COMPUTER GAME PROGRAMMING ASSIGNMENT 1

Stuart Marples – SM861 – H00156296

Ruth Aylett & Stefano Padilla
F20GP

Contents

1. Vector Class Listings

Lab 1 vector listings

```
//slides 14 + 15 direction vectors
void dVector(glm::vec3 P1, glm::vec3 P2) {
	glm::vec3 dv = glm::vec3(P2[0] - P1[0], P2[1] - P1[1], P2[2] - P1[2]);

	cout << "direction vector: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << "\n";
}

//slide 16 represent any point along vector
void pdVector(glm::vec3 P1, glm::vec3 P2, float frac) {

	glm::vec3 dv = glm::vec3(P1[0] + (frac*P2[0]), P1[1] + (frac*P2[1]), P1[2] + (frac*P2[2]));

	cout << "point on direction vector: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << "\n";
}

//slide 17 addition of 2 vectors
void addVectors(glm::vec3 P1, glm::vec3 P2) {
	glm::vec3 dv = glm::vec3(P1[0] + P2[0], P1[1] + P2[1], P1[2] + P2[2]);

	cout << "Addition of vectors: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << "\n";
}

//slide 18 scalar vectors, same direction, length is scaled
void pdVector(glm::vec3 P1, float frac) {
	glm::vec3 dv = glm::vec3(frac*P1[0], frac*P1[1], frac*P1[2]);

	cout << "scalar vector: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << "\n";
}

//slide 19 vector length
int lVector(glm::vec3 P1) {
	float len = sqrt((P1[0] * P1[0]) + (P1[1] * P1[1]) + (P1[2] * P1[2]));

	cout << "vector length: " << len << "\n";
	return len;
}

//slide 20 normalised (unit) vectors
glm::vec3 nVector(glm::vec3 P1) {
	float len = lVector(P1);
	glm::vec3 dv = glm::vec3((P1[0] / len), (P1[1] / len), (P1[2] / len));

	cout << "normalised vector: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << "\n";
	return dv;
}

//slide 21 direction cosines
void dCosine(glm::vec3 P1) {
	glm::vec3 norm = nVector(P1);
	//multiply by 180 over pie to print in degrees
	double a = (acos(norm[0]) * (180 / 3.14159265));
	double b = (acos(norm[1]) * (180 / 3.14159265));
	double c = (acos(norm[2]) * (180 / 3.14159265));

	cout << "direction cosines: \nA:" << a << "\n";
	cout << "B:" << b << "\n";
	cout << "C:" << c << "\n";
}

//slide 22 scalar products of 2 vectors
void sVectors(glm::vec3 P1, glm::vec3 P2) {
	int dv = (P1[0] * P2[0] + P1[1] * P2[1] + P1[2] * P2[2]);

	cout << "Scalar Product: " << dv << "\n";
}

//slide 24 cross product
```

```
void cVectors(glm::vec3 P1, glm::vec3 P2) {
	glm::vec3 dv = glm::vec3((P1[1] * P2[2]) - (P1[2] * P2[1]),
		(P1[2] * P2[0]) - (P1[0] * P2[2]),
		(P1[0] * P2[1]) - (P1[1] * P2[0]));

	cout << "Cross Product: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << "\n";
}

//matrix rotation 2D is two matricies multiplied together
void rddMatrix(glm::vec2 P1, double ang) {
	glm::vec2 dv = glm::vec2(((cos(ang) - sin(ang))*P1[0]) + ((cos(ang) - sin(ang))*P1[1]),
		((sin(ang) - cos(ang))*P1[0]) + ((sin(ang) - cos(ang))*P1[1]));

	cout << "2D Rotation: " << dv[0] << ", ";
	cout << dv[1] << "\n";
}

//3D x Rotation is two matricies multiplied together
void xdddMatrix(glm::vec3 P1, double ang) {
	glm::vec4 dv = glm::vec4(P1[0], (cos(ang) * P1[1]) - (P1[2] * sin(ang)), (P1[1] * sin(ang)) + (P1[2] *
cos(ang)), 1);

	cout << "X 3D Rotation: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << ", ";
	cout << dv[3] << "\n";
}

//3D Y Rotation is two matricies multiplied together
void ydddMatrix(glm::vec3 P1, double ang) {
	glm::vec4 dv = glm::vec4((cos(ang) * P1[0]) - (P1[2] * sin(ang)), P1[1], (-(P1[0] * sin(ang))) + (P1[2] *
cos(ang)), 1);

	cout << "Y 3D Rotation: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << ", ";
	cout << dv[3] << "\n";
}

//3D Z Rotation is two matricies multiplied together
void zdddMatrix(glm::vec3 P1, double ang) {
	glm::vec4 dv = glm::vec4((cos(ang) * P1[0]) - (P1[1] * sin(ang)), (sin(ang) * P1[0]) + (P1[1] * cos(ang)),
P1[2], 1);

	cout << "Z 3D Rotation: " << dv[0] << ", ";
	cout << dv[1] << ", ";
	cout << dv[2] << ", ";
	cout << dv[3] << "\n";
}
```

Vector listings done differently

```
class vec3 {
public:
	vec3();
	vec3(float, float, float);
	bool operator==(vec3 rhs);
	vec3 operator+(vec3 rhs);
	vec3 operator-(vec3 rhs);
	vec3 operator*(vec3 rhs);
	vec3 operator/(vec3 rhs);
	vec3 operator+(float scalar);
	vec3 operator-(float scalar);
	vec3 operator*(float scalar);
	vec3 operator/(float scalar);
	vec3 cross(vec3 rhs);
	float dot(vec3 rhs);
	float length();

	float x;
	float y;
	float z;
};

vec3::vec3() {}

vec3::vec3(float, float, float) {}

bool vec3::operator==(vec3 rhs) {
	return(x == rhs.x && y == rhs.y && z == rhs.z);
}

vec3 vec3::operator+(vec3 rhs) {
         return vec3(x + rhs.x,
                  y + rhs.y,
                  z + rhs.z);
}

vec3 vec3::operator-(vec3 rhs) {
         return vec3(x - rhs.x,
                  y - rhs.y,
                  z - rhs.z);
}

vec3 vec3::operator*(vec3 rhs) {
         return vec3(x * rhs.x,
                  y * rhs.y,
                  z * rhs.z);
}

vec3 vec3::operator/(vec3 rhs) {
         return vec3(x / rhs.x,
                  y / rhs.y,
                  z / rhs.z);
}

vec3 vec3::operator/(float scalar) {
         return vec3(x / scalar,
                  y / scalar,
                  z / scalar);
}

vec3 vec3::operator*(float scalar) {
         return vec3(x * scalar,
                  y * scalar,
                  z * scalar);
}

vec3 vec3::operator+(float scalar) {
         return vec3(x + scalar,
                  y + scalar,
                  z + scalar);
}

vec3 vec3::operator-(float scalar) {
         return vec3(x - scalar,
                  y - scalar,
                  z - scalar);
}

float vec3::dot(vec3 rhs) {
         return (x * rhs.x +
                  y * rhs.y +
                  z * rhs.z);
}

vec3 vec3::cross(vec3 rhs) {
         return vec3(y * rhs.z - z * rhs.y,
                  z * rhs.x - x * rhs.z,
                  x * rhs.y - y * rhs.x);
}

float vec3::length() {
         return float(sqrt(x*x + y*y + z*z));
}
```

2.1. Particle Explosion

This lab uses the same Shapes.h, Shapes.cpp, Graphics.h and Graphics.cpp shown in section 2.2 for the bouncing ball.

Particle.h

```
#pragma once
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>

class Particle {
	public:
		Particle();
		~Particle();

		void CreateParticle(int l);
		void Draw();
		void Update(double currentTime, Graphics myGraphics);

		int timer;


		glm::vec3   position, colour, direction;
		float speed = 0.03f;
		Sphere      mySphere;


};
```

Particle.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include <GLM/gtx/transform.hpp>

#include "Graphics.h"
#include "Shapes.h"
#include "Particle.h"


Particle::Particle() {

};

Particle::~Particle() {

};

void Particle::CreateParticle(int l) {
	position = glm::vec3(0.0f, 0.0f, -6.0f);
	mySphere.Load();

	timer = l;



	float r = static_cast <float> (rand()) / static_cast <float> (RAND_MAX) * (1.0f - (-1.0f)) + (-1.0f);
	float g = static_cast <float> (rand()) / static_cast <float> (RAND_MAX) * (1.0f - (-1.0f)) + (-1.0f);
	float b = static_cast <float> (rand()) / static_cast <float> (RAND_MAX) * (1.0f - (-1.0f)) + (-1.0f);

	mySphere.fillColor = glm::vec4(r, g, b, 1.0f);
	mySphere.lineColor = glm::vec4(r, g, b, 1.0f);

	float x = static_cast <float> (rand()) / static_cast <float> (RAND_MAX) * (1.0f - (-1.0f)) + (-1.0f);
	float y = static_cast <float> (rand()) / static_cast <float> (RAND_MAX) * (1.0f - (-1.0f)) + (-1.0f);
	float z = static_cast <float> (rand()) / static_cast <float> (RAND_MAX) * (1.0f - (-1.0f)) + (-1.0f);

	direction = glm::vec3(x, y, z);
}

void Particle::Draw() {
	mySphere.Draw();
}

void Particle::Update(double currentTime, Graphics myGraphics) {

	glm::vec3 vel = direction * speed;

	position = position + vel;
```

```
         timer--;


         //Translations and scaling for sphere
         glm::mat4 mv_matrix_sphere =
                   glm::translate(glm::vec3(position)) *
                   glm::scale(glm::vec3(0.2f, 0.2f, 0.2f));
         glm::mat4(1.0f);
         mySphere.mv_matrix = mv_matrix_sphere;
         mySphere.proj_matrix = myGraphics.proj_matrix;



}
```

Source.cpp

```
#include <iostream>
#include <vector>
using namespace std;

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include <GLM/gtx/transform.hpp>

#include "graphics.h"
#include "shapes.h"
#include "Particle.h"
#include "Emitter.h"


// FUNCTIONS
void render(double currentTime);
void update(double currentTime);
void startup();
void onResizeCallback(GLFWwindow* window, int w, int h);
void onKeyCallback(GLFWwindow* window, int key, int scancode, int action, int mods);

// VARIABLES
bool               running = true;
Graphics  myGraphics;                  // Runing all the graphics in this object
Sphere             mySphere;

Emitter e;

float t = 0.001f;                    // Global variable for animation

int main()
{
         int errorGraphics = myGraphics.Init();                  // Launch window and graphics context
         if (errorGraphics) return 0;                             //Close if something went wrong...
         startup();                                                                                          // Setup all
necessary information for startup (aka. load texture, shaders, models, etc).
                                                                                                                    //
Mixed graphics and update functions - declared in main for simplicity.
         glfwSetWindowSizeCallback(myGraphics.window, onResizeCallback);                    // Set callback for
resize
         glfwSetKeyCallback(myGraphics.window, onKeyCallback);                                         // Set
Callback for keys


                                                                  // MAIN LOOP run until the window is closed
         do {
                   double currentTime = glfwGetTime();            // retrieve timelapse
                   glfwPollEvents();                                         // poll callbacks
                   update(currentTime);                                      // update (physics, animation,
structures, etc)
                   render(currentTime);                                      // call render function.
                   glfwSwapBuffers(myGraphics.window);            // swap buffers (avoid flickering and tearing)
                   running &= (glfwGetKey(myGraphics.window, GLFW_KEY_ESCAPE) == GLFW_RELEASE);         // exit if
escape key pressed
                   running &= (glfwWindowShouldClose(myGraphics.window) != GL_TRUE);
         } while (running);
         myGraphics.endProgram();                        // Close and clean everything up...
         cout << "\nPress any key to continue...\n";
         cin.ignore(); cin.get(); // delay closing console to read debugging errors.
         return 0;
}

void startup() {

         // Calculate proj_matrix for the first time.
```

```
	myGraphics.aspect = (float)myGraphics.windowWidth / (float)myGraphics.windowHeight;
	myGraphics.proj_matrix = glm::perspective(glm::radians(50.0f), myGraphics.aspect, 0.1f, 1000.0f);

	// Load Geometry
	e.Startup();
	myGraphics.SetOptimisations();
}

void update(double currentTime) {
	e.Update(currentTime, myGraphics);
}

void render(double currentTime) {
	// Clear viewport - start a new frame.
	myGraphics.ClearViewport();
	// Draw particle from emitter function
	e.render();
}

void onResizeCallback(GLFWwindow* window, int w, int h) {	// call everytime the window is resized
	myGraphics.windowWidth = w;
	myGraphics.windowHeight = h;

	myGraphics.aspect = (float)w / (float)h;
	myGraphics.proj_matrix = glm::perspective(glm::radians(50.0f), myGraphics.aspect, 0.1f, 1000.0f);
}

void onKeyCallback(GLFWwindow* window, int key, int scancode, int action, int mods) { // called everytime a key is
pressed
	if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
		glfwSetWindowShouldClose(window, GLFW_TRUE);
}
```

Emitter.h

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include "Particle.h"

class Emitter {
public:
	Emitter();
	~Emitter();

	void Startup();
	void render();
	void Update(double currentTime, Graphics myGraphics);

	Particle myParticles[250];

};
```

Emitter.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include <GLM/gtx/transform.hpp>

#include "graphics.h"
#include "shapes.h"
#include "Emitter.h"
#include "Particle.h"

Emitter::Emitter() {

};

Emitter::~Emitter() {

};

void Emitter::Startup() {

	for (int i = 0; i < 250; i++) {
		int lifespan = 110 + (rand() % (int)(250 - 110 + 1));
		myParticles[i].CreateParticle(lifespan);
	}

}

void Emitter::render() {
	for (int i = 0; i < 250; i++) {
			myParticles[i].Draw();
		}

}

void Emitter::Update(double currentTime, Graphics myGraphics) {
		for (int i = 0; i < 250; i++) {
			myParticles[i].Update(currentTime, myGraphics);
			if (myParticles[i].timer == 0) {
				myParticles[i].CreateParticle(200);
			}
		}



}
```

2.2. Physically Driven Bouncing Ball

Source.cpp

```
#include <iostream>
#include <vector>
using namespace std;

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include <GLM/gtx/transform.hpp>

#include "graphics.h"
#include "shapes.h"

// FUNCTIONS
void render(double currentTime);
void update(double currentTime);
void startup();
void onResizeCallback(GLFWwindow* window, int w, int h);
void onKeyCallback(GLFWwindow* window, int key, int scancode, int action, int mods);

// VARIABLES
bool                running = true;

Graphics  myGraphics;                 // Runing all the graphics in this object
Sphere              mySphere;
Cube                myCube;

float friction = 0.3f;
float gravity = 0.001f;
float speed = 0.0f;
float initspeed = 0.0f;
bool apply = true;
float top = 2.2f;
float t = 0.001f;                     // Global variable for animation

glm::vec3 sposition = glm::vec3(0.0f, 2.2f, -6.0f);
glm::vec3 cposition = glm::vec3(0.0f, -2.0f, -6.0f);

int main()
{
         int errorGraphics = myGraphics.Init();                  // Launch window and graphics context
         if (errorGraphics) return 0;                            //Close if something went wrong...

         startup();                                                                    // Setup all
necessary information for startup (aka. load texture, shaders, models, etc).

                                                                                                 //
Mixed graphics and update functions - declared in main for simplicity.
         glfwSetWindowSizeCallback(myGraphics.window, onResizeCallback);                   // Set callback for
resize
         glfwSetKeyCallback(myGraphics.window, onKeyCallback);                                   // Set
Callback for keys


                                                                  // MAIN LOOP run until the window is closed
         do {
                  double currentTime = glfwGetTime();            // retrieve timelapse
                  glfwPollEvents();                                                  // poll callbacks
                  update(currentTime);                                               // update (physics, animation,
structures, etc)
                  render(currentTime);                                               // call render function.

                  glfwSwapBuffers(myGraphics.window);            // swap buffers (avoid flickering and tearing)

                  running &= (glfwGetKey(myGraphics.window, GLFW_KEY_ESCAPE) == GLFW_RELEASE);        // exit if
escape key pressed
                  running &= (glfwWindowShouldClose(myGraphics.window) != GL_TRUE);
         } while (running);

         myGraphics.endProgram();                         // Close and clean everything up...

         cout << "\nPress any key to continue...\n";
         cin.ignore(); cin.get(); // delay closing console to read debugging errors.

         return 0;
}

void startup() {
	// Calculate proj_matrix for the first time.
	myGraphics.aspect = (float)myGraphics.windowWidth / (float)myGraphics.windowHeight;
	myGraphics.proj_matrix = glm::perspective(glm::radians(50.0f), myGraphics.aspect, 0.1f, 1000.0f);

	// Load Geometry
	myCube.Load();
	mySphere.Load();
	mySphere.fillColor = glm::vec4(0.4f, 0.0f, 0.4f, 1.0f); // You can change the shape fill colour, line 
colour or linewidth
	myGraphics.SetOptimisations();		// Cull and depth testing
}

void update(double currentTime) {

	//apply used to stop operations on ball after stopped moving
	if (apply == true) {
		if (speed < 0.001 && speed > -0.001) {
			top = sposition[1];
		}
		//code for making the ball collide and bounce
		if ((sposition[1] - 0.49) < (cposition[1] + 0.5) && ((sposition[0] - 0.5 < cposition[0] + 0.5) || 
(sposition[0] + 0.5 < cposition[0] - 0.5)) && ((sposition[2] - 0.5 < cposition[2] + 0.5) || (sposition[2] + 0.5 < 
cposition[2] - 0.5))) {
			float r = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
			float g = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
			float b = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
			mySphere.fillColor = glm::vec4(r, g, b, 1.0f);
			sposition[1] = cposition[1] + 1.0f;
			speed = -(speed - (speed*friction));
			float end = top - sposition[1];
			//makes ball stop bouncing
			if (end < 0.065) {
				apply = false;
			}
		}
		//code to make ball fall
		else {
			initspeed = speed;
			speed = (gravity * currentTime) + initspeed;
		}
		sposition[1] -= speed;
	}

	glm::mat4 mv_matrix_cube =
		glm::translate(cposition) *
		glm::rotate(t, glm::vec3(0.0f, 1.0f, 0.0f)) *
		glm::rotate(t, glm::vec3(1.0f, 0.0f, 0.0f)) *
		glm::mat4(1.0f);
	myCube.mv_matrix = mv_matrix_cube;
	myCube.proj_matrix = myGraphics.proj_matrix;


	// calculate Sphere movement
	glm::mat4 mv_matrix_sphere =
		glm::translate(sposition) *
		glm::rotate(-t, glm::vec3(0.0f, 1.0f, 0.0f)) *
		glm::rotate(-t, glm::vec3(1.0f, 0.0f, 0.0f)) *
		glm::mat4(1.0f);
	mySphere.mv_matrix = mv_matrix_sphere;
	mySphere.proj_matrix = myGraphics.proj_matrix;

}

void render(double currentTime) {
	// Clear viewport - start a new frame.
	myGraphics.ClearViewport();

	// Draw
	mySphere.Draw();
	myCube.Draw();

}

void onResizeCallback(GLFWwindow* window, int w, int h) {	// call everytime the window is resized
	myGraphics.windowWidth = w;
	myGraphics.windowHeight = h;

	myGraphics.aspect = (float)w / (float)h;
	myGraphics.proj_matrix = glm::perspective(glm::radians(50.0f), myGraphics.aspect, 0.1f, 1000.0f);
}

void onKeyCallback(GLFWwindow* window, int key, int scancode, int action, int mods) { // called everytime a key is 
pressed
	if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
		glfwSetWindowShouldClose(window, GLFW_TRUE);
```

```cpp
}

class vec3 {
public:
	vec3();
	vec3(float, float, float);
	bool operator==(vec3 rhs);
	vec3 operator+(vec3 rhs);
	vec3 operator-(vec3 rhs);
	vec3 operator*(vec3 rhs);
	vec3 operator/(vec3 rhs);
	vec3 operator+(float scalar);
	vec3 operator-(float scalar);
	vec3 operator*(float scalar);
	vec3 operator/(float scalar);
	vec3 cross(vec3 rhs);
	float dot(vec3 rhs);
	float length();

	float x;
	float y;
	float z;
};

vec3::vec3() {}

vec3::vec3(float, float, float) {}

bool vec3::operator==(vec3 rhs) {
	return(x == rhs.x && y == rhs.y && z == rhs.z);
}

vec3 vec3::operator+(vec3 rhs) {
	return vec3(x + rhs.x,
		y + rhs.y,
		z + rhs.z);
}

vec3 vec3::operator-(vec3 rhs) {
	return vec3(x - rhs.x,
		y - rhs.y,
		z - rhs.z);
}

vec3 vec3::operator*(vec3 rhs) {
	return vec3(x * rhs.x,
		y * rhs.y,
		z * rhs.z);
}

vec3 vec3::operator/(vec3 rhs) {
	return vec3(x / rhs.x,
		y / rhs.y,
		z / rhs.z);
}

vec3 vec3::operator/(float scalar) {
	return vec3(x / scalar,
		y / scalar,
		z / scalar);
}

vec3 vec3::operator*(float scalar) {
	return vec3(x * scalar,
		y * scalar,
		z * scalar);
}

vec3 vec3::operator+(float scalar) {
	return vec3(x + scalar,
		y + scalar,
		z + scalar);
}

vec3 vec3::operator-(float scalar) {
	return vec3(x - scalar,
		y - scalar,
		z - scalar);
}

float vec3::dot(vec3 rhs) {
	return (x * rhs.x +
		y * rhs.y +
		z * rhs.z);
```

```
}

vec3 vec3::cross(vec3 rhs) {
	return vec3(y * rhs.z - z * rhs.y,
		z * rhs.x - x * rhs.z,
		x * rhs.y - y * rhs.x);
}

float vec3::length() {
	return float(sqrt(x*x + y*y + z*z));
}
```

Graphics.cpp

```
#include <iostream>
#include <vector>
using namespace std;

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include <GLM/gtx/transform.hpp>

#include "Graphics.h"

Graphics::Graphics() {

};

Graphics::~Graphics() {

};

int Graphics::Init() {
	if (!glfwInit()) {                                                          // Checking for GLFW
		cout << "Could not initialise GLFW...";
		return 1;
	}

	glfwSetErrorCallback(ErrorCallbackGLFW);       // Setup a function to catch and display all GLFW errors.

	hintsGLFW();                                                                        // Setup glfw with
various hints.

                                                                                                              //
Start a window using GLFW
	string title = "My OpenGL Application";
	window = glfwCreateWindow(windowWidth, windowHeight, title.c_str(), NULL, NULL);
	if (!window) {                                                                      // Window or OpenGL
context creation failed
		cout << "Could not initialise GLFW...";
		endProgram();
		return 1;
	}

	glfwMakeContextCurrent(window);                                     // making the OpenGL context current

                                                                                                              //
Start GLEW (note: always initialise GLEW after creating your window context.)
	glewExperimental = GL_TRUE;                                         // hack: catching them all - forcing
newest debug callback (glDebugMessageCallback)
	GLenum errGLEW = glewInit();
	if (GLEW_OK != errGLEW) {                                           // Problems starting GLEW?
		cout << "Could not initialise GLEW...";
		endProgram();
		return 1;
	}

	SetupRender();

	return 0;
}

void Graphics::hintsGLFW() {
	glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);                     // Create context in debug mode -
for debug message callback
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
}

void ErrorCallbackGLFW(int error, const char* description) {
	cout << "Error GLFW: " << description << "\n";
}
```

```
void Graphics::endProgram() {
	glfwMakeContextCurrent(window);		// destroys window handler
	glfwTerminate();   // destroys all windows and releases resources.
}

void Graphics::SetupRender() {
	glfwSwapInterval(1);		// Ony render when synced (V SYNC)

	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	glfwWindowHint(GLFW_SAMPLES, 0);
	glfwWindowHint(GLFW_STEREO, GL_FALSE);
}

void Graphics::SetOptimisations() {
	glEnable(GL_CULL_FACE);
	glFrontFace(GL_CCW);

	glEnable(GL_DEPTH_TEST);
	glDepthFunc(GL_LEQUAL);
}

void Graphics::ClearViewport() {
	glViewport(0, 0, windowWidth, windowHeight);
	static const GLfloat silver[] = { 0.9f, 0.9f, 0.9f, 1.0f };
	glClearBufferfv(GL_COLOR, 0, silver);
	static const GLfloat one = 1.0f;
	glClearBufferfv(GL_DEPTH, 0, &one);
}
```

Graphics.h

```
#pragma once

#include <iostream>
#include <vector>
using namespace std;

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include <GLM/gtx/transform.hpp>

void ErrorCallbackGLFW(int error, const char* description);

class Graphics {
public:
	Graphics();
	~Graphics();

	int Init();
	void hintsGLFW();
	void SetupRender();
	void endProgram();
	void SetOptimisations();
	void ClearViewport();


	GLFWwindow*			window;
	int					windowWidth = 640;
	int					windowHeight = 480;
	float		aspect;
	glm::mat4		proj_matrix = glm::mat4(1.0f);

};
```

Shapes.cpp

```
#include "shapes.h"

#include <iostream>

#include <sstream>



#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include <GLM/glm.hpp>
```

```
Shapes::Shapes() {

};

Shapes::~Shapes() {

}

void Shapes::LoadObj() {

        std::vector< glm::vec3 > obj_vertices;
        std::vector< unsigned int > vertexIndices;
        istringstream rawDataStream(rawData);
        string dataLine;  int linesDone = 0;

        while (std::getline(rawDataStream, dataLine)) {
                if (dataLine.find("v ") != string::npos) {// does this line have a vector?
                        glm::vec3 vertex;

                        int foundStart = dataLine.find(" ");  int foundEnd = dataLine.find(" ", foundStart + 1);
                        vertex.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        vertex.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        vertex.z = stof(dataLine.substr(foundStart, foundEnd - foundStart));

                        obj_vertices.push_back(vertex);
                }
                else if (dataLine.find("f ") != string::npos) { // does this line defines a triangle face?
                        string parts[3];

                        int foundStart = dataLine.find(" ");  int foundEnd = dataLine.find(" ", foundStart + 1);
                        parts[0] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        parts[1] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        parts[2] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

                        for (int i = 0; i < 3; i++) {                   // for each part

                                vertexIndices.push_back(stoul(parts[i].substr(0, parts[i].find("/"))));

                                int firstSlash = parts[i].find("/"); int secondSlash = parts[i].find("/", firstSlash + 1);

                                if (firstSlash != (secondSlash + 1)) {      // there is texture coordinates.
                // add code for my texture coordintes here.
                                }
                        }
                }

                linesDone++;
        }

        for (unsigned int i = 0; i < vertexIndices.size(); i += 3) {
                vertexPositions.push_back(obj_vertices[vertexIndices[i + 0] - 1].x);
                vertexPositions.push_back(obj_vertices[vertexIndices[i + 0] - 1].y);
                vertexPositions.push_back(obj_vertices[vertexIndices[i + 0] - 1].z);

                vertexPositions.push_back(obj_vertices[vertexIndices[i + 1] - 1].x);
                vertexPositions.push_back(obj_vertices[vertexIndices[i + 1] - 1].y);
                vertexPositions.push_back(obj_vertices[vertexIndices[i + 1] - 1].z);

                vertexPositions.push_back(obj_vertices[vertexIndices[i + 2] - 1].x);
                vertexPositions.push_back(obj_vertices[vertexIndices[i + 2] - 1].y);
                vertexPositions.push_back(obj_vertices[vertexIndices[i + 2] - 1].z);
        }
}


void Shapes::Load() {
        static const char * vs_source[] = { R"(
#version 330 core

in vec4 position;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
```

```
void main(void){
		gl_Position = proj_matrix * mv_matrix * position;
}
)" };

		static const char * fs_source[] = { R"(
#version 330 core

uniform vec4 inColor;
out vec4 color;

void main(void){
		color = inColor;
}
)" };

		program = glCreateProgram();
		GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
		glShaderSource(fs, 1, fs_source, NULL);
		glCompileShader(fs);
		checkErrorShader(fs);

		GLuint vs = glCreateShader(GL_VERTEX_SHADER);
		glShaderSource(vs, 1, vs_source, NULL);
		glCompileShader(vs);
		checkErrorShader(vs);

		glAttachShader(program, vs);
		glAttachShader(program, fs);

		glLinkProgram(program);

		mv_location = glGetUniformLocation(program, "mv_matrix");
		proj_location = glGetUniformLocation(program, "proj_matrix");
		color_location = glGetUniformLocation(program, "inColor");

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

		glGenBuffers(1, &buffer);
```

```
	glBindBuffer(GL_ARRAY_BUFFER, buffer);
	glBufferData(GL_ARRAY_BUFFER,
		vertexPositions.size() * sizeof(GLfloat),
		&vertexPositions[0],
		GL_STATIC_DRAW);
	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
	glEnableVertexAttribArray(0);

	glLinkProgram(0);		// unlink
	glDisableVertexAttribArray(0); // Disable
	glBindVertexArray(0);	// Unbind
}

void Shapes::Draw() {
	glUseProgram(program);
	glBindVertexArray(vao);
	glEnableVertexAttribArray(0);

	glUniformMatrix4fv(proj_location, 1, GL_FALSE, &proj_matrix[0][0]);
	glUniformMatrix4fv(mv_location, 1, GL_FALSE, &mv_matrix[0][0]);

	glUniform4f(color_location, fillColor.r, fillColor.g, fillColor.b, fillColor.a);
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
	glDrawArrays(GL_TRIANGLES, 0, vertexPositions.size() / 3);

	glUniform4f(color_location, lineColor.r, lineColor.g, lineColor.b, lineColor.a);
	glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  glLineWidth(lineWidth);
	glDrawArrays(GL_TRIANGLES, 0, vertexPositions.size() / 3);
}


void Shapes::checkErrorShader(GLuint shader) {
	// Get log length
	GLint maxLength;
	glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &maxLength);

	// Init a string for it
	std::vector<GLchar> errorLog(maxLength);

	if (maxLength > 1) {
		// Get the log file
```

```
				glGetShaderInfoLog(shader, maxLength, &maxLength, &errorLog[0]);

				cout << "--------------Shader compilation error-------------\n";
				cout << errorLog.data();
		}
}

Cube::Cube() {
		// Exported from Blender a cube by default (OBJ File)
		rawData = R"(
v 0.500000 -0.500000 -0.500000
v 0.500000 -0.500000 0.500000
v -0.500000 -0.500000 0.500000
v -0.500000 -0.500000 -0.500000
v 0.500000 0.500000 -0.499999
v 0.499999 0.500000 0.500000
v -0.500000 0.500000 0.500000
v -0.500000 0.500000 -0.500000
f 1 3 4
f 8 6 5
f 5 2 1
f 6 3 2
f 7 4 3
f 1 8 5
f 1 2 3
f 8 7 6
f 5 6 2
f 6 7 3
f 7 8 4
f 1 4 8)";

		LoadObj();
}

Cube::~Cube() {

}

Sphere::Sphere() {

		rawData = R"(
o Sphere

v -0.097545 0.490393 0.000000

v -0.277785 0.415735 0.000000

v -0.415735 0.277785 0.000000

v -0.490393 0.097545 0.000000

v -0.490393 -0.097545 0.000000

v -0.415735 -0.277785 0.000000

v -0.277785 -0.415735 0.000000

v -0.097545 -0.490393 0.000000

v -0.090120 0.490393 -0.037329

v -0.256640 0.415735 -0.106304

v -0.384089 0.277785 -0.159095

v -0.453064 0.097545 -0.187665

v -0.453064 -0.097545 -0.187665

v -0.384089 -0.277785 -0.159095

v -0.256640 -0.415735 -0.106304

v -0.090120 -0.490393 -0.037329

v -0.068975 0.490393 -0.068975

v -0.196424 0.415735 -0.196424

v -0.293969 0.277785 -0.293969

v -0.346760 0.097545 -0.346760

v -0.346760 -0.097545 -0.346760

v -0.293969 -0.277785 -0.293969

v -0.196424 -0.415735 -0.196424

v -0.068975 -0.490393 -0.068975

v -0.037329 0.490393 -0.090120

v -0.106304 0.415735 -0.256640

v -0.159095 0.277785 -0.384089

v -0.187665 0.097545 -0.453064

v -0.187665 -0.097545 -0.453064

v -0.159095 -0.277785 -0.384089

v -0.106304 -0.415735 -0.256640

v -0.037329 -0.490393 -0.090120

v 0.000000 0.490393 -0.097545

v 0.000000 0.415735 -0.277785

v 0.000000 0.277785 -0.415735

v 0.000000 0.097545 -0.490393

v 0.000000 -0.097545 -0.490393

v 0.000000 -0.277785 -0.415735

v 0.000000 -0.415735 -0.277785

v 0.000000 -0.490393 -0.097545
```

```
v 0.037329 0.490393 -0.090120

v 0.106304 0.415735 -0.256640

v 0.159095 0.277785 -0.384089

v 0.187665 0.097545 -0.453064

v 0.187665 -0.097545 -0.453064

v 0.159095 -0.277785 -0.384089

v 0.106304 -0.415735 -0.256640

v 0.037329 -0.490393 -0.090120

v 0.068975 0.490393 -0.068975

v 0.196424 0.415735 -0.196424

v 0.293969 0.277785 -0.293969

v 0.346760 0.097545 -0.346760

v 0.346760 -0.097545 -0.346760

v 0.293969 -0.277785 -0.293969

v 0.196424 -0.415735 -0.196424

v 0.068975 -0.490393 -0.068975

v 0.090120 0.490393 -0.037329

v 0.256640 0.415735 -0.106304

v 0.384089 0.277785 -0.159095

v 0.453064 0.097545 -0.187665

v 0.453064 -0.097545 -0.187665

v 0.384089 -0.277785 -0.159095

v 0.256640 -0.415735 -0.106304

v 0.090120 -0.490393 -0.037329

v 0.097545 0.490393 0.000000

v 0.277785 0.415735 -0.000000

v 0.415735 0.277785 0.000000

v 0.490393 0.097545 0.000000

v 0.490393 -0.097545 0.000000

v 0.415735 -0.277785 0.000000

v 0.277785 -0.415735 0.000000

v 0.097545 -0.490393 -0.000000

v 0.090120 0.490393 0.037329

v 0.256640 0.415735 0.106304

v 0.384089 0.277785 0.159095

v 0.453064 0.097545 0.187665

v 0.453064 -0.097545 0.187665

v 0.384089 -0.277785 0.159095

v 0.256640 -0.415735 0.106304

v 0.090120 -0.490393 0.037329

v 0.068975 0.490393 0.068975
```

```
v 0.196424 0.415735 0.196424

v 0.293969 0.277785 0.293969

v 0.346760 0.097545 0.346760

v 0.346760 -0.097545 0.346760

v 0.293969 -0.277785 0.293969

v 0.196424 -0.415735 0.196424

v 0.068975 -0.490393 0.068975

v 0.000000 -0.500000 0.000000

v 0.037329 0.490393 0.090120

v 0.106304 0.415735 0.256640

v 0.159095 0.277785 0.384089

v 0.187665 0.097545 0.453064

v 0.187665 -0.097545 0.453064

v 0.159095 -0.277785 0.384089

v 0.106304 -0.415735 0.256640

v 0.037329 -0.490393 0.090120

v 0.000000 0.490393 0.097545

v 0.000000 0.415735 0.277785

v 0.000000 0.277785 0.415735

v 0.000000 0.097545 0.490392

v 0.000000 -0.097545 0.490392

v 0.000000 -0.277785 0.415735

v 0.000000 -0.415735 0.277785

v 0.000000 -0.490393 0.097545

v -0.037329 0.490393 0.090120

v -0.106304 0.415735 0.256640

v -0.159095 0.277785 0.384089

v -0.187665 0.097545 0.453063

v -0.187665 -0.097545 0.453063

v -0.159095 -0.277785 0.384089

v -0.106304 -0.415735 0.256640

v -0.037329 -0.490393 0.090120

v -0.068975 0.490393 0.068975

v -0.196424 0.415735 0.196424

v -0.293969 0.277785 0.293969

v -0.346760 0.097545 0.346760

v -0.346760 -0.097545 0.346760

v -0.293969 -0.277785 0.293969

v -0.196423 -0.415735 0.196424

v -0.068975 -0.490393 0.068975

v 0.000000 0.500000 0.000000
```

```
v -0.090120 0.490393 0.037329
v -0.256640 0.415735 0.106304
v -0.384088 0.277785 0.159095
v -0.453063 0.097545 0.187665
v -0.453063 -0.097545 0.187665
v -0.384088 -0.277785 0.159095
v -0.256640 -0.415735 0.106304
v -0.090120 -0.490393 0.037329
s off
f 7 14 15
f 3 10 11
f 12 3 11
f 8 15 16
f 5 12 13
f 2 125 124
f 2 9 10
f 6 13 14
f 89 8 16
f 122 17 9
f 7 128 6
f 20 27 28
f 8 129 7
f 22 29 30
f 19 26 27
f 29 36 37
f 31 22 30
f 89 16 24
f 26 33 34
f 24 31 32
f 28 35 36
f 122 25 17
f 27 34 35
f 37 44 45
f 38 29 37
f 89 24 32
f 42 33 41
f 32 39 40
f 36 43 44
f 31 38 39
f 122 33 25
f 43 34 42
```

```
f 45 52 53

f 46 37 45

f 89 32 40

f 43 50 51

f 48 39 47

f 52 43 51

f 39 46 47

f 50 41 49

f 122 41 33

f 53 60 61

f 47 54 55

f 46 53 54

f 48 55 56

f 60 51 59

f 58 49 57

f 122 49 41

f 89 40 48

f 61 68 69

f 55 62 63

f 54 61 62

f 51 58 59

f 58 65 66

f 68 59 67

f 122 57 49

f 56 63 64

f 89 48 56

f 63 70 71

f 62 69 70

f 59 66 67

f 69 76 77

f 66 73 74

f 122 65 57

f 64 71 72

f 76 67 75

f 89 56 64

f 79 70 78

f 70 77 78

f 67 74 75

f 77 84 85

f 72 79 80

f 122 73 65
```

```
f 76 83 84

f 89 64 72

f 74 81 82

f 87 78 86

f 86 77 85

f 75 82 83

f 85 93 94

f 80 87 88

f 84 92 93

f 122 81 73

f 89 72 80

f 91 81 90

f 87 95 96

f 86 94 95

f 83 91 92

f 94 101 102

f 93 100 101

f 89 80 88

f 122 90 81

f 91 98 99

f 88 96 97

f 95 102 103

f 92 99 100

f 102 109 110

f 96 103 104

f 122 98 90

f 89 88 97

f 99 106 107

f 105 96 104

f 109 100 108

f 108 99 107

f 110 117 118

f 104 111 112

f 122 106 98

f 89 97 105

f 107 114 115

f 103 110 111

f 117 108 116

f 113 104 112

f 108 115 116

f 118 126 127
```

```
f 120 111 119

f 122 114 106

f 115 123 124

f 111 118 119

f 89 105 113

f 113 120 121

f 126 116 125

f 119 127 128

f 116 124 125

f 120 128 129

f 89 113 121

f 121 129 130

f 122 123 114

f 89 121 130

f 122 1 123

f 89 130 8

f 3 126 125

f 5 126 4

f 15 22 23

f 10 17 18

f 24 15 23

f 13 20 21

f 18 25 26

f 14 21 22

f 21 28 29

f 12 19 20

f 11 18 19

f 1 124 123

f 122 9 1

f 6 127 5

f 7 6 14

f 3 2 10

f 12 4 3

f 8 7 15

f 5 4 12

f 2 3 125

f 2 1 9

f 6 5 13

f 7 129 128

f 20 19 27

f 8 130 129
```

```
f 22 21 29

f 19 18 26

f 29 28 36

f 31 23 22

f 26 25 33

f 24 23 31

f 28 27 35

f 27 26 34

f 37 36 44

f 38 30 29

f 42 34 33

f 32 31 39

f 36 35 43

f 31 30 38

f 43 35 34

f 45 44 52

f 46 38 37

f 43 42 50

f 48 40 39

f 52 44 43

f 39 38 46

f 50 42 41

f 53 52 60

f 47 46 54

f 46 45 53

f 48 47 55

f 60 52 51

f 58 50 49

f 61 60 68

f 55 54 62

f 54 53 61

f 51 50 58

f 58 57 65

f 68 60 59

f 56 55 63

f 63 62 70

f 62 61 69

f 59 58 66

f 69 68 76

f 66 65 73

f 64 63 71
```

```
f 76 68 67

f 79 71 70

f 70 69 77

f 67 66 74

f 77 76 84

f 72 71 79

f 76 75 83

f 74 73 81

f 87 79 78

f 86 78 77

f 75 74 82

f 85 84 93

f 80 79 87

f 84 83 92

f 91 82 81

f 87 86 95

f 86 85 94

f 83 82 91

f 94 93 101

f 93 92 100

f 91 90 98

f 88 87 96

f 95 94 102

f 92 91 99

f 102 101 109

f 96 95 103

f 99 98 106

f 105 97 96

f 109 101 100

f 108 100 99

f 110 109 117

f 104 103 111

f 107 106 114

f 103 102 110

f 117 109 108

f 113 105 104

f 108 107 115

f 118 117 126

f 120 112 111

f 115 114 123

f 111 110 118
```

f 113 112 120

f 126 117 116

f 119 118 127

f 116 115 124

f 120 119 128

f 121 120 129

f 3 4 126

f 5 127 126

f 15 14 22

f 10 9 17

f 24 16 15

f 13 12 20

f 18 17 25

f 14 13 21

f 21 20 28

f 12 11 19

f 11 10 18

f 1 2 124

f 6 128 127

)";

        LoadObj();

}

Sphere::~Sphere() {

}

Shapes.h

```
#pragma once

#include <iostream>
#include <vector>
using namespace std;

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>

class Shapes {

public:
         Shapes();
         ~Shapes();

         void Load();
         void Draw();
         void  checkErrorShader(GLuint shader);

         vector<GLfloat> vertexPositions;

         GLuint          program;
         GLuint          vao;
```

```
        GLuint          buffer;
        GLint           mv_location;
        GLint           proj_location;
        GLint           color_location;
        glm::mat4          proj_matrix = glm::mat4(1.0f);
        glm::mat4          mv_matrix = glm::mat4(1.0f);

        glm::vec4 fillColor = glm::vec4(1.0, 0.0, 0.0, 1.0);
        glm::vec4 lineColor = glm::vec4(0.0, 0.0, 0.0, 1.0);
        float lineWidth = 2.0f;

protected:
        string rawData;                        // Import obj file from Blender (note: no textures or UVs).
        void LoadObj();
};

class Sphere : public Shapes {
public:
        Sphere();
        ~Sphere();
};

class Cube : public Shapes {
public:
        Cube();
        ~Cube();
};
```

2.3. Flocking Boids in 3D

Not completed.

2.4. A* Planning search

Uses same shapes.h ,graphics.h, graphics.cpp as the bouncing ball demo.

Source.cpp

```
#include <iostream>
#include <vector>
using namespace std;

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>
#include <GLM/gtx/transform.hpp>

#include "graphics.h"
#include "shapes.h"

// FUNCTIONS
void render(double currentTime);
void update(double currentTime);
void startup();
void onResizeCallback(GLFWwindow* window, int w, int h);
void onKeyCallback(GLFWwindow* window, int key, int scancode, int action, int mods);

bool blocked(int x, int y);
//void up(int x, int y, int count);
//void down(int x, int y, int count);
//void left(int x, int y, int count);
//void right(int x, int y, int count);

void findFastestRoute(int spx, int spy, int epx, int epy);

glm::vec2 visited[400];
glm::vec2 possible[3];
glm::vec2 impossible[400];
glm::vec2 previous;
glm::vec2 dir[400];
int xdif;
int ydif;
glm::vec2 curpos;
bool db = false;
bool ub = false;
bool lb = false;
bool rb = false;
bool stuck = false;
int s = 0;


// VARIABLES
bool                running = true;

Graphics  myGraphics;                 // Runing all the graphics in this object
                                                            //Cube             myCube;
Cube                test3;

int INSTANCE_COUNT = 400;
const int oMax = 80;
int obstacle = 80;
int startposx;
int startposy;
int endposx;
int endposy;
int obstpla[oMax];
float t = 0.001f;                   // Global variable for animation


int main()
{
         //Code for entering starting positions---------------------------------------------------------------------
-----------------------------------------
         cout << "Please enter a starting x and y position between (0,0) and (19,19)" << "\n";
         cin >> startposx;
         if (startposx <= 19 && startposx >= 0) {}
         else {
                  while (startposx > 19 || startposx < 0) {
                           cout << "Please enter a starting x Coordinate that is between 0 and 19" << "\n";
                           cin >> startposx;
                  }
                  cout << "Please enter a starting y coordinate between 0 and 19" << "\n";
         }
         cin >> startposy;
         if (startposy <= 19 && startposy >= 0) {}
```

```
	else {
		while (startposy > 19 || startposy < 0) {
			cout << "Please enter a starting y Coordinate that is between 0 and 19" << "\n";
			cin >> startposy;
		}
	}
	//Code for entering starting positions---------------------------------------------------------------------
-----------------------------------------
	cout << "Please enter an ending x and y position between (0,0) and (19,19)" << "\n";
	cin >> endposx;
	if (endposx <= 19 && endposx >= 0) {}
	else {
		while (endposx > 19 || endposx < 0) {
			cout << "Please enter an ending x Coordinate that is between 0 and 19" << "\n";
			cin >> endposx;
		}
		cout << "Please enter an ending y coordinate between 0 and 19" << "\n";
	}
	cin >> endposy;
	if (endposy <= 19 && endposy >= 0) {}
	else {
		while (endposy > 19 || endposy < 0) {
			cout << "Please enter a ending y Coordinate that is between 0 and 19" << "\n";
			cin >> endposy;
		}
	}

	if (startposy == endposy && startposx == endposx) {
		while ((startposy == endposy && startposx == endposx) || (endposx > 19 || endposx < 0) || (endposy
> 19 || endposy < 0) || (startposx > 19 || startposx < 0) || (startposy > 19 || startposy < 0)) {
			cout << "Please enter an ending x and y Coordinate between 0 and 19 that is different to
the starting position" << "\n";
			cin >> endposx;
			cin >> endposy;
		}
	}


	int errorGraphics = myGraphics.Init();			// Launch window and graphics context
	if (errorGraphics) return 0;				//Close if something went wrong...

	startup();								// Setup all
necessary information for startup (aka. load texture, shaders, models, etc).

										//
Mixed graphics and update functions - declared in main for simplicity.
	glfwSetWindowSizeCallback(myGraphics.window, onResizeCallback);			// Set callback for
resize
	glfwSetKeyCallback(myGraphics.window, onKeyCallback);					// Set
Callback for keys


							// MAIN LOOP run until the window is closed
	do {
		double currentTime = glfwGetTime();		// retrieve timelapse
		glfwPollEvents();						// poll callbacks
		update(currentTime);					// update (physics, animation,
structures, etc)
		render(currentTime);					// call render function.

		glfwSwapBuffers(myGraphics.window);		// swap buffers (avoid flickering and tearing)

		running &= (glfwGetKey(myGraphics.window, GLFW_KEY_ESCAPE) == GLFW_RELEASE);	// exit if
escape key pressed
		running &= (glfwWindowShouldClose(myGraphics.window) != GL_TRUE);
		findFastestRoute(startposx, startposy, endposx, endposy);

	} while (running);

	myGraphics.endProgram();			// Close and clean everything up...

	cout << "\nPress any key to continue...\n";
	cin.ignore(); cin.get(); // delay closing console to read debugging errors.

	return 0;
}

void startup() {

	// Calculate proj_matrix for the first time.
	myGraphics.aspect = (float)myGraphics.windowWidth / (float)myGraphics.windowHeight;
	myGraphics.proj_matrix = glm::perspective(glm::radians(50.0f), myGraphics.aspect, 0.1f, 1000.0f);

	// Load Geometry
	test3.Load();
```

```
		test3.fillColor = glm::vec4(0.0f, 1.0f, 0.0f, 1.0f);
		test3.lineColor = glm::vec4(1.0f, 1.0f, 1.0f, 1.0f);
		if (obstacle <= 0) {}
		else {
			for (int j = 0; j < oMax; j++) {
				obstpla[j] = rand() % INSTANCE_COUNT;
				obstacle--;
			}
		}


		//myCube.Load();
		//myCube.fillColor = glm::vec4(0.0f, 1.0f, 0.0f, 1.0f);
		//myCube.lineColor = glm::vec4(0.0f, 1.0f, 0.0f, 1.0f);
		myGraphics.SetOptimisations();			// Cull and depth testing

	}

	void update(double currentTime) {
		/*
		glm::mat4 mv_matrix_cube =
		glm::translate(glm::vec3(-1.0f, -1.0f, -6.0f)) *
		glm::rotate(t, glm::vec3(0.0f, 0.0f, 1.0f)) *
		glm::rotate(t, glm::vec3(0.0f, 1.0f, 0.0f)) *
		glm::mat4(1.0f);
		myCube.mv_matrix = mv_matrix_cube;
		myCube.proj_matrix = myGraphics.proj_matrix;*/
	}

	void render(double currentTime) {
		// Clear viewport - start a new frame.
		myGraphics.ClearViewport();


		for (int i = 0; i < 20; i++) {
			for (int n = 0; n < 20; n++) {
				test3.fillColor = glm::vec4(0.0f, 1.0f, 0.0f, 1.0f);
				for (int o = 0; o < oMax; o++) {
					if ((i == startposy && n == startposx) || (i == endposy && n == endposx)) {
						test3.fillColor = glm::vec4(1.0f, 0.0f, 0.0f, 1.0f);
					}
					else if ((20 * n) + i == obstpla[o]) {
						test3.fillColor = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);
					}
				}
				glm::mat4 mv_matrix_cube3 =
					glm::translate(glm::vec3((n*0.2f) - 2.0f, (i*0.2f) - 2.0f, -6.0f)) *
					glm::rotate(t, glm::vec3(0.0f, 0.0f, 1.0f)) *
					glm::rotate(t, glm::vec3(0.0f, 1.0f, 0.0f)) *
					glm::mat4(1.0f);
				test3.mv_matrix = mv_matrix_cube3;
				test3.proj_matrix = myGraphics.proj_matrix;
				test3.Draw();
			}
		}

		// Draw
		//myCube.Draw();
		//test.Draw();
		//glUnmapBuffer(GL_ARRAY_BUFFER);

	}

	void onResizeCallback(GLFWwindow* window, int w, int h) {	// call everytime the window is resized
		myGraphics.windowWidth = w;
		myGraphics.windowHeight = h;

		myGraphics.aspect = (float)w / (float)h;
		myGraphics.proj_matrix = glm::perspective(glm::radians(50.0f), myGraphics.aspect, 0.1f, 1000.0f);
	}

	void onKeyCallback(GLFWwindow* window, int key, int scancode, int action, int mods) { // called everytime a key is
	pressed
		if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
			glfwSetWindowShouldClose(window, GLFW_TRUE);

	}

	void findFastestRoute(int spx, int spy, int epx, int epy) {
		glm::vec2 endpos = glm::vec2(epx, epy);
		curpos = glm::vec2(spx, spy);
		dir[0] = glm::vec2(0,0);
		visited[0] = curpos;
		int i = 1;
```

```
    //loops until it finds the end
    while (curpos != endpos) {
        cout << "x " << curpos[0] << " y " << curpos[1] << "\n";
        //initialises the possible grid spaces to impossible values
        for (int j = 0; j < 4; j++) {
            possible[j] = glm::vec2(-6, -6);
        }
        //find difference between current position to end position
        xdif = curpos[0] - endpos[0];
        ydif = curpos[1] - endpos[1];

        //finds which paths are blocked
        ub = blocked(curpos[0], curpos[1] + 1);
        db = blocked(curpos[0], curpos[1] - 1);
        rb = blocked(curpos[0] + 1, curpos[1]);
        lb = blocked(curpos[0] - 1, curpos[1]);

        //if paths are not blocked add them to possible movements
        if (ub != true) {
            possible[0] = glm::vec2(curpos[0], curpos[1] + 1);
        }
        if (db != true) {
            possible[1] = glm::vec2(curpos[0], curpos[1] - 1);
        }
        if (lb != true) {
            possible[2] = glm::vec2(curpos[0] - 1, curpos[1]);
        }
        if (rb != true) {
            possible[3] = glm::vec2(curpos[0] + 1, curpos[1]);
        }

        //if current position is the same as previous then you are stuck and need to figure out a way out
        for (int j = 0; j < 4; j++) {
            if (curpos == possible[j]) {
                stuck = true;
            }
            else if (possible[j] == previous) {
                impossible[s] = previous;
                s++;
            }
            cout << "start" << previous[0] << previous[1] << " " << possible[j][0] << possible[j][1]
<< "\n";
            cout << "impossible" << impossible[s - 1][0] << impossible[s - 1][1] << "\n";
        }

        //loops through all possible movements to find best movement
        for (int j = 0; j < 4; j++) {
            if (possible[j] != glm::vec2(-6, -6)) {
                glm::vec2 tempdif = possible[i] - (endpos[0], endpos[1]);
                //if the curpos is stuck then will add it to impossible array to avoid going
back there
                if (stuck == true) {
                    impossible[s] = curpos;
                    s++;
                    stuck = false;
                }
                //if a possible action is impossible then dont go to it
                for (int k = 0; k <= s; k++) {
                    if (possible[j] == impossible[k]) {
                        possible[j] = glm::vec2(-6, -6);
                    }
                }
                //tries to take the best action available
                if (tempdif[1] < ydif) {
                    previous = curpos;
                    curpos = possible[j];
                    dir[i] = tempdif;
                    visited[i] = curpos;
                    i++;
                    break;
                }else if (tempdif[1] < ydif) {
                    previous = curpos;
                    curpos = possible[j];
                    dir[i] = tempdif;
                    visited[i] = curpos;
                    i++;
                    break;
                }
                //takes another movement if the best action is not available
                else if (tempdif[0] > xdif) {
                    previous = curpos;
                    curpos = possible[j];
                    dir[i] = tempdif;
                    visited[i] = curpos;
                    i++;
```

```
                                        break;
                                }
                                else if (tempdif[1] > ydif) {
                                        previous = curpos;
                                        curpos = possible[j];
                                        dir[i] = tempdif;
                                        visited[i] = curpos;
                                        i++;
                                        break;
                                }
                        }
                }

                if (xdif == 0 && ydif == 0) {
                        cout << "yes";
                        break;
                }
        }
        cout << "x " << curpos[0] << " y " << curpos[1] << "\n";

        }

bool blocked(int x, int y) {
        for (int i = 0; i < oMax; i++) {
                if ((20 * y) + x == obstpla[i]) {
                        return true;
                }
        }
        return false;
}
```

Shapes.cpp

```
#include "shapes.h"
#include <iostream>
#include <sstream>

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GLM/glm.hpp>

Shapes::Shapes() {

};

Shapes::~Shapes() {

}

void Shapes::LoadObj() {

        std::vector< glm::vec3 > obj_vertices;
        std::vector< unsigned int > vertexIndices;
        istringstream rawDataStream(rawData);
        string dataLine;  int linesDone = 0;

        while (std::getline(rawDataStream, dataLine)) {
                if (dataLine.find("v ") != string::npos) {     // does this line have a vector?
                        glm::vec3 vertex;

                        int foundStart = dataLine.find(" ");  int foundEnd = dataLine.find(" ", foundStart + 1);
                        vertex.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        vertex.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        vertex.z = stof(dataLine.substr(foundStart, foundEnd - foundStart));

                        obj_vertices.push_back(vertex);
                }
                else if (dataLine.find("f ") != string::npos) { // does this line defines a triangle face?
                        string parts[3];

                        int foundStart = dataLine.find(" ");  int foundEnd = dataLine.find(" ", foundStart + 1);
                        parts[0] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        parts[1] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

                        foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
                        parts[2] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

                        for (int i = 0; i < 3; i++) {                  // for each part
```

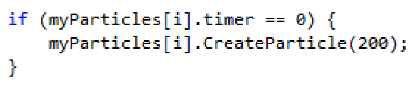
```
                                        vertexIndices.push_back(stoul(parts[i].substr(0, parts[i].find("/"))));

                                        int firstSlash = parts[i].find("/"); int secondSlash = parts[i].find("/",
firstSlash + 1);

                                        if (firstSlash != (secondSlash + 1)) {         // there is texture coordinates.

                    // add code for my texture coordintes here.
                                        }
                              }
                    }

                    linesDone++;
          }

          for (unsigned int i = 0; i < vertexIndices.size(); i += 3) {
                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 0] - 1].x);
                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 0] - 1].y);
                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 0] - 1].z);

                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 1] - 1].x);
                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 1] - 1].y);
                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 1] - 1].z);

                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 2] - 1].x);
                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 2] - 1].y);
                    vertexPositions.push_back(obj_vertices[vertexIndices[i + 2] - 1].z);
          }
}


void Shapes::Load() {
          static const char * vs_source[] = { R"(
#version 330 core

in vec4 position;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void){
          gl_Position = proj_matrix * mv_matrix * position;
}
)" };

          static const char * fs_source[] = { R"(
#version 330 core

uniform vec4 inColor;
out vec4 color;

void main(void){
          color = inColor;
}
)" };

          program = glCreateProgram();
          GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
          glShaderSource(fs, 1, fs_source, NULL);
          glCompileShader(fs);
          checkErrorShader(fs);

          GLuint vs = glCreateShader(GL_VERTEX_SHADER);
          glShaderSource(vs, 1, vs_source, NULL);
          glCompileShader(vs);
          checkErrorShader(vs);

          glAttachShader(program, vs);
          glAttachShader(program, fs);

          glLinkProgram(program);

          mv_location = glGetUniformLocation(program, "mv_matrix");
          proj_location = glGetUniformLocation(program, "proj_matrix");
          color_location = glGetUniformLocation(program, "inColor");

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

          glGenBuffers(1, &buffer);
          glBindBuffer(GL_ARRAY_BUFFER, buffer);
          glBufferData(GL_ARRAY_BUFFER,
                    vertexPositions.size() * sizeof(GLfloat),
                    &vertexPositions[0],
                    GL_STATIC_DRAW);
```

```
	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
	glEnableVertexAttribArray(0);

	glLinkProgram(0);  // unlink
	glDisableVertexAttribArray(0); // Disable
	glBindVertexArray(0);       // Unbind
}

void Shapes::Draw() {
	glUseProgram(program);
	glBindVertexArray(vao);
	glEnableVertexAttribArray(0);

	glUniformMatrix4fv(proj_location, 1, GL_FALSE, &proj_matrix[0][0]);
	glUniformMatrix4fv(mv_location, 1, GL_FALSE, &mv_matrix[0][0]);

	glUniform4f(color_location, fillColor.r, fillColor.g, fillColor.b, fillColor.a);
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
	glDrawArrays(GL_TRIANGLES, 0, vertexPositions.size() / 3);

	glUniform4f(color_location, lineColor.r, lineColor.g, lineColor.b, lineColor.a);
	glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  glLineWidth(lineWidth);
	glDrawArrays(GL_TRIANGLES, 0, vertexPositions.size() / 3);
}


void Shapes::checkErrorShader(GLuint shader) {
	// Get log length
	GLint maxLength;
	glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &maxLength);

	// Init a string for it
	std::vector<GLchar> errorLog(maxLength);

	if (maxLength > 1) {
		// Get the log file
		glGetShaderInfoLog(shader, maxLength, &maxLength, &errorLog[0]);

		cout << "--------------Shader compilation error-------------\n";
		cout << errorLog.data();
	}
}

Cube::Cube() {
	// Exported from Blender a cube by default (OBJ File)
	rawData = R"(
v 0.100000 -0.100000 -0.100000
v 0.100000 -0.100000 0.100000
v -0.100000 -0.100000 0.100000
v -0.100000 -0.100000 -0.100000
v 0.100000 0.100000 -0.099999
v 0.099999 0.100000 0.100000
v -0.100000 0.100000 0.100000
v -0.100000 0.100000 -0.100000
f 1 3 4
f 8 6 5
f 5 2 1
f 6 3 2
f 7 4 3
f 1 8 5
f 1 2 3
f 8 7 6
f 5 6 2
f 6 7 3
f 7 8 4
f 1 4 8)";

	LoadObj();
}

Cube::~Cube() {

}
```

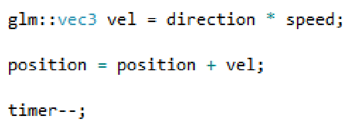
3.1. Statement on each demo

Particle Explosion

To create the particle explosion I used a class and instanced 250 same sized balls with random colours, a speed and a random direction to allow for a colourful animation. The balls were assigned a lifespan based on a timer so that when enough time had passed the balls would start to disappear and then would continue to go back to the starting location to be assigned a new colour and direction and their lifespan would be set back to 200. On the update method the balls would move by using their direction and speed to work out how far to move while also decreaseing the time left on their timer.



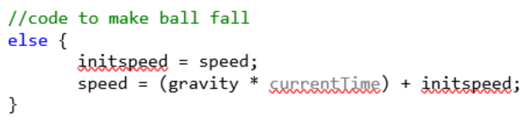
```
if (myParticles[i].timer == 0) {
    myParticles[i].CreateParticle(200);
}
```



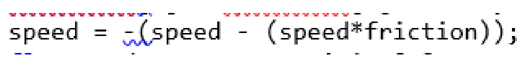
```
glm::vec3 vel = direction * speed;

position = position + vel;

timer--;
```

Bouncing ball

I stored the vertices of the ball and the cube used in this demo in shapes.cpp. To detect the collision, I took the size of the cube to find the edges, and then took the radius of the sphere to create a cube around the ball to detect any collisions. The bouncing ball uses the formula v = gt + u which creates a speed based on the gravity, time and last known speed of the ball. This will allow the ball to pick up speed until it detects a collision where the speed variable is reversed and has a “friction” variable applied to it to make the ball bounce less high each time until it comes to a rest. The speed is taken away from the spheres y coordinate on each update, which results in blocky movement in high speeds but works perfectly in these circumstances. Once the ball has stopped bouncing the physics is not being run on the ball so there are less resources being wasted on the ball when nothing is happening.



```
//code to make ball fall
else {
      initspeed = speed;
      speed = (gravity * currentTime) + initspeed;
}
```



```
speed = -(speed - (speed*friction));
```

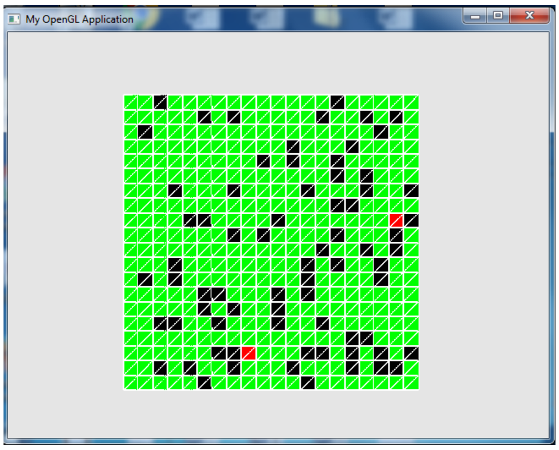
Flocking Boids

Not completed as I ran out of time due to other responsibilities and the difficulty of the task.

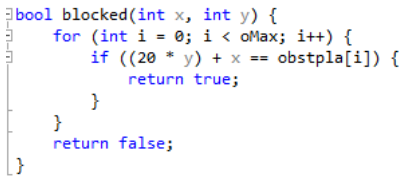
A* Search

To display the 20 x 20 grid for this search I created a grid of green cubes using instancing which is made of smaller version of the same cube used in the bouncing ball example. The start and end points are entered by the user through the command line and are changed to red while the obstacles are randomly selected and changed to black to allow for easy visualisation on the grid. Through the search the cubes will change colour when they have been used as part of the search and the best path will change to a different colour to show the best path available.

4. A* Search walkthrough



The A* search algorithm was incomplete but I have shown the code I managed to create which generates a grid of squares that randomly generate and place obstacles (which are then stored into an array for later use), a user entered start and stop space and the empty space. The code could start to move towards the end location but would get stuck in corners where obstacles are blocking the good directions that move towards the end.



```
bool blocked(int x, int y) {
    for (int i = 0; i < oMax; i++) {
        if ((20 * y) + x == obstpla[i]) {
            return true;
        }
    }
    return false;
}
```

The code would find the unblocked locations around its current location using the method "blocked" and would try to move closer to the point and each successful move would be added to an array of directions for the fastest route. It would find the difference from its current location and the end location and compare this difference to the difference of any possible moves to judge how good a move it is to make which allows for movement in the correct direction. The grid would not allow for diagonal movements and therefore could only move up, down, left or right to use less computational power and be more simplistic.