



Universidad Nacional Autónoma de México

Facultad de Ingeniería

Computación Gráfica e Interacción Humano
Computadora

Grupo: 05 - Semestre: 2023-1

Documentación:

Manual Técnico

Fecha de entrega: 10/01/2022

Profesor:

Carlos Aldair Román Balbuena.

Alumno:

Cruz Rangel Leonardo Said



Índice

1. Cámara sintética	3
2. Objetos estáticos	4
3. Objetos dinámicos	6
4. Información de las animaciones	7
5. Conclusiones	23
6. Referencias	23
7. Enlace a recorrido virtual	24
8. Enlace a demostración de animaciones	24



1. Cámara sintética

Para usar la cámara sintética es necesario importar en el programa principal el encabezado **camera.h**, brindado por el profesor de laboratorio durante las prácticas.

```
#include <camera.h>
```

Posteriormente para usar las funciones de dicho encabezado se tiene que crear un objeto que pertenezca a esa clase, de tal forma que el código queda de la siguiente forma:

```
// camera
Camera camera(glm::vec3(0.0f, 200.0f, 700.0f));
```

Dentro del encabezado podemos encontrar constantes por defecto con las cuales se define la velocidad (SPEED) a la que se mueve la cámara, su sensibilidad, el zoom que aplica, etc. Dichos valores pueden modificarse según la experiencia de cada usuario y su comodidad al moverse por la escena.

```
// Default camera values
const float YAW      = -90.0f;
const float PITCH     = 0.0f;
const float SPEED     = 0.5f;
const float SENSITIVITY = 0.7f;
const float ZOOM      = 45.0f;
```

Para mover la cámara con las teclas del teclado, se declaró la función **my_input**. Esta función permite detectar cuando alguien presiona una tecla en el programa. Se trata de una acción de escucha proporcionada por GLFW, y luego verificamos qué tecla se ha presionado con una estructura de control if, como se muestra a continuación:

```
void my_input(GLFWwindow *window, int key, int scancode, int action, int mode)
{
    if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if(glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, (float)deltaTime);
    if(glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
```



```

camera.ProcessKeyboard(BACKWARD, (float)deltaTime);
if(glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    camera.ProcessKeyboard(LEFT, (float)deltaTime);
if(glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    camera.ProcessKeyboard(RIGHT, (float)deltaTime);

```

Para el caso de la cámara se hace uso de **FORWARD**, **BACKWARD**, **LEFT** y **RIGHT** para especificar el movimiento al que se verá sometida la cámara al presionar una tecla.

2. Objetos estáticos

Es requisito imprescindible importar los modelos en formato .OBJ para poder utilizar objetos estáticos en el proyecto, ya que el programa está diseñado para trabajar casi exclusivamente con este tipo de archivo, pues es el que tiene mayor compatibilidad y genera menos errores en su importación.

Para organizar y llevar un registro de los objetos que se añaden al proyecto, se ha creado la carpeta '**objects**' dentro de la carpeta '**resources**', cuya ruta completa es la siguiente: **ProyectoFinal/ProyectoFinal/resources/objects**. Esta carpeta, a su vez, contiene subcarpetas que almacenan tanto las **texturas** como los archivos **.OBJ** y **.MTL** correspondientes a cada uno de los elementos que forman parte de la escena final del proyecto. Es importante asegurarse de que el archivo **.MTL** hace referencia correctamente a las texturas, ya que de lo contrario éstas no se cargarán en el programa.

A continuación se muestra el proceso de carga de un objeto estático a la escena:

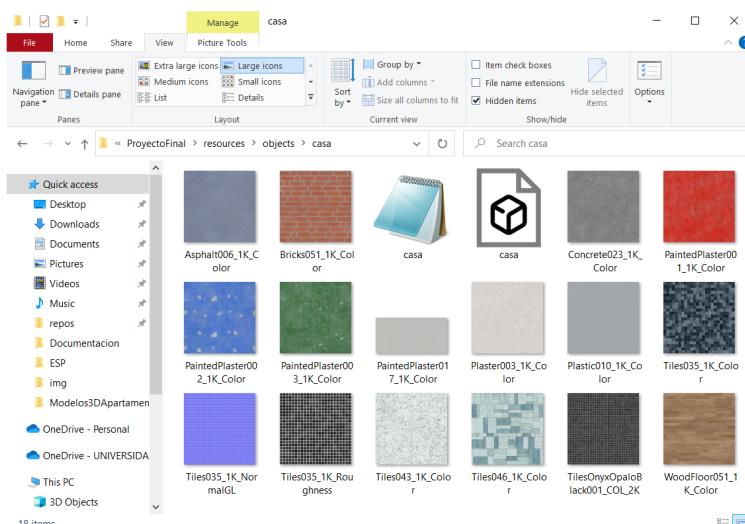


Figura 1. Archivos correspondientes al modelo de la casa.



Una vez que se han almacenado correctamente los archivos en la carpeta ‘**objects**’, el siguiente paso es declarar dicho objeto. En este caso, al tratarse de un objeto estático se usa el **Static Shader** y la clase **Model**.

A continuación se muestra la declaración de algunos de los objetos estáticos del proyecto:

// **CARGA DE MODELOS**

// -----

// **MODELOS ESTÁTICOS**

// -----

// **ARBUSTOS**

Model **arbustos**("resources/objects/arbustos/bush.obj");

// **CASA**

Model **casa**("resources/objects/casa/casa.obj");

// **COCINA**

Model **licuadora**("resources/objects/cocina/blender.obj");

Model **comedor**("resources/objects/cocina/comedor.obj");

Model **estufa**("resources/objects/cocina/estufa.obj");

Model **mantel**("resources/objects/cocina/mantel.obj");

Model **microwave**("resources/objects/cocina/microwave.obj");

Model **mueble**("resources/objects/cocina/mueble.obj");

Model **platos**("resources/objects/cocina/platos.obj");

Model **refrigerador**("resources/objects/cocina/refrigerador.obj");

Model **rollo**("resources/objects/cocina/rollo.obj");

// **CUARTO1**

// **cama1**

Model **almohadas1**("resources/objects/cuarto1/cama/almohadas.obj");

Model **base1**("resources/objects/cuarto1/cama/base.obj");

Model **cabecera1**("resources/objects/cuarto1/cama/cabecera.obj");

Model **colchon1**("resources/objects/cuarto1/cama/colchon.obj");

Model **sabana1**("resources/objects/cuarto1/cama/sabana.obj");



Finalmente solo queda instanciar los objetos, aplicarles las transformaciones deseadas y llamar al **Static Shader** para que puedan dibujarse los objetos en la escena.

```
// -----
// ESCENARIO
// -----
staticShader.use();
staticShader.setMat4("projection", projection);
staticShader.setMat4("view", view);

model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(30.0f));
staticShader.setMat4("model", model);
piso.Draw(staticShader);
```

3. Objetos dinámicos

El proceso para importar modelos dinámicos es similar al que se utiliza para los objetos estáticos, con la única diferencia de que es necesario importar los objetos en formato **.dae**, ya que este es el único archivo de animación compatible con el **Anim Shader** en este proyecto.

La declaración de un objeto dinámico queda de la siguiente forma:

```
// MODELOS DINÁMICOS
// -----
ModelAnim running("resources/objects/Deportista/Running.dae");
running.initShaders(animShader.ID);
```

Posteriormente solo se llama al objeto y de igual manera se le aplican transformaciones de ser necesario, así como el uso del Anim Shader para dibujar el objeto.

```
// ANIMACIONES
// -----
// Remember to activate the shader with the animation
```



```

animShader.use();
animShader.setMat4("projection", projection);
animShader.setMat4("view", view);

animShader.setVec3("material.specular", glm::vec3(0.5f));
animShader.setFloat("material.shininess", 32.0f);
animShader.setVec3("light.ambient", ambientColor);
animShader.setVec3("light.diffuse", diffuseColor);
animShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);
animShader.setVec3("light.direction", lightDirection);
animShader.setVec3("viewPos", camera.Position);

model = glm::translate(glm::mat4(1.0f), glm::vec3(-255.936f + movSport_x, 8.3391f,
-428.04f + movSport_z));
model = glm::rotate(model, glm::radians(orientation), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.35f));
animShader.setMat4("model", model);
running.Draw(animShader);

```

Esto solo dibuja al objeto en el escenario haciendo una animación infinita en su mismo lugar. Por lo que para definir un recorrido se requiere realizar modificaciones en la función Animate, como las que se muestran a continuación.

4. Información de las animaciones

Animación del Coche:

Para fines relacionados con esta animación se crearon las siguientes variables globales:

```

float movAuto_x = 0.0f,
        movAuto_z = 0.0f,
        orienta = 0.0f,
        giollantas = 0.0f,
        movAuto_y = 0.0f,
        escala = 30.0f;

```

Además de que el objeto se dividió en carrocería y llanta.



La variable movAuto aplica la operación de traslación en los tres ejes (x, y, z) sobre los vértices que componen la carrocería del coche. Con orienta se indica un ángulo de rotación para la carrocería en el eje Y, así mismo con giollantas se rota las llantas gradualmente sobre el eje X. Finalmente, con escala se modifica el tamaño del objeto dentro de la escena.

Se dibuja el coche con los valores iniciales, además de que se crea un modelo temporal en la carrocería que después se utiliza para definirlo como punto de partida para dibujar las llantas, esto crea un modelo jerárquico que condiciona a las llantas para que se muevan junto con la carrocería.

//COCHE

```

model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-91.692f + movAuto_x, 39.831f + movAuto_y,
92.421f + movAuto_z));
tmp = model = glm::rotate(model, glm::radians(orienta), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(escala));
staticShader.setMat4("model", model);
coche.Draw(staticShader);

model = glm::translate(tmp, glm::vec3(-24.8199f, -20.589f, -61.383f));
model = glm::scale(model, glm::vec3(escala));
model = glm::rotate(model, glm::radians(giollantas), glm::vec3(1.0f, 0.0f, 0.0f));
staticShader.setMat4("model", model);
llanta.Draw(staticShader); //izq delantera

model = glm::translate(tmp, glm::vec3(25.05f, -20.589f, -61.383f));
model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(escala));
model = glm::rotate(model, glm::radians(giollantas), glm::vec3(-1.0f, 0.0f, 0.0f));
staticShader.setMat4("model", model);
llanta.Draw(staticShader); //der delantera

model = glm::translate(tmp, glm::vec3(-24.8199f, -20.589f, 29.823));
model = glm::scale(model, glm::vec3(escala));
model = glm::rotate(model, glm::radians(giollantas), glm::vec3(1.0f, 0.0f, 0.0f));

```



```
staticShader.setMat4("model", model);
llanta.Draw(staticShader);//izq trasera
```

```
model = glm::translate(tmp, glm::vec3(25.05f, -20.589f, 29.823f));
model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(escala));
model = glm::rotate(model, glm::radians(giollantas), glm::vec3(-1.0f, 0.0f, 0.0f));
staticShader.setMat4("model", model);
llanta.Draw(staticShader);//der trasera
```

Para la animación del coche se definió una ruta dentro de la escena en donde el coche deberá pasar por diferentes estados, en total, 6 estados del 1 al 6 que se representan aplicando la estructura de control Switch/Case en donde el coche y las llantas cambiarán su posición, rotarán, etc.

Esta animación termina al llegar al sexto estado, en donde el coche cambia su escala a 0 y por ende desaparece de la escena.

//Vehículo

```
if (animacion)
{
    switch (car)
    {
        case 1:
            movAuto_z += 3.0f;
            giollantas += 3.0f;
            if (movAuto_z >= 240.0f) {
                orienta = -90.0f;
                movAuto_y = -9.5595;
                car = 2;
            }
            break;
        case 2:
            movAuto_x -= 3.0f;
            giollantas += 3.0f;
```



```
if(movAuto_x <= -240.0f)
{
    orienta = -180.0f;
    car = 3;
}
break;

case 3:
    movAuto_z -= 3.0f;
    giollantas += 3.0f;
    if(movAuto_z <= 0.0f) {
        car = 4;
    }
    break;

case 4:
    movAuto_z += 5.0f;
    giollantas -= 5.0f;
    if(movAuto_z >= 320.0f) {
        orienta = -90.0f;
        car = 5;
    }
    break;

case 5:
    movAuto_x += 5.0f;
    giollantas -= 5.0f;
    if(movAuto_x >= 500.0f) {
        orienta = 0.0f;
        car = 6;
    }
    break;

case 6:
    movAuto_z -= 5.0f;
    giollantas -= 5.0f;
    if(movAuto_z <= -600.0f) {
        escala = 0.0f;
        animacion = false;
    }
}
```

```

        }
        break;
    }
}

```

Para activar la animación el usuario deberá presionar la tecla ‘SPACE’ y una vez que el coche haya desaparecido tendrá la posibilidad de restablecer la animación al presionar la tecla ‘R’.

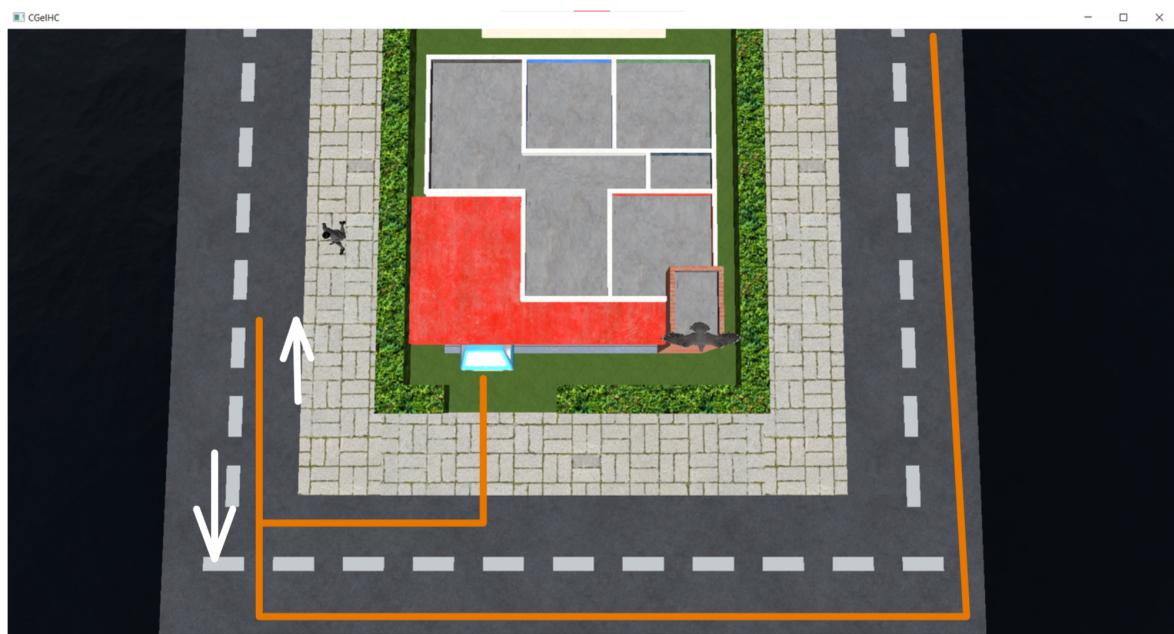


Figura 2. Ruta que sigue el coche hasta terminar su animación.

Animación del Deportista:

Para generar esta animación se crearon las siguientes variables globales:

```

float movSport_x = 0.0f,
      movSport_z = 0.0f,
      orientation = 0.0f;

```

Que controlan el movimiento del deportista sobre el eje x y z, además de una variable que modifica la orientación a la que apunta el deportista.

Esta animación es la única que utiliza objetos dinámicos, en concreto, el modelo del deportista. Este modelo se descargó de Mixamo junto con la animación 'Running', y se



especificó que la animación no tuviera desplazamiento para que éste se generara a través de código.

El objeto se dibujó a través del Anim Shader, pero antes se le aplicaron transformaciones de translación, rotación y escala junto con las variables que permiten la rotación y el movimiento sobre el eje X y Z.

```
model = glm::translate(glm::mat4(1.0f), glm::vec3(-255.936f + movSport_x, 8.3391f,
-428.04f + movSport_z));
model = glm::rotate(model, glm::radians(orientation), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.35f));
animShader.setMat4("model", model);
running.Draw(animShader);
```

A través de una estructura Switch se escoge entre los diferentes estados por los que pasará el deportista, que en este caso corresponden a su movimiento sobre el eje X y el Z. Dentro de cada Case se encuentra una sentencia if que hace una validación con la cual se permite el cambio de estado y giro del deportista para que siga una ruta determinada.

```
switch (sport)
{
    case 1:
        movSport_z += 5.0f;
        if(movSport_z >= 700.0f) {
            orientation = 90.0f;
            sport = 2;
        }
        break;
    case 2:
        movSport_x += 5.0f;
        if(movSport_x >= 500.0f) {
            orientation = 180.0f;
            sport = 3;
        }
        break;
}
```

case 3:

```
movSport_z -= 5.0f;  
if(movSport_z <= 0.0f)  
{  
    orientation = 0.0f;  
    movSport_z = 0;  
    movSport_x = 0;  
    sport = 1;  
}  
break;  
default:  
break;  
}
```

No es necesario que el usuario oprima una tecla para que la animación comience a ejecutarse, pues se ha definido para que se repita cada ciclo del programa.



Figura 3. Ruta que sigue el deportista.

Animación del Halcón:

Esta animación se hizo por medio de KeyFrames usando la línea del tiempo que incluye el código y que es capaz de guardar las propiedades (traslación, rotación, escala, etc.) de un



objeto en un momento dado para posteriormente reproducirlas y crear una animación sencilla por medio de interpolación automática.

Para este fin se utilizan las siguientes variables globales, las primeras tres son para modificar la posición del halcón en los tres ejes, rotAla permite controlar la rotación de las Alas sobre el eje Z y giro para controlar la orientación del halcón en el eje Y.

Seguidamente se definen los incrementos (inc) que serán los valores para los frames intermedios entre cada KeyFrame, es decir, el interpolado.

//Keyframes (Manipulación y dibujo)

```
float posX = 0.0f,
      posY = 0.0f,
      posZ = 0.0f,
      rotAla = 0.0f,
      giro = 0.0f;

float incX = 0.0f,
      incY = 0.0f,
      incZ = 0.0f,
      rotInc = 0.0f,
      giroInc = 0.0f;
```

Posteriormente se definen las funciones que nos permiten guardar las propiedades de los objetos (los KeyFrames) y algunas otras que son necesarias para la interpolación. Así mismo se describen variables y constantes para establecer la cantidad máxima de keyframes que podemos guardar, la cantidad de pasos entre cada frame, etc.

```
#define MAX_FRAMES 18
int i_max_steps = 60;
int i_curr_steps = 0;
typedef struct _frame
{
    //Variables para GUARDAR Key Frames
    float posX;           //Variable para PosicionX
    float posY;           //Variable para Posicion Y
```



```
float posZ;           //Variable para PosicionZ
float rotAla;
float giro;

}FRAME;

FRAME KeyFrame[MAX_FRAMES];
int FrameIndex = 17;           //introducir datos
bool play = false;
int playIndex = 0;

void saveFrame(void)
{
    //printf("frameindex %d\n", FrameIndex);
    //std::cout << "Frame Index = " << FrameIndex << std::endl;

    KeyFrame[FrameIndex].posX = posX;
    KeyFrame[FrameIndex].posY = posY;
    KeyFrame[FrameIndex].posZ = posZ;

    KeyFrame[FrameIndex].rotAla = rotAla;
    KeyFrame[FrameIndex].giro = giro;

    cout << posX << endl;
    cout << posY << endl;
    cout << posZ << endl;
    cout << rotAla << endl;
    cout << giro << endl;

    FrameIndex++;
}

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



```

posY = KeyFrame[0].posY;
posZ = KeyFrame[0].posZ;

rotAla = KeyFrame[0].rotAla;
giro = KeyFrame[0].giro;
}

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

    rotInc = (KeyFrame[playIndex + 1].rotAla - KeyFrame[playIndex].rotAla) /
i_max_steps;
    giroInc = (KeyFrame[playIndex + 1].giro - KeyFrame[playIndex].giro) /
i_max_steps;

}

```

Después, en la función Animate se describen los pasos y el proceso que sigue la interpolación al ejecutar la animación, es decir al darle “play” para que se haga la interpolación entre cada KeyFrame guardado.

```

if(play)
{
    if(i_curr_steps >= i_max_steps) //end of animation between frames?
    {
        playIndex++;
        if(playIndex > FrameIndex - 2)      //end of total animation?
        {
            //std::cout << "Animation ended" << std::endl;
        }
    }
}

```



```
//printf("termina anim\n");
playIndex = 0;
play = false;
}
else //Next frame interpolations
{
    i_curr_steps = 0; //Reset counter
        //Interpolation
    interpolation();
}
}
else
{
    //Draw animation
    posX += incX;
    posY += incY;
    posZ += incZ;

    rotAla += rotInc;
    giro += giroInc;

    i_curr_steps++;
}
}
```

En el programa, para guardar un KeyFrame se tiene que presionar la tecla ‘L’ y para correr la animación se tiene que presionar la tecla ‘P’.

Hasta este momento solo se tiene una forma de guardar KeyFrames para crear animaciones que duren únicamente el tiempo de ejecución del programa, pues una vez que este se cierre la animación se perderá.

Por lo anterior, se añadió código adicional para leer archivos .txt que contuviera los valores de las propiedades de los objetos para cada KeyFrame y así poder ejecutar la animación al



presionar la tecla ‘P’, por lo que ahora podemos crear animaciones más largas y almacenarlas en archivos .txt.

//ANIMACIÓN KEYFRAMES CON ARCHIVO

//1. Abrimos el archivo

```
ifstream archivo;
archivo.open("resources/animacion/keyframes.txt");
```

//2. Leemos línea por línea

```
if(archivo.is_open()) {
    for (int i = 0; i < MAX_FRAMES; i++) {
        for (int j = 0; j < 5; j++) {
            getline(archivo, line);
            switch (j) {
                case 0: KeyFrame[i].posX = stof(line);
                break;
                case 1: KeyFrame[i].posY = stof(line);
                break;
                case 2: KeyFrame[i].posZ = stof(line);
                break;
                case 3: KeyFrame[i].rotAla = stof(line);
                break;
                case 4: KeyFrame[i].giro = stof(line);
                break;
            }
        }
    }
}
```

//3. Cerramos el archivo

```
archivo.close();
```



Precisamente esto fue lo que se hizo con el halcón, primero se creó una animación dentro del programa y se almacenaron las propiedades del objeto para cada KeyFrame en un archivo .txt que después se lee, y al presionar la tecla ‘P’ se hace la interpolación para crear la animación de vuelo para el halcón en donde se modifica su posición y orientación.

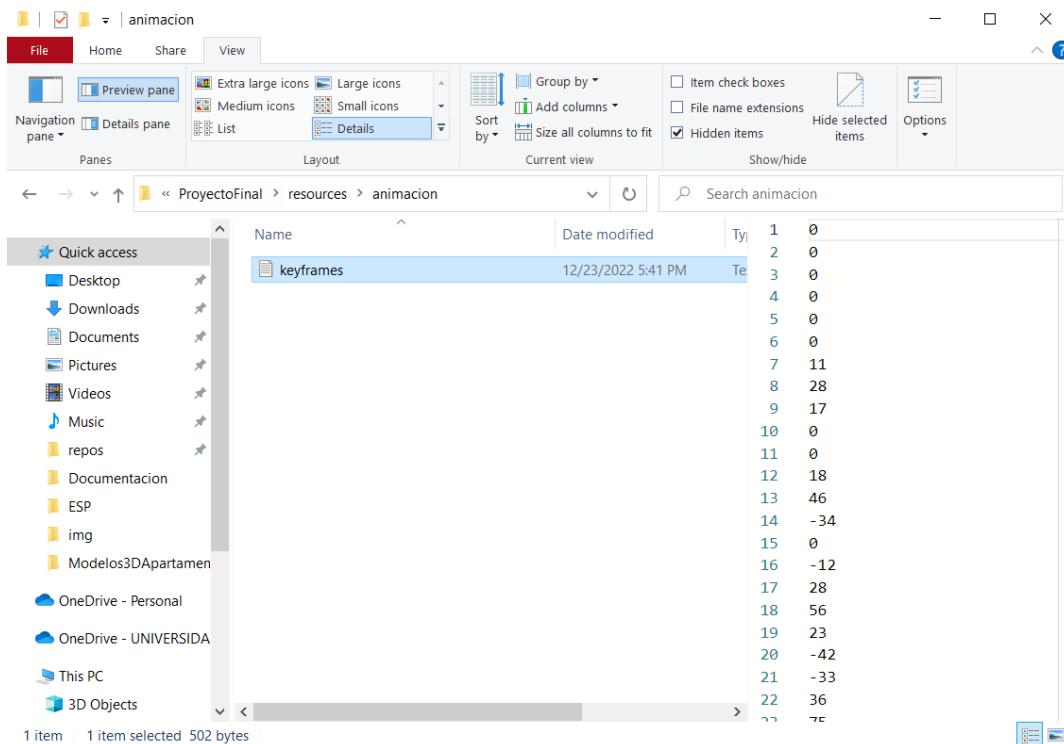


Figura 4. Ubicación del archivo que contiene los valores para los KeyFrames.

Dibujo del halcón, se dibuja junto con las variables que permiten su cambio de posición y rotación, además se crea una matriz temporal que sirve como punto de partida para dibujar las alas y que estas se muevan junto con el cuerpo del halcón:

//AVE

```
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(119.259f, 155.056f, 139.356f));
model = glm::translate(model, glm::vec3(posX, posY, posZ));
tmp = model = glm::rotate(model, glm::radians(giro), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(20.0f));
staticShader.setMat4("model", model);
cuerpo.Draw(staticShader);
```

//Ala izq



```
model = glm::translate(tmp, glm::vec3(-2.883f, 7.0504f, 0.51212f));
model = glm::rotate(model, glm::radians(-rotAla), glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::scale(model, glm::vec3(20.0f));
staticShader.setMat4("model", model);
alaizq.Draw(staticShader);
```

//Ala der

```
model = glm::translate(tmp, glm::vec3(2.883f, 7.0504f, 0.51212f));
model = glm::rotate(model, glm::radians(rotAla), glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::scale(model, glm::vec3(20.0f));      staticShader.setMat4("model", model);
alader.Draw(staticShader);
```

Recordatorio: Para ejecutar la animación debe presionar la tecla ‘P’.

Animación del Helicóptero a control remoto:

Para fines relacionados con esta animación se crearon las siguientes variables globales:

```
float movHeli_x = 0.0f,
      movHeli_y = 0.0f,
      movHeli_z = 0.0f,
      orient = 0.0f;
```

Que como ya se ha explicado, permiten modificar la posición del helicóptero en los diferentes ejes de referencia además de su rotación para apuntarlo en diferentes direcciones.

La animación sigue un proceso muy similar al del coche. Se dividió el modelo en cuerpo del helicóptero y en hélices. Además de que su recorrido se define por medio de estados y una estructura if-else. El helicóptero cuenta con 6 estados o recorridos diferentes.

//Helicoptero

```
if (animacion2) {
    if (recorrido1) {
        movHeli_x = 0.0f;
        movHeli_z += 3.0f;
        orient = 0.0f;
    if (movHeli_z >= 200.0f) {
```



```
    recorrido1 = false;
    recorrido5 = true;
}
}

else if (recorrido2) {
    movHeli_x -= 4.0f;
    movHeli_z = 200.0f;
    orient = 270.0f;
    if (movHeli_x <= -250.0f) {
        recorrido2 = false;
        recorrido3 = true;
    }
}

else if (recorrido3) {
    movHeli_x = -250.0f;
    movHeli_z -= 3.0f;
    orient = 180.0f;
    if (movHeli_z <= -200.0f) {
        recorrido3 = false;
        recorrido4 = true;
    }
}

else if (recorrido4) {
    movHeli_x += 4.0f;
    movHeli_z = -200.0f;
    orient = 90.0f;
    if (movHeli_x >= 0.0f) {
        recorrido4 = false;
        recorrido1 = true;
    }
}

else if (recorrido5) {
    movHeli_x -= 3.0f;
    movHeli_z -= 3.0f * (400.0f / 250.0f);
    orient = 212.0f;
}
```

```

if(movHeli_x <= -250.0f && movHeli_z <= -200.0f) {
    recorrido5 = false;
    recorrido6 = true;
}
else if(recorrido6) {
    movHeli_x += 3.0f;
    movHeli_z += 3.0f * (200.0f / 250.0f);
    orient = 90.0f - 38.65f;
    if(movHeli_x >= 0.0f && movHeli_z >= 0.0f) {
        recorrido6 = false;
        recorrido1 = true;
    }
}
};


```

La principal diferencia con la animación del coche es que esta animación presenta una complejidad más alta, pues contempla giros que se hacen de forma gradual e incluso cambios en la velocidad del helicóptero a lo largo de sus recorridos. Para lograr esto, se requirió de cálculos matemáticos y experimentación a través de prueba y error hasta lograr el comportamiento esperado.

Para activar la animación, presionar la tecla ‘5’.



Figura 5. Recorrido con curvas que realiza el helicóptero.



5. Conclusiones

El resultado obtenido de este proyecto me ha dejado satisfecho, ya que por medio de las herramientas y habilidades adquiridas durante el curso fue posible reflejar de forma adecuada la idea plasmada en la propuesta; creando una fachada de una casa con los elementos esenciales y sus respectivos exteriores para darle realismo.

El proyecto requirió una significativa inversión de tiempo y esfuerzo debido a la complejidad del modelado, texturizado y animado de modelos para cumplir con los requisitos del proyecto. Sin embargo, considero que existe la oportunidad de añadir más detalles a la casa, como incrementar la cantidad de modelos, añadir diferentes tipos de iluminación y animaciones, entre otros.

En conclusión, se han alcanzado los objetivos planteados en este proyecto al lograr plasmar las ideas desarrolladas en la propuesta y aplicar el conocimiento adquirido durante el curso de laboratorio y teoría. Esta experiencia sin duda será valiosa para mi desarrollo como Ingeniero en Computación, ya que los gráficos tienen una amplia presencia en la vida cotidiana y es importante contar con las habilidades necesarias para desempeñarse de manera efectiva en la profesión.

6. Referencias

Computación Gráfica Introducción y Conceptos Fundamentales. (n.d.). U-Cursos. Retrieved

December 8, 2022, from

https://www.u-cursos.cl/ingenieria/2010/2/CC3501/1/material_docente/bajar%3Fid_material%3D306628

Ilet. (n.d.). *La computación gráfica: Historia, objetivos y aplicaciones.* Ilet. Retrieved

December 8, 2022, from <https://ilet.mx/toluca/concepto-computacion-grafica/>

Plano de casa de 1 piso con 3 habitaciones 2 baños (DWG). (n.d.). Verplanos.com. Retrieved

November 18, 2022, from



<https://verplanos.com/plano-de-casa-con-cubierta-mediterranea-de-1-piso-y-3-habitaciones/>

Villalta, H. (n.d.). *Tutorial Comentado Apartamento en Blender 3.0*. YouTube. Retrieved

November 20, 2022, from

<https://www.youtube.com/watch?v=lekcqJNMGHA&t=7490s>

7. Enlace a recorrido virtual

<https://www.youtube.com/watch?v=XWHkK6jfIg>

8. Enlace a demostración de animaciones

<https://www.youtube.com/watch?v=0kX-gMDSoKM>