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: _____

Contenido:

Introducción

La computación grafica es un campo en constante crecimiento que abarca el procesamiento y generación de gráficos en tiempo real. OpenGL es una API que proporciona las herramientas necesarias para crear estos gráficos 2D y 3D con técnicas especificas de implementación.

En este documento se mostrará el detalle de implementación técnico que se llevo a cabo para incluir todos los elementos dentro del escenario, estos corresponden a la carga de modelos, recorrido, texturizado, animación e iluminación.

La carga de modelos implica el proceso de importar con el fin de representar los objetos tridimensionales creados a partir de un entorno gráfico, que en este caso se elaboró a partir de las herramientas de 3dsMax y Blender, estos modelos cargados en OpenGL permiten la lectura para el renderizado del modelo con el fin de dibujar las caras y los vértices utilizando los shaders y la configuración de transformaciones adecuada.

El recorrido de la cámara implica el control, posición y orientación de la cámara en el entorno, esta permite al usuario simular el movimiento y la perspectiva en la escena.

La iluminación es una técnica para crear apariencia realista en los entornos generados, en este punto se cuentan con técnicas de iluminación: ambiental, difusa y especular con el fin de resaltar las superficies y objetos en el escenario.

La animación permite la interacción y el movimiento de los objetos en la escena, para lograr este tipo de efecto se utiliza la interpolación, transformaciones básicas y la actualización continua de renderizado.

Este proyecto busca implementar todas las técnicas mencionadas anteriormente con el fin de crear un entorno interactivo y visualmente agradable para lograr una experiencia grafica inmersiva y de calidad.

El proyecto basa su estructura en los siguientes componentes clave:

Recorrido

Cámara isométrica

En los códigos se cuenta con dos archivos que implementan la cámara: *camera.h* y *camera.cpp*. En un inicio, en el archivo main se define la proyección:

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

Camera.h

En camera.h, se declaran dos funciones las cuales servirán para configurar los objetos y la cámara, con el fin de implementar la cámara:

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

Y en general, las variables a utilizar son:

```
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
```

El zoom se coloca en 30 debido a que esté es el tamaño máximo del escenario completo en los ejes X y Z. En un inicio, la cámara isométrica no se muestra.

Camera.cpp

En camera.cpp, la primera función es:

```
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;
}
```

Esta función es usada por todos los objetos dibujados en la escena. Lo que hace son dos rotaciones a diferentes ángulos, con el fin de simular que los ejes X, Y, y Z estén a 120º uno del otro. Entonces, cuando la cámara isométrica está activa, todos los objetos pasan por dos transformaciones antes de sus transformaciones propias.

Dentro de la función `void Camera::keyControl(bool* keys, GLfloat deltaTime)`, se realiza el manejo de las teclas WASD de manera diferente dependiendo de la cámara. Para el caso de la cámara isométrica, estas teclas modifican la posición de la cámara. Si la condición `if (isometric == false)` no se cumple, entonces la cámara isométrica está activa, y las teclas realizan:

```
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);
```

De la misma forma, las flechas arriba y abajo acercan o alejan la cámara:

```
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;
}
```

Dentro de la función `glm::mat4 Camera::calculateViewMatrix()` se obtiene el valor de lookAt. Para cuando la cámara isométrica está activa, se manda la siguiente línea:

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

Archivo main:

Dentro del ciclo main que dibuja toda la escena, se realiza:

```
	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);
	}
```

De esta forma, cuando la cámara isométrica está activa, se usa una proyección ortogonal la cual recibe como parámetros el Zoom, y 30 y -30 para Y, con el fin de que todos los elementos de la escena se puedan dibujar sin que se tapen por el alcance de la cámara.

Como se mencionó anteriormente, la función `glm::mat4 Camera::ConfIsometric(glm::mat4 model)` realiza dos rotaciones adicionales para cada modelo, siempre y cuando la cámara isométrica esté activa. Así que para cada asignación de la matriz identidad de la matriz *model*, se pone la siguiente línea:

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

Iluminación

Tipo DirectionalLight

Para este tipo de iluminación se considera que únicamente debe definirse 1, ya que representa la luz natural y la que nos permitirá modificar el ciclo de día o noche con los parametros de intensidad:

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

	unsigned int pointLightCount = 0;
```

Los primeros tres parámetros corresponden al color de la luz, al ser de tipo natural se define con color blanco, es decir, que se representen todos los colores, posteriormente se definen los parámetros de intensidad, el primero corresponde a la región cónica y el segundo a la intensidad de iluminación; finalmente se define la dirección que tiene la luz y, considerando que es una luz que viene de la parte superior se asigna al eje “y” negativo.

Para modificar la intensidad de la luz natural dependiendo del ciclo del día nos apoyamos de un contador, de los ciclos por segundos obtenidos del programa y de una bandera para cambiar el estado del ciclo; cuando el contador se iguale a una cantidad de ciclos de reloj, entonces verificamos el estado de la bandera, si se encuentra en el día, cambiamos el estado a noche y viceversa; posteriormente reiniciamos el estado del contador para iniciar la cuenta de un nuevo ciclo.

Posteriormente con el estado de la bandera, modificamos la intensidad de la luz dependiendo si es de día: true o false; para el ciclo de noche la intensidad es menor mientras que para el día esta intensidad aumenta.

```
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);
```

Tipo PointLight

Lámpara – Faro

Para este tipo de iluminación se busca implementar la luz puntual que genera al encender una lampara real, estas estarán distribuidas a lo largo del escenario con el fin de crear un ambiente real, se considera el apagado automático dependiendo del ciclo del día, para la noche estas se prenderán automáticamente. Comenzamos definiendo un vector de tipo PointLight con el máximo de dimensión que corresponde al total de luces

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

Dentro de la función main se definen cada uno de los PointLights con la siguiente estructura:

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

Los primeros 3 parámetros de tipo GLfloat (red, green, blue) corresponden al color RGB que se requiera establecer para esa luz, con el parámetro dIntensity definimos la región cónica del alcance y con el parámetro aIntensity determinamos cuando decrece la intensidad de la luz a medida que nos alejamos desde el centro del cono; los siguientes parámetros nos determinan la posición en los tres ejes coordenados y, finalmente los últimos tres parámetros corresponden a los coeficientes de atenuación que se calculan a partir de una ecuación de segundo grado. Una vez especificado lo anterior, se definen las luces del tipo PointLight:

```
    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++;
```

Se definieron con un color rojo, con una intensidad definida de la misma forma para todas, la ubicación corresponde a la ubicación del faro en el plano y los coeficientes de atenuación se definen la misma forma para cada una de ellas.

Luz- Estrella

Adicionalmente se define un PointLight con color amarillo en la posición donde se encuentra la estrella, con una intensidad mayor para abarcar el área total.

```
	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++;
```

Para que las luces definidas anteriormente únicamente se enciendan en el ciclo de día nocturno, enviamos la lista del shader de las luces puntuales que se deben encender cuando sea de noche, en caso de que sea el ciclo de día entonces se envía un parámetro de 1 para indicar que no se debe cargar la información mientras ese ciclo este presente:

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

Luz- Fuente

Se define una iluminación en donde se le da el color azul, con una región de 2.0; adicionalmente se indica la posición en el plano donde se encuentra la fuente, finalmente se definen los parámetros de atenuación calculados a partir de la ecuación de segundo grado.

```
	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++;
```

Tipo SpotLight

Este tipo de luces, al tener una dirección, se decidió incluir dos luces las cuales se ubican en dos esquinas del escenario, y que apuntan hacia la cabaña

Definición de luces

Se comienza definiendo las luces:

```
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++;
```

Estas luces tienen un valor de 5 tanto para la componente ambiental como para la difusa, y en los valores de la ecuación de segundo grado se eligieron esos valores puesto que permiten una mayor apreciación de la luz.

Control de las luces

Con el fin de que el usuario pueda controlar las luces en el escenario, se agregaron variables y banderas las cuales indica cuando una cierta luz está activa o no.

Window.h

En este archivo se definen las variables:

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

Las cuales están inicializadas en cero. Esto hace que las luces no se dibujen al ejecutar el programa. Dentro del mismo archivo, se declararon los get:

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

Window.cpp

Para lograr el control del teclado, se optó por utilizar los números 1, 2, 3 y 4, en donde los primeros 2 manipulan la primer luz, y los siguientes a la segunda:

```
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};
```

Para ver que luces se van a modificar, se usan los métodos definidos anteriormente, y de acuerdo al valor de estos, se modifica el arreglo de bools:

```
		// 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

Para la parte de las animaciones, estás se dividen en 3:

Animaciones simples

Águila – Baúl

Para esta animación se busca que el águila en un inicio se encuentre dentro del baúl que está al interior de la cabaña. Cuando el usuario presiona la tecla B, el baúl se abre y sale volando el águila de su interior, dé la vuelta, y se regrese al baúl, volviendo a dar la vuelta para recuperar su posición original, y finalmente cerrar el baúl. Las variables usadas para esto son:

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

window.h:

Se declara una bandera que servirá para indicar en que momento se debe iniciar la animación:

```
bool abrirBaul = false;
```

Además, se declaran dos métodos, uno servirá para obtener el valor de la bandera, y otro para reiniciarlo en falso cuando la animación haya concluido:

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

window.cpp:

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;
```

Archivo main:

Se creo una función llamada `void AnimateEagle()` en donde se comienza preguntando si la bandera está en true. Cuando la bandera es vuelve true, la animación comienza abriendo la tapa del baúl, en la cual se usa una bandera inicializada en true.

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

```
			rotaTapaBaul = 90.0f;
			animAguila = true;
		}
		else rotaTapaBaul += 3.0f * deltaTime;
	}
```

Cuando el baúl se abre por completo, la bandera `animAguila` se vuelve true. Al momento de que está bandera se vuelve true, otra bandera ayuda a saber si el águila apenas va a salir del baúl o si ya va de regreso. Como `regresaAguila` se inicializa en false, el águila sale del baúl. En ese momento comienza a mover las alas. Con el fin de no usar funciones y de no volver a la animación compleja, se usan comparaciones para mover las alas de arriba abajo y viceversa.

```
		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;
			}
		}
```

Al mismo tiempo, se incrementa la posición del águila en Z hasta 20, y Y hasta 10. Cuando se llega al punto máximo, el águila rota en el aire mientras sigue moviendo las alas. Una vez que complete la rotación, la bandera `regresaAguila` se vuelve true, y la águila regresa.

```
		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;
                        }
                    }
                }
            }
```

El regreso del águila es igual que la ida, con la diferencia de que ya no mueve las alas, ya que se deja caer. Cuando el águila regresa a su posición original, rota nuevamente mientras vuelve a mover las alas, y al llegar a su posición original, las banderas del águila vuelven a sus valores iniciales, a su vez que `openTapa` se vuelve false.

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

Finalmente, la tapa se vuelve a cerrar, y la bandera que se activa por teclado se pone en false, para que se pueda volver a ejecutar la animación al presionar la tecla B.

Hélice del molino:

En esta animación se hace girar la hélice del molino a diferentes velocidades. Para esto se usa una variable que depende de un número aleatorio. En total se tienen 5 velocidades a las cuales la hélice puede girar, dependiendo del valor de la variable. Las variables utilizadas son:

```
//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;
```

Cada que se completa una vuelta de la hélice, se genera un nuevo número aleatorio que define el nuevo valor del Offset de la rotación, en donde toma valores de 0.2 a 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 $\sin(180º) = \text{sen}(-180º) = \sin(0º)$, así que $\sin(90º)$ 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;
    }
```

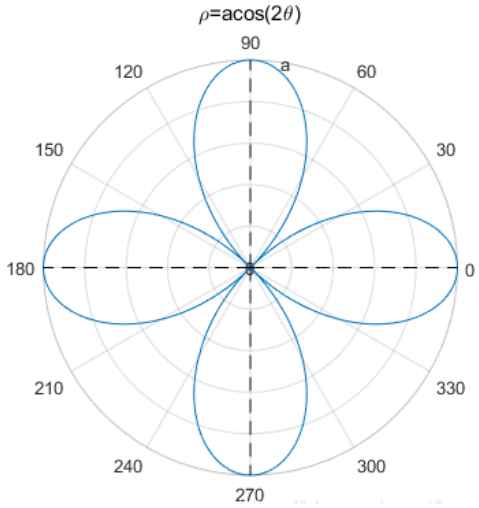
Ahora la cola rota tanto en X como en Z, pero la que decide cuando generar el siguiente numero aleatorio es la rotación de la primera parte de la cola en X. En realidad, puede ser cualquiera, debido a que las 3 variables en X varían a la misma velocidad. Cuando la cola recupera su posición original,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

Los frames están guardados en un archivo externo llamado KeyFrames.txt, el cuál tiene almacenado 15 frames. Para poder leerlos, se generó la función siguiente:

```
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();
}
```

Se comienza por la lectura del archivo. Siempre y cuando la lectura del archivo sea exitosa, se analiza línea por línea. De acuerdo con la forma en la que se almacenan los datos:

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

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

Se obtienen un par de subcadenas, una para el índice, y otra para el propio valor. La estructura del Switch sirve para saber cuál dato se está leyendo: si i = 0, se lee el movimiento en X, si vale 1, se obtiene la posición en Y, y si vale 2, se obtiene el giro. Cuando se lee el giro, i se reinicia a 0 ya que el siguiente valor a leer es una posición en X.
La variable `linead` ayuda a ver hasta que índice se obtienen las subcadenas, ya que por ejemplo, sí se llega al índice 10:

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

La obtención de las subcadenas ya no es igual, si no que ahora se toman dos caracteres para el índice, y un carácter desplazado a la derecha de más para obtener el valor.
El resto de las funciones proporcionadas no se modificaron, con la excepción de que ahora no se tiene la necesidad de guardar frames, por lo que esas funciones no están presentes.

El renderizado de los modelos se basó en el esfuerzo de los conocimientos aprendidos a lo largo del curso conociendo las aplicaciones de desarrollo para el modelado 3D como 3DsMax y Blender, en ellas se diseñaron los modelos que formarían parte del escenario a recrear así como la textura de ellos; sin embargo, se debe reconocer que algunos modelos fueron obtenidos a partir de terceros que dedican el tiempo a recrear objetos de la vida real, con ello, y con los derechos obtenidos con modelos para uso libre se descargaron y se optimizaron con el fin de recrear y hacer uso de ellos en el armado del diorama completo.

A continuación, se hace referencia a las paginas en las que se obtuvieron los modelos descargados:

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