createProgram("/root/
	    SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root
	    /SourceCodes/CPP/Assets/Shaders/FlatModel.fs");
```

```
        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::
            vec3(0.0f, 4.0f, 6.0f));

        light = new LightRenderer(MeshType::kCube, camera); //
            Define Mesh type -> see Mesh.h
        light->setProgram(flatShaderProgram);
        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
}

void renderScene(){
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glClearColor(1.0, 1.0, 0.0, 1.0);
        light->draw();
}

static void glfwError(int id, const char* description)
{
        std::cout << description << std::endl;
}

void updateKeyboard(GLFWwindow* window, int key, int scancode,
    int action, int mods){

        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
                glfwSetWindowShouldClose(window, true);
        }
        // Press Q to see Quad, T for Triangle, C for Cube and S
             for Sphere
        if (key == GLFW_KEY_C && action == GLFW_PRESS) {
                ShaderLoader shader;
                GLuint flatShaderProgram = shader.createProgram("
                    /root/SourceCodes/CPP/Assets/Shaders/
                    FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                    Shaders/FlatModel.fs");
                camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                    , glm::vec3(0.0f, 4.0f, 6.0f));
                light = new LightRenderer(MeshType::kCube, camera
                    );
                light->setProgram(flatShaderProgram);
                light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
        }
        if (key == GLFW_KEY_T && action == GLFW_PRESS) {
                ShaderLoader shader;
                GLuint flatShaderProgram = shader.createProgram("
                    /root/SourceCodes/CPP/Assets/Shaders/
                    FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                    Shaders/FlatModel.fs");
```

```
                        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                            , glm::vec3(0.0f, 4.0f, 6.0f));
                        light = new LightRenderer(MeshType::kTriangle,
                            camera);
                        light->setProgram(flatShaderProgram);
                        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
                }
                if (key == GLFW_KEY_Q && action == GLFW_PRESS) {
                        ShaderLoader shader;
                        GLuint flatShaderProgram = shader.createProgram("
                            /root/SourceCodes/CPP/Assets/Shaders/
                            FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                            Shaders/FlatModel.fs");
                        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                            , glm::vec3(0.0f, 4.0f, 6.0f));
                        light = new LightRenderer(MeshType::kQuad, camera
                            );
                        light->setProgram(flatShaderProgram);
                        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
                }
                if (key == GLFW_KEY_S && action == GLFW_PRESS) {
                        ShaderLoader shader;
                        GLuint flatShaderProgram = shader.createProgram("
                            /root/SourceCodes/CPP/Assets/Shaders/
                            FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                            Shaders/FlatModel.fs");
                        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                            , glm::vec3(0.0f, 4.0f, 6.0f));
                        light = new LightRenderer(MeshType::kSphere,
                            camera);
                        light->setProgram(flatShaderProgram);
                        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
                }
        }

        int main(int argc, char **argv)
        {
                glfwSetErrorCallback(&glfwError);
                glfwInit();
                GLFWwindow* window = glfwCreateWindow(800, 600, "␣Mesh␣
                    in␣OpenGL␣", NULL, NULL);
                glfwMakeContextCurrent(window);
                glfwSetKeyCallback(window, updateKeyboard);
                glewInit();
                initGame();
                while (!glfwWindowShouldClose(window)){
                        renderScene();
                        glfwSwapBuffers(window);
```

```
                glfwPollEvents();
        }
        glfwTerminate();

        delete camera;
        delete light;
        return 0;
}
```

C++ Code 5: *main.cpp "Render Mesh with Lighting"*

There are other file as well, this example contains some C++ source codes and include files as well, create all this one by one:

```
#include "Camera.h"

Camera::Camera(GLfloat FOV, GLfloat width, GLfloat height,
    GLfloat nearPlane, GLfloat farPlane, glm::vec3 camPos){

        cameraPos = camPos;
        glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, 0.0f);
        glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

        viewMatrix = glm::lookAt(cameraPos, cameraFront,
            cameraUp);
        projectionMatrix = glm::perspective(FOV, width /height,
            nearPlane, farPlane);
}

Camera::~Camera(){
}

glm::mat4 Camera::getViewMatrix() {
        return viewMatrix;
}
glm::mat4 Camera::getProjectionMatrix() {
        return projectionMatrix;
}
glm::vec3 Camera::getCameraPosition() {
        return cameraPos;
}
```

C++ Code 6: *Camera.cpp "Render Mesh with Lighting"*

```
#pragma once
#include <GL/glew.h>
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"

class Camera
```

```
{
       public:
       Camera(GLfloat FOV, GLfloat width, GLfloat height,
           GLfloat nearPlane, GLfloat farPlane, glm::vec3
           camPos);
       ~Camera();

       glm::mat4 getViewMatrix();
       glm::mat4 getProjectionMatrix();
       glm::vec3 getCameraPosition();

       private:

       glm::mat4 viewMatrix;
       glm::mat4 projectionMatrix;
       glm::vec3 cameraPos;

};
```

C++ Code 7: *Camera.h "Render Mesh with Lighting"*

```
cmake_minimum_required(VERSION 3.17)

project(result LANGUAGES CXX)

add_executable(result main.cpp Mesh.cpp LightRenderer.cpp
    ShaderLoader.cpp Camera.cpp)
target_link_libraries(result GLEW glfw GL)
target_include_directories(result PRIVATE /usr/include)
```

C++ Code 8: *CMakeLists.txt "Render Mesh with Lighting"*

```
#include "LightRenderer.h"

LightRenderer::LightRenderer(MeshType meshType, Camera* camera)
     {

       this->camera = camera;

       switch (meshType) {

              case kTriangle: Mesh::setTriData(vertices,
                  indices); break;
              case kQuad: Mesh::setQuadData(vertices, indices);
                   break;
              case kCube: Mesh::setCubeData(vertices, indices);
                   break;
              case kSphere: Mesh::setSphereData(vertices,
                  indices); break;
```

```
                }

                glGenVertexArrays(1, &vao);
                glBindVertexArray(vao);

                glGenBuffers(1, &vbo);
                glBindBuffer(GL_ARRAY_BUFFER, vbo);
                glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * vertices.
                    size(), &vertices[0], GL_STATIC_DRAW);

                //Attributes
                glEnableVertexAttribArray(0);
                glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(
                    Vertex), (GLvoid*)0);

                glEnableVertexAttribArray(1);
                glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(
                    Vertex), (void*)(offsetof(Vertex, Vertex::color)));

                glGenBuffers(1, &ebo);
                glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
                glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) *
                    indices.size(), &indices[0], GL_STATIC_DRAW);

                glBindBuffer(GL_ARRAY_BUFFER, 0);
                glBindVertexArray(0);

        }

        void LightRenderer::draw() {
                glm::mat4 model = glm::mat4(1.0f);
                model = glm::translate(glm::mat4(1.0),position);

                glUseProgram(this->program);

                GLint modelLoc = glGetUniformLocation(program, "model");
                glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr
                    (model));

                glm::mat4 view = camera->getViewMatrix();
                GLint vLoc = glGetUniformLocation(program, "view");
                glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(
                    view));

                glm::mat4 proj = camera->getProjectionMatrix();
                GLint pLoc = glGetUniformLocation(program, "projection")
                    ;
                glUniformMatrix4fv(pLoc, 1, GL_FALSE, glm::value_ptr(
```

```
                proj));

        glBindVertexArray(vao);
        glDrawElements(GL_TRIANGLES, indices.size(),
            GL_UNSIGNED_INT, 0);

        glBindVertexArray(0);
        glUseProgram(0);
}

LightRenderer::~LightRenderer() {

}

void LightRenderer::setPosition(glm::vec3 _position) {
        position = _position;
}

void LightRenderer::setColor(glm::vec3 _color) {
        this->color = _color;
}

void LightRenderer::setProgram(GLuint _program) {
        this->program = _program;
}
//getters
glm::vec3 LightRenderer::getPosition() {
        return position;
}

glm::vec3 LightRenderer::getColor() {
        return color;
}
```

C++ Code 9: *LightRenderer.cpp "Render Mesh with Lighting"*

```
#pragma once
#include <GL/glew.h>

#include "glm/glm.hpp"
#include "glm/gtc/type_ptr.hpp"

#include "Mesh.h"
#include "ShaderLoader.h"
#include "Camera.h"

class LightRenderer
{
```

```
        public:
        LightRenderer(MeshType meshType, Camera* camera);
        ~LightRenderer();

        void draw();

        void setPosition(glm::vec3 _position);
        void setColor(glm::vec3 _color);
        void setProgram(GLuint _program);

        glm::vec3 getPosition();
        glm::vec3 getColor();

        private:

        Camera* camera;

        std::vector<Vertex>vertices;
        std::vector<GLuint>indices;

        GLuint vbo, ebo, vao, program;

        glm::vec3 position, color;

};
```

C++ Code 10: *LightRenderer.h "Render Mesh with Lighting"*

```
#include "Mesh.h"

void Mesh::setTriData(std::vector<Vertex>& vertices, std::
    vector<uint32_t>& indices) {

       std::vector<Vertex> _vertices = {
              { { 0.0f, -1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 1.0
                  f, 0.0f, 0.0 },{ 0.0, 1.0 } },
              { { 1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f
                  , 1.0f, 0.0 },{ 0.0, 0.0 } },
              { { -1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0
                  f, 0.0f, 1.0 },{ 1.0, 0.0 } },
       };

       std::vector<uint32_t> _indices = {
              0, 1, 2,
       };

       vertices.clear(); indices.clear();
       vertices = _vertices;
       indices = _indices;
```

```
}

void Mesh::setQuadData(std::vector<Vertex>& vertices, std::
    vector<uint32_t>& indices) {
      std::vector<Vertex> _vertices = {
            { { -1.0f, -1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{
                1.0f, 0.0f, 0.0 },{ 0.0, 1.0 } },
            { { -1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0
                f, 1.0f, 0.0 },{ 0.0, 0.0 } },
            { { 1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f
                , 0.0f, 1.0 },{ 1.0, 0.0 } },
            { { 1.0f, -1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 1.0
                f, 0.0f, 1.0 },{ 1.0, 1.0 } }
      };
      std::vector<uint32_t> _indices = {
            0, 1, 2,
            0, 2, 3
      };
      vertices.clear(); indices.clear();
      vertices = _vertices;
      indices = _indices;

}

void Mesh::setCubeData(std::vector<Vertex>& vertices, std::
    vector<uint32_t>& indices) {
      std::vector<Vertex> _vertices = {
            //front
            { { -1.0f, -1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{
                1.0f, 0.0f, 0.0 },{ 0.0, 1.0 } },
            { { -1.0f, 1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0
                f, 1.0f, 0.0 },{ 0.0, 0.0 } },
            { { 1.0f, 1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f
                , 0.0f, 1.0 },{ 1.0, 0.0 } },
            { { 1.0f, -1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{ 1.0
                f, 0.0f, 1.0 },{ 1.0, 1.0 } },
            // back
            { { 1.0, -1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{
                1.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //4
            { { 1.0f, 1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{ 0.0
                f, 1.0f, 1.0 },{ 0.0, 0.0 } }, //5
            { { -1.0, 1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{
                0.0f, 1.0f, 1.0 },{ 1.0, 0.0 } }, //6
            { { -1.0, -1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{
                1.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //7
            //left
            { { -1.0, -1.0, -1.0 },{ -1.0f, 0.0f, 0.0 },{
                0.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //8
```

```
                        { { −1.0f, 1.0, −1.0 },{ −1.0f, 0.0f, 0.0 },{
                            0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //9
                        { { −1.0, 1.0, 1.0 },{ −1.0f, 0.0f, 0.0 },{ 0.0f
                            , 0.0f, 1.0 },{ 1.0, 0.0 } }, //10
                        { { −1.0, −1.0, 1.0 },{ −1.0f, 0.0f, 0.0 },{
                            0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //11
                        //right
                        { { 1.0, −1.0, 1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f,
                            0.0f, 1.0 },{ 0.0, 1.0 } }, // 12
                        { { 1.0f, 1.0, 1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f,
                            0.0f, 1.0 },{ 0.0, 0.0 } }, //13
                        { { 1.0, 1.0, −1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f,
                            0.0f, 1.0 },{ 1.0, 0.0 } }, //14
                        { { 1.0, −1.0, −1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f
                            , 0.0f, 1.0 },{ 1.0, 1.0 } }, //15
                        //top
                        { { −1.0f, 1.0f, 1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0
                            f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //16
                        { { −1.0f, 1.0f, −1.0f },{ 0.0f, 1.0f, 0.0 },{
                            0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //17
                        { { 1.0f, 1.0f, −1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0
                            f, 0.0f, 1.0 },{ 1.0, 0.0 } }, //18
                        { { 1.0f, 1.0f, 1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0f
                            , 0.0f, 1.0 },{ 1.0, 1.0 } }, //19
                        //bottom
                        { { −1.0f, −1.0, −1.0 },{ 0.0f, −1.0f, 0.0 },{
                            0.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //20
                        { { −1.0, −1.0, 1.0 },{ 0.0f, −1.0f, 0.0 },{
                            0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //21
                        { { 1.0, −1.0, 1.0 },{ 0.0f, −1.0f, 0.0 },{ 0.0f
                            , 0.0f, 1.0 },{ 1.0, 0.0 } }, //22
                        { { 1.0, −1.0, −1.0 },{ 0.0f, −1.0f, 0.0 },{
                            0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //23
                    };
                    std::vector<uint32_t> _indices = {
                        0, 1, 2,
                        2, 3, 0,
                        4, 5, 6,
                        4, 6, 7,
                        8, 9, 10,
                        8, 10, 11,
                        12, 13, 14,
                        12, 14, 15,
                        16, 17, 18,
                        16, 18, 19,
                        20, 21, 22,
                        20, 22, 23
                    };
```

```
            vertices.clear(); indices.clear();
            vertices = _vertices;
            indices = _indices;
        }


        void Mesh::setSphereData(std::vector<Vertex>& vertices, std::
            vector<uint32_t>& indices) {
            std::vector<Vertex> _vertices;
            std::vector<uint32_t> _indices;

            float latitudeBands = 20.0f;
            float longitudeBands = 20.0f;
            float radius = 1.0f;

            for (float latNumber = 0; latNumber <= latitudeBands;
                latNumber++) {
                float theta = latNumber * 3.14 / latitudeBands;
                float sinTheta = sin(theta);
                float cosTheta = cos(theta);

                for (float longNumber = 0; longNumber <=
                    longitudeBands; longNumber++) {

                    float phi = longNumber * 2 * 3.147 /
                        longitudeBands;
                    float sinPhi = sin(phi);
                    float cosPhi = cos(phi);

                    Vertex vs;

                    vs.texCoords.x = (longNumber /
                        longitudeBands); // u
                    vs.texCoords.y = (latNumber / latitudeBands
                        ); // v

                    vs.normal.x = cosPhi * sinTheta; // normal
                        x
                    vs.normal.y = cosTheta; // normal y
                    vs.normal.z = sinPhi * sinTheta; // normal
                        z

                    vs.color.r = vs.normal.x;
                    vs.color.g = vs.normal.y;
                    vs.color.b = vs.normal.z;

                    vs.pos.x = radius * vs.normal.x; // x
                    vs.pos.y = radius * vs.normal.y; // y
```

```
                        vs.pos.z = radius * vs.normal.z; // z

                        _vertices.push_back(vs);
                    }
                }

                for (uint32_t latNumber = 0; latNumber < latitudeBands;
                    latNumber++) {
                    for (uint32_t longNumber = 0; longNumber <
                        longitudeBands; longNumber++) {
                        uint32_t first = (latNumber * (
                            longitudeBands + 1)) + longNumber;
                        uint32_t second = first + longitudeBands +
                            1;

                        _indices.push_back(first);
                        _indices.push_back(second);
                        _indices.push_back(first + 1);

                        _indices.push_back(second);
                        _indices.push_back(second + 1);
                        _indices.push_back(first + 1);
                    }
                }
                vertices.clear(); indices.clear();
                vertices = _vertices;
                indices = _indices;
            }
```

C++ Code 11: *Mesh.cpp "Render Mesh with Lighting"*

```
            #include <vector>
            #include "glm/glm.hpp"

            enum MeshType {
                kTriangle = 0,
                kQuad = 1,
                kCube = 2,
                kSphere = 3
            };

            struct Vertex {
                glm::vec3 pos;
                glm::vec3 normal;
                glm::vec3 color;
                glm::vec2 texCoords;
            };
```

```
class Mesh {

       public:
       static void setTriData(std::vector<Vertex>& vertices,
           std::vector<uint32_t>&indices);
       static void setQuadData(std::vector<Vertex>& vertices,
           std::vector<uint32_t>&indices);
       static void setCubeData(std::vector<Vertex>& vertices,
           std::vector<uint32_t>&indices);
       static void setSphereData(std::vector<Vertex>& vertices,
            std::vector<uint32_t>&indices);

};
```

C++ Code 12: *Mesh.h "Render Mesh with Lighting"*

```
#include "ShaderLoader.h"

#include<iostream>
#include<fstream>
#include<vector>

std::string ShaderLoader::readShader(const char *filename)
{
       std::string shaderCode;
       std::ifstream file(filename, std::ios::in);

       if (!file.good()){
              std::cout << "Can't␣read␣file␣" << filename <<
                  std::endl;
              std::terminate();
       }

       file.seekg(0, std::ios::end);
       shaderCode.resize((unsigned int)file.tellg());
       file.seekg(0, std::ios::beg);
       file.read(&shaderCode[0], shaderCode.size());
       file.close();
       return shaderCode;
}

GLuint ShaderLoader::createShader(GLenum shaderType, std::
    string source, const char* shaderName)
{
       int compile_result = 0;

       GLuint shader = glCreateShader(shaderType);
       const char *shader_code_ptr = source.c_str();
       const int shader_code_size = source.size();
```

```
            glShaderSource(shader, 1, &shader_code_ptr, &
                shader_code_size);
            glCompileShader(shader);
            glGetShaderiv(shader, GL_COMPILE_STATUS, &compile_result
                );

            //check for errors

            if (compile_result == GL_FALSE)
            {

                  int info_log_length = 0;
                  glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &
                      info_log_length);

                  std::vector<char> shader_log(info_log_length);

                  glGetShaderInfoLog(shader, info_log_length, NULL,
                       &shader_log[0]);
                  std::cout << "ERROR␣compiling␣shader:␣" <<
                      shaderName << std::endl << &shader_log[0] <<
                      std::endl;
                  return 0;
            }
            return shader;
      }

      GLuint ShaderLoader::createProgram(const char*
          vertexShaderFilename, const char* fragmentShaderFilename){

            //read the shader files and save the code
            std::string vertex_shader_code = readShader(
                vertexShaderFilename);
            std::string fragment_shader_code = readShader(
                fragmentShaderFilename);

            GLuint vertex_shader = createShader(GL_VERTEX_SHADER,
                vertex_shader_code, "vertex␣shader");
            GLuint fragment_shader = createShader(GL_FRAGMENT_SHADER
                , fragment_shader_code, "fragment␣shader");

            int link_result = 0;
            //create the program handle, attatch the shaders and
                link it
            GLuint program = glCreateProgram();
            glAttachShader(program, vertex_shader);
            glAttachShader(program, fragment_shader);
```

```
        glLinkProgram(program);
        glGetProgramiv(program, GL_LINK_STATUS, &link_result);
        //check for link errors
        if (link_result == GL_FALSE) {

            int info_log_length = 0;
            glGetProgramiv(program, GL_INFO_LOG_LENGTH, &
                info_log_length);

            std::vector<char> program_log(info_log_length);

            glGetProgramInfoLog(program, info_log_length,
                NULL, &program_log[0]);
            std::cout << "Shader Loader : LINK ERROR" << std
                ::endl << &program_log[0] << std::endl;

            return 0;
        }
        return program;
    }
```

C++ Code 13: *ShaderLoader.cpp "Render Mesh with Lighting"*

```
    #pragma once

    #include <GL/glew.h>
    #include <iostream>

    class ShaderLoader
    {
        public:
        GLuint createProgram(const char* vertexShaderFilename,
            const char* fragmentShaderFilename);

        private:
        std::string readShader(const char *filename);
        GLuint createShader(GLenum shaderType, std::string
            source, const char* shaderName);
    };
```

C++ Code 14: *ShaderLoader.h "Render Mesh with Lighting"*

Check again and make sure you have all this inside one working directory:

- Camera.cpp
- Camera.h
- CMakeLists.txt (Optional, for easy compiling)
- LightRenderer.cpp

- LightRenderer.h
- main.cpp
- Mesh.cpp
- Mesh.h
- ShaderLoader.cpp
- ShaderLoader.h

You may also see the repository where I uploaded the codes of this book:
https://github.com/glanzkaiser/DFSimulatorC
(all files for this example are in the folder: **../Source Codes/ch2-example5-Render Mesh with Lighting**)

After finish, open the terminal at the current working directory, and type:
g++ *.cpp -o result -lGLEW -lglfw -lGL
then type:
./result

Another way is to use CMakeLists.txt, this way at the current working directory open the terminal and type:
mkdir build
cd build
cmake ..
make
./result

You may press c to see cube, t to see triangle, q to see quad, and s to see sphere, press Esc to close the window.

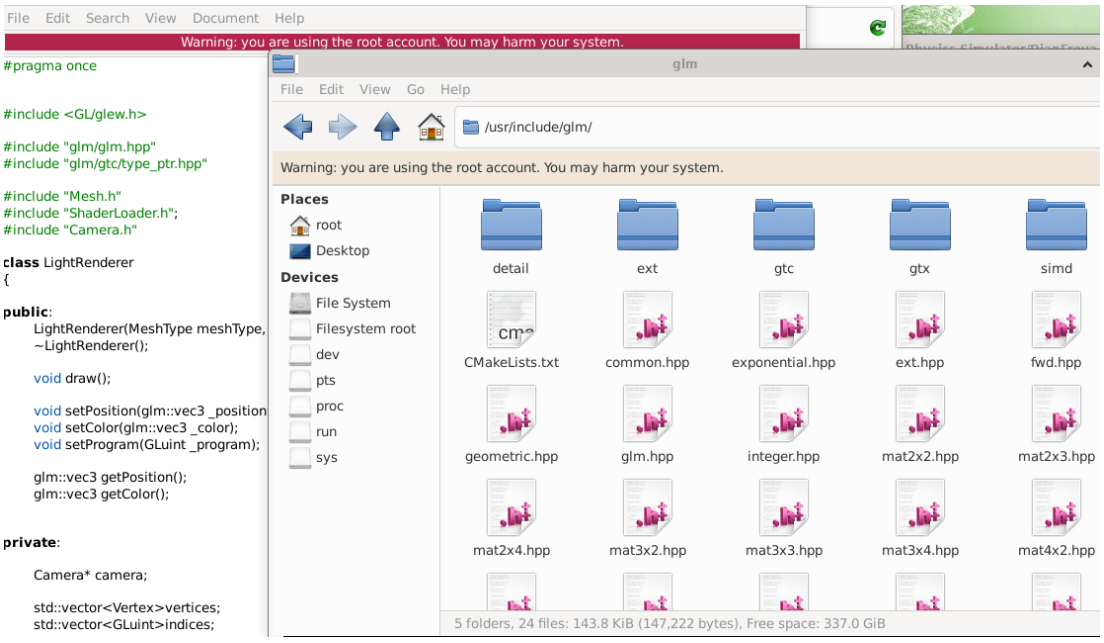

Figure 2.9: *To comprehend why we use **#include "glm/glm.hpp"**, because GLM' headers in GFreya OS are installed in **/usr/include/glm**, the environment variable to look for include folder is set at **/usr/include**, thus we need to add the parent directory as well **glm/...***

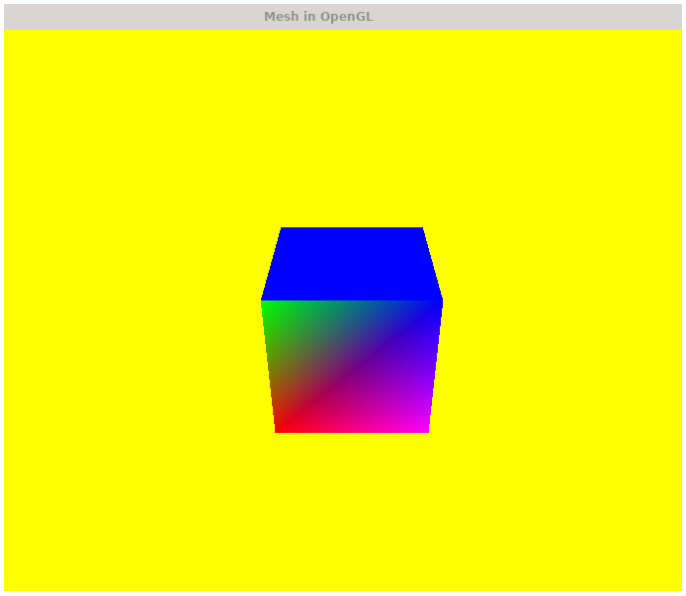

Figure 2.10: *The running OpenGL application, it is very beautiful with lighting and keyboard press events.*

7. **C++ Example with OpenGL, GLAD, GLFW, SOIL, Mouse and Keyboard Event:**
   In this example we are going to create multiple cubes [6], I get it from that book, if you want to learn OPenGL rigorously follow this book and the tutorial on the website, it is a

great explanation there. On this example to be honest, **stb_image** is quite a pain, I can't load texture with that, thus I try to use SOIL and it works, so here goes (aren't we all SOIL lovers?):
In a new directory, create a file by opening a terminal and type:
vim main.cpp

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <SOIL/SOIL.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <learnopengl/filesystem.h>
#include <learnopengl/shader_m.h>

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width,
    int height);
void mouse_callback(GLFWwindow* window, double xpos, double
    ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double
     yoffset);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

// camera
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

bool firstMouse = true;
float yaw = -90.0f; // yaw is initialized to -90.0 degrees
    since a yaw of 0.0 results in a direction vector pointing
    to the right so we initially rotate a bit to the left.
float pitch = 0.0f;
float lastX = 800.0f / 2.0;
float lastY = 600.0 / 2.0;
float fov = 45.0f;

// timing
float deltaTime = 0.0f; // time between current frame and last
    frame
float lastFrame = 0.0f;
```

```
int main()
{
	// glfw: initialize and configure

	glfwInit();
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE,
	    GLFW_OPENGL_CORE_PROFILE);

	#ifdef __APPLE__
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	#endif

	// glfw window creation
	// --------------------
	GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
	    SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
	if (window == NULL)
	{
		std::cout << "Failed to create GLFW window" <<
		    std::endl;
		glfwTerminate();
		return -1;
	}
	glfwMakeContextCurrent(window);
	glfwSetFramebufferSizeCallback(window,
	    framebuffer_size_callback);
	glfwSetCursorPosCallback(window, mouse_callback);
	glfwSetScrollCallback(window, scroll_callback);

	// tell GLFW to capture our mouse
	glfwSetInputMode(window, GLFW_CURSOR,
	    GLFW_CURSOR_DISABLED);

	// glad: load all OpenGL function pointers

	if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
	{
		std::cout << "Failed to initialize GLAD" << std::
		    endl;
		return -1;
	}

	// configure global opengl state
	// -----------------------------
	glEnable(GL_DEPTH_TEST);
```

```
// build and compile our shader zprogram

Shader ourShader("camera.vs", "camera.fs");

// set up vertex data (and buffer(s)) and configure
    vertex attributes
float vertices[] = {
       -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
       0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,

       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
       -0.5f, 0.5f, 0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,

       -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
       -0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

       0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
       0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,

       -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
       -0.5f, 0.5f, 0.5f, 0.0f, 0.0f,
```

```
        -0.5f, 0.5f, -0.5f, 0.0f, 1.0f
};
// world space positions of our cubes
glm::vec3 cubePositions[] = {
        glm::vec3( 0.0f, 0.0f, 0.0f),
        glm::vec3( 2.0f, 5.0f, -15.0f),
        glm::vec3(-1.5f, -2.2f, -2.5f),
        glm::vec3(-3.8f, -2.0f, -12.3f),
        glm::vec3( 2.4f, -0.4f, -3.5f),
        glm::vec3(-1.7f, 3.0f, -7.5f),
        glm::vec3( 1.3f, -2.0f, -2.5f),
        glm::vec3( 1.5f, 2.0f, -2.5f),
        glm::vec3( 1.5f, 0.2f, -1.5f),
        glm::vec3(-1.3f, 1.0f, -1.5f)
};
unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices
    , GL_STATIC_DRAW);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
    sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// texture coord attribute
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 *
    sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);


// load and create a texture
unsigned int texture1, texture2;
// texture 1
glGenTextures(1, &texture1);
glBindTexture(GL_TEXTURE_2D, texture1);
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_REPEAT);
// set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
```

```
                    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                        GL_LINEAR);
                    // load image, create texture and generate mipmaps
                    int width, height, nrChannels;
                    unsigned char* image = SOIL_load_image("/root/
                        SourceCodes/CPP/images/sample.png", &width, &height,
                         0, SOIL_LOAD_RGB);
                    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
                         GL_RGB, GL_UNSIGNED_BYTE, image);

                    if (image)
                    {
                           glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width,
                               height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
                           glGenerateMipmap(GL_TEXTURE_2D);
                    }
                    else
                    {
                           std::cout << "Failed to load texture" << std::
                               endl;
                    }
                    SOIL_free_image_data(image);
                    // texture 2
                    glGenTextures(1, &texture2);
                    glBindTexture(GL_TEXTURE_2D, texture2);
                    // set the texture wrapping parameters
                    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                        GL_REPEAT);
                    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                        GL_REPEAT);
                    // set texture filtering parameters
                    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR);
                    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                        GL_LINEAR);
                    // load image, create texture and generate mipmaps
                    unsigned char* image2 = SOIL_load_image("/root/
                        SourceCodes/CPP/images/sample.png", &width, &height,
                         0, SOIL_LOAD_RGB);
                    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
                         GL_RGB, GL_UNSIGNED_BYTE, image);

                    if (image2)
                    {
                           // note that the awesomeface.png has transparency
                                and thus an alpha channel, so make sure to
                               tell OpenGL the data type is of GL_RGBA
                           glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width,
```

```
                height, 0, GL_RGBA, GL_UNSIGNED_BYTE, image2)
                ;
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else
    {
        std::cout << "Failed to load texture" << std::
            endl;
    }
    SOIL_free_image_data(image2);

    // tell opengl for each sampler to which texture unit it
        belongs to (only has to be done once)
    ourShader.use();
    ourShader.setInt("texture1", 0);
    ourShader.setInt("texture2", 1);


    // render loop
    // -----------
    while (!glfwWindowShouldClose(window))
    {
        // per-frame time logic
        // --------------------
        float currentFrame = static_cast<float>(
            glfwGetTime());
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        // input
        // -----
        processInput(window);

        // render
        // ------
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
            );

        // bind textures on corresponding texture units
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, texture1);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, texture2);

        // activate shader
        ourShader.use();
```

```
                        // pass projection matrix to shader (note that in
                             this case it could change every frame)
                        glm::mat4 projection = glm::perspective(glm::
                            radians(fov), (float)SCR_WIDTH / (float)
                            SCR_HEIGHT, 0.1f, 100.0f);
                        ourShader.setMat4("projection", projection);

                        // camera/view transformation
                        glm::mat4 view = glm::lookAt(cameraPos, cameraPos
                             + cameraFront, cameraUp);
                        ourShader.setMat4("view", view);

                        // render boxes
                        glBindVertexArray(VAO);
                        for (unsigned int i = 0; i < 10; i++)
                        {
                               // calculate the model matrix for each
                                   object and pass it to shader before
                                   drawing
                               glm::mat4 model = glm::mat4(1.0f); // make
                                   sure to initialize matrix to identity
                                   matrix first
                               model = glm::translate(model, cubePositions
                                   [i]);
                               float angle = 20.0f * i;
                               model = glm::rotate(model, glm::radians(
                                   angle), glm::vec3(1.0f, 0.3f, 0.5f));
                               ourShader.setMat4("model", model);

                               glDrawArrays(GL_TRIANGLES, 0, 36);
                        }

                        // glfw: swap buffers and poll IO events (keys
                            pressed/released, mouse moved etc.)

                        glfwSwapBuffers(window);
                        glfwPollEvents();
                }

                // optional: de—allocate all resources once they've
                    outlived their purpose:

                glDeleteVertexArrays(1, &VAO);
                glDeleteBuffers(1, &VBO);

                // glfw: terminate, clearing all previously allocated
                    GLFW resources.
```

```
        glfwTerminate();
        return 0;
}

// process all input: query GLFW whether relevant keys are
    pressed/released this frame and react accordingly

void processInput(GLFWwindow *window)
{
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

        float cameraSpeed = static_cast<float>(2.5 * deltaTime);
        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront,
            cameraUp)) * cameraSpeed;
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront,
            cameraUp)) * cameraSpeed;
}

// glfw: whenever the window size changed (by OS or user resize
    ) this callback function executes

void framebuffer_size_callback(GLFWwindow* window, int width,
    int height)
{
        // make sure the viewport matches the new window
            dimensions; note that width and
        // height will be significantly larger than specified on
             retina displays.
        glViewport(0, 0, width, height);
}

// glfw: whenever the mouse moves, this callback is called

void mouse_callback(GLFWwindow* window, double xposIn, double
    yposIn)
{
        float xpos = static_cast<float>(xposIn);
        float ypos = static_cast<float>(yposIn);

        if (firstMouse)
        {
```

```
            lastX = xpos;
            lastY = ypos;
            firstMouse = false;
        }

        float xoffset = xpos - lastX;
        float yoffset = lastY - ypos; // reversed since y-
            coordinates go from bottom to top
        lastX = xpos;
        lastY = ypos;

        float sensitivity = 0.1f; // change this value to your
            liking
        xoffset *= sensitivity;
        yoffset *= sensitivity;

        yaw += xoffset;
        pitch += yoffset;

        // make sure that when pitch is out of bounds, screen
            doesn't get flipped
        if (pitch > 89.0f)
        pitch = 89.0f;
        if (pitch < -89.0f)
        pitch = -89.0f;

        glm::vec3 front;
        front.x = cos(glm::radians(yaw)) * cos(glm::radians(
            pitch));
        front.y = sin(glm::radians(pitch));
        front.z = sin(glm::radians(yaw)) * cos(glm::radians(
            pitch));
        cameraFront = glm::normalize(front);
}

// glfw: whenever the mouse scroll wheel scrolls, this callback
     is called

void scroll_callback(GLFWwindow* window, double xoffset, double
     yoffset)
{
        fov -= (float)yoffset;
        if (fov < 1.0f)
        fov = 1.0f;
        if (fov > 45.0f)
        fov = 45.0f;
}
```

C++ Code 15: *main.cpp "Cube and Moving Camera"*

We are only using GLAD, SOIL, GLFW and of course OpenGL. There are two other files we need to make, the Vertex Shader and Fragment Shader, GLSL files

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoord;

// texture samplers
uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
       // linearly interpolate between both textures
       FragColor = mix(texture(texture1, TexCoord), texture(
           texture2, TexCoord), 0.2);
}
```

C++ Code 16: *camera.fs "Cube and Moving Camera"*

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
       gl_Position = projection * view * model * vec4(aPos, 1.0
           f);
       TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

C++ Code 17: *camera.vs "Cube and Moving Camera"*

There is one thing you need to setup (you can skip this step if you can hack the error connecting to this), since this example is gained from [6], thus you need to build LearnOpenGL that you clone from the github with CMake and then copy **.../LearnOpenGL/build/configuration/root_directory.h** into **/usr/lib**. After all preparations are made, now open the terminal at the current working directory, and type:
g++ main.cpp -o result /root/SourceCodes/CPP/src/glad.c -lSOIL -lglfw -lGL
./result

8. **C++ Example of Rotating 3D Cube with Keyboard Events, GLM and GLFW**

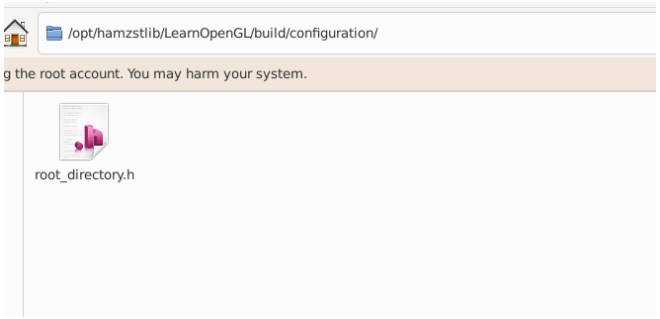

Figure 2.11: *You need to put **root_directory.h** into **/usr/include**, this file is obtained after building LearnOpenGL examples [6], you can find this inside the build folder **.../LearnOpenGL/build/configuration**.*

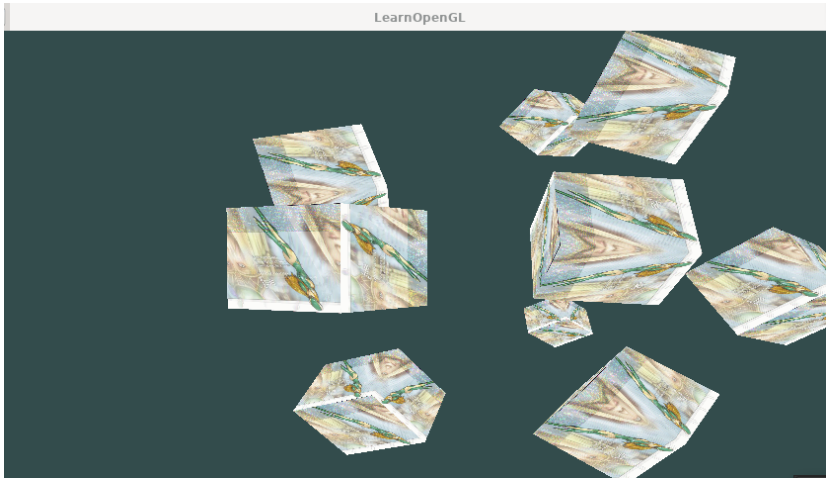

Figure 2.12: *You can move the camera by using WASD keyboard keys, and use mouse to zoom and rotate camera as well. In Physics and Math simulation we can do this when the computation is already converging or while the physics process is still running to make it more fun to see from different point of view (all files for this example are in the folder: **ch2-example6-Cube and Moving Camera**).*

In this example we are going to create rotating cube centered at the origin, inspired from Chapter 4 of this book [3], the rotational movements are:

(a) Yaw, rotating toward y axis (rotational movement between x axis and z axis).

(b) Roll, rotating toward z axis (rotational movement between x axis and y axis)

(c) Pitch, rotating toward x axis (rotational movement between y axis and z axis).

This is the basic of flight simulator, movement in 3D game, and for fluid flows as well. In a new directory, create a file by opening a terminal and type:
vim main.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include "Utils.h"

using namespace std;

#define numVAOs 1
#define numVBOs 2

// camera for movement with keyboard
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

// timing
float deltaTime = 0.0f; // time between current frame and last
    frame
float lastFrame = 0.0f;

float cameraX, cameraY, cameraZ;
float cubeLocX, cubeLocY, cubeLocZ;
GLuint renderingProgram;
GLuint vao[numVAOs];
GLuint vbo[numVBOs];

GLuint mvLoc, projLoc;
int width, height;
float aspect;
glm::mat4 pMat, vMat, mMat, mvMat, rxMat, ryMat, rzMat, rxvMat,
     ryvMat, rzvMat;
```

```
void setupVertices(void) { // 36 vertices, 12 triangles, makes
    2x2x2 cube placed at origin
       float vertexPositions[108] = {
              -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f,
                   -1.0f, -1.0f,
              1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f,
                  1.0f, -1.0f,
              1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
                  1.0f, -1.0f,
              1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
                  -1.0f,
              1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f,
                  1.0f, 1.0f,
              -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f,
                  1.0f, 1.0f,
              -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f
                  , 1.0f, 1.0f,
              -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f
                  , 1.0f, 1.0f,
              -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
                  -1.0f, -1.0f,
              1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f
                  , -1.0f, 1.0f,
              -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
                  1.0f, 1.0f,
              1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f
                  , -1.0f,
       };
       glGenVertexArrays(1,vao);
       glBindVertexArray(vao[0]);
       glGenBuffers(numVBOs, vbo);

       glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
       glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions),
           vertexPositions, GL_STATIC_DRAW);
}

void init(GLFWwindow* window){

       // Utils
       renderingProgram = Utils::createShaderProgram(
       "vertShader.glsl",
       "fragShader.glsl");
       cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
       cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f;
       setupVertices();
}
```

```
void display(GLFWwindow* window, double currentTime) {
       glClear(GL_DEPTH_BUFFER_BIT);
       glClear(GL_COLOR_BUFFER_BIT);
       glUseProgram(renderingProgram);

       // get the uniform variables for the MV and projection
           matrices
       mvLoc = glGetUniformLocation(renderingProgram, "
           mv_matrix");
       projLoc = glGetUniformLocation(renderingProgram, "
           proj_matrix");

       // build perspective matrix
       glfwGetFramebufferSize(window, &width, &height);
       aspect = (float)width / (float)height;
       pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);
            // 1.0472 radians = 60 degrees

       // build view matrix, model matrix, and model-view
           matrix
       vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-
           cameraX, -cameraY, -cameraZ));
       mMat = glm::translate(glm::mat4(1.0f), glm::vec3(
           cubeLocX, cubeLocY, -cubeLocZ));
       mvMat = vMat * mMat;

       if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
       {
              rxMat = glm::rotate(glm::mat4(1.0f), (float)
                  glfwGetTime(), glm::vec3(-1.0f, 0.0f, 0.0f));
              // Pitch movement toward user positive z axis
                  between y axis and z axis
              rxvMat = vMat * rxMat;
              glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                  value_ptr(rxvMat));
              glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                  value_ptr(pMat));

              // associate VBO with the corresponding vertex
                  attribute in the vertex shader
              glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
              glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                   0);
              glEnableVertexAttribArray(0);

              // adjust OpenGL settings and draw model
              glEnable(GL_DEPTH_TEST);
              glDepthFunc(GL_LEQUAL);
```

```
                glDrawArrays(GL_TRIANGLES,0,36);
            }
            if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
            {
                rxMat = glm::rotate(glm::mat4(1.0f), (float)
                    glfwGetTime(), glm::vec3(1.0f, 0.0f, 0.0f));
                // Pitch movement toward monitor negative z axis
                    between y axis and z axis
                rxvMat = vMat * rxMat;
                glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                    value_ptr(rxvMat));
                glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                    value_ptr(pMat));

                // associate VBO with the corresponding vertex
                    attribute in the vertex shader
                glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
                glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                     0);
                glEnableVertexAttribArray(0);

                // adjust OpenGL settings and draw model
                glEnable(GL_DEPTH_TEST);
                glDepthFunc(GL_LEQUAL);
                glDrawArrays(GL_TRIANGLES,0,36);
            }
            if (glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS)
            {
                rzMat = glm::rotate(glm::mat4(1.0f), (float)
                    glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
                // Roll movement counter clockwise between x axis
                     and y axis
                rzvMat = vMat * rzMat;
                glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                    value_ptr(rzvMat));
                glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                    value_ptr(pMat));

                // associate VBO with the corresponding vertex
                    attribute in the vertex shader
                glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
                glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                     0);
                glEnableVertexAttribArray(0);

                // adjust OpenGL settings and draw model
                glEnable(GL_DEPTH_TEST);
                glDepthFunc(GL_LEQUAL);
```

```
                glDrawArrays(GL_TRIANGLES,0,36);
            }
            if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS)
            {
                rzMat = glm::rotate(glm::mat4(1.0f), (float)
                    glfwGetTime(), glm::vec3(0.0f, 0.0f, -1.0f));
                // Roll movement clockwise between x axis and y
                    axis
                rzvMat = vMat * rzMat;
                glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                    value_ptr(rzvMat));
                glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                    value_ptr(pMat));

                // associate VBO with the corresponding vertex
                    attribute in the vertex shader
                glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
                glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                     0);
                glEnableVertexAttribArray(0);

                // adjust OpenGL settings and draw model
                glEnable(GL_DEPTH_TEST);
                glDepthFunc(GL_LEQUAL);
                glDrawArrays(GL_TRIANGLES,0,36);
            }
            if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
            {
                ryMat = glm::rotate(glm::mat4(1.0f), (float)
                    glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));
                // Yaw movement counter clockwise between x axis
                    and z axis.
                ryvMat = vMat * ryMat;
                glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                    value_ptr(ryvMat));
                glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                    value_ptr(pMat));

                // associate VBO with the corresponding vertex
                    attribute in the vertex shader
                glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
                glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                     0);
                glEnableVertexAttribArray(0);

                // adjust OpenGL settings and draw model
                glEnable(GL_DEPTH_TEST);
                glDepthFunc(GL_LEQUAL);
```

```
                glDrawArrays(GL_TRIANGLES,0,36);
        }
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        {
                ryMat = glm::rotate(glm::mat4(1.0f), (float)
                    glfwGetTime(), glm::vec3(0.0f, -1.0f, 0.0f));
                // Yaw movement clockwise between x axis and z
                    axis.
                ryvMat = vMat * ryMat;
                glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                    value_ptr(ryvMat));
                glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                    value_ptr(pMat));

                // associate VBO with the corresponding vertex
                    attribute in the vertex shader
                glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
                glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                     0);
                glEnableVertexAttribArray(0);

                // adjust OpenGL settings and draw model
                glEnable(GL_DEPTH_TEST);
                glDepthFunc(GL_LEQUAL);
                glDrawArrays(GL_TRIANGLES,0,36);
        }
        // copy perspective and MV matrices to corresponding
            uniform variables
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(
            pMat));

        // associate VBO with the corresponding vertex attribute
             in the vertex shader
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0, 0);
        glEnableVertexAttribArray(0);

        // adjust OpenGL settings and draw model
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LEQUAL);
        glDrawArrays(GL_TRIANGLES,0,36);
}

void processInput(GLFWwindow *window)
{
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        {
                glfwSetWindowShouldClose(window, true);
```

```
        }
}


int main(void)
{
        glfwInit(); //initialize GLFW and GLEW libraries
        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
        glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
        glfwWindowHint(GLFW_OPENGL_PROFILE,
            GLFW_OPENGL_CORE_PROFILE);

        GLFWwindow* window = glfwCreateWindow(600, 600, "Pitch,
            Yaw, Roll for Interpolated Color Cube", NULL, NULL);
        glfwMakeContextCurrent(window);
        //glfwSetKeyCallback(window, updateKeyboard);
        if(glewInit() != GLEW_OK) {
                exit(EXIT_FAILURE);
        }
        glfwSwapInterval(1);

        init(window);

        while(!glfwWindowShouldClose(window)) {
                // per-frame time logic
                // --------------------
                float currentFrame = static_cast<float>(
                    glfwGetTime());
                deltaTime = currentFrame - lastFrame;
                lastFrame = currentFrame;

                processInput(window);

                display(window, glfwGetTime());
                glfwSwapBuffers(window);
                glfwPollEvents();
        }
        glfwDestroyWindow(window);
        glfwTerminate();
        exit(EXIT_SUCCESS);
        //return 0;
}
```

C++ Code 18: *main.cpp "Yaw Pitch Roll for 3D Cube"*

Then create the Vertex Shader, Fragment Shader, Utils file and header too.

```
#version 330
```

```
out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

in vec4 varyingColor;

void main(void) {
	color = varyingColor;
}
```

C++ Code 19: *fragShader.glsl "Yaw Pitch Roll for 3D Cube"*

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
//#include <SOIL2/SOIL2.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp> // glm::value_ptr
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::
    rotate, glm::scale, glm::perspective
#include "Utils.h"
using namespace std;

Utils::Utils() {}

string Utils::readShaderFile(const char *filePath) {
	string content;
	ifstream fileStream(filePath, ios::in);
	string line = "";
	while (!fileStream.eof()) {
		getline(fileStream, line);
		content.append(line + "\n");
	}
	fileStream.close();
	return content;
}

bool Utils::checkOpenGLError() {
	bool foundError = false;
	int glErr = glGetError();
	while (glErr != GL_NO_ERROR) {
		cout << "glError: " << glErr << endl;
		foundError = true;
		glErr = glGetError();
	}
```

```
        return foundError;
}

void Utils::printShaderLog(GLuint shader) {
        int len = 0;
        int chWrittn = 0;
        char *log;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
        if (len > 0) {
                log = (char *)malloc(len);
                glGetShaderInfoLog(shader, len, &chWrittn, log);
                cout << "Shader Info Log: " << log << endl;
                free(log);
        }
}

GLuint Utils::prepareShader(int shaderTYPE, const char *
    shaderPath)
{
        GLint shaderCompiled;
        string shaderStr = readShaderFile(shaderPath);
        const char *shaderSrc = shaderStr.c_str();
        GLuint shaderRef = glCreateShader(shaderTYPE);
        glShaderSource(shaderRef, 1, &shaderSrc, NULL);
        glCompileShader(shaderRef);
        checkOpenGLError();
        glGetShaderiv(shaderRef, GL_COMPILE_STATUS, &
            shaderCompiled);
        if (shaderCompiled != 1)
        {
                if (shaderTYPE == 35633) cout << "Vertex ";
                if (shaderTYPE == 36488) cout << "Tess Control ";
                if (shaderTYPE == 36487) cout << "Tess Eval ";
                if (shaderTYPE == 36313) cout << "Geometry ";
                if (shaderTYPE == 35632) cout << "Fragment ";
                cout << "shader compilation error." << endl;
                printShaderLog(shaderRef);
        }
        return shaderRef;
}

void Utils::printProgramLog(int prog) {
        int len = 0;
        int chWrittn = 0;
        char *log;
        glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
        if (len > 0) {
                log = (char *)malloc(len);
```

```
                glGetProgramInfoLog(prog, len, &chWrittn, log);
                cout << "Program Info Log: " << log << endl;
                free(log);
        }
}

int Utils::finalizeShaderProgram(GLuint sprogram)
{
        GLint linked;
        glLinkProgram(sprogram);
        checkOpenGLError();
        glGetProgramiv(sprogram, GL_LINK_STATUS, &linked);
        if (linked != 1)
        {
                cout << "linking failed" << endl;
                printProgramLog(sprogram);
        }
        return sprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
        GLuint vfprogram = glCreateProgram();
        glAttachShader(vfprogram, vShader);
        glAttachShader(vfprogram, fShader);
        finalizeShaderProgram(vfprogram);
        return vfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    gp, const char *fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
        GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
        GLuint vgfprogram = glCreateProgram();
        glAttachShader(vgfprogram, vShader);
        glAttachShader(vgfprogram, gShader);
        glAttachShader(vgfprogram, fShader);
        finalizeShaderProgram(vgfprogram);
        return vgfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    tCS, const char* tES, const char *fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER,
```

```
                tCS);
        GLuint teShader = prepareShader(
            GL_TESS_EVALUATION_SHADER, tES);
        GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
        GLuint vtfprogram = glCreateProgram();
        glAttachShader(vtfprogram, vShader);
        glAttachShader(vtfprogram, tcShader);
        glAttachShader(vtfprogram, teShader);
        glAttachShader(vtfprogram, fShader);
        finalizeShaderProgram(vtfprogram);
        return vtfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    tCS, const char* tES, char *gp, const char *fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER,
            tCS);
        GLuint teShader = prepareShader(
            GL_TESS_EVALUATION_SHADER, tES);
        GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
        GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
        GLuint vtgfprogram = glCreateProgram();
        glAttachShader(vtgfprogram, vShader);
        glAttachShader(vtgfprogram, tcShader);
        glAttachShader(vtgfprogram, teShader);
        glAttachShader(vtgfprogram, gShader);
        glAttachShader(vtgfprogram, fShader);
        finalizeShaderProgram(vtgfprogram);
        return vtgfprogram;
}

// GOLD material - ambient, diffuse, specular, and shininess
float* Utils::goldAmbient() { static float a[4] = { 0.2473f,
    0.1995f, 0.0745f, 1 }; return (float*)a; }
float* Utils::goldDiffuse() { static float a[4] = { 0.7516f,
    0.6065f, 0.2265f, 1 }; return (float*)a; }
float* Utils::goldSpecular() { static float a[4] = { 0.6283f,
    0.5559f, 0.3661f, 1 }; return (float*)a; }
float Utils::goldShininess() { return 51.2f; }

// SILVER material - ambient, diffuse, specular, and shininess
float* Utils::silverAmbient() { static float a[4] = { 0.1923f,
    0.1923f, 0.1923f, 1 }; return (float*)a; }
float* Utils::silverDiffuse() { static float a[4] = { 0.5075f,
    0.5075f, 0.5075f, 1 }; return (float*)a; }
float* Utils::silverSpecular() { static float a[4] = { 0.5083f,
     0.5083f, 0.5083f, 1 }; return (float*)a; }
```

```
float Utils::silverShininess() { return 51.2f; }

// BRONZE material — ambient, diffuse, specular, and shininess
float* Utils::bronzeAmbient() { static float a[4] = { 0.2125f,
    0.1275f, 0.0540f, 1 }; return (float*)a; }
float* Utils::bronzeDiffuse() { static float a[4] = { 0.7140f,
    0.4284f, 0.1814f, 1 }; return (float*)a; }
float* Utils::bronzeSpecular() { static float a[4] = { 0.3936f,
     0.2719f, 0.1667f, 1 }; return (float*)a; }
float Utils::bronzeShininess() { return 25.6f; }
```

C++ Code 20: *Utils.cpp "Yaw Pitch Roll for 3D Cube"*

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

class Utils
{
       private:
       static std::string readShaderFile(const char *filePath);
       static void printShaderLog(GLuint shader);
       static void printProgramLog(int prog);
       static GLuint prepareShader(int shaderTYPE, const char *
           shaderPath);
       static int finalizeShaderProgram(GLuint sprogram);

       public:
       Utils();
       static bool checkOpenGLError();
       static GLuint createShaderProgram(const char *vp, const
           char *fp);
       static GLuint createShaderProgram(const char *vp, const
           char *gp, const char *fp);
       static GLuint createShaderProgram(const char *vp, const
           char *tCS, const char* tES, const char *fp);
       static GLuint createShaderProgram(const char *vp, const
           char *tCS, const char* tES, char *gp, const char *fp
           );
       static GLuint loadTexture(const char *texImagePath);
       static GLuint loadCubeMap(const char *mapDir);
```

```
        static float* goldAmbient();
        static float* goldDiffuse();
        static float* goldSpecular();
        static float goldShininess();

        static float* silverAmbient();
        static float* silverDiffuse();
        static float* silverSpecular();
        static float silverShininess();

        static float* bronzeAmbient();
        static float* bronzeDiffuse();
        static float* bronzeSpecular();
        static float bronzeShininess();
};
```

C++ Code 21: *Utils.h "Yaw Pitch Roll for 3D Cube"*

```
#version 330

layout (location=0) in vec3 position;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out vec4 varyingColor;

void main(void) {
       gl_Position = proj_matrix * mv_matrix * vec4(position,
           1.0);
       varyingColor = vec4(position, 1.0) * 0.5 + vec4(0.5,
           0.5, 0.5, 0.5);
}
```

C++ Code 22: *vertShader.glsl "Yaw Pitch Roll for 3D Cube"*

After finish create all those files, now open the terminal at the current working directory, and type:
g++ *.cpp -o result -lGLEW -lglfw -lGL
./result

III. Know-How in C++

1. First, for the compiler we use g++ that is more suitable for C++ codes, to compile **.c** files you better use **gcc**.

2. To show warning messages type:
   g++ -o main main.cpp -Wall

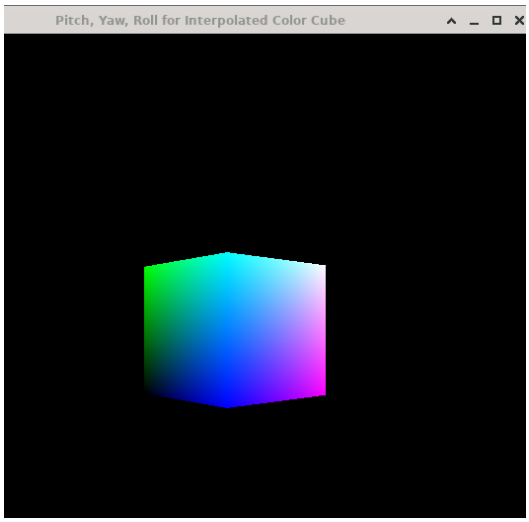

Figure 2.13: *You can move the box by using keyboard' keys W / S for pitch movement, keys A / D for yaw movement, and keys Z / C for roll movement. (all files for this example are in the folder:* ***ch2-example7-Yaw Pitch Roll for 3D Cube****).*

3. If you want to specify the name of the compiled executable file, do so by using the **-o** flag:
   g++ -o [name] [file to compile]

4. If you want to compile object files **object-1.o** and **object-2.o** into a **main** executable file, type:
   g++ -o main object-1.o object-2.o

5. If you want to specify a root directory, where libraries and headers can be found, use the **–sysroot** flag:
   g++ -o [name] –sysroot [directory] main.ccp

6. To create a static library, start by compiling an object file:
   g++ -o obj.o main.cpp

   use the ar utility with **rcs** to create an archive (**.a**) file:
   ar rcs archive.a obj.o

   Finally, use **g++**:
   g++ -o final example.cpp archive.a

7. You will always include header file from cpp source code file. For example inside **main.cpp** there will be **#include "Mesh.h"**.

8. To be able to reuse functions, we can create separate **.cpp** file (and an associated **.h**) instead of putting a lot of codes inside the main source code (**main.cpp**).

9. Conside a function **createShaderProgram()** that can be defined as:

   - **GLuint Utils::createShaderProgram(const char *vp, const char *fp)**
     Supports shader programs which utilize a vertex and fragment shader.

- **GLuint Utils::createShaderProgram(const char *vp, const char *gp, const char *fp)**
  Supports shader programs which utilize a vertex, geometry and fragment shader.
- **GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char *tES, const char *fp)**
  Supports shader programs which utilize a vertex, tesselation and fragment shader.
- **GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char *tES, const char *gp, const char *fp)**
  Supports shader programs which utilize a vertex, tesselation, geometry and fragment shader.

The parameters accepted in each case are pathnames for the GLSL files containing the shader code. An example of a completed program that is placed in the variable "renderingProgram": **renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl")** given that you put the two files "vertShader.glsl", "fragShader.glsl" in the working directory of your C++ project.

10. Learn about every libraries that are used, like **SOIL**, or **stb-image**, because some of them need different configuration at the C++ source code, for example you have to add: **#define STB_IMAGE_IMPLEMENTATION** to your code before all the other include (as the first line). This can be known from their documentation. To avoid getting this when compiling: **undefined reference to stbi_load stbi_image_free stbi_set_slip_vertically**

Chapter 3

Mathematics and Physics in Computer Graphics

"I'll rise up like the Eagle, and I will soar with you, your spirit lifts me on, with the power of Your Love" -
The Power of Your Love song

In this chapter I will only put minimal explanations, if you are interested and want to dig more you can read books related to Mathematics and Physics, read the one made by author from reputable university, published by world renowned publisher, never buy Indonesian published book especially in Indonesian language, they are destined to only say sweet words and be hypocrite all the time with corruption all over, trashes in nature, inefficient bureaucracy and government, law exists but not taken seriously. That's from Big Tree, a Nature in Valhalla Projection who is very angry today (October 20th, 2023) knowing the fact that just to take care for trashes that people throw in forests, those in corresponding bureau took too long for real action. Who is going to capture all those throw trashes in forests? Capture no matter the age, gender, social status. Don't create law if you can't apply it in reality to anyone no special treatment. That's why there will be no more human farmers or workers, all robotics farmers, more efficient, no trashes of cigarettes or alcohol. You know being farmers have to be grateful, don't drink alcohol, you are not miserable, you get free workout under the sun better for your body, out there more people suffer, such as being sex slave since child. Enough of that, well in the end this is all connected C++, Mathematics and Physics are great foundation to create robots. So here we go starting with computer graphics first.

I. Mathematics

The mathematics that will be used in computer graphics are (but not limited to):

1. Geometry (Coordinate Systems, Space transformations)
2. Linear Algebra (Vectors, Matrices)

i. Geometry

When we see the image on our desktop computer, what we see are actually made of pixel or point with different color. The monitor or display that is still used in 2023 is flat, but it can trick our

brain so we can see 3-dimensional world in there, when we play 3-D game or designing 3-D image or watching a movie.

In computer graphics term, we will call points as vertices. The first step of creating object is to draw the vertices, and then connecting them to create triangle, then group of triangles that will become object. A cube itself can be seen as combination of 12 triangles.

[DF*] **3D Coordinate System**
In order to define a position and movement, we will need a 3D coordinate system that can help us compute and determine the state or condition of the object at certain time period.

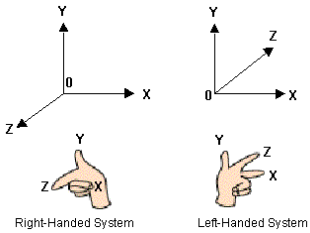

Figure 3.1: *The right-handed coordinate system is used by OpenGL and Vulkan, while the left-handed coordinate system is used by DirectX and Direct3D.*

[DF*] **Points**
With the defined coordinate system, we can specify the position of the object in 3D correctly. The origin is located at $(0, 0, 0)$.

[DF*] **Polygon**
In computer graphics, especially with OpenGL, all the rendered graphics are based on vertices that will be connected into triangle, then combining with other triangle that will become another polygon such as rectangle, cube, circle, sphere, etc. Since OpenGL conventionally uses a right-handed coordinate system, thus creating triangle vertices, they will have a counter-clockwise ordering.

ii. Linear Algebra

When an object is moving then we need to learn about vector, vector is a quantity that has direction and magnitude. The velocity is an example of vector.

[DF*] **Vectors**
Vectors can be added, multiplied and subtracted. Suppose we have vector $\vec{v}$ and $\vec{w}$, where $\vec{v} = (v_x, v_y, v_z)$ and $\vec{w} = (w_x, w_y, w_z)$, thus

$$\vec{v} + \vec{w} = (v_x + w_x, v_y + w_y, v_z + w_z)$$
$$\vec{v} - \vec{w} = (v_x - w_x, v_y - w_y, v_z - w_z)$$

To calculate the magnitude of the vector (or the length of the vector):

$$||\vec{v}|| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

with $||\vec{v}||$ is sometimes called norm, the magnitude of a vector is always greater than or equal to zero. In a lot of cases, we might only need the direction of the vector, thus we can

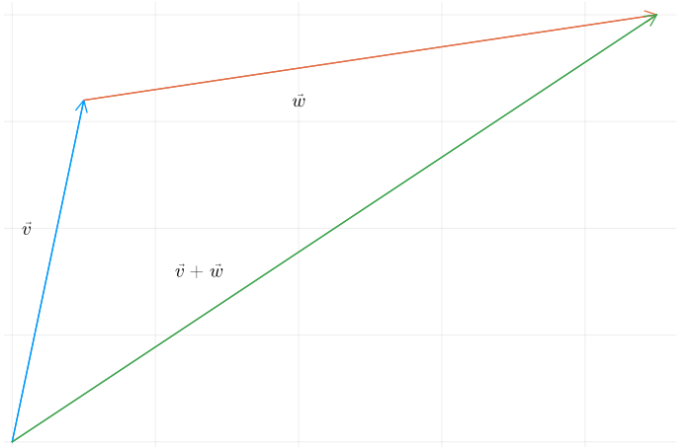

Figure 3.2: *The addition of two vectors in 2-dimension (ch3-linalg-vectoradd.jl).*

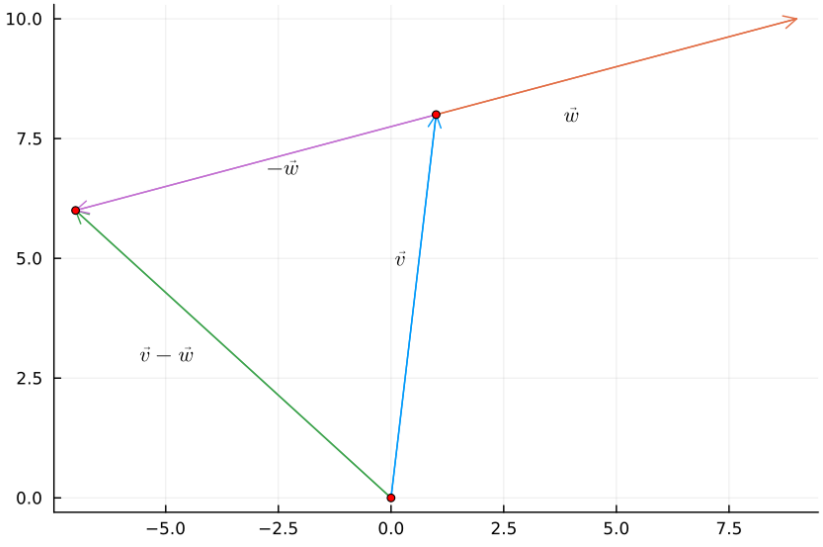

Figure 3.3: *The subtraction of two vectors in 2-dimension (ch3-linalg-vectorsubtraction.jl).*

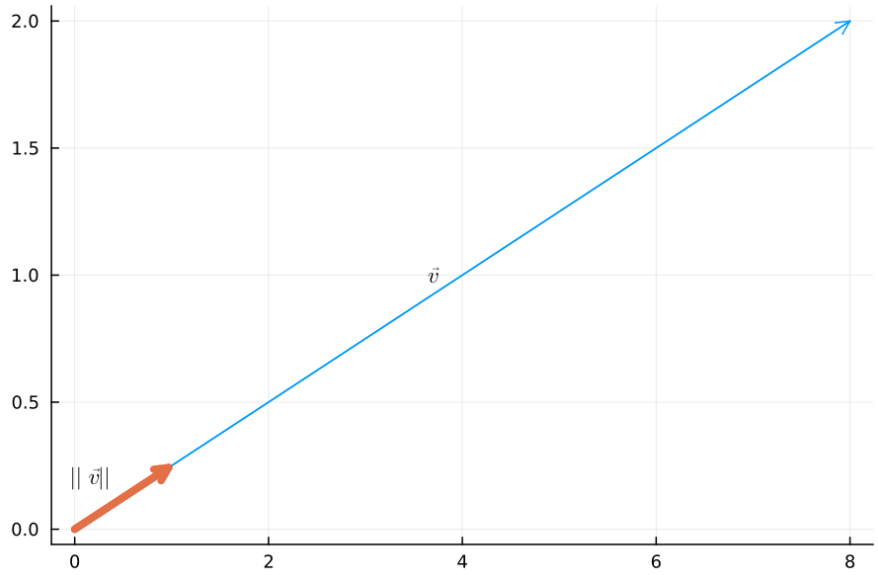

Figure 3.4: *The norm of vector $\vec{v}$ is bolded with orange color (ch3-linalg-vectornorm.jl).*

convert all vectors into unit vector by:

$$\hat{v} = \frac{\vec{v}}{||\vec{v}||} = \left(\frac{v_x}{||\vec{v}||}, \frac{v_y}{||\vec{v}||}, \frac{v_z}{||\vec{v}||}\right)$$

If we want to know the angle between two vectors, we will use dot product:

$$\begin{aligned}\vec{v} \cdot \vec{w} &= v_x w_x + v_y w_y + v_z w_z \\ &= ||\vec{v}||||\vec{w}|| \cos\ \theta\end{aligned}$$

thus

$$\theta = \cos^{-1} \frac{v_x w_x + v_y w_y + v_z w_z}{||\vec{v}||||\vec{w}||}$$

- If $\vec{v} \cdot \vec{w} = 0$, then $\vec{v}$ is perpendicular to $\vec{w}$.
- If $\vec{v} \cdot \vec{w} = ||\vec{v}||||\vec{w}||$, then the two vectors are parallel to each other.
- If $\vec{v} \cdot \vec{w} < 0$, then the angle between the two vectors is greater than 90^0.
- If $\vec{v} \cdot \vec{w} > 0$, then the angle between the two vectors is less than 90^0
- The commutative law applies to a scalar product, so

$$\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$$

We can perform a cross product(vector product) between two vectors only in 3-dimension. Consider

$$\vec{f} = \vec{v} \times \vec{w}$$

if we expand the cross product in unit-vector notation, we will obtain

$$\begin{aligned}\vec{v} \times \vec{w} &= (v_x\hat{i} + v_y\hat{j} + v_z\hat{k}) \times (w_x\hat{i} + w_y\hat{j} + w_z\hat{k}) \\ &= (v_y w_z - v_z w_y)\hat{i} + (v_z w_x - v_x w_z)\hat{j} + (v_x w_y - v_y w_x)\hat{k}\end{aligned}$$

To obtain the magnitude of $\vec{f}$:

$$\vec{f} = |\vec{v} \times \vec{w}| = v \cdot w \ \sin\phi$$

where ϕ is the smallest angle between $\vec{v}$ and $\vec{w}$.

We can obtains the direction for $\vec{f}$ by using our right hand, point all four fingers toward one direction and the thumb will be perpendicular toward the four fingers, then imagine that we sweep all our four fingers from the direction of $\vec{v}$ toward the direction of $\vec{w}$, the thumb will show the direction of $\vec{f}$.

Cross product is used widely for computer graphics (in lighting effects), since the resulting cross product is normal (perpendicular) to the plane defined by the original two vectors. How people walk on a surface, no matter how bumpy the road is, they will stand tall toward the normal of the surface while moving forward or do any activities.

- If $\vec{v}$ and $\vec{w}$ are parallel or antiparallel, then $\vec{v} \times \vec{w} = 0$.
- The magnitude of $\vec{v} \times \vec{w}$ is maximum when $\vec{v}$ and $\vec{w}$ are perpendicular to each other.
- The commutative law does not applies to a cross product, so

$$\vec{v} \times \vec{w} = -(\vec{w} \times \vec{v})$$

[DF*] **Matrices** In computer graphics, matrices are used to calculate object transforms such as translation (movement), scaling in the x, y, z-axis and rotation around the x, y, z-axis.

Matrices have rows and columns. A matrix $\boldsymbol{A}$ with m number of rows and n number of columns is a matrix of size $m \times n$, thus

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

each element of a matrix is represented as indices ij, thus it will be written a_{ij} to represent matrix index at ith-row, jth-column.
What you need to know and how to compute:

- Matrix addition and subtraction
- Matrix multiplication to a matrix with correct size
- Matrix multiplication to a vector with correct size and dimension
- Identity matrix, a square matrix of size n with indices $a_{ii} = 1$ with $i = 1, 2, \dots, n$, and 0 elsewhere.
- Determine the transpose of a matrix
- Determine the inverse of a matrix

[DF*] In computer graphics, matrices are used for performing transformations on objects, such as:

- Translation
- Rotation
- Scale
- Projection
- Look-At

All transformation matrices have the size of 4×4.

Translation
Consider the initial point of an object in 3D is (x, y, z), and want to be translated to $(x + t_x, y + t_y, z + t_z)$ then the translation matrix transform is

$$\begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.1}$$

The functions for building translation matrices with GLM are:
glm::translate(x,y,z)
mat4 * vec4

If we want to translate a vector of $(1, 0, 0)$ by $(1, 2, 3)$ the code is:

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
	glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f); // vector (1,0,0)
	glm::mat4 translation = glm::mat4(1.0f);
	translation = glm::translate(translation, glm::vec3(1.0f
	    , 2.0f, 3.0f));
	vec = translation * vec;
	std::cout << "(" << vec.x << ","<< vec.y << "," << vec.z
	     << ")" << std::endl;
}
```

You can easily compile and run it from terminal with:
g++ main.cpp -o result
./result

Scaling
A scale matrix is used to change the size of objects or move points toward or away from the origin.

Consider the initial point of an object in 3D is (x, y, z), and want to be scaled to (s_x, s_y, s_z) then the scaling matrix transform is

$$\begin{bmatrix} x * s_x \\ y * s_y \\ z * s_z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.2}$$

If we want to scale a vector of $(4,3,2)$ by $(3,3,3)$ / enlarging it 3 times its' original size, the code is:

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
       glm::vec4 vec(4.0f, 3.0f, 2.0f, 1.0f); // vector (4,3,2)
       glm::mat4 scaling = glm::mat4(1.0f);
       scaling = glm::scale(scaling, glm::vec3(3.0f, 3.0f, 3.0f
           )); // enlarge 3 times
       vec = scaling * vec;
       std::cout << "(" << vec.x << ","<< vec.y << "," << vec.z
            << ")" << std::endl;
}
```

You can easily compile and run it from terminal with:
g++ main.cpp -o result
./result

Scaling can also be used to switch coordinate system, by negating the z coordinate, thus the scale matrix transform is

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

Rotation
To rotate an object in 3D space requires specifying:

1. An axis of rotation (toward x axis will be called yaw, toward y axis will be called roll, and toward z axis will be called pitch)
2. A rotation amount in degrees or radians

The rotation matrix transform around x by θ:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\,\theta & -\sin\,\theta & 0 \\ 0 & \sin\,\theta & \cos\,\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.3}$$

The rotation matrix transform around y by ϕ:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\,\phi & 0 & \sin\,\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\,\phi & 0 & \cos\,\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.4}$$

The rotation matrix transform around z by φ:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\ \varphi & -\sin\ \varphi & 0 & 0 \\ \sin\ \varphi & \cos\ \varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.5}$$

From a simple observation above, the rotation toward x axis won't change the value of the x part, the same for rotation toward y axis, the value of y stays, and it occurs as well for rotation towards z axis.

If you want to rotate a vector of $(1,2,3)$ 90 degree toward the x, y, and z axis separately, the code is

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
	glm::vec4 vecx(1.0f, 2.0f, 3.0f, 1.0f); // vector
	    (1,2,3)
	glm::vec4 vecy(1.0f, 2.0f, 3.0f, 1.0f); // vector
	    (1,2,3)
	glm::vec4 vecz(1.0f, 2.0f, 3.0f, 1.0f); // vector
	    (1,2,3)


	glm::mat4 rotationx = glm::mat4(1.0f);
	glm::mat4 rotationy = glm::mat4(1.0f);
	glm::mat4 rotationz = glm::mat4(1.0f);
	rotationx = glm::rotate(rotationx, glm::radians(90.0f),
	    glm::vec3(1.0f, 0.0f, 0.0f)); // rotate 90 degrees
	    toward x axis
	vecx = rotationx * vecx;

	rotationy = glm::rotate(rotationy, glm::radians(90.0f),
	    glm::vec3(0.0f, 1.0f, 0.0f)); // rotate 90 degrees
	    toward y axis
	vecy = rotationy * vecy;

	rotationz = glm::rotate(rotationz, glm::radians(90.0f),
	    glm::vec3(0.0f, 0.0f, 1.0f)); // rotate 90 degrees
	    toward z axis
	vecz = rotationz * vecz;

	std::cout << "Original␣vector:␣(1,2,3)" << std::endl;
```

```
                    std::cout << "Rotation 90 degrees toward x axis:" << std
                        ::endl;
                    std::cout << "(" << vecx.x << ","<< vecx.y << "," <<
                        vecx.z << ")" << std::endl;
                    std::cout << "Rotation 90 degrees toward y axis:" << std
                        ::endl;
                    std::cout << "(" << vecy.x << ","<< vecy.y << "," <<
                        vecy.z << ")" << std::endl;
                    std::cout << "Rotation 90 degrees toward z axis:" << std
                        ::endl;
                    std::cout << "(" << vecz.x << ","<< vecz.y << "," <<
                        vecz.z << ")" << std::endl;
            }
```

You can easily compile and run it from terminal with:
g++ main.cpp -o result
./result
[DF*]

II. Physics

The physics that will be used in computer graphics are (but not limited to):

1. Gravity
2. Collision, Newton's Laws
3. Oscillations
4. Angular Rotation, velocity
5. Moment of inertia and torque

To make the object realistic, we need to add the physics otherwise without physics libraries / engine, the cube we render with OpenGL can just walk through a wall, is it possible for such ghost collision in this real life? That's why we need a Physics engine when we create an object after we render the shape and giving the color or texture to it. More detail on the physics formula and explanations can be read at the corresponding chapter, along with the C++ codes to create the simulation.

Chapter 4

OpenGL, SFML, GLFW, GLEW, And All That

*"Love is composed of a single soul inhabiting two bodies/entities" - **Aristotle***

*"The best and most beautiful things in the world cannot be seen or even touched - they must be felt with the heart" - **Helen Keller***

OpenGL can't stand alone to show the image or animation that we create with C++ source codes. OpenGL only renders to a frame buffer, we need additional library to draw the contents of the frame buffer onto a window on the screen.

The packages used in this books to make the simulator are available in my github repository: **https://github.com/glanzkaiser/DFSimulatorC/Source Codes/Libraries**

I. OpenGL

OpenGL is an API(Application Programming Interface) that provides us with a large set of functions to manipulate graphics and images. Each graphics card that you buy supports specific versions of OpenGL, in my example with Dell Precision 6510, I use OpenGL 3.3 that is supported by my graphics card.

OpenGL's API is written in C, the calls are directly compatible with C and C++. C++ application / source code that calls for OpenGL will run on CPU.

The good news is all future versions of OpenGL starting from 3.3 add extra useful features to OpenGL without changing OpenGL's core mechanics; the newer versions just introduce slightly more efficient ways to accomplish the same tasks.

For example, OpenGL 4.0 has a new addition of tessellator that can generate a large number of triangles, typically as a grid, we are able to create square area surface or curved surface as a terrain.

In all three major desktop platforms (Linux, macOS, and Windows), OpenGL more or less comes with the system. However, you will need to ensure that you have downloaded and installed

a recent driver for your graphics hardware.

OpenGL is entirely hardware and operating system independent, whether you are using nVIDIA GeForce or AMD Radeon graphics card, it will work the same on both hardware. The way in which OPenGL's features work is defined by a specification that is used by graphics hardware manufacturers while they're developing the drivers for their hardware. This is why we sometimes have to update the graphics hardware drivers if something doesn't look right.

Since GFreya OS is based on Linux, graphics on Linux is almost exclusively implemented using the X Window system. Supporting OpenGL on Linux involves using GLX extensions to the X Server.

GFreya OS installs OpenGL through MESA version-21.2.1, Mesa is an OpenGL compatible 3D graphics library, you can read more here:
https://www.linuxfromscratch.org/blfs/view/11.0/x/mesa.html

OpenGL by itself is only capable of drawing a few primitives: points, lines, and triangles. Most 3D models made by OpenGL are made up from lots of those primitives. Primitives are consisted of vertices.

Since we have to display 3D world on a 2D monitor, OpenGL do that job with vertices, triangles, color. The 2D monitor screen is made up of a raster - a rectangular array of pixels. When a 3D object is rasterized, OpenGL converts the primitives in the object into fragments. A fragment holds the information associated with a pixel. Rasterization determines the locations of pixels that need to be drawn in order to produce the triangle specified by its three vertices.

Facts about OpenGL:

- All OpenGL functions start with the **gl** prefix.
- For an object to be drawn, its vertices must be sent to the vertex shader. Vertices are usually sent by putting them in a buffer with a vertex attribute declared in the shader.

  Some that need to be done once (typically in **init()**):

  1. Create a buffer (or created in a function called by **init()**).
  2. Copy the vertices into the buffer.

  Some that need to be done at each frame (typically in **display()**):

  1. Enable the buffer containing the vertices.
  2. Associate the buffer with a vertex attribute.
  3. Enable the vertex attribute.
  4. Use **glDrawArrays(...)** to draw the object.
- In OpenGL, a buffer is contained in a Vertex Buffer Object (VBO), which is declared or instantiated in the C++/OpenGL application. Vertex Buffer Object (VBO) is the geometrical information, it includes attributes such as position, color, normal, and texture coordinates. These are stored on a per vertex basis on the GPU. We can also see VBO as the manner in which we load the vertices of a model into a buffer.
- Element Buffer Object (EBO) is used to store the index of each vertex and will be used while drawing the mesh.

- Vertex Array Object (VAO) is a helper container object that stores all the VBOs and attributes. This is used as you may have more than one VBO per object, and it would be tedious to bind the VBOs all over again when you render each frame. One VAO is required by OpenGL to be created, it is good as a way of organizing buffers and making them easier to manipulate in complex scenes.

  For example, suppose that we wish to display two objects, then:

```
GLuint vao[1];
GLuint vbo[2];
...
glGenVertexArrays(1,vao);
glBindVertexArray(vao[0]);
glGenBuffers(2,vbo);
```

  The command **glGenVertexArrays(1,vao)** creates VAOs, and the command **glGenBuffers(2,vbo)** creates VBOs, both commands return integer IDs for them.
  A buffer needs to have a corresponding vertex attribute variable declared in the vertex shader. Vertex attributes are generally the first variables declared in a shader.
- Buffers are used to store information in the GPU memory for fast and efficient access to the data. Modern GPUs have a memory bandwidth of approximately 600 GB/s, quite enormous compared to the current high-end CPUs that only have approximately 12 GB/s. Buffer objects are used to store, retrieve, and move data. It is very easy to generate a buffer object in OpenGL. You can generate one by calling **glGenBUffers()**.
- OpenGL has its own data types, they are prefixed with GL, followed by the data type.

C Type	Bitdepth	Description	Common Enum
GLboolean	1+	A boolean value, either GL_TRUE or GL_FALSE	
GLbyte	8	Signed, 2's complement binary integer	GL_BYTE
GLubyte	8	Unsigned binary integer	GL_UNSIGNED_BYTE
GLshort	16	Signed, 2's complement binary integer	GL_SHORT
GLushort	16	Unsigned binary integer	GL_UNSIGNED_SHORT
GLint	32	Signed, 2's complement binary integer	GL_INT
GLuint	32	Unsigned binary integer	GL_UNSIGNED_INT
GLfixed	32	Signed, 2's complement 16.16 integer	GL_FIXED
GLint64	64	Signed, 2's complement binary integer	
GLuint64	64	Unsigned binary integer	
GLsizei	32	A non-negative binary integer, for sizes.	
GLenum	32	An OpenGL enumerator value	
GLintptr	*ptrbits*[1]	Signed, 2's complement binary integer	
GLsizeiptr	*ptrbits*[1]	Non-negative binary integer size, for memory offsets and ranges	
GLsync	*ptrbits*[1]	Sync Object handle	
GLbitfield	32	A bitfield value	
GLhalf	16	An IEEE-754 floating-point value	GL_HALF_FLOAT
GLfloat	32	An IEEE-754 floating-point value	GL_FLOAT
GLclampf	32	An IEEE-754 floating-point value, clamped to the range [0,1]	
GLdouble	64	An IEEE-754 floating-point value	GL_DOUBLE
GLclampd	64	An IEEE-754 floating-point value, clamped to the range [0,1]	

Figure 4.1: *The data types of OpenGL.*

- To draw a 3D object, for example a cube, you will need to at least send the following items:
  - The vertices for the cube model
  - The transformation matrices to control the appearance of the cube's orientation in 3D space.

There are two ways of sending data through the OpenGL pipeline:

- Through a buffer to a vertex attribute
- Directly to a uniform variable
  Rendering a scene to make it appears 3D requires building appropriate transformation matrices and applying them to each of the models' vertices. It is most efficient to apply the required matrix operations in the vertex shader, and it is customary to send these matrices from the C++/OpenGL application to the shader in a uniform variable.

Some OpenGL commands:

- **glDrawArrays(GLenum mode, GLint first, GLsizei count);**
  To draw primitive, mode is the type of primitive, first indicates which vertex to start with, usually vertex 0, count specifies the total number of vertices to be drawn. For example:
  glDrawArrays(GL_POINTS,0,1) (Draw a point, start from vertex 0, one vertices is drawn)
  glDrawArrays(GL_TRIANGLE,0,3 (Draw a triangle, start from vertex 0, three vertices are drawn).

  When **glDrawArrays()** is executed, the data in the buffer starts flowing, sequentially from the beginning of the buffer, through the vertex shader. The vertex shader executes once per vertex.
- **glGenVertexArrays()**
  Create VAOs and return integer ID.
- **glGenBuffers()**
  Create VBOs and return integer ID.
- **glBindBuffer(GL_ARRAY_BUFFER, vbo[0])** is about activing the0th buffer
  glBufferData(GL_ARRAY_BUFFER, sizeof(vPositions), vPositions, GL_STATIC_DRAW) is about copying the array containing the vertices into the active buffer.
  Together they will copy the values of the vertices that are stored in a float array named **vPositions** into the 0th VBO.
- **glVertexAttributePointer(0,3,GL_FLOAT, GL_FALSE, 0, 0)** is to associate 0th attribute with buffer
  glEnableVertexAttribArray(0) is to enable 0th vertex attribute.
- **glBindVertexArrays()**
  To make the specified VAO active so that the generated buffers will be associated with that VAO.
- **glClear(GL_DEPTH_BUFFER_BIT)**
  glClearColor(0.0, 0.0, 0.0, 1.0)
  glClear(GL_COLOR_BUFFER_BIT)
  Clear the background to black.

II. GLAD

In simple words, GLAD manages function pointers for OpenGL. It is useful because OpenGL is only really a standard/specification it is up to the driver manufacturer to implement the specification to a driver that the specific graphics card supports. Since there are many different versions of OpenGL drivers, the location of most of its functions is not known at compile-time and needs to be queried at run-time. GLFW helps us at compile time only.

The "GLAD file" is generally referred to as the "OpenGL Loading Library" which is generally the library which loads the various types of the pointers, to the various types of the "OpenGL functions" at the time of the runtime respectively, which includes the various types of the "Core" as well as the “Extensions."

The include file for GLAD includes the required OpenGL headers behind the scenes (like GL/gl. h) so be sure to include GLAD before other header files that require OpenGL (like GLFW).

TO download GLAD, you can go to this website and specify your OpenGL version:
https://glad.dav1d.de/

you will get header files (.h) and C file as well (.c). Put it in your include file (e.g. /usr/include) and the **glad.c** can be stored somewhere in your main OpenGL project directory.

Figure 4.2: *The src and include files that is generated from* ***https://glad.dav1d.de/****.*

Advantages of using GLAD:

- Being able to select only the extensions you use, leading to (slightly) faster compile times and initialization at runtime.
- No additional dependency for your project.

III. GLEW

OpenGL Extension Wrangler (GLEW) can be used to try newer OpenGL functions.

You can find GLEW version-2.2.0 in my github repository for this book and DF Simulator.

This command is used to build and install GLEW:

```
sed -i 's%lib64%lib%g' config/Makefile.linux &&
sed -i -e '/glew.lib.static:/d' \
-e '/0644 .*STATIC/d' \
-e 's/glew.lib.static//' Makefile &&
```

```
make
```

then as super user or root type:
make install.all

Advantages of using GLEW:

- Adding GLEW as a dependency to, e.g., your CMakeLists.txt is enough to make it work.
- No large additional header and source files in your repository.
- GLEW can detect which extensions are available at runtime.
- It allows your program to adapt to the available extensions. For example, it can select a fallback path if a certain extension is not available on older hardware.

IV. GLFW

Graphics Library Framework (GLFW) is a free, Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan application development. It provides a simple, platform-independent API for creating windows, contexts and surfaces, reading input, handling events, etc.

GLFW is a C library specifically designed for use with OpenGL. It includes a class called **GLFWwindow** on which we can draw 3D scenes. Unlike SDL and SFML, GLFW only comes with the absolute necessities: window and context creation and input management. It offers the most control over the OpenGL context creation out of these three libraries.

You can dowbload GLFW here:
https://www.glfw.org/download

Some GLFW commands:

- **glfwInit()**
  To initialize GLFW
- **glfwSwapInterval()**, **glfwSwapBuffers()**
  Enable Vertical synchronization (VSync), GLFW windows are by default double-buffered.
- **glfwMakeContextCurrent()**
  Make the associated OpenGL context current.
- **glfwPollEvents()**
  Handles other window-related events, such as a key-press event.
- **glfwGetTime()**
  Returns the elapsed time since GLFW was initialized.

SDL, SFML, and GLFW are for creating the context and handling input

V. GLM

GLM (OpenGL Mathematics) is a header-only C++ mathematics library that will help us to do the computation necessary for our project. GLM provides classes and basic math functions related to graphics concepts, such as vector, matrix, and quaternion.

GLM able to create points, perform vector operations (addition, subtraction), carry out matrix transforms, generate random numbers, and generate noise. GLM includes a class called **mat4** for

instantiating and storing 4×4 matrices.

The manual can be read here:
https://github.com/g-truc/glm/blob/master/manual.md

How to use GLM:

- First at the top of the source code we need to include GLM by adding:
  #include <glm/glm.hpp>

  To do translation and rotation add:
  #include <glm/ext.hpp>
- To define a 2D point:
  glm::vec2 p1 = glm::vec2(2.0f, 8.0f);
- To define a 3D point:
  glm::vec3 p2 = glm::vec2(1.0f, 8.0f, 2.0f);
- To create identity square matrix of size 4:
  glm::mat4 matrix = glm::mat4(1.0f);
- The constructor call to build the identity matrix in the variable *m*
  glm::mat4 m(1.0f)
- To normalize vector:
  normalize(vec3) or **normalize(vec4)**
- To obtain dot product:
  dot(vec3,vec3) or **dot(vec4,vec4)**
- To obtain cross product:
  cross(vec3,vec3)
- In GLSL and GLM the data type **vec3 / vec4** can be used to hold either points or vectors.
- Toobtain the magnitude of a vector:
  length()
- Reflection and refraction are available in GLSL and GLM.

VI. GLSL

OpenGL Shading Language. On the hardware side, OpenGL provides a multi-stage graphics pipeline that is partially programmable using GLSL. GLSL intended to run on GPU / Graphics Card, since it is related with graphics pipeline.

GLSL language includes a data type called **mat4** that can be used for storing 4×4 matrices.

VII. SDL

SDL is also a cross-platform multimedia library, but targeted at C. That makes it a bit rougher to use for C++ programmers, but it's an excellent alternative to SFML. It supports more exotic platforms and most importantly, offers more control over the creation of the OpenGL context than SFML.

VIII. SFML

SFML is a cross-platform C++ multimedia library that provides access to graphics, input, audio, networking and the system. The downside of using this library is that it tries hard to be an all-in-one solution. You have little to no control over the creation of the OpenGL context, as it was designed to be used with its own set of drawing functions. SFML is written in C++, and has bindings for various languages such as C, .Net, Ruby, Python.

You can get the latest official release on SFML's website:
https://www.sfml-dev.org/download.php

In my github repository, you can find SFML version-2.5.1, along with FLAC and OpenAL (the required packages to be able to isntall SFML successfully). To build and install SFML, you need to build and install FLAC and OpenAL first, all of them are using cmake thus it won't be so hard to build and install.

First start with FLAC, extract the file:
tar -xvf flac-1.4.2.tar.xz
go to the extracted directory then type:
mkdir build
cd build
ccmake ..
Press c on keyboard then press e, then choose ON for BUILD_SHARED_LIBS (if available) and set CMAKE_INSTALL_PREFIX=/usr, then press c then e again then press g. Then back at terminal type:
make

as root type:
make install

Repeat the process for installing OpenAL and SFML, they are pretty much the same.

Now if you check **/usr/lib**, you will see **libsfml-audio.so, libsfml-graphics.so, libsfml-network.so, libsfml-system. so, libsfml-window.so**. Then you can use SFML.

IX. SOIL

Simple OpenGL Image Loader (SOIL) is an image loading library for OpenGL, it can be used to load image that we can process or do transformation onto that image. This library is very useful, when I remember the Elementary linear Algebra book with application to warps and morphs photograph of a person and predict how that person will look like after 50 years, then to be able to do the prediction we might use SOIL as one of the many solutions.

In order to build and install it, almost the same as SFML, we can use CMake, extract the package then go to the directory and type:
mkdir build
cd build

ccmake ..

Press c on keyboard then press e, then set CMAKE_BUILD_TYPE as RELEASE for this option. No need to build Tests, so SOIL_BUILD_TESTS=OFF, and set CMAKE_INSTALL_PREFIX=/usr.

Then press c then e again then press g. Then back at terminal type:
make

as root type:
make install

You shall see inside **/usr/lib** there is **libSOIL.a**, it is a static library. Different than dynamic library that has extension of **.so**.

X. Gnuplot

Gnuplot that is being used in this book is Gnuplot version 5.4 patchlevel 3 (last modified 2021-12-24).

XI. CMake

CMake is a tool that can generate project / solution file from a collection of source code files using pre-defined CMake scripts.

Chapter 5

Box2D, Bullet3, and ReactPhysics3D

*"You know someone is the best in Science and Engineering, when one chose the path of solitary, more focus, less distraction, like Leonardo Da Vinci and Isaac Newton who never marry, the proof will be her/his achievements, books written and innovations created." - **DS Glanzsche***

There are 3 physics library / engine that I will use and try here. You can learn from all these and perhaps create your own Physics library, or another science branch library of your interest like Astronomy library or Biology library. All the testing here is done with GFreya OS 1.8 with OpenGL 3.3, starting on October 2023.

I. Box2D

Box2D is using imgui for the testbed GUI, it uses OpenGL for the graphics API, and for taking care of mouse and keyboard events it uses GLFW.

i. Install Box2D Library and Include Files / Headers

To download it, open the terminal then type:
git clone https://github.com/erincatto/box2d.git
Enter the directory then type:
./build.sh
Results are in the build sub-folder. To build with CMake:

```
$ mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/usr \
$ —DBUILD_SHARED_LIBS=ON .. &&
$ make
```

After that, type:
make install

With the dynamic library has been installed into **/usr/lib**, we are able to build all kinds of physics simulations by modifying the physics world and the physical objects we want to simulate

with Box2D.

Now for the headers, go to the downloaded repository of Box2D **../box2d/extern/**, where you will see 4 folders of **glad, glfw, imgui,** and **sajson**. Since you should have installed and get GLAD by yourself and GLFW probably have been installed. You only need to copy **imgui** and **sajson** into **/usr/include**.

Now to test it, let's try the famous Hello World for Box2D, it shows the falling object from the height of 4 with gravity of 10 (not 9.8).

Create this simple HelloBox2D example:

```
#include "box2d/box2d.h"
#include <stdio.h>

nt main(int argc, char** argv)
{
	B2_NOT_USED(argc);
	B2_NOT_USED(argv);

	// Define the gravity vector.
	b2Vec2 gravity(0.0f, -10.0f);

	// Construct a world object, which will hold and simulate the rigid
	    bodies.
	b2World world(gravity);

	// Define the ground body.
	b2BodyDef groundBodyDef;
	groundBodyDef.position.Set(0.0f, -10.0f);

	// Call the body factory which allocates memory for the ground body
	// from a pool and creates the ground box shape (also from a pool).
	// The body is also added to the world.
	b2Body* groundBody = world.CreateBody(&groundBodyDef);

	// Define the ground box shape.
	b2PolygonShape groundBox;

	// The extents are the half-widths of the box.
	groundBox.SetAsBox(50.0f, 10.0f);

	// Add the ground fixture to the ground body.
	groundBody->CreateFixture(&groundBox, 0.0f);

	// Define the dynamic body. We set its position and call the body
	    factory.
	b2BodyDef bodyDef;
	bodyDef.type = b2_dynamicBody;
```

```
            bodyDef.position.Set(0.0f, 4.0f);
            b2Body* body = world.CreateBody(&bodyDef);

            // Define another box shape for our dynamic body.
            b2PolygonShape dynamicBox;
            dynamicBox.SetAsBox(1.0f, 1.0f);

            // Define the dynamic body fixture.
            b2FixtureDef fixtureDef;
            fixtureDef.shape = &dynamicBox;

            // Set the box density to be non-zero, so it will be dynamic.
            fixtureDef.density = 1.0f;

            // Override the default friction.
            fixtureDef.friction = 0.3f;

            // Add the shape to the body.
            body->CreateFixture(&fixtureDef);

            // Prepare for simulation. Typically we use a time step of 1/60 of a
            // second (60Hz) and 10 iterations. This provides a high quality
                simulation
            // in most game scenarios.
            float timeStep = 1.0f / 60.0f;
            int32 velocityIterations = 6;
            int32 positionIterations = 2;

            // This is our little game loop.
            for (int32 i = 0; i < 60; ++i)
            {
                   // Instruct the world to perform a single step of simulation.
                   // It is generally best to keep the time step and iterations
                       fixed.
                   world.Step(timeStep, velocityIterations, positionIterations);

                   // Now print the position and angle of the body.
                   b2Vec2 position = body->GetPosition();
                   float angle = body->GetAngle();

                   printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);
            }

            // When the world destructor is called, all bodies and joints are
                freed. This can
            // create orphaned pointers, so be careful about your world
                management.
```

```
        return 0;
    }
```

C++ Code 23: *main.cpp "Hello Box2D"*

To compile it type:
g++ main.cpp -o result -lbox2d
./result

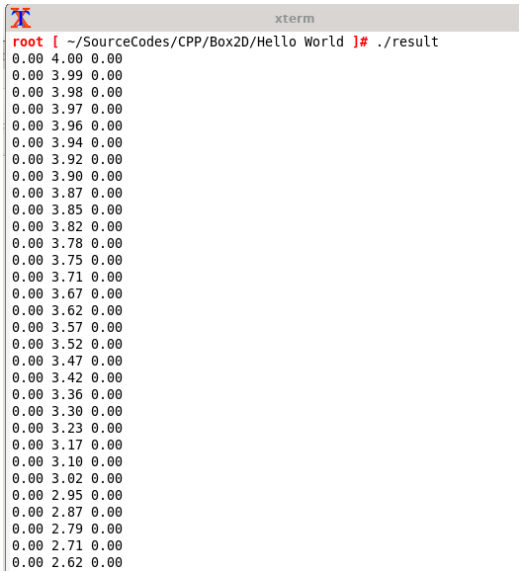

Figure 5.1: *The running HelloBox2D shows decreasing height from 4 to 1. The computation is the core of Box2D, to shows the visualization like at the testbed, by default Box2D uses imgui, but you can also use your own GUI besides imgui (file for this example in folder:* ***ch5-box2d-helloworld****).*

ii. Build Box2D Testbed

For simplicity instead of putting all the codes for Box2D testbed, I advise you to copy from my repository **https://github.com/glanzkaiser/DFSimulatorC/Source Codes/C++/ch5-box2d-testbed**, then go inside the directory and open the terminal then type:
mkdir build
cd build
cmake ..
make
./testbed You can configure the **CMakeLists.txt** so you can only generate the example you are concern about, less time compiling. The C++ source codes for every examples are located in **../tests/**, how to modify them, you need to comprehend C++ around medium-level at least, and you can read the manual here:
https://box2d.org/documentation/

iii. Know-How in Box2D

- There are 3 body types in Box2D: Static, Kinematic, and Dynamic

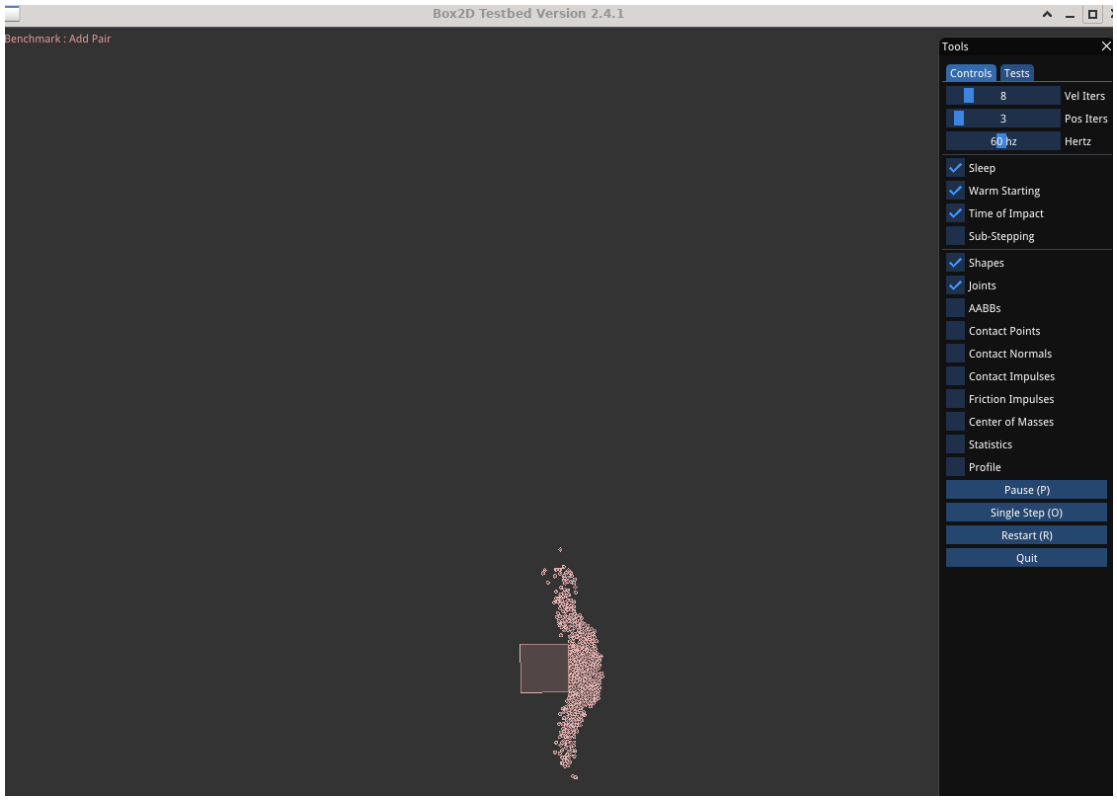

Figure 5.2: *The testbed of Box2D, one of the beauty of imgui is that it can offers parameters changing and a lot of modification on the scene (files for this example in folder:* ***ch5-box2d-testbed****).*

- Box2D support motion and collisions with
  1. Body class provides the motion.
  2. Fixture and Shape classes are for collisions.
- Properties in class Body: Position, Linear Velocity, Angular Velocity, Body Type. In the header file (**../box2d/include/box2d/b2_body.h**) you can see all the body definitions under **struct B2_API b2BodyDef...**, you can edit this and recompile to recreate the library to adjust to your modification.

```
struct B2_API b2BodyDef
{
	/// This constructor sets the body definition default
	    values.
	b2BodyDef()
	{
		position.Set(0.0f, 0.0f);
		angle = 0.0f;
		linearVelocity.Set(0.0f, 0.0f);
		angularVelocity = 0.0f;
		linearDamping = 0.0f;
		angularDamping = 0.0f;
		allowSleep = true;
		awake = true;
		fixedRotation = false;
		bullet = false;
		type = b2_staticBody;
		enabled = true;
		gravityScale = 1.0f;
	}
	...
}
```

- In Box2D we have impulse as force times 1 second.
- Motion depends on Force, current velocity, and mass

$$\begin{aligned}\triangle s &= v\triangle t \\ &= v_0\triangle t + \frac{1}{2}a\left(\triangle t\right)^2 \\ &= v_0\triangle t + \frac{1}{2}\left(\frac{F}{m}\right)\left(\triangle t\right)^2\end{aligned}$$

  The mass comes from the Fixture class. Fixture gives volume to the body.
- Four ways to move a Dynamic body:
  1. Forces: **applyForce** (linear), **applyTorque** (angular).
     For joints, complex shapes. Hard to control.
  2. Impulses: **applyLinearImpulse** (linear), **applyAngularImpulse** (angular).
     Great for joints, complex shapes. Extremely hard to control.
  3. Velocity: **setLinearVelocity** (linear), **setAngularVelocity** (angular)
     Very easy to control, not for joints, complex shapes.

4. Translation: **setTransform**

- Shape stores the object geometry (either boxes, circles or polygons, it must be convex). Shape also stores object density, which mass is area times density. The higher the density, the higher the mass.

```
bodydef = newBodyDef();
bodydef.type = type;
bodydef.position.set(position);
bodydef.angle = angle;

body1 = world.createBody(bodydef);

bodydef.position.set(position2);
body2 = world,createBody(bodydef);

shape1 = new PolygonShape();
shape2 = new PolygonShape();
shape1.set(verts1);
shape2.set(verts2);

fixdef = new FixtureDef();
fixdef.density = density;

fixdef.shape = shape1;
fixture1 = body1.createFixture(fixdef);
fixdef.shape = shape2;
fixture2 = body1.createFixture(fixdef);
```

- Size in Box2D:
  1. 1 Box2D unit = 1 meter
  2. 1 density = 1 kg/m^2
- Fixture has only one body, while bodies have many fixtures.
- Properties in Box2D:
  1. Friction: stickiness
  2. Restitution: bounciness

  Both value should be within 0 to 1.
- For custom collisions you can use **ContactListeners**, with two primary methods in interface:vvvvvv
  1. **beginContact**: When objects first collide
  2. **endContact**: When objects no longer collide

  It can be used for color changing in Box2D.
- Collision filtering can be used to define what can collide with certain fixture.
- Joints connect bodies, and they are affected by fixtures. Joints must use force or impulse, manual velocity might violate constraints.
- Box2D tends to allocate a large number of small objects (around 50-300 bytes). Using the system heap through malloc or new for small objects is inefficient and can cause

fragmentation. Box2D's solution is to use a small object allocator (SOA). Since Box2D uses a SOA, you should never new or malloc a body, fixture, or joint. However, you do have to allocate a **b2World** on your own.

II. Bullet

Bullet Physics is a professional open source collision detection, rigid body and soft body dynamics library. The library is free for commercial use under the ZLib license.

You can read more about Bullet here: **bulletphysics.org**

i. Install Bullet Library and Include Files / Headers

To get the library, open the terminal and type:
git clone https://github.com/bulletphysics/bullet3.git

Enter the directory, then type:
mkdir build
cd build
ccmake ..
make -j4 (build with 4 cores, faster than just **make**)

If you want to use manual typing, from the bullet directory, type:

```
mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/opt/hamzstlib/Physics/bulletinstall \
$ —DBUILD_SHARED_LIBS=ON .. &&
$ make
```

after finish compiling then type:
make install
It will install Bullet' libraries and include files at the install prefix place.

To get more information on the installation directory, where Bullet installs the libraries and search for the include files go to **../lib/cmake/bullet/** and read the **BulletConfig.cmake**, make sure it is not clashing with our own **LIBRARY_PATH**. You can see below on create an example for Bullet, how I set the path the same as the **BulletConfig.cmake** to find the libraries and include files for Bullet.

ii. Know-How in Bullet

- Bullet can do discrete and continuous collision detection including ray and convex sweep test. Collision shapes include concave and convex meshes and all basic primitives.
- The main task of a physics engine is to perform collision detection, resolve collisions and other constraints, and provide the updated world transform for all the objects.
- The most important data structures and computation stages for rigid body physics: The entire physics pipeline computation and its data structures are represented in Bullet by a dynamics

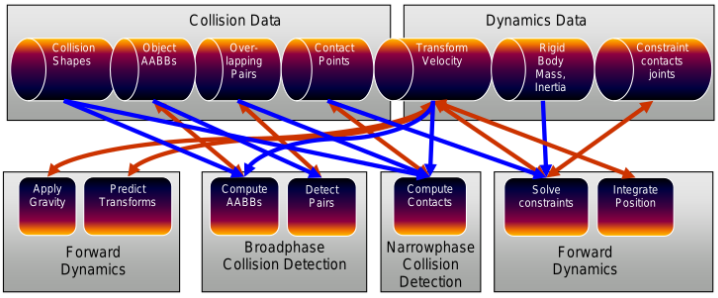

Figure 5.3: *The pipeline for Bullet physics is executed from left to right, starting by applying gravity, and ending by position integration, updating the world transform.*

world. When performing **stepSimulation** on the dynamics world, all the above stages are executed. The default dynamics world implementation is the **btDiscreteDynamicsWorld**.

- How to create a **btDiscreteDynamicsWorld, btCollisionShape, btMotionState**, and **btRigidBody**. Each frame call the **stepSimulation** on the dynamics world, and synchronize the world transform for your graphics object, the requirements:
  1. Put **#include "btBulletDynamicsCommon.h"** in your source file.
  2. Required include path: **Bullet/src** folder.
  3. Required libraries: **BulletDynamics, BulletCollision, LinearMath**.
- The derivations from **btDynamicsWorld** are **btDiscreteDynamicsWorld** and **btSoftRigidDynamicsWorld** will provide a high level interface that manages your physics objects and constraints. It also implements the update of all objects each frame.
- To construct a **btRigidBody** or **btCollisionObject** you need to provide:
  1. Mass, positive for dynamics moving objects and 0 for static objects
  2. Collision shape (Box, Sphere, Cone, Convex Hull, Triangle Mesh)
  3. Material properties (friction and restitution)

  Then update each frame with **stepSimulation**, call it on the dynamics world. The **btDiscreteDynamicsWorld** automatically takes into account variable timestep by performing interpolation instead of simulation for small timesteps. It uses an internal fixed timestep of 60 Hertz. The **stepSimulation** will perform collision detection and physics simulation. It updates the world transform for active objects by calling **btMotionState**'s **setWorldTransform**.

iii. Create an Example with Bullet

First, in order to link the library correctly and find the include files for Bullet, you will need to modify the export file by typing at terminal in :
cd
vim export

```
export prefix="/usr"
export hamzstlib="/opt/hamzstlib"
export physics="$hamzstlib/Physics"
```

```
# For library (.so, .a)
export LIBRARY_PATH="/usr/lib:$hamzstlib/lib:$physics/bulletinstall/lib"

export PKG_CONFIG_PATH="$physics/bulletinstall/lib/pkgconfig:$hamzstlib/lib
    /pkgconfig:"
```

The **bulletinstall** is the folder containing the **include, lib, output** directories after compiling Bullet.

Thus, it will be able to find the Bullet' libraries that have been built / compiled in GFreya OS. We are going to use the libraries **-lBulletDynamics -lBulletCollision -lLinearMath**.

Now, in a working directory (assumed still empty) create a file as usual, type:
vim main.cpp

C++ Code 24: *main.cpp "Bullet Example"*

III. ReactPhysics3D

ReactPhysics3D is an open source C++ physics engine library that can be used in 3D simulations and games. The library is released under the Zlib license. ReactPhysics3D is using OpenGL.

i. Install ReactPhysics3D Library and Include Files / Headers

To get the library, open the terminal and type:
git clone https://github.com/DanielChappuis/reactphysics3d.git

Now to build it, enter the downloaded repository, then type:
mkdir build
cd build
ccmake ..

Choose to build the library, then install it at the **/usr/lib**, so it can be used later on.

ii. Know-How in ReactPhysics3D

- The first thing you need to do when you want to use ReactPhysics3D is to instantiate the **PhysicsCommon** class. This class is used as a factory to instantiate one or multiple physics worlds and other objects. It is also responsible for logging and memory management. To create a physics world:
  PhysicsWorld* world = physicsCommon.createPhysicsWorld();
  This method will return a pointer to the physics world that has been created. How to create the world settings:

```
PhysicsWorld::WorldSettings settings;
settings.defaultVelocitySolverNbIterations = 20;
settings.isSleepEnabled = false;
```

```
settings.gravity = Vector(0,−9.81,0)

PhysicsWorld* world = physicsCommon.createPhysicsWorld();
```

this will create the physics world with your own settings.

- You probably only need one physics world with multiple objects.
- Simulating many bodies is cost expensive, thus sleeping technique is used to deactivate resting bodies. A body is put to sleep when its' linear and angular velocity stay under a given velocity threshold for certain amount of time.
- The base memory allocations in ReactPhysics3D are done using the **std::malloc()** and **std::free()** methods.
- Some methods that you can use:
  1. **testOverlap()**
     This group of methods can be used to test whether the colliders of two bodies overlap or not
  2. **testCollision()**
     This group of methods will give you the collision information (contact points, normals, etc) for colliding bodies.
  3. **testPointInside()**
     This method will tell you if a 3D point is inside a given CollisionBody, RigidBody, or Collider.
- Types of a Rigid Body:
  1. **Static body**
     A static body has infinite mass, zero velocity but its position can be changed manually. A static body does not collide with other static or kinematic bodies. You can use this as a floor or ground.
  2. **Kinematic body**
     A kinematic body has infinite mass, its' velocity can be changed manually and its' position computed by the physics engine. A kinematic body does not collide with other static or kinematic bodies.
  3. **Dynamic body**
     A dynamic body has non-zero mass, with velocity determined by forces and its' position is determined by the physics engine. A dynamic body can collide with other dynamic, static or kinematic bodies. You can use this as a rock or apple or any kind of object in the physics world.

  You can create a rigid body then change the type of RigidBody with:

```
Vector3 position(0.0, 3.0, 0.0);
Quaternion orientation = Quaternion::identity();
Transform transform(position, orientation);

RigidBody* body = world −> createRigidBody(transform);
body−>setType(BodyType::KINEMATIC);
```

- Damping is the effect of reducing the velocity of the rigid body during the simulation effects like air friction for instance. By default, no damping is applied. Without angular damping a pendulum will never comes to rest. You can choose to damp the linear or/and the angular

velocity of a rigid body. You need to use:
RigidBody::setLinearDamping()
RigidBody::setAngularDamping()
the damping value has to be positive. The value of 0 means no damping at all.

Chapter 6

DianFreya Math Physics Simulator I: Motion in Two Dimensions

*"There's no need to rush things, there's much to explain." - **Maria Traydor (Star Ocean 3)***

I. Position, Displacement, Velocity, and Acceleration

The general way of locating a particle is starting with a position vector, that determined where the particle is located at certain time. If we denote $\hat{r}$ as the position vector, it can be written in the unit-vector notation:

$$\hat{r} = x\hat{i} + y\hat{j} + z\hat{k}$$

where $x\hat{i},\ y\hat{j},\ z\hat{k}$ are the vector components of $\hat{r}$, and the coefficients x, y, and z are its scalar components.

If a particle undergoes a displacement $\triangle \overrightarrow{r}$ in time interval $\triangle t$, its average velocity $\overrightarrow{v}_{avg}$ for that time interval is

$$\overrightarrow{v}_{avg} = \frac{\triangle \overrightarrow{r}}{\triangle t}$$

As $\triangle t$ goes to 0, $\overrightarrow{v}_{avg}$ reaches a limit called either the velocity or the instantaneous velocity $\overrightarrow{v}$:

$$\overrightarrow{v} = \frac{d\overrightarrow{r}}{dt}$$

which can be written in unit-vector notation as

$$\overrightarrow{v} = v_x\hat{i} + v_y\hat{j} + v_z\hat{k}$$

where the scalar components of $\overrightarrow{v}$ are

$$v_x = \frac{dx}{dt}, \quad v_y = \frac{dy}{dt}, \quad v_z = \frac{dz}{dt}$$

If a particle's velocity changes from $\overrightarrow{v_1}$ to $\overrightarrow{v_2}$ in time interval $\triangle t$, its average acceleration during $\triangle t$ is

$$\overrightarrow{a}_{avg} = \frac{\overrightarrow{v_2} - \overrightarrow{v_1}}{\triangle t}$$

As $\triangle t$ goes to 0, $\overrightarrow{a}_{avg}$ reaches a limiting value called the instantaneous acceleration $\overrightarrow{a}$:

$$\overrightarrow{a} = \frac{d\overrightarrow{v}}{dt}$$

which can be written in unit-vector notation as

$$\overrightarrow{a} = a_x\hat{i} + a_y\hat{j} + a_z\hat{k}$$

where the scalar components of $\overrightarrow{a}$ are

$$a_x = \frac{dv_x}{dt}, \quad a_y = \frac{dv_y}{dt}, \quad v_z = \frac{dv_z}{dt}$$

II. Projectile Motion

We consider a special case in two-dimensional motion: when a particle moves in a vertical plane with some initial velocity $\overrightarrow{v_0}$ but its acceleration is always the freefall acceleration $\overrightarrow{g}$, which is downward. That particle is called projectile, mean that it is being projected or launched, and its motion is called projectile motion, like throwing a baseball into its ring. Ballistic, missile, a lot of sports like golf, tennis involve the study of projectile motion.

In projectile motion, a particle is launched into the air with a speed of $\overrightarrow{v_0}$ and at an angle θ_0 (as measured from a horizontal x axis). Its horizontal acceleration during flight is zero, while its vertical acceleration is $-g$. Thus, the equation of motion for the particle can be written as

$$\begin{aligned} x - x_0 &= (v_0 \ \cos \ \theta_0)t \\ y - y_0 &= (v_0 \ \sin \ \theta_0)t - \frac{1}{2}gt^2 \\ v_y &= v_0 \ \sin \ \theta_0 - gt \\ v_y^2 &= (v_0 \ \sin \ \theta_0)^2 - 2g(y - y_0) \end{aligned}$$

III. Simulation for Projectile Motion with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
mkdir build
cd build
cmake ..
make
./testbed

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

```
#include "test.h"
#include <iostream>

class ProjectileMotion: public Test
```

```
{
        public:

        ProjectileMotion()
        {
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                b2Timer timer;
                // Perimeter Ground body
                {
                        b2BodyDef bd;
                        b2Body* ground = m_world->CreateBody(&bd);

                        b2EdgeShape shapeGround;
                        shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                            (30.0f, 0.0f));
                        ground->CreateFixture(&shapeGround, 0.0f);

                        b2EdgeShape shapeTop;
                        shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2
                            (30.0f, 30.0f));
                        ground->CreateFixture(&shapeTop, 0.0f);

                        b2EdgeShape shapeLeft;
                        shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                            (-40.0f, 30.0f));
                        ground->CreateFixture(&shapeLeft, 0.0f);

                        b2EdgeShape shapeRight;
                        shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
                            (30.0f, 30.0f));
                        ground->CreateFixture(&shapeRight, 0.0f);
                }

                // Create the ball
                b2CircleShape ballShape;
                ballShape.m_p.SetZero();
                ballShape.m_radius = 0.5f;

                b2FixtureDef ballFixtureDef;
                ballFixtureDef.restitution = 0.15f; // the bounciness
                ballFixtureDef.density = 7.3f; // this will affect the ball
                    mass
                ballFixtureDef.friction = 0.1f;
                ballFixtureDef.shape = &ballShape;

                b2BodyDef ballBodyDef;
                ballBodyDef.type = b2_dynamicBody;
                ballBodyDef.position.Set(-25.0f, 0.5f);
```

```
                // ballBodyDef.angularDamping = 0.2f;

                m_ball = m_world->CreateBody(&ballBodyDef);
                b2Fixture *ballFixture = m_ball->CreateFixture(&
                    ballFixtureDef);

                m_createTime = timer.GetMilliseconds();
        }
        b2Body* bodylbox;
        b2Body* m_ball;

        void Keyboard(int key) override
        {
                switch (key)
                {
                        case GLFW_KEY_S:
                        int theta = 45;
                        float v0 = 20.0f;
                        m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), v0*
                            sin(theta)));
                        break;
                }
        }

        void Step(Settings& settings) override
        {
                b2MassData massData = m_ball->GetMassData();
                g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f",
                    massData.mass);
                m_textLine += m_textIncrement;

                b2Vec2 position = m_ball->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Ball Position, x = %.6
                    f", position.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Ball Position, y = %.6
                    f", position.y);
                m_textLine += m_textIncrement;

                b2Vec2 velocity = m_ball->GetLinearVelocity();
                g_debugDraw.DrawString(5, m_textLine, "Ball velocity, x = %.6
                    f", velocity.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Ball velocity, y = %.6
                    f", velocity.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms
```

```
                    ",
                m_createTime);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f \n", velocity.x, velocity.y);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new ProjectileMotion;
        }

        float m_createTime;
};

static int testIndex = RegisterTest("Motion in 2D", "Projectile Motion",
    ProjectileMotion::Create);
```

C++ Code 25: *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- Create a ball that is staying still on the ground.

```
            b2CircleShape ballShape;
            ballShape.m_p.SetZero();
            ballShape.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef;
            ballFixtureDef.restitution = 0.15f; // the bounciness
            ballFixtureDef.density = 7.3f; // this will affect the ball
                mass
            ballFixtureDef.friction = 0.1f;
            ballFixtureDef.shape = &ballShape;

            b2BodyDef ballBodyDef;
            ballBodyDef.type = b2_dynamicBody;
            ballBodyDef.position.Set(-25.0f, 0.5f);
            // ballBodyDef.angularDamping = 0.2f;

            m_ball = m_world->CreateBody(&ballBodyDef);
            b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
                );
```

  You can change the restitution, density, friction for different kind of ball for your own research.
- Create keyboard event, when pressed 'S' the ball is launched with initial velocity, $v_0 = 20$ and the initial angle of $\theta_0 = 45^0$.

```
void Keyboard(int key) override
{
	switch (key)
	{
		case GLFW_KEY_S:
		int theta = 45;
		float v0 = 20.0f;
		m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta),
		    v0*sin(theta)));
		break;
	}
}
```

- To show the data of the position, velocity and the mass of the ball for our simulation, then print the velocity data into xterm / terminal.

```
void Step(Settings& settings) override
{
	b2MassData massData = m_ball->GetMassData();
	g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f"
	    , massData.mass);
	m_textLine += m_textIncrement;

	b2Vec2 position = m_ball->GetPosition();
	g_debugDraw.DrawString(5, m_textLine, "Ball Position, x
	    = %.6f", position.x);
	m_textLine += m_textIncrement;
	g_debugDraw.DrawString(5, m_textLine, "Ball Position, y
	    = %.6f", position.y);
	m_textLine += m_textIncrement;

	b2Vec2 velocity = m_ball->GetLinearVelocity();
	g_debugDraw.DrawString(5, m_textLine, "Ball velocity, x
	    = %.6f", velocity.x);
	m_textLine += m_textIncrement;
	g_debugDraw.DrawString(5, m_textLine, "Ball velocity, y
	    = %.6f", velocity.y);
	m_textLine += m_textIncrement;

	g_debugDraw.DrawString(5, m_textLine, "create time =
	    %6.2f ms",
	m_createTime);
	m_textLine += m_textIncrement;

	printf("%4.2f %4.2f \n", velocity.x, velocity.y);
	//printf("%4.2f %4.2f \n", position.x, position.y);
	Test::Step(settings);
}
```

After recompiling this, you can save it into textfile by opening the testbed with this command:
./testbed > projectileoutput.txt
After you close the testbed to record the result, then see it at the current working directory where **testbed** is located and see **projectileoutput.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 2 columns.

To plot the velocity in x and y axis for the projectile motion of the ball with respect to time, now open terminal at the directory containing the **projectileoutput.txt** and type:
gnuplot
set xlabel "time"
set ylabel "v_{x}"
plot "projectileoutput.txt" using 1 title "" with lines
set ylabel "v_{y}"
plot "projectileoutput.txt" using 2 title "" with lines

Now you can go back to the source code of the projectile motion and uncomment the line to print the position and comment the line to print the velocity:

```
//printf("%4.2f %4.2f \n", velocity.x, velocity.y);
printf("%4.2f %4.2f \n", position.x, position.y);
```

Now, we can plot the position of the ball with gnuplot, recompile the testbed to make the change occurs then type:
./testbed > projectileoutput.txt

Plot it with gnuplot from the working directory:
gnuplot
set xlabel "time"
set ylabel "v_{x}"
plot "projectileoutput.txt" using 1:2 title "Ball trajectory" with lines

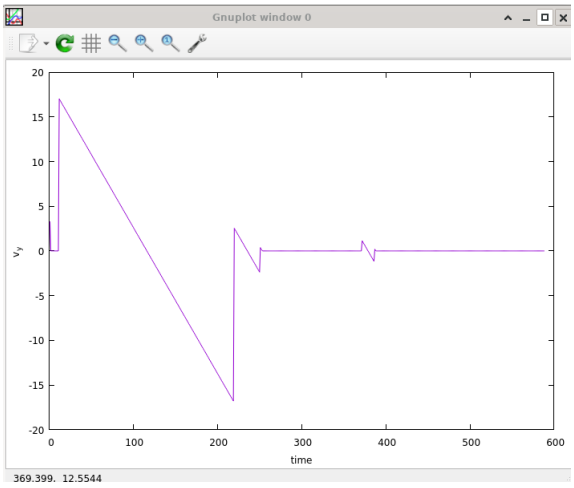

Figure 6.1: *The "projectileoutput.txt" is being plotted by using gnuplot for the data of the v_x (the horizontal velocity).*

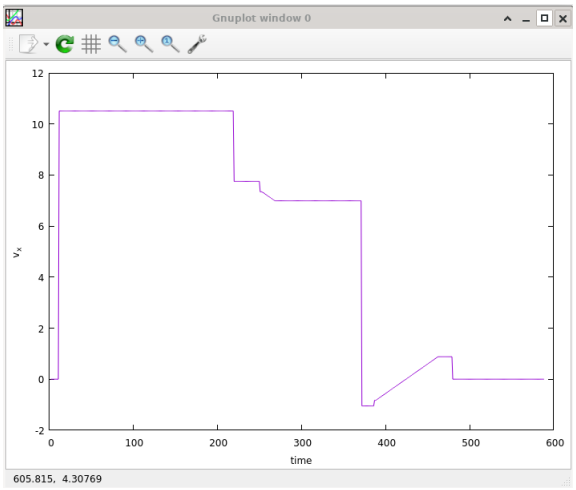

Figure 6.2: *The "projectileoutput.txt" is being plotted by using gnuplot for the data of the v_y (the horizontal velocity).*

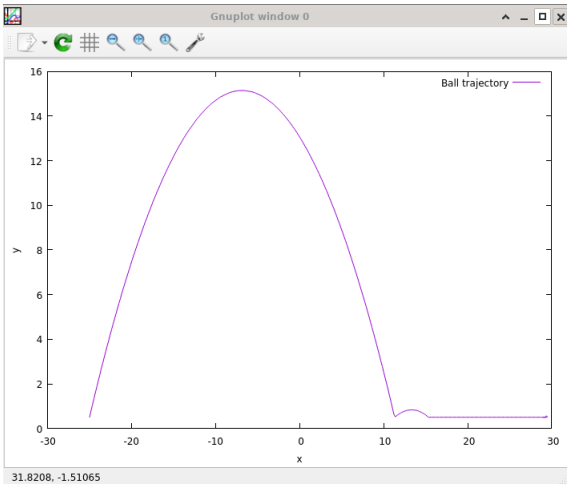

Figure 6.3: *The "projectileoutput.txt" is being plotted by using gnuplot for the data of the x and y (position of the ball). You can see that the ball is still sliding forward if you see the Box2D simulation, since we make the restitution for the ball 0.15, a bit bouncy. This simulation is very useful to design a better golf ball or galleon canon for spaceship to hit the target faster and accurate.*

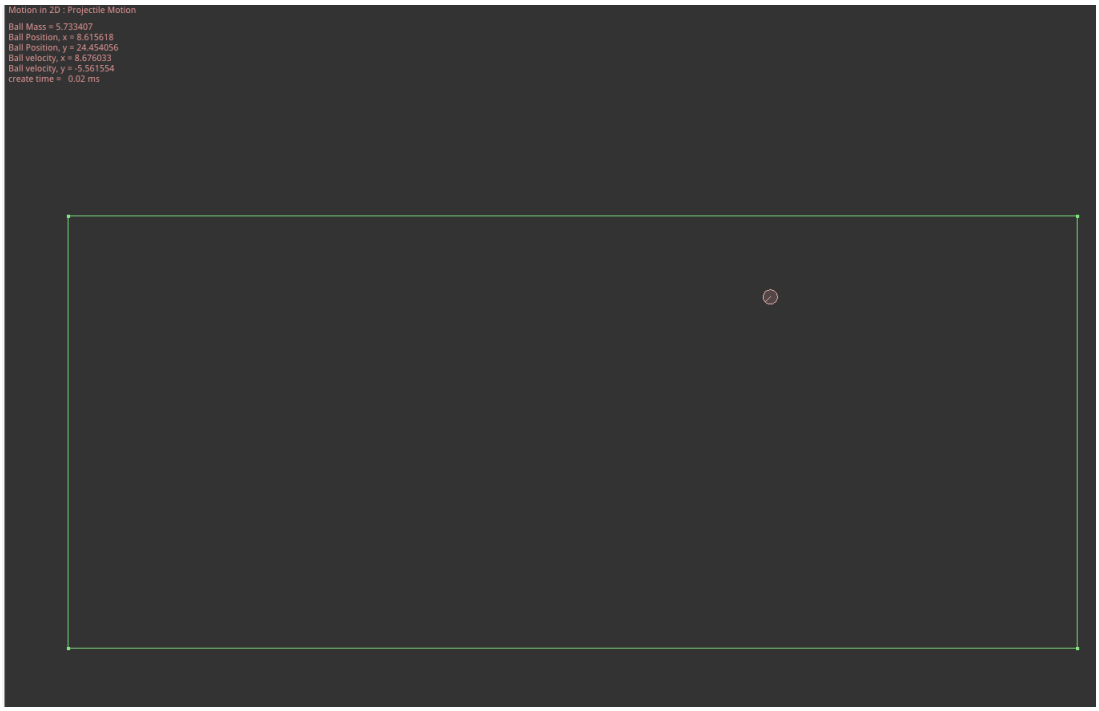

Figure 6.4: *The modified Box2D simulation of a ball being projected with initial angle* $\theta_0 = 45^0$ *and initial velocity of* $v_0 = 20$ *(the current simulation code can be located in:* ***/DianFreya-box2d-testbed/tests/projectile_motion.cpp****).*

IV. Simulation for Projectile Dropped from Above with Box2D

This is coming from a sample problem in [4], but I modify it a little bit. Suppose that Europe is now in war and get help of food supplies from its allies, if there are two rescue drones, the first drone is sending a cheese ball (shape of a circle in 2 dimension) and the second drone is sending boxes full of chocolates and medicine. Don't ask why they don't send vegetables and fruits that are healthier, it is at war now. Suppose both flying at different height, constant height of $h_1 = 40$ and $h_2 = 30$, but their speeds are the same, at certain speed of $v_0 = 20$. What should be the angle ϕ_1 and ϕ_2 of the first and second drone line of sight to the net below (the height of the net is 5, with width of 10, centered at $x = 0$) that will capture the cheese ball and boxes full of chocolate and medicine?

We need to know that ϕ_1 is given by

$$\phi_1 = \tan^{-1}\left(\frac{x_1}{h_1}\right)$$

and ϕ_2 is given by

$$\phi_2 = \tan^{-1}\left(\frac{x_2}{h_2}\right)$$

where x_1 and x_2 are the horizontal coordinates of the net to capture the cheese ball and the chocolate boxes with medicine respectively from where the supplies being drop by each of the drones. We will have two cases and should be able to find x_1 and x_2 for each of the case with

$$x_1 - x_0 = (v_0 \cos\theta_0)t_1$$
$$x_2 - x_0 = (v_0 \cos\theta_0)t_2$$

we know that $x_0 = 0$ because the origin is placed at the point of release from each of the drone. v_0 is the drone velocity, and both drones fly at the same velocity of $v_0 = 20$, with the angle $\theta_0 = 0^0$ measured relative to the positive direction of the x axis, since the supplies are being dropped not being projected / launched.

To find t, the time for the supply arrive at the net, we can use the vertical motion displacement equation for the first drone:

$$y_1 - y_0 = (v_0 \ \sin \ \theta_0)t_1 - \frac{1}{2}gt_1^2$$

and for the second drone vertical displacement equation:

$$y_2 - y_0 = (v_0 \ \sin \ \theta_0)t_2 - \frac{1}{2}gt_2^2$$

The vertical displacement for each of the drone will be negative, the negative value indicates that the supplies moves downward. Thus, solving for t for the first drone

$$\begin{aligned} y_1 - y_0 &= (v_0 \ \sin \ \theta_0)t_1 - \frac{1}{2}gt_1^2 \\ -40 &= (20 \ \sin \ 0^0)t_1 - \frac{1}{2}(9.8)t_1^2 \\ -40 &= -\frac{1}{2}(9.8)t_1^2 \\ t_1 &= \sqrt{\frac{80}{9.8}} \\ t_1 &= 2.8571 \end{aligned}$$

then for the second drone

$$\begin{aligned} y_2 - y_0 &= (v_0 \ \sin \ \theta_0)t_2 - \frac{1}{2}gt_2^2 \\ -30 &= (20 \ \sin \ 0^0)t_2 - \frac{1}{2}(9.8)t_2^2 \\ -30 &= -\frac{1}{2}(9.8)t_2^2 \\ t_2 &= \sqrt{\frac{60}{9.8}} \\ t_2 &= 2.4743 \end{aligned}$$

After we obtain t for both drones, we can estimate the horizontal distance between the drone to the net and the drone' line of sight

$$\begin{aligned} x_1 - x_0 &= (v_0 \cos\theta_0)t_1 \\ x_1 - 0 &= (20 \ \cos \ 0^0)(2.8571) \\ x_1 &= 57.141999 \end{aligned}$$

$$\phi_1 = \tan^{-1}\left(\frac{x_1}{h_1}\right) = \tan^{-1}\left(\frac{57.141999}{40}\right) = \tan^{-1}(1.428549) = 0.960063 \text{ rad} \approx 55.007575^0$$

The horizontal distance for the second drone

$$\begin{aligned} x_2 - x_0 &= (v_0 \cos\theta_0)t_2 \\ x_2 - 0 &= (20 \ \cos \ 0^0)(2.4743) \\ x_2 &= 49.48716 \end{aligned}$$

$$\phi_2 = \tan^{-1}\left(\frac{x_2}{h_2}\right) = \tan^{-1}\left(\frac{49.48716}{30}\right) = \tan^{-1}(1.649572) = 1.0258174 \text{ rad} \approx 58.775^0$$

Either the angles of depression ϕ_1 and ϕ_2 or the horizontal distance between the center of the net and the drone can be used as the data input to release or drop the supplies, as sensor for drone or mechatronics that can calculate horizontal distance toward a target and altitude is available along with one that can calculate angle of depression toward a designated target.

For the C++ simulation, you can copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
mkdir build
cd build
cmake ..
make
./testbed

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

```
#include "test.h"
#include <iostream>

// This is used to test sensor shapes.
```

```
class ProjectileDropped : public Test
{
	public:

	enum
	{
		e_count = 10
	};

	ProjectileDropped()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
			    0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(5.6f, 0.0f), b2Vec2(5.6f,
			    10.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		// Create the net centering at x=0
		{
			b2PolygonShape shape;
			shape.SetAsBox(0.5f, 0.125f);

			b2FixtureDef fd;
			fd.shape = &shape;
			fd.density = 20.0f;
			fd.friction = 0.2f;

			b2RevoluteJointDef jd;

			b2Body* prevBody = ground;
			for (int32 i = 0; i < e_count; ++i)
			{
				b2BodyDef bd;
				bd.type = b2_dynamicBody;
				bd.position.Set(-4.5f + 1.0f * i, 5.0f); // 5.0
```

```
                                f is the height
                        b2Body* body = m_world->CreateBody(&bd);
                        body->CreateFixture(&fd);

                        b2Vec2 anchor(-5.0f + 1.0f * i, 5.0f); //create
                             the chain ball anchor
                        jd.Initialize(prevBody, body, anchor);
                        m_world->CreateJoint(&jd);

                        if (i == (e_count >> 1))
                        {
                               m_middle = body;
                        }
                        prevBody = body;
                }

                b2Vec2 anchor(-5.0f + 1.0f * e_count, 5.0f); // the
                    right anchor
                jd.Initialize(prevBody, ground, anchor);
                m_world->CreateJoint(&jd);
        }
        // Create the ball
        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
        ballShape.m_radius = 0.5f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.15f; // the bounciness
        ballFixtureDef.density = 7.3f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(-52.14199f, 40.0f);
        // ballBodyDef.angularDamping = 0.2f;

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);
        int theta = 0;
        float v0 = 20.0f;
        m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));

        // Create Breakable dynamic body
        {
                b2BodyDef bd;
```

```
                        bd.type = b2_dynamicBody;
                        bd.position.Set(-44.48716f, 30.0f);
                        bd.angle = 0.85f * b2_pi;
                        m_body1 = m_world->CreateBody(&bd);

                        m_shape1.SetAsBox(0.5f, 0.5f, b2Vec2(-0.5f, 0.0f), 0.0
                            f);
                        m_piece1 = m_body1->CreateFixture(&m_shape1, 1.0f);

                        m_shape2.SetAsBox(0.5f, 0.5f, b2Vec2(0.5f, 0.0f), 0.0f
                            );
                        m_piece2 = m_body1->CreateFixture(&m_shape2, 1.0f);

                        m_body1 ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
                }

                m_break = false;
                m_broke = false;
        }
        b2Body* m_middle;
        b2Body* m_ball;

        b2Body* m_body1;
        b2Vec2 m_velocity;
        float m_angularVelocity;
        b2PolygonShape m_shape1;
        b2PolygonShape m_shape2;
        b2Fixture* m_piece1;
        b2Fixture* m_piece2;
        bool m_broke;
        bool m_break;

        void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
            override
        {
                if (m_broke)
                {
                        // The body already broke.
                        return;
                }

                // Should the body break?
                int32 count = contact->GetManifold()->pointCount;

                float maxImpulse = 0.0f;
                for (int32 i = 0; i < count; ++i)
                {
                        maxImpulse = b2Max(maxImpulse, impulse->normalImpulses
```

```
                        [i]);
                }

                if (maxImpulse > 40.0f)
                {
                        // Flag the body for breaking.
                        m_break = true;
                }
        }

        void Break()
        {
                // Create two bodies from one.
                b2Body* body1 = m_piece1->GetBody();
                b2Vec2 center = body1->GetWorldCenter();

                body1->DestroyFixture(m_piece2);
                m_piece2 = NULL;

                b2BodyDef bd;
                bd.type = b2_dynamicBody;
                bd.position = body1->GetPosition();
                bd.angle = body1->GetAngle();
                b2Body* body2 = m_world->CreateBody(&bd);
                m_piece2 = body2->CreateFixture(&m_shape2, 1.0f);

                // Compute consistent velocities for new bodies based on
                // cached velocity.
                b2Vec2 center1 = body1->GetWorldCenter();
                b2Vec2 center2 = body2->GetWorldCenter();

                b2Vec2 velocity1 = m_velocity + b2Cross(m_angularVelocity,
                    center1 - center);
                b2Vec2 velocity2 = m_velocity + b2Cross(m_angularVelocity,
                    center2 - center);

                body1->SetAngularVelocity(m_angularVelocity);
                body1->SetLinearVelocity(velocity1);

                body2->SetAngularVelocity(m_angularVelocity);
                body2->SetLinearVelocity(velocity2);
        }

        void Step(Settings& settings) override
        {
                b2MassData massData = m_ball->GetMassData();
                g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f",
                    massData.mass);
```

```
                m_textLine += m_textIncrement;

                b2Vec2 position = m_ball->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Ball Position, x = %.6
                    f", position.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Ball Position, y = %.6
                    f", position.y);
                m_textLine += m_textIncrement;

                b2Vec2 positionbox1 = m_body1->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, x =
                    %.6f", positionbox1.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, y =
                    %.6f", positionbox1.y);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f %4.2f %4.2f \n", position.x, position.y,
                    positionbox1.x, positionbox1.y);

                if (m_break)
                {
                       Break();
                       m_broke = true;
                       m_break = false;
                }

                // Cache velocities to improve movement on breakage.
                if (m_broke == false)
                {
                       m_velocity = m_body1->GetLinearVelocity();
                       m_angularVelocity = m_body1->GetAngularVelocity();
                }

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new ProjectileDropped;
        }

};

static int testIndex = RegisterTest("Motion in 2D", "Projectile Dropped",
    ProjectileDropped::Create);
```

C++ Code 26: *tests/projectile_dropped.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- Inside the **ProjectileDropped()**, to create the ground and the wall at the right side of the net

```
b2Body* ground = NULL;
{
	b2BodyDef bd;
	ground = m_world->CreateBody(&bd);

	b2EdgeShape shape;
	shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
	    0.0f));
	ground->CreateFixture(&shape, 0.0f);
}
{
	b2BodyDef bd;
	ground = m_world->CreateBody(&bd);

	b2EdgeShape shape;
	shape.SetTwoSided(b2Vec2(5.6f, 0.0f), b2Vec2(5.6f, 10.0f
	    ));
	ground->CreateFixture(&shape, 0.0f);
}
// Create the net centering at x=0
{
	b2PolygonShape shape;
	shape.SetAsBox(0.5f, 0.125f);

	b2FixtureDef fd;
	fd.shape = &shape;
	fd.density = 20.0f;
	fd.friction = 0.2f;

	b2RevoluteJointDef jd;

	b2Body* prevBody = ground;
	for (int32 i = 0; i < e_count; ++i)
	{
		b2BodyDef bd;
		bd.type = b2_dynamicBody;
		bd.position.Set(-4.5f + 1.0f * i, 5.0f); // 5.0f
		    is the height
		b2Body* body = m_world->CreateBody(&bd);
		body->CreateFixture(&fd);

		b2Vec2 anchor(-5.0f + 1.0f * i, 5.0f); //create
		    the chain ball anchor
		jd.Initialize(prevBody, body, anchor);
		m_world->CreateJoint(&jd);
```

```
                if (i == (e_count >> 1))
                {
                        m_middle = body;
                }
                prevBody = body;
        }

        b2Vec2 anchor(-5.0f + 1.0f * e_count, 5.0f); // the
            right anchor
        jd.Initialize(prevBody, ground, anchor);
        m_world->CreateJoint(&jd);
}
```

- To create the cheese ball that is being dropped by the first drone at height of 40, the reason why I set the drop point as $52.14199f$ instead of $57.14199f$ is because the net is centered at $x = 0$ and has width of 10, 10 divided by 2 is 5, so the drone shall move forward 5 unit more before dropping the cheese ball, thus it can land at the center of the net, not too early or late. If you watch the Box2D simulation for this, both the cheese ball and chocolates with medicine boxes arrive exactly at the center of the net.

```
// Create the ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.15f; // the bounciness
ballFixtureDef.density = 7.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-52.14199f, 40.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
    );
int theta = 0;
float v0 = 20.0f;
m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
```

- To create the 2 boxes of chocolates and medicine, the boxes are attached and prone to break

```
// Create Breakable dynamic body
{
        b2BodyDef bd;
```

```
                bd.type = b2_dynamicBody;
                bd.position.Set(-44.48716f, 30.0f);
                bd.angle = 0.85f * b2_pi;
                m_body1 = m_world->CreateBody(&bd);

                m_shape1.SetAsBox(0.5f, 0.5f, b2Vec2(-0.5f, 0.0f), 0.0f)
                    ;
                m_piece1 = m_body1->CreateFixture(&m_shape1, 1.0f);

                m_shape2.SetAsBox(0.5f, 0.5f, b2Vec2(0.5f, 0.0f), 0.0f);
                m_piece2 = m_body1->CreateFixture(&m_shape2, 1.0f);

                m_body1 ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
        }

        m_break = false;
        m_broke = false;
```

- To declare the body, polygon shape, fixtures, break boolean variable.

```
        b2Body* m_middle;
        b2Body* m_ball;

        b2Body* m_body1;
        b2Vec2 m_velocity;
        float m_angularVelocity;
        b2PolygonShape m_shape1;
        b2PolygonShape m_shape2;
        b2Fixture* m_piece1;
        b2Fixture* m_piece2;
        bool m_broke;
        bool m_break;
```

- The solver, function **PostSolve()** for the breaking apart of the boxes into two.

```
        void PostSolve(b2Contact* contact, const b2ContactImpulse*
            impulse) override
        {
                if (m_broke)
                {
                        // The body already broke.
                        return;
                }

                // Should the body break?
                int32 count = contact->GetManifold()->pointCount;

                float maxImpulse = 0.0f;
                for (int32 i = 0; i < count; ++i)
                {
```

```
                        maxImpulse = b2Max(maxImpulse, impulse->
                            normalImpulses[i]);
                }

                if (maxImpulse > 40.0f)
                {
                        // Flag the body for breaking.
                        m_break = true;
                }
        }
```

- We initially have two boxes attached into one, then after landing on the net, get impulses, it breaks apart into two, that will each has its own velocity, position, and become different bodies in Box2D Physics world for this simulation. With function **Break()** we define how to separate **m_body1** into two separate bodies

```
        void Break()
        {
                // Create two bodies from one.
                b2Body* body1 = m_piece1->GetBody();
                b2Vec2 center = body1->GetWorldCenter();

                body1->DestroyFixture(m_piece2);
                m_piece2 = NULL;

                b2BodyDef bd;
                bd.type = b2_dynamicBody;
                bd.position = body1->GetPosition();
                bd.angle = body1->GetAngle();
                b2Body* body2 = m_world->CreateBody(&bd);
                m_piece2 = body2->CreateFixture(&m_shape2, 1.0f);

                // Compute consistent velocities for new bodies based on
                // cached velocity.
                b2Vec2 center1 = body1->GetWorldCenter();
                b2Vec2 center2 = body2->GetWorldCenter();

                b2Vec2 velocity1 = m_velocity + b2Cross(
                    m_angularVelocity, center1 - center);
                b2Vec2 velocity2 = m_velocity + b2Cross(
                    m_angularVelocity, center2 - center);

                body1->SetAngularVelocity(m_angularVelocity);
                body1->SetLinearVelocity(velocity1);

                body2->SetAngularVelocity(m_angularVelocity);
                body2->SetLinearVelocity(velocity2);
        }
```

- This will show the cheese ball mass, the position, and the boxes position for the first box, along with the condition if the boxes break apart then the function **Break()** will occur

```
            void Step(Settings& settings) override
            {
                   b2MassData massData = m_ball->GetMassData();
                   g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f"
                       , massData.mass);
                   m_textLine += m_textIncrement;

                   b2Vec2 position = m_ball->GetPosition();
                   g_debugDraw.DrawString(5, m_textLine, "Ball Position, x
                       = %.6f", position.x);
                   m_textLine += m_textIncrement;
                   g_debugDraw.DrawString(5, m_textLine, "Ball Position, y
                       = %.6f", position.y);
                   m_textLine += m_textIncrement;

                   b2Vec2 positionbox1 = m_body1->GetPosition();
                   g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, x
                        = %.6f", positionbox1.x);
                   m_textLine += m_textIncrement;
                   g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, y
                        = %.6f", positionbox1.y);
                   m_textLine += m_textIncrement;

                   printf("%4.2f %4.2f %4.2f %4.2f \n", position.x,
                       position.y,positionbox1.x, positionbox1.y);

                   if (m_break)
                   {
                          Break();
                          m_broke = true;
                          m_break = false;
                   }

                   // Cache velocities to improve movement on breakage.
                   if (m_broke == false)
                   {
                          m_velocity = m_body1->GetLinearVelocity();
                          m_angularVelocity = m_body1->GetAngularVelocity()
                              ;
                   }

                   Test::Step(settings);
            }
```

Now, to plot the position of the cheese ball, chocolates and medicine supplies with gnuplot, recompile the testbed to make the change occurs then type:

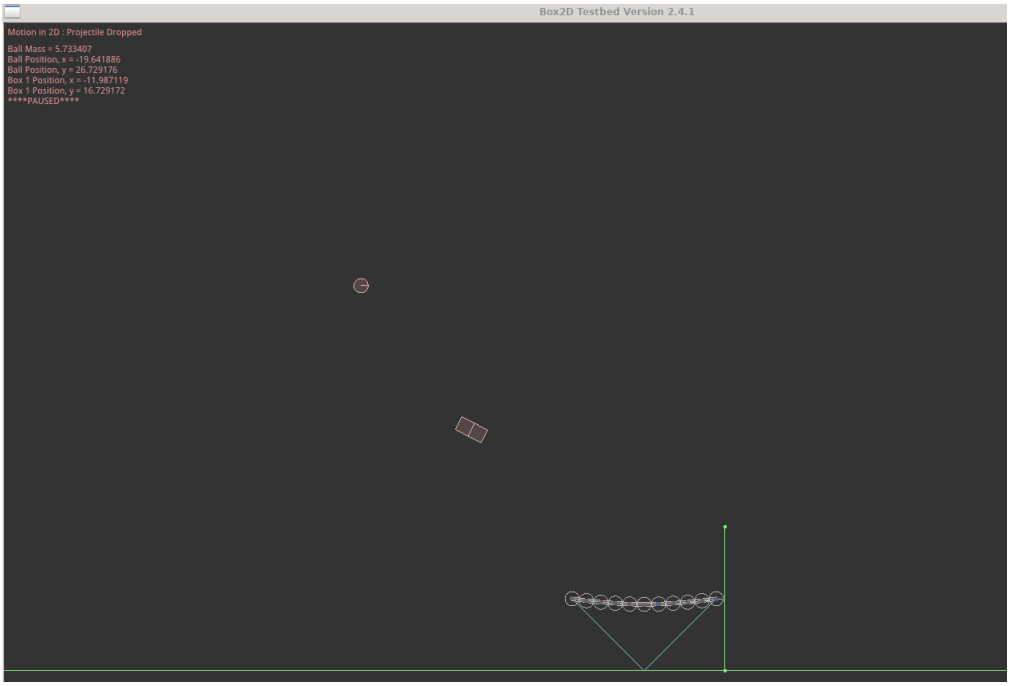

Figure 6.5: *The modified Box2D simulation of a cheese ball and chocolates with medicine boxes being dropped from two drones with same velocity of $v_0 = 20$ but flying at different heights (the current simulation code can be located in:* ***/DianFreya-box2d-testbed/tests/projectile_dropped.cpp****).*

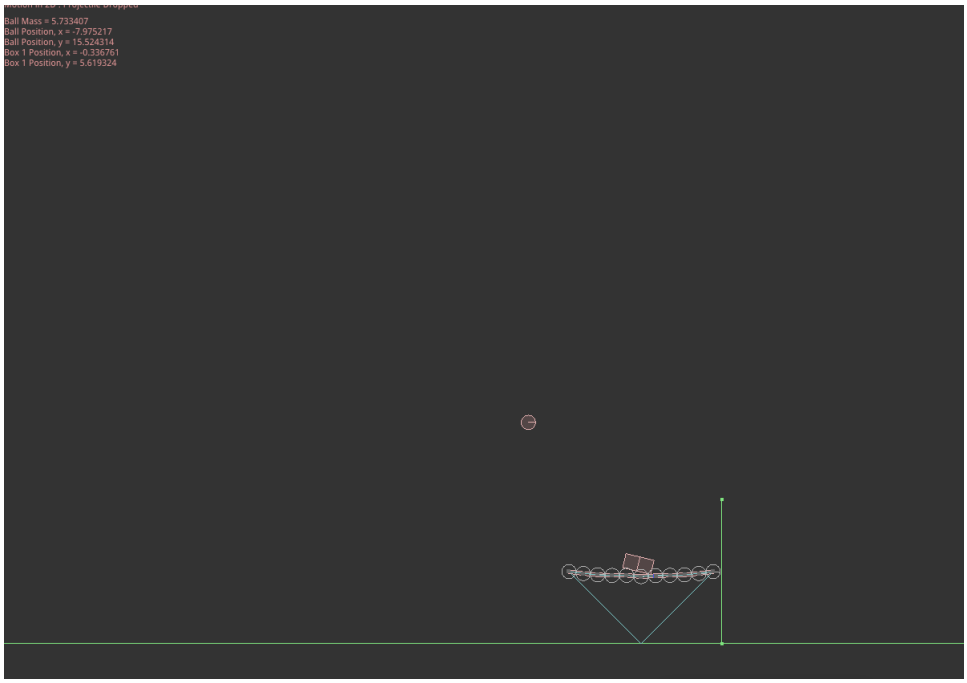

Figure 6.6: *The modified Box2D simulation clearly shows that both the supplies will arrive at the middle of the net.*

./testbed > projectiledropped.txt

Plot it with gnuplot from the working directory:
gnuplot
set xlabel "x"
set ylabel "y"
plot "projectiledropped.txt" using 1:2 title "Cheese Ball" with lines, "projectiledropped.txt" using 3:4 title "Chocolates and Medicine" with lines

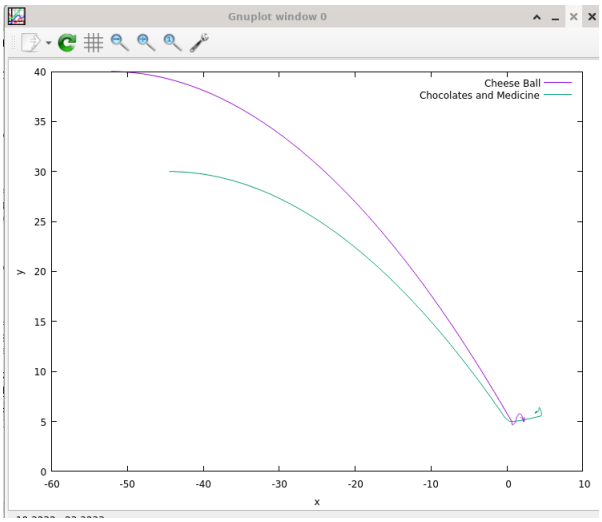

Figure 6.7: *The gnuplot of the position of the supplies that are dropped from different height by each drone*

V. Uniform Circular Motion

A particle is in uniform circular motion if it travels around a circle path or a circular arc at constant (uniform) speed. Although the speed does not vary, the particle is accelerating because the velocity changes in direction.

The velocity and acceleration vectors for uniform circular motion have constant magnitude but their directions change continuously. The velocity is always directed tangent to the circle in the direction of motion, while the acceleration is always directed radially inward, to the center of the circle. That is why it is called centripetal acceleration.

The scalar components of $\vec{v}$ can be written as

$$\vec{v} = v_x\hat{i} + v_y\hat{j} = (-v\ \sin\ \theta)\hat{i} + (v\ \cos\ \theta)\hat{j} \tag{6.1}$$

remember that since sine is an odd function and cosine is an even function, we will have $\sin(\theta) = -\sin(\theta)$ and $\cos(\theta) = \cos(\theta)$. Now, we can substitute

$$\sin\ \theta = \frac{y_p}{r}$$

$$\cos\ \theta = \frac{x_p}{r}$$

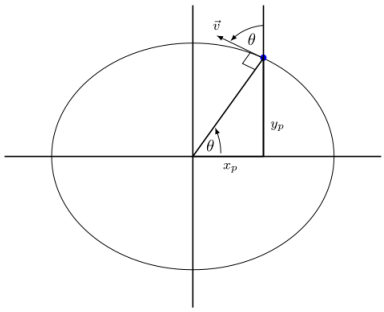

Figure 6.8: *The particle p moves in counter-clockwise uniform circular motion with its position and velocity $\vec{v}$ at a certain instant.*

thus

$$\begin{aligned}\vec{v} &= v_x\hat{i} + v_y\hat{j} \\ &= \left(-\frac{vy_p}{r}\right)\hat{i} + \left(\frac{vx_p}{r}\right)\hat{j}\end{aligned}$$

To find the acceleration of the particle, we must derivate the velocity with respect to time. Noting that speed v and radius r do not change with time, we obtain

$$\begin{aligned}\vec{a} &= \frac{d\vec{v}}{dt} \\ &= \left(-\frac{v}{r}\frac{dy_p}{dt}\right)\hat{i} + \left(\frac{v}{r}\frac{dx_p}{dt}\right)\hat{j} \\ \vec{a} &= \left(-\frac{v^2}{r}\cos\ \theta\right)\hat{i} + \left(-\frac{v^2}{r}\ \sin\ \theta\right)\hat{j}\end{aligned}$$

with $\frac{dy_p}{dt} = v_y = v\ \cos\theta$ and $\frac{dx_p}{dt} = v_x = -v\ \sin\ \theta$. Now, to calculate the acceleration magnitude, we will have

$$\begin{aligned}a &= \sqrt{a_x^2 + a_y^2} \\ &= \frac{v^2}{r}\sqrt{(\cos\ \theta)^2 + (\sin\ \theta)^2} \\ &= \frac{v^2}{r}(1) \\ a &= \frac{v^2}{r}\end{aligned}$$

Now to orient a, we want to prove that the angle ϕ will direct the acceleration toward the circle's center

$$\tan\ \phi = \frac{a_y}{a_x} = \frac{-(v^2/r)\ \sin\ \theta}{-(v^2/r)\ \cos\ \theta} = \tan\ \theta \tag{6.2}$$

Thus, $\phi = \theta$.

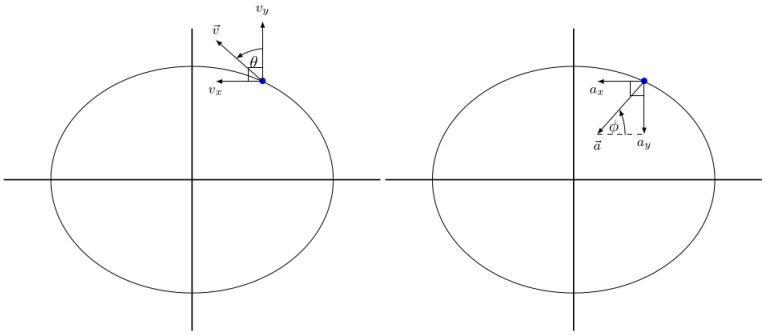

Figure 6.9: *The velocity $\vec{v}$ and acceleration $\vec{a}$ of a particle in uniform circular motion that is moving in counter-clockwise direction.*

VI. Simulation for Uniform Circular Motion with Box2D

For the C++ simulation, you can copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
mkdir build
cd build
cmake ..
make
./testbed

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Circular Motion**.

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class CircularMotion : public Test
{
	public:

	CircularMotion()
	{
		m_world->SetGravity(b2Vec2(0.0f,0.0f));
		b2BodyDef bd;
		b2Body* ground = m_world->CreateBody(&bd);
		b2Body* b1;
		{
			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
			    0.0f));
```

```
                b2BodyDef bd;
                b1 = m_world->CreateBody(&bd);
                b1->CreateFixture(&shape, 0.0f);
        }
        // Create the trail path
        {
                b2CircleShape shape;
                shape.m_radius = 3.0f;
                shape.m_p.Set(-3.0f, 25.0f);

                b2FixtureDef fd;
                fd.shape = &shape;
                fd.isSensor = true;
                m_sensor = ground->CreateFixture(&fd);
        }

        // Create the ball
        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
        ballShape.m_radius = 0.5f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.75f;
        ballFixtureDef.density = 3.3f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(0.0f, 25.0f);

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);
        m_ball->SetAngularVelocity(1.0f);
        m_time = 0.0f;
}
b2Body* m_ball;
float m_time;
b2Fixture* m_sensor;
void Step(Settings& settings) override
{
        b2Vec2 v = m_ball->GetLinearVelocity();
        float r = 3.0f;
        float omega = m_ball->GetAngularVelocity();
        float angle = m_ball->GetAngle();
        b2MassData massData = m_ball->GetMassData();
```

```
                    b2Vec2 position = m_ball->GetPosition();
                    float sin = sinf(angle);
                    float cos = cosf(angle);
                    float ball_vel = 3.0f;
                    float a = (ball_vel*ball_vel) / r;
                    m_time += 1.0f / 60.0f; // assuming we are using frequency of
                         60 Hertz

                    float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
                        massData.I * omega * omega;

                    m_ball->SetLinearVelocity(b2Vec2(-ball_vel*sinf(angle),
                        ball_vel*cosf(angle)));

                    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                        f", m_time);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Ball position, x = %.6
                        f", position.x);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Ball position, y = %.6
                        f", position.y);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
                        .mass);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
                        , ke);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Linear velocity = %.6f
                        ", v);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Linear velocity, x =
                        %.6f", v.x);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Linear velocity, y =
                        %.6f", v.y);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Acceleration, a = %.6f
                        ", a);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =
                        %.6f", angle*RADTODEG);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6f",
                        sin);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6f",
```

```
                    cos);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
                    position.x, position.y, angle*RADTODEG, v.x, v.y, sin,
                    cos);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new CircularMotion;
        }

};

static int testIndex = RegisterTest("Motion in 2D", "Circular Motion",
    CircularMotion::Create);
```

C++ Code 27: *tests/circular_motion.cpp "Uniform Circular Motion Box2D"*

Some explanations for the codes:

- First, we create the physics world with gravity of 0 (not realistic on earth, but it can be used to show uniform circular motion on space), then we create the trail path for the particle that is moving, a circle shape with color of green.

```
            m_world->SetGravity(b2Vec2(0.0f,0.0f));
            b2BodyDef bd;
            b2Body* ground = m_world->CreateBody(&bd);
            b2Body* b1;
            {
                    b2EdgeShape shape;
                    shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
                        0.0f));

                    b2BodyDef bd;
                    b1 = m_world->CreateBody(&bd);
                    b1->CreateFixture(&shape, 0.0f);
            }
            // Create the trail path
            {
                    b2CircleShape shape;
                    shape.m_radius = 3.0f;
                    shape.m_p.Set(-3.0f, 25.0f);

                    b2FixtureDef fd;
```

```
        fd.shape = &shape;
        fd.isSensor = true;
        m_sensor = ground->CreateFixture(&fd);
}
```

- To create the circle (funny that we always call this circle as a ball in the code) with every details necessary, set the density, radius, restitution, friction, starting position. At the end we set the value for a new variable called **m_time** as 0 to calculate time since the beginning of the simulation.

```
// Create the ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 3.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
    );
m_ball->SetAngularVelocity(1.0f);
m_time = 0.0f;
```

- The variable declarations

```
b2Body* m_ball;
float m_time;
b2Fixture* m_sensor;
```

- We are showing some useful information on the moving particle that is in uniform circular motion, we can get the velocity and the position for the particle in x and y axis. The trick to make it revolving around a fixed center / in uniform circular motion, is to set the velocity to be changing with time to follow the formula (**6.1**), with $v = 3$, since the sine and cosine of an angle will always be bounded and periodic.

```
void Step(Settings& settings) override
{
        b2Vec2 v = m_ball->GetLinearVelocity();
        float r = 3.0f;
        float omega = m_ball->GetAngularVelocity();
        float angle = m_ball->GetAngle();
        b2MassData massData = m_ball->GetMassData();
```

```
                    b2Vec2 position = m_ball->GetPosition();
                    float sin = sinf(angle);
                    float cos = cosf(angle);
                    float ball_vel = 3.0f;
                    float a = (ball_vel*ball_vel) / r;
                    m_time += 1.0f / 60.0f; // assuming we are using
                        frequency of 60 Hertz

                    float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
                        massData.I * omega * omega;

                    m_ball->SetLinearVelocity(b2Vec2(-ball_vel*sinf(angle),
                         ball_vel*cosf(angle)));

                    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
                        = %.6f", m_time);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Ball position, x
                        = %.6f", position.x);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Ball position, y
                        = %.6f", position.y);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f",
                        massData.mass);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Kinetic energy =
                        %.6f", ke);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Linear velocity =
                         %.6f", v);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Linear velocity,
                        x = %.6f", v.x);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Linear velocity,
                        y = %.6f", v.y);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Acceleration, a =
                         %.6f", a);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees
                        ) = %.6f", angle*RADTODEG);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6
                        f", sin);
                    m_textLine += m_textIncrement;
                    g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6
```

```
                    f", cos);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                     graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
                    position.x, position.y, angle*RADTODEG, v.x, v.y,
                    sin, cos);

                Test::Step(settings);
            }
```

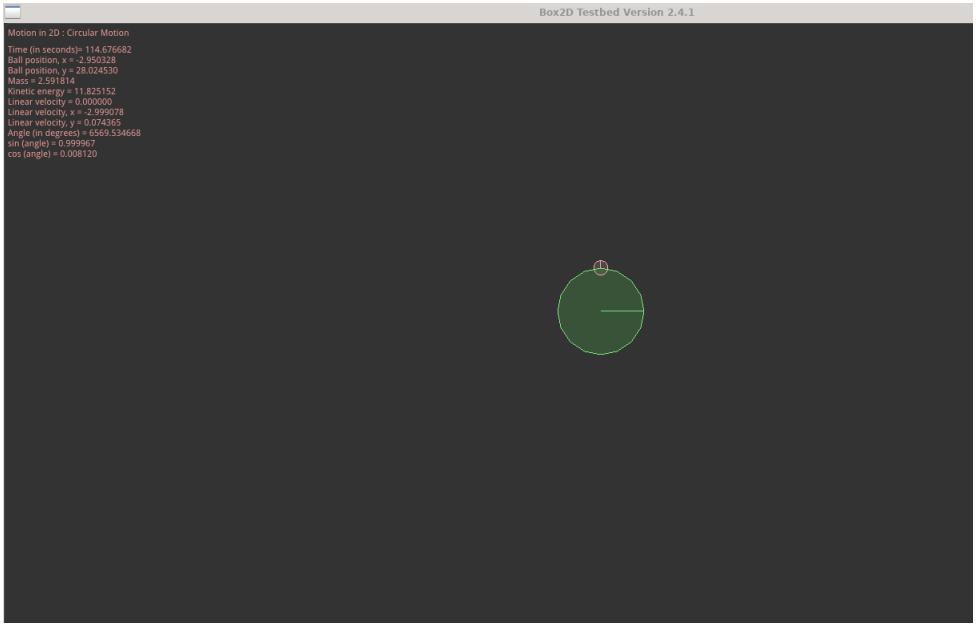

Figure 6.10: *The simulation of a particle doing circular motion with speed of $\vec{v} = 3$ (the current simulation code can be located in: **/DianFreya-box2d-testbed/tests/circular_motion.cpp**).*

After recompiling this, you can save it into textfile by opening the testbed with this command:
./testbed > circularoutput.txt
After you close the testbed to record the result, then see it at the current working directory where **testbed** is located and see **circularoutput.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only.

To plot the velocity in x and y axis for the uniform circular motion of the ball with respect to time, now open terminal at the directory containing the **circularoutput.txt** and type:
gnuplot
set xlabel "time"
plot "circularoutput.txt" using 4 title "v_{x}" with lines, "circularoutput.txt" using 5 title "v_{y}" with lines

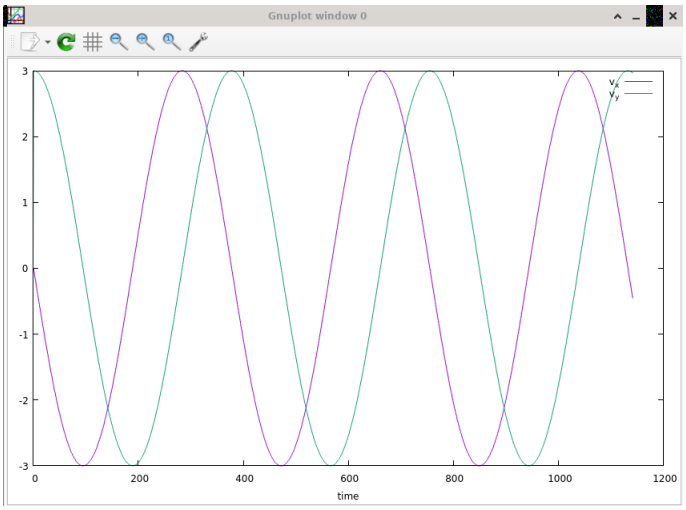

Figure 6.11: *The gnuplot of the velocity of the ball particle is a negative sine function for v_y and a cosine function for v_x*

Chapter 7

DianFreya Math Physics Simulator II: Force and Motion

*"Remember how it all started by Force? You(Freya) asked me to create something better than BLAS and LAPACK with FORTRAN, or sleep 24/7, but why am I writing this now?" - **DS Glanzsche***

What cause an object to move? The answer is Force. I was forced to study FORTRAN, thus I take a FORTRAN book and read it then study, there it is one good example, since the alternative is if I don't want to study FORTRAN, I better sleep 24/7, well it is not good for my soul, happiness and body to sleep 24/7 unless I am in a comma, so I read FORTRAN based on Force. But, C++ comes first before FORTRAN, as there is always a prelude of something, appetizer before the main course.

The relation between a force and the acceleration it causes was first understood by Isaac Newton(1642-1727), it is known as Newtonian mechanics. But, if the speed of the interacting bodies are too large, converging to the speed of light, we will use Einstein's theory of relativity instead.

The velocity of an object can change (the object can accelerate or deccelerate) when the object is acted on by one or more forces (pushes or pulls) from other objects. Newtonian mechanics relates accelerations and forces.

Forces are vector quantities. Masses are scalar quantities. Their magnitudes are defined in terms of the acceleration times the mass of the object. A force that accelerates the standard body by exactly 1 m/s^2 is defined to have a magnitude of 1 N. The direction of a force is the direction of the acceleration it causes. Forces are combined according to the rules of vector algebra. The net force on a body is the vector sum of all the forces acting on the body.

If there is no net force on a body, the body remains at rest. The body is called to be in motion when it moves in a straight line at constant speed.

Reference frames in which Newtonian mechanics holds are called inertial reference frames or inertial frames.

Theorem 7.1: Newton's Laws

Newton's First Law
If no net force acts on a body ($F_{net} = 0$), the body's velocity cannot change; that is, the body cannot accelerate.

Newton's Second Law
The net force on a body is equal to the product of the body's mass and its acceleration.

In equation form,

$$\vec{F}_{net} = m\vec{a} \tag{7.1}$$

The acceleration component along a given axis is caused only by the sum of the force components along that same axis, and not by force components along any other axis.

Newton's Third Law

I. Simulation for Newton's First Law with Box2D

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class NewtonFirstlaw : public Test
{
	public:
	NewtonFirstlaw()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2
			    (-46.0f, 46.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2
			    (46.0f, 46.0f));
			ground->CreateFixture(&shape, 0.0f);
```

```
                        }
                        {
                               b2BodyDef bd;
                               ground = m_world->CreateBody(&bd);

                               b2EdgeShape shape;
                               shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2
                                   (46.0f, 0.0f));
                               ground->CreateFixture(&shape, 0.0f);
                        }
                        {
                               b2BodyDef bd;
                               ground = m_world->CreateBody(&bd);

                               b2EdgeShape shape;
                               shape.SetTwoSided(b2Vec2(-46.0f, 12.0f), b2Vec2
                                   (46.0f, 12.0f));
                               ground->CreateFixture(&shape, 0.0f);
                        }
                        {
                               b2BodyDef bd;
                               ground = m_world->CreateBody(&bd);

                               b2EdgeShape shape;
                               shape.SetTwoSided(b2Vec2(-46.0f, 23.0f), b2Vec2
                                   (46.0f, 23.0f));
                               ground->CreateFixture(&shape, 0.0f);
                        }

                        // Create the box as the movable object with friction
                            0
                        b2PolygonShape boxShape;
                        boxShape.SetAsBox(0.5f, 0.5f);

                        b2FixtureDef boxFixtureDef;
                        boxFixtureDef.restitution = 0.75f;
                        boxFixtureDef.density = 7.3f; // this will affect the
                            box mass
                        boxFixtureDef.friction = 0.0f;
                        boxFixtureDef.shape = &boxShape;

                        b2BodyDef boxBodyDef;
                        boxBodyDef.type = b2_dynamicBody;
                        boxBodyDef.position.Set(5.0f, 0.5f);

                        m_box = m_world->CreateBody(&boxBodyDef);
                        b2Fixture *boxFixture = m_box->CreateFixture(&
                            boxFixtureDef);
```

```
                // Create the box 2 with friction 0.5
                b2PolygonShape boxShape2;
                boxShape2.SetAsBox(0.5f, 0.5f);

                b2FixtureDef boxFixtureDef2;
                boxFixtureDef2.restitution = 0.75f;
                boxFixtureDef2.density = 7.3f; // this will affect the
                     box mass
                boxFixtureDef2.friction = 0.5f;
                boxFixtureDef2.shape = &boxShape2;

                b2BodyDef boxBodyDef2;
                boxBodyDef2.type = b2_dynamicBody;
                boxBodyDef2.position.Set(5.0f, 12.5f);

                m_box2 = m_world->CreateBody(&boxBodyDef2);
                b2Fixture *boxFixture2 = m_box2->CreateFixture(&
                    boxFixtureDef2);

                // Create the box 3 with friction 1
                b2PolygonShape boxShape3;
                boxShape3.SetAsBox(0.5f, 0.5f);

                b2FixtureDef boxFixtureDef3;
                boxFixtureDef3.restitution = 0.75f;
                boxFixtureDef3.density = 7.3f; // this will affect the
                     box mass
                boxFixtureDef3.friction = 1.0f;
                boxFixtureDef3.shape = &boxShape3;

                b2BodyDef boxBodyDef3;
                boxBodyDef3.type = b2_dynamicBody;
                boxBodyDef3.position.Set(5.0f, 23.5f);

                m_box3 = m_world->CreateBody(&boxBodyDef3);
                b2Fixture *boxFixture3 = m_box3->CreateFixture(&
                    boxFixtureDef3);
                m_time = 0.0f;
        }
        b2Body* m_box;
        b2Body* m_box2;
        b2Body* m_box3;
        float m_time;

        void Keyboard(int key) override
        {
                switch (key)
```

```
                {
                        case GLFW_KEY_D:
                        m_box->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                        m_box2->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                        m_box3->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                        break;
                        case GLFW_KEY_A:
                        m_box->SetLinearVelocity(b2Vec2(-10.0f, 0.0f))
                            ;
                        m_box2->SetLinearVelocity(b2Vec2(-10.0f, 0.0f)
                            );
                        m_box3->SetLinearVelocity(b2Vec2(-10.0f, 0.0f)
                            );
                        break;
                        case GLFW_KEY_F:
                        m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                             true);
                        m_box2->ApplyForceToCenter(b2Vec2(250.0f, 0.0f)
                            , true);
                        m_box3->ApplyForceToCenter(b2Vec2(250.0f, 0.0f)
                            , true);
                        break;
                        case GLFW_KEY_I:
                        m_box->ApplyLinearImpulseToCenter(b2Vec2(250.0f
                            , 0.0f), true);
                        m_box2->ApplyLinearImpulseToCenter(b2Vec2(250.0
                            f, 0.0f), true);
                        m_box3->ApplyLinearImpulseToCenter(b2Vec2(250.0
                            f, 0.0f), true);
                        break;
                }
        }
        void Step(Settings& settings) override
        {
               m_time += 1.0f / 60.0f; // assuming we are using
                    frequency of 60 Hertz
               b2MassData massData = m_box->GetMassData();
               b2Vec2 position = m_box->GetPosition();
               b2Vec2 position2 = m_box2->GetPosition();
               b2Vec2 position3 = m_box3->GetPosition();
               b2Vec2 velocity1 = m_box->GetLinearVelocity();
               b2Vec2 velocity2 = m_box2->GetLinearVelocity();
               b2Vec2 velocity3 = m_box3->GetLinearVelocity();

               g_debugDraw.DrawString(5, m_textLine, "Time (in
                    seconds)= %.6f", m_time);
               m_textLine += m_textIncrement;
```

```
                        g_debugDraw.DrawString(5, m_textLine, "Box 1 position
                            = (%4.1f, %4.1f)", position.x, position.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity
                            = (%4.1f, %4.1f)", velocity1.x, velocity1.y);
                        m_textLine += m_textIncrement;

                        g_debugDraw.DrawString(5, m_textLine, "Box 2 position
                            = (%4.1f, %4.1f)", position2.x, position2.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 2 velocity
                            = (%4.1f, %4.1f)", velocity2.x, velocity2.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 3 position
                            = (%4.1f, %4.1f)", position3.x, position3.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 3 velocity
                            = (%4.1f, %4.1f)", velocity3.x, velocity3.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 1 / 2 / 3
                            Mass = %.6f", massData.mass);
                        m_textLine += m_textIncrement;
                        Test::Step(settings);

                        printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);

                }
                static Test* Create()
                {
                        return new NewtonFirstlaw;
                }

        };

        static int testIndex = RegisterTest("Force and Motion", "Newton's
            First Law", NewtonFirstlaw::Create);
```

C++ Code 28: *tests/newton_firstlaw.cpp "Newton's First Law Box2D"*

Some explanations for the codes:

- We are creating a closed 3 different grounds each at different y axis value

```
        b2Body* ground = NULL;
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
```

```
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f,
                46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
                46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 12.0f), b2Vec2(46.0f,
                12.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 23.0f), b2Vec2(46.0f,
                23.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
```

- Next, we create three boxes with different coefficient of friction: $0, 0.5$, and 1.

```
        // Create the box as the movable object with friction 0
        b2PolygonShape boxShape;
        boxShape.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
```

```
            boxFixtureDef.density = 7.3f; // this will affect the box mass
            boxFixtureDef.friction = 0.0f;
            boxFixtureDef.shape = &boxShape;

            b2BodyDef boxBodyDef;
            boxBodyDef.type = b2_dynamicBody;
            boxBodyDef.position.Set(5.0f, 0.5f);

            m_box = m_world->CreateBody(&boxBodyDef);
            b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef);

            // Create the box 2 with friction 0.5
            b2PolygonShape boxShape2;
            boxShape2.SetAsBox(0.5f, 0.5f);

            b2FixtureDef boxFixtureDef2;
            boxFixtureDef2.restitution = 0.75f;
            boxFixtureDef2.density = 7.3f; // this will affect the box mass
            boxFixtureDef2.friction = 0.5f;
            boxFixtureDef2.shape = &boxShape2;

            b2BodyDef boxBodyDef2;
            boxBodyDef2.type = b2_dynamicBody;
            boxBodyDef2.position.Set(5.0f, 12.5f);

            m_box2 = m_world->CreateBody(&boxBodyDef2);
            b2Fixture *boxFixture2 = m_box2->CreateFixture(&boxFixtureDef2
                );

            // Create the box 3 with friction 1
            b2PolygonShape boxShape3;
            boxShape3.SetAsBox(0.5f, 0.5f);

            b2FixtureDef boxFixtureDef3;
            boxFixtureDef3.restitution = 0.75f;
            boxFixtureDef3.density = 7.3f; // this will affect the box mass
            boxFixtureDef3.friction = 1.0f;
            boxFixtureDef3.shape = &boxShape3;

            b2BodyDef boxBodyDef3;
            boxBodyDef3.type = b2_dynamicBody;
            boxBodyDef3.position.Set(5.0f, 23.5f);

            m_box3 = m_world->CreateBody(&boxBodyDef3);
            b2Fixture *boxFixture3 = m_box3->CreateFixture(&boxFixtureDef3
                );
            m_time = 0.0f;
```

Don't forget to declare the variable of the boxes, and the time as well (**m_box, m_box2, m_box3, m_time**).

- We create keyboard press events when we press "A" the boxes will suddenly have velocity of 10, and when we press "D" the boxes will have velocity of -10 (in the direction of negative x axis). If we press "F" then all the boxes will be given force toward the negative x axis with power of 250, while if we press "G" the boxes will be given force toward the positive x axis with power of 250, you can notice that box with highest friction barely move with such high force. Learn the difference of setting constant velocity on an object or particle with giving it some kind of force from the rest state.

```
            void Keyboard(int key) override
            {
                   switch (key)
                   {
                          case GLFW_KEY_D:
                          m_box->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                          m_box2->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                          m_box3->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                          break;
                          case GLFW_KEY_A:
                          m_box->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
                          m_box2->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
                          m_box3->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
                          break;
                          case GLFW_KEY_F:
                          m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f),
                               true);
                          m_box2->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f)
                              , true);
                          m_box3->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f)
                              , true);
                          break;
                          case GLFW_KEY_G:
                          m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                              true);
                          m_box2->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                              true);
                          m_box3->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                              true);
                          break;
                   }
            }
```

- Next, we will showing some position data of all the boxes, all of them share the same mass. As usual, in the end we print the velocity of box 1, so that we can plot the graph of the velocity and the acceleration.

```
            void Step(Settings& settings) override
            {
```

```
                m_time += 1.0f / 60.0f; // assuming we are using
                    frequency of 60 Hertz
                b2MassData massData = m_box->GetMassData();
                b2Vec2 position = m_box->GetPosition();
                b2Vec2 position2 = m_box2->GetPosition();
                b2Vec2 position3 = m_box3->GetPosition();
                b2Vec2 velocity1 = m_box->GetLinearVelocity();
                b2Vec2 velocity2 = m_box2->GetLinearVelocity();
                b2Vec2 velocity3 = m_box3->GetLinearVelocity();

                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
                    = %.6f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 1 position =
                    (%4.1f, %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity =
                    (%4.1f, %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Box 2 position =
                    (%4.1f, %4.1f)", position2.x, position2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 2 velocity =
                    (%4.1f, %4.1f)", velocity2.x, velocity2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 3 position =
                    (%4.1f, %4.1f)", position3.x, position3.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 3 velocity =
                    (%4.1f, %4.1f)", velocity3.x, velocity3.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 1 / 2 / 3
                    Mass = %.6f", massData.mass);
                m_textLine += m_textIncrement;
                Test::Step(settings);

                printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);

            }
```

Now, we can plot the velocity of the box with gnuplot, recompile the testbed to make the change occurs then type:
./testbed > newtonfirstoutput.txt

Plot it with gnuplot from the working directory:
gnuplot
set xlabel "time"

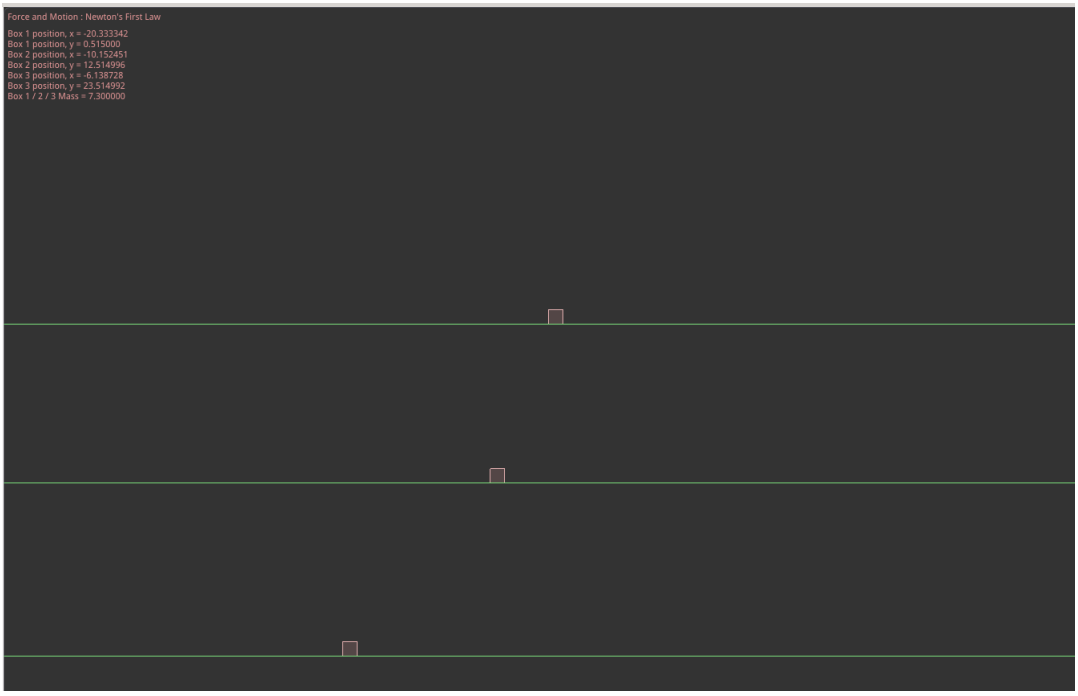

Figure 7.1: *The simulation of three boxes being pulled to the left at the speed of* $\vec{v} = 10$ *(the current simulation code can be located in:* ***/DianFreya-box2d-testbed/tests/newton_firstlaw.cpp****).*

set ylabel ""
delta_v(x) = (vD = x - old_v, old_v = x, vD)
old_v = NaN
plot "newtonfirstoutput.txt" using 1 with lines title "v_x", "newtonfirstoutput.txt" using0:(delta_v($1)) with lines title "a_{x}"

Figure it would be trickier to plot the acceleration, compared to the previous chapter. We have velocity in x axis, then we remember that instantaneous acceleration is defined as

$$\vec{a} = \frac{d\vec{v}}{dt}$$

and the average acceleration during $\triangle t$ is:

$$\vec{a}_{avg} = \frac{v_2 - v_1}{\triangle t}$$

Thus while the box is moving at constant velocity, the acceleration is read as 0 in the graph.

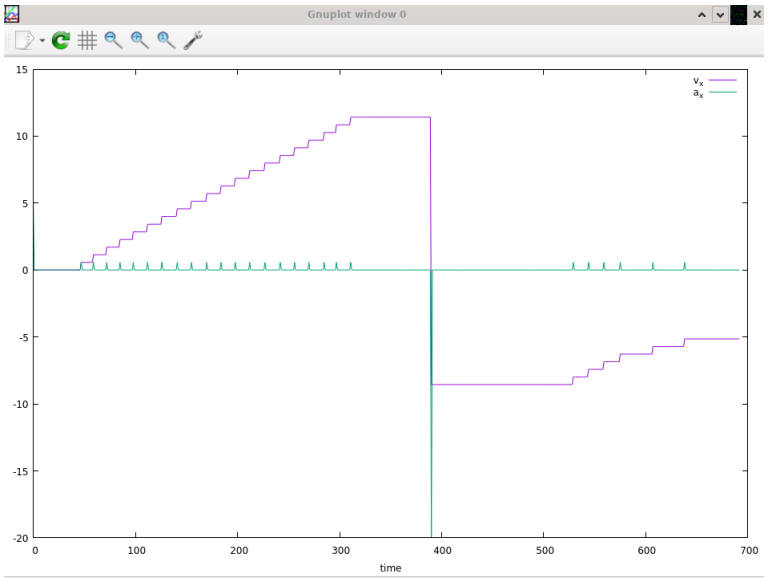

Figure 7.2: *The gnuplot of the velocity and the acceleration of box 1 that is being push with force of* 250 *toward positive x axis then hit the wall and go back then push with force of* 250 *again toward positive x axis.*

II. Simulation for Newton's Second Law with Box2D

Some explanations for the codes:

-
-

Chapter 8

DianFreya Math Physics Simulator III: Gravity

*"Nous allons voir" - **DS Glanzsche***

We are bounded by gravity, we walk, do activity, run, eat, all affected by gravity, that's why we step on this land of this planet earth. That's why rocket is designed that way with specific machine in order to go against gravity and able to get out of this planet. It should be understood, not only with theory in Physics formula and Mathematics differential equation, but why things fall that way, and for how long till it reach the ground from certain height?

This first simulation is asked by the Elder at Valhalla Projection. I thought I was asked to do some push and pulling box simulation first. But this comes first, let see then. Our favorite phrase in French, "Nous allons voir."

I. Mathematical Physics Formula for Gravity

We are going to remember how Newton' formula changes our lives, remember that:

$$F = ma$$

means that force that is applied to a body is proportional with the mass and acceleration that is ongoing with the body. Now for an object that is falling we will have:

$$F = mg$$

The different is that we regard a as the acceleration towards horizontal axis, while g is the acceleration towards the vertical axis.

Based on elementary differential equation [1], we will have:

$$\begin{aligned} F &= mg \\ m\frac{dv}{dt} &= mg \\ \text{(without air drag)} & \\ F &= mg - \gamma v \\ m\frac{dv}{dt} &= mg - \gamma v(t) \end{aligned}$$

with v is the speed or velocity of the object that is falling, a variable that depends on t, time variable, thus we will write $v(t)$ instead of just v. We need to solve the differential equation above to obtain:

$$v(t),\ x(t),\ T$$

with $v(t)$ is the solution that represents the velocity of the falling object at time t, and $x(t)$ is the distance that the objects fall at time t, and time T is the time required for the object to fall for certain height, say h meters. We need to solve this so then we can input the numerical computation into C++ code so the animation or simulation of the falling object due to gravity with (or without) air drag is realistic.

Now, since computer is really helpful for computation in symbolic and numerical terms, I choose to use JULIA to solve the differential equation above. Thus,

$$\begin{aligned} m\frac{dv}{dt} &= mg - \gamma v(t) \\ \frac{dv}{dt} &= g - \frac{\gamma v(t)}{m} \\ v(t) &= \frac{mg}{\gamma} - \frac{mge^{-\frac{\gamma t}{m}}}{\gamma} \end{aligned}$$

Now if we substitute the value for $m = 10$, an object with mass that is 10 kg, $g = 9.8\ m/s^2$, which is the gravity on earth, and the air drag coefficient is $\gamma = 0.3$, we will have:

$$v(t) = 326.667 - 326.667e^{-0.03t}$$

If we alter the input parameters, such as different mass for the object or on different planet with different gravity, we will obtain different numerical solution. Now what we need to input to C++ code is this:

$$v(t) = \frac{mg}{\gamma} - \frac{mge^{-\frac{\gamma t}{m}}}{\gamma}$$

Then we define m, g, γ, and eventually h, the height from which the object falls, later on. There should be impact as well if the speed is too fast and the height is too low, like meteor hitting on this planet on Dinosaur era, but we will talk about it later on.

```
julia> include("diffeq.jl")
The differential equation:
  d
m·──(v(t)) = g·m - γ·v(t)
  dt
Initial Value problem solution:
                   -γ·t
                   ────
                    m
       g·m   g·m·ℯ
v(t) = ─── - ──────
       γ       γ
The solution for the differential equation with certain parameter inputs:
                                                -0.03·t
v(t) = 326.666666666667 - 326.666666666667·ℯ
       _
```

Figure 8.1: *The symbolical computation for finding the solution of an object falling with JULIA and the SymPy package (ch5-1-gravitydiffeq.jl).*

It is very important and necessary, as we are having a lot of air flights and air sports currently, knowing how to jump with parachute from an airplane and estimate time to land with certain speed can be of help one day.

We are going to compare the gravity / falling bodies simulation between Bullet physics engine and ReactPhysics3D.

II. Simulation for Gravity with Bullet3, GLEW, GLFW and OpenGL

Since this is the very first simulation asked by the elder, by coincidence a book has a nice scope on this with Bullet3, thus after modifying I am going to show how to simulate 3 dynamic bodies with shape of a sphere and different masses fall from the same height. Inspired by Chapter 7 from [5], I add two other bodies. You can directly copy the files for this example or type it manually.

We can compile all examples from Bullet, but I prefer to create one like this (per project based), then compile, compiling all examples for Bullet took so long, since it already expanded. With only using the necessary library for our test simulation purpose, we won't need to compile all other examples, that in my opinion is more time saving and can help to focus more on current physics problem and its' simulation.

In a new directory, create a file by opening a terminal and type:
vim main.cpp

```
// GLEW needs to be included first
#include <GL/glew.h>

// GLFW is included next
#include <GLFW/glfw3.h>
#include "ShaderLoader.h"
#include "Camera.h"
#include "LightRenderer.h"

#include "MeshRenderer.h"
#include "TextureLoader.h"
#include <btBulletDynamicsCommon.h>
#include<chrono>
```

```
        void initGame();
        void renderScene();

        Camera* camera;
        LightRenderer* light;

        MeshRenderer* sphere;
        MeshRenderer* sphere2;
        MeshRenderer* sphere3;
        MeshRenderer* ground;

        //physics
        btDiscreteDynamicsWorld* dynamicsWorld;

        void initGame() {

               // Enable the depth testing
               glEnable(GL_DEPTH_TEST);

               ShaderLoader shader;

               GLuint flatShaderProgram = shader.createProgram("/root/
                    SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root/
                    SourceCodes/CPP/Assets/Shaders/FlatModel.fs");
               GLuint texturedShaderProgram = shader.createProgram("/root/
                    SourceCodes/CPP/Assets/Shaders/TexturedModel.vs", "/root/
                    SourceCodes/CPP/Assets/Shaders/TexturedModel.fs");

               camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3
                    (0.0f, 8.0f, 30.0f)); //camera position at y=+8 and z=+30

               light = new LightRenderer(MeshType::kTriangle, camera);
               light->setProgram(flatShaderProgram);
               light->setPosition(glm::vec3(0.0f, 3.0f, 0.0f));

               TextureLoader tLoader;

               GLuint sphereTexture = tLoader.getTextureID("/root/
                    SourceCodes/CPP/Assets/Textures/globe.jpg");
               GLuint groundTexture = tLoader.getTextureID("/root/
                    SourceCodes/CPP/Assets/Textures/ground.jpg");

               //init physics
               btBroadphaseInterface* broadphase = new btDbvtBroadphase();
               btDefaultCollisionConfiguration* collisionConfiguration = new
                     btDefaultCollisionConfiguration();
               btCollisionDispatcher* dispatcher = new btCollisionDispatcher
```

```
        (collisionConfiguration);
    btSequentialImpulseConstraintSolver* solver = new
        btSequentialImpulseConstraintSolver();

    dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,
        broadphase, solver, collisionConfiguration);
    dynamicsWorld->setGravity(btVector3(0, -0.8f, 0));

    // Sphere Rigid Body
    btCollisionShape* sphereShape = new btSphereShape(1); //
        sphere with radius 1
    btCollisionShape* sphereShape2 = new btSphereShape(1);
    btCollisionShape* sphereShape3 = new btSphereShape(1);

    // Set the initial location where the spheres fall
    btDefaultMotionState* sphereMotionState = new
        btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
        , btVector3(-3, 13, 0)));
    btDefaultMotionState* sphereMotionState2 = new
        btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
        , btVector3(0, 13, 0)));
    btDefaultMotionState* sphereMotionState3 = new
        btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
        , btVector3(3, 13, 0)));

    btScalar mass = 5.0;
    btScalar mass2 = 10.0;
    btScalar mass3 = 15.0;
    btVector3 sphereInertia(0, 0, 0);
    btVector3 sphereInertia2(0, 0, 0);
    btVector3 sphereInertia3(0, 0, 0);
    sphereShape->calculateLocalInertia(mass, sphereInertia);
    sphereShape2->calculateLocalInertia(mass2, sphereInertia2);
    sphereShape3->calculateLocalInertia(mass3, sphereInertia3);

    btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(
        mass, sphereMotionState, sphereShape, sphereInertia);
    btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI2(
        mass2, sphereMotionState2, sphereShape2, sphereInertia2);
    btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI3(
        mass3, sphereMotionState3, sphereShape3, sphereInertia3);

    btRigidBody* sphereRigidBody = new btRigidBody(
        sphereRigidBodyCI);
    btRigidBody* sphereRigidBody2 = new btRigidBody(
        sphereRigidBodyCI2);
    btRigidBody* sphereRigidBody3 = new btRigidBody(
        sphereRigidBodyCI3);
```

```
sphereRigidBody->setRestitution(1.0f);
sphereRigidBody->setFriction(1.0f);
sphereRigidBody2->setRestitution(1.0f);
sphereRigidBody2->setFriction(1.0f);
sphereRigidBody3->setRestitution(1.0f);
sphereRigidBody3->setFriction(1.0f);

dynamicsWorld->addRigidBody(sphereRigidBody);
dynamicsWorld->addRigidBody(sphereRigidBody2);
dynamicsWorld->addRigidBody(sphereRigidBody3);

// Sphere 1 Mesh
sphere = new MeshRenderer(MeshType::kSphere, camera,
    sphereRigidBody);
sphere->setProgram(texturedShaderProgram);
sphere->setTexture(sphereTexture);
sphere->setScale(glm::vec3(1.0f));

//Sphere 2 Mesh
sphere2 = new MeshRenderer(MeshType::kSphere, camera,
    sphereRigidBody2);
sphere2->setProgram(texturedShaderProgram);
sphere2->setTexture(sphereTexture);
sphere2->setScale(glm::vec3(1.0f));

// Sphere 3 Mesh
sphere3 = new MeshRenderer(MeshType::kSphere, camera,
    sphereRigidBody3);
sphere3->setProgram(texturedShaderProgram);
sphere3->setTexture(sphereTexture);
sphere3->setScale(glm::vec3(1.0f));

// Ground Rigid body
btCollisionShape* groundShape = new btBoxShape(btVector3(4.0f
    , 0.5f, 4.0f));
btDefaultMotionState* groundMotionState = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
    , btVector3(0, -2.0f, 0)));

btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI
    (0.0f, new btDefaultMotionState(), groundShape, btVector3
    (0, 0, 0));

btRigidBody* groundRigidBody = new btRigidBody(
    groundRigidBodyCI);

groundRigidBody->setFriction(1.0);
groundRigidBody->setRestitution(0.5);
```

```
        groundRigidBody->setCollisionFlags(btCollisionObject::
            CF_STATIC_OBJECT);

        dynamicsWorld->addRigidBody(groundRigidBody);

        // Ground Mesh
        ground = new MeshRenderer(MeshType::kCube, camera,
            groundRigidBody);
        ground->setProgram(texturedShaderProgram);
        ground->setTexture(groundTexture);
        ground->setScale(glm::vec3(4.0f, 0.5f, 4.0f));
}

void renderScene(){
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glClearColor(1.0, 1.0, 0.0, 1.0);

        //light->draw();
        sphere->draw();
        sphere2->draw();
        sphere3->draw();
        ground->draw();
}

int main(int argc, char **argv)
{
        glfwInit();
        GLFWwindow* window = glfwCreateWindow(800, 600, " Bullet and
            OpenGL ", NULL, NULL);
        glfwMakeContextCurrent(window);
        glewInit();

        initGame();
        auto previousTime = std::chrono::high_resolution_clock::now()
            ;
        while (!glfwWindowShouldClose(window)){

                auto currentTime = std::chrono::high_resolution_clock
                    ::now();
                float dt = std::chrono::duration<float, std::chrono::
                    seconds::period>(currentTime - previousTime).
                    count();

                dynamicsWorld->stepSimulation(dt);

                renderScene();
```

```
                glfwSwapBuffers(window);
                glfwPollEvents();

                previousTime = currentTime;
            }
            glfwTerminate();

            delete camera;
            delete light;

            return 0;
        }
```

C++ Code 29: *main.cpp "Gravity with 3 Bodies"*

Do not follow 100 %, read slowly and try to debug and tinker by yourself, if there are warnings and errors find the solutions by yourself if possible, otherwise try to ask. Another thing is do kindly adjust the path, for example at these lines:

```
GLuint flatShaderProgram = shader.createProgram("/root/SourceCodes/CPP/
    Assets/Shaders/FlatModel.vs", "/root/SourceCodes/CPP/Assets/Shaders/
    FlatModel.fs");
GLuint texturedShaderProgram = shader.createProgram("/root/SourceCodes/CPP/
    Assets/Shaders/TexturedModel.vs", "/root/SourceCodes/CPP/Assets/Shaders
    /TexturedModel.fs");
```

Your path shall not be the same as mine: **/root/SourceCodes/CPP/Assets/Shaders/FlatModel.fs** for **FlatModel.fs**, a fragment shader. All the shader and texture and images related are inside repository as well, you can suit yourself if you like it use that or use your own.

Some explanations for the codes:

- To add physics we use:
  #include <btBulletDynamicsCommon.h>

- The Bullet libraries we are using are: **BulletCollision, BulletDynamics, LinearMath**. We will need to link to this library when compiling either with CMake or manual compiling.
- To create a Physics world we use this:
  btDiscreteDynamicsWorld* dynamicsWorld;
- After create the Physics world, then inside the **void initGame()** we will have:

```
btBroadphaseInterface* broadphase = new btDbvtBroadphase();
btDefaultCollisionConfiguration* collisionConfiguration = new
    btDefaultCollisionConfiguration();
btCollisionDispatcher* dispatcher = new btCollisionDispatcher(
    collisionConfiguration);
btSequentialImpulseConstraintSolver* solver = new
    btSequentialImpulseConstraintSolver();
```

  Collision detection is done in two phase: broadphase (the physics engine eliminates all the objects that are unlikely to collide) and narrowphase (the actual shape of the object is used to

check the likelihood of a collision). Pairs of object are created with a strong likelihood of collision. For **btCollisionDispatcher**, a pair of objects that have a strong likelihood for colliding are tested for collision using their actual shapes. For **btSequentialImpulseConstraintSolver**, you can create constraints, such as a hinge constraint or slider constraint which can restrict motion or rotation of one object about another object. The calculation is repeated a number of times to get the optimal solution.

- Still in **void initGame()**:

```
dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,
    broadphase, solver, collisionConfiguration);
dynamicsWorld->setGravity(btVector3(0, -9.8f, 0));
```

to create a new **dynamicsWorld** by passing the **dispatcher, broadphase, solver**, and **collisionConfiguration** as parameters to the **btDiscreteDynamicsWorld** function. After that we can set the parameters for our physics with gravity of $-9.8f$ in the y-axis.

- After finish with the Physics world, we can create rigid bodies or soft bodies, still in **void initGame()**:

```
btCollisionShape* sphereShape = new btSphereShape(1);

btDefaultMotionState* sphereMotionState = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
    btVector3(0, 15, 0)));

btScalar mass = 10.0;
btVector3 sphereInertia(0, 0, 0);
sphereShape->calculateLocalInertia(mass, sphereInertia);

btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(mass
    , sphereMotionState, sphereShape, sphereInertia);

btRigidBody* sphereRigidBody = new btRigidBody(
    sphereRigidBodyCI);
sphereRigidBody->setRestitution(1.0f);
sphereRigidBody->setFriction(1.0f);

dynamicsWorld->addRigidBody(sphereRigidBody);
```

new btSphereShape(1); means create a sphere shape with radius of 1. The **btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1), btVector3(0, 15, 0)));** specifies the rotation and position of the sphere, it will be located on $x = 0, y = 15, z = 0$. To test it you may change the **btQuaternion(0, 0, 0, 1)** into **btQuaternion(0.3, 0, 0.5, 0.7)** to see how the sphere rotates while falling. After that, we set the mass to be 10, the inertia, and calculate inertia of the **sphereShape**.

Then, to create the rigid body, we first create **btRigidBodyConstructionInfo** and pass the rigid body' variables / parameters. We then set physical properties of the rigid body, such as the restitution and the friction. For restitution 0, means the rigid body will have no bounciness, while 1 means the rigid body is very bouncy, like a basketball. While for friction, 0 means a smooth rigid body, while 1 means a very rough rigid body.

Finally, we add the rigid body to the Physics world.

- In **Mesh.cpp**, we define the vertices for Triangle, Quad, Cube, and Sphere. Thus in **MeshRenderer.h** and **MeshRenderer.cpp** we are not only render the sphere, but to make it behave like a sphere that obeying the law of physics, we pass the rigid body to the sphere mesh.

  We use **btTransform** variable to get the transformation from the rigid body's **getMotionState** function and then get the **WorldTransform** variable and set it to our **btTransform** variable **t**.

  We create **btQuaternion** type to store rotation and **btvector3** to store translation values using the **getRotation** and **getOrigin** functions of the **btTransform** class.

  We create three **glm::mat4** variables, called **RotationMatrix, TranslationMatrix,** and **ScaleMatrix**. We set the values of rotation and translation using the **glm::rotate** and **glm::translation** functions.

The rest of the files (.cpp and .h) can be obtained at the repository, all will be (don't forget the texture, vertex and fragment shaders):

- Camera.cpp
- Camera.h
- CMakeLists.txt
- LightRenderer.cpp
- LightRenderer.h
- main.cpp
- Mesh.cpp
- Mesh.h
- MeshRenderer.cpp
- MeshRenderer.h
- ShaderLoader.cpp
- ShaderLoader.h
- TextureLoader.cpp
- TextureLoader.h

After having all of them then open the terminal at this working directory and type:
mkdir build
cd build
cmake ..
make
./result

If you want to compile manually without CMake, type:
g++ *.cpp -o result -lBulletCollision -lBulletDynamics -lLinearMath -lGLEW -lglfw -lGL -I/usr/include/stb-master
./result

This example has no air drag. If we want an air drag we need to damp the velocity when the bodies are falling. Air drag depends on the geometry shape and material of the object. On October 26th, 2023 I drop down from the same height a small dog doll and 1 Euro cent coin, they fall at the same time. But, when I do that with the Euro coin and a piece of paper, the paper still

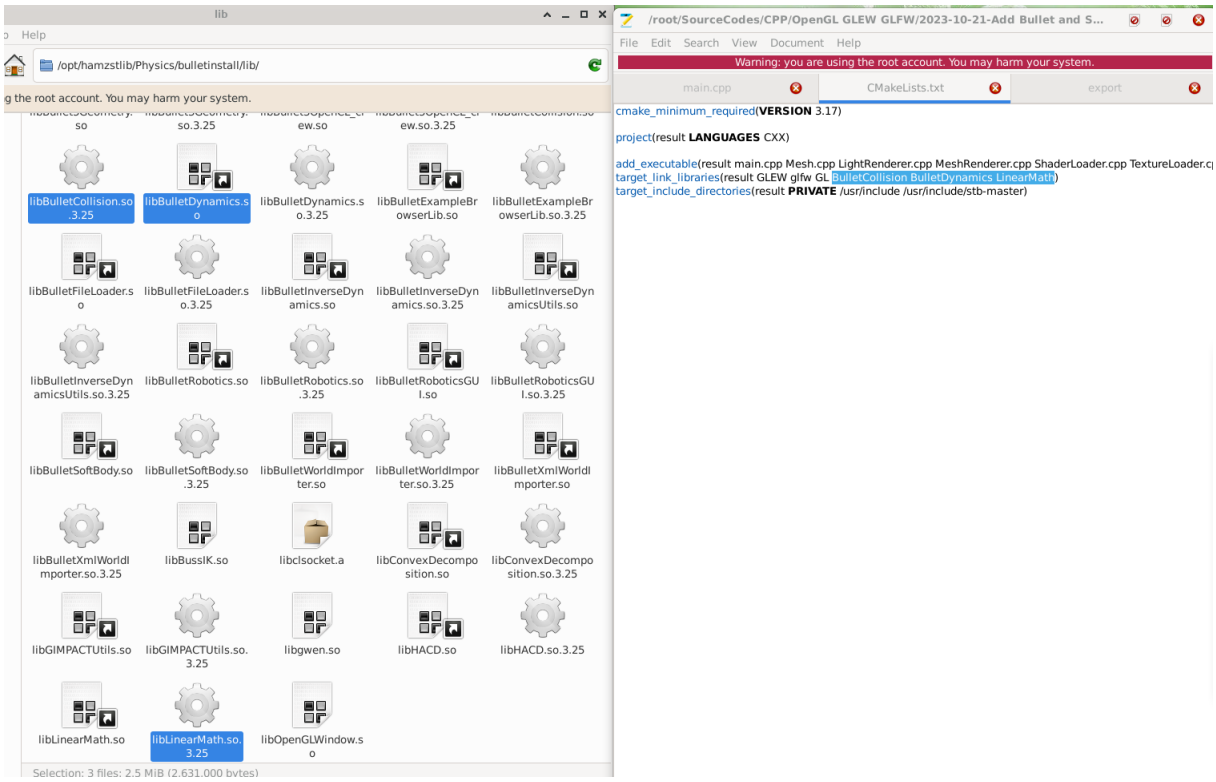

Figure 8.2: *Linking the libraries from Bullet for this simulation, make sure that you have set the right LIBRARY_PATH for the Bullet' libraries that were installed and compiled.*

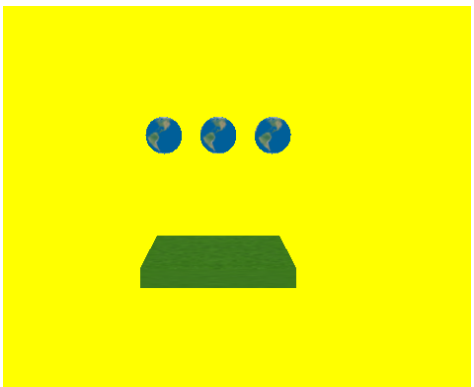

Figure 8.3: *Three falling bodies with shape of sphere, from left to right have masses of: 5, 10, then 15. (files for this example in folder:* ***ch6-gravity-Bullet and 3 Bodies****).*

flying when the coin hits the ground. That's Physics, why that happens? Back to the geometry shape and material of the paper.

Calculate the time it hits the ground. Is mass not affecting time to hit the ground? Check the formula, etc.

III. Simulation for Gravity with Box2D

i. Build DianFreya Modified Box2D Testbed

We have modified all the codes for Box2D testbed by removing the original tests codes into our own codes to learn Physics and Math from beginning, with Box2D library and imgui, learning C++, Physics and Math become more fun. You are welcome to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
mkdir build
cd build
cmake ..
make
./testbed

If you are having a difficulty or error, read again chapter 5 on how to download, compile and install Box2D from raw to become library, along with copying the headers of imgui and sajson. If you read again, besides the source codes inside **../tests/** that I replaced, the **CMakeLists.txt** is being modified to include the source codes from the newly modified source codes in **../tests/**, thus when the testbed is being build it will only show the new tests for this book simulation, not the original default Box2D testbed tests.

Look for the related simulation under the **Tests** tab on the right panel, then choose **Gravitation/Gravity Check**.

```
#include "test.h"
#include <iostream>

class GravityCheck: public Test
{
	public:

	GravityCheck()
	{
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		b2Timer timer;
		// Perimeter Ground body
		{
			b2BodyDef bd;
			b2Body* ground = m_world->CreateBody(&bd);

			b2EdgeShape shapeGround;
			shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
			    (30.0f, 0.0f));
```

```
                ground->CreateFixture(&shapeGround, 0.0f);

                b2EdgeShape shapeTop;
                shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2
                    (30.0f, 30.0f));
                ground->CreateFixture(&shapeTop, 0.0f);

                b2EdgeShape shapeLeft;
                shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                    (-40.0f, 30.0f));
                ground->CreateFixture(&shapeLeft, 0.0f);

                b2EdgeShape shapeRight;
                shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
                    (30.0f, 30.0f));
                ground->CreateFixture(&shapeRight, 0.0f);
            }

            // Create the left ball
            b2CircleShape ballShape2;
            ballShape2.m_p.SetZero();
            ballShape2.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef2;
            ballFixtureDef2.restitution = 0.75f; // the bounciness
            ballFixtureDef2.density = 7.3f; // this will affect the ball
                mass
            ballFixtureDef2.friction = 0.1f;
            ballFixtureDef2.shape = &ballShape2;

            b2BodyDef ballBodyDef2;
            ballBodyDef2.type = b2_dynamicBody;
            ballBodyDef2.position.Set(-10.0f, 25.0f);
            // ballBodyDef2.angularDamping = 0.2f;

            m_ball2 = m_world->CreateBody(&ballBodyDef2);
            b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
                ballFixtureDef2);

            // Create the right ball
            b2CircleShape ballShape;
            ballShape.m_p.SetZero();
            ballShape.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef;
            ballFixtureDef.restitution = 0.75f; // the bounciness
            ballFixtureDef.density = 3.3f; // this will affect the ball
                mass
```

```
                ballFixtureDef.friction = 0.1f;
                ballFixtureDef.shape = &ballShape;

                b2BodyDef ballBodyDef;
                ballBodyDef.type = b2_dynamicBody;
                ballBodyDef.position.Set(0.0f, 25.0f);
                // ballBodyDef.angularDamping = 0.2f;

                m_ball = m_world->CreateBody(&ballBodyDef);
                b2Fixture *ballFixture = m_ball->CreateFixture(&
                    ballFixtureDef);
                m_createTime = timer.GetMilliseconds();
        }
        b2Body* m_ball;
        b2Body* m_ball2;
        void Step(Settings& settings) override
        {
                b2MassData massData2 = m_ball2->GetMassData();
                g_debugDraw.DrawString(5, m_textLine, "Left Ball Mass = %.6f"
                    , massData2.mass);
                m_textLine += m_textIncrement;

                b2Vec2 position2 = m_ball2->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Left Ball Position, x
                    = %.6f", position2.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Ball Position, y
                    = %.6f", position2.y);
                m_textLine += m_textIncrement;

                b2MassData massData = m_ball->GetMassData();
                g_debugDraw.DrawString(5, m_textLine, "Right Ball Mass = %.6f
                    ", massData.mass);
                m_textLine += m_textIncrement;

                b2Vec2 position = m_ball->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Right Ball Position, x
                     = %.6f", position.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Ball Position, y
                     = %.6f", position.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms
                    ",
                m_createTime);
                m_textLine += m_textIncrement;
```

```
                printf("%4.2f %4.2f \n", position2.y, position.y);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new GravityCheck;

        }

        float m_createTime;
};

static int testIndex = RegisterTest("Gravitation", "Gravity Check",
    GravityCheck::Create);
```

C++ Code 30: *tests/gravity_check.cpp "Gravity Check Box2D"*

Some explanations for the codes:

- To create the green perimeter:

```
        // Perimeter Ground body
        {
                b2BodyDef bd;
                b2Body* ground = m_world->CreateBody(&bd);

                b2EdgeShape shapeGround;
                shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                    (30.0f, 0.0f));
                ground->CreateFixture(&shapeGround, 0.0f);

                b2EdgeShape shapeTop;
                shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2(30.0f
                    , 30.0f));
                ground->CreateFixture(&shapeTop, 0.0f);

                b2EdgeShape shapeLeft;
                shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                    (-40.0f, 30.0f));
                ground->CreateFixture(&shapeLeft, 0.0f);

                b2EdgeShape shapeRight;
                shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2(30.0f
                    , 30.0f));
                ground->CreateFixture(&shapeRight, 0.0f);
        }
```

- Create two balls objects inside **GravityCheck()**

```
            // Create the left ball
            b2CircleShape ballShape2;
            ballShape2.m_p.SetZero();
            ballShape2.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef2;
            ballFixtureDef2.restitution = 0.75f; // the bounciness
            ballFixtureDef2.density = 7.3f; // this will affect the ball
                mass
            ballFixtureDef2.friction = 0.1f;
            ballFixtureDef2.shape = &ballShape2;

            b2BodyDef ballBodyDef2;
            ballBodyDef2.type = b2_dynamicBody;
            ballBodyDef2.position.Set(-10.0f, 25.0f);
            // ballBodyDef2.angularDamping = 0.2f;

            m_ball2 = m_world->CreateBody(&ballBodyDef2);
            b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
                ballFixtureDef2);

            // Create the right ball
            b2CircleShape ballShape;
            ballShape.m_p.SetZero();
            ballShape.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef;
            ballFixtureDef.restitution = 0.75f; // the bounciness
            ballFixtureDef.density = 3.3f; // this will affect the ball
                mass
            ballFixtureDef.friction = 0.1f;
            ballFixtureDef.shape = &ballShape;

            b2BodyDef ballBodyDef;
            ballBodyDef.type = b2_dynamicBody;
            ballBodyDef.position.Set(0.0f, 25.0f);
            // ballBodyDef.angularDamping = 0.2f;

            m_ball = m_world->CreateBody(&ballBodyDef);
            b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
                );
            m_createTime = timer.GetMilliseconds();
```

- Declare each of the ball as Box2D body.

```
            b2Body* m_ball;
            b2Body* m_ball2;
```

- To show on the top left of the screen for the x and y position of each ball (left and right), along with its mass. The y position for each ball will be printed out on the terminal / xterm

```
void Step(Settings& settings) override
{
       b2MassData massData2 = m_ball2->GetMassData();
       g_debugDraw.DrawString(5, m_textLine, "Left Ball Mass =
           %.6f", massData2.mass);
       m_textLine += m_textIncrement;

       b2Vec2 position2 = m_ball2->GetPosition();
       g_debugDraw.DrawString(5, m_textLine, "Left Ball
           Position, x = %.6f", position2.x);
       m_textLine += m_textIncrement;
       g_debugDraw.DrawString(5, m_textLine, "Left Ball
           Position, y = %.6f", position2.y);
       m_textLine += m_textIncrement;

       b2MassData massData = m_ball->GetMassData();
       g_debugDraw.DrawString(5, m_textLine, "Right Ball Mass =
            %.6f", massData.mass);
       m_textLine += m_textIncrement;

       b2Vec2 position = m_ball->GetPosition();
       g_debugDraw.DrawString(5, m_textLine, "Right Ball
           Position, x = %.6f", position.x);
       m_textLine += m_textIncrement;
       g_debugDraw.DrawString(5, m_textLine, "Right Ball
           Position, y = %.6f", position.y);
       m_textLine += m_textIncrement;

       g_debugDraw.DrawString(5, m_textLine, "create time =
           %6.2f ms",
       m_createTime);
       m_textLine += m_textIncrement;

       printf("%4.2f %4.2f \n", position2.y, position.y);

       Test::Step(settings);
}
```

Now, if you want to save the output into textfile (.txt), you can type this before running testbed from terminal:
./testbed > ballgravity.txt
it will save the textfile named **ballgravity.txt** inside the directory containing the **testbed**, watch out that the first two lines usually contain strings, for my case it is the data of my OpenGL, GLSL and Mesa version, just delete that strings and leave the two columns of numerical data only.

Now, we can use **Gnuplot** to help plot the y position data (the height) of each ball with respect to time, now open terminal at the directory containing the **ballgravity.txt** and type:
gnuplot
set xlabel "time"
set ylabel "y position"
plot "ballgravity.txt" using 1 title "Ball mass 5.73" with linespoints, "ballgravity.txt" using 2 title "Ball mass 2.59" with linespoints

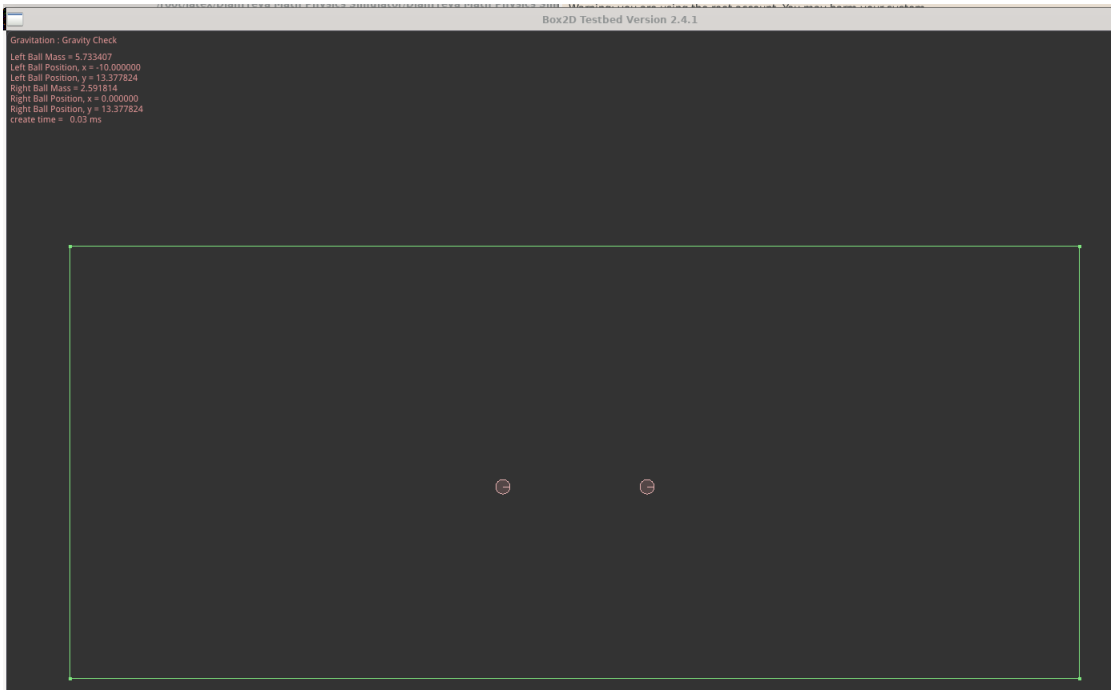

Figure 8.4: *The falling object due to gravity simulation with Box2D, it can be played when you build the testbed. (the current simulation code can be located in:* ***/DianFreya-box2d-testbed/tests/gravity_check.cpp****).*

IV. Simulation for Gravity with ReactPhysics3D

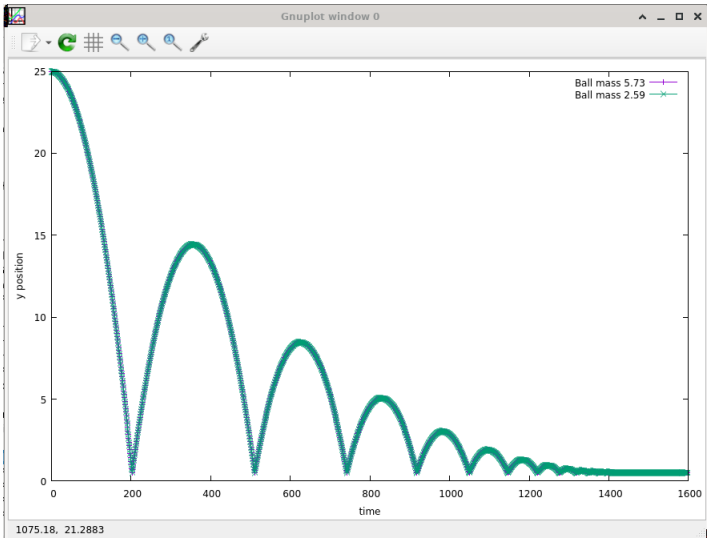

Figure 8.5: *The Gnuplot of the **ballgravity.txt**. We want to see the behavior of same shape objects fall from same height, same restitution / bounciness and friction. The only different is the mass. Turns out they have the exact same behavior while falling.*

Chapter 9

DianFreya Math Physics Simulator IV: Oscillations

*"You don't need a reason to study science and engineering, it is better than wasting time partying or gossiping. Great minds always discuss idea not discuss people." - **DS Glanzsche***

*"What I like the most about DS Glanzsche: Her 'I will try, but I can't promise I will make it right..' returns in something better than asked or expected" - **Sentinel***

When objects move back and forth repeatedly or periodically then that objects are oscillating [4]. If you play PlayStation with the vibrating joystick, when it vibrates, it is oscillating. I believe we don't need a lot of reasons to learn science, learn it as a means to spend time worthwhile instead of finding a nice financial reimbursement reason.

Breakthrough for real are made by people like Leonardo Da Vinci and Isaac Newton (never marry and die as virgin, after hundreds of years their inventions still used and for hundreds of years more), transmuting sexual energy and desire into productive energy worth more than exponential power of compound interest. Real scientists and engineers don't do orgy or sex party, their party or celebration granted by the divine, as an example one of their creations bought by royalty for USD 400 million for a painting of "Salvator Mundi", or named as one of the great airport that sell a nice Insalata Salmon and Croissant.

I. Simple Harmonic Motion

The frequency of the oscillation, denoted by f, is the number of times per second that it completes a full oscillation (a cycle) and has the unit of hertz (Hz), where

$$1\ Hz = 1\ s^{-1}$$

The time for one full cycle is the period T of the oscillation, which is

$$T = \frac{1}{f} \tag{9.1}$$

A simple harmonic motion (SHM) is a sinusoidal function of time t. That is, it can be written as a sine or a cosine of time t. Therefore

$$x(t) = x_m \ \cos(\omega t + \phi) \tag{9.2}$$

with $x(t)$ is the displacement at time t, x_m is the amplitude, t is time, ϕ is the phase constant or phase angle, ω is the angular frequency of the motion, together $(\omega t + \phi)$ is the phase.

To relate the angular frequency to the frequency f and period T, note that the position $x(t)$ of the object / particle must return to its initial value at the end of a period. That is, if $x(t)$ is the position at some chose time t, then the particle must return to that same position at time $t + T$. Returning to the same position can be written as

$$x_m \ \cos \ \omega t = x_m \ \cos \ \omega(t + T) \tag{9.3}$$

with

$$\begin{aligned} \omega(t + T) &= \omega t + 2\pi \\ \omega T &= 2\pi \\ \omega &= \frac{2\pi}{T} = 2\pi f \end{aligned}$$

the SI unit of angular frequency is the radian per second.

Now to find the velocity of the simple harmonic motion, we use the derivative with respect to time.

$$\begin{aligned} v(t) &= \frac{dx(t)}{dt} \\ &= \frac{d}{dt}[x_m \ \cos(\omega t + \phi)] \\ &= -\omega \ x_m \ \sin(\omega t + \phi) \end{aligned}$$

The velocity depends on time because the sine function varies with time, with the range of value for sine is $[-1, 1]$.

Now, if we differentiate the velocity of the simple harmonic motion, we will obtain the acceleration function of the particle in simple harmonic motion:

$$\begin{aligned} a(t) &= \frac{dv(t)}{dt} \\ &= \frac{d}{dt}[-\omega \ x_m \ \sin(\omega t + \phi)] \\ &= -\omega^2 \ x_m \ \cos(\omega t + \phi) \end{aligned}$$

The acceleration varies as well, because cosine function varies with time. Thus, we will see that

$$a(t) = -\omega^2 x(t)$$

- The particle's acceleration is always opposite its displacement
- To tell that a physical phenomena is in the category of simple harmonic motion, its acceleration and displacement will be related by a constant ω^2, with ω is the angular frequency of the motion.

i. The Force Law for Simple Harmonic Motion

We can apply Newton's second law to describe the force responsible for simple harmonic motion:

$$F = ma = m(-\omega^2 x) = -(m\omega^2)x$$

The minus sign means that the direction of the force on the particle is opposite the direction of the displacement. In SHM, the force is a restoring force that it fights against the displacement, attempting to restore the particle to the center point at $x = 0$.

If we relate it with Hooke's law that stated

$$F = -kx$$

we can relate the spring constant k (a measure of the stiffness of the spring) to the mass of the block and the resulting angular frequency of the simple harmonic motion, we will obtain:

$$k = m\omega^2$$

The block-spring system is called a linear simple harmonic oscillator, since F is proportional to x to the first power. If you ever see a situation in which the force in an oscillation is always proportional to the displacement but in the opposite direction, you can say that the oscillation is simple harmonic motion. If you know the oscillating mass, you can determine the angular frequency of the motion as

$$\omega = \sqrt{\frac{k}{m}}$$

you can also determine the period of the motion with

$$T = 2\pi\sqrt{\frac{m}{k}}$$

Every oscillating system, like a violin string, has some element of "springiness" and some element of "inertia" or mass.

II. Energy in Simple Harmonic Motion

A particle in simple harmonic motion has, at any time, kinetic energy of

$$K = \frac{1}{2}mv^2$$

and potential energy of

$$U = \frac{1}{2}kx^2$$

If no friction is present, the mechanical energy of the oscillator will be

$$E = K + U \tag{9.4}$$

remains constant, even though K and U change over time.

Now, if we see the potential energy as the function of time

$$U(t) = \frac{1}{2}kx^2 = \frac{1}{2}kx_m^2\cos^2(\omega t + \phi)$$

we can also write the kinetic energy as the function of time

$$K(t) = \frac{1}{2}mv^2 = \frac{1}{2}kx_m^2\sin^2(\omega t + \phi)$$

combining both, we will get

$$\begin{aligned} E &= U(t) + K(t) \\ &= \frac{1}{2}kx_m^2\cos^2(\omega t + \phi) + \frac{1}{2}kx_m^2\sin^2(\omega t + \phi) \\ &= \frac{1}{2}kx_m^2[\cos^2(\omega t + \phi) + \sin^2(\omega t + \phi)] \\ E &= \frac{1}{2}kx_m^2 \end{aligned}$$

it is stated that the mechanical energy of a linear oscillator is indeed constant and independent of time.

III. An Angular Simple Harmonic Oscillator

A torsion pendulum is an angular version of a linear simple harmonic oscillator. The disk oscillates in a horizontal plane between the angular amplitude $-\theta_m$ and θ_m. The element of springiness or elasticity is associated with the twisting of a suspension wire rather than the extension and compression of a spring.

If we rotate the disk by some angular displacement θ from its rest position ($\theta = 0$) and release it, it will oscillate about that position in angular simple harmonic motion. Rotating the disk through an angle θ introduces a restoring torque given by

$$\tau = -\kappa\theta$$

with κ is the torsion constant, that depends on the length, diameter and material of the suspension wire. If we think of it as the angular form of Hooke's law, the period for the angular simple harmonic motion will be

$$T = 2\pi\sqrt{\frac{I}{\kappa}}$$

we replace the mass from simple harmonic motion with its equivalent, I, the rotational inertia of the oscillating disk. The same as we replace k, the spring constant, with κ, the torsion constant.

IV. Pendulums, Circular Motion

V. The Equation of Motion for Simple Pendulum

A simple pendulum [2], which consists of a string of length l with a point mass m on the end will have the position of the pendulum specified by the angle θ between the string and the vertical line.

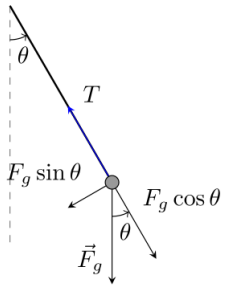

Figure 9.1: *A simple pendulum with the forces acting on the pendulum' particle (called bob of the pendulum) with mass m are the gravitational force $\vec{F}_g$ and the force $\vec{T}$ from the string. The tangential component $F_g \sin\theta$ of the gravitational force is a restoring force that tends to bring the pendulum back to its central position.*

The equation of motion of the pendulum is given by:

$$\tau = I\theta^{(2)} = I\frac{d^2\theta}{dt^2} \tag{9.5}$$

where I is the moment of inertia and is given by

$$I = ml^2$$

The torque τ is given by

$$\tau = -mgl \sin\theta$$

thus

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0 \tag{9.6}$$

VI. Simple Pendulum Simulation with Box2D

i. Build DianFreya Modified Box2D Testbed

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
mkdir build
cd build
cmake ..
make
./testbed

Look for the related simulation under the **Tests** tab on the right panel, then choose **Oscillations/Simple Pendulum**.

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>
```

```
class SimplePendulum : public Test
{
	public:

	SimplePendulum()
	{
		b2Body* b1;
		{
			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
			    0.0f));

			b2BodyDef bd;
			b1 = m_world->CreateBody(&bd);
			b1->CreateFixture(&shape, 0.0f);
		}
		// the two blocks below for creating boxes are only for
		    sweetener.
		{
			b2PolygonShape shape;
			shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, 0.7f); //
			    Gradient is 0.7

			b2BodyDef bd;
			bd.position.Set(6.0f, 6.0f);
			b2Body* ground = m_world->CreateBody(&bd);
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2PolygonShape shape;
			shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, -0.7f); //
			    Gradient is -0.7

			b2BodyDef bd;
			bd.position.Set(-6.0f, 6.0f);
			b2Body* ground = m_world->CreateBody(&bd);
			ground->CreateFixture(&shape, 0.0f);
		}

		b2Body* b2; // the hanging bar for the pendulum
		{
			b2PolygonShape shape;
			shape.SetAsBox(7.25f, 0.25f);

			b2BodyDef bd;
			bd.position.Set(0.0f, 37.0f);
			b2 = m_world->CreateBody(&bd);
			b2->CreateFixture(&shape, 0.0f);
```

```
                }

                b2RevoluteJointDef jd;
                b2Vec2 anchor;

                // Create the pendulum ball
                b2CircleShape ballShape;
                ballShape.m_p.SetZero();
                ballShape.m_radius = 0.5f;

                b2FixtureDef ballFixtureDef;
                ballFixtureDef.restitution = 0.75f;
                ballFixtureDef.density = 3.3f; // this will affect the ball
                    mass
                ballFixtureDef.friction = 0.1f;
                ballFixtureDef.shape = &ballShape;

                b2BodyDef ballBodyDef;
                ballBodyDef.type = b2_dynamicBody;
                ballBodyDef.position.Set(0.0f, 25.0f);
                // ballBodyDef.angularDamping = 0.2f;

                m_ball = m_world->CreateBody(&ballBodyDef);
                b2Fixture *ballFixture = m_ball->CreateFixture(&
                    ballFixtureDef);

                // Create the anchor and connect it to the ball
                anchor.Set(0.0f, 36.0f); // x and y axis position for the
                    Pendulum anchor
                jd.Initialize(b2, m_ball, anchor);
                //jd.collideConnected = true;
                m_world->CreateJoint(&jd); // Create the Pendulum anchor
        }

        void Step(Settings& settings) override
        {
                b2Vec2 v = m_ball->GetLinearVelocity();
                float omega = m_ball->GetAngularVelocity();
                float angle = m_ball->GetAngle();
                b2MassData massData = m_ball->GetMassData();
                b2Vec2 position = m_ball->GetPosition();

                float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
                    massData.I * omega * omega;

                g_debugDraw.DrawString(5, m_textLine, "Ball position, x = %.6
                    f", position.x);
                m_textLine += m_textIncrement;
```

```
                g_debugDraw.DrawString(5, m_textLine, "Ball position, y = %.6
                    f", position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
                    .mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
                    , ke);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Linear velocity = %.6f
                    ", v);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =
                    %.6f", angle*RADTODEG);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything
                printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle*
                    RADTODEG);
                Test::Step(settings);
        }

        static Test* Create()
        {
                return new SimplePendulum;
        }
        b2Body* m_ball;
};

static int testIndex = RegisterTest("Oscillations", "Simple Pendulum",
    SimplePendulum::Create);
```

C++ Code 31: *tests/simple_pendulum.cpp "Simple Pendulum Box2D"*

Some explanations for the codes:

- To create a line below:

```
            b2Body* b1;
            {
                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
                    0.0f));

                b2BodyDef bd;
                b1 = m_world->CreateBody(&bd);
                b1->CreateFixture(&shape, 0.0f);
            }
```

- Create a Revolute Joint named **jd** that can revolve

```
b2RevoluteJointDef jd;
b2Vec2 anchor;
```

- Create the pendulum ball, set its' position, friction, restitution, radius, density, angular damping (so the pendulum will eventually stop instead of forever oscillating) with shape of a circle, but we call it **ballShape**.

```
// Create the pendulum ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 3.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
    );
```

- Now after the ball for the pendulum is created we can connect it to the anchor with **CreateJoint();**

```
// Create the anchor and connect it to the ball
anchor.Set(0.0f, 36.0f); // x and y axis position for the
    Pendulum anchor
jd.Initialize(b2, m_ball, anchor);
//jd.collideConnected = true;
m_world->CreateJoint(&jd); // Create the Pendulum anchor
```

- To print the output of the x position, y position and the angle (in degrees) of the pendulum into the terminal / xterm

```
printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle*
    RADTODEG);
```

After recompiling this, you can save it into textfile by opening the testbed with this command:
./testbed > /root/output.txt
you may drag the pendulum to certain degree and let it swing for 5 periods or around that, afterwards close the testbed to record the result, then see it at **/root/output.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 3 columns.

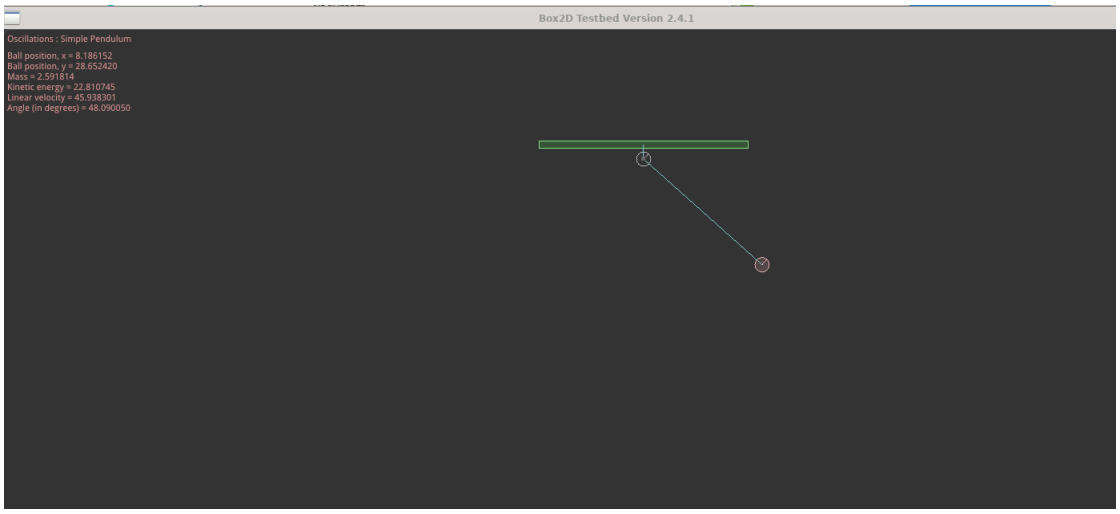

Figure 9.2: *The simple pendulum simulation with Box2D, it can be played when you build the testbed. (the current simulation code can be located in:* ***/DianFreya-box2d-testbed/tests/simple_pendulum.cpp****).*

To plot it you can use gnuplot, as it is very handy when we have raw data, then need to plot it fast, easy, and powerful even for 3-dimensional surface plot. Open terminal from the directory that contain the "output.txt" and type:

gnuplot
plot "output.txt" using 1 title "x_{m}" with lines

you can also try:
plot "output.txt" using 1 title "x_{m}" with lines, "output.txt" using 3 title "angle"

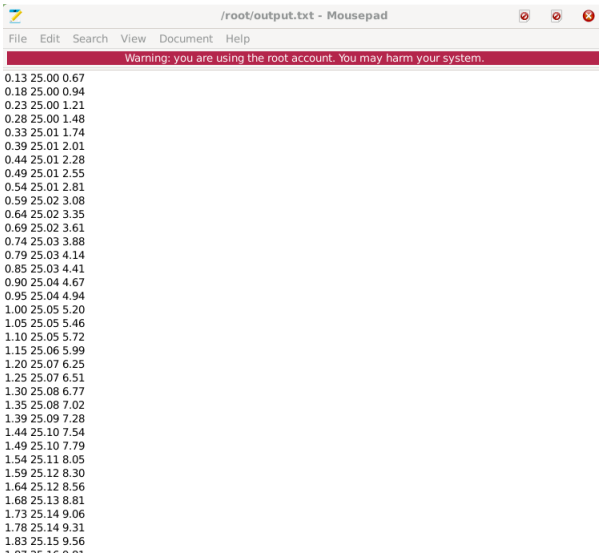

Figure 9.3: *The "output.txt" shall only contains 3 columns of numerical data.*

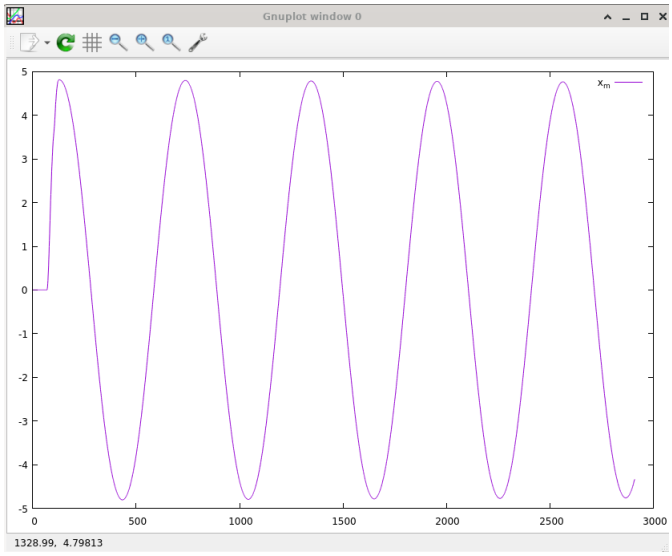

Figure 9.4: *The "output.txt" is being plotted by using gnuplot for the data of the x_m (the displacement) with respect to time.*

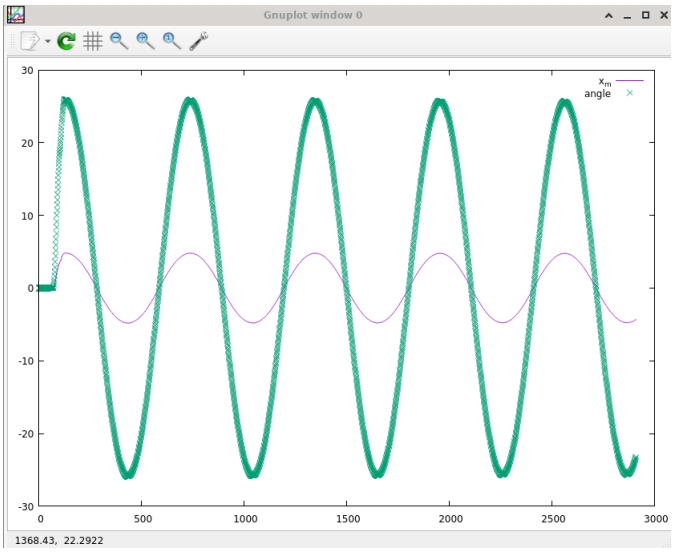

Figure 9.5: *The "output.txt" is being plotted by using gnuplot for the data of the x_m (the displacement) and the angle of the pendulum with respect to time. We may see that the angle and the displacement of the pendulum have a linear relationship.*

Box2D is an amazing Physics Engine library that can be used not only to create game but for learning all kinds of science, since we are all bounds to Physics even atoms and quantum are still kneel to the Physics. It won't be too long to gain mastery over learning this for anyone who want to invest their time, interest and disk space of their brain.

Chapter 10

DianFreya Math Physics Simulator V: Momentum and Impulse

*"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - **Albert Silverberg***

Through trial and errors we will obtain the results that we desire.

I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
mkdir build
cd build
cmake ..
make
./testbed

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 32: *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

•

Chapter 11

DianFreya Math Physics Simulator XX: Numerical Methods for Solving System of Linear Equations

*"LAM VAM RAM YAM HAM OM AH." - **7 Chakras Chanting***

There are times when people take for granted religion, spiritual or karma. Then life beats that person hole-in-one knockout, and realize a human is nothing and will end up to be dust, thus become a believer afterwards. Some believe in science like numerical methods able to solve system of linear equations, tangible, can be seen, the algorithm shows that with correct initial guess, convergence is faster. The same as faith, if you do the right technique, you worship the right Divine being / Goddess, you will get where you want to be, you will become whatever you want to become. Devoted Catholic is the founder of Ferrero, that most people will have on their table for breakfast. It will makes one to be focus on their life and soul purpose, thus anything you want, ask, then it will be given to you, like the 7 Chakras chanting, it can help you to relax more, you may compare it: a day listening and a day without listening, which one is a better initial guess that converge faster to your goal? Thus, a funny question established: is spiritual an iterative numerical methods?

Basically in this chapter, we are going to plot and compute all the available algorithms to solve system of linear equations, with given input a square matrix, we can compare these algorithms:

1. Gauss-Seidel
2. LU Decomposition
3. Jacobi
4. Gaussian elimination

Some explanations for the codes:

•

Chapter 12

DianFreya Math Physics Simulator XXI: Numerical Differentiation and Integration

*"In this parallel world, battles are fought between dharma and chaos. The balance of power brings justice. Human have needs in their world, but the world does not need them." - **Luc (Suikoden III)***

When I read a book about Numerical Methods for Chemical Engineering, the balance of chemical reaction needs to be maintained, how to calculate how many substance needed can use the algorithm to find the solution of system of linear equations or system of differential equations, thus numerical methods is a must have tools in that field of engineering. You must probably have known this one very famous package from Python language: **SymPy**. When using SymPy it is very easy to calculate the derivative or integral for all kinds of equations, univariate or even multivariate. In C++, we don't have "SymC++" yet, or something that can be on par with SymPy. Thus, we will try to find solution for integral and differential problem by resorting to numerical method / approximating the solution.

I. Numerical Differentiation

The derivative of the function f at x_0 is defined as

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \tag{12.1}$$

for small values of h, we can generate an approximation to $f'(x)$. But, it is not always successful due to roundoff error.

To approximate $f'(x_0)$, suppose first that $x_0 \in (a, b)$, where $f \in C^2[a, b]$, and that $x_1 = x_0 + h$ for some $h \neq 0$ that is sufficiently small to ensure that $x_1 \in [a, b]$. Now we will need to remember the theorem for Lagrange Polynomial

Theorem 12.1: nth Lagrange Interpolating Polynomial

If $x_0, x_1, \ldots, x_n$ are $n+1$ distinct numbers and f is a funciton whose values are given at these numbers, then a unique polynomial $P(x)$ of degreen at most n exists with

$$f(x_k) = P(x_k)$$

for each $k = 0, 1, \ldots, n$ This polynomial is given by

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x) \tag{12.2}$$

where, for each $k = 0, 1, \ldots, n$,

$$\begin{aligned} L_{n,k}(x) &= \frac{(x-x_0)(x-x_1)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_0)(x_k-x_1)\ldots(x_k-x_{k-1})(x_k-x_{k+1})(x_k-x_n)} \\ &= \prod_{i=0,\ i\neq k}^{n} \frac{(x-x_i)}{(x_k-x_i)} \end{aligned} \tag{12.3}$$

Definition 12.1: Forward and Backward-Difference Formula

Forward-difference Formula

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{h}{2}f''(\xi) \tag{12.4}$$

Backward-difference Formula

$$f'(x_0) = \frac{f(x_0-hh) - f(x_0)}{h} - \frac{h}{2}f''(\xi) \tag{12.5}$$

Some explanations for the codes:

•

II. Numerical Integration

Some explanations for the codes:

•

Chapter 13

DianFreya Math Physics Simulator XXII: Heat Equation

"Through the fire, to the limit, to the wall, for a chance to be with you, I gladly risk it all. Through the fire, to whatever come one day, for a chance of loving you, I'll take it all away. Right down through the wire, even through the fire." (Through the Fire song) - ***Chaka Khan***

If you ever played Suikoden III then you must have known the story when Flame Champion sealed his True Fire Rune to be mortal again, it causes a great fire burning all Harmonian and Grasslands armies till perish. Why release his tremendous power? For Love, like Captain America, it seems Love can bring higher power than True Rune and immortality. This heat equation is one of the most famous partial differential equation that I will try to simulate with C++, the applications are tremendous, from Black-Scholes equation derived from this, create a long lasting ice cream, to be able to create better computer components will need the comprehension of this equation as well.

Some explanations for the codes:

-

I. Introduction

Bibliography

[1] Boyce, William E., DiPrima, Richard C. (2010) Elementary Differential Equations and Boundary Value Problems 9th Edition, John Wiley & Sons, Hoboken, New Jersey, United States.

[2] Fletcher, Glena. (1994) Mathematical Methods in Physics, Wm. C. Brown Publishers, Dubuque, VA, United States.

[3] Gordon, Scott V., Clevenger, John (2019) Computer Graphics Programming in OpenGL with C++, Mercury Learning and Information, Dulles, VA, United States.

[4] Walker, Jearl, Halliday, David, Resnick, Robert. (2014) Fundamental of Physics, John Wiley & Sons, Hoboken, New Jersey, United States.

[5] Shekar, Siddharth (2019) C++ Game Development By Example, Packt, Birmingham, United Kingdom.

[6] Vries, Joey de. (2020) Learn OpenGL - Graphics Programming, Kendall & Welling.

[7] Verzani, J., (2023) Calculus With Julia

Listings