28-10-2015

Sebastian Guisado Barreras y Susana Pineda De Luelmo

Práctica Opcional 1

Informática Gráfica

**Parte Opcional:**

1. **Controla el giro de la cámara utilizando el ratón:**

Lo primero que vamos a hacer es importar la API de Windows (Windows.h), esto nos permite obtener el ID de la ventana que tenemos en activo. Con esto podremos obtener las coordenadas del ratón en relación a la ventana que estamos usando para renderizar nuestro proyecto. Esto se encuentra dentro del archivo includes.h.

Incluimos la API de Windows mediante:

#include <windows.h>

Declaramos la función a la que vamos a llamar cuando se utilice el ratón. Nos devolverá la tecla pulsada y la posición del ratón.

Definimos también la función getMousePos() que nos va a devolver la posición del ratón. Y dos variables booleanas que serán las encargadas de informarnos si el botón derecho o izquierdo están pulsados.

Finalmente declaramos currentWindow que almacenará el ID de la ventana activa para la función posterior.

HWND currentWindow;

Mediante currentWindow = GetForegroundWindow()en el main del archivo main.cpp asignamos la ventana actual a currentWindow.

A continuación inicializamos la variable lastMousePos, de tipo POINT ya que es el encargado de almacenar la última posición del ratón y este tipo es el utilizado por la función que nos devuelve la posición del ratón.

POINT lastMousePos = {-1, -1};

Una vez dentro de la función idleFunc hacemos que si se detecta un click derecho currentPosition tome la posición de getMousePos (explicada e implementada más adelante) y después compare la última posición del ratón. Si la última posición del ratón es (-1,-1) la última posición se actualiza a la posición actual. Si la última posición es diferente a la actual se comparan sus valores y se asigna un movimiento a cada uno llamando a las funciones implementadas para el movimiento de la cámara por teclado. La implementación de esta parte sería la siguiente:

if (leftClick()) {

POINT currentPosition = getMousePos();

if (lastMousePos.x == -1 && lastMousePos.y == -1) lastMousePos = currentPosition;

if (!comparePoints(lastMousePos, currentPosition)){

if (currentPosition.x > lastMousePos.x) rotate\_camera(IZQUIERDA, 0.02f);

if (currentPosition.x < lastMousePos.x) rotate\_camera(DERECHA, 0.02f);

if (currentPosition.y > lastMousePos.y) rotate\_camera(ABAJO, 0.02f);

if (currentPosition.y < lastMousePos.y) rotate\_camera(ARRIBA, 0.02f);

lastMousePos = currentPosition;

}

}

Para saber si el boton derecho o el izquierdo han sido pulsados contamos con

bool leftClick(){ return (GetKeyState(VK\_LBUTTON) & 0x80) != 0; }

bool rightClick(){ return (GetKeyState(VK\_RBUTTON) & 0x80) != 0; }

Que son dos funciones implementadas en la API de Windows.

Por otra parte tenemos la función getMousePos () que es la encargada de devolvernos en coordenadas de la ventana activa la posición del ratón mediante las funciones GetCursorPos y ScreenToClient.

POINT getMousePos(){

POINT mousePos;

GetCursorPos(&mousePos);

if (ScreenToClient(currentWindow, &mousePos)) {

return mousePos;

}

}

También necesitamos comparar las dos posiciones del ratón lo cual haremos mediante la función comparePoints la cual nos devolverá true si son iguales y false en caso contrario.

De esta forma quedaría implementado el movimiento de la cámara mediante el ratón.

1. **Crea un tercer cubo y hazlo orbitar alrededor del primero. Define su movimiento utilizando curvas de Bézier, splines cúbicos o polinomios de interpolación de CatmullRom.**

Para este paso lo primero que tenemos que hacer es crear el tercer cubo, para ello utilizaremos la función definida en el punto 2 de la parte obligatoria y a continuación definiremos sus propiedades:

Primero creamos los identificadores de los objetos de la escena:

int mainCube = -1, secondCube = -1;

Nota: vamos a utilizar los mismos identificadores que en el apartado 2 de la parte obligatoria pero los objetos van a ser los de la parte 3 de la parte opcional.

Al igual que en la parte obligatoria vamos a contar con la definición de los diferentes objetos a partir de las cuales crearemos los objetos que vamos a visualizar para ello subiremos la información a la tarjeta gráfica, para esto nos va a pedir el número de triángulos de vértices, etc. Y nos devuelve un identificador para poder modificar dicho objeto.

#define moon IGlib::createObj(moonNTriangleIndex, moonNVertex, moonTriangleIndex, moonVertexPos, moonVertexColor, moonVertexNormal, moonVertexTexCoord, moonVertexTangent);

#define skullkid IGlib::createObj(skullNTriangleIndex, skullNVertex, skullTriangleIndex, skullVertexPos, skullVertexColor, skullVertexNormal, skullVertexTexCoord, skullVertexTangent);

#define castle IGlib::createObj(castleNTriangleIndex, castleNVertex, castleTriangleIndex, castleVertexPos, castleVertexColor, castleVertexNormal, castleVertexTexCoord, castleVertexTangent);

A continuación vamos a pasar a definir en el main los dos objetos de la escena. El objeto principal mainCube no va a sufrir ninguna alteración en comparación con el apartado 2 de la parte obligatoria (si no fuese porque hemos cambiado el objeto que se va a visualizar, ya no es un cubo):

mainCube = skullkid;

glm::mat4 modelMat = glm::mat4(1.0f);

IGlib::setModelMat(mainCube, modelMat);

IGlib::addColorTex(mainCube, "skull.png");//añadimos una textura al objeto

Creamos el segundo objeto que va a ser el que se mueva describiendo una curva de Bézier (le vamos a asignar otro modelo):

secondCube = moon;

glm::mat4 secondCubeMat = glm::mat4(1.0f);

IGlib::setModelMat(secondCube, secondCubeMat);

IGlib::addColorTex(secondCube, "moon.png"); // añadimos una textura al objeto.

Una vez implementada esta parte en el main del archivo main.cpp de nuestro proyecto pasamos a implementar la parte encargada de la animación de nuestro cubo (en nuestro caso la luna). Esta parte del código se encuentra en idleFunc () la cual es llamada por el procesador cada vez que este está ocioso, es decir, se esta ejecutando constantemente.

Lo primero que vamos a hacer es crear una variable que se encargue de dar valor a la posición inicial de la curva de Bezier, en nuestro caso será el punto t=0:

static float t2 = 0.0f;

Después declaramos que si t es menor que 1 esta se vaya incrementando, por el contrario, si t es mayor que 1 pasa a valer 0, de esta forma nos aseguramos de que hace una curva cerrada.

if (t < 1.0f) t += 0.0002f;

if (t > 1.0f) t = 0.0f;

A continuación pasamos a definir los puntos de control de la curva de Bézier, vamos a seleccionar 5 puntos para poder crear la curva cerrada sin que de problemas notables (saltos del punto final al inicial):

glm::vec2 p0 = { 0, 1 };

glm::vec2 p1 = { 0, 5 };

glm::vec2 p2 = { 0, 1 };

glm::vec2 p3 = { 0, -3 };

glm::vec2 p4 = { 0, 1 };

Una vez que tenemos los puntos de control definidos pasamos a definir las funciones que se encargaran del movimiento de nuestra luna mediante la fórmula general de las curvas de Bézier:

float x = std::pow(1 - t, 4) \* p0.x + 4 \* std::pow(1 - t, 3) \* t \* p1.x + 6 \* std::pow(1 - t, 2) \* std::pow(t, 2) \* p2.x + 4 \* (1 - t) \* std::pow(t, 3) \* p3.x + std::pow(t, 4) \* p4.x;

float y = std::pow(1 - t, 4) \* p0.y + 4 \* std::pow(1 - t, 3) \* t \* p1.y + 6 \* std::pow(1 - t, 2) \* std::pow(t, 2) \* p2.y + 4 \* (1 - t) \* std::pow(t, 3) \* p3.y + std::pow(t, 4) \* p4.y;

Una vez que tenemos definidas las funciones que nos van a dar la posición de nuestro segundo cubo (luna) Pasamos a implementar el resto de movimientos que realiza. Se va a mover de la misma forma que en el apartado 2 de la parte obligatoria pero moviéndose también en Y mediante una curva de Bézier:

glm::mat4 secondCubeMat = glm::mat4(1.0f); //posicion

glm::mat4 rotation = glm::rotate(secondCubeMat, angle, glm::vec3(0.0f, 1.0f, 0.0f)); // Rotación sobre si mismo

glm::mat4 desplz = glm::translate(rotation, glm::vec3(2.0f, 0.0f, 0.0f)); // Desplazamiento sobre si mismo tras la rotación

secondCubeMat = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, y, 0.0f)) \* desplz \* rotation; // Multiplicamos el resultado final (desplz) con la rotación para hacer al cubo girar también sobre si mismo

secondCubeMat = glm::scale(secondCubeMat, glm::vec3(0.5f));

IGlib::setModelMat(secondCube, secondCubeMat);

1. **Crea un nuevo modelo y añádelo a la escena:**

**c) Opción 3: Carga un fichero de un formato conocido.**

Dada la dificultad para renderizar con ASSIMP a la vez que con la librería IGlib hemos decidido crear una función que con la ayuda de esta librería auxiliar exporte lo necesario para crear un archivo Header.h que podamos cargar directamente en nuestro proyecto. Para importar ASSIMP en el proyecto hemos tenido que compilar los binarios necesarios para Visual Studio, para ello usamos el compilador Cmake que nos permite seleccionar para que plataforma queremos usar ASSIMP, de esta forma podemos importar la librería.

Tras compilarlo copiamos los archivos necesarios en las carpetas del proyecto, dentro de la carpeta libs. Una vez tenemos los archivos necesarios dentro de la carpeta del proyecto tenemos que ir a propiedades del proyecto y dentro de la opción vinculador>entrada lo añadimos a dependencias externas. Una vez ha hecho esto ya podemos incluir las funciones de ASSIMP dentro de nuestro proyecto.

Lo primero que tenemos que hacer es incluir la librería en el código mediante:

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include <sstream>

Con esto ya tenemos todo listo para utilizar ASSIMP. Además hemos incluido una librería que nos servirá para poder editar el archivo Header con la información del modelo que queremos exportar.

Una vez dentro de la función exportHeaderFromObj, función que nos va a devolver un fichero con la información del modelo, cargamos la información del modelo en la variable scene de tipo aiScene:

Assimp::Importer importer;

Const aiScene\* scene = importer.ReadFile("castle.obj",

aiProcess\_CalcTangentSpace |

aiProcess\_Triangulate |

aiProcess\_JoinIdenticalVertices |

aiProcess\_SortByPType);

Almacenamos el mesh del objeto, dentro de la variable scene, en la variable mesh de tipo aiMesh.

aiMesh \*mesh = scene->mMeshes[0];

Para evitar las comas que nos dejaría en los decimales al crear el fichero por estar en idioma castellano tenemos que cambiarlo a inglés, de esta forma nos pondrá puntos.

std::locale::global(std::locale("english"));

Abrimos, o creamos en caso de que no exista, el archivo file.h y seleccionamos la opción de escritura (“w”).

Para comenzar a escribir el archivo Header tenemos que guardar el número de vértices y de triángulos que tiene el modelo para ello usamos un stringStream para almacenar los datos que escribiremos.

std::stringstream tempInfo;

tempInfo << "const int cubeNVertex = " << mesh->mNumVertices << ";\n";

tempInfo << "const int cubeNTriangleIndex = " << mesh->mNumFaces << ";\n";

fprintf(f, "%s\n", tempInfo.str().c\_str());

Ahora tenemos que ir almacenando la información de vértices, triángulos, normales, coordenadas de texturas y tangentes, para ello creamos un bucle for que recorra las variables en las que esta guardada la información del modelo. Esta información esta guardada en distintos arrays dentro de la variable mesh.

En el caso de los triángulos tenemos un doble bucle for que recorre en primer lugar mesh->mFaces (que almacena los triángulos) y una vez seleccionado el triangulo recorre mesh->mFaces[i].mIndices (que almacena los índices de los triángulos). Una vez hemos terminado de recorrer el bucle for se escribe la información en el archivo Header.

std::stringstream tempTriangleIndex;

tempTriangleIndex << "const unsigned int cubeTriangleIndex[] = {\n";

for (int i = 0; i < mesh->mNumFaces; i++) {

for (int e = 0; e < mesh->mFaces[i].mNumIndices; e++) {

tempTriangleIndex << mesh->mFaces[i].mIndices[e] << "y ";

}

tempTriangleIndex << " //" << i << "\n";

}

tempTriangleIndex << "};\n\n";

fprintf(f, "%s\n", tempTriangleIndex.str().c\_str());

El resto de datos se guardarán de la misma forma en el archivo, cambiando el formato en el que se tiene que procesar. Una vez llegamos al final se cierra el archivo con la función fclose.

Una vez tengamos el archivo file.h creado tenemos que abrirlo con un editor de texto y tenemos que reemplazar las comas que nos aparecen en los índices de los triángulos por puntos, al pasar la información al archivo en vez de poner un punto al pasar de los 999 pone una coma y al ejecutar el proyecto con ese header nos daría error. Una vez cambiadas esas comas por puntos simplemente tenemos que cambiar las “y” que hemos puesto por comas, las hemos puesto para poder hacer este doble cambio de forma automática mediante cambiar y reemplazar del editor de texto y no tener que ir una por una.