

Bartosz_Galantowicz_1700994_ ACCA5017_Games_Architecture _&_Mechanics_Assig

by Bartosz Galantowicz

FILE	2275_BARTOSZ_GALANTOWICZ_BARTOSZ_GALANTOWICZ_1700994_AC CA5017_GAMES_ARCHITECTURE__MECHANICS_ASSIG_254066_792703 506.DOCX (1.48M)		
TIME SUBMITTED	07-MAY-2020 05:36PM (UTC+0100)	WORD COUNT	1949
SUBMISSION ID	126658033	CHARACTER COUNT	9883

Bartosz Galantowicz

1700994

Part 1b

Games Architecture and Mechanics

ACCA5017

Gordon Dickers

1 ABSTRACT

The report aims to research features and patterns used in developing games engines. Moreover, the studies are based on given game engine prototype, which can be extended and modified. Hence, the various programming approaches have been incorporated into the prototype, to analyse how game engines are built and how they work.

2 CONTENTS

Abstract.....	2
Introduction	3
Patterns Implementation.....	3
Singleton Pattern	3
Timer	4
Game Loop Pattern	6
Command Pattern	8
Bitmap loader.....	12
Adding normals and texture coordinates	15
Bibliography	17
Appendix	18
Test Cases.....	18

3 INTRODUCTION

The given prototype game engine has been programmed in C/C++ language. Therefore, the same language was applied to achieve the designated goals of the assignment. Moreover, the SDL library was incorporated to provide low-level access to audio, input devices and graphic hardware via OpenGL.

Due to the research on games design patterns, the following examples have been studied:

- Singleton Pattern
- Timer
- Game Loop Pattern
- Command Pattern

Additionally, integration of the object loader tool and the game engine was required to read 3D meshes and modifying them. Also, it was managed to code a simple bitmap file loader, which is responsible for generating textures.

4 PATTERNS IMPLEMENTATION

4.1 SINGLETON PATTERN

Singleton pattern is used when a single instance of the object is wanted over a full period of a program work time. Thus, one class based on the singleton pattern can be applied to coordinate events and actions across the program functionality. The idea for the program is to make sure if there is not any other object of that instance. Otherwise, the instance is deleted.

In games development, the singleton pattern is a handful to organises a game structure. Due to avoid incorrect behaviours of the software, objects like a timer or input manager should be initialised only once by an application. Moreover, they must be accessible globally in the program. Thereby the mentioned examples are perfect representations of singleton pattern implementation.

```

17 // Initialise the instance to NULL
18 timer* timer::sInstance = NULL;
19
20 // Use singleton pattern for the timer
21 timer * timer::instance()
22 {
23     // Declare the instance for the timer if its NULL
24     if (sInstance == NULL)
25     {
26         sInstance = new timer();
27     }
28
29     return sInstance;
30 }

```

```

13 // Initialise the instance to NULL
14 inputManager *inputManager::sInstance = NULL;
15
16 // Use singleton pattern for the input manager
17 inputManager * inputManager::instance()
18 {
19     // Declare the instance for the input manager if its NULL
20     if (sInstance == NULL)
21     {
22         sInstance = new inputManager();
23     }
24
25     return sInstance;
26 }

```

Fig.1 The examples of using the singleton pattern in the code.

4.2 TIMER

The timer is the object of the class, which is in charge of keeping track of time in a computer program. There should be one instance of this object when it is used in runtime. Furthermore, it is the base for other patterns in games programming. Allowing to obtain framerate independence and to manage the time are the vast advantages of this method. Therefore, by calculation of time between frames, it is possible to run the program with the same speed on devices with various hardware specification.

The implemented timer for the purpose of the assignment is simple, and its main role is to count the time. The start of this process is possible due to `SDL_GetTicks()` method, which returns the number of milliseconds since the SDL library initialisation. Furthermore, arranging framerate independence is allowed by the computation of elapsed ticks and the difference between frames.

```
// Initialise the timer class
timer *timer = timer::instance(); // Timer class is responsible for manage the time
```

```
// Application loop - keep working when is not game over
while (!gameOver)
{
    // Update the timer at the beginning of each iteration
    timer->update();

    // Keep framerate independency
    while (timer->getDeltaTime() >= 1.0f / FRAME_RATE)
    {
        // Process input
        inputManager->update();
        inputManager->move(pmesh);
        gameOver = inputManager->gameOver(); // Check if it is game over (press esc to exit the loop)

        // Update

        // Update game objects
        for (size_t i = 0; i < 15; i++)
        {
            gameObjects[i].update(i+1, timer->getLag());
        }

        myDisplay.update();

        std::cout << "delta time: " << timer->getDeltaTime() << std::endl; // Display delta time
        std::cout << "FPS: " << (1.0f/timer->getDeltaTime()) << std::endl; // Display FPS

        // Restart timer
        timer->reset();
    }

    // Render
    myDisplay.render(pmesh);
}
```

```
FPS: 58.8235
delta time: 0.017
FPS: 58.8235
delta time: 0.017
FPS: 58.8235
delta time: 0.017
FPS: 58.8235
delta time: 0.017
```

Fig.2 The example of using the timer by the main loop of the program.

Consequently, the properties of the timer are applied to the main game loop. If the delta time is equal or bigger than the time needed to generate one frame, then the update methods may be called.

```
// By multiplying by deltaTime the movement is framerate independent
if (keyDown(SDL_SCANCODE_W))
{
    std::cout << "W has been pressed" << std::endl;
    pmesh->translate(0, 1 * timer->getDeltaTime(), 0);
}
```

Fig.3 Delta time can be used to set framerate independence.

Delta time can also be used in other patterns. For instance, by multiplying the translate value of y by delta time, the growth in y will be constant.

4.3 GAME LOOP PATTERN

Game loop pattern, also called the main loop, is fundamental for each game program. The idea is that the game is kept running by a continuous controlled loop until a quit is decided by a user. The structure of a game loop is:

- Process input
- Update states
- Render objects
- Repeat the above at a provided rate

The timer pattern can be used in the main loop, which results in framerates independence in the program. Namely, the elapsed time is taken into consideration by the loop to perform input, update and render methods over a specified amount of time. The program was set to display 60 frames per second.

```
// Application loop - keep working when is not game over
while (!gameOver)
{
    // Update the timer at the beginning of each iteration
    timer->update();

    // Keep framerate independency
    while (timer->getDeltaTime() >= 1.0f / FRAME_RATE)
    {
        // Process input
        inputManager->update();
        inputManager->move(pmesh);
        gameOver = inputManager->gameOver(); // Check if it is game over (press esc to exit the loop)

        // Update

        // Update game objects
        for (size_t i = 0; i < 15; i++)
        {
            gameObjects[i].update(i+1, timer->getLag());
        }

        myDisplay.update();

        std::cout << "delta time: " << timer->getDeltaTime() << std::endl; // Display delta time
        std::cout << "FPS: " << (1.0f / timer->getDeltaTime()) << std::endl; // Display FPS

        // Restart timer
        timer->reset();
    }

    // Render
    myDisplay.render(pmesh);
}
```

Fig.4 The main loop of the program.

The work of the coded game loop is dependent on the value in `gameOver` variable, which can be modified by the input of a user. Inside the loop, the timer is updated, and the delta time value checked that if it is true, the proceeding input begins. In result, the appropriate methods are updated. While all updates are done, the timer is reset, and the render functions can be called.

```
delta time: 0.017  
FPS: 58.8235
```

Fig.5 The program is set to display approximately 60 frames per second.

Testing the game loop was carried out by modification a mesh object during the work of the program. The mesh object has a form of a 3D cube model, and it is displayed in the application window. Nevertheless, the object can be manipulated by taking input from the user and updating its position on the screen.

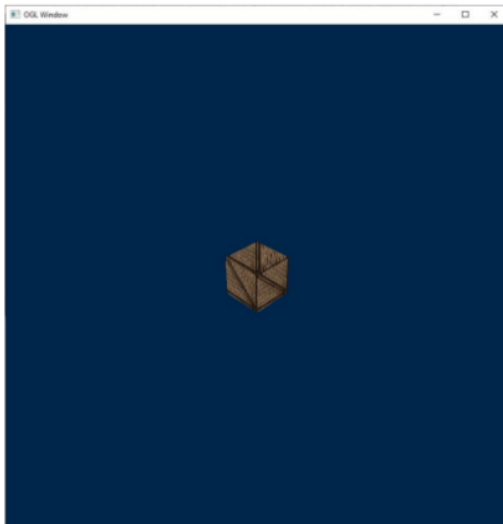


Fig.6 The example of a rendered object.

Another developed test is performed by updating multiple game objects in the game loop. The time of the update of a single object is displayed as a result of this operation.

```
// Creating game objects to perform the update test  
gameObject *gameObjects = new gameObject[15];
```

```
// Update game objects  
for (size_t i = 0; i < 15; i++)  
{  
    ...  
    gameObjects[i].update(i+1, timer->getLag());  
}
```

```

delta time: 0.017
FPS: 58.8235
Game object 1 has been updated in 1 ms
Game object 2 has been updated in 1 ms
Game object 3 has been updated in 1 ms
Game object 4 has been updated in 1 ms
Game object 5 has been updated in 1 ms
Game object 6 has been updated in 1 ms
Game object 7 has been updated in 1.05882 ms
Game object 8 has been updated in 1.05882 ms
Game object 9 has been updated in 1.05882 ms
Game object 10 has been updated in 1.11765 ms
Game object 11 has been updated in 1.11765 ms
Game object 12 has been updated in 1.11765 ms
Game object 13 has been updated in 1.17647 ms
Game object 14 has been updated in 1.17647 ms
Game object 15 has been updated in 1.17647 ms

```

Fig.7 Creating game objects in the program and testing the time of their updates.

4.4 COMMAND PATTERN

The command pattern is designed to process requests by binding relevant actions. The example of using the command pattern is the implementation of a method which would allow for processing input from a user. Therefore, meaningful actions can be performed after pressing a button, which is assigned to the appropriate function.

```

48 // Process input
49 inputManager->update();
50 inputManager->move(pmesh);
51 gameOver = inputManager->gameOver(); // Check if it is game over (press esc to exit the loop)
52

```

Fig.8 The example of the process input implementation.

Furthermore, the simple translate() function has been added to existed mesh class to test the functionality of the object by transforming its position.

```

void Mesh::translate(float x, float y, float z)
{
    for (int i = 0; i < vertexCount; i++)
    {
        pVertices[i].pos.x += x;
        pVertices[i].pos.y += y;
        pVertices[i].pos.z += z;
    }
}

```

Fig.9 The translate() method is used to move a rendered object.

The command pattern approach has been incorporated into the game engine in the form of the simple input manager class, which the main task is to handle user input. This is

accomplished by identifying the location of a pressed key, according to SDL scancode. Thus, the two basic methods have been programmed based on this pattern. This allows moving the object by pressing W, A, S, D keys and using ESCAPE key to stop the game loop and exit the program. Moreover, the ability to moving an object in the game engine could be a tool to test its functionality.

```
63 void inputManager::move(Mesh *pmesh)
64 {
65     timer *timer = timer::instance();
66
67     // By multiplying by deltaTime the movement is framerate independent
68     if (keyDown(SDL_SCANCODE_W))
69     {
70         std::cout << "W has been pressed" << std::endl;
71         pmesh->translate(0, 1 * timer->getDeltaTime(), 0);
72     }
73
74     if (keyDown(SDL_SCANCODE_A))
75     {
76         std::cout << "A has been pressed" << std::endl;
77         pmesh->translate(-1 * timer->getDeltaTime(), 0, 0);
78     }
79
80     if (keyDown(SDL_SCANCODE_S))
81     {
82         std::cout << "S has been pressed" << std::endl;
83         pmesh->translate(0, -1 * timer->getDeltaTime(), 0);
84     }
85
86     if (keyDown(SDL_SCANCODE_D))
87     {
88         std::cout << "D has been pressed" << std::endl;
89         pmesh->translate(1 * timer->getDeltaTime(), 0, 0);
90     }
91 }

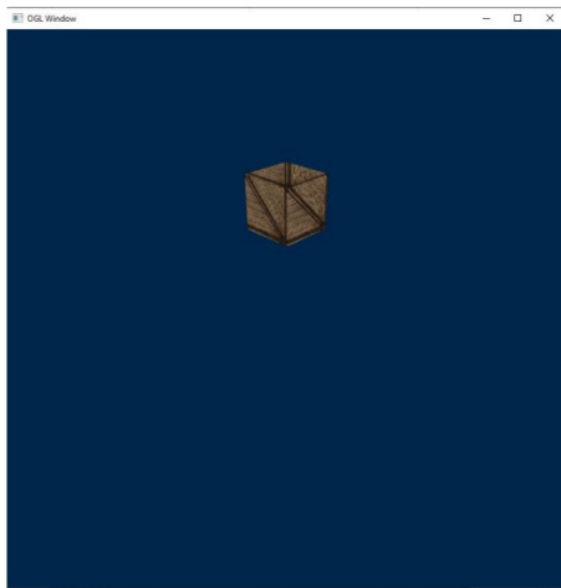
```

```
48 // Return true if the escape button has been pressed
49 bool inputManager::gameOver()
50 {
51     if (keyDown(SDL_SCANCODE_ESCAPE))
52     {
53         std::cout << "ESCAPE has been pressed" << std::endl;
54         return true;
55     }
56     else
57     {
58         return false;
59     }
60 }

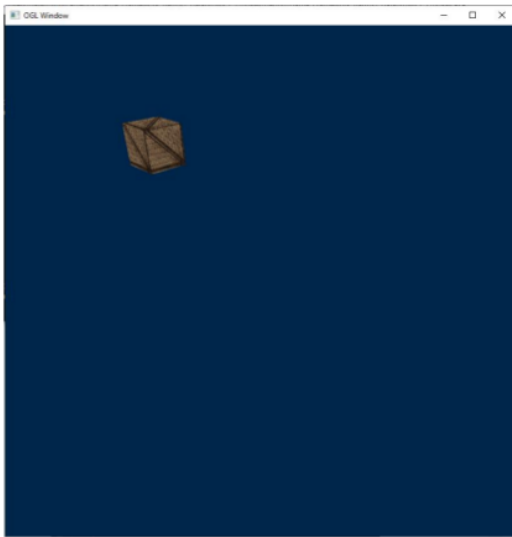
```

Fig.10 The input manager is responsible for reading input, which is obtained by the program to move rendered objects.

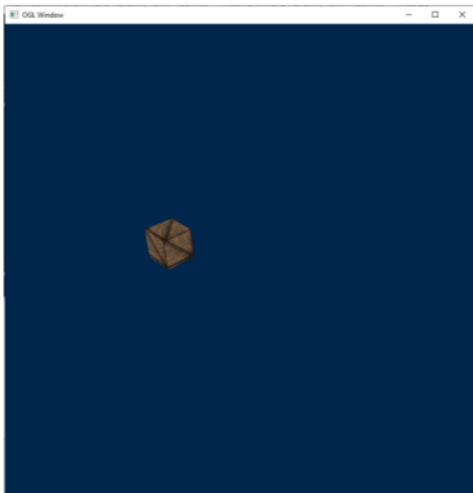
```
W has been pressed  
delta time: 0.017  
FPS: 58.8235  
W has been pressed  
delta time: 0.017  
FPS: 58.8235  
W has been pressed  
event number:769  
delta time: 0.017  
FPS: 58.8235  
delta time: 0.017  
FPS: 58.8235
```



```
A has been pressed  
delta time: 0.017  
FPS: 58.8235  
A has been pressed  
delta time: 0.017  
FPS: 58.8235  
A has been pressed  
delta time: 0.017  
FPS: 58.8235  
A has been pressed  
delta time: 0.017  
FPS: 58.8235
```



```
S has been pressed  
delta time: 0.017  
FPS: 58.8235  
S has been pressed  
delta time: 0.017  
FPS: 58.8235  
S has been pressed  
delta time: 0.017  
FPS: 58.8235  
S has been pressed  
event number:1024
```



```
D has been pressed
delta time: 0.017
FPS: 58.8235
D has been pressed
delta time: 0.017
FPS: 58.8235
D has been pressed
delta time: 0.017
FPS: 58.8235
D has been pressed
delta time: 0.017
FPS: 58.8235
```

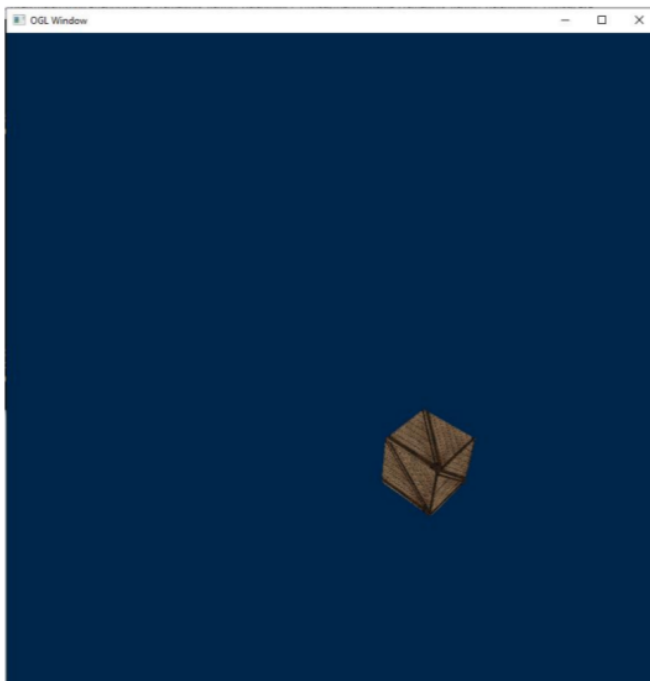


Fig.11 The examples of moving the rendered object in various directions.

5 BITMAP LOADER

It was tried to program a bitmap file loader to the application. In result, the simple texture loader was created for the mesh class. Thereby, it is allowed to initialise a mesh by providing the directory of the bitmap file.

The texture is loaded from a provided .bmp file and bind to a variable, which can be used to map the textures. Thereby, the draw function in the mesh class has been modified to drawing the textured object.

```

4  Mesh::Mesh(const char *objFile, const char *bmpFile)
5  {
6      myobj.objloader(objFile);
7      objTextCoord();
8      objPoly2Vertex();
9      loadTexture(bmpFile);
10 }
11
12
13 Mesh::Mesh(const char *objFile)
14 {
15     myobj.objloader(objFile);
16     objTextCoord();
17     objPoly2Vertex();
18 }
19
20 Mesh::Mesh()
21 {
22     myobj.objloader("cube.obj");
23     objTextCoord();
24     objPoly2Vertex();
25 }

```

Fig.12 Various constructors have been created to allow a user to manipulate in generating meshes. The specific data can be set after providing object file data and relevant bitmap data. Also, if the argument is void type, the default object is set to "cube" object file.

```

49 void Mesh::draw()
50 {
51     //glBindVertexArray(vertexArrayObject);
52     //glDrawArrays(GL_TRIANGLES, 0, vertexCount);
53     /*for (int i = 0; i < vertexCount; i+=3)
54     {
55         glBegin(GL_TRIANGLES);
56
57         glVertex3f(pVertices[i].pos.x, pVertices[i].pos.y, pVertices[i].pos.z);
58
59         glVertex3f(pVertices[i+1].pos.x, pVertices[i+1].pos.y, pVertices[i+1].pos.z);
60
61         glVertex3f(pVertices[i+2].pos.x, pVertices[i+2].pos.y, pVertices[i+2].pos.z);
62
63         glEnd();
64     }*/
65
66     // Drawing the textured mesh
67     for (int i = 0; i < vertexCount; i += 3)
68     {
69         glBegin(GL_TRIANGLES);
70         glTexCoord2f(0.0f, 0.0f); // Set the current texture coordinates
71         glVertex3f(pVertices[i].pos.x, pVertices[i].pos.y, pVertices[i].pos.z); // Specify a vertex
72         glTexCoord2f(0.0f, 1.0f);
73         glVertex3f(pVertices[i + 1].pos.x, pVertices[i + 1].pos.y, pVertices[i + 1].pos.z);
74         glTexCoord2f(1.0f, 1.0f);
75         glVertex3f(pVertices[i + 2].pos.x, pVertices[i + 2].pos.y, pVertices[i + 2].pos.z);
76         glTexCoord2f(1.0f, 0.0f);
77         glVertex3f(pVertices[i + 3].pos.x, pVertices[i + 3].pos.y, pVertices[i + 3].pos.z);
78         glEnd();
79     }
80 }

```

Fig.13 The example of drawing textured mesh function.

```

123 void Mesh::loadTexture(const char * filename)
124 {
125     unsigned char * data; // Pointer to the retrieved data
126
127     FILE * file; // File pointer
128
129     file = fopen(filename, "rb"); // Open the file
130
131     // Set the of the texture
132     int width = 256;
133     int height = 256;
134
135     // Allocates a block of size bytes of memory, returning a pointer to the beginning of the block
136     data = (unsigned char *)malloc(width * height * 3);
137
138     // Reads an array of count elements, each one with a size of size bytes, from the stream and stores them in the block of memory specified by ptr
139     fread(data, width * height * 3, 1, file);
140
141     fclose(file); // Closes the file stream
142
143     // Save to the data variable
144     for (int i = 0; i < width * height; ++i)
145     {
146         int index = i * 3;
147         unsigned char b, r;
148         b = data[index];
149         r = data[index + 2];
150
151         data[index] = r;
152         data[index + 2] = b;
153     }
154
155     glGenTextures(1, &texture); // Generates texture name
156     glBindTexture(GL_TEXTURE_2D, texture); // Binds a named texture to a texturing target
157
158     glPixelStorei(GL_UNPACK_ALIGNMENT, 1); // Sets pixel storage mode
159
160     // Sets texture parameters
161     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
162     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
163     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
164     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
165
166     if (data)
167     {
168         glEnable(GL_TEXTURE_2D); // Enable or disable server - side GL capabilities
169
170         glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE); // Sets texture environment parameters
171
172         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data); // Specify a two - dimensional texture image
173         glGenerateMipmap(GL_TEXTURE_2D); // Generates mipmaps for a specified texture object
174     }
175     else
176     {
177         std::cout << "Failed to load texture" << std::endl;
178     }
179
180     free(data); // Deallocates the memory previously allocated by malloc
181 }

```

Fig.14 The designed bitmap loader allows for loading textures.

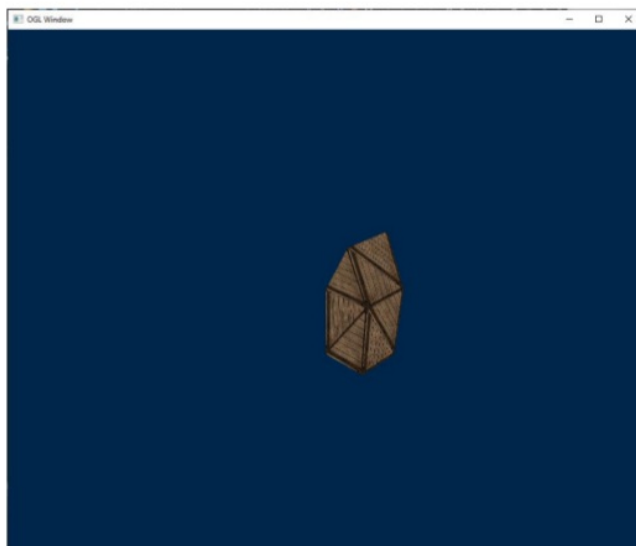


Fig.15 The example of the textured mesh object.

6 ADDING NORMALS AND TEXTURE COORDINATES

Adding normals and texture coordinates have been implemented by creating specific methods in objectloader class and mesh class. Nevertheless, the object is not rendered correctly, when texture coordinates are applied to the textured mesh.

```
44 // objectloader.h
45 int getNumNormals(void) { return normcoord_qty; } // Return number of normals
46 normcoord_type getNormals(int i) { return normcoord[i]; } // Return normal
47
48 int getNumTextCoord(void) { return mapcoord_qty; } // Return number of texture coordinates
49 mapcoord_type getTextCoord(int i) { return mapcoord[i]; } // Return texture coordinates
50
```

```
45 // mesh.h
46 // Set the list of texture coordinates
47 void objTextCoord(void);
48
49 // Set the list of normals
50 void objNormal(void);
51
```

```
62 private:
63
64 Vertex *pVertices; // Store the list of verticies
65 TextCoord *pTextCoord; // Store the list of texture coordinates
66 NormCoord *pNormals; // Store the list of normals
67
```

```
139 void Mesh::objTextCoord(void)
140 {
141     int textCoordCount = myobj.getNumTextCoord();
142     pTextCoord = new TextCoord[textCoordCount];
143
144     for (int i = 0; i < vertexCount; i++) //for each polygon
145     {
146         pTextCoord[i].u = myobj.getTextCoord(i).u;
147         pTextCoord[i].v = myobj.getTextCoord(i).v;
148     }
149 }
150
151 void Mesh::objNormal(void)
152 {
153     int normCount = myobj.getNumNormals();
154     pNormals = new NormCoord[normCount];
155
156     for (int i = 0; i < vertexCount; i++) //for each polygon
157     {
158         pNormals[i].normal.x = myobj.getNormals(i).i;
159         pNormals[i].normal.y = myobj.getNormals(i).j;
160         pNormals[i].normal.z = myobj.getNormals(i).k;
161     }
162 }
```

Fig.14 The initialised variables and function for storing texture coordinates and normals.

```
88 // Drawing the textured mesh with using added texture coordinates
89 for (int i = 0; i < vertexCount; i += 3)
90 {
91     glBegin(GL_TRIANGLES);
92     glTexCoord2f(pTexCoord[i].u, pTexCoord[i].v); // Set the current texture coordinates
93     glVertex3f(pVertices[i].pos.x, pVertices[i].pos.y, pVertices[i].pos.z); // Specify a vertex
94     std::cout << pTexCoord[i].u << ", " << pTexCoord[i].v << std::endl;
95
96     glTexCoord2f(pTexCoord[i + 1].u, pTexCoord[i + 1].v);
97     glVertex3f(pVertices[i + 1].pos.x, pVertices[i + 1].pos.y, pVertices[i + 1].pos.z);
98     std::cout << pTexCoord[i+1].u << ", " << pTexCoord[i+1].v << std::endl;
99
100
101     glTexCoord2f(pTexCoord[i + 2].u, pTexCoord[i + 2].v);
102     glVertex3f(pVertices[i + 2].pos.x, pVertices[i + 2].pos.y, pVertices[i + 2].pos.z);
103
104     glTexCoord2f(pTexCoord[i + 3].u, pTexCoord[i + 3].v);
105     glVertex3f(pVertices[i + 3].pos.x, pVertices[i + 3].pos.y, pVertices[i + 3].pos.z);
106     glEnd();
107 }
108 }
```

Fig.17 The example of function which renders a textured mesh involving previously added texture coordinates.

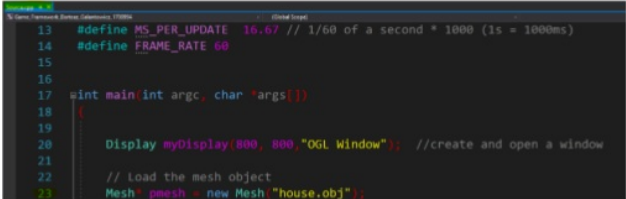
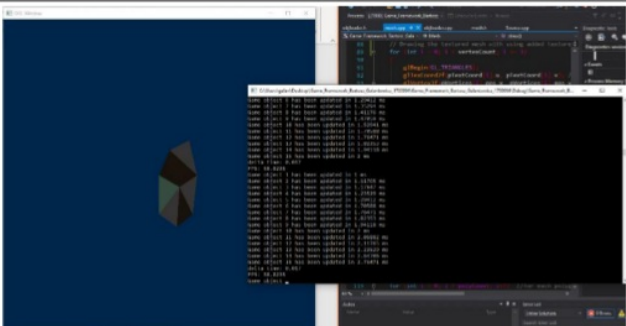
7 BIBLIOGRAPHY

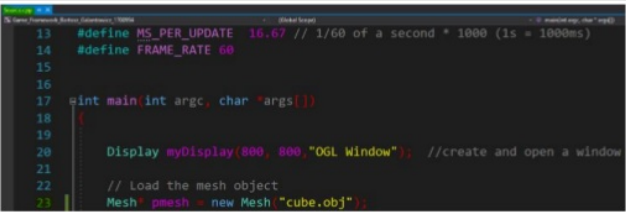
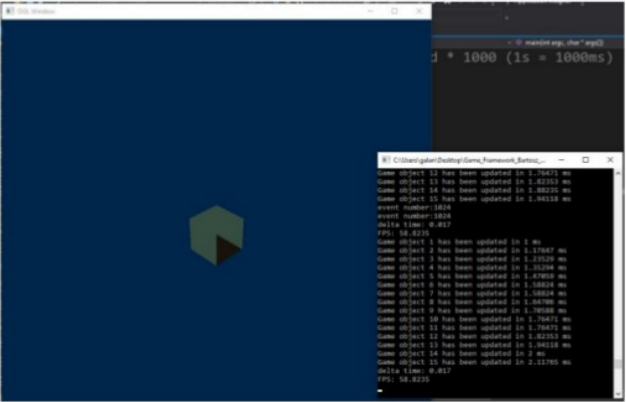
- [1] J. C. V. Scott Gordon, Computer Graphics Programming in OpenGL with C++, David Pallai, 2019, pp. 1-5, 27, 36-46, 68-82, 103-112.
- [2] E. Lengyel, Foundations of Game Engine Development, vol. 1, Lincoln, California: Terathon Software LLC, 2016, pp. 97-124.
- [3] R. Nystrom, Game Programming Patterns, Robert Nystrom, 2014, pp. 19-87, 123-139.
- [4] Lazy Foo' Productions, "Beginning Game Programming v2.0," [Online]. Available: Beginning Game Programming v2.0. [Accessed April 2020].
- [5] SDL, "SDL Library Documentation," [Online]. Available: <https://www.libsdl.org/release/SDL-1.2.15/docs/html/guidevideo.html>. [Accessed April 2020].

8 APPENDIX

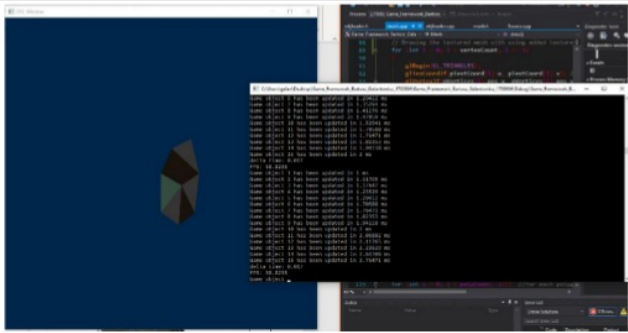
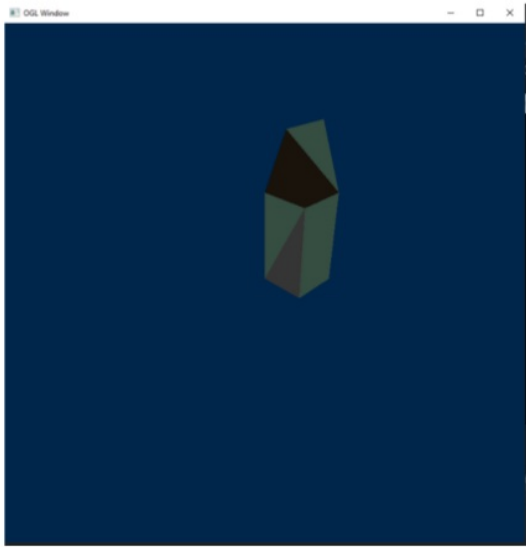
8.1 TEST CASES

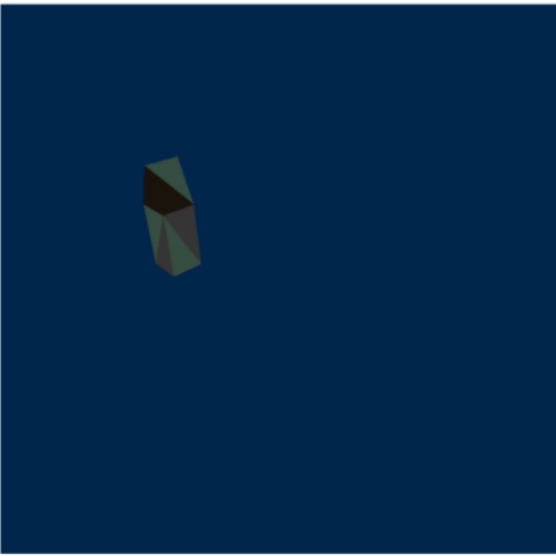
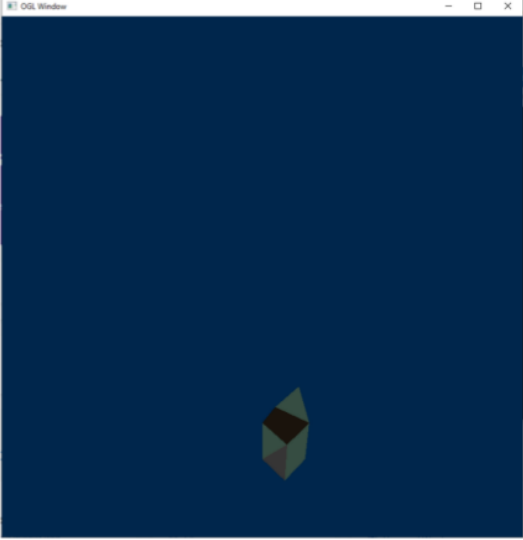
The test cases are presented in this section of the report as evidence of evaluation and testing the developed software.

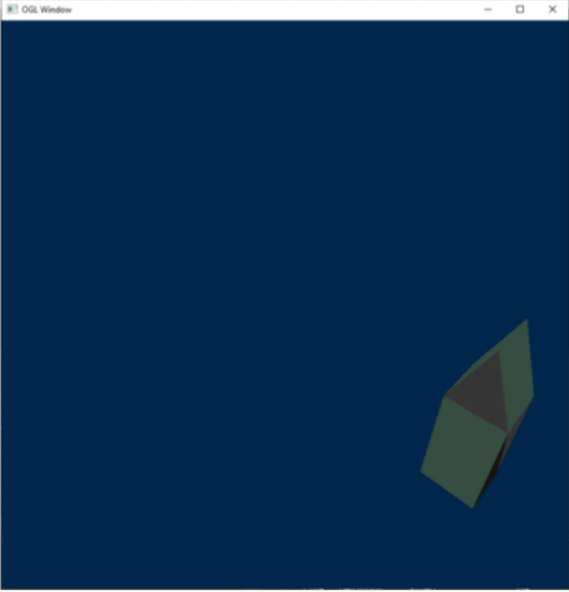
Test Case Id: ObjectRender		Test Purpose: To check if the cube and house objects can be rendered.	
Environment: Windows 10 Pro Microsoft Visual Studio Community 2017 v.15.9.15			
Preconditions: The program has been started.			
Test Case Steps:			
Step No	Procedure	Response	Pass/Fail
1	Locate the Source.cpp and set mesh *pmesh object (Line 23) to "house.obj".	 The new mesh object is going to load.	Pass
2	Press "CTRL" + "F5" buttons to run the debug mode.	 The application windows are displayed. The "house" is rendered in the OGL window.	Pass

3	Locate the Source.cpp and set mesh *pmesh object (Line 23) to "cube.obj".	 The new mesh object is going to load.	Pass
4	Press "CTRL" + "F5" buttons to run the debug mode.	 The application windows are displayed. The "cube" is rendered in the OGL window.	Pass
Comments: All steps passed.			
Author: Bartosz Galantowicz			

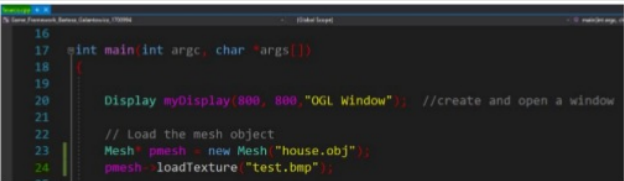
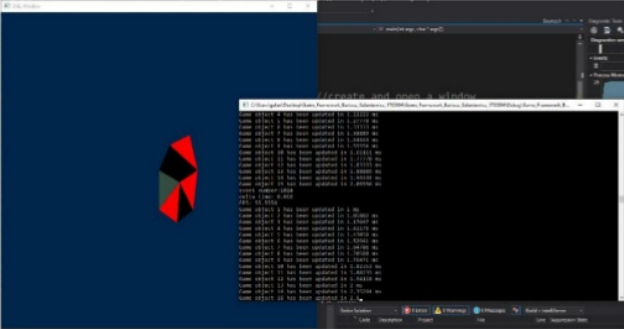
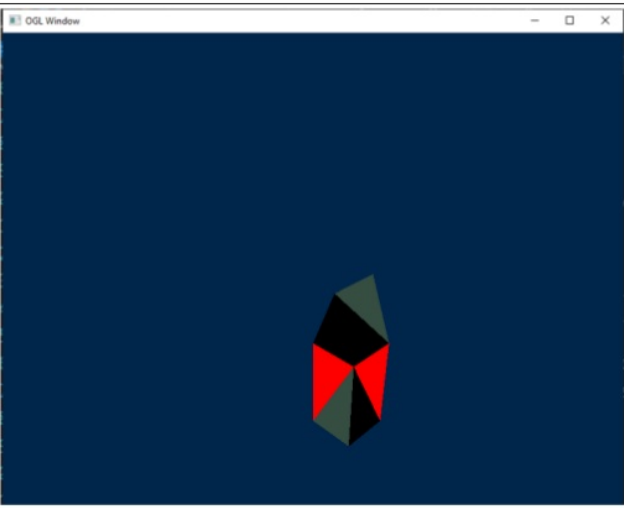
Test Case Id: ObjectTranslation		Test Purpose: To check if the rendered object can be moved.	
Environment: Windows 10 Pro Microsoft Visual Studio Community 2017 v.15.9.15			
Preconditions: The program has been started.			
Test Case Steps:			
Step No	Procedure	Response	Pass/ Fail

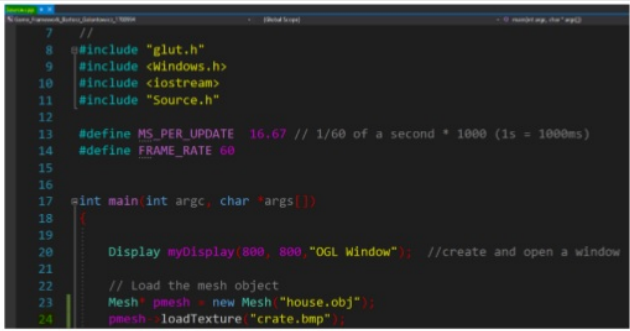
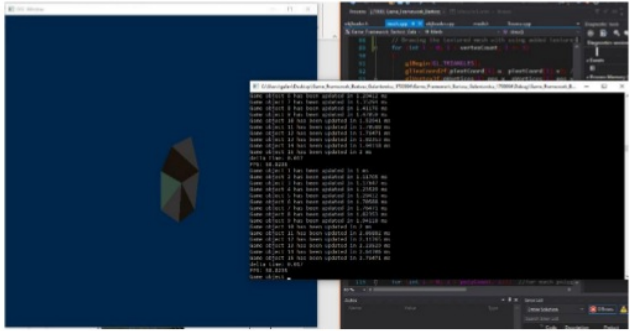
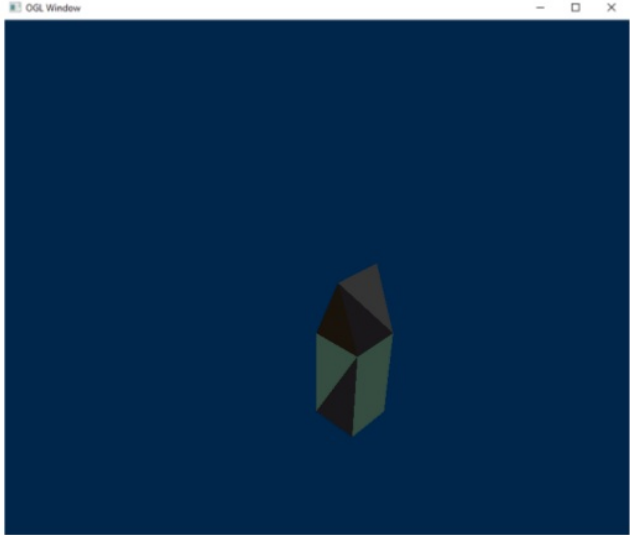
1	Press "CTRL" + "F5" buttons to run the debug mode.	 The application windows are displayed. The "house" is rendered in the OGL window.	Pass
2	Move the object up by pressing "w" on the keyboard	 The object is moving up while "w" button is pressed.	Pass

3	Move the object left by pressing "a" on the keyboard	 The object is moving left while "a" button is pressed.	Pass
4	Move the object down by pressing "s" on the keyboard	 The object is moving down while "s" button is pressed.	Pass

	Move the object right by pressing "d" on the keyboard	 The object is moving right while "d" button is pressed.	Pass
Comments: All steps passed.			
Author: Bartosz Galantowicz			

Test Case Id: TextureLoader		Test Purpose: To check if various bitmaps can be load as textures.	
Environment: Windows 10 Pro Microsoft Visual Studio Community 2017 v.15.9.15			
Preconditions: The program has been started.			
Test Case Steps:			
Step No	Procedure	Response	Pass/ Fail

1	Locate the Source.cpp and set the argument of the loadTexture function (Line 24) to "test.bmp".	 <pre> 16 17 int main(int argc, char *args[]) 18 { 19 Display myDisplay(800, 800, "OGL Window"); //create and open a window 20 21 // Load the mesh object 22 Mesh* pmesh = new Mesh("house.obj"); 23 pmesh->loadTexture("test.bmp"); 24 </pre> The new texture is going to load.	Pass
2	Press "CTRL" + "F5" buttons to run the debug mode.	 The application windows are displayed. The "house" is rendered in the OGL window.	Pass
3	Check if the textures have been loaded properly.	 The textures have been loaded. Nevertheless, they were not loaded properly.	Fail

4	Locate the Source.cpp and set the argument of the loadTexture function (Line 24) to "test.bmp".	 <pre> 7 // 8 #include "glut.h" 9 #include <Windows.h> 10 #include <iostream> 11 #include "Source.h" 12 13 #define MS_PER_UPDATE 16.67 // 1/60 of a second * 1000 (1s = 1000ms) 14 #define FRAME_RATE 60 15 16 17 int main(int argc, char *args[]) 18 { 19 // Create an OpenGL window 20 Display myDisplay(800, 800, "OGL Window"); //create and open a window 21 22 // Load the mesh object 23 Mesh pmesh = new Mesh("house.obj"); 24 pmesh->loadTexture("crate.bmp"); </pre> The new texture is going to load.	Pass
5	Press "CTRL" + "F5" buttons to run the debug mode.	 The application windows are displayed. The "house" is rendered in the OGL window.	Pass
6	Check if the textures have been loaded properly.	 The textures have been loaded. Nevertheless, they were not loaded properly.	Fail

Comments:

Steps 1, 2, 4 and 5 have been passed.

Steps 3 and 6 have failed – the textures were not loaded properly.

Author: Bartosz Galantowicz

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

86/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

GRADING FORM: SMU GENERIC_GJ

**BARTOSZ
GALANTOWICZ**

86

RESEARCH-INFORMED LITERATURE



Good work here. Refs appropriate. Well done

8

KNOWLEDGE AND UNDERSTANDING



Covers requisite areas and includes new work here.
Again Well done.

17

ANALYSIS



Appropriate patterns identified and designed and integrated into the framework.

17

PRACTICAL APPLICATION AND DEP.



A very good effort. Implementation and testing is of a good standard. Well done. Textures included as well. just a little more effort required here to solve the texture problem - this was not necessary for the assignment - well done for trying!

40

SKILLS FOR PROFESSIONAL PRAC.



Well structured. Good use of SE practices here. Good referencing.

4