UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA ELÉCTRICA

INGENIERÍA EN COMPUTACIÓN

LABORATORIO DE COMPUTACIÓN GRÁFICA e INTERACCIÓN HUMANO COMPUTADORA

MANUAL TÉCNICO DE PROYECTO

NOMBRES COMPLETOS:	**Nº de Cuenta:**
1.- Barrios López Francisco	1.- 317082555
2.- González Cuellar Arturo	2.- 317325281
3.- Galdamez Pozoz Yav Farid	3.-

GRUPO DE LABORATORIO: 02

GRUPO DE TEORÍA: 04

SEMESTRE 2023-2

FECHA DE ENTREGA LÍMITE: 4 de junio de 2023

CALIFICACIÓN: ________

Contenido:

Introduction

Computer graphics is a rapidly growing field that encompasses the processing and generation of real-time graphics. OpenGL is an API that provides the necessary tools for creating these 2D and 3D graphics with specific implementation techniques.
This document will showcase the technical implementation details carried out to include all elements within the scene, including model loading, traversal, texturing, animation, and lighting.
Model loading involves the process of importing three-dimensional objects created within a graphic environment. In this case, these models were created using tools such as 3dsMax and Blender. The models loaded in OpenGL allow for reading and rendering, drawing faces and vertices using shaders and appropriate transformation settings.
Camera traversal involves the control, position, and orientation of the camera within the environment, allowing the user to simulate movement and perspective in the scene.

Lighting is a technique used to create a realistic appearance in the generated environments. At this point, there are lighting techniques such as ambient, diffuse, and specular lighting to enhance surfaces and objects in the scene.
Animation enables interaction and movement of objects in the scene. To achieve this effect, interpolation, basic transformations, and continuous rendering updates are used.
This project aims to implement all the techniques in order to create an interactive and visually pleasing environment, providing an immersive and high-quality graphical experience.

The project is based on the following key components:

Traversal

Isometric camera

In the code, there are two files that implement the camera: *camera.h* and *camera.cpp*. Initially, in the main file, the projection is defined as follows:

```
glm::mat4 projection = glm::perspective(45.0f, (GLfloat)mainWindow.getBufferWidth() / mainWindow.getBufferHeight(), 0.1f, 1000.0f);
```

Camera.h

In camera.h, two functions are declared, which will be used to configure the objects and the camera for implementing the isometric camera:

```
	// Funciones para cámara isométrica
	glm::mat4 ConfIsometric(glm::mat4 model);
	bool getIsometric() { return isometric; }
	GLfloat getZoom() { return zoom; }
```

And in general, the variables to be used are:

```
	glm::vec3 iso_position = glm::vec3(-1.0f, 0.0f, 1.0f);// Posicion camara isometrica
	GLfloat iso_right = 0.0f;                             // Derecha
	GLfloat iso_up = 0.0f;                                // Arriba
	GLfloat zoom = 30.0f;
	bool isometric = false;                               // Activar / desactivar camara
```

The zoom is set to 30 because it is the maximum size of the entire scene in the X and Z axes. Initially, the isometric camera is not shown.

Camera.cpp

In camera.cpp code, the first function is:

```
glm::mat4 Camera::ConfIsometric(glm::mat4 model) {
	model = glm::rotate(model, glm::radians(45.0f), glm::vec3(1.0f, 0.0f, 0.0f));
	model = glm::rotate(model, glm::radians(35.2644f), glm::vec3(0.0f, 0.0f, 1.0f));
	return model;
}
```

This function is used by all objects drawn in the scene. What it does are two rotations at different angles, in order to simulate that the X, Y, and Z axes are 120º from each other. Then, when the isometric camera is active, all objects go through two transformations before their own transformations.

Within the `void Camera::keyControl(bool* keys, GLfloat deltaTime)` function, the handling of the WASD keys is performed differently depending on the camera. In the case of the isometric camera, these keys modify the camera's position. If the condition `if (isometric == false)` is not met, it means that the isometric camera is active, and the keys perform the following actions:

```
		if (keys[GLFW_KEY_W]) {
			iso_up += 1.0f;
			if (iso_up >= 20.0f) iso_up = 20.0f;
		}
		if (keys[GLFW_KEY_S]) {
			iso_up -= 1.0f;
			if (iso_up <= -20.0f) iso_up = -20.0f;
		}
		if (keys[GLFW_KEY_A]) {
			iso_right -= 1.0f;
			if (iso_right <= -20.0f) iso_right = -20.0f;
		}
		if (keys[GLFW_KEY_D]) {
			iso_right += 1.0f;
			if (iso_right >= 20.0f) iso_right = 20.0f;
		}
		iso_position = glm::vec3(iso_right, iso_up, iso_right);
```

In the same way, the up and down arrows zoom in or out of the camera:

```
		if (keys[GLFW_KEY_DOWN]) {
			zoom += 1.0f;
			if (zoom >= 30.0f) zoom = 40.0f;
		}
		if (keys[GLFW_KEY_UP]) {
			zoom -= 1.0f;
			if (zoom <= 5.0f) zoom = 5.0f;
		}
```

Within the same `function glm::mat4 Camera::calculateViewMatrix()` gets the value of lookAt. For when the isometric camera is active, the following line is sent:

```
glm::lookAt(iso_position, iso_position + glm::vec3(1.0f, 0.0f, -1.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

Main file:

Within the main cycle that draws the whole scene, it is done:

```
	if (camera.getIsometric() == false) {
		projection = glm::perspective(45.0f, (GLfloat)mainWindow.getBufferWidth() / mainWindow.getBufferHeight(), 0.1f, 1000.0f);
	}
	else {
		projection = glm::ortho(-camera.getZoom(), camera.getZoom(), -camera.getZoom(), camera.getZoom(), -30.0f, 30.f);
	}
```

In this way, when the isometric camera is active, an orthogonal projection is used which receives as parameters the Zoom, and 30 and -30 for Y, so that all the elements of the scene can be drawn without being covered by the scope of the camera.

As mentioned above, the `glm::mat4 Camera::ConfIsometric(glm::mat4 model)` Performs two additional rotations for each model, as long as the isometric camera is active. So for each assignment of the matrix identity of the matrix model, the following line is put:

```
model = glm::mat4(1.0);
if (camera.getIsometric()) model = camera.ConfIsometric(model);
```

Ilumination

DirectionalLight

For this type of lighting, it is considered that only one light should be defined as it represents natural light and allows us to modify the day-night cycle with intensity parameters:

```
mainLight = DirectionalLight(1.0f, 1.0f, 1.0f,
        1.0f, 0.3f,
        0.0f, 0.0f, -1.0f);

unsigned int pointLightCount = 0;
```

The first three parameters correspond to the color of the light. Since it is a natural light, it is defined with white color, meaning that all colors are represented. The intensity parameters determine the cone region and the lighting intensity. Finally, the direction of the light is defined, assuming it comes from the upper part and assigning it to the negative y-axis.

To modify the intensity of the natural light based on the day-night cycle, a counter, clock cycles per second obtained from the program, and a flag to change the cycle state are used. When the counter reaches a certain number of clock cycles, the state of the flag is checked. If it is in the day state, it changes to night and vice versa. Then, the counter is reset to start counting a new cycle.

Based on the state of the flag, the intensity of the light is adjusted. During the night cycle, the intensity is lower, while during the day cycle, the intensity increases.

```
if (ciclos >= CLOCKS_PER_SEC) {
                if (dia) dia = false;
                else dia = true;
                ciclos = 0;
        }
        else ciclos++;
        if (dia) mainLight.setIntensity(1.0f);
        else mainLight.setIntensity(0.4f);
```

PointLight

Lamp

For this type of lighting, we aim to implement point lights that simulate real lamp lights. These lights are distributed throughout the scene to create a realistic environment. Automatic switching on/off based on the day-night cycle is considered. We start by defining a vector of PointLight type with the maximum dimension corresponding to the total number of lights:

```
	PointLight pointLights[MAX_POINT_LIGHTS];
```

Within the main function, each PointLight is defined using the following structure:

```
PointLight(GLfloat red, GLfloat green, GLfloat blue,
	     GLfloat aIntensity, GLfloat dIntensity,
	     GLfloat xPos, GLfloat yPos, GLfloat zPos,
	     GLfloat con, GLfloat lin, GLfloat exp);
```

The first three parameters (red, green, blue) represent the RGB color of the light. The dIntensity parameter defines the cone region, and the aIntensity parameter determines the intensity decay as we move away from the center of the cone. The next three parameters determine the position in three-dimensional coordinates. Finally, the last three parameters represent the attenuation coefficients calculated based on a quadratic equation. Once the above is specified, the PointLights are defined:

```
	pointLights[1] = PointLight(1.0f, 0.0f, 0.0f,
		0.35f, 2.0f,
		-21.8f, 4.7f, -15.4f,
		0.3f, 0.2f, 0.1f);
	pointLightCount++;

	pointLights[2] = PointLight(1.0f, 0.0f, 0.0f,
		0.4f, 2.0f,
		21.8f, 4.7f, 15.4f,
		0.3f, 0.2f, 0.1f);
	pointLightCount++;

	pointLights[3] = PointLight(1.0f, 0.0f, 0.0f,
		0.4f, 2.0f,
		21.8f, 4.7f, 0.4f,
		0.3f, 0.2f, 0.1f);
	pointLightCount++;

	pointLights[4] = PointLight(1.0f, 0.0f, 0.0f,
		0.4f, 2.0f,
		21.8f, 4.7f, -15.4f,
		0.3f, 0.2f, 0.1f);
	pointLightCount++;
```

The lights were defined with a red color and the intensity was set the same for all of them. The position corresponds to the location of the lighthouse in the plane, and the attenuation coefficients are defined in the same way for each of them.

Light-Star

Additionally, a PointLight with yellow color is defined in the position where the star is located, with a greater intensity to cover the total area.

```
	pointLights[5] = PointLight(1.0f, 1.0f, 0.0f,
		1.5f, 2.0f,
		0.4f, 6.7f, 4.5f,
		0.3f, 0.2f, 0.1f);
	pointLightCount++;
```

So that the lights defined above are only switched on in the night day cycle, we send the list of the shader of the point lights that must be turned on when it is night, in case it is the day cycle then a parameter of 1 is sent to indicate that the information should not be loaded while that cycle is present:

```
	if (!dia) shaderList[0].SetPointLights(pointLights, pointLightCount);
		else shaderList[0].SetPointLights(pointLights, 1);
		shaderList[0].SetSpotLights(spotLights, spotLightCount);
```

light source

An illumination is defined where the color blue is given, with a region of 2.0; Additionally, the position in the plane where the source is located is indicated, finally the attenuation parameters calculated from the second degree equation are defined.

```
	pointLights[0] = PointLight(0.0f, 1.0f, 1.0f,
		0.35f, 2.0f,
		12.0f, 0.1f, 12.0f,
		0.3f, 0.2f, 0.1f);
	pointLightCount++;
```

SpotLight type

This type of lights, having a direction, it was decided to include two lights which are located in two corners of the stage, pointing towards the cabin.

Definition of lights

Start by defining the lights:

```
unsigned int spotLightCount = 0;
	////luces para la cabaña
	spotLights[0] = SpotLight(0.0f, 1.0f, 0.0f,
		5.0f, 5.0f,
		15.0f, 1.5f, -20.0f,
		-0.5f, 0.0f, 0.5f,
		1.0f, 0.1f, 0.01f,
		20.0f);
	spotLightsAux[0] = spotLights[0];
	spotLightCount++;

	spotLights[1] = SpotLight(0.0f, 0.0f, 1.0f,
		5.0f, 5.0f,
		-15.0f, 1.5f, -20.0f,
		0.5f, 0.0f, 0.5f,
		1.0f, 0.1f, 0.01f,
            20.0f);
        spotLightsAux[1] = spotLights[1];
        spotLightCount++;
```

These lights have a value of 5 for both the ambient and diffuse components, and in the values of the second degree equation these values were chosen since they allow a greater appreciation of the light.

Control of lights

In order to allow the user to control the lights on the stage, variables and flags were added which indicate when a certain light is active or not.

Window.h

The variables are defined in this file:

```
        bool faro1 = false;
        bool faro2 = false;
```

Which are initialized to zero. This causes the lights not to be drawn when the program is executed. Within the same file, the get:

```
        bool getFaro1() { return faro1; }
        bool getFaro2() { return faro2; }
```

Window.cpp

To achieve keyboard control, we chose to use the numbers 1, 2, 3 and 4, where the first 2 manipulate the first light, and the following 2 manipulate the second light:

```
        if (key == GLFW_KEY_1) theWindow->faro1 = false;
        if (key == GLFW_KEY_2) theWindow->faro1 = true;
        if (key == GLFW_KEY_3) theWindow->faro2 = false;
        if (key == GLFW_KEY_4) theWindow->faro2 = true;
```

Archivo main

Dentro de este archivo, se inicia declarando un arreglo de Spot Lights auxiliar, además de un arreglo de bools que indican, por su índice, cual luz está activa y cual no:

```
SpotLight spotLightsAux[MAX_SPOT_LIGHTS];
bool spotActivo[2] = {false, false};
```

Within this file, we start by declaring an array of auxiliary Spot Lights, as well as an array of bools that indicate, by their index, which light is active and which is not:

```
// ver que luces están encendidas
                if (mainWindow.getFaro1() == true) spotActivo[0] = true;
                else spotActivo[0] = false;
                if (mainWindow.getFaro2() == true) spotActivo[1] = true;
                else spotActivo[1] = false;
```

Finalmente, se analiza cada valor del arreglo de bools, y cuando el índice actual sea verdadero, se le manda la definición del arreglo principal al arreglo de SpotLights auxiliar. Con esto, al finalizar, solo se le manda el arreglo auxiliar y la cuenta de las luces activas al Shader.

```
                indice = 0;
```

```
            for (int i = 0; i < spotLightCount; i++) {
                    if (spotActivo[i] == true) {
                            spotLightsAux[indice] = spotLights[i];
                            indice++;
                    }
            }
            shaderList[0].SetSpotLights(spotLightsAux, indice);
```

Animación

For the animations part, they are divided into 3 parts:

Simple animations

Eagle – Chest

For this animation we want the eagle to be inside the trunk that is inside the cabin. When the user presses the B key, the trunk opens and the eagle flies out of the trunk, turns around, returns to the trunk, turns around again to recover its original position, and finally closes the trunk. The variables used for this are:

```
// aguila - baul
float rotaAlasAguila, movAguilaZ, movAguilaY;
float rotaTapaBaul, rotaAguila;
bool animAguila, openTapa, regresaAguila, arribaAlas;
```

window.h:
A flag is declared to indicate when to start the animation:

```
bool abrirBaul = false;
```

In addition, two methods are declared, one will serve to obtain the value of the flag, and the other to false reset it when the animation is finished:

```
    bool getabrirBaul() { return abrirBaul; }
    void setabriBaul(bool abrir) { abrirBaul = abrir; }
```

window.cpp:
In this file only the flag is set to true when pressing the B key:

```
if (key == GLFW_KEY_B) theWindow->abrirBaul = true;
```

Archivo main:
A function called void AnimateEagle() was created where it starts by asking if the flag is set to true. When the flag is set to true, the animation starts by opening the trunk lid, which uses a flag initialized to true.

```
            if (openTapa) {
                    if (rotaTapaBaul >= 90.0f) {
                            rotaTapaBaul = 90.0f;
```

```
                animAguila = true;
            }
            else rotaTapaBaul += 3.0f * deltaTime;
        }
```

When the trunk is fully opened, the animEagle flag becomes true. At the moment this flag becomes true, another flag helps to know if the eagle is just about to leave the trunk or if it is already on its way back. Since returnEagle is initialized to false, the eagle leaves the trunk. At that moment it starts to move its wings. In order not to use functions and not to go back to the complex animation, comparisons are used to move the wings up and down.

```
            if (!regresaAguila) {

                if (arribaAlas) {
                    if (rotaAlasAguila >= 30.0f) {
                        rotaAlasAguila = 30.0f;
                        arribaAlas = false;
                    }
                    else rotaAlasAguila += 10.0f * deltaTime;
                }
                else {
                    if (rotaAlasAguila <= -30.0f) {
                        rotaAlasAguila = -30.0f;
                        arribaAlas = true;
                    }
                    else rotaAlasAguila -= 10.0f * deltaTime;
                }

                if (movAguilaZ >= 20.0f) movAguilaZ = 20.0f;
                else movAguilaZ += 0.2f * deltaTime;
                if (movAguilaY >= 10.0f) movAguilaY = 10.0f;
                else movAguilaY += 0.1f * deltaTime;

                if (movAguilaZ == 20.0f && movAguilaY == 10.0f) {
                    if (rotaAguila >= 180.0f) {
                        rotaAguila = 180.0f;
                        regresaAguila = true;
                    }
                    else rotaAguila += 2.0f * deltaTime;
                }
            }
```

At the same time, you increase the eagle's position in Z to 20, and Y to 10. When the maximum point is reached, the eagle rotates in the air while continuing to move the wings. Once it completes the rotation, the flag returnsEagle turns true, and the eagle returns.

```
            if (regresaAguila) {
                if (movAguilaZ <= 0.0f) movAguilaZ = 0.0f;
                else movAguilaZ -= 0.3f * deltaTime;
                if (movAguilaY <= 0.0f) movAguilaY = 0.0f;
                else movAguilaY -= 0.15f * deltaTime;

                if (movAguilaZ == 0.0f && movAguilaY == 0.0f) {
                    if (rotaAguila <= 0.0f) {
```

```
                                        rotaAguila = 0.0f;
                                        regresaAguila = false;
                                        openTapa = false;
                                        animAguila = false;
                                        rotaAlasAguila = 0.0f;
                                }
                                else {
                                        rotaAguila -= 2.0f * deltaTime;
                                        if (arribaAlas) {
                                                if (rotaAlasAguila >= 30.0f) {
                                                        rotaAlasAguila = 30.0f;
                                                        arribaAlas = false;
                                                }
                                                else rotaAlasAguila += 10.0f * deltaTime;
                                        }
                                        else {
                                                if (rotaAlasAguila <= -30.0f) {
                                                        rotaAlasAguila = -30.0f;
                                                        arribaAlas = true;
                                                }
                                                else rotaAlasAguila -= 10.0f * deltaTime;
                                        }
                                }
                        }
                }
```

The return of the eagle is the same as the outgoing, with the difference that it no longer moves its wings, as it is dropped. When the eagle returns to its original position, it rotates again as it moves its wings again, and when it reaches its original position, the eagle's flags return to their initial values, and openTapa becomes false.

```
                if (!openTapa) {
                        if (rotaTapaBaul <= 0.0f) {
                                rotaTapaBaul = 0.0f;
                                openTapa = true;
                                mainWindow.setabriBaul(false);
                        }
                        else rotaTapaBaul -= 3.0f * deltaTime;
                }
```

Finally, the lid is closed again, and the keyboard-activated flag is set to false, so that the animation can be rerun by pressing the B key.

Animated ship:

In this animation a tour of the ship is made from an initial position, when you press the key 1 that corresponds to the ignition of a spotLight type light, the ship begins the journey to where there is a cow illuminated by this light, once it reaches it descends and lifts it to a certain height, then lowers it again and returns to its initial position:

```
        if (mainWindow.anim() == 0) {
                girarnave -= 0.5 * deltaTime;
        }
```

```
            if (mainWindow.anim() == 1) {
                girarnave += 0.5 * deltaTime;
                if (bande == 1) {
                    if (movnave_z > -15) {
                        movnave_x += 0.065f * deltaTime;
                        movnave_z -= 0.1f * deltaTime;
                    }
                    if (movnave_z < -15) {
                        if (movnave_y > 5) {
                            movnave_y -= 0.03f * deltaTime;
                        }
                        if (movnave_y < 5) {
                            bande = 3;
                        }
                    }
                }
```

When number key 2 is pressed to turn off the light, then the animation will stop and the state in which it was left will be saved, however the ship will continue to rotate to give that effect of realism to the animation. When the 1 key is pressed again, the animation will continue with its normal cycle until it finishes and restarts while it is turned on.

```
if (bande == 3) {
                if (movnave_y < 12) {
                    movnave_y += 0.03f * deltaTime;
                    movvaca += 0.03f * deltaTime;
                }
                if (movnave_y > 12) {
                    bande = 4;
                }
            }
            if (bande == 4) {
                if (movnave_y > 4.8) {
                    movnave_y -= 0.03f * deltaTime;
                    movvaca -= 0.03f * deltaTime;

                }
                if (movnave_y < 4.8) {
                    bande = 5;

                }

            }

            if (bande == 5) {
                if (movnave_y < 19) {
                    movnave_y += 0.03f * deltaTime;
                }
                if (movnave_y > 19) {
                    if (movnave_z < 0) {
                        movnave_x -= 0.065f * deltaTime;
```

```
                                        //movnave_y -= 0.01f * deltaTime;
                                        movnave_z += 0.1f * deltaTime;
                                }

                                if (movnave_z > 0) {
                                        bande = 1;
                                }
                        }
                }
        }

}
```

Mill propeller:

In this animation the propeller of the mill is rotated at different speeds. For this we use a variable that depends on a random number. In total there are 5 speeds at which the propeller can rotate, depending on the value of the variable. The variables used are:

```
//molino
float mov_mol, mov_molof;
int velocidad_mol;
```

And their initial values are:

```
        mov_mol = 0.0f; mov_molof = 0.2f;
        velocidad_mol = (rand() % 5) + 1;
```

Each time a revolution of the helix is completed, a new random number is generated that defines the new value of the Offset of the rotation, where it takes values from 0.2 to 1:

```
        if (velocidad_mol == 1) mov_molof = 0.2;
        else if (velocidad_mol == 2) mov_molof = 0.4;
        else if (velocidad_mol == 3) mov_molof = 0.6;
        else if (velocidad_mol == 4) mov_molof = 0.8;
        else mov_molof = 1.0f;
        if (mov_mol >= 360.0f) {
                mov_mol = 0.0f;
                velocidad_mol = (rand() % 5) + 1;
        }
        mov_mol += mov_molof * deltaTime;
```

Animaciones complejas

Para estas animaciones se realizaron funciones para cada animación, las cuales son llamadas en cada ciclo del reloj, y la cuales modifican las variables y banderas involucradas en cada animación.

Caballo

La idea de esta animación es que el caballo mueva aleatoriamente cada pata, como una especie de baile. De la misma forma, mueve la cabeza en ocasiones repetidas y la cola se balancea de un lado a otro.

Las variables utilizadas para lograr esto, declaradas en el archivo main, son:

```
// caballo
float rotCola, rotColaPunta;
float rotPataFLU, rotPataFLD;
bool plusPataFLU, plusPataFLD;
float rotPataFRU, rotPataFRD;
bool plusPataFRU, plusPataFRD;
float rotPataBLU, rotPataBLD;
bool plusPataBLU, plusPataBLD;
float rotPataBRU, rotPataBRD;
bool plusPataBRU, plusPataBRD;
float rotaCabeza, lossTimeC;
int accionC, movCabezaC;
bool condPataFR[2] = { false, false }, condPataFL[2] = { false, false };
bool condPataBR[2] = { false, false }, condPataBL[2] = { false, false };
```

El modelo del caballo está compuesto por 4 patas las cuales se dividen en dos, así como dos partes de la cola, la cabeza y el cuerpo del caballo.
Las primeras dos variables mueven la cola, las siguientes 16 variables se encargan de mover las 4 patas del caballo. Para este caso, cada pata cuenta con dos partes, así que, si el caballo alza una pata, la parte superior de la pata rotará a un sentido, mientras que la parte inferior rotará al sentido contrario. Como la pata se alza y regresa a su posición original, se usan las variables de plus, una para cada parte de cada pata. Todas las variables de las patas inician en 0 y los plus de la parte de arriba comienzan en true, mientras que el resto está en false:

```
	rotCola = 0.0f;
	rotColaPunta = 0.0f;
	rotPataFLU = 0.0f; rotPataFLD = 0.0f;
	plusPataFLU = true; plusPataFLD = false;
```

Las variables `rotaCabeza` y `lossTimeC` sirven para controlar el movimiento de la cabeza, ya que se buscaba que el caballo no estuviera moviendo la cabeza todo el tiempo. De acuerdo con un número aleatorio, se escoge si el caballo mueve la cabeza o no. `lossTimeC` sirve para que el caballo no mueva su cabeza durante un tiempo igual al que tardaría moviéndola.

```
movCabezaC = (rand() % 4) + 1;
```

Se escogió un numero aleatorio de 0 a 4 con el fin de ampliar un poco la cantidad de opciones en las que pueda caer el número, y no se tenga la ilusión de que siempre está realizando la misma acción. Entonces, cuando el valor del número aleatorio es 1 o 2 se mueve la cabeza, en caso contrario no lo hace.

```
if (movCabezaC <= 2) {
		if (rotaCabeza >= 180.0f) {
			rotaCabeza = -180.0f;
			movCabezaC = (rand() % 4) + 1;
		}
		else rotaCabeza += 3.0f * deltaTime;
	}
	else {
		if (lossTimeC >= 180.0f) {
			lossTimeC = -180.0f;
			movCabezaC = (rand() % 4) + 1;
		}
		else lossTimeC += 4.0f * deltaTime;
	}
```

La variable `accionC` también obtiene un valor de un número aleatorio, y sirve para indicar cual pata se moverá. Las variables `condPataFR[2]` y demás, sirven como condicionales para saber si la pata ha regresado completamente a su posición original. El siguiente fragmento de código prácticamente se repite 4 veces pero modifica variables distintas dependiendo de la pata.

```
	if (accionC == 1) {
		// Pata delantera izquierda
		if (plusPataFLU) {
			rotPataFLU -= 4.0f * deltaTime;
			if (rotPataFLU <= -180.0f) plusPataFLU = false;
		}
		else {
			rotPataFLU += 4.0f * deltaTime;
			if (rotPataFLU >= 0.0f) {
				plusPataFLU = true;
				rotPataFLU = 0.0f;
				condPataFL[0] = true;
			}
		}
		if (!plusPataFLD) {
			rotPataFLD += 4.0f * deltaTime;
			if (rotPataFLD >= 180.0f) plusPataFLD = true;
		}
		else {
			rotPataFLD -= 4.0f * deltaTime;
			if (rotPataFLD <= 0.0f) {
				plusPataFLD = false;
				rotPataFLD = 0.0f;
				condPataFL[1] = true;
			}
		}
		if (condPataFL[0] && condPataFL[1]) {
			accionC = rand() % 5 + 1;
			condPataFL[0] = false;
			condPataFL[1] = false;
		}
	}
```

Finalmente, al mandar estas variables a los modelos, se les manda dentro de una función sin, con el fin de suavizar un poco los movimientos. Por ejemplo, para la cabeza:

```
model = glm::rotate(model, glm::sin(glm::radians(rotaCabeza)) / 8, glm::vec3(1.0f, 0.0f, 0.0f));
```

Se le manda dentro de un sin dividido entre 8, ya que como al usar seno y la variable ir desde -180 a 180, se sabe que $\sin(180^{\circ}) = \text{sen}(-180^{\circ}) = \sin\,(0^{\circ})$, así que $\sin(90^{\circ})$ será la máxima rotación de la cabeza, así que al dividirla entre 8, se llega a que la cabeza rota 11.25º positivos y 11.25º negativos.

En las patas y la cola sucede lo mismo, aunque por ejemplo, para la pata frontal de la izquierda (la parte superior), se multiplica el seno por PI y luego se divide entre 3.

```
model = glm::rotate(model, PI * sin(glm::radians(rotPataFLU)) / 3, glm::vec3(1.0f, 0.0f, 0.0f));
```

Vaca

Para la animación de la vaca, esta debe de dar vueltas en círculos mientras mueve sus patas y su cola. Comenzando por las variables utilizadas:

```
// vaca
float rotaCola1ZV, rotaCola2ZV, rotaCola3ZV;
float rotaCola1XV, rotaCola2XV, rotaCola3XV;
int accionColaV;
float rotarVaca;
float rotPataVacaFR, rotPataVacaFL, rotPataVacaBR, rotPataVacaBL;
float movVacaX, movVacaZ;
```

El modelo de la vaca se compone de varias partes: la cabeza, el cuerpo, las 4 patas y la cola dividida en 3 segmentos. Las primeras 6 variables sirven para rotar la cola tanto en X como en Z. El entero `accionColaV` se basa en un numero aleatorio, y sirve para hacer que la cola se mueva de un lado a otro, o que la vaca levante la cola. `rotarVaca` hace que la vaca camine y también para que rote en el sentido en el que va caminando. Después se tienen las variables para cada pata, y finalmente las variables que determinan la posición de la vaca dependiendo de la rotación de esta.

Todas las variables son inicializadas en 0, excepto por la variable que define como se moverá la cola:

```
accionColaV = (rand() % 6) + 1;
```

Para cuando el valor de la variable vale 4 o menos, entonces la cola se moverá de un lado a otro. Para esto:

```
	if (accionColaV <= 4) {
		if (rotaCola2ZV >= 180.0f) rotaCola2ZV = -180.0f;
		else rotaCola2ZV += 3.0f * deltaTime;
		if (rotaCola3ZV >= 180.0f) rotaCola3ZV = -180.0f;
		else rotaCola3ZV += 3.0f * deltaTime;
		if (rotaCola1ZV >= 180.0f) rotaCola1ZV = -180.0f;
		else if (rotaCola1ZV >= -1.0f && rotaCola1ZV < 0.0f){
			rotaCola1ZV = 0.0f;
			rotaCola2ZV = 0.0f;
			rotaCola3ZV = 0.0f;
			accionColaV = (rand() % 6) + 1;
		}
		else rotaCola1ZV += 3.0f * deltaTime;
	}
```

Cada parte de la cola tiene una variable asociada la cual es la que varía. En este caso, solo se rota en Z, y cuando la cola regresa a su posición original, se vuelve a obtener un numero aleatorio.
Para cuando el número es 5 o 6, entonces la vaca alzará la cola. Se escogieron estos valores para que la vaca levante la cola de manera ocasional y no se vea tan continuo.

```
	else {		// alzar la cola
		if (rotaCola1ZV >= 180.0f) rotaCola1ZV = -180.0f;
		else rotaCola1ZV += 1.5f * deltaTime;
		if (rotaCola2ZV >= 180.0f) rotaCola2ZV = -180.0f;
		else rotaCola2ZV += 1.5f * deltaTime;
		if (rotaCola3ZV >= 180.0f) rotaCola3ZV = -180.0f;
		else rotaCola3ZV += 1.5f * deltaTime;

		if (rotaCola2XV >= 180.0f) rotaCola2XV = 0.0f;
		else rotaCola2XV += 3.0f * deltaTime;
```

```
            if (rotaCola3XV >= 180.0f) rotaCola3XV = 0.0f;
            else rotaCola3XV += 3.0f * deltaTime;
            if (rotaCola1XV >= 180.0f) {
                  rotaCola1XV = 0.0f;
                  rotaCola2XV = 0.0f;
                  rotaCola3XV = 0.0f;
                  accionColaV = (rand() % 6) + 1;
            }
            else rotaCola1XV += 3.0f * deltaTime;
      }
```

Ahora la cola rota tanto en X como en Z, pero la que decide cuando generar el siguiente numero aleatorio es la rotación de la primera parte de la cola en X. En realidad, puede ser cualquiera, debido a que las 3 variables en X varían a la misma velocidad. Cuando la cola recupera su posición original, se vuelve a generar el numero aleatorio.

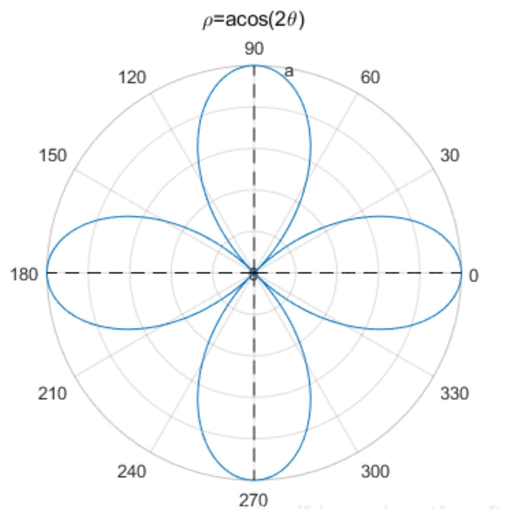
Finalmente, las patas se mueven continuamente, y de acuerdo con el valor de la rotación de la vaca, se modifica la posición tanto en X como en Z, de acuerdo con la ecuación del circulo en coordenadas polares pasadas a cartesianas:

```
if (rotPataVacaFL >= 180.0f) rotPataVacaFL = -180.0f;
      else rotPataVacaFL += 3.0f * deltaTime;
      if (rotPataVacaBR >= 180.0f) rotPataVacaBR = -180.0f;
      else rotPataVacaBR += 3.0f * deltaTime;
      if (rotPataVacaBL <= -180.0f) rotPataVacaBL = 180.0f;
      else rotPataVacaBL -= 3.0f * deltaTime;
      if (rotPataVacaFR <= -180.0f) rotPataVacaFR = 180.0f;
      else rotPataVacaFR -= 3.0f * deltaTime;
      rotarVaca += 0.15f * deltaTime;
      movVacaX = 5 * cos(glm::radians(rotarVaca));
      movVacaZ = 5 * sin(glm::radians(rotarVaca));
```

De acuerdo con el mismo principio del caballo, cada variable es pasada a los modelos dentro de un seno y dividido por algún valor, con el fin de que se suavicen los movimientos y no se vea tan repentino el cambio entre aumento y disminución de las variables.

Mariposa

La idea de la animación de la mariposa es que este volando en un ciclo que no parece tener definición, ya que como se sabe, en la vida real las mariposas vuelan dando pequeñas vueltas o cambiando su dirección aleatoriamente. Esto es difícil de programar, así que se trató de obtener algo parecido con ayuda de la siguiente función:



Las variables utilizadas para lograrlo son:

```
// mariposa
float rotaAlasButterfly, movMariposaX, movMariposaZ, movMariposaY;
float rotaMariposaY, anguloMariposa;
```

En primer lugar, las alas de la mariposa siempre van a estar moviéndose.

```
	if (rotaAlasButterfly >= 180.0f) rotaAlasButterfly = -180.0f;
	else rotaAlasButterfly += 10.0f * deltaTime;
```

Esto al mandarse a las alas de la mariposa, se sigue el mismo principio que el de las patas del caballo: mandarlo en una función sin y dividirlo entre un valor:

```
model = glm::rotate(model, PI * sin(glm::radians(rotaAlasButterfly)) / 4, glm::vec3(0.0f, 0.0f, 
1.0f));
```

Para el recorrido de la mariposa, se cuenta con una ecuación polar de la forma $r = \alpha\sin{(2\theta)}$, en donde el ángulo toma valores $0 < \theta < 2\pi$, así que para pasarlo a forma cartesiana:

$$x = \alpha\sin(2\theta) * \cos{(\theta)}$$
$$z = \alpha\sin(2\theta) * \sin{(\theta)}$$

Finalmente, en el eje Y solo se multiplican ambos valores, con el fin de que el vuelo de la mariposa no sea totalmente plano:

```
//3sin(2t) 0.0 < t < 2pi
if (anguloMariposa >= 180.0f) anguloMariposa = -180.0f;
else anguloMariposa += 0.5f * deltaTime;

movMariposaX = 8 * glm::sin(glm::radians(2 * anguloMariposa)) * 
glm::cos(glm::radians(anguloMariposa));
movMariposaZ = 8 * glm::sin(glm::radians(2 * anguloMariposa)) * 
glm::sin(glm::radians(anguloMariposa));

movMariposaY = (movMariposaX * movMariposaZ) / 4;
```

Animación por KeyFrames

En esta animación se decidió animar una estrella aparte de la estrella con la que ya se contaba. Esta segunda estrella es más pequeña, y realiza un trayecto alrededor del escenario mientras da vueltas sobre sí misma. Las variables usadas son:

```
// estrella - keyframes
float reproduciranimacion, habilitaranimacion;
std::string content;
```

De esta forma, también se declaran variables que servirán para indicar la posición de la estrella, y del movimiento que debe de tener. Se definen como máximo 30 frames, con 90 interpolaciones, y en total se tienen 15 estados

```
// variables afectadas por keyframes
//NEW// Keyframes
float posXStar = 0.0, posYStar = 0.0, posZStar = 0.0;     // translate - Estado Inicial
float movStar_x = 0.0f, movStar_y = 0.0f;                             //dezplazamiento en X
o Y
float giroStar = 0;                                                         //rotate

#define MAX_FRAMES 30           //Numero máximo de frames
int i_max_steps = 90;           //cuantas interpolaciones se van a obtener - fluidez de la
animacion
int i_curr_steps = 15;
```

Además, se tiene una estructura de datos que tiene como variables los movimientos de la estrella y su giro, y sus respectivos incrementos.

```
typedef struct _frame
{
	//Variables para GUARDAR Key Frames
	float movStar_x;         //Variable para PosicionX
	float movStar_y;         //Variable para PosicionY
	float movStar_xInc;      //Variable para IncrementoX
	float movStar_yInc;      //Variable para IncrementoY
	float giroStar;
	float giroStarInc;
}FRAME;
```

Finalmente, se declaran los frames:

```
FRAME KeyFrame[MAX_FRAMES];
int FrameIndex = 15;                  //introducir datos
bool play = false;
int playIndex = 0;
```

The frames are stored in an external file called KeyFrames.txt, which has 15 frames stored in it. In order to read them, the following function was generated:

```
void readFile() {
	int i = 0, linead = 0;
	std::string indice, valor;
	int index;
	float value;
	std::ifstream fileStream("KeyFrames.txt", std::ios::in);
	if (!fileStream.is_open()) {
		printf("Failed to read %s! File doesn't exist.", "KeyFrames.txt");
		content = "";
	}

	std::string line = "";
	while (!fileStream.eof()) {
		std::getline(fileStream, line);

		if (linead < 10) indice = line.substr(9, 1);
		else indice = line.substr(9, 2);
		index = std::stoi(indice);
		switch (i) {
		case 0:
			if (linead < 10) valor = line.substr(24, line.size() - 26);
			else valor = line.substr(25, line.size() - 27);
			value = std::stof(valor);
			KeyFrame[index].movStar_x = value;
			std::cout << "KeyFrame[" << index << "].movStar_x = " << value << ";" << 
std::endl;
			i++;
			break;
		case 1:
			if (linead < 10) valor = line.substr(24, line.size() - 26);
			else valor = line.substr(25, line.size() - 27);
			value = std::stof(valor);
			KeyFrame[index].movStar_y = value;
			std::cout << "KeyFrame[" << index << "].movStar_y = " << value << ";" << 
std::endl;
			i++;
			break;
		case 2:
			if (linead < 10) valor = line.substr(23, line.size() - 25);
			else valor = line.substr(24, line.size() - 26);
			value = std::stof(valor);
			KeyFrame[index].giroStar = value;
			std::cout << "KeyFrame[" << index << "].giroStar = " << value << ";\n" << 
std::endl;
			i = 0;
			linead++;
			break;
		}
	}
	fileStream.close();
}
```

We start by reading the file. As long as the reading of the file is successful, it is analyzed line by line. According to the form in which the data is stored:

```
KeyFrame[0].movStar_x = 0.0f;
KeyFrame[0].movStar_z = 0.0f;
KeyFrame[0].giroStar = 0.0f;
```

A pair of substrings is obtained, one for the index, and one for the value itself. The Switch structure is used to know which data is being read: if i = 0, the movement in X is read, if it is 1, the position in Y is obtained,

and if it is 2, the turn is obtained. When the turn is read, i is reset to 0 since the next value to be read is a position in X.
The variable linead helps to see up to what index the substrings are obtained, since for example, yes, index 10 is reached:

```
KeyFrame[10].movStar_x = -10.0f;
KeyFrame[10].movStar_z = -5.0f;
KeyFrame[10].giroStar = 180.0f;
```

The substring retrieval is no longer the same, but now takes two characters for the index, and one more right shifted character to get the value.
The rest of the functions provided are unchanged, except that now there is no need to save frames, so these functions are not present.

The rendering of the models was based on the effort of the knowledge learned throughout the course knowing the development applications for 3D modeling as 3DsMax and Blender, in them the models that would be part of the scenario to recreate as well as the texture of them were designed; However, it must be recognized that some models were obtained from third parties that dedicate time to recreate real life objects, with this, and with the rights obtained with models for free use, they were downloaded and optimized in order to recreate and make use of them in the assembly of the complete diorama.

Next, reference is made to the pages where the downloaded models were obtained:

Descarga gratuita de modelos 3D. (n.d.). Open3dmodel.com; Open3dModel. Retrieved May 28, 2023, from https://open3dmodel.com/es/

Modelos 3D gratis - Free3d.com. (n.d.). Free3d.com. Retrieved May 28, 2023, from https://free3d.com/es/

Modelos en 3D para profesionales. (n.d.). Turbosquid.com. Retrieved May 28, 2023, from https://www.turbosquid.com/es/

The models resource. (n.d.). Models-resource.com. Retrieved May 28, 2023, from https://www.models-resource.com/

Así mismo se hace referencia a los modelos que se descargaron para el armado del diorama en OpenGL:

GianaSistersFan64 (sf) Árbol del pantano. Hermanas De Giana: Sueños retorcidos. Disponible en The Models Source
Modelo original: BrittanyOfKoppai (sf) Caballo. La Leyenda de Zelda: Ocarina of Time 3D. Disponible en The Models Resource
Modelo original: Garamonde (sf) Mariposa Amarilla. Cruce de animales: gente de la ciudad. Disponible en The Models Resource

Modelo original: KleinStudio (sf) vaca. La Leyenda de Zelda: Majora's Mask 3D. Disponible en The Models Resource

Modelo original: IndigoPupper (sf) Sección Harry Potter y el Prisionero de Azkaban. Disponible en The Models Resource

Cow frame test - Download Free 3D model by Nyilonelycompany. (2021, August 13).

modelo 3d Águila gratis - TurboSquid 1045001. (2016, June 10). Turbosquid.com. https://www.turbosquid.com/es/3d-models/eagle-rigged-fbx-free/1045001

modelo 3d Caballo (1) gratis - TurboSquid 810753. (2014, March 30). Turbosquid.com. https://www.turbosquid.com/es/3d-models/free-low--horse-3d-model/810753

Modelo 3d sin molino de viento - .3ds - Open3dModel. (2014, December 27). Open3dmodel.com. https://open3dmodel.com/es/3d-models/windmill-free-3d-model_12296.html

modelo 3d Small Arch - Moss 1 gratis - TurboSquid 1987223. (2022, November 14). Turbosquid.com. https://www.turbosquid.com/es/3d-models/small-arch-moss-1-3d-model-1987223

printable_models, S. by. (n.d.). *Cat v1 Free 3D Model - .obj .stl - Free3D*. Free3d.com. Retrieved May 28, 2023, from https://free3d.com/3d-model/cat-v1--522281.html