Computación Gráfica e Interacción Humano-Computadora.

Grupo 4.

Profesor: Ing. Carlos Aldair Roman Balburna.

Alumno: Jorge Octavio Barcenas Avelar.

Proyecto Final.

Manual Técnico.

Contenido

Introducción

El curso de Computación Gráfica e Interacción Humano-Computadora, es aquella materia que acerca al alumno a conocer las técnicas básicas de la computación gráfica, el siguiente trabajo está dirigido a que el alumno demuestre su conocimiento adquirido tanto en la parte teórica como práctica.

Objetivo

El alumno deberá aplicar y demostrar los conocimientos adquiridos durante todo el curso.

Alcance

Se recreará el primer piso de la casa de los personajes principales de la serie “Hora de aventura”, el cual está conformado por un cuarto que sirve como cocina y sala, y un segundo cuarto que sirve como baño, en un espacio tridimensional utilizando el software de modelado Maya y Open GL.

Se modelaran 11 objetos y 4 personajes.

Se modelará la fachada de la casa.

Se tendrán 5 animaciones, de las cuales 3 son sencillas y 2 complejas.

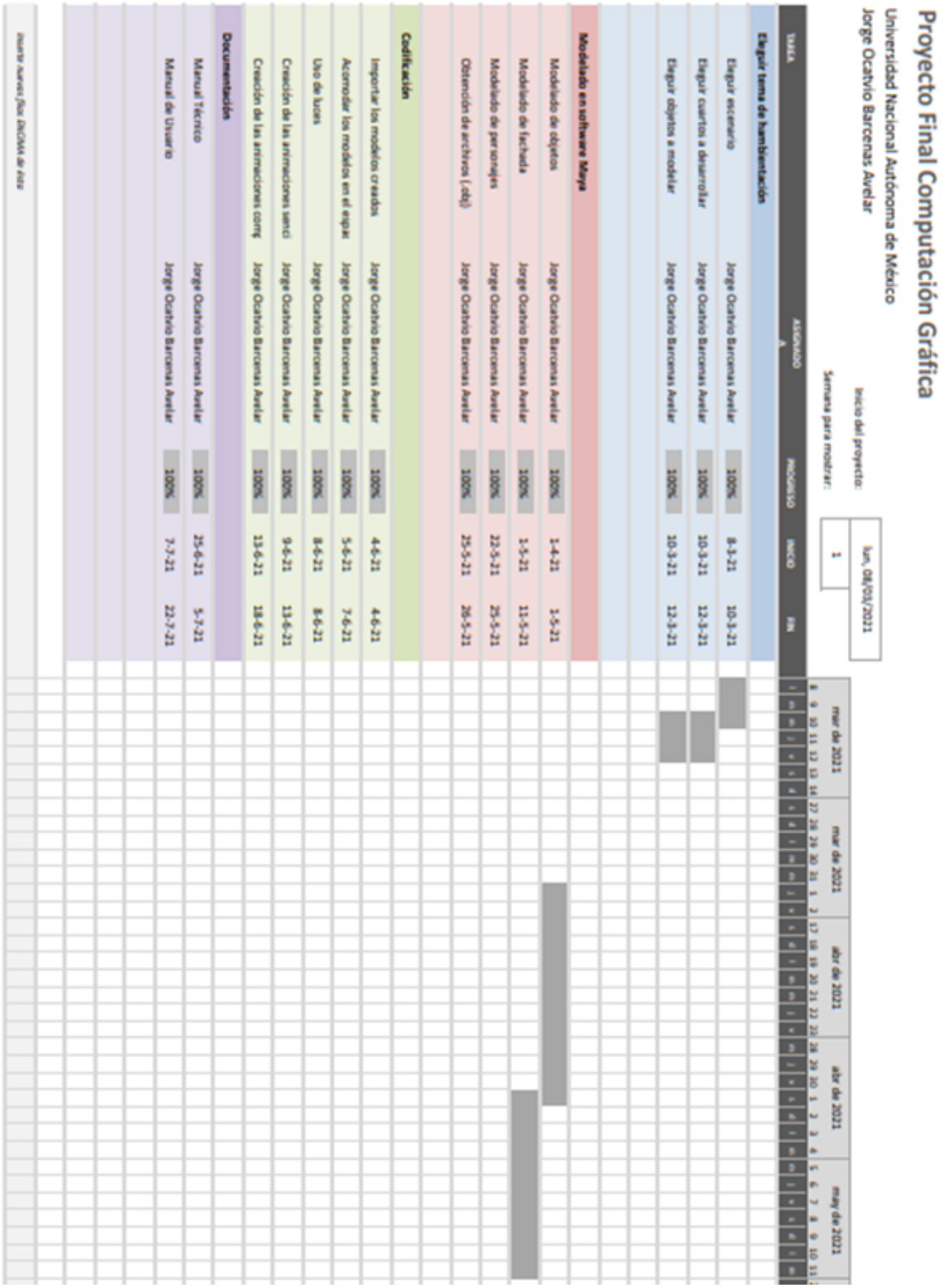
Se hará uso del manejo de una cámara con la cual se pueda recorrer el ambiente creado.

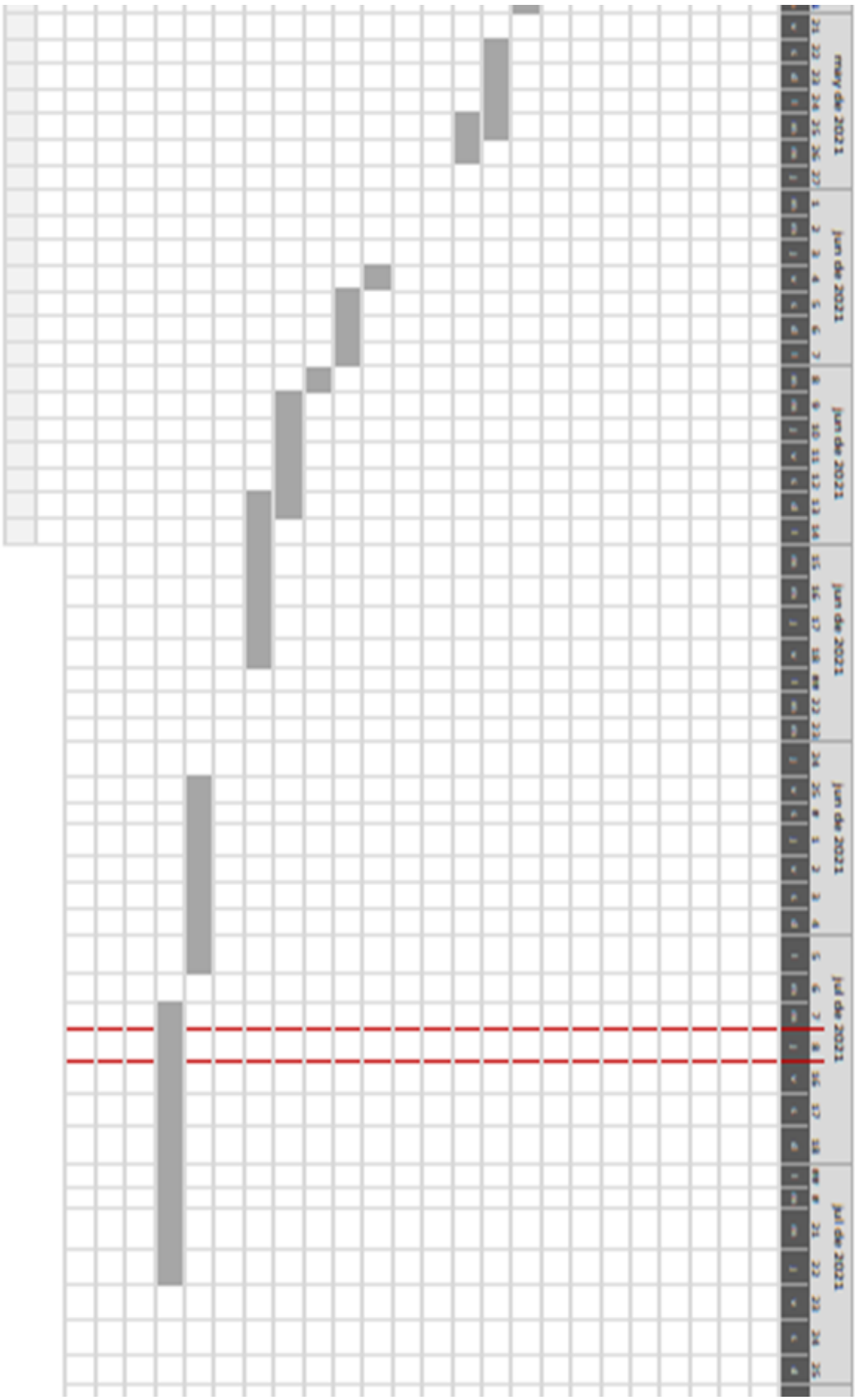
Limitantes

La falta de conocimiento de las herramientas dentro del software de modelado Maya.

Es necesario mencionar que para todas las rutas que se usen en este documento se tomará como punto de partida la página inicial de nuestro repositorio https://github.com/JOBA25/ProyectoComputacionGrafica

Diagrama de Gantt





Para una mejor visualización consultar el archivo “pfcg.xlsx” en la carpeta “Manules y Documentos”.

Marco Teórico.

Shader.

Un shader es un conjunto de instrucciones y cálculos que serán ejecutadas en la tarjeta gráfica utilizando lenguaje GLSL (OpenGL Shading Language), con el shader es posible mandar información tal como lo son color, luz, textura, posición, rotación, escalamiento, etc.

Vertex shader.

El vertex shader es el responsable de ser quien reciba la información de primitivas, bordes o vértices, además de información de traslación, rotación o escalamiento.

Fragment shader.

El fragment shader es quien recibe la información del vertex shader que paso por el proceso de rasterización donde se obtiene cada uno de los pixeles, se hace un muestreo entre todos los píxeles y se entintan o texturizan según sea necesario.

VBO (Vertex buffer object).

Es el buffer de memoria que contiene la información acerca de los vértices donde se manda información de posición, color, normales, etc.

VAO (Vertex array object).

Es un objeto el cual contiene uno o más VBO dándoles un índice dentro de sí.

EBO (Element buffer object).

El EBO nos ayuda a indexar la información dentro del VAO y facilitar su manejo.

Proyección en perspectiva.

En la proyección en perspectiva tendremos un punto de fuga o centro de proyección, esto es la principal característica de esta proyección lo cual trae consigo beneficios como son el obtener la sensación de profundidad en los objetos.

Modelado jerárquico.
Es un modelo de datos en el cual los datos son organizados en una estructura parecida a un árbol. La estructura permite a la información tener relaciones del tipo padre/hijo, donde cada padre puede tener varios hijos pero un hijo solo puede tener un padre. Este modelo es utilizado para obtener las animaciones por keyframes.

Cámara sintética.
Este tipo de cámara nos permitirá poder movernos dentro del ambiente creado.

Iluminación.

- Componente ambiental: Es la luz que va a incidir en todo nuestro espacio.
- Componente difusa: este componente simula el impacto direccional que un rayo de luz tiene sobre un objeto.
- Componente especular: Simula el punto brillante de una luz que aparece en los objetos en los que incide.

Luz direccional.
permite iluminar todo el espacio de nuestro ambiente, simulando al sol, ya que emana haces de luz en una dirección específica.

Point light o luz puntual.
Esta luz emana haces de luz en todas direcciones para formar un punto de luz, simulando un foco.
Este tipo de luz utiliza un factor de atenuación el cual está dado por la siguiente fórmula:

$$F_{att} = \frac{1.0}{1.0 + K_L * d + K_q * d^2}$$

donde el término lineal K_L se multiplica por la distancia que reduce la intensidad de forma lineal. El término cuadrático K_q se multiplica por el cuadrado de la distancia y establece una disminución cuadrática de la intensidad de la fuente de luz.

Spotlight o luz de reflector.
Esta luz simula el efecto de una lámpara o reflector, en donde sus rayos son acotados, además se hace uso de un cono interior y uno exterior.
en esta Luz son usados los siguientes términos:
LightDir es el vector que apunta desde el fragmento a la fuente de luz.
SpotDir es la dirección a la que apunta el proyector.
Phi es el ángulo de corte que especifica el radio del foco. Todo lo que esté fuera de este ángulo no estará iluminado por el foco.
Theta es el ángulo entre el vector LightDir y el vector SpotDir. Este ángulo debe ser menor al de phi para estar dentro del foco.

Textura.
Es una imagen tratada en el espacio (u, v) que es asignada a una figura tridimensional para darle un mayor realismo.

Skybox.
Es una técnica de texturizado para el ambiente el cual consiste en crear un cubo gigante y texturizarlo por dentro.

Animación.
Una animación es una visualización de los cambios de las propiedades de un objeto.

Animación por keyframes.
En este tipo de animación se tiene un conjunto de posiciones las cuales ordenamos en forma cronológica, tomamos una posición y la posición siguiente a esta y las nombraremos posición inicial y posición final, el siguiente paso será calcular las posiciones intermedias necesarias para poder llegar de la posición inicial a la final por medio del uso de la interpolación.

Descripción del código

A continuación se hará la descripción de las partes más importantes dentro del código usado. Comenzaremos con los archivos no principales.

En la siguiente ruta “ProyectoFinal/External Libraries” se encuentran diferentes carpetas de las cuales haremos mención de sus funciones:

- La carpeta “GLFW” contiene los archivos necesarios que ayudarán al manejo de ventanas.
- La carpeta “GLEW” contiene archivos necesarios para el manejo de funciones de OpenGL.
- La carpeta “glm” ayuda a realizar los cálculos matemáticos necesarios para trabajar las matrices de vista, proyección y modelo y poder trabajar con la traslación y rotación de los objetos.
- La carpeta “assimp” nos va a permitir trabajar con modelos 3D.

Dentro de la ruta “ProyectoFinal/ProyectoFinal/” se encuentran los archivos principales para el funcionamiento del proyecto.

El archivo “camera.h” sirve para crear, posicionar la cámara sintentica, definir valores de configuración, como la sensibilidad, al rango de apertura, la velocidad con que se moverá y contiene las funciones que nos ayudarán a atrapar los eventos del teclado y el movimiento del mouse.

Al archivo lighting.vs se le mandan las posiciones del fragmento, las normales y coordenadas de textura, se calculan la matrices de proyección, vista y modelo, para que pueda devolver las normales, la posición del fragmento que se esté trabajando y las coordenadas de textura. Se hace la multiplicación de las matrices proyección, vista y modelo para conseguir la posición de la cámara, se calcula la posición de fragmento multiplicando la matriz de modelo por la posición, se trabaja la normalización de las normales para que apunten correctamente, después de esto se pasa al fragment shader.

Al archivo lighting.frag en un inicio se define el número total de point lights que se usarán, se recibe la estructura de los materiales con sus componentes ambiental, difusa, especular y brillo, la estructura de la luz direccional con sus componentes de dirección, ambiental, difusa y especular, la estructura de las point light con sus componentes de posición, ambiental, difusa, especular, la estructura de la spotlight con sus componentes de posición, dirección, cono interior, cono exterior, ambiental, difusa, especular y sus términos lineal, constante, cuadrático, así como los vectores de posición de fragmento, normal y de coordenadas de textura provenientes del vertex shader, y como salida se tendrá el color de ese fragmento.

El archivo lamp.vs recibe la posición, las matrices de vista, modelo y proyección, y manda la posición al fragment shader.

El archivo “lamp.frag” únicamente devuelve el color.

El archivo “Mesh.h” es el encargado de mandar el objeto a “Model.h” y mandarlo al shader.

El archivo “Model.h” es el encargado de texturizar y dibujar los objetos. Una de las funciones más importantes dentro de este archivo es la función “Draw”, la cual manda la geometría al shader.

El archivo “modelLoading.vs” es el encargo de funcionar como vertex shader, recibiendo posiciones, normales y coordenadas de textura, las matrices de modelo, vista y proyección y mandanda toda esta información al fragment shader.

El archivo “modelLoading.frag” es el responsable de desplegar la textura difusa del modelo 3D.

Los archivos “SkyBox.vs” y “SkyBox.frag” ayudan poner en práctica la técnica de skybox capturando la posición de las imágenes y se le envían al cubo gigante que rodea nuestro ambiente.

El archivo “Main_proyectoFinal.cpp” es donde se tendrá el hilo principal del programa.
Al comienzo del archivo se tiene la inclusión de bibliotecas y la declaración de variables y estructuras necesarias.
Iniciamos con la inclusión de bibliotecas:

```cpp
#include <iostream>
#include <cmath>

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

// GLFW
#include <GLFW/glfw3.h>

// Other Libs
#include "stb_image.h"

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

//Load Models
#include "SOIL2/SOIL2.h"


// Other includes
#include "Shader.h"
#include "Camera.h"
#include "Model.h"
#include "Texture.h"
```

En seguida se encuentra la declaración de las funciones que usaremos en el código:

```cpp
// Function prototypes
void KeyCallback(GLFWwindow *window, int key, int scancode, int action, int mode);
void MouseCallback(GLFWwindow *window, double xPos, double yPos);
void DoMovement();
void animacion();
```

A continuación se tiene las variables necesarias para las dimensiones de la ventana en que se verá la ejecución del programa y para el uso de la cámara:

```
34      // Window dimensions
35      const GLuint WIDTH = 1000, HEIGHT = 800;
36      int SCREEN_WIDTH, SCREEN_HEIGHT;
37
38      // Camera
39      Camera  camera(glm::vec3(0.0f, 3.0f, 0.0f));
40      GLfloat lastX = WIDTH / 2.0;
41      GLfloat lastY = HEIGHT / 2.0;
42      bool keys[1024];
43      bool firstMouse = true;
```

Declaramos las variables globales que se obtendrán los valores de las estructuras creadas para los personajes y que se llevarán a los respectivos modelos para observar las animaciones por Key Frame, y declaramos las constantes del número máximo de posiciones que tienen cada uno:

```
54  // variables usadas para las animaciones de Keyframes
55  float rotPiernaIzqBmo = 0, rotPiernaDerBmo = 0, rotBrazoIzqBmo = 0, rotBrazoDerBmo = 0;
56
57  glm::vec3 PosIni(0.0f, 0.0f, 0.0f);
58  float posX = PosIni.x, posY = PosIni.y, posZ = PosIni.z;
59  float rotPiernaIzqFinn = 0, rotPiernaDerFinn = 0;
60  float rot = 0, posXPiDe = -0.7, posZPiDe = 0;
61  float posXPiIz = 0.7, posZPiIz = 0;
62  bool animFinn = true;
63
64  //máximo de posiciones
65  #define MAX_FRAMES_BMO 4
66  #define MAX_FRAMES_FINN 13
```

Declaramos la estructura que usará el personaje “BMO”, declaramos un arreglo de la estructura mencionada y variables que servirán para el recorrido de esta estructura:

```
74  typedef struct _frame
75  {
76
77      float rotPiernaIzqBmo;
78      float rotIncPiernaIzqBmo;
79
80      float rotPiernaDerBmo;
81      float rotIncPiernaDerBmo;
82
83      float rotBrazoIzqBmo;
84      float rotIncBrazoIzqBmo;
85
86      float rotBrazoDerBmo;
87      float rotIncBrazoDerBmo;
88
89  }FRAME;
90
91  FRAME KeyFrameBMO[MAX_FRAMES_BMO];
92  //introducir datos
93  int FrameIndexBMO = 0;
94  bool playB = false;
95  int playIndex = 0;
```

Hacemos lo mismo con el personaje “Finn”, como se ve en la imagen del código es una estructura más compleja, ya que tendrá una animación más desarrollada:

```
97  typedef struct
98  {
99      //Variables para GUARDAR Key Frames
100     float posX;     //Variable para PosicionX
101     float posZ;     //Variable para PosicionZ
102     float incX;     //Variable para IncrementoX
103     float incZ;     //Variable para IncrementoZ
104
105     float posXPiDe;      //al momento de dar vuelta deberemos afectar la posicion
106     float posZPiDe;
107     float posXIncPiDe;   //de las piernas
108     float posZIncPiDe;
109
110     float posZPiIz;
111     float posXPiIz;
112     float posZIncPiIz;
113     float posXIncPiIz;
114
115     float rot;
116     float rotInc;
117     float rotPiernaIzqFinn;
118     float rotIncPiernaIzqFinn;
119     float rotPiernaDerFinn;
120     float rotIncPiernaDerFinn;
121 }FRAME_F;
122
123 FRAME_F KeyFrameFinn[MAX_FRAMES_FINN];
124 //introducir datos
125 int FrameIndexFinn = 0;
126 bool playF = false;
127 int playIndexF = 0;
```

Se declara un arreglo de vectores, los cuales contendrán las coordenadas de las point lights, dentro de nuestro ambiente:

```
130     // Positions of the point lights
131     glm::vec3 pointLightPositions[] = {
132         glm::vec3(27.0f,15.0f,0.0f),
133         glm::vec3(-30.0f,15.0f,54.0f),
134         glm::vec3(-30.0f,15.0f,-25.0f),
135         glm::vec3(27.0f,15.0f,-29.0f)
136     };
137
```

Se declaran las variables utilizadas para las animaciones de otros dos personajes (un pato de hule y un caracol) y de un objeto (un hotcake), mediante animaciones sencillas:

```
141     //variables de animación pato en bañera
142     float patoX = 45.0;
143     float patoZ = -34.0;
144     float patoRot = 0.0;
145
146     bool circuitoPato = false;
147     bool recorrido1Pato = true;
148     bool recorrido2Pato = false;
149
150     //variables de animación Caracol
151     float caracolX = -29.0;
152     float caracolZ = 51.0;
153     float caracolRot = 0.0;
154
155     bool circuitoCaracol = false;
156     bool recorrido1Caracol = true;
157     bool recorrido2Caracol = false;
158     bool recorrido3Caracol = false;
159     bool recorrido4Caracol = false;
160
161     //variables de animación hotcake
162     float hotcakeY = 6.85;
163     float hotcakeRot = 0;
164     bool circuitoHotcake = false;
165     bool recorrido1Hotcake = true;
166     bool recorrido2Hotcake = false;
167     bool recorrido3Hotcake = false;
```

Una vez que se termina la sección de declaraciones, comienza la declaración de funciones.

Las siguientes funciones (“saveFrame”, ”resetElements”,”interpolation” e “interpolacionF”) son usadas para el proceso de animación por keyFrames.

La función “saveFrame” nos ayuda a salvar las posiciones de los personajes y el máximo de estas.

La función “resetElements” sirve para poder resetear los valores de las coordenadas de extremidades del personaje “BMO” a un valor inicial, ya que es en sus extremidades donde se verá la animación:

```
328     void resetElements(void)
329     {
330         rotPiernaIzqBmo = KeyFrameBMO[0].rotPiernaIzqBmo;
331         rotPiernaDerBmo = KeyFrameBMO[0].rotPiernaDerBmo;
332         rotBrazoIzqBmo = KeyFrameBMO[0].rotBrazoIzqBmo;
333         rotBrazoDerBmo = KeyFrameBMO[0].rotBrazoDerBmo;
334
335
336     }
```

Las funciones “interpolation” e “interpolacionF” nos ayudan a hacer la interpolación entre cada una de las posiciones que guardamos en la función “saveFrame”, encontrando el incremento necesario para poder llegar de una posición inicial a una final en 190 pasos:

```
338 void interpolation(void)
339 {
340
341
342     KeyFrameBMO[playIndex].rotIncPiernaIzqBmo = (KeyFrameBMO[playIndex + 1].rotPiernaIzqBmo - KeyFrameBMO[playIndex].rotPiernaIzqBmo) / i_max_steps;
343     KeyFrameBMO[playIndex].rotIncPiernaDerBmo = (KeyFrameBMO[playIndex + 1].rotPiernaDerBmo - KeyFrameBMO[playIndex].rotPiernaDerBmo) / i_max_steps;
344     KeyFrameBMO[playIndex].rotIncBrazoIzqBmo = (KeyFrameBMO[playIndex + 1].rotBrazoIzqBmo - KeyFrameBMO[playIndex].rotBrazoIzqBmo) / i_max_steps;
345     KeyFrameBMO[playIndex].rotIncBrazoDerBmo = (KeyFrameBMO[playIndex + 1].rotBrazoDerBmo - KeyFrameBMO[playIndex].rotBrazoDerBmo) / i_max_steps;
346 }
347
348
349
350 void interpolacionF(void) {
351     KeyFrameFinn[playIndexF].incX              = (KeyFrameFinn[playIndexF + 1].posX           - KeyFrameFinn[playIndexF].posX)           / i_max_steps;
352     KeyFrameFinn[playIndexF].incZ              = (KeyFrameFinn[playIndexF + 1].posZ           - KeyFrameFinn[playIndexF].posZ)           / i_max_steps;
353     KeyFrameFinn[playIndexF].rotInc            = (KeyFrameFinn[playIndexF + 1].rot            - KeyFrameFinn[playIndexF].rot)            / i_max_steps;
354     KeyFrameFinn[playIndexF].rotIncPiernaDerFinn = (KeyFrameFinn[playIndexF + 1].rotPiernaDerFinn - KeyFrameFinn[playIndexF].rotPiernaDerFinn) / i_max_steps;
355     KeyFrameFinn[playIndexF].rotIncPiernaIzqFinn = (KeyFrameFinn[playIndexF + 1].rotPiernaIzqFinn - KeyFrameFinn[playIndexF].rotPiernaIzqFinn) / i_max_steps;
356     KeyFrameFinn[playIndexF].posXIncPiDe       = (KeyFrameFinn[playIndexF + 1].posXPiDe       - KeyFrameFinn[playIndexF].posXPiDe)       / i_max_steps;
357     KeyFrameFinn[playIndexF].posZIncPiDe       = (KeyFrameFinn[playIndexF + 1].posZPiDe       - KeyFrameFinn[playIndexF].posZPiDe)       / i_max_steps;
358     KeyFrameFinn[playIndexF].posXIncPiIz       = (KeyFrameFinn[playIndexF + 1].posXPiIz       - KeyFrameFinn[playIndexF].posXPiIz)      / i_max_steps;
359     KeyFrameFinn[playIndexF].posZIncPiIz       = (KeyFrameFinn[playIndexF + 1].posZPiIz       - KeyFrameFinn[playIndexF].posZPiIz)      / i_max_steps;
360 }
```

Dentro de la función “animacion” se tienen los diferentes recorridos de las animaciones sencillas, en estos recorridos lo que se hace es que se aumenta una variable dentro de un bucle mientras se dibuja el cambio de esa variable hasta llegar a un cierto valor para seguir con el movimiento de la siguiente variable o terminar la animación:

```
1058    void animacion()
1059    {
1060        //movimiento del pato
1061        if (circuitoPato) {
1062            if (recorrido1Pato) {
1063                patoZ += 0.05f;
1064
1065
1066                if (patoZ > -26 ) {
1067                    recorrido1Pato = false;
1068                    recorrido2Pato = true;
1069                    patoRot = 180.0f;
1070                }
1071            }
1072
1073            if (recorrido2Pato) {
1074                patoZ -= 0.05f;
1075
1076                if (patoZ < -34) {
1077                    recorrido2Pato = false;
1078                    recorrido1Pato = true;
1079                    patoRot = 0.0f;
1080                }
1081            }
1082
1083
1084        }
```

Dentro de esta función también se encuentra el código necesario para obtener las animaciones complejas por key frame, donde se hace algo similar a las animaciones sencillas, pero aquí se manda a llamar la función de interpolación y se utiliza el aumento obtenido en esta función para copiar este valor dentro de alguna de las variables globales y se manda a donde se dibuja los modelos y sea posible observar los diferentes cambios:

```
1167        //Movimiento del personaje BMO
1168    if (playB)
1169    {
1170        if (i_curr_steps >= i_max_steps)
1171        {
1172            playIndex++;
1173            if (playIndex > MAX_FRAMES_BMO - 2)
1174            {
1175                printf("termina anim\n");
1176                playIndex = 0;
1177                playB = false;
1178            }
1179            else
1180            {
1181                i_curr_steps = 0; //Reset counter
1182                                  //Interpolation
1183                interpolation();
1184            }
1185        }
1186        else
1187        {
1188            rotPiernaIzqBmo += KeyFrameBMO[playIndex].rotIncPiernaIzqBmo;
1189            rotPiernaDerBmo += KeyFrameBMO[playIndex].rotIncPiernaDerBmo;
1190            rotBrazoDerBmo += KeyFrameBMO[playIndex].rotIncBrazoDerBmo;
1191            rotBrazoIzqBmo += KeyFrameBMO[playIndex].rotIncBrazoIzqBmo;
1192
1193            i_curr_steps++;
1194        }
1195
1196    }
```

La función “KeyCallback” ayuda a detectar cuando alguna de las animaciones por key frames es activada o cuando se desea salir de la ventana, esto es al presionar algunas de las teclas seleccionadas, las cuales son; la tecla F activa la animación del personaje “Finn”, esta animación solo puede ser activada una vez, la tecla B activa la animación del personaje BMO, esta animación puede repetirse n cantidad de veces y la tecla Esc permite terminar con el programa y salir de la ventana.

```
1244    // Is called whenever a key is pressed/released via GLFW
1245    void KeyCallback(GLFWwindow *window, int key, int scancode, int action, int mode)
1246    {
1247        //tecla B es asignada para activar la animación de BMO
1248        if (keys[GLFW_KEY_B])
1249        {
1250            if (playB == false && (FrameIndexBMO > 1))
1251            {
1252                resetElements();
1253                //First Interpolation
1254                interpolation();
1255                playB = true;
1256                playIndex = 0;
1257                i_curr_steps = 0;
1258            }
1259            else
1260            {
1261                playB = false;
1262            }
1263        }
1264
```

```
1264
1265        //tecla F es asignada para activar la animación de Finn
1266        if (keys[GLFW_KEY_F])
1267        {
1268            if (playF == false && (FrameIndexFinn > 1))
1269            {
1270                //First Interpolation
1271                interpolacionF();
1272                playF = true;
1273                playIndexF = 0;
1274                i_curr_steps_f = 0;
1275            }
1276            else
1277            {
1278                playF = false;
1279            }
1280        }
1281
1282        if (GLFW_KEY_ESCAPE == key && GLFW_PRESS == action)
1283        {
1284            glfwSetWindowShouldClose(window, GL_TRUE);
1285        }
1286
```

La función “MouseCallback” permite mover la cámara en las componentes “x” y “y” dentro del ambiente usando el movimiento del mouse.

```
1316    void MouseCallback(GLFWwindow *window, double xPos, double yPos)
1317    {
1318
1319        if (firstMouse)
1320        {
1321            lastX = xPos;
1322            lastY = yPos;
1323            firstMouse = false;
1324        }
1325
1326        GLfloat xOffset = xPos - lastX;
1327        GLfloat yOffset = lastY - yPos;  // Reversed since y-coordinates go from bottom to left
1328
1329        lastX = xPos;
1330        lastY = yPos;
1331
1332        camera.ProcessMouseMovement(xOffset, yOffset);
1333    }
```

La Función “DoMovement” contiene la funcionalidad para activar las animaciones sencillas y para poder moverse hacia adelante, atras, izquierda y derecha dentro del ambiente, según sea la tecla que se presione. La tecla P activa la animación del pato de hule que se encuentra dentro de la tina de baño, la tecla C activa la animación del caracol que está debajo del escritorio café, la tecla H activa la animación del hotcake que se encuentra dentro del sartén sobre la estufa, la tecla W permite moverse hacia adelante, la tecla S hacia atrás, la tecla A hacia la izquierda y la tecla D hacia la derecha, estas últimas cuatro teclas toman de referencia hacia donde se esté viendo con la cámara.

```
1358
1359        // Camera controls
1360        if (keys[GLFW_KEY_W] || keys[GLFW_KEY_UP])
1361        {
1362            camera.ProcessKeyboard(FORWARD, deltaTime);
1363
1364        }
1365
1366        if (keys[GLFW_KEY_S] || keys[GLFW_KEY_DOWN])
1367        {
1368            camera.ProcessKeyboard(BACKWARD, deltaTime);
1369
1370
1371        }
1372
1373        if (keys[GLFW_KEY_A] || keys[GLFW_KEY_LEFT])
1374        {
1375            camera.ProcessKeyboard(LEFT, deltaTime);
1376
1377
1378        }
1379
1380        if (keys[GLFW_KEY_D] || keys[GLFW_KEY_RIGHT])
1381        {
1382            camera.ProcessKeyboard(RIGHT, deltaTime);
1383        }
1384
```

La función “main” como su nombre lo indica es la función principal del programa.
En un inicio se inicializa la ventana donde veremos nuestro ambiente, se le asignan los valores de ancho, alto y el nombre que queremos asignarle a la ventana, se revisa en caso de que haya ocurrido un error para poder mandar un mensaje a la consola o mandar la información al buffer para obtener la ventana:

```
363 int main()
364 {
365     // Init GLFW
366     glfwInit(); //inciciacion de la ventana
367
368
369     // Create a GLFWwindow object that we can use for GLFW's functions
370     //Ininiciación de la ventana con ancho, alto, nombre de la ventana c
371     GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Proyecto Final Barcenas Jorge", nullptr, nullptr);
372
373     //se revisa si la existe algun error y en caso de que lo haya se manda un mensaje de error a la consola
374     if (nullptr == window)
375     {
376         std::cout << "Failed to create GLFW window" << std::endl;
377         glfwTerminate();
378
379         return EXIT_FAILURE;
380     }
381
382     glfwMakeContextCurrent(window);
383
384     //se manda la información al buffer para que pinte la ventana
385     glfwGetFramebufferSize(window, &SCREEN_WIDTH, &SCREEN_HEIGHT);
```

Se verifica si existe un error con GLEW para que en caso de haberlo se mande un error, en caso de no haberlo se define el viewport con sus dimensiones:

```
395     // Initialize GLEW to setup the OpenGL Function pointers
396     //se verifica si existe error con GLEW
397     if (GLEW_OK != glewInit())
398     {
399         std::cout << "Failed to initialize GLEW" << std::endl;
400         return EXIT_FAILURE;
401     }
402
403     // Define the viewport dimensions
404     glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
405
```

Se importan los archivos vertex shader y fragment shader para poder crear los shaders, al igual que se crean los objetos de tipo Model que reciben la ruta para importar los archivos object de nuestro modelos:

```
411     Shader lightingShader("Shaders/lighting.vs", "Shaders/lighting.frag");
412     Shader lampShader("Shaders/lamp.vs", "Shaders/lamp.frag");
413     Shader SkyBoxshader("Shaders/SkyBox.vs", "Shaders/SkyBox.frag");
414
415     //muebles
416     Model Fachada((char*)"Models/fachada/fachada.obj");
417     Model Hielera((char*)"Models/hielera/hielera.obj");
418     Model LavaManos((char*)"Models/lavamanos/lavamanos.obj");
419     Model Estufa((char*)"Models/estufa/estufa.obj");
420     Model Sarten((char*)"Models/sarten/sarten.obj");
421     Model Hotcake((char*)"Models/sarten/hotcake.obj");
422     Model Escritorio((char*)"Models/escritorio/escritorio.obj");
423     Model LavaPlatos((char*)"Models/lavaPlatos/lavaplatos.obj");
424     Model SillonCafe((char*)"Models/sillonCafe/sillonCafe.obj");
425     Model SillonRojo((char*)"Models/sillonRojo/sillonrojo.obj");
426     Model Wc((char*)"Models/wc/wc.obj");
427     Model TinaBaño((char*)"Models/tinaCortina/tina.obj");
428     Model CortinaBaño((char*)"Models/tinaCortina/cortina.obj");
429     //personajes
430     Model Pato((char*)"Models/pato/pato.obj");
431
432     Model Caracol((char*)"Models/caracol/caracol.obj");
433
434     Model BmoCuerpo((char*)"Models/bmo/cuerpo.obj");
435     Model BmoBrazoDerecho((char*)"Models/bmo/brazoDerecho.obj");
436     Model BmoBrazoIzquierdo((char*)"Models/bmo/brazoIzquierdo.obj");
437               //usaremos el mismo modelo ya que son iguales
438     Model BmoPierna((char*)"Models/bmo/pierna.obj");
439
440     Model FinnTorso((char*)"Models/finn/torsofinn.obj");
441     Model FinnPierIzq((char*)"Models/finn/piernaizquierda.obj");
442     Model FinnPierDer((char*)"Models/finn/piernaderecha.obj");
443
```

Se inicializan las posiciones de las animaciones por key frames:

```
449     for(int i=0; i<MAX_FRAMES_BMO; i++)
450     {
451
452         KeyFrameBMO[i].rotPiernaIzqBmo = 0;
453         KeyFrameBMO[i].rotIncPiernaIzqBmo = 0;
454         KeyFrameBMO[i].rotPiernaDerBmo = 0;
455         KeyFrameBMO[i].rotIncPiernaDerBmo = 0;
456         KeyFrameBMO[i].rotBrazoIzqBmo = 0;
457         KeyFrameBMO[i].rotIncBrazoIzqBmo = 0;
458         KeyFrameBMO[i].rotBrazoDerBmo = 0;
459         KeyFrameBMO[i].rotIncBrazoDerBmo = 0;
460     }
461
462     for (int i = 0; i < MAX_FRAMES_FINN; i++)
463     {
464         KeyFrameFinn[i].posX = 0;
465         KeyFrameFinn[i].incX = 0;
466         KeyFrameFinn[i].incZ = 0;
467         KeyFrameFinn[i].rotPiernaIzqFinn = 0;
468         KeyFrameFinn[i].rotIncPiernaIzqFinn = 0;
469         KeyFrameFinn[i].rotPiernaDerFinn = 0;
470         KeyFrameFinn[i].rotIncPiernaDerFinn = 0;
471         KeyFrameFinn[i].rot = 0;
472         KeyFrameFinn[i].rotInc = 0;
473     }
474
```

Se crean un arreglo que contiene los vértices de posición, normales y coordenadas en texturas, un arreglo que contiene los vértices usados para la creación del cubo con el cual se texturiza el ambiente y un arreglo donde se tendrán los índices:

```
478     GLfloat vertices[] =
```

```
526     GLfloat skyboxVertices[] = {
```

```
572     GLuint indices[] =
```

Se reserva memoria para un arreglo de vértices (VAO) y dos buffers (VBO y EBO), se enlaza el buffer, se le asigna el array object, se obtiene el VBO y se le manda la información que trabajaremos, el tamaño del arreglo y el arreglo mismo de vértices:

```
590     GLuint VBO, VAO, EBO;
591     glGenVertexArrays(1, &VAO);
592     glGenBuffers(1, &VBO);
593     glGenBuffers(1, &EBO);
594
595
596     glBindVertexArray(VAO);
597     glBindBuffer(GL_ARRAY_BUFFER, VBO);
598     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
599
600     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
601     glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Se indica las posiciones donde se encontrarán, las coordenadas de posiciones, las normales y las coordenadas de texturas:

```
605 // Position attribute
606 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid *)0);
607 glEnableVertexAttribArray(0); //se enlazan
608 // Normals attribute
609 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid *)(3 * sizeof(GLfloat)));
610 glEnableVertexAttribArray(1);
611 // Texture Coordinate attribute
612 glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid *)(6 * sizeof(GLfloat)));
613 glEnableVertexAttribArray(2);
614 glBindVertexArray(0); //se desenlaza para limpiar la memoria
```

Obtenemos los vertex buffer object y el vertex array object usados para el skybox:

```
628 //SkyBox
629 GLuint skyboxVBO, skyboxVAO;
630 glGenVertexArrays(1, &skyboxVAO);
631 glGenBuffers(1,&skyboxVBO);
632 glBindVertexArray(skyboxVAO);
633 glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
634 glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices),&skyboxVertices,GL_STATIC_DRAW);
635 glEnableVertexAttribArray(0);
636 glVertexAttribPointer(0, 3, GL_FLOAT,GL_FALSE, 3 * sizeof(GLfloat), (GLvoid *)0);
```

Se indican la direcciones donde se encuentran las imágenes usadas para la texturización del ambiente y se manda a cargar a memoria:

```
638     // arreglo de caras del cubo
639     vector<const GLchar*> faces;
640     faces.push_back("SkyBox/left.tga");
641     faces.push_back("SkyBox/right.tga");
642     faces.push_back("SkyBox/top.tga");
643     faces.push_back("SkyBox/bottom.tga");
644     faces.push_back("SkyBox/back.tga");
645     faces.push_back("SkyBox/front.tga");
646
647     //se manda a la biblioteca a texturizar y se carge en memoria
648     GLuint cubemapTexture = TextureLoading::LoadCubemap(faces);
649
```

Indicamos el tipo de proyección que usaremos, en este caso usaremos una proyección en perspectiva y le pasamos la apertura de la cámara sintética:

```
650     //se trabajará con una proyección de perspectiva
651     glm::mat4 projection = glm::perspective(camera.GetZoom(), (GLfloat)SCREEN_WIDTH / (GLfloat)SCREEN_HEIGHT, 0.1f, 1000.0f);
652
```

Entramos en un ciclo en donde constantemente se revisará si hay cambios en los objetos, en un inicio se calcula el deltatime entre el último frame y el frame actual:

```
654     while (!glfwWindowShouldClose(window))
655     {
656
657         // Calculate deltatime of current frame
658         GLfloat currentFrame = glfwGetTime();
659         deltaTime = currentFrame - lastFrame;
660         lastFrame = currentFrame;
```

Se mandan a llamar las siguientes funciones que nos permiten cachar eventos provenientes de teclado:

```
664         glfwPollEvents();
665         DoMovement();
666         animacion();
```

Declaramos el uso de luz direccional, 4 point lights y un spotlight, en el caso la luz direccional es necesario especificar la su dirección, su componente ambiental, difusa y especular, a las point light es necesario especificar su posición, su componente ambiental, difusa, especular, constante, lineal y cuadrática, y en el caso de la spotlight necesita de las mismas componentes que la point light, además del valor para su cono interior y su cono exterior :

```
//declaramos el uso de luz direccional, los 4 point light que usaremos y el spotlight
// Directional light
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.2f, -1.0f, -0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.3f, 0.3f, 0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.4f, 0.4f, 0.4f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 0.5f, 0.5f, 0.5f);


// Point light 1
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].position"), pointLightPositions[0].x, pointLightPositions[0].y, pointLightPositions[0].z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].ambient"), 0.5f, 0.5f, 0.5f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].diffuse"), 0.05f, 0.05f, 0.05f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].specular"), 0.05f, 0.05f, 0.05f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].linear"), 0.09f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].quadratic"), 0.032f);



// Point light 2
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].position"), pointLightPositions[1].x, pointLightPositions[1].y, pointLightPositions[1].z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].ambient"), 0.5f, 0.5f, 0.5f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].diffuse"), 0.05f, 0.05f, 0.05f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].specular"), 0.05f, 0.05f, 0.05f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].linear"), 0.09f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].quadratic"), 0.032f);

// Point light 3
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].position"), pointLightPositions[2].x, pointLightPositions[2].y, pointLightPositions[2].z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].ambient"), 0.5f, 0.5f, 0.5f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].diffuse"), 0.05f, 0.05f, 0.05f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].specular"), 0.05f, 0.05f, 0.05f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].linear"), 0.09f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].quadratic"), 0.032f);

// Point light 4
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].position"), pointLightPositions[3].x, pointLightPositions[3].y, pointLightPositions[3].z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].ambient"), 0.5f, 0.5f, 0.5f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].diffuse"), 0.05f, 0.05f, 0.05f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].specular"), 0.05f, 0.05f, 0.05f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].linear"), 0.09f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].quadratic"), 0.032f);

// SpotLight
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.position"), camera.GetPosition().x, camera.GetPosition().y, camera.GetPosition().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.direction"), camera.GetFront().x, camera.GetFront().y, camera.GetFront().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.ambient"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.linear"), 0.09f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.quadratic"), 0.032f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.cutOff"), glm::cos(glm::radians(12.5f)));
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.outerCutOff"), glm::cos(glm::radians(15.0f)));
```

Se declaran las matrices de vista, proyección y modelo y estas dos primeras se mandan al shader:

```
GLint modelLoc = glGetUniformLocation(lightingShader.Program, "model");
GLint viewLoc = glGetUniformLocation(lightingShader.Program, "view");
GLint projLoc = glGetUniformLocation(lightingShader.Program, "projection");

// Pass the matrices to the shader
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));
```

Se hace el dibujado de los modelos, para todos los objetos se necesita de la obtención de la matriz de vista proveniente del archivo “camera.h”, se setea la matriz de modelo, dependiendo de que se necesite se puede trasladar o rotar el objeto y finalmente se le informa al sheader que se ha realizado un cambio en la matriz de modelo y se manda a dibujar.

La mayoría de los objetos son importados como un solo modelo, pero en aquellos modelos en los que aplicaremos una animación por keyframe son importados en diferentes “piezas” y serán acomodados dentro del ambiente utilizando modelado jerárquico. Además, en todos

los objetos que tienen alguna animación es necesario utilizar variables ya sea para poder moverse en alguna dirección o para hacer alguna rotación.

A continuación se crean las matrices de vista, modelo y proyección necesarias para la creación de los cubos que representarán el origen de los point light y repetimos el proceso usado con los objetos de nuestro ambiente, pero en este caso se usa un ciclo para que nos ayudará a obtener las posiciones de cada uno y dibujarlos uno a uno:

```
991
992         // Set matrices
993         glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
994         glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));
995         model = glm::mat4(1);
996         model = glm::translate(model, lightPos);
997         //model = glm::scale(model, glm::vec3(0.2f)); // Make it a smaller cube
998         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
999         // Draw the light object (using light's vertex attributes)
1000        glBindVertexArray(lightVAO);
1001        for (GLuint i = 0; i < 4; i++)
1002        {
1003            model = glm::mat4(1);
1004            model = glm::translate(model, pointLightPositions[i]);
1005            model = glm::scale(model, glm::vec3(0.2f)); // Make it a smaller cube
1006            glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
1007            glDrawArrays(GL_TRIANGLES, 0, 36);
1008        }
1009        glBindVertexArray(0);
1010
```

Una vez saliendo del ciclo while, liberamos la memoria usada por los VAO, VBO y EBO usados y salimos de la función “main”.

```
    glDeleteVertexArrays(1, &VAO);
    glDeleteVertexArrays(1, &lightVAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    glDeleteVertexArrays(1, &skyboxVAO);
    glDeleteBuffers(1, &skyboxVBO);
    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();



    return 0;
```

Conclusión

En conclusión es posible decir que el proyecto se ha finalizado de manera exitosa, cumpliendo cada uno de los requisitos teóricos y demostrando que hemos tenido una buena comprensión de los mismos. Además de tomar conciencia de la complejidad e importancia de la computación gráfica en el mundo actual.