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: 28 de mayo de 2023

CALIFICACIÓN: ________

Contents:

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 - well

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.
Start by defining the lights:

Definition of lights

The lights are defined as follows:

```
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.

Light control

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

In this file the variables are defined as:

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

Which are initialized as zero. This means that the lights will not be rendered when running the program. Additionally, the "get" functions were declared within the same file.:

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

Window.cpp

To achieve keyboard control, the numbers 1, 2, 3, and 4 were chosen, where the first two manipulate the first light, and the next two 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;
```

Main file

Within the main file, an auxiliary array of Spot Lights is initialized, along with a boolean array that indicates, by index, which light is active and which is not.:

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

To determine which lights will be modified, the previously defined methods are used, and based on the value returned by these methods, the boolean array is modified:

```
                // 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;
```

Finally, each value of the boolean array is analyzed, and when the current index is true, the definition of the main array is passed to the auxiliary Spot Lights array. By doing this, at the end, only the auxiliary array and the count of active lights are passed to the shader.

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

```
	}
	shaderList[0].SetSpotLights(spotLightsAux, indice);
```

Animation

For the animations, these are divided in 3:

Simple animations

Eagle – Chest

To achieve this animation, the initial position of the eagle is inside the chest located inside the cabin. When the user presses the B key, the chest opens, and the eagle flies out. The eagle then rotates, returns to the chest, rotates again to its original position, and finally, the chest closes. The variables used for this are:

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

window.h:

A flag variable is declared to indicate the moment when the animation should start:

```
bool abrirBaul = false;
```

Additionally, two methods are declared. One will be used to get the value of the flag, and the other to reset it to false when the animation has finished:

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

window.cpp:

In this file only the flag is set to true when the B key is pressed:

En este archivo únicamente se coloca en true la bandera al momento de presionar la tecla B:

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

Main file:

A function called `void AnimateEagle()` was created where it starts by checking if the flag is true. When the flag becomes true, the animation begins by opening the chest lid, using a flag that is initialized as true.

```
		if (openTapa) {
			if (rotaTapaBaul >= 90.0f) {
				rotaTapaBaul = 90.0f;
				animAguila = true;
			}
			else rotaTapaBaul += 3.0f * deltaTime;
		}
```

When the chest is fully opened, the `animAguila` flag becomes true. Once this flag becomes true, another flag is used to determine if the eagle is just about to leave the chest or if it is already returning. Since `regresaAguila` is initialized as false, the eagle flies out of the chest. At that moment, the eagle starts flapping its wings. In order to avoid using functions and to keep the animation simple, 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, the eagle's position is incremented in the Z direction up to 20 and in the Y direction up to 10. When it reaches the maximum point, the eagle rotates in the air while continuing to flap its wings. Once it completes the rotation, the `regresaAguila` flag becomes 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 similar to the departure, with the difference that it no longer flaps its wings as it descends. When the eagle returns to its original position, it rotates again while flapping its wings, and upon reaching 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 chest lid is closed again, and the flag that is activated by keyboard input is set to false, allowing the animation to be executed again by pressing the B key.

Mill´s propeller:

In this animation, the mill's propeller is rotated at different speeds. This is achieved using a variable that depends on a random number. There are a total of 5 speeds at which the propeller can rotate, depending on the value of the variable. The variables to used are:
```
//molino
float mov_mol, mov_molof;
int velocidad_mol;
```

Y sus valores iniciales son:
```
    mov_mol = 0.0f; mov_molof = 0.2f;
    velocidad_mol = (rand() % 5) + 1;
```

Every time the propeller completes a full rotation, a new random number is generated to define the new value of the rotation offset. This offset value can range 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;
```

Complex animations

For these animations, functions were created for each animation, which are called in each clock cycle, and they modify the variables and flags involved in each animation.

Horse

The idea of this animation is to randomly move each leg of the horse, like a sort of dance. Similarly, it moves the head in repeated motions, and the tail sways from side to side.
The variables used to achieve this, declared in the main file, are:

```
// 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 };
```

The horse model consists of 4 legs, which are divided into two parts, as well as two parts of the tail, the head, and the body of the horse.
The first two variables are responsible for moving the tail, while the next 16 variables are used to move the 4 legs of the horse. In this case, each leg has two parts, so if the horse lifts one leg, the upper part of the leg rotates in one direction, while the lower part rotates in the opposite direction. As the leg lifts and returns to its original position, the "plus" variables are used, one for each part of each leg. All the variables for the legs start at 0, and the "plus" variables for the upper part start as true, while the rest are set to false:

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

The variables "rotaCabeza" and "lossTimeC" are used to control the movement of the head, as the intention is to avoid the horse constantly moving its head. Based on a random number, it is determined whether the horse will move its head or not. "lossTimeC" is used to prevent the horse from moving its head for a duration equal to the time it would take to move it.

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

The random number is chosen from the range 0 to 4 in order to provide a wider range of options for the horse's actions and avoid the impression of it always performing the same action. When the random number is 1 or 2, the horse moves its head; otherwise, it does not. This introduces some variability in the head movement of the horse.

```
	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;
        }
```

The variable "accionC" also obtains a random number value and serves to indicate which leg will move. The variables "condPataFR[2]" and so on serve as conditionals to determine if the leg has returned to its original position completely. The following code snippet is repeated four times, but it modifies different variables depending on the leg.

```
        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;
            }
        }
```

Finally, when passing these variables to the models, they are sent within a "sin" function to smooth out the movements. For example, for the head:

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

It is sent within a sine function, divided by 8. This is because when using the sine function and the variable ranging from -180 to 180 degrees, it is known that sin(180º) = sin(-180º) = sin(0º). Therefore, sin(90º) will be the maximum rotation of the head. By dividing it by 8, it results in the head rotating 11.25º positively and 11.25º negatively.
The same applies to the legs and the tail. For example, for the front left leg (the upper part), the sine is multiplied by PI and then divided by 3.

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

Cow

Fort the cow animation, the cow have to walk in circles while move its legs and its tail. Starting with the variables:

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

The cow model consists of several parts: the head, the body, the 4 legs, and the tail divided into 3 segments. The first 6 variables are used to rotate the tail in both the X and Z directions. The integer variable "accionColaV" is based on a random number and is used to make the tail move from side to side or make the cow lift its tail. "rotarVaca" is used to make the cow walk and also to rotate it in the direction it is walking. After that, there are variables for each leg, and finally, there are variables that determine the position of the cow depending on its rotation.

All variables are initialized to 0, except for the variable that defines how the tail will move:

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

When the value of the variable is 4 or less, then the queue will move from side to side. For this:

```
	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;
	}
```

Each part of the tail has an associated variable that varies. In this case, it only rotates in Z, and when the tail returns to its original position, a random number is obtained again.

When the number is 5 or 6, the cow will raise its tail. These values were chosen so that the cow lifts its tail occasionally and it doesn't appear too continuous.

```
	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;
	}
```

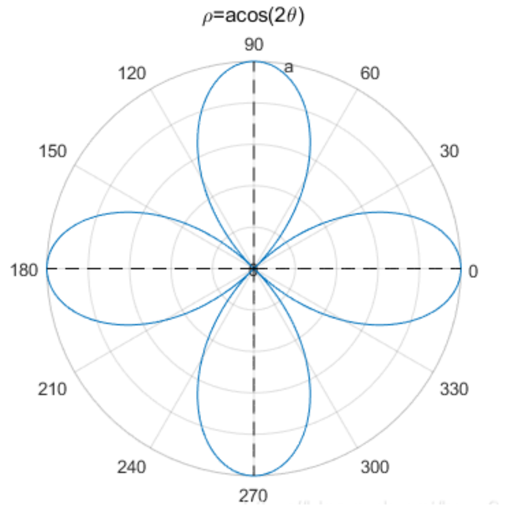
Now the tail rotates in both X and Z axes, but the rotation of the first part of the tail in X determines when to generate the next random number. In reality, it can be any of the three variables in X because they all vary at the same speed. When the tail returns to its original position, a new random number is generated.
Finally, the legs move continuously, and based on the rotation value of the cow, the position is modified in both X and Z axes according to the equation of a circle in polar coordinates converted to Cartesian coordinates:

```
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));
```

According to the same principle as the horse, each variable is passed through sine functions and divided by a certain value in order to smooth out the movements and make the transition between increasing and decreasing variables less abrupt.

Butterfly

The idea behind the butterfly animation is to make it fly in a cycle that appears undefined, as real-life butterflies tend to fly by taking small turns or changing their direction randomly. Programming this behavior exactly is challenging, so we attempted to achieve a similar effect using the following function



The variables used to do this possible are:

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

Firstly, the butterfly's wings will always be in motion.

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

When applying it to the butterfly's wings, the same principle as the horse's legs is followed: passing it through a sine function and dividing it by a value:

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

For the trajectory of the butterfly, we have a polar equation of the form $r = \alpha \sin{(2\theta)}$, where the angle take values $0 < \theta < 2\pi$.To convert it to Cartesian form:

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

Finally, in the Y-axis, both values are multiplied to add a vertical component to the butterfly's flight, making it not entirely flat:

```
//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;
```

Animation by KeyFrames

In this animation, it was decided to animate an additional star apart from the existing star. This second star is smaller and follows a path around the scene while rotating on its own axis. The variables used are:

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

In this way, variables are also declared to indicate the position of the star and the movement it should have. A maximum of 30 frames are defined with 90 interpolations, and a total of 15 states are present.

```
// 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;
```

Additionally, there is a data structure that includes variables for the star's movements, rotation, and their respective increments.

```
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;
```

Finally, the frames are declared:

```
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 contains 15 frames. To read them, the following function was created:

```
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();
}
```

The process begins with reading the file. As long as the file reading is successful, each line is analyzed based on how the data is stored:

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

The code obtains two substrings: one for the index and another for the actual value. The structure of the switch statement is used to determine which data is being read. If the index variable (i) is 0, it reads the movement in the X-axis; if it is 1, it obtains the position in the Y-axis; and if it is 2, it retrieves the rotation

value. When the rotation value is read, i is reset to 0 since the next value to be read is a position in the X-axis.
The variable "linead" helps determine the index up to which the substrings are obtained. For example, if it reaches index 10:

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

the process of obtaining the substrings changes: now, two characters are taken for the index, and an additional right-shifted character is used to obtain the value.
The rest of the provided functions remain unchanged, except that there is no need to save frames, so those functions are not present.

The rendering of the models was based on the knowledge gained throughout the course, exploring the applications of 3D modeling and development tools such as 3ds Max and Blender. In these applications, the models that would be part of the scene were designed, along with their textures. However, it should be noted that some models were obtained from third parties who dedicate their time to recreating real-life objects. With the appropriate rights obtained for freely usable models, they were downloaded and optimized to be used in the creation of the complete diorama.

The following are references to the websites 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/

The reference to the downloaded models used for building the diorama in OpenGL is made:

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