MANUAL TECNICO

Ambiente Virtual

LABORATORIO DE COMPUTACION GRAFICA E INTERACCION HUMANO COMPUTADORA

GRUPO 08
Herrera Cordero Gustavo

Contenido

Objetivos

- El alumno aplicara lo visto durante el curso y generara un ambiente virtual con referencia a una propuesta.
- El alumno diseñara, modelara y posicionara elementos creados por el mismo, así como recursos que se obtuvieron de internet.
- El alumno documentará lo realizado en un repositorio de GitHub.

Diagrama de Gantt

Entorno Virtual

El siguiente Diagrama de Gantt refleja el tiempo que se tomo para realizar las actividades correspondientes al proyecto final de la materia Laboratorio de Computación Grafica e Interacción Humano-Computadora del Grupo 08.

Actividad/ Día	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Modelado de Objetos	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓									
Manual de Usuario																			✓	✓
Manual Técnico																			✓	✓
Carga de Modelos en VS											✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Desarrollo de Código															✓	✓	✓	✓	✓	✓
Animaciones															✓	✓	✓	✓	✓	

Alcance del proyecto

Este proyecto tiene como objetivo primordial que el alumno desarrolle los conocimientos adquiridos a lo largo del curso, de igual manera este proyecto final tiene como alcance el desarrollo, aprendizaje y practica de los conocimientos en programación que el alumno viene solventando a lo largo de su carrera universitaria. De igual manera, se tiene como uno de los principales alcances el aplicar los conocimientos de la materia, esto es aprender a modelar en software de diseño como lo son Maya, 3D Max, Blender, etc. Esto con el objetivo de que el alumno sea capaz de comprender la complejidad de crear objetos mediante primitivas básicas, además, de la texturización de dichos objetos y el correcto manejo de los canales de colores.

A su vez, el alumno emplea técnicas vistas durante el curso sobre animación y a representar movimientos cotidianos en un entorno virtual, de nuevo, mediante los conocimientos adquiridos en la materia así como en la parte teórica.

Documentación de código

```
#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"
```

```
// 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();

// Window dimensions
const GLuint WIDTH = 1240, HEIGHT = 920;
int SCREEN_WIDTH, SCREEN_HEIGHT;

// Camara
Camera  camera(glm::vec3(0.0f, -40.0f, 90.0f));
GLfloat lastX = WIDTH / 2.0;
GLfloat lastY = HEIGHT / 2.0;
bool keys[1024];
bool firstMouse = true;
float range = 0.0f;

// Atributos para iluminacion
glm::vec3 lightPos(0.0f, 0.0f, 0.0f);
glm::vec3 LightP1(0.0f, 0.0f, 0.0f);
glm::vec3 PosIni(-55.0f, -48.0f, -40.0f);
bool active;

// Posicion de PointLights
glm::vec3 pointLightPositions[] = {
        glm::vec3(0.7f,  60.2f,  2.0f),
        glm::vec3(0.7f,  60.2f,  2.0f),
```

```
        glm::vec3(0.7f,  60.2f,  2.0f),
        glm::vec3(0.7f,  60.2f,  2.0f)
};

//variables globales dentro del entorno
float movKitX = 0.0;
float movKitZ = 0.0;
float rotKit = 0.0;
bool circuito = false;
bool recorrido1 = true;
bool recorrido2 = false;
bool recorrido3 = false;
bool recorrido4 = false;
bool recorrido5 = false;

//Variables globales para animacion
float door,door1,chest= 0.0f;
glm::vec3 jau(0.0f,1.0f,0.0f);

// Deltatime
GLfloat deltaTime = 0.0f;   // Time between current frame and last frame
GLfloat lastFrame = 0.0f;   // Time of last frame

// Keyframes
float posX = PosIni.x, posY = PosIni.y, posZ = PosIni.z, rotder,rotizq,rotc = 0;

#define MAX_FRAMES 9
int i_max_steps = 50;
```

```
int i_curr_steps = 0;
typedef struct _frame
{
	//Variables para GUARDAR Key Frames
	float posX;		//Variable para PosicionX
	float posY;		//Variable para PosicionY
	float posZ;		//Variable para PosicionZ
	float incX;		//Variable para IncrementoX
	float incY;		//Variable para IncrementoY
	float incZ;		//Variable para IncrementoZ
	float rotder;   //Variable para rotacion en Brazo derecho
	float rotizq;   //Variable para rotacion en brazo izquierdo
	float rotc;     //variable para rotacion en cuerpo del personaje
	float rotInc;   //Variable para rotacion gradual
	float rotInc2;  //Variable para rotacion gradual
	float rotInc3;  //Variable para rotacion gradual
}FRAME;

FRAME KeyFrame[MAX_FRAMES];
int FrameIndex = 7;			//introducir datos
bool play = false;
int playIndex = 0;

void resetElements(void)
{
	posX = KeyFrame[0].posX;
	posY = KeyFrame[0].posY;
	posZ = KeyFrame[0].posZ;
```

```
	rotder = KeyFrame[0].rotder;
	rotizq = KeyFrame[0].rotizq;
	rotc = KeyFrame[0].rotc;
}
//Funcion de interpolacion de Keyframes
void interpolation(void)
{

	KeyFrame[playIndex].incX = (KeyFrame[playIndex + 1].posX - KeyFrame[playIndex].posX) / i_max_steps;
	KeyFrame[playIndex].incY = (KeyFrame[playIndex + 1].posY - KeyFrame[playIndex].posY) / i_max_steps;
	KeyFrame[playIndex].incZ = (KeyFrame[playIndex + 1].posZ - KeyFrame[playIndex].posZ) / i_max_steps;

	KeyFrame[playIndex].rotInc = (KeyFrame[playIndex + 1].rotder - KeyFrame[playIndex].rotder) / i_max_steps;
	KeyFrame[playIndex].rotInc2 = (KeyFrame[playIndex + 1].rotizq - KeyFrame[playIndex].rotizq) / i_max_steps;
	KeyFrame[playIndex].rotInc3 = (KeyFrame[playIndex + 1].rotc - KeyFrame[playIndex].rotc) / i_max_steps;
}

int main()
{
	// Init GLFW
	glfwInit();
	// Set all the required options for GLFW
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
```

```
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

	// Create a GLFWwindow object that we can use for GLFW's functions
	GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Proyecto
Final Gustavo Herrera Cordero", nullptr, nullptr);

	if (nullptr == window)
	{
		std::cout << "Failed to create GLFW window" << std::endl;
		glfwTerminate();

		return EXIT_FAILURE;
	}

	glfwMakeContextCurrent(window);

	glfwGetFramebufferSize(window, &SCREEN_WIDTH,
&SCREEN_HEIGHT);

	// Set the required callback functions
	glfwSetKeyCallback(window, KeyCallback);
	glfwSetCursorPosCallback(window, MouseCallback);
	printf("%f", glfwGetTime());

	// GLFW Options
```

```
	glfwSetInputMode(window, GLFW_CURSOR,
GLFW_CURSOR_DISABLED);


	// Set this to true so GLEW knows to use a modern approach to retrieving
function pointers and extensions
	glewExperimental = GL_TRUE;
	// Initialize GLEW to setup the OpenGL Function pointers
	if (GLEW_OK != glewInit())
	{
		std::cout << "Failed to initialize GLEW" << std::endl;
		return EXIT_FAILURE;
	}

	// Define the viewport dimensions
	glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);

	// OpenGL options
	glEnable(GL_DEPTH_TEST);
	glEnable(GL_BLEND);
	glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

	Shader lightingShader("Shaders/lighting.vs", "Shaders/lighting.frag");
	Shader lampShader("Shaders/lamp.vs", "Shaders/lamp.frag");
	Shader SkyBoxshader("Shaders/SkyBox.vs", "Shaders/SkyBox.frag");

	//Carga de modelos realizados externamente
	Model piña((char*)"Models/Piña/pina.obj");
	Model puerta((char*)"Models/puerta/puerta.obj");
	Model cofre((char*)"Models/cofre/cofre.obj");
```

```
Model cofre2((char*)"Models/cofre/cofre2.obj");
Model jaula((char*)"Models/jaula/jaula.obj");
Model bote((char*)"Models/bote/bote.obj");
Model bob((char*)"Models/bob/bobc.obj");
Model bobizq((char*)"Models/bob/bobizq.obj");
Model bobder((char*)"Models/bob/bobder.obj");

//Inicializacion de los Keyframe para almacenar datos
for (int i = 0; i < MAX_FRAMES; i++)
{
	KeyFrame[i].posX = 0;
	KeyFrame[i].incX = 0;
	KeyFrame[i].incY = 0;
	KeyFrame[i].incZ = 0;
	KeyFrame[i].rotder = 0;
	KeyFrame[i].rotInc = 0;
	KeyFrame[i].rotizq = 0;
	KeyFrame[i].rotInc2 = 0;
	KeyFrame[i].rotc = 0;
	KeyFrame[i].rotInc3 = 0;

}
//Valores que toman los Keyframe para animacion
KeyFrame[0].rotder = 0;
KeyFrame[1].rotder = 15;
KeyFrame[2].rotder = -15;
KeyFrame[3].rotder = 15;
KeyFrame[4].rotder = -15;
```

```
KeyFrame[0].rotizq = 0;
KeyFrame[1].rotizq = 15;
KeyFrame[2].rotizq = -15;
KeyFrame[3].rotizq = 15;
KeyFrame[4].rotizq = -15;
KeyFrame[0].rotc = 0;
KeyFrame[1].rotc = 15;
KeyFrame[2].rotc = -15;
KeyFrame[3].rotc = 15;
KeyFrame[4].rotc = -15;

// Set up vertex data (and buffer(s)) and attribute pointers
GLfloat vertices[] =
{
        // Positions          // Normals           // Texture Coords
        -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,
        0.5f, -0.5f, -0.5f,   0.0f,  0.0f, -1.0f,  1.0f,  0.0f,
        0.5f,  0.5f, -0.5f,   0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
        0.5f,  0.5f, -0.5f,   0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
        -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  1.0f,
        -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,

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

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

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

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

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

```
};

//vertices para la creacion del Skybox
GLfloat skyboxVertices[] = {
        // Positions
        -1.0f,  1.0f, -1.0f,
        -1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,

        -1.0f, -1.0f,  1.0f,
        -1.0f, -1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,

        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,

        -1.0f, -1.0f,  1.0f,
        -1.0f,  1.0f,  1.0f,
```

```
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f, -1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,

        -1.0f,  1.0f, -1.0f,
        1.0f,  1.0f, -1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f, -1.0f,

        -1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f,  1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f,  1.0f,
        1.0f, -1.0f,  1.0f
};

//Generacion de primitivas
GLuint indices[] =
{  // Note that we start from 0!
        0,1,2,3,
        4,5,6,7,
        8,9,10,11,
        12,13,14,15,
        16,17,18,19,
```

```
        20,21,22,23,
        24,25,26,27,
        28,29,30,31,
        32,33,34,35
};

// Positions all containers
glm::vec3 cubePositions[] = {
        glm::vec3(0.0f,  0.0f,  0.0f),
        glm::vec3(2.0f,  5.0f, -15.0f),
        glm::vec3(-1.5f, -2.2f, -2.5f),
        glm::vec3(-3.8f, -2.0f, -12.3f),
        glm::vec3(2.4f, -0.4f, -3.5f),
        glm::vec3(-1.7f,  3.0f, -7.5f),
        glm::vec3(1.3f, -2.0f, -2.5f),
        glm::vec3(1.5f,  2.0f, -2.5f),
        glm::vec3(1.5f,  0.2f, -1.5f),
        glm::vec3(-1.3f,  1.0f, -1.5f)
};


// First, set the container's VAO (and VBO)
GLuint VBO, VAO, EBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);

glBindVertexArray(VAO);
```

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


        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);


        // Position attribute
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat),
(GLvoid*)0);
        glEnableVertexAttribArray(0);
        // Normals attribute
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat),
(GLvoid*)(3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(1);
        // Texture Coordinate attribute
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat),
(GLvoid*)(6 * sizeof(GLfloat)));
        glEnableVertexAttribArray(2);
        glBindVertexArray(0);


        // Then, we set the light's VAO (VBO stays the same. After all, the vertices
are the same for the light object (also a 3D cube))
        GLuint lightVAO;
        glGenVertexArrays(1, &lightVAO);
        glBindVertexArray(lightVAO);
        // We only need to bind to the VBO (to link it with glVertexAttribPointer), no
need to fill it; the VBO's data already contains all we need.
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        // Set the vertex attributes (only position data for the lamp))
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)0); // Note that we skip over the other data in our buffer object (we don't need the normals/textures, only positions).

glEnableVertexAttribArray(0);

glBindVertexArray(0);


//SkyBox

GLuint skyboxVBO, skyboxVAO;

glGenVertexArrays(1, &skyboxVAO);

glGenBuffers(1, &skyboxVBO);

glBindVertexArray(skyboxVAO);

glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);

glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices, GL_STATIC_DRAW);

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);


// Carga de Texturas para el Skybox

vector<const GLchar*> faces;

faces.push_back("SkyBox/lado1.tga");

faces.push_back("SkyBox/lado1.tga");

faces.push_back("SkyBox/sky.tga");//cielo

faces.push_back("SkyBox/suelo.tga");//suelo

faces.push_back("SkyBox/lado1.tga");

faces.push_back("SkyBox/lado1.tga");


GLuint cubemapTexture = TextureLoading::LoadCubemap(faces);
```

```
	glm::mat4 projection = glm::perspective(camera.GetZoom(), (GLfloat)SCREEN_WIDTH / (GLfloat)SCREEN_HEIGHT, 0.1f, 1000.0f);


	//Carga de texturas para elementos creados por primitivas
	GLuint texture1;
	glGenTextures(1, &texture1);


	int textureWidth, textureHeight, nrChannels;
	stbi_set_flip_vertically_on_load(true);
	unsigned char* image;
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST_MIPMAP_NEAREST);
	// Diffuse map
	image = stbi_load("images/escotilla.jpg", &textureWidth, &textureHeight, &nrChannels, 0);
	glBindTexture(GL_TEXTURE_2D, texture1);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, textureWidth, textureHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
	glGenerateMipmap(GL_TEXTURE_2D);
	if (image)
	{
		glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, textureWidth, textureHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
		glGenerateMipmap(GL_TEXTURE_2D);
	}
```

```
else
{
    std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(image);


// Game loop
while (!glfwWindowShouldClose(window))
{

    // Calculate deltatime of current frame
    GLfloat currentFrame = glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // Check if any events have been activiated (key pressed, mouse moved etc.) and call corresponding response functions
    glfwPollEvents();
    DoMovement();
    animacion();


    // Clear the colorbuffer
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //Load Model
```

```
// Use cooresponding shader when setting uniforms/drawing objects
lightingShader.Use();
GLint viewPosLoc = glGetUniformLocation(lightingShader.Program, "viewPos");
glUniform3f(viewPosLoc, camera.GetPosition().x, camera.GetPosition().y, camera.GetPosition().z);
// Set material properties
glUniform1f(glGetUniformLocation(lightingShader.Program, "material.shininess"), 32.0f);
// == ==========================
// Here we set all the uniforms for the 5/6 types of lights we have. We have to set them manually and index
// the proper PointLight struct in the array to set each uniform variable. This can be done more code-friendly
// by defining light types as classes and set their values in there, or by using a more efficient uniform approach
// by using 'Uniform buffer objects', but that is something we discuss in the 'Advanced GLSL' tutorial.
// == ==========================
// Directional light
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.2f, -1.0f, -0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.93f, 0.93f, 0.93f);
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"), 0.05f, 0.05f, 0.05f);

        glUniform3f(glGetUniformLocation(lightingShader.Program,
"pointLights[0].diffuse"), LightP1.x, LightP1.y, LightP1.z);

        glUniform3f(glGetUniformLocation(lightingShader.Program,
"pointLights[0].specular"), LightP1.x, LightP1.y, LightP1.z);

        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"), 0.05f, 0.05f, 0.05f);

        glUniform3f(glGetUniformLocation(lightingShader.Program,
"pointLights[1].diffuse"), 1.0f, 1.0f, 0.0f);

        glUniform3f(glGetUniformLocation(lightingShader.Program,
"pointLights[1].specular"), 1.0f, 1.0f, 0.0f);

        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"), 0.05f, 0.05f, 0.05f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].diffuse"), 0.0f, 1.0f, 1.0f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].specular"), 0.0f, 1.0f, 1.0f);
        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"), 0.05f, 0.05f, 0.05f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].diffuse"), 1.0f, 0.0f, 1.0f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].specular"), 1.0f, 0.0f, 1.0f);
        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)));

        // Set material properties
        glUniform1f(glGetUniformLocation(lightingShader.Program, "material.shininess"), 32.0f);

        // Create camera transformations
        glm::mat4 view;
        view = camera.GetViewMatrix();
```

```
		// Get the uniform locations
		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));

		glBindVertexArray(VAO);
		glm::mat4 model(1);

		//Dibujo de modelos previamente cargados

		//piña
		glm::mat4 model1(1);
		model1 = glm::translate(model1, glm::vec3(0.0f, -50.0f, 0.0f));
		glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model1));
		piña.Draw(lightingShader);
		//puerta
		glm::mat4 model2(1);
		model2 = glm::rotate(model2, glm::radians(door), glm::vec3(0.0f, 1.0f, 0.0f));
		model2 = glm::translate(model2, glm::vec3(-3.5f, -45.0f, 20.5f));
		model2 = glm::rotate(model2, glm::radians(door1), glm::vec3(0.0f, 1.0f, 0.0f));
```

```
		glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model2));
		puerta.Draw(lightingShader);
		//PuertaTrasera
		glm::mat4 model3(1);
		model3 = glm::translate(model3, glm::vec3(3.5f, -45.0f, -19.5f));
		model3 = glm::rotate(model3, glm::radians(-18.0f), glm::vec3(0.0f,
1.0f, 0.0f));
		glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model3));
		puerta.Draw(lightingShader);
		//cofre
		glm::mat4 model4(1);
		model4 = glm::translate(model4, glm::vec3(12.5f, -49.0f, 0.5f));
		model4 = glm::scale(model4, glm::vec3(2.0f));
		model4 = glm::rotate(model4, glm::radians(200.0f), glm::vec3(0.0f,
1.0f, 0.0f));
		glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model4));
		cofre.Draw(lightingShader);
		//Tapa de cofre
		glm::mat4 model5(1);
		model5 = glm::translate(model5, glm::vec3(12.5f, -49.0f, 0.5f));
		model5 = glm::scale(model5, glm::vec3(2.0f));
		model5 = glm::rotate(model5, glm::radians(200.0f), glm::vec3(0.0f,
1.0f, 0.0f));
		model5 = glm::rotate(model5, glm::radians(chest), glm::vec3(0.0f,
0.0f, 1.0f));
		glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model5));
		cofre2.Draw(lightingShader);
```

```
glBindVertexArray(0);
//jaula
glm::mat4 model6(1);
model6 = glm::translate(model6, glm::vec3(10.5f, -25.0f, 0.5f));
model6 = glm::scale(model6, glm::vec3(2.0f));
model6 = glm::rotate(model6, (float)glfwGetTime(), glm::vec3(jau));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model6));
jaula.Draw(lightingShader);
//escotilla con primitivas basicas
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glBindVertexArray(VAO);
glm::mat4 model7(1);
model7 = glm::translate(model7, glm::vec3(-5.0f, -49.0f, 11.5f));
model7 = glm::scale(model7, glm::vec3(5.0f, 0.2f, 5.0f));
model7 = glm::rotate(model7, glm::radians(90.0f), glm::vec3(1.0f,
0.0f, 0.0f));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model7));
glDrawArrays(GL_TRIANGLES, 0, 36);
//bote
glm::mat4 model8(1);
model8 = glm::translate(model8,
PosIni+glm::vec3(movKitX,0.0f,movKitZ));
model8 = glm::scale(model8, glm::vec3(2.0f));
model8 = glm::rotate(model8, glm::radians(rotKit), glm::vec3(0.0f,
1.0f, 0.0f));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model8));
```

```
		bote.Draw(lightingShader);
		//Cuerpo Bob
		glm::mat4 model9(1);
		model9 = glm::translate(model9, glm::vec3(20.0f, -40.0f, 32.0f));
		model9 = glm::scale(model9, glm::vec3(0.08f));
		model9 = glm::rotate(model9, glm::radians(-45.0f), glm::vec3(0.0f,
1.0f, 0.0f));
		model9 = glm::rotate(model9, glm::radians(-rotc), glm::vec3(0.0f, 1.0f,
1.0f));
		glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model9));
		bob.Draw(lightingShader);
		//Brazo izquierdo
		glm::mat4 model10(1);
		model10 = glm::translate(model10, glm::vec3(20.0f, -40.0f, 32.0f));
		model10 = glm::scale(model10, glm::vec3(0.08f));
		model10 = glm::rotate(model10, glm::radians(-45.0f), glm::vec3(0.0f,
1.0f, 0.0f));
		model10 = glm::rotate(model10, glm::radians(-rotizq), glm::vec3(0.0f,
1.0f, 1.0f));
		glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model10));
		bobizq.Draw(lightingShader);
		//Brazo derecho
		glm::mat4 model11(1);
		model11 = glm::translate(model11, glm::vec3(20.0f, -40.0f, 32.0f));
		model11 = glm::scale(model11, glm::vec3(0.08f));
		model11 = glm::rotate(model11, glm::radians(-rotder), glm::vec3(0.0f,
0.0f, 1.0f));
		model11 = glm::rotate(model11, glm::radians(-45.0f), glm::vec3(0.0f,
1.0f, 0.0f));
```

```
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model11));
        bobder.Draw(lightingShader);


        glBindVertexArray(0);


        // Also draw the lamp object, again binding the appropriate shader
        lampShader.Use();
        // Get location objects for the matrices on the lamp shader (these
could be different on a different shader)
        modelLoc = glGetUniformLocation(lampShader.Program, "model");
        viewLoc = glGetUniformLocation(lampShader.Program, "view");
        projLoc = glGetUniformLocation(lampShader.Program, "projection");


        // Set matrices
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE,
glm::value_ptr(projection));
        model = glm::mat4(1);
        model = glm::translate(model, lightPos);
        //model = glm::scale(model, glm::vec3(0.2f)); // Make it a smaller cube
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
        // Draw the light object (using light's vertex attributes)
        glBindVertexArray(lightVAO);
        for (GLuint i = 0; i < 4; i++)
        {
                model = glm::mat4(1);
                model = glm::translate(model, pointLightPositions[i]);
                model = glm::scale(model, glm::vec3(0.2f)); // Make it a smaller
cube
```

```
                glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model));
                glDrawArrays(GL_TRIANGLES, 0, 36);
            }
            glBindVertexArray(0);


            glDepthFunc(GL_LEQUAL);  // Change depth function so depth test
passes when values are equal to depth buffer's content
            SkyBoxshader.Use();
            view = glm::mat4(glm::mat3(camera.GetViewMatrix()));        //
Remove any translation component of the view matrix
            glUniformMatrix4fv(glGetUniformLocation(SkyBoxshader.Program,
"view"), 1, GL_FALSE, glm::value_ptr(view));
            glUniformMatrix4fv(glGetUniformLocation(SkyBoxshader.Program,
"projection"), 1, GL_FALSE, glm::value_ptr(projection));

            // skybox cube
            glBindVertexArray(skyboxVAO);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
            glDrawArrays(GL_TRIANGLES, 0, 36);
            glBindVertexArray(0);
            glDepthFunc(GL_LESS); // Set depth function back to default

            // Swap the screen buffers
            glfwSwapBuffers(window);
        }

        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;
}

// Moves/alters the camera positions based on user input
void DoMovement()
{
        // Controles de camara
        if (keys[GLFW_KEY_W] || keys[GLFW_KEY_UP])
        {
                camera.ProcessKeyboard(FORWARD, deltaTime);

        }

        if (keys[GLFW_KEY_S] || keys[GLFW_KEY_DOWN])
        {
                camera.ProcessKeyboard(BACKWARD, deltaTime);


        }

        if (keys[GLFW_KEY_A] || keys[GLFW_KEY_LEFT])
```

```
{
        camera.ProcessKeyboard(LEFT, deltaTime);



}

if (keys[GLFW_KEY_D] || keys[GLFW_KEY_RIGHT])
{
        camera.ProcessKeyboard(RIGHT, deltaTime);



}
//animacion abre y cierra puerta
if (keys[GLFW_KEY_T])
{
        door -= 0.05f;
        door1 -= 1.0f;
        if (door<90 && door1<10.0f)
        {
                door1 = -100.0f;
                door = -4.5f;
        }
}
if (keys[GLFW_KEY_G])
{
        door += 0.05f;
        door1 += 1.0f;
        if (door < 90 && door1 < 10.0f)
```

```
		{
			door1 = 0.0f;
			door = 0.0f;
		}
	}
	//Animacion abre y cirra cofre
	if (keys[GLFW_KEY_Y])
	{
		chest += 1.0f;
		if (chest > 60.0f )
		{
			chest = 60.0f;
		}
	}
	if (keys[GLFW_KEY_H])
	{
		chest -= 1.0f;
		if (chest < 0.0f)
		{
			chest = 0.0f;
		}
	}
	//Animacion de rotacion de Jaula
	if (keys[GLFW_KEY_U])
	{
		jau = glm::vec3(0.0f,1.0f,0.0f);
	}
	if (keys[GLFW_KEY_J])
```

```
	{
		jau= glm::vec3(0.0f, -1.0f, 0.0f);
	}
	//Animacion para automovil y el circuito fuera de la casa
	if (keys[GLFW_KEY_I])
	{
		circuito = true;
	}
	if (keys[GLFW_KEY_K])
	{
		circuito = false;
	}
	//Valores para animacion por KeyFrames
	if (keys[GLFW_KEY_L])
	{
		rotder = 15.0F;
		rotder = -15.0f;
		rotizq = 15.0f;
		rotizq = -15.0f;
		rotc = 15.0f;
		rotc = -15.0f;
	}
}

void animacion()
{
	//Inicio de animacion por KeyFrames
	if (play)
```

```
{
        if (i_curr_steps >= i_max_steps)
        {
                playIndex++;
                if (playIndex > FrameIndex - 2)    //end of total animation?
                {
                        printf("termina anim\n");
                        playIndex = 0;
                        play = false;
                }
                else //Next frame interpolations
                {
                        i_curr_steps = 0; //Reset counter
                                                //Interpolation
                        interpolation();
                }
        }
        else
        {
                //Draw animation
                posX += KeyFrame[playIndex].incX;
                posY += KeyFrame[playIndex].incY;
                posZ += KeyFrame[playIndex].incZ;

                rotder += KeyFrame[playIndex].rotInc;
                rotizq += KeyFrame[playIndex].rotInc2;
                rotc += KeyFrame[playIndex].rotInc3;
                i_curr_steps++;
```

```
        }

}

//Movimiento bote
if (circuito)
{
        if (recorrido1)
        {
                movKitZ += 0.1f;
                if (movKitZ > 95)
                {
                        recorrido1 = false;
                        recorrido2 = true;
                }
        }
        if (recorrido2)
        {
                rotKit = 90.0f;
                movKitX += 0.1f;
                if (movKitX > 110)
                {
                        recorrido2 = false;
                        recorrido3 = true;

                }
        }
```

```
if (recorrido3)
{
        rotKit = 180.0f;
        movKitZ -= 0.1f;
        if (movKitZ < -16)
        {
                recorrido3 = false;
                recorrido4 = true;
        }
}

if (recorrido4)
{
        rotKit = 270;
        movKitX -= 0.1f;
        if (movKitX < 0)
        {
                recorrido4 = false;
                recorrido5 = true;
        }
}
if (recorrido5)
{
        rotKit = 0;
        movKitZ += 0.1f;
        if (movKitZ > 0)
        {
                recorrido5 = false;
```

```
                recorrido1 = true;
            }
        }
    }
}


// Is called whenever a key is pressed/released via GLFW
void KeyCallback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    //presion de tecla para generar la visualizacion de animacion por KeyFrames
    if (keys[GLFW_KEY_L])
    {
        if (play == false && (FrameIndex > 1))
        {

            posX = 0;
            posY = 0;
            posZ = 0;
            rotizq = 0;
            rotder = 0;
            rotc = 0;
            resetElements();
            //First Interpolation
            interpolation();

            play = true;
```

```
                playIndex = 0;
                i_curr_steps = 0;
        }
        else
        {
                play = false;
        }

}

if (GLFW_KEY_ESCAPE == key && GLFW_PRESS == action)
{
        glfwSetWindowShouldClose(window, GL_TRUE);
}

if (key >= 0 && key < 1024)
{
        if (action == GLFW_PRESS)
        {
                keys[key] = true;
        }
        else if (action == GLFW_RELEASE)
        {
                keys[key] = false;
        }
}
//Cierre de programa
if (keys[GLFW_KEY_SPACE])
```

```
        {
                active = !active;
                if (active)
                        LightP1 = glm::vec3(1.0f, 0.0f, 0.0f);
                else
                        LightP1 = glm::vec3(0.0f, 0.0f, 0.0f);
        }
}

void MouseCallback(GLFWwindow* window, double xPos, double yPos)
{
        //Movimeintos para mouse
        if (firstMouse)
        {
                lastX = xPos;
                lastY = yPos;
                firstMouse = false;
        }

        GLfloat xOffset = xPos - lastX;
        GLfloat yOffset = lastY - yPos;  // Reversed since y-coordinates go from
bottom to left

        lastX = xPos;
        lastY = yPos;

        camera.ProcessMouseMovement(xOffset, yOffset);
}
```