Portfolio 1 – 3DGE

Pramod Nepal

Contents

Lab 01

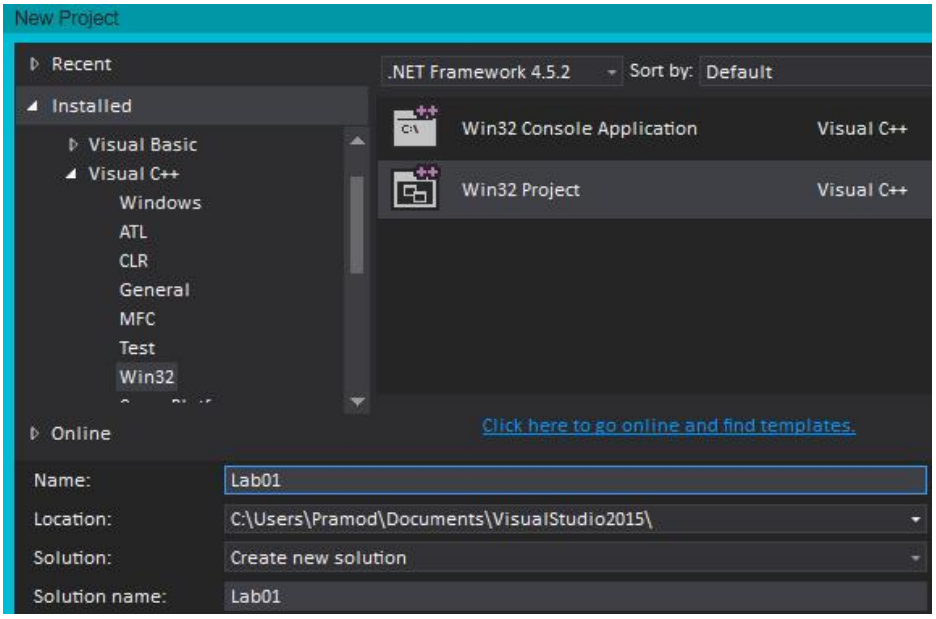
The objective of **Lab 1** was to learn to use Visual Studio for creating games. There were two tasks during the lab. One was to create a graphical window and setup OpenGL context for drawing OpenGL objects. The next task was to create a console window and discover how Visual Studio can be used to create console based games.

Part 1 (OpenGL Window)

The objective of first part of Lab 1 was to learn the setup of Visual Studio for OpenGL context creation. The lab also focused on adding third party libraries to Visual Studio. As an example, every OpenGL project in the lab includes The **OpenGL Extension Wrangler Library (GLEW)**. GLEW provides a way for the code to check at runtime which OpenGL extensions are supported by the target graphics card and link them dynamically. This part of the lab was titled "Getting Started with OpenGL".

To create a window, I first downloaded GLEW Windows 32-binary from http://glew.sourceforge.net. To use GLEW, the project had to know about glew32.dll file. Therefore, the path to bin/Release/Win32 directory of the GLEW library (after extraction) had to be added to the **PATH** environment variable.

To create a new Windows Project in Visual Studio, I used the **File>New>Project** menu option from the menu bar. In the New Project dialog box, I expanded **Templates->Visual C++** and selected **Win32 Project** from **Win32** template option.



In addition to adding glew.dll in the PATH environment variable, the project should also be able to use GLEW object library (.lib) and header (.h) files to compile and link to create the game executable. After creating the Win32 Project I browsed to the project properties. The project properties window can be accessed by right clicking the Project in the **Solution Explorer** and clicking **Properties**. The header and lib locations in the filesystem can be added in this window. I added GLEW header location to the project by adding it to **Include Directories** and the library file by adding it to **Library Directories** text boxes in **Configuration Properties** > **VC Directories**.

Next part of the lab was to add code to create a basic window. For this, I added a c++ file called main.cpp by right clicking the Project (**Lab01**) in the **Solution Explorer** and selecting **Add > New Item**. In the next window, I select the **C++ File(.cpp)** and typed the name **main.cpp** as the name for the file. In this file, I added following code. The code snippet acts as the entry point for the program (main).

```
#pragma comment(lib, "glew32.lib")
#pragma comment(lib, "opengl32.lib")

#include <Windows.h>

#include <gl\glew.h>
#include <gl\wglew.h>

#include <string>
using std::string;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)
{
	WindowStruct window = CreateCustomWindow("Hello, Title!");
	if (window.hWnd != NULL) {
		Context context = CreateOGLWindow(window);
		if (context.hdc != NULL) {
			ShowWindow(window.hWnd);

			glClearColor(0.1f, 0.1f, 0.9f, 0.0f);
			MessageLoop(context);
			CleanUp(window, context);
		}
	}

	return 0;
}
```

Above code uses several custom methods and struct variables. The CreateCustomWindow is a function that creates a window object and sets several attributes to it. Some of such attributes are window position, size and title. This function also registers the callback function called WndProc to listen for Window messages. Messages represent events that direct actions to be taken by the application. The CreateCustomWindow function returns WindowStruct struct object that has a handle for the created window object.

```
struct WindowStruct
{
	HINSTANCE hInstance;
	HWND hWnd;
};

WindowStruct CreateCustomWindow(const string &title)
{
	WindowStruct window;
	window.hInstance = 0;
	window.hWnd = NULL;


	window.hInstance = GetModuleHandle(NULL);
```

```
	if (window.hInstance == 0)
		return window;

	WNDCLASS windowClass;
	windowClass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
	windowClass.lpfnWndProc = (WNDPROC)WndProc;
	windowClass.cbClsExtra = 0;
	windowClass.cbWndExtra = 0;
	windowClass.hInstance = window.hInstance;
	windowClass.hIcon = LoadIcon(NULL, IDI_WINLOGO);
	windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);
	windowClass.hbrBackground = NULL;
	windowClass.lpszMenuName = NULL;
	windowClass.lpszClassName = title.c_str();

	if (!RegisterClass(&windowClass)) {
		window.hInstance = 0;
		window.hWnd = NULL;
		return window;
	}

	window.hWnd = CreateWindowEx(
		WS_EX_APPWINDOW | WS_EX_WINDOWEDGE,
		title.c_str(),
		title.c_str(),
		WS_OVERLAPPEDWINDOW,
		CW_USEDEFAULT, 0, 700, 700,
		NULL,
		NULL,
		window.hInstance,
		NULL
		);

	if (window.hWnd == NULL) {
		window.hInstance = 0;
		window.hWnd = NULL;
	}

	return window;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
	switch (message) {
	case WM_CLOSE:
	case WM_DESTROY:
		PostQuitMessage(0);
		break;
	}

	return DefWindowProc(hWnd, message, wParam, lParam);
}
```

After the window is created and if the window handle is not null (i.e., the window got created), the code creates an OpenGL window. This behavior is highlighted in following code. The function CreateOGLWindow returns a context object that is a struct that has handles for the OpenGL rendering context (HGLRC) and device context (hdc).

```
struct Context {
	HGLRC hrc;
	HDC hdc;
};

Context CreateOGLWindow(const WindowStruct &window)
{
	Context context;
	context.hdc = NULL;
	context.hrc = NULL;

	if (window.hWnd == NULL)
		return context;

	context.hdc = GetDC(window.hWnd);
	if (context.hdc == NULL)
		return context;

	PIXELFORMATDESCRIPTOR pfd;
	memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
	pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
	pfd.dwFlags = PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW;
	pfd.iPixelType = PFD_TYPE_RGBA;
	pfd.cColorBits = 32;
	pfd.cDepthBits = 32;

	int pixelFormat = ChoosePixelFormat(context.hdc, &pfd);
	if (pixelFormat == 0) {
		context.hdc = NULL;
		context.hrc = NULL;
		return context;
	}

	BOOL result = SetPixelFormat(context.hdc, pixelFormat, &pfd);
	if (!result) {
		context.hdc = NULL;
		context.hrc = NULL;
		return context;
	}

	HGLRC tempOpenGLContext = wglCreateContext(context.hdc);
	wglMakeCurrent(context.hdc, tempOpenGLContext);

	if (glewInit() != GLEW_OK) {
		context.hdc = NULL;
		context.hrc = NULL;
		return context;
	}

	return context;
}
```

If the returned OpenGL handle object is not null, the WinMain function calls ShowWindow function by passing the window handle as its parameter. ShowWindow function in turn calls ShowWindow function from the Windows API. The main function sets the background color of the window to blue using the

OpenGL function glClearColor. After the window is created the main function enters into a message loop and listens for different inputs passed to the window.

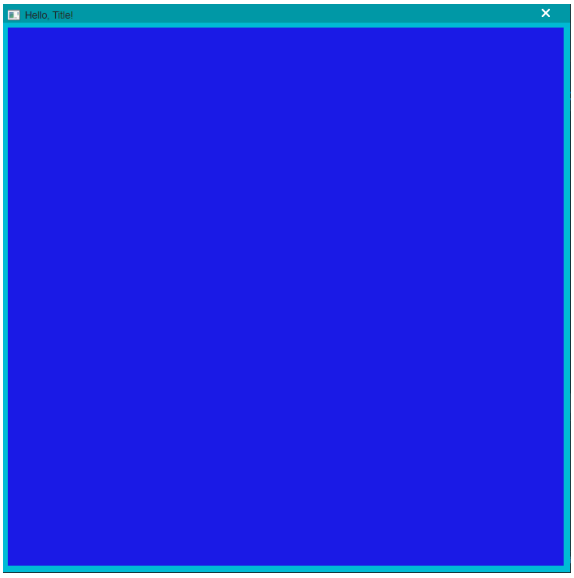
```
void ShowWindow(HWND hWnd)
{
	if (hWnd != NULL) {
		ShowWindow(hWnd, SW_SHOW);
		UpdateWindow(hWnd);
	}
}

void MessageLoop(const Context &context)
{
	bool timeToExit = false;
	MSG msg;
	while (!timeToExit) {
		if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
			if (msg.message == WM_QUIT) {
				timeToExit = true;
			}
			else {
				TranslateMessage(&msg);
				DispatchMessage(&msg);
			}
		}
		else {
			glViewport(0, 0, 500, 500);
			glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);
			SwapBuffers(context.hdc);
		}
	}
}
```

The code sample above handles WM_QUIT message. In other words, if a user presses the quit button on the game window, the loop exits and the main function now calls the CleanUp function by passing window and the context object as its parameters.

```
void CleanUp(const WindowStruct &window, const Context &context)
{
	if (context.hdc != NULL) {
		wglMakeCurrent(context.hdc, 0);
		wglDeleteContext(context.hrc);
		ReleaseDC(window.hWnd, context.hdc);
	}
}
```

The cleanup function releases the handle and device context.

If we run the project using **Debug > Start Debugging** menu option, it creates a window as shown below.



Some note-worthy attributes from above code are listed below with their simple description.

- HINSTANCE: A handle to an instance of the module in memory.
- LPSTR: A pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters.
- HWND: A handle to the window instance. A handle hides real memory address from the API user. Resolving a handle into a pointer locks the memory, and releasing the handle invalidates the pointer.
- GetModuleHandle: Retrieves a module handle for specified module. The module must have been loaded by the calling process. If NULL parameter is passed, GetModuleHandle returns a handle to the file used to create the calling process (i.e.; the .exe file).
- WNDCLASS: Contains window class attributes. The RegisterClass registers these attributes.
- CALLBACK: Alias for __stdcall. The __stdcall calling convention is used to call WIN32 API functions.
- WPARAM: Message parameter.
- LPARAM: Message parameter.
- WM_CLOSE: A signal for terminating the window.
- WM_DESTROY: Message sent when a window is being destroyed.
- PostQuitMessage(0): Indicates to the system that a tread has made a request to terminate. This method is used in response to a WM_DESTROY message. The parameter 0 indicates it wants to quit normally.
- DefWindowProc: Calls the default window procedure to provide default processing for any window messages that an application does not process. This function ensures that every message is processed.
- PeekMessage: The PeekMessage function dispatches incoming sent messages, checks the thread message queue for a posted message, and retrieves the message (if any exist).
- WM_QUIT: Indicates a request to terminate an application. This message is generated after the application calls PostQuitMessage function.

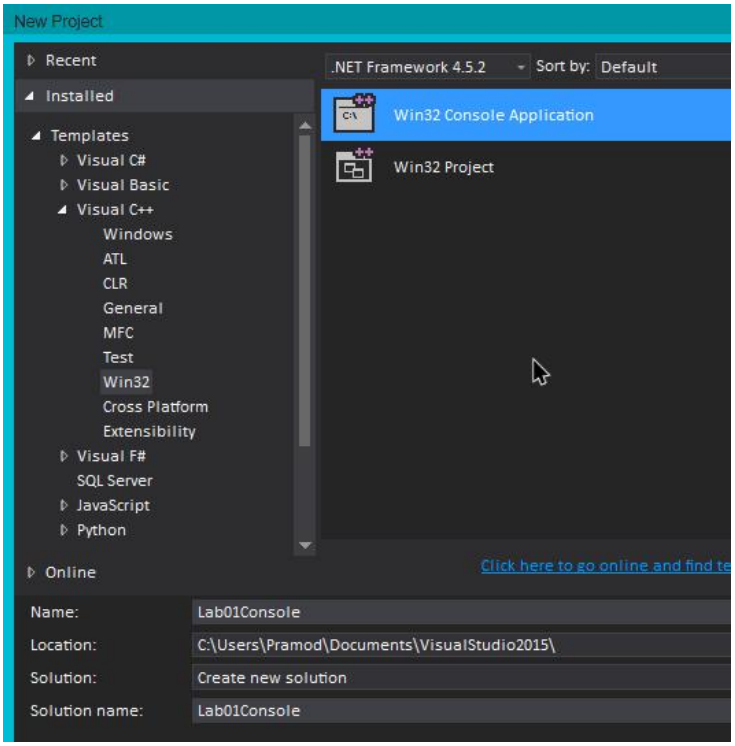
- TranslateMessage: Translates virtual-key messages into character messages. The character message are posted to the calling thread's message queue to be read the next time the thread calls the GetMessage or PeekMessage function.
- DispatchMessage: Dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.
- The #pragma comment(lib "filename.lib") code causes the linker to search for the named lib file while linking.
- HGLRC: A handle to the OpenGL rendering context. An OpenGL rendering context is a port through which all OpenGL commands pass. Every thread that makes OpenGL calls must have a current rendering context. Rendering contexts link OpenGL to the Windows windowing systems.
- HDC: A handle to a device context. A device context is a Windows data structure containing information about the drawing attributes of a device such as a display or a printer. All drawing calls are made through a device-context object, which encapsulates the Windows APIs for drawing lines, shapes, and text.
- PIXELFORMATDESCRIPTOR: describes the pixel format of a drawing surface.
- ChoosePixelFormat: Attempts to match an appropriate pixel format supported by a device context to a given pixel format specification.
- SetPixelFormat: Sets the pixel format of the specified device context to the format specified by the pixelFormat index.
- wglCreateContext:Creates a new OpenGL rendering context, which is suitable for drawing on the device referenced by hdc. The rendering context has the same pixel format as the device context.
- wglMakeCurrent: Makes a specified OpenGL rendering context the calling thread's current rendering context. All subsequent OpenGL calls made by the thread are drawn on the device identified by hdc. The function wglMakeCurrent can also be used to change the calling thread's current rendering context so it's no longer current.
- glewInit: Used to initialize the extension entry points.
- glViewport: Sets the viewport. It specifies the affine transformation of x and y from normalized device coordinates to window coordinates.
- glClear: Clear buffers to preset values. Different values are passed to select different buffers. E.g., GL_COLOR_BUFFER_BIT indicates the buffer currently enabled for color writing, GL_DEPTH_BUFFER_BIT indicates the depth buffer and GL_STENCIL_BUFFER_BIT indicates the stencil buffer.
- SwapBuffers: Exchanges the front and back buffers if the current pixel format for the window referenced by the specified device context includes a back buffer.
- wglDeleteContext: Deletes a specified OpenGL rendering context.
- ReleaseDC: Releases a device context (DC), freeing it for use by other applications.
- glClearColor: Specifies the red, green, blue, and alpha values used by glClear to clear the color buffers. Values specified by glClearColor are clamped to the range 0 to 1.

Part 2 (Simple Console Application)

In this part of the Lab I created a console based dungeon map. The application creates a map using char inputs, it shows the basics of window setup for console based games. Moreover, this project also reads the

Dungeon coordinates from a text file called Dungeon.txt. Therefore, this application shows a way to externalize data.

To create a Console application, in **Visual Studio**, I create a Win32 Project just like before (similar as creating a Win32 GUI application), but during the Win32 **Application Wizard**, I selected **Console application** instead of Windows application. Another way to create a Console application is to select Win32 Console Application from **File > New Project > Templates > Visual C++ > Win32 > Win32 Console Application**.




```
#include <Windows.h>
#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <cstdlib> // system()
#include <conio.h> // Visual C++ console only!
using namespace std;

COORD GetConsoleSize()
{
	CONSOLE_SCREEN_BUFFER_INFO csbi;
	GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &csbi);
	COORD size;
	size.X = csbi.srWindow.Right;
	size.Y = csbi.srWindow.Bottom;
	return size;
}
```

```
int main(void)
{

	system("cls");

	COORD size = GetConsoleSize();


	vector<Wall> walls;
	ReadDungeon("Dungeon.txt", walls);

	for (int i = 0; i < walls.size(); i++) {
		DrawWall(walls[i]);
	}

	GotoXY(0, size.Y);

	cout << "Press any key to exit...";

	_getch();

	return 0;
}
```

The code snippet above shows the key code that creates the console application. The main function uses GetConsoleSize function to get the size of the created console window. It then creates vector of Wall objects. It populates the walls vector with the Dungeon coordinates by reading the coordinates values from Dungeon.txt using ReadDungeon function. The ReadDungeon function uses fstream to read in the text file's text and set it to the appropriate attribute of the Wall object (struct). The main function iterates through the walls vector and calls DrawWall function to draw coordinates for each walls object in the vector. The ReadDungeon function looks like the following code.

```
struct Wall {
	int startX, startY;
	int direction;
	char character;
	int length;
};

void GotoXY(int x, int y)
{
	COORD coord;
	coord.X = x;
	coord.Y = y;
	SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}

void DrawWall(const Wall &wall)
{

	if (wall.direction == 0) {
		for (int i = wall.startX; i < wall.startX + wall.length; i++) {
			GotoXY(i, wall.startY);
			cout << wall.character;
```

```
			}
		}
		else if (wall.direction == 1) {
			for (int i = wall.startY; i < wall.startY + wall.length; i++) {
				GotoXY(wall.startX, i);
				cout << wall.character;
			}
		}
}

void ReadDungeon(string filename, vector<Wall>& walls)
{
		ifstream fin;
		fin.open(filename);
		if (fin.fail()) {
			cout << "\nThere was error opening the file." << endl;
			exit(1);
		}

		while (!fin.eof()) {
			Wall wall;
			fin >> wall.startX >> wall.startY >> wall.direction >> wall.character
				>> wall.length;
			if (fin.fail()) {
				break;
			}
			walls.push_back(wall);
		}
}
```

The DrawWall function above uses GotoXY function to select the coordinates where the next character is going to be printed. The GotoXY function uses SetConsoleCursorPosition to set the coordinates for character output.

Part of Dungeon.txt looks like this.

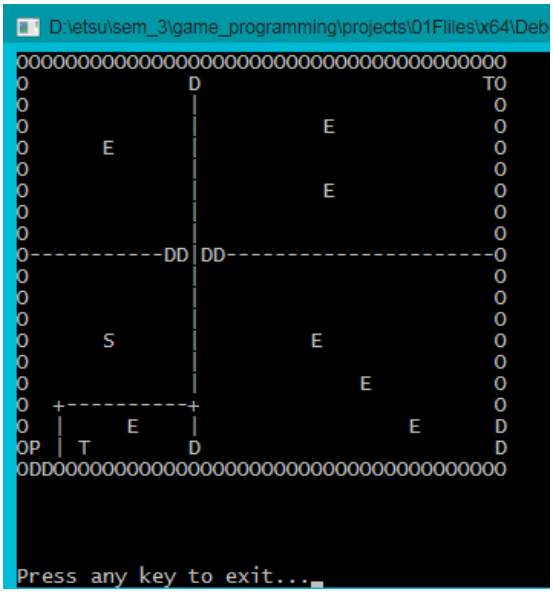
```
0 0 0 O 40
0 1 1 O 19
1 19 0 D 2
3 19 0 O 37
39 1 1 O 16
```

Similarly, ReadDungeon function looks like this.

```
void ReadDungeon(string filename, vector<Wall>& walls)
{
		ifstream fin;
		fin.open(filename);
		if (fin.fail()) {
			cout << "\nThere was error opening the file." << endl;
			exit(1);
		}

		while (!fin.eof()) {
			Wall wall;
			fin >> wall.startX >> wall.startY >> wall.direction >> wall.character
```

```
                >> wall.length;
            if (fin.fail()) {
                break;
            }
            walls.push_back(wall);
        }
}
```

The final output after compiling and running the project looks like the following window.



Lab 02 (The Presentation and Console Subsystems)

The objective of this lab was to understand the presentation of the window and use the Console sub-system to log debug messages.

The lab sets up the Logger class to log messages to the standard output (console). Using the Logger as the printing class, the project creates a menu options for the user to change the background color and color behavior depending upon the keys pressed.

For creating the window the project uses several classes. First, a GenericWindow class is created that creates several functions for its derived classes to implement.

```
#pragma once
#ifndef GENERIC_WINDOW
#define GENERIC_WINDOW

#include <string>
using std::string;
using std::wstring;

class Logger;

class GenericWindow
{
protected:
	Logger *logger;

public:
	GenericWindow();
	virtual ~GenericWindow();

	virtual bool create() = 0;
	virtual void show() = 0;
	virtual void listenForEvents() = 0;

	void setLogger(Logger *logger) { this->logger = logger; }

	void log(string text, bool debug=false);
	void log(wstring text, bool debug=false);
};

#endif
```

It can be inferred from above code that Logger is passed to GenericWindow through the setLogger function. The log method calls **logger->log** method with the passed message to log messages to the console window. The WindowsConsoleLogger class creates the logger by creating the stdandard output handle using GetStdHandle method. The log method in WindowsConsoleLogger class creates a new buffer based on the size of string passed as the parameter and calls write method. The write method calls WriteConsoleW method of the windows console API. The following code snippet show this behavior.

```
void WindowsConsoleLogger::create()
{
	this->stdOut = NULL;
```

```
	if (AllocConsole() == 0) return;

	this->stdOut = GetStdHandle(STD_OUTPUT_HANDLE);
	if (this->stdOut == INVALID_HANDLE_VALUE || this->stdOut == NULL) {
		this->stdOut = NULL;
		return;
	}
}

void WindowsConsoleLogger::write()
{
	if (this->stdOut != NULL) {
		DWORD check;
		WriteConsoleW(this->stdOut, (void*)this->buffer, this->size, &check, NULL);
	}
}

void WindowsConsoleLogger::log(string text)
{
	text += "\n";
	this->size = text.length();
	if (this->size > BUFFER_SIZE) {
		this->size = BUFFER_SIZE;
	}
	if (this->buffer) delete[] this->buffer;
	this->buffer = new wchar_t[this->size];

	int i = 0;
	for (i = 0; i < this->size; i++) {
		this->buffer[i] = text[i];
	}
	write();
}
```

Another important part of this lab is the window creation code. The window creation code is written in Win32Window class. Win32OGLWindow class extends this class and creates OpenGL context for drawing OpenGL objects. Finally, MyGameWindow extends Win32OGLWindow, which I used to setup the background behavior like changing background color based of different key press. Win32Window class breaks the code we saw in Lab 01 into few interesting functions like create, show, listenForEvents, and runOneFrame. The create function is similar to CreateCustomWindow, the show function is similar to ShowWindow, listenForEvents is similar to MessageLoop (excluding the else block), and runOneFrame (overridden in Win32OGLWindow class) is similar to MessageLoop (the else block code) function we saw in Lab 01 code. In addition, the Win32OGLWindow class overrides the create method, which is similar to CreateOGLWindow function we saw in Lab 01 code. The create method of Win32OGLWindow class also adds code for logging OpenGL version currently in use. Following code snippet shows the windows creation functionality.

```
bool Win32OGLWindow::create()
{
	if (!Win32Window::create()) return false;

	this->deviceContext = GetDC(this->windowHandle);
	if (this->deviceContext == NULL) {
		this->log("Could not get the device context!");
```

```
		return false;
	}

	PIXELFORMATDESCRIPTOR pfd;
	memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
	pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
	pfd.dwFlags = PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW;
	pfd.iPixelType = PFD_TYPE_RGBA;
	pfd.cColorBits = 32;
	pfd.cDepthBits = 32;

	int pixelFormat = ChoosePixelFormat(this->deviceContext, &pfd);
	if (pixelFormat == 0) {
		this->log("Could not choose the pixel format!");
		return false;
	}

	BOOL result = SetPixelFormat(this->deviceContext, pixelFormat, &pfd);
	if (!result) {
		this->log("Could not set the pixel format!");
		return false;
	}

	HGLRC tempOpenGLContext = wglCreateContext(this->deviceContext);
	wglMakeCurrent(this->deviceContext, tempOpenGLContext);

	if (glewInit() != GLEW_OK) {
		this->log("Could not initialize glew!");
		return false;
	}

	// Read the opengl version
	stringstream stringStream;
	stringStream << glGetString(GL_VERSION);
	getline(stringStream, this->oglVersion);

	this->log("OpenGL version: " + this->oglVersion);

	return true;
}
```

The MyGameWindow class extends Win32OGLWindow and sets several variables to store state of color and color change.

```
#pragma once
#ifndef MY_GAME_WINDOW
#define MY_GAME_WINDOW

#include "Win32OGLWindow.h"


class MyGameWindow :
	public Win32OGLWindow
{
private:
	struct RGBA {
```

```
		float red, green, blue, alpha;
	};

	RGBA background;
	static MyGameWindow *self;
	float delta = 0.0001f;

	enum COLOR_STATE {
		NO_COLOR, BLACK_RED_BLACK, BLACK_WHITE_BLACK, RED_GREEN_BLUE_RED
	};

	enum TMP_COLOR_STATE {
		RED_GREEN, GREEN_BLUE, BLUE_GREEN, GREEN_RED
	};

	COLOR_STATE currentState;
	TMP_COLOR_STATE tmpColorState;

	void clearScreen(float red, float green, float blue);

public:
	MyGameWindow();
	virtual ~MyGameWindow();


protected:
	void runOneFrame();
	void update();
	void render();

	static LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam);
};

#endif
```

The RGBA struct is used to keep track of the background red, green, blue and alpha color values. I use self variable pointer to point to an instance of MyGameWindow. Since the variable is static, it remains in memory until the program is terminated. The self variable is used by code that works for class level variable. The tmpColorState is used to keep track of internal transition when changing colors from Red > Green > Blue > Green > Red. The clearScreen method sets the background color of the window.

Note: For simplicity, I use arrow (>) to show transition of colors. E.g., Black > White means the color transitions from Black to White.

```
void MyGameWindow::clearScreen(float red, float green, float blue)
{
	self->background.red = red;
	self->background.green = green;
	self->background.blue = blue;
}

void MyGameWindow::runOneFrame()
{
	update();
	render();
```

```
}
```

The runOneFrame function calls the update and render functions. The update function is responsible for updating the scene coordinates, which in the context of the application is background color values addition based on delta values. Similarly, the render function renders (draws) the background based on the updated coordinates. This function uses clearScreen to set the new values for rgb color values.

```
void MyGameWindow::update()
{
	switch (currentState) {
	case BLACK_RED_BLACK:
		this->background.red += delta;

		if (this->background.red >= 1.0f || this->background.red <= 0)
			delta *= -1.0;
		break;


	case BLACK_WHITE_BLACK:
		this->background.red += delta;
		this->background.green += delta;
		this->background.blue += delta;

		if (this->background.red >= 1.0f || this->background.red <= 0)
			delta *= -1.0;
		break;

	case RED_GREEN_BLUE_RED:

		switch (tmpColorState) {
		case RED_GREEN:
			this->background.red -= delta;
			this->background.green += delta;

			if (this->background.red < 0.0f || this->background.green > 1.0f) {
				clearScreen(0.0f, 1.0f, 0.0f);
				tmpColorState = GREEN_BLUE;
			}
			break;

		case GREEN_BLUE:
			this->background.green -= delta;
			this->background.blue += delta;

			if (this->background.green < 0.0f || this->background.blue > 1.0f) {
				clearScreen(0.0f, 0.0f, 1.0f);
				tmpColorState = BLUE_GREEN;
			}
			break;

		case BLUE_GREEN:
			this->background.blue -= delta;
			this->background.green += delta;

			if (this->background.blue < 0.0f || this->background.green > 1.0f) {
```

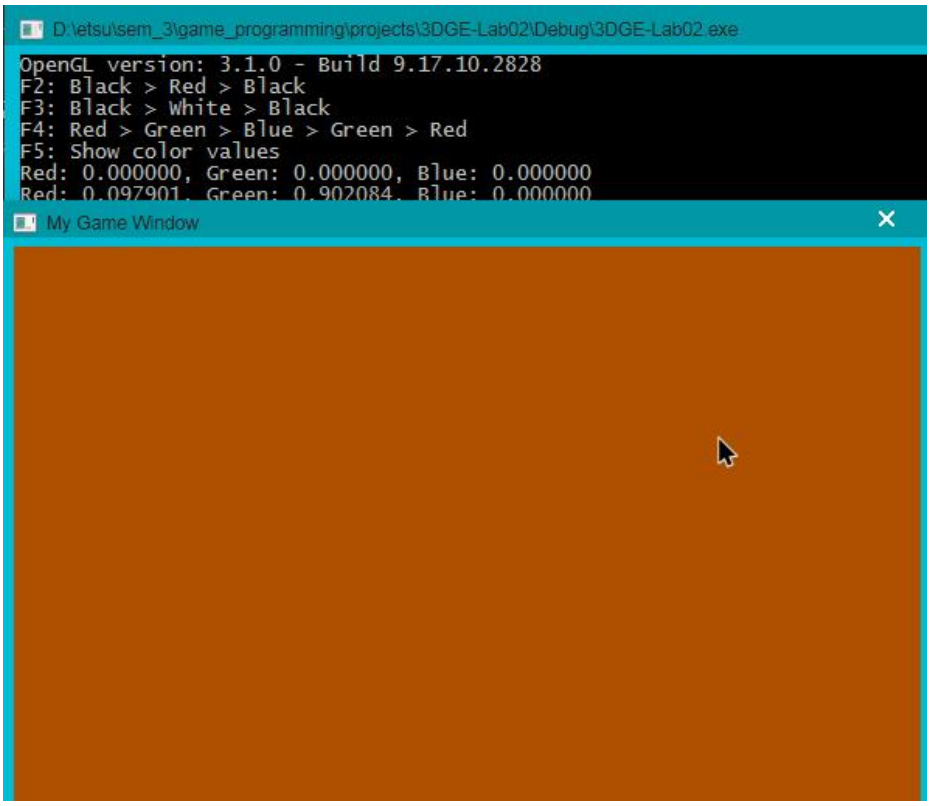
```
                        clearScreen(0.0f, 1.0f, 0.0f);
                        tmpColorState = GREEN_RED;
                }
                break;

        case GREEN_RED:
                this->background.green -= delta;
                this->background.red += delta;

                if (this->background.red > 1.0f || this->background.green < 0.0f) {
                        clearScreen(1.0f, 0.0f, 0.0f);
                        tmpColorState = RED_GREEN;
                }
                break;
        }

        break;

    }
}

void MyGameWindow::render()
{
    glViewport(0, 0, this->width, this->height);

    glClearColor(
        this->background.red,
        this->background.green,
        this->background.blue,
        this->background.alpha);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    SwapBuffers(this->deviceContext);
}
```

As mentioned in Lab 01 section, WndProc method is responsible for listening to window events through messages (especially key presses). This application handles F1, ESCAPE, F2, F3, F4 and F5 keys. On F1 key presses, it prints the menu using the Logging function. Escape quits the application, F2 changes the background color to Black > Red > Black, F3 changes the color to Black > White > Black, F4 changes background color to Red > Green > Blue > Green > Red and F5 prints the red, green and blue color values for currently used color for the background. This functionality can be seen in following code sample.

```
LRESULT CALLBACK MyGameWindow::WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
    switch (message) {
    case WM_KEYDOWN:
        if (wParam == VK_F1) {
            self->log("F2: Black > Red > Black");
            self->log("F3: Black > White > Black");
            self->log("F4: Red > Green > Blue > Green > Red");
            self->log("F5: Show color values");
        }
        else if (wParam == VK_ESCAPE) {
            self->log("Exiting!");
            self->exit();
```

```
                }
                else if (wParam == VK_F2) {
                        self->currentState = BLACK_RED_BLACK;
                        self->clearScreen(0.0f, 0.0f, 0.0f);
                        self->update();
                }
                else if (wParam == VK_F3) {
                        self->currentState = BLACK_WHITE_BLACK;
                        self->clearScreen(0.0f, 0.0f, 0.0f);
                        self->update();
                }
                else if (wParam == VK_F4) {
                        self->currentState = RED_GREEN_BLUE_RED;
                        self->clearScreen(1.0f, 0.0f, 0.0f);
                        self->tmpColorState = RED_GREEN;
                        self->update();
                }
                else if (wParam == VK_F5) {
                        string colorStr = "Red: " + to_string(self->background.red) + ",
Green: " + to_string(self->background.green) +
                                ", Blue: " + to_string(self->background.blue);
                        self->log(colorStr);

                }
                return 0;
        }

        return Win32OGLWindow::WndProc(hWnd, message, wParam, lParam);
}
```

After running the application it displays following output.


D:\etsu\sem_3\game_programming\projects\3DGE-Lab02\Debug\3DGE-Lab02.exe
OpenGL version: 3.1.0 - Build 9.17.10.2828
F2: Black > Red > Black
F3: Black > White > Black
F4: Red > Green > Blue > Green > Red
F5: Show color values
Red: 0.000000, Green: 0.000000, Blue: 0.000000
Red: 0.097901, Green: 0.902084, Blue: 0.000000
My Game Window

Lab 03 (Basic Rendering)

The objective of this lab was to learn rendering OpenGL objects using the OpenGL context we created in Lab 01 and Lab 02.

This lab uses several classes to create and organize the OpenGL drawing objects. However, since we decoupled related concerns into related classes none of the newly added classes affect our window creation code from Lab 02. The Shader class that creates the OpenGL objects is injected into the GameWindow class and only change it brings to render method from Lab 02 is the following code.

```
void GameWindow::render()
{
	glViewport(0, 0, this->width, this->height);

	glClearColor(
		this->background.red,
		this->background.green,
		this->background.blue,
		this->background.alpha
	);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

	this->shader->renderObjects();

	SwapBuffers(this->deviceContext);
}
```

The second last line of above code snippet is the only added line to the render method that is different from previous lab. The OGLShader class extends Shader class. This is the class where I define all the OpenGL objects that is created for this lab.

In the graphics pipeline vertices for the drawing objects such as triangles, lines and points are defined using coordinates values. Each of these objects is defined using the x, y, z coordinate values and corresponding color values that define the red, green, blue and alpha colors. The alpha color value defines the transparency of the color. In OpenGL, every object is defined using the primitives like points, lines and triangles. As an example a square may be composed of two triangles. The vertices that represent an object are processed by a small program in the graphics card called vertex shader. The vertex shader transforms the 3D coordinates suitable for display in the 2D screen where we see the objects. The vertex shader also passes down the color coordinates it received during processing down the graphics pipeline. The graphics card uses the transformed input coordinates to create basic primitive shapes. This step is called shape assembly. The next optional step is carried out by geometry shader. It can modify the shapes passed from shape assembly, discard or pass it down without any modification. This step can help reduce the amount of data transfer from PC to the graphics card. After this step, rasterizer turns visible parts of the shapes into pixel-sized fragments. The fragment shader processes fragments from the rasterization process into a set of color and a single depth values. A fragment is a position on the window (x, y), a depth value (z), plus all the interpolated data from previous stages. The output from fragment shader can be used to create the final output. Some other processing like lighting and shadowing could also be performed before composing the final scene.

For drawing the primitives, the primitive data is loaded into vertex buffer objects. A Vertex Buffer Object (VBO) is a memory buffer in the high speed memory of the video card designed to hold information about vertices. As an example the program may define two VBOs, one that describes the coordinates of our vertices and another that describes the color associated with each vertex. VBOs can also store information such as normals, texcoords and indicies.

A Vertex Array Object (VAO) is an object that contains one or more Vertex Buffer Objects and designed to store the information for a complete rendered object.

In OGLShader class, I declare the OpenGL objects I intend to draw for this lab.

```
#pragma once
#ifndef OGL_SHADER
#define OGL_SHADER

#include "Shader.h"

#include <gl/glew.h>

class Logger;

class OGLShader : public Shader
{
protected:
	struct Vertex {
		GLfloat x, y, z;
		GLfloat red, green, blue;
	}trianglesVertexData[6], linesVertexData[4], pointsVertexData[4];

	GLuint vertexShader, fragmentShader, shaderProgram;

	GLuint trianglesVboId, linesVboId, pointsVboId, triangleStripVboId;

	GLuint vaoId;

	Vertex myPoint;
	GLuint myPointVbo;

	Vertex myLine[2];
	GLuint myLineVbo;

	Vertex myTriangle[3];
	GLuint myTriangleVbo;

	Vertex myTriangleStrip[7];
	GLuint myTriangleStripVbo;

public:
	OGLShader(void);
	~OGLShader(void);

	bool create();
	void renderObjects();

protected:
	bool setupShaders();
```

```
        void createPrimitives();
        void renderTriangles(GLuint vbo, GLsizei numberOfVertices);
        void renderTriangleStrip(GLuint vbo, GLsizei numberOfVertices);
        void renderLines(GLuint vbo, GLsizei numberOfVertices);
        void renderPoints(GLuint vbo, GLsizei numberOfVertices);

        GLuint createVBOHandle(GLenum target, const void* bufferData, GLsizei bufferSize);
        GLuint compileShader(GLenum type, const GLchar* source);
        void showInfoLog(GLuint object, PFNGLGETSHADERIVPROC glGet__iv,
PFNGLGETSHADERINFOLOGPROC glGet__InfoLog);
        GLuint linkShader(GLuint vertexShader, GLuint fragmentShader);

        void enablePositionsAndColor();
};

#endif
```

In the above code snippet Vertex struct is used to hold the coordinate and the corresponding color values. Although several VBO objects are defined only one VAO object variable is defined. As described above, we can use one VAO to hold several VBOs.

The create method of the OGLShader class calls the setupShaders method to setup the shader for drawing the objects, and then it calls the createPrimitives function to do all the drawing. This can be seen in following code snippet.

```
bool OGLShader::create()
{
        if (!this->setupShaders()) return false;
        this->createPrimitives();
        return true;
}

bool OGLShader::setupShaders()
{
        GLchar* vertexSource =
                "#version 330\n"\
                "layout(location = 0) in vec3 position;\n"\
                "layout(location = 1) in vec3 vertexColor;\n"\
                "out vec4 fragColor;\n"\
                "void main()\n"\
                "{\n"\
                "    gl_Position = vec4(position, 1.0);\n" \
                "    fragColor = vec4(vertexColor, 1.0);\n" \
                "}\n";
        vertexShader = compileShader(GL_VERTEX_SHADER, vertexSource);
        if (vertexShader == 0) return false;

        GLchar* fragmentSource =
                "#version 330\n"\
                "in vec4 fragColor;\n"\
                "out vec4 color;\n"\
                "void main()\n"\
                "{\n"\
                "    color = fragColor;\n"\
```

```
                "}\n";
        fragmentShader = compileShader(GL_FRAGMENT_SHADER, fragmentSource);
        if (fragmentShader == 0) return false;

        shaderProgram = linkShader(vertexShader, fragmentShader);
        glDeleteShader(vertexShader);
        glDeleteShader(fragmentShader);
        return true;
}


GLuint OGLShader::compileShader(GLenum type, const GLchar* source)
{
        GLint length = sizeof(GLchar) * strlen(source);
        GLuint shader = glCreateShader(type);
        glShaderSource(shader, 1, (const GLchar**)&source, &length);
        glCompileShader(shader);
        GLint shaderOk = 0;
        glGetShaderiv(shader, GL_COMPILE_STATUS, &shaderOk);
        if (!shaderOk) {
                logger->log("makeShader: Failed to compile the shader!");
                showInfoLog(shader, glGetShaderiv, glGetShaderInfoLog);
                glDeleteShader(shader);
                shader = 0;
        }
        return shader;
}

GLuint OGLShader::linkShader(GLuint vertexShader, GLuint fragmentShader)
{
        GLuint program = glCreateProgram();
        glAttachShader(program, vertexShader);
        glAttachShader(program, fragmentShader);
        glLinkProgram(program);
        GLint programOk = 0;
        glGetProgramiv(program, GL_LINK_STATUS, &programOk);
        if (!programOk) {
                logger->log("linkShader: Failed to link the shader!");
                showInfoLog(program, glGetProgramiv, glGetProgramInfoLog);
                glDeleteShader(program);
                program = 0;
        }
        return program;
}

void OGLShader::showInfoLog(GLuint object, PFNGLGETSHADERIVPROC glGet__iv,
PFNGLGETSHADERINFOLOGPROC glGet__InfoLog)
{
        GLint logLength;
        glGet__iv(object, GL_INFO_LOG_LENGTH, &logLength);
        char* log = (char*)malloc(logLength);
        glGet__InfoLog(object, logLength, NULL, log);
        logger->log(string(log));
        free(log);
}

void OGLShader::createPrimitives()
```

```
{
	glGenVertexArrays(1, &vaoId);
	glBindVertexArray(vaoId);

	trianglesVboId = createVBOHandle(
		GL_ARRAY_BUFFER, trianglesVertexData, sizeof(trianglesVertexData));
	linesVboId = createVBOHandle(GL_ARRAY_BUFFER, linesVertexData,
sizeof(linesVertexData));
	pointsVboId = createVBOHandle(GL_ARRAY_BUFFER, pointsVertexData,
sizeof(pointsVertexData));
	this->myPointVbo = this->createVBOHandle(GL_ARRAY_BUFFER, &(this->myPoint),
sizeof(this->myPoint));
	this->myLineVbo = this->createVBOHandle(GL_ARRAY_BUFFER, this->myLine,
sizeof(this->myLine));
	this->myTriangleVbo = this->createVBOHandle(GL_ARRAY_BUFFER, this->myTriangle,
sizeof(this->myTriangle));
	this->myTriangleStripVbo = this->createVBOHandle(GL_ARRAY_BUFFER, this-
>myTriangleStrip, sizeof(this->myTriangleStrip));

	glBindVertexArray(0);
}

GLuint OGLShader::createVBOHandle(GLenum target, const void* bufferData, GLsizei
bufferSize)
{
	GLuint vboID = 0;
	// Declare the buffer object and store a handle to the object
	glGenBuffers(1, &vboID);
	// Bind the object to the binding target
	glBindBuffer(target, vboID);
	// Allocate memory in the GPU for the buffer bound to the binding target and then
copy the data
	glBufferData(target, bufferSize, bufferData, GL_STATIC_DRAW);
	// Good practice to cleanup by unbinding
	glBindBuffer(target, 0);
	return vboID;
}
```

The setupShaders method defines GLSL shader code for both vertex shader and fragment shader. It uses compileShader to compile the shader code. It then links both the shader objects and assigns it to shaderProgram. The shaderProgram object is then used while creating the OpenGL objects through glUseProgram(shaderProgram) code. E.g., To create a line (myLine, defined in OGLShader.h) the coordinates can be defined inside the constructor of OGLShader class like the following code.

```
	this->myLine[0].x = 0.9f;
	this->myLine[0].y = -0.9f;
	this->myLine[0].z = 0.0f;
	this->myLine[0].red = 0.0f;
	this->myLine[0].green = 1.0f;
	this->myLine[0].blue = 0.0f;

	this->myLine[1].x = 0.7f;
	this->myLine[1].y = -0.7f;
	this->myLine[1].z = 0.0f;
	this->myLine[1].red = 0.0f;
	this->myLine[1].green = 1.0f;
```

```
	this->myLine[1].blue = 0.0f;
```

Since the GameWindow class calls the renderObjects method of the OGLShader class, the renderObjects method calls the appropriate functions to create each object (renderLines in this case).

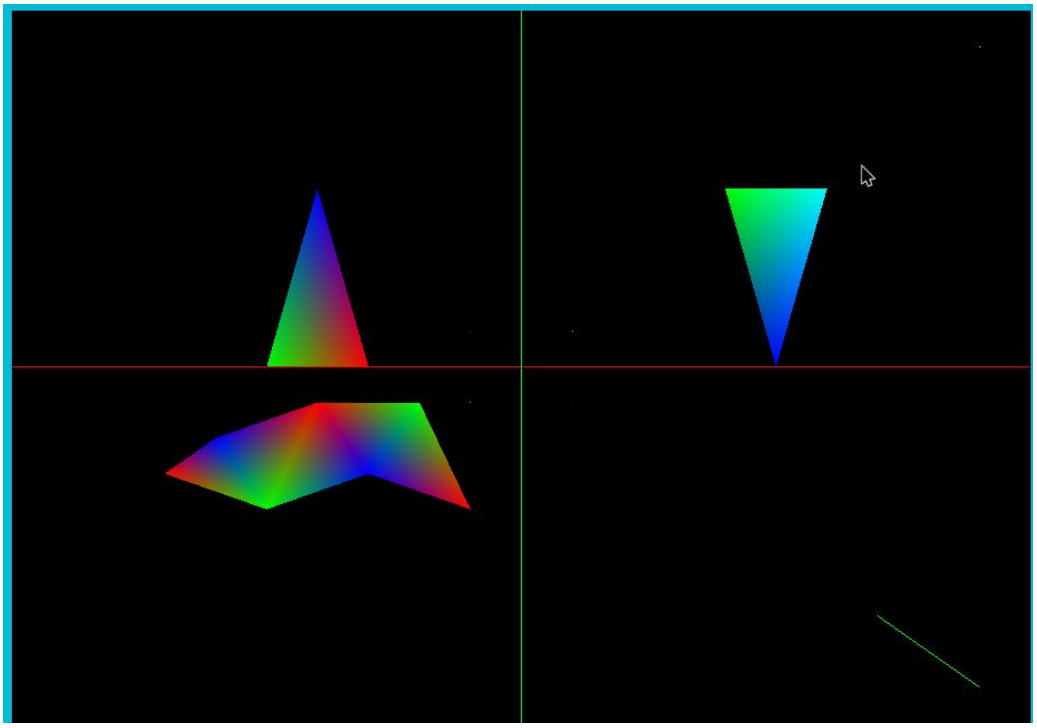
```
void OGLShader::renderObjects()
{
	this->renderLines(this->myLineVbo, 2);
	// This function also calls all other object generation functions
}
```

The second parameter is number of vertices and the first parameter is the VBO object. The renderLines method can be used to create lines. It uses the VAO object, binds the buffer to VBO and draws the lines (GL_LINES) using glDrawArrays function.

```
void OGLShader::renderLines(GLuint vbo, GLsizei numberOfVertices)
{
	glBindVertexArray(vaoId);
	glUseProgram(shaderProgram);
	glBindBuffer(GL_ARRAY_BUFFER, vbo);

	enablePositionsAndColor();

	glDrawArrays(GL_LINES, 0, numberOfVertices);

	glDisableVertexAttribArray(0);
	glDisableVertexAttribArray(1);
	glUseProgram(0);
	glBindVertexArray(0);
}


void OGLShader::enablePositionsAndColor()
{
	// Positions
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(
		0,
		3,			// Each position has 3 components
		GL_FLOAT,		// Each component is a 32-bit floating point value
		GL_FALSE,
		sizeof(Vertex), // The number of bytes to the next position
		(void*)0		// Byte offset of the first position in the array
		);
	// Colors
	glEnableVertexAttribArray(1);
	glVertexAttribPointer(
		1,
		3,				// Each color has 3 components
		GL_FLOAT,			// Each component is a 32-bit floating point
value
		GL_FALSE,
		sizeof(Vertex),			// The number of bytes to the next color
		(void*)(sizeof(GLfloat) * 3) // Byte offset of the first color in the array
		);
}
```

The enablePositionAndColor method specifies how position and color values are distributed in our myLine Vertex array. It is up to the programmer to form a strategy to store position and color values in the array.

The final output looks like this.



Lab 04 (Simple Shaders Part 2)

The objective of this lab was to practice with different shader programs to render the objects. The lab mostly focuses on varying the size of the drawing world.

I practiced changing the size by varying the coordinates of the OpenGL rendering window from -1.0 and 1.0 to -100 and 100 and -1000 to 1000 by multiplying gl_Position in the vertex shader GLSL code with appropriate floating point values (changed the precision). Following code sample shows how the coordinates can be specified from -1000 to 1000 range.

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 vertexColor;
out vec4 fragColor;

void main()
{
	vec4 transformed = vec4(position * .001, 1.0);
	gl_Position = transformed;
	fragColor = vec4(vertexColor, 1.0);
}
```

OGLShaderCompiler::compile method compiles the shaders.

```
void OGLShaderCompiler::compile(GLenum type, const GLchar* source)
{
	GLint length = sizeof(GLchar) * strlen(source);
	this->handle = glCreateShader(type);
	glShaderSource(this->handle, 1, (const GLchar**)&source, &length);
	glCompileShader(this->handle);
	GLint shaderOk = 0;
	glGetShaderiv(this->handle, GL_COMPILE_STATUS, &shaderOk);
	if (!shaderOk) {
		if (this->logger) {
			this->logger->log("makeShader: Failed to compile the shader!");
		}
		showInfoLog(this->handle, glGetShaderiv, glGetShaderInfoLog);
		glDeleteShader(this->handle);
		this->handle = 0;
	}
}
```

In this lab, we externalize the shader files. For the vertex shader, the WinMain function reads the content of the shader file using TextFileReader class's readContents method and passes it to the constructor of OGLVertexShader. The WinMain function also creates the fragment shader using OGLFragmentShader. Since no shader file was passed to OGLFragmentShader, it uses its stockSource method to define a fragment shader in code. The OGLShaderProgram is then responsible for attaching both the shaders to the program, which is done in the link method. The link method is called from setupShaders method (as described in Lab 03 code section). Following code samples shows the uniqueness of setupShaders as compared to Lab 03.

```
bool OGLRenderer::setupShaders()
{
```

```
	bool result = false;
	result = this->vertexShader->compile();
	if (!result) return false;

	result = this->fragmentShader->compile();
	if (!result) return false;

	return this->shaderProgram->link(
		this->vertexShader->getHandle(),
		this->fragmentShader->getHandle()
		);
}
```

The rendering of the objects is done by OGLObject's renderObject method. This method is generic and now we no longer have to define same code over and over again to create objects. The glDrawArrays function is passed the object type using object->primitiveType. Following code shows the renderObject method.

```
void OGLObject::renderObject(GLuint shaderProgramHandle, VBOObject * object)
{
	glBindVertexArray(this->vao);
	glUseProgram(shaderProgramHandle);
	glBindBuffer(GL_ARRAY_BUFFER, object->vbo);

	// Positions
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(
		0,
		object->positionComponent.count,
		object->positionComponent.type,
		GL_FALSE,
		object->positionComponent.bytesToNext,
		(void*)object->positionComponent.bytesToFirst
		);

	// Colors
	glEnableVertexAttribArray(1);
	glVertexAttribPointer(
		1,
		object->colorComponent.count,
		object->colorComponent.type,
		GL_FALSE,
		object->colorComponent.bytesToNext,
		(void*)object->colorComponent.bytesToFirst
		);

	glDrawArrays(object->primitiveType, 0, object->numberOfVertices);

	glDisableVertexAttribArray(0);
	glDisableVertexAttribArray(1);
	glUseProgram(0);
	glBindVertexArray(0);
}
```

This method is called from OGLRenderer on each object. The objects variable is a map variable that uses user defined unique name as the key and the corresponding OGLObject as the value. The OGLRenderer class defines each object to be rendered and calls the renderObjects to render the objects. The renderObjects method iterates through the map and calls the renderObject method in the OGLObject class that we saw above.

```
	this->objects["PLT"] = new OGLObject("PLT");
	this->objects["square"] = new OGLObject("square");
	this->objects["A"] = new OGLObject("A");
	this->objects["secondsquare"] = new OGLObject("secondsquare");
```

```
void OGLRenderer::renderObjects()
{
	for (auto iterator = objects.begin(); iterator != objects.end(); iterator++) {
		iterator->second->render(this->shaderProgram->getHandle());
	}
}
```

The OGLRenderer create method calls all the methods that set the color and position coordinates and creates VBO for each object.

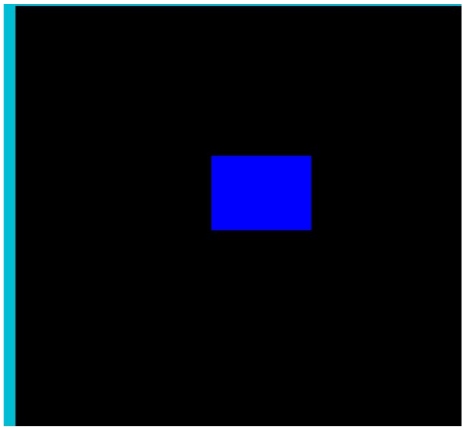
```
bool OGLRenderer::create()
{
	if (this->setupShaders()) {
		this->createStockObject();
		this->createSquare();
		this->createA();
		this->createSecondSquare();
		return true;
	}
	return false;
}
```

A square creating function looks like the following code.

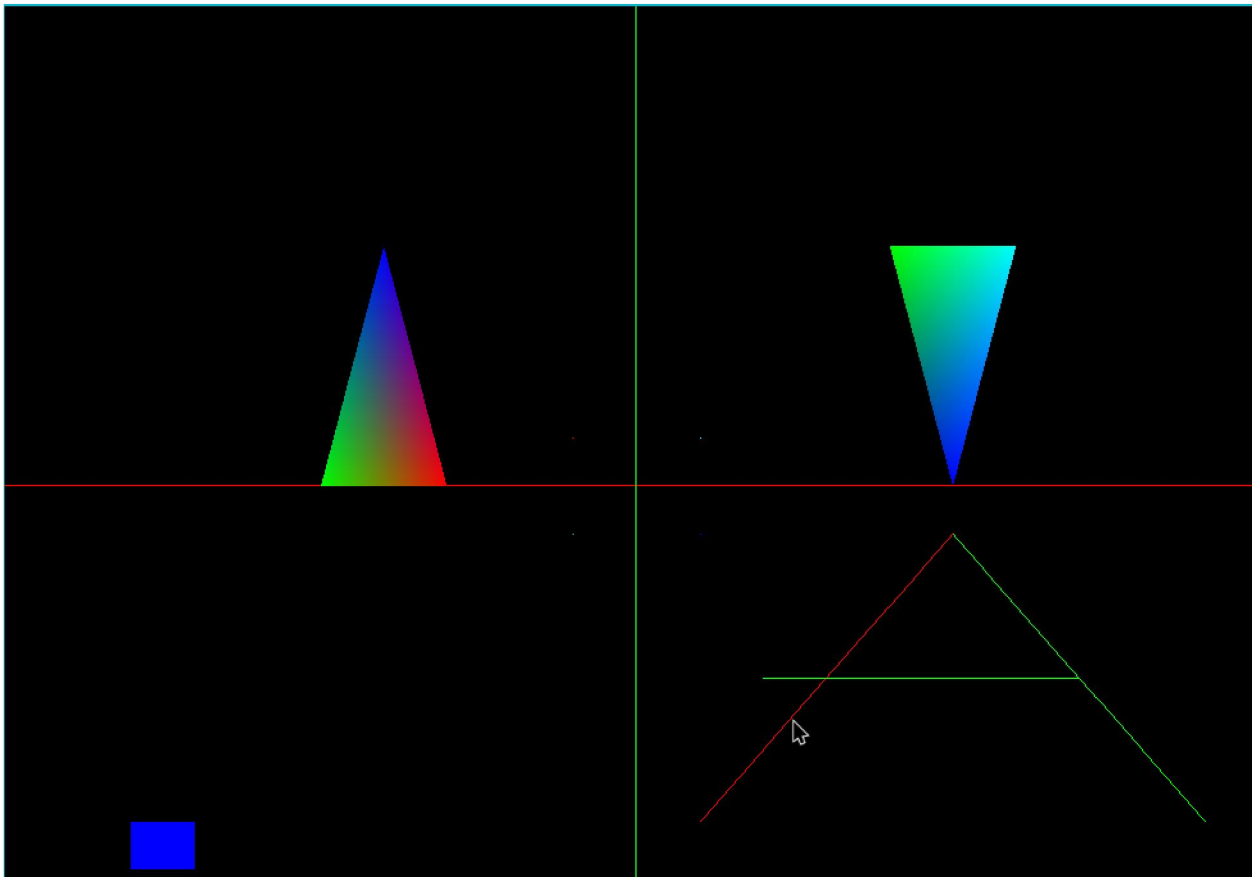
```
void OGLRenderer::createSquare()
{
	squareVertexData[0].x = -800.0f;
	squareVertexData[0].y = 700.0f;
	squareVertexData[0].z = 0.0f;
	squareVertexData[0].red = 0.0f;
	squareVertexData[0].green = 0.0f;
	squareVertexData[0].blue = 1.0f;
	squareVertexData[1].x = -700.0f;
	squareVertexData[1].y = 700.0f;
	squareVertexData[1].z = 0.0f;
	squareVertexData[1].red = 0.0f;
	squareVertexData[1].green = 0.0f;
	squareVertexData[1].blue = 1.0f;
	squareVertexData[2].x = -800.0f;
	squareVertexData[2].y = 800.0f;
```

```
        squareVertexData[2].z = 0.0f;
        squareVertexData[2].red = 0.0f;
        squareVertexData[2].green = 0.0f;
        squareVertexData[2].blue = 1.0f;

        squareVertexData[3].x = -700.0f;
        squareVertexData[3].y = 700.0f;
        squareVertexData[3].z = 0.0f;
        squareVertexData[3].red = 0.0f;
        squareVertexData[3].green = 0.0f;
        squareVertexData[3].blue = 1.0f;
        squareVertexData[4].x = -700.0f;
        squareVertexData[4].y = 800.0f;
        squareVertexData[4].z = 0.0f;
        squareVertexData[4].red = 0.0f;
        squareVertexData[4].green = 0.0f;
        squareVertexData[4].blue = 1.0f;

        squareVertexData[5].x = -800.0f;
        squareVertexData[5].y = 800.0f;
        squareVertexData[5].z = 0.0f;
        squareVertexData[5].red = 0.0f;
        squareVertexData[5].green = 0.0f;
        squareVertexData[5].blue = 1.0f;

        VBOObject * triangles = OGLObject::createVBOObject("triangles");
        triangles->buffer = squareVertexData;
        triangles->primitiveType = GL_TRIANGLES;
        triangles->bufferSizeInBytes = sizeof(squareVertexData);
        triangles->numberOfVertices = 6;
        triangles->positionComponent.count = 3;
        triangles->positionComponent.type = GL_FLOAT;
        triangles->positionComponent.bytesToFirst = 0;
        triangles->positionComponent.bytesToNext = sizeof(Vertex);
        triangles->colorComponent.count = 3;
        triangles->colorComponent.type = GL_FLOAT;
        triangles->colorComponent.bytesToFirst = sizeof(GLfloat) * 3;
        triangles->colorComponent.bytesToNext = sizeof(Vertex);
        this->objects["square"]->addVBOObject(triangles);


}
```

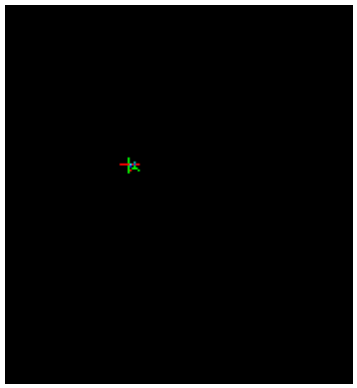
As it can be seen from above code, the coordinates are specified in a -1000 to 1000 range in both x and y coordinates and 0 for z coordinate. Upon executing above code with the shader we defined above, we get following output.



If we change the vertex shader to the file that sets gl_Position from -1 to 1, we get following output.



The square we saw above gets hidden, because the coordinates are out of scope for current screen. If we change the gl_Position range from -100 to 100 by multiplying it with 0.01, then above screen gets scaled down and appears as shown in following diagram.



In this lab, I learned different aspects using shader program. I was able to transform the coordinate system that suited my purpose. I also learned the methodology to read shaders from external file. The drawing function is now generic and takes into consideration for presence of external shader files and uses the handle to reference the appropriate object. If there is no external shader file defined, it uses the default shader program defined in one of its methods. This lab also expanded on separation of concerns by grouping related functionality into their own classes and created an API for employing the class's objective. At this point, the program has already grown to be large enough to complicate things, if related functionalities are not grouped and defined properly.

Lab 05 (3D Projection)

The objective of this lab was to create 3D objects using OpenGL drawing functions.

Until this lab we have been using 3D drawing functions to create 2D objects. We created 2D objects by setting z coordinate of all our OpenGL objects as 0. This lab also uses third party libraries for math calculation and other utility functions for creating 3D objects. The unofficial OpenGL software development kit can be found at http://glsdk.sourceforge.net/docs/html/index.html. I download and compiled glsdk using Visual Studio. I included the headers for GLM, glutil and boost libraries to **Include Directories**. I also included the glutil library directory to **Library Directories**. This project also employs keyboard input and uses the left and right arrow keys to move the camera around the y axis and up and down keys to move around x axis.

To create an object I declared the generator function in ObjectGenerator.h E.g.,

```
static float* generateXZSurface(float width, float depth);
```

The generateXZSurface creates a 2D surface, but now it uses the x-coordinate as width and the z-coordinate as depth to create the surface. Before this we used the x and y coordinates to draw the objects.

The generateXZSurface function returns an array of data object that contain the position coordinates and color values for two triangles that compose the surface.

Before creating the surface the origin (center) has to be identified.

```
// This object's origin is in the center.
float halfWidth = width / 2.0f;
float halfDepth = depth / 2.0f;
```

Since there is no square object in OpenGL, we have to use primitive types like lines, points and triangles to create our objects. Therefore, the surface is composed of two triangles each defined with 4 coordinate values. The fourth coordinate is the homogenous coordinate that is used for projecting a 3D scene onto a 2D screen. The homogenous coordinate is set to 1, because 0 means two parallel lines meet at infinity (or never meet with each other). Also, the color is defined using 4 values (i.e.; red, green, blue and alpha). Altogether it requires 8 values to represent one point. As a strategy to go through the array to read position and color values the 0^{th} index of the array is used to store the overall size of the drawing positions and colors. Counting the first array, we have altogether 49 array indices to store the coordinates and colors.

a. First array object contains the size of all the other coordinate objects.
b. Two triangles each with 3 vertices are required to create a rectangular surface
c. Each vertex is represented with 8 values (i.e.; 4 position values and 4 color values)
d. Therefore altogether it required 49 ((1 + 3 * 2 * (4 + 4)) data objects to store all the values for the surface.

```
float* data = new float[49];

// The first element stores the number of values
data[0] = 48;
```

```
int i = 1; // index
                        // Positions
                        // A

data[i++] = -halfWidth; // x
data[i++] = 0.0f;       // y
data[i++] = -halfDepth; // z
data[i++] = 1.0f;       // w
// B
data[i++] = -halfWidth;
data[i++] = 0.0f;
data[i++] = halfDepth;
data[i++] = 1.0f;
```

Above code sample shows the value of data element index 0 and corresponding coordinate values A and B. In the square, if the vertices are called A, B, C and D. We have to create triangles that consist of A, B, C and B, C, D vertices. The code employs position first strategy. After all the positions have been defined, the color values are defined.

```
// Colors
// A
data[i++] = 0.0f; // red
data[i++] = 0.4f; // green
data[i++] = 0.0f; // blue
data[i++] = 1.0f; // alpha

// B
data[i++] = 0.0f;
data[i++] = 0.4f;
data[i++] = 0.0f;
data[i++] = 1.0f;
```

Once the coordinates are defined, the drawing object is added in OGLRenderer create method with following code.

```
this->objects["Surface"] = new OGLObject("Surface");
       this->objects["Surface"]->setVertexData(ObjectGenerator::generateXZSurface(10,
20));
data = this->objects["Surface"]->getVertexData();
VBOObject * triangles = OGLObject::createVBOObject("triangles");
triangles->buffer = &data[1];
triangles->primitiveType = GL_TRIANGLES;
triangles->bufferSizeInBytes = (unsigned int)data[0] * sizeof(float);
triangles->numberOfVertices = 6;
triangles->positionComponent.count = 4;
triangles->positionComponent.type = GL_FLOAT;
triangles->positionComponent.bytesToFirst = 0;
triangles->positionComponent.bytesToNext = 4 * sizeof(float);
triangles->colorComponent.count = 4;
triangles->colorComponent.type = GL_FLOAT;
triangles->colorComponent.bytesToFirst = sizeof(float) * 24;
triangles->colorComponent.bytesToNext = 4 * sizeof(float);
this->objects["Surface"]->addVBOObject(triangles);
this->objects["Surface"]->shaderProgram = shaderProgram3d;
```

The constructor also defines the handle for shaderProgram, stockShader. It uses ShaderManager to get the handle.

```
GLuint shaderProgram3d = this->shaderMgr->getShaderHandle("ShaderProgram3d");
GLuint stockShader = this->shaderMgr->getShaderHandle("StockShader");
GLuint shaderProgram3dv3 = this->shaderMgr->getShaderHandle("ShaderProgram3dv3");
```

The create method transforms the local coordinates into the world coordinates. The camera coordinates are transformed to screen coordinates and then the camera position is updated.

```
GLuint localToWorldMatrixUnif = glGetUniformLocation(shaderProgram3d,
"localToWorldMatrix");
	this->cameraToScreenMatrixUnif = glGetUniformLocation(shaderProgram3d,
"cameraToScreenMatrix");

	this->camera.setPosition(15.0f, this->phi, this->theta);
	this->updateCameraPosition();
```

The program then defines how the surfaces are to be interpreted. It sets the vertices to be read to counter-clockwise direction for the back surface. It defines culling for the invisible surface. Similarly, it enables depth testing that checks if the fragments have passed the depth range enabled for the application. The fragments that fail the depth testing are discarded.

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glFrontFace(GL_CCW);

glEnable(GL_DEPTH_TEST);
glDepthMask(GL_TRUE);
glDepthFunc(GL_LEQUAL);
glDepthRange(0.0f, 1.0f);
```

The keyboard keys up, down, left and right are set to call moveCamera functions. E.g., Right key calls OGLRenderer::moveCameraRight function. This behavior is defined in GameWindow::WndProc method.

```
case VK_RIGHT:
	renderer->moveCameraRight();
	break;

void OGLRenderer::moveCameraRight()
{
	this->theta += 5;
	if (this->theta > 360) this->theta -= 360;

	this->camera.setPosition(15.0f, this->phi, this->theta);
	this->updateCameraPosition();
}
```

The OGLRenderer::updateCameraPosition updates the camera position according to the position set up camera.setPosition in respective moveCamera methods.

```
void OGLRenderer::updateCameraPosition()
{
	GLuint shaderProgram3d = this->shaderMgr->getShaderHandle("ShaderProgram3d");
```

```
	GLuint shaderProgram3dv3 = this->shaderMgr->getShaderHandle("ShaderProgram3dv3");

	glUseProgram(shaderProgram3d);
	glUniformMatrix4fv(
		glGetUniformLocation(shaderProgram3d, "worldToCameraMatrix"), 1, GL_FALSE,
glm::value_ptr(this->camera.orientation));
	glUseProgram(0);

	glUseProgram(shaderProgram3dv3);
	glUniformMatrix4fv(
		glGetUniformLocation(shaderProgram3dv3, "worldToCameraMatrix"), 1,
GL_FALSE, glm::value_ptr(this->camera.orientation));
	glUseProgram(0);
}
```

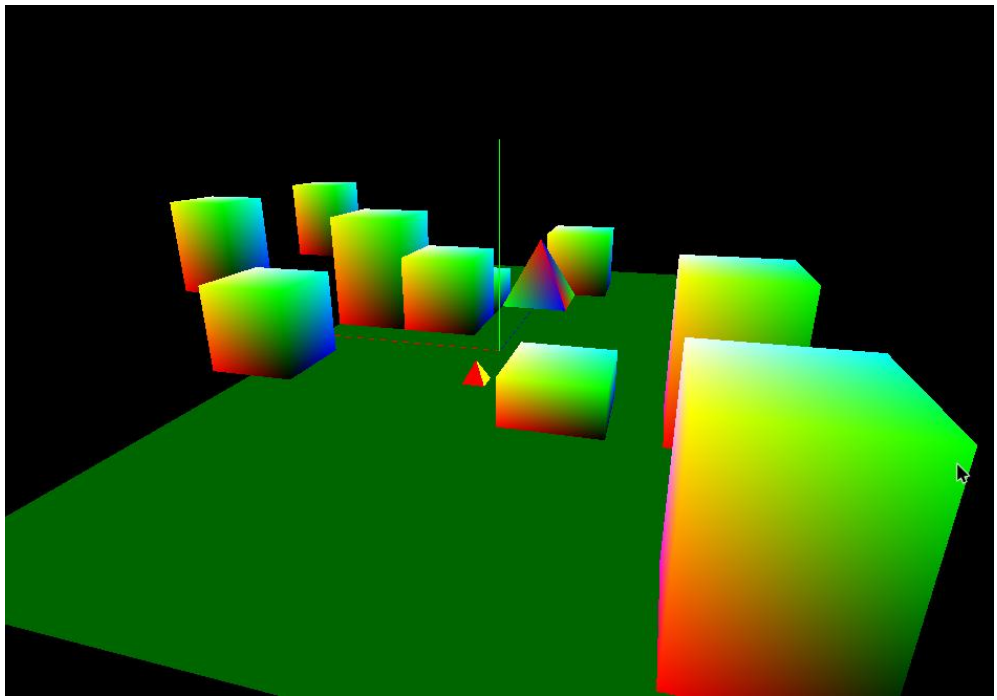
Once again, the coordinates are transformed when selecting the shader program. The rendering is again done by renderObjects method.

```
void OGLRenderer::renderObjects()
{
	GLuint shaderProgram;
	GLuint localToWorldMatrixUnif;
	for (auto iterator = this->objects.begin();
	iterator != this->objects.end();
		iterator++) {
		shaderProgram = iterator->second->shaderProgram;
		localToWorldMatrixUnif = glGetUniformLocation(shaderProgram,
"localToWorldMatrix");
		glUseProgram(shaderProgram);
		glUniformMatrix4fv(
			localToWorldMatrixUnif, 1, GL_FALSE,
			glm::value_ptr(iterator->second->referenceFrame.orientation));
		glUseProgram(0);
		iterator->second->render();
	}

}
```

The method updateViewingFrustum sets the viewing frustum. Objects outside viewing frustum are not visible.

```
void OGLRenderer::updateViewingFrustum(string shaderName, float fov, float aspectRatio,
float zNear, float zFar)
{
	GLuint shaderProgram = this->shaderMgr->getShaderHandle(shaderName);

	glutil::MatrixStack persMatrix;
	persMatrix.Perspective(fov, aspectRatio, zNear, zFar);

	glUseProgram(shaderProgram);
	glUniformMatrix4fv(this->cameraToScreenMatrixUnif, 1, GL_FALSE,
glm::value_ptr(persMatrix.Top()));
	glUseProgram(0);
}
```

The lab ends after creating several 3D objects into the scene. After adding cube, pyramid, surface, lines the final output looks like the following screenshot.



Lab 06 (3D Transforms)

The objective of this lab was to transform 3D objects. The lab uses rotation and translation to add different behaviors to the 3D objects.

To show the effect, first I create a Pyramid class that extends OGL3DObject class. OGL3DObject extends OGLObject, which in fact extends the GameObject class. The GameObject class has behavior attached to it. The Pyramid class's update method calls the Behavior class's update method with elapsedTime to update the behavior.

To create a behavior I extend the Behavior class and override the update method. Although the method passes elapsedTime it does not use it. This parameter will be used in future labs to create animations. As an example of creating a behavior, I create a class called LoopyBehavior that extends the Behavior class and overrides the update method as shown in follow code sample.

```
void LoopyBehavior::update(GameObject *object, float elapsedTimeMS)
{
	OGL3DObject *theObject = (OGL3DObject *)object;
	theObject->referenceFrame.rotateY(this->speed);
	theObject->referenceFrame.moveForward(this->distance);
}
```

As can be seen from above code, a LoopyBehavior consists of rotation in the y-axis while moving forward. The behavior is applied to the designated object while it is being created in the loadObjects method of the StockObjectLoader class.

```
	for (int i = 0; i < 10; i++) {
		float height = rand() % 3 + 1.0f;
		float x = rand() % 20 - 10.0f;
		float z = rand() % 20 - 10.0f;
		string name = "Pyramid" + std::to_string(i + 3);

		object = new Pyramid(name);

		float range = (float)(rand() % 101)/100;
		float range1 = (float(rand() % 5) / 10 + 0.1);

		//object->setBehavior(new RotateYBehavior(range));
		object->setBehavior(new LoopyBehavior(range1, 0.01));
		//object->setBehavior(new CustomBehavior(range1, 0.01));

		object->setVertexData(ObjectGenerator::generatePyramid(2, 2, height));
		data = object->getVertexData();
		triangles = OGLObject::createVBOObject("triangles");
		triangles->buffer = &data[1];
		triangles->primitiveType = GL_TRIANGLES;
		triangles->bufferSizeInBytes = (unsigned int)data[0] * sizeof(float);
		triangles->numberOfVertices = 18;
		triangles->positionComponent.count = 4;
		triangles->positionComponent.type = GL_FLOAT;
		triangles->positionComponent.bytesToFirst = 0;
		triangles->positionComponent.bytesToNext = 4 * sizeof(float);
		triangles->colorComponent.count = 4;
		triangles->colorComponent.type = GL_FLOAT;
```

```
            triangles->colorComponent.bytesToFirst = sizeof(float) * 72;
            triangles->colorComponent.bytesToNext = 4 * sizeof(float);
            object->addVBOObject(triangles);
            object->referenceFrame.setPosition(x, 0.0f, z);

            gameObjectManager->addObject(name, object);
      }
```

The sample code creates 10 pyramids with random size and random time for the behavior (loopy behavior in this case. Also, if we look at the commented section there are other behaviors defined as well. For the above code LoopyBehavior is set as current behavior.

This lab also introduces code for changing camera orientation. Camera orientation is changed based on rotation (done by pressing arrow keys on the keyboard) along the spherical surface around the scene. The updateOrientation method of OGLSphericalCamera defines code for updating the orientation of the camera.

```
void OGLSphericalCamera::updateOrientation()
{
      this->cameraSpherical.rho = this->rho;
      // phi = 0 and theta = 0 should look down the z-axis
      this->cameraSpherical.phi = DegToRad(this->phiDegrees);
      this->cameraSpherical.theta = DegToRad(this->thetaDegrees);


      // See
http://en.wikipedia.org/wiki/Spherical_coordinate_system#Cartesian_coordinates
      // Spherical to Cartesian
      // x = r * sin(theta) * sin(phi)
      // y = r * cos(phi)
      // z = r * cos(theta) * sin(phi)

      float fSinTheta = sinf(this->cameraSpherical.theta);
      float fCosTheta = cosf(this->cameraSpherical.theta);
      float fCosPhi = cosf(this->cameraSpherical.phi);
      float fSinPhi = sinf(this->cameraSpherical.phi);

      glm::vec3 dirToCamera(fSinTheta * fSinPhi, fCosPhi, fCosTheta * fSinPhi);
      glm::vec3 position = (dirToCamera * cameraSpherical.rho) + this->target;

      glm::vec3 lookDir = glm::normalize(this->target - position);

      glm::vec3 upDir = glm::normalize(this->up);

      glm::vec3 rightDir = glm::normalize(glm::cross(lookDir, upDir));
      glm::vec3 perpUpDir = glm::cross(rightDir, lookDir);

      glm::mat4 rotMat(1.0f);
      rotMat[0] = glm::vec4(rightDir, 0.0f);
      rotMat[1] = glm::vec4(perpUpDir, 0.0f);
      rotMat[2] = glm::vec4(-lookDir, 0.0f); // Negate the look direction to point at
the look point

      // Since we are building a world-to-camera matrix, we invert the matrix.
      // Transposing a rotation matrix inverts it!
      rotMat = glm::transpose(rotMat);
```

```
        // Translate the camera to the origin
        glm::mat4 transMat(1.0f);
        transMat[3] = glm::vec4(-position, 1.0f);

        // Translate and then rotate
        this->orientation = rotMat * transMat;
}
```

For rotation, degrees are first transformed into radians and converted using the rotation matrix. E.g.,

Rotation matrix around y-axis is as follows.

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

The code sample that uses above transformation is shown in following snippet.

```
void OGLReferenceFrame::rotateY(float degrees)
{
        float radians = DegToRad(degrees);
        float cosTheta = cosf(radians);
        float sinTheta = sinf(radians);

        glm::mat4 rotYMat(1.0f);

        rotYMat[0] = glm::vec4(cosTheta, 0.0f, -sinTheta, 0.0f);
        rotYMat[1] = glm::vec4(0.0f, 1.0f, 0.0f, 0.0f);
        rotYMat[2] = glm::vec4(sinTheta, 0.0, cosTheta, 0.0f);

        this->orientation *= rotYMat;
}
```

Similarly, rotation around x-axis and is as follows.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

And, rotation around z-axis is as follows.

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If instead of adding the behavior, if the Pyramid object wanted to set rotation in z-axis it could call the rotateZ method of OGLReferenceFrame's class from its update method.

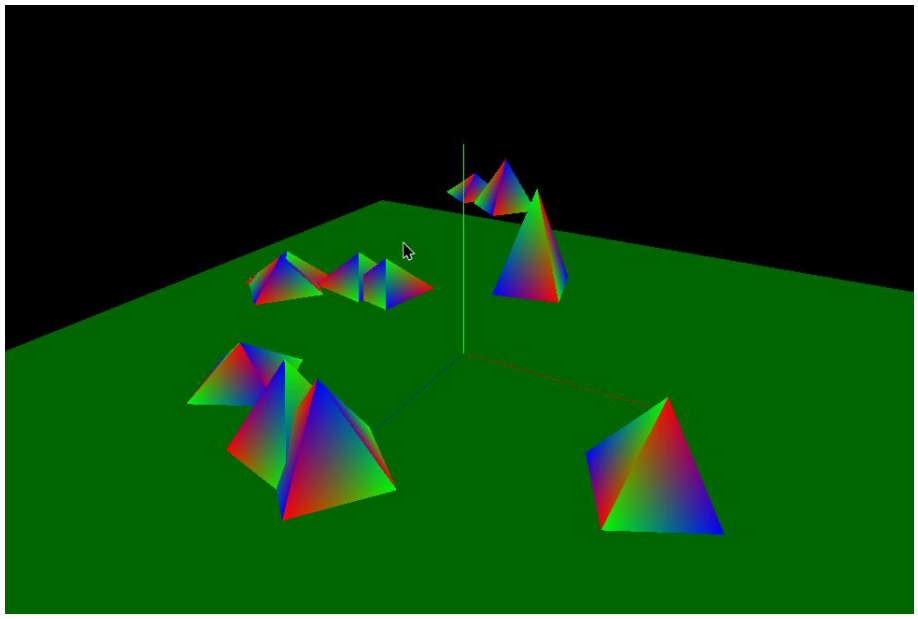
```
this->referenceFrame.rotateZ(0.1f);
```

To move an object, OGLReferenceFrame class also defines a move method. This method is used by various other move methods to move the object in different orientations.

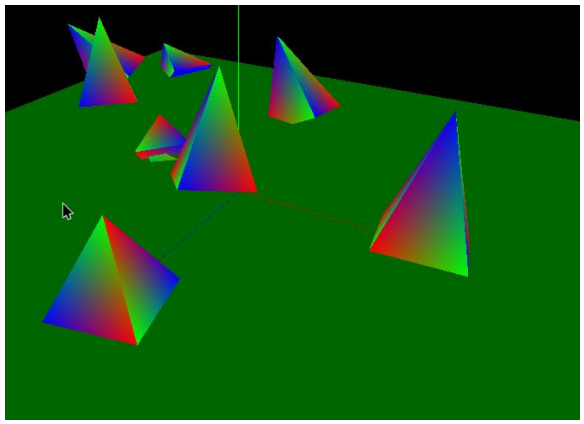
```
void OGLReferenceFrame::move(const glm::vec3& direction, float speed)
{
	glm::mat4 translateMat(1.0f);
	translateMat[3] = glm::vec4(direction * speed, 1.0f);
	this->orientation *= translateMat;
}

void OGLReferenceFrame::moveForward(float speed)
{
	// The 3rd column is the z-axis
	this->move(glm::vec3(this->orientation[2]), speed);
}
```

With movement and rotation we can add several behaviors to the 3D object. As defined above the LoopyBehavior is added to all the pyramids drawn in the scene. As before, the game objects are added using GameObjectManagement class as map with keys and corresponding values.

When we run the application with LoopyBehavior as the active Behavior for the Pyramid objects it looks like the following screenshot.



In this Lab I also created a custom behavior. I use rotateZ with cosine of random angle, rotateY with sine of random angle and moveForward with 0.001 as the speed to create the behavior. The output is as the following screenshot.



Lab 07 (Basic Animation)

The objective of this lab was to add simple animation to the 3D objects.

In this lab, the GameEngine class's run method uses the WindowsTimer for animation. GameEngine::setupGame method also sets the animation behavior to the OGLObject objects.

Following code sets BackFourthBehavior for the pyramid object.

```
object = (OGLObject*)
            this->graphics->getGameWorld()->getObjectManager()->getObject("Red
Pyramid");
object->referenceFrame.rotateY(45.0f);
object->setBehavior(new BackForthBehavior(15.0f));
```

The Behavior class adds the logic by setting a maxDistance or maxRotation. The code uses these variables based on the nature of the behavior. The BackForthBehavior class create a behavior by creating a loop animation by using different States. The movement loop is created by transitioning between the MOVING_FORWARD and MOVING_BACKWARD enum variables. Following code sample shows the behavior of this finite state machine.

```
enum State { MOVING_FORWARD, MOVING_BACKWARD };
```

The update method of BackFourthBehavior class adds following logic for the animation.

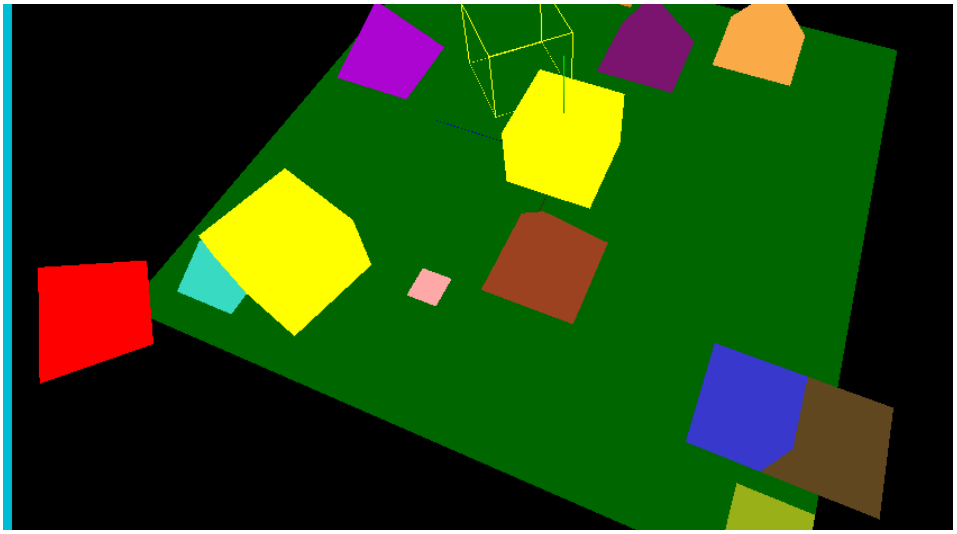
```
void BackForthBehavior::update(GameObject *object, float elapsedSeconds)
{
	OGLObject* obj = (OGLObject*)object;
	float delta = 10.0f * elapsedSeconds;
	this->distanceMoved += delta;
	switch (this->state) {
	case MOVING_BACKWARD:
		if (this->distanceMoved >= this->maxDistance) {
			this->state = MOVING_FORWARD;
			delta = this->distanceMoved - this->maxDistance;
			this->distanceMoved = 0;
		}
		obj->referenceFrame.moveBackward(delta);
		break;
	case MOVING_FORWARD:
		if (this->distanceMoved >= this->maxDistance) {
			this->state = MOVING_BACKWARD;
			delta = this->distanceMoved - this->maxDistance;
			this->distanceMoved = 0;
		}
		obj->referenceFrame.moveForward(delta);
		break;
	}
}
```

As can be seen from above code if the object is moving forward (set in the constructor) and if the distance moved by the object is greater than the max distance set for the object, the State is switched and the object

starts moving in the backward direction. This cycle continues to create the animation. During the lab, I also created other behaviors like FourPointBehavior and PatrolBehavior. In the FourPointBehavior, the object first moves forward and once it reaches the distance defined by maxDistance variable, it switches to MOVING_LEFT state. It moves to the left until it again reaches the maxDistance values, and then switches to MOVING_FORWARD. Again after reaching the maxDistance value in MOVING_FORWARD state, it switches to MOVING_RIGHT. From MOVING_RIGHT it switches to MOVING_BACKWARD and finally reaches the original position. All of these movements create a loop and thus the object keeps on following the same behavior to define the animation. If we observe the overall behavior, a square movement animation is created using the four states and corresponding methods in the OGLReferenceFrame object. The code sample that defines all the movement is listed below. Also, as mentioned before all the movement methods (moveForward, moveBackward, moveRight and moveLeft) use the move method.

```
oid OGLReferenceFrame::move(const glm::vec3& direction, float speed)
{
	glm::mat4 translateMat(0.0f);
	translateMat[3] = glm::vec4(direction * speed, 0.0f);

	this->orientation += translateMat;
}

void OGLReferenceFrame::moveForward(float speed)
{
	// The 3rd column is the z-axis
	this->move(glm::vec3(this->orientation[2]), speed);
}

void OGLReferenceFrame::moveBackward(float speed)
{
	// The 3rd column is the z-axis
	this->move(glm::vec3(-this->orientation[2]), speed);
}

void OGLReferenceFrame::moveRight(float speed)
{
	// The 1st column is the x-axis
	this->move(glm::vec3(-this->orientation[0]), speed);
}


void OGLReferenceFrame::moveLeft(float speed)
{
	// The 1st column is the x-axis
	this->move(glm::vec3(this->orientation[0]), speed);
}
```

Another interesting code of the lab was the addition of mouse movement. The PCInputSystem class adds the mouse and keyboard key behaviors. The screen follows the mouse movement and additionally with keyboard up, down, right and left keys the scene can be moved to the corresponding directions. The final output can be seen in following screenshot.



This lab added the experience of simple animation to the graphics objects. It also introduced mouse movement and used high speed windows timer for animation.

Project 1

The objective of this project was to externalize some of the game window's attributes and use the Configuration component to read the configuration data.

The Configuration class contains several parameters like title, size, position and color values. The Configuration receives filename as parameter for the constructor, which it uses to open the file and read its values. The read values are then set while creating the window setting OpenGL background properties. All of the window creation and OpenGL initialization code is written in main.cpp. The Configuration class only contains necessary information to get and set values from the config file.

```
class Configuration {
private:
	string title;
	int startX;
	int startY;
	int width;
	int height;
	float colorRed;
	float colorGreen;
	float colorBlue;

	void parseConfigFile(string filename, map<string, string> &fileMap);

public:
	Configuration(string filename);
	string getTitle() { return title; }
	int getStartX() { return startX; }
	int getStartY() { return startY; }
	int getWidth() { return width; }
	int getHeight() { return height; }
	float getColorRed() { return colorRed; }
	float getColorGreen() { return colorGreen; }
	float getColorBlue() { return colorBlue; }
};
```

The class defines above mentioned variables and adds accessor and mutators to manipulate those variables. The parseConfigFile method reads the configuration file called window.config that contains values for above mentioned variables.

```
title : Game Window
startX : 80
startY : 80
width : 400
height : 600
colorRed : 0.6
colorGreen : 0.2
colorBlue : 0.1
```

The parseConfigFile method uses ifstream object to read in file contents and add it as key value to fileMap map object. The parseConfigFile also uses external file to trim spaces around the text before and after splitting the text using ":" character as separator.

```
void Configuration::parseConfigFile(string filename, map<string, string> &fileMap)
{
	ifstream fin;
	fin.open(filename);

	if (fin.fail()) {
		cout << "\nThere was error opening the file." << endl;
		exit(1);
	}

	string line;
	string key;
	string value;

	// Debug
	std::ofstream log;
	log.open("log.txt");

	while (!fin.eof()) {
		getline(fin, line);
		int colonPosition = line.find(":");
		key = trim_copy(line.substr(0, colonPosition));
		//value = line.substr(colonPosition + 1);
		value = trim_copy(line.substr(colonPosition + 1));
		fileMap[key] = value;
	}

	fin.close();

	// Debug
	if (debug) {
		for (map<string, string>::const_iterator it = fileMap.begin(); it != 
fileMap.end(); ++it) {
			log << it->first << ":" << it->second << endl;
		}
	}


	log.close();
}
```

Since the read string is character values, it has to be converted to integer and float for proper processing. The code uses std::stoi and std::stof methods to convert character array to integer and float values.

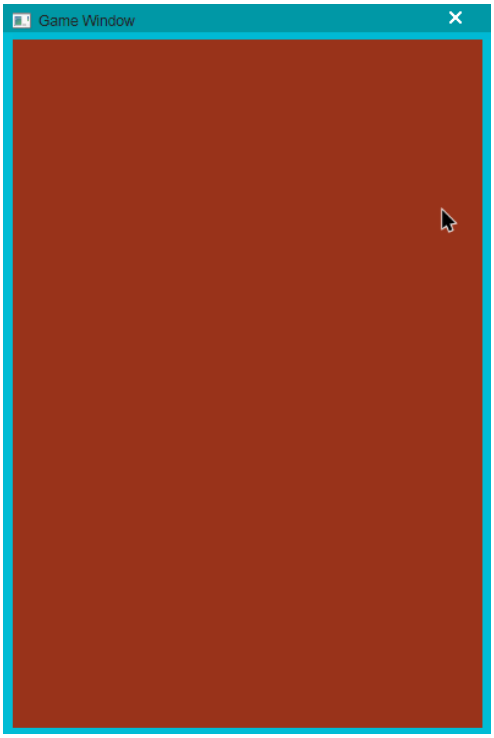
```
title = fileMap["title"];
startX = std::stoi(fileMap["startX"]);
startY = std::stoi(fileMap["startY"]);
width = std::stoi(fileMap["width"]);
height = std::stoi(fileMap["height"]);
colorRed = std::stof(fileMap["colorRed"]);
colorGreen = std::stof(fileMap["colorGreen"]);
colorBlue = std::stof(fileMap["colorBlue"]);
```

In the main method the accessor methods of Configuration class are used to get the values for the required attributes while constructing the window.

```
window.hWnd = CreateWindowEx(
              WS_EX_APPWINDOW | WS_EX_WINDOWEDGE,
               title.c_str(),
              title.c_str(),
              WS_OVERLAPPEDWINDOW,
              configuration->getStartX(),
              configuration->getStartY(),
              configuration->getWidth(),
              configuration->getHeight(),
              NULL,
              NULL,
              window.hInstance,
              NULL
              );


// Color
glClearColor(configuration->getColorRed(),
configuration->getColorGreen(),
configuration->getColorBlue(),
0.0f);
```

The color is set in WinMain method after getting device context for OpenGL. The CreateOGLWindow class creates the OpenGL context using the wglCreateContext method from wingdi windows api. The OpenGL context is then set using the Context struct and returned. The CreateOGLWindow class defines several parameters to PIXELFORMATDESCRIPTOR to define the pixel such as color bits, color depth. Using the ChoosePixelFormat and SetPixelFormat windows api function it sets the pixel for the opengl device context.

After executing the code, the final output looks like following screenshot.

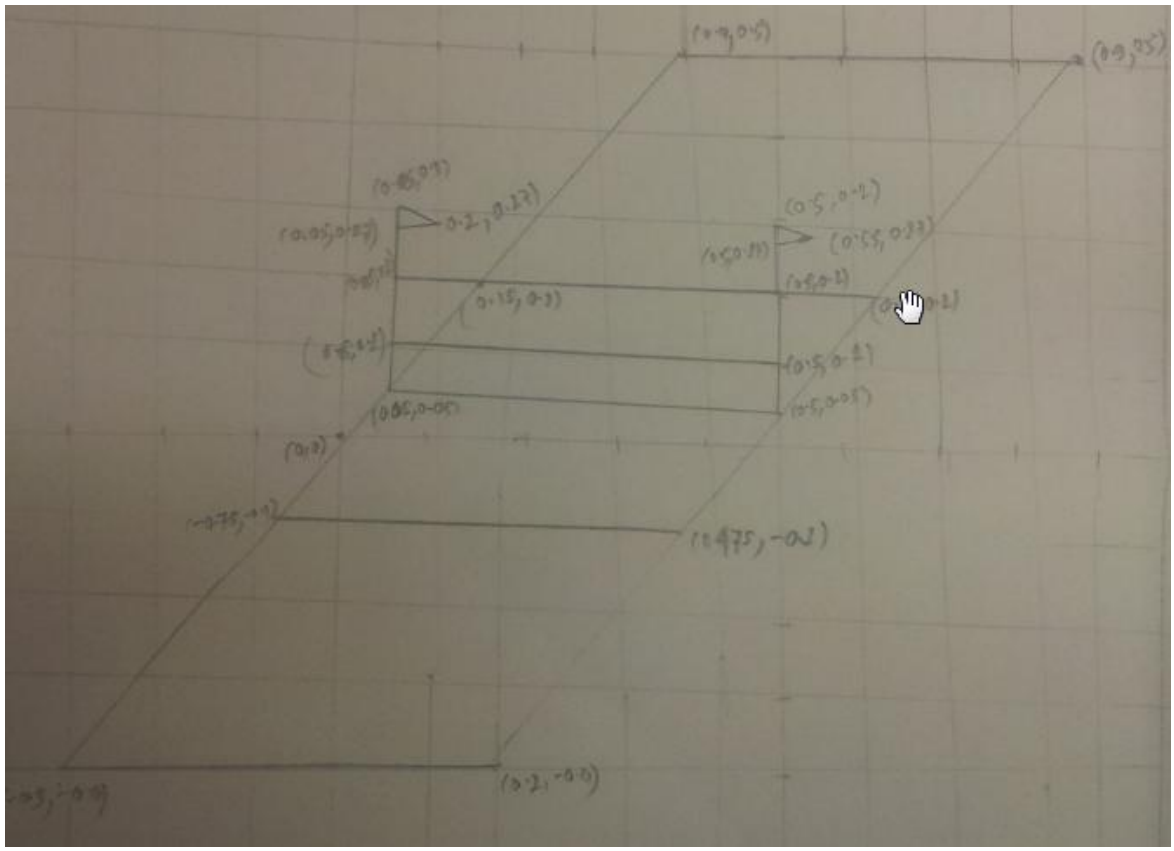


Project 2

The objective of Project 2 was to design a moderately complex 2D scene centered on the origin. The scene has to be fitted within the dimension of the rectangular aread defined between -1 and 1 in X and Y axes. Although the vertices used three coordinates the Z coordinate was set to 0 to draw the objects as 2D objects.

Part 1 (Proposal)

The first set of work for the project was to propose the scene by drawing the scene manually in a paint program or on paper. I drew the diagram of a tennis court in paper.



I also proposed the coordinates as shown in the diagram above in a text file with X,Y,Z coordinate system.

Point 1
(0.1, 0.8, 0.0) # Point
(1.0, 0.0, 0.0) # Color red

Point 2 # Color green
(0.2, 0.8, 0.0)
(0.0, 1.0, 0.0)

Part 2 (Drawing 2D Scene)

The objective of the Second part of the project was to draw the scene using OpenGL drawing functions. The project also required the OpenGL configuration to be stored externally in a configuration file. Additionally, data for the scene was to be stored in a data file.

I used Lab 2 as the basis for creating this project as Lab 2 implemented the changing background color functionality. I then used parts of project 1 to read Configuration from externally stored configuration file.

I create the scene using OGLRenderer class's create method. In the scene I added two points and created the court. The whole court scene is a surface with two poles in the center, a net and two flags. The court also contains the line markers. I used triangles to create the court surface, lines to draw lines in the court. I also used lines to define poles in the center of the court. I created the net using two triangles. I also used triangles to create two flags, that is attached at the top of the poles.

Since the requirement of the project was to externalize the coordinates and colors, the vertex data is read from drawing.dat file. The values are read using ReadGraphicsFile method of the Core class. The ReadGraphicsFile method reads the text in the date file using ifstream and sets the corresponding values in the Vertex struct. It then adds the Vertex object in a vector.

In the OGLRenderer class's createUI method, the coordinates are read into siVertexData vector of type Vertex. The vector data can be accessed using indices just like accessing data from an array. Following is the sample code for creating two points defined in createUI method.

```
core.ReadGraphicsFile(GRAPHICS_FILE, siVertexData);
int vertexSize = 6 * sizeof(GLfloat);

// Create two points (2 vertices)
VBOObject * points = OGLObject::createVBOObject("points");
points->buffer = &siVertexData[0]; // Starting at 7th line
points->primitiveType = GL_POINTS;
points->bufferSizeInBytes = 2 * vertexSize;
points->numberOfVertices = 2;
points->positionComponent.count = 3;
points->positionComponent.type = GL_FLOAT;
points->positionComponent.bytesToFirst = 0;
points->positionComponent.bytesToNext = sizeof(Vertex);
points->colorComponent.count = 3;
points->colorComponent.type = GL_FLOAT;
points->colorComponent.bytesToFirst = sizeof(GLfloat) * 3;
points->colorComponent.bytesToNext = sizeof(Vertex);
this->objects["points"]->addVBOObject(points);
```

The changing background is defined in GameWindow update method. I set the background to change from black to blue and back to black. I also made sure that the changing background did not interfere with the colors used for any of the drawn objects. Therefore I selected all the object colors that did not fall between black and blue.

The final output looks like the following screenshot.