UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA ELÉCTRICA

INGENIERÍA EN COMPUTACIÓN

LABORATORIO DE COMPUTACIÓN GRÁFICA e INTERACCIÓN HUMANO COMPUTADORA

EJERCICIOS DE CLASE Nº 5

NOMBRE COMPLETO: Razo Villeda Fernando

Nº de Cuenta: 318299475

GRUPO DE LABORATORIO: 2

GRUPO DE TEORÍA: 4

SEMESTRE 2024-2

FECHA DE ENTREGA LÍMITE: 16/03/2024

CALIFICACIÓN: _____

Ejercicio 1: Importar por separado y agregar jerarquía

El código carga varios modelos 3D y los renderiza por separado. Los modelos cargados son:
- Cuerpo
- Cabeza
- Mandíbula
- Una pata delantera y una trasera.

La jerarquía se implementa mediante el uso de transformaciones de matrices. Cada modelo tiene su propia matriz de modelo (model) que se utiliza para aplicar transformaciones de traslación, rotación y escala.

Por ejemplo, la posición y orientación de la cabeza y la mandíbula están relacionadas con la posición y orientación del cuerpo, mientras que las posiciones y orientaciones de las patas están relacionadas con la posición y orientación del cuerpo también.

A medida que se dibuja cada componente de Goddard en el bucle principal del programa, las transformaciones se aplican en cascada según la jerarquía. Por ejemplo:

1. La posición y orientación del cuerpo se establecen en la matriz de modelo inicial.
2. Se trasladan y rotan las patas en relación con el cuerpo.
3. Cada pata puede tener su propia rotación adicional según el ángulo de articulación específico que se le haya asignado.
4. Luego, se traslada y rota la cabeza y la mandíbula en relación con el cuerpo.

Por lo tanto, la jerarquía se establece asegurando que cada componente se posicione y rote correctamente en relación con el nodo padre en la jerarquía. Esto se logra mediante la manipulación adecuada de las matrices de modelo y la aplicación de las transformaciones necesarias.

```cpp
/*
Práctica 5: Optimización y Carga de Modelos
*/
 //para cargar imagen
 #define STB_IMAGE_IMPLEMENTATION

#include <stdio.h>
#include <string.h>
#include <cmath>
#include <vector>
#include <math.h>

#include <glew.h>
#include <glfw3.h>

#include <glm.hpp>
#include <gtc\matrix_transform.hpp>
#include <gtc\type_ptr.hpp>
//para probar el importer
//#include<assimp/Importer.hpp>

#include "Window.h"
#include "Mesh.h"
#include "Shader_m.h"
#include "Camera.h"
#include "Sphere.h"
#include"Model.h"
#include "Skybox.h"

 const float toRadians = 3.14159265f / 180.0f;
 //float angulocola = 0.0f;
 Window mainWindow;
 std::vector<Mesh*> meshList;
 std::vector<Shader> shaderList;


 Camera camera;
 //cada variable es el modelo que utilizaré
 Model Goddard_M, mandibula, cabeza, piernaAdelante, piernaAtras, cuerpo;
 Skybox skybox;

GLfloat deltaTime = 0.0f;
GLfloat lastTime = 0.0f;
static double limitFPS = 1.0 / 60.0;


// Vertex Shader
static const char* vShader = "shaders/shader_m.vert";

// Fragment Shader
static const char* fShader = "shaders/shader_m.frag";




void CreateObjects()
{
    unsigned int indices[] = {
        0, 3, 1,
        1, 3, 2,
        2, 3, 0,
        0, 1, 2
    };

    GLfloat vertices[] = {
        //  x      y      z          u     v          nx    ny    nz
            -1.0f, -1.0f, -0.6f,     0.0f, 0.0f,     0.0f, 0.0f, 0.0f,
            0.0f, -1.0f, 1.0f,       0.5f, 0.0f,     0.0f, 0.0f, 0.0f,
            1.0f, -1.0f, -0.6f,      1.0f, 0.0f,     0.0f, 0.0f, 0.0f,
            0.0f, 1.0f, 0.0f,        0.5f, 1.0f,     0.0f, 0.0f, 0.0f
    };

    unsigned int floorIndices[] = {
        0, 2, 1,
        1, 2, 3
    };

    GLfloat floorVertices[] = {
        -10.0f, 0.0f, -10.0f,   0.0f, 0.0f,     0.0f, -1.0f, 0.0f,
        10.0f, 0.0f, -10.0f,    10.0f, 0.0f,    0.0f, -1.0f, 0.0f,
```

```
        -10.0f, 0.0f, 10.0f,    0.0f, 10.0f,    0.0f, -1.0f, 0.0f,
        10.0f, 0.0f, 10.0f,     10.0f, 10.0f,   0.0f, -1.0f, 0.0f
    };


    Mesh* obj1 = new Mesh();
    obj1->CreateMesh(vertices, indices, 32, 12);
    meshList.push_back(obj1);

    Mesh* obj2 = new Mesh();
    obj2->CreateMesh(vertices, indices, 32, 12);
    meshList.push_back(obj2);

    Mesh* obj3 = new Mesh();
    obj3->CreateMesh(floorVertices, floorIndices, 32, 6);
    meshList.push_back(obj3);


}


void CreateShaders()
{
    Shader* shader1 = new Shader();
    shader1->CreateFromFiles(vShader, fShader);
    shaderList.push_back(*shader1);
}



int main()
{
    mainWindow = Window(1366, 768); // 1280, 1024 or 1024, 768
    mainWindow.Initialise();

    CreateObjects();
    CreateShaders();

    camera = Camera(glm::vec3(0.0f, 0.5f, 7.0f), glm::vec3(0.0f, 1.0f, 0.0f), -60.0f, 0.0f, 0.3f, 1.0f);
```

```

    cabeza = Model();
    cabeza.LoadModel("modelos/cabeza.obj");
    cuerpo = Model();
    cuerpo.LoadModel("modelos/cuerpo.obj");
    mandibula = Model();
    mandibula.LoadModel("modelos/mandibula.obj");
    piernaAdelante = Model();
    piernaAdelante.LoadModel("modelos/pataAdelante.obj");
    piernaAtras = Model();
    piernaAtras.LoadModel("modelos/pataAtras.obj");


    std::vector<std::string> skyboxFaces;
    skyboxFaces.push_back("Textures/Skybox/cupertin-lake_rt.tga");
    skyboxFaces.push_back("Textures/Skybox/cupertin-lake_lf.tga");
    skyboxFaces.push_back("Textures/Skybox/cupertin-lake_dn.tga");
    skyboxFaces.push_back("Textures/Skybox/cupertin-lake_up.tga");
    skyboxFaces.push_back("Textures/Skybox/cupertin-lake_bk.tga");
    skyboxFaces.push_back("Textures/Skybox/cupertin-lake_ft.tga");

    skybox = Skybox(skyboxFaces);

    GLuint uniformProjection = 0, uniformModel = 0, uniformView = 0, uniformEyePosition = 0,
        uniformSpecularIntensity = 0, uniformShininess = 0;
    GLuint uniformColor = 0;
    glm::mat4 projection = glm::perspective(45.0f, (GLfloat)mainWindow.getBufferWidth() / mainWindow.getBufferHeight(), 0.1f, 1000.0f);


    glm::mat4 model(1.0);
    glm::mat4 modelaux(1.0);
    glm::vec3 color = glm::vec3(1.0f, 1.0f, 1.0f);

    ////Loop mientras no se cierra la ventana
    while (!mainWindow.getShouldClose())
    {
        GLfloat now = glfwGetTime();
        deltaTime = now - lastTime;
        deltaTime += (now - lastTime) / limitFPS;
```

```
        lastTime = now;

        //Recibir eventos del usuario
        glfwPollEvents();
        camera.keyControl(mainWindow.getsKeys(), deltaTime);
        camera.mouseControl(mainWindow.getXChange(), mainWindow.getYChange());

        // Clear the window
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        //Se dibuja el Skybox
        skybox.DrawSkybox(camera.calculateViewMatrix(), projection);
        shaderList[0].UseShader();
        uniformModel = shaderList[0].GetModelLocation();
        uniformProjection = shaderList[0].GetProjectionLocation();
        uniformView = shaderList[0].GetViewLocation();
        uniformColor = shaderList[0].getColorLocation();
        glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
        glUniformMatrix4fv(uniformView, 1, GL_FALSE, glm::value_ptr(camera.calculateViewMatrix()));

        // INICIA DIBUJO DEL PISO
        color = glm::vec3(0.5f, 0.5f, 0.5f); //piso de color gris
        model = glm::mat4(1.0);
        model = glm::translate(model, glm::vec3(0.0f, -2.0f, 0.0f));
        model = glm::scale(model, glm::vec3(30.0f, 1.0f, 30.0f));
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
        glUniform3fv(uniformColor, 1, glm::value_ptr(color));
        meshList[2]->RenderMesh();

        // Cuerpo
        color = glm::vec3(0.0f, 1.0f, 0.0f); // Define el color del cuerpo de Goddard como verde.
        model = glm::mat4(1.0); // Inicializa la matriz de modelo como la identidad.
        model = glm::translate(model, glm::vec3(0.0f, 0.8f, 0.0f)); // Traslada el cuerpo a la posición deseada.
        modelaux = model; // Almacena una copia de la matriz de modelo para su posterior uso.
        glUniform3fv(uniformColor, 1, glm::value_ptr(color)); // Asigna el color al shader.
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model)); // Asigna la matriz de modelo al shader.
        cuerpo.RenderModel(); // Renderiza el cuerpo de Goddard.

        // Pata frontal izquierda
        model = modelaux; // Reinicializa la matriz de modelo con la copia almacenada.
        model = glm::translate(model, glm::vec3(1.0f, -0.6f, -9.4f)); // Traslada la pata a la posición deseada.
        model = glm::rotate(model, glm::radians(mainWindow.getarticulacion2()), glm::vec3(0.0f, 0.0f, 1.0f)); // Rota la pata según el ángulo de articulac
        color = glm::vec3(1.0f, 1.0f, 0.0f); // Define el color de la pata de Goddard como amarillo.
        glUniform3fv(uniformColor, 1, glm::value_ptr(color)); // Asigna el color al shader.
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model)); // Asigna la matriz de modelo al shader.
        piernaAdelante.RenderModel(); // Renderiza la pata delantera izquierda de Goddard.

        // Pata frontal derecha
        model = modelaux; // Reinicializa la matriz de modelo con la copia almacenada.
        model = glm::translate(model, glm::vec3(1.0f, -0.6f, -10.6f)); // Traslada la pata a la posición deseada.
        model = glm::rotate(model, glm::radians(mainWindow.getarticulacion1()), glm::vec3(0.0f, 0.0f, 1.0f)); // Rota la pata según el ángulo de articulac
        color = glm::vec3(1.0f, 1.0f, 0.0f); // Define el color de la pata de Goddard como amarillo.
        glUniform3fv(uniformColor, 1, glm::value_ptr(color)); // Asigna el color al shader.
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model)); // Asigna la matriz de modelo al shader.
        piernaAdelante.RenderModel(); // Renderiza la pata delantera derecha de Goddard.

        // Pata posterior izquierda
        model = modelaux; // Reinicializa la matriz de modelo con la copia almacenada.
        model = glm::translate(model, glm::vec3(-0.2f, -1.3f, -9.4f)); // Traslada la pata a la posición deseada.
        model = glm::rotate(model, glm::radians(mainWindow.getarticulacion4()), glm::vec3(0.0f, 0.0f, 1.0f)); // Rota la pata según el ángulo de articulac
        color = glm::vec3(1.0f, 1.0f, 0.0f); // Define el color de la pata de Goddard como amarillo.
        glUniform3fv(uniformColor, 1, glm::value_ptr(color)); // Asigna el color al shader.
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model)); // Asigna la matriz de modelo al shader.
        piernaAtras.RenderModel(); // Renderiza la pata posterior izquierda de Goddard.

        // Pata posterior derecha
        model = modelaux; // Reinicializa la matriz de modelo con la copia almacenada.
        model = glm::translate(model, glm::vec3(-0.2f, -1.3f, -10.6f)); // Traslada la pata a la posición deseada.
        model = glm::rotate(model, glm::radians(mainWindow.getarticulacion3()), glm::vec3(0.0f, 0.0f, 1.0f)); // Rota la pata según el ángulo de articulac
        color = glm::vec3(1.0f, 1.0f, 0.0f); // Define el color de la pata de Goddard como amarillo.
        glUniform3fv(uniformColor, 1, glm::value_ptr(color)); // Asigna el color al shader.
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model)); // Asigna la matriz de modelo al shader.
        piernaAtras.RenderModel(); // Renderiza la pata posterior derecha de Goddard.
```

```
234         // Cabeza
235         model = modelaux; // Reinicializa la matriz de modelo con la copia almacenada.
236         model = glm::translate(model, glm::vec3(1.8f, 0.48f, -10.4f)); // Traslada la cabeza a la posición deseada.
237         model = glm::rotate(model, glm::radians(mainWindow.getarticulacion5()), glm::vec3(0.0f, 0.0f, 1.0f)); // Rota la ca
238         color = glm::vec3(0.0f, 0.0f, 1.0f); // Define el color de la cabeza de Goddard como azul.
239         glUniform3fv(uniformColor, 1, glm::value_ptr(color)); // Asigna el color al shader.
240         glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model)); // Asigna la matriz de modelo al shader.
241         cabeza.RenderModel(); // Renderiza la cabeza de Goddard.
242 
243         // Mandíbula
244         model = glm::translate(model, glm::vec3(1.0f, 0.2f, 0.5f)); // Traslada la mandíbula a la posición deseada.
245         model = glm::rotate(model, glm::radians(mainWindow.getarticulacion6()), glm::vec3(0.0f, 0.0f, 1.0f)); // Rota la ma
246         color = glm::vec3(1.0f, 0.0f, 1.0f); // Define el color de la mandíbula de Goddard como rosa.
247         glUniform3fv(uniformColor, 1, glm::value_ptr(color)); // Asigna el color al shader.
248         glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model)); // Asigna la matriz de modelo al shader.
249         mandibula.RenderModel(); // Renderiza la mandíbula de Goddard.
250 
251         glUseProgram(0);
252         mainWindow.swapBuffers();
253     }
254 
255     return 0;
256 }
257 
```

Problemas presentados

No se presentaron problemas en este ejercicio.

Ejercicio 2: Agregar rotaciones a las patas de forma independiente para simular el avance de Goddard, limitar la rotación a 45° en cada sentido

Para este ejercicio implementó dentro del archivo Window.cpp el control de movimiento de diferentes partes de Goddard, mediante teclas específicas del teclado. Cada tecla corresponde a una acción de movimiento específica para una parte del modelo. Aquí está el detalle:

- Las teclas `T` y `F` controlan el movimiento de la pata delantera izquierda hacia adelante y hacia atrás, respectivamente.
- Las teclas `Y` y `G` controlan el movimiento de la pata delantera derecha hacia adelante y hacia atrás, respectivamente.
- Las teclas `U` y `H` controlan el movimiento de la pata trasera izquierda hacia adelante y hacia atrás, respectivamente.
- Las teclas `I` y `J` controlan el movimiento de la pata trasera derecha hacia adelante y hacia atrás, respectivamente.
- Las teclas `O` y `K` controlan el movimiento de la cabeza hacia arriba y hacia abajo, respectivamente.
- Las teclas `P` y `L` controlan el movimiento de la mandíbula hacia arriba y hacia abajo, respectivamente.

Para cada tecla presionada, se verifica si la articulación correspondiente del modelo tiene espacio para moverse en la dirección deseada. Si es así, se ajusta el ángulo de articulación en incrementos de 5 grados según la dirección de movimiento indicada.

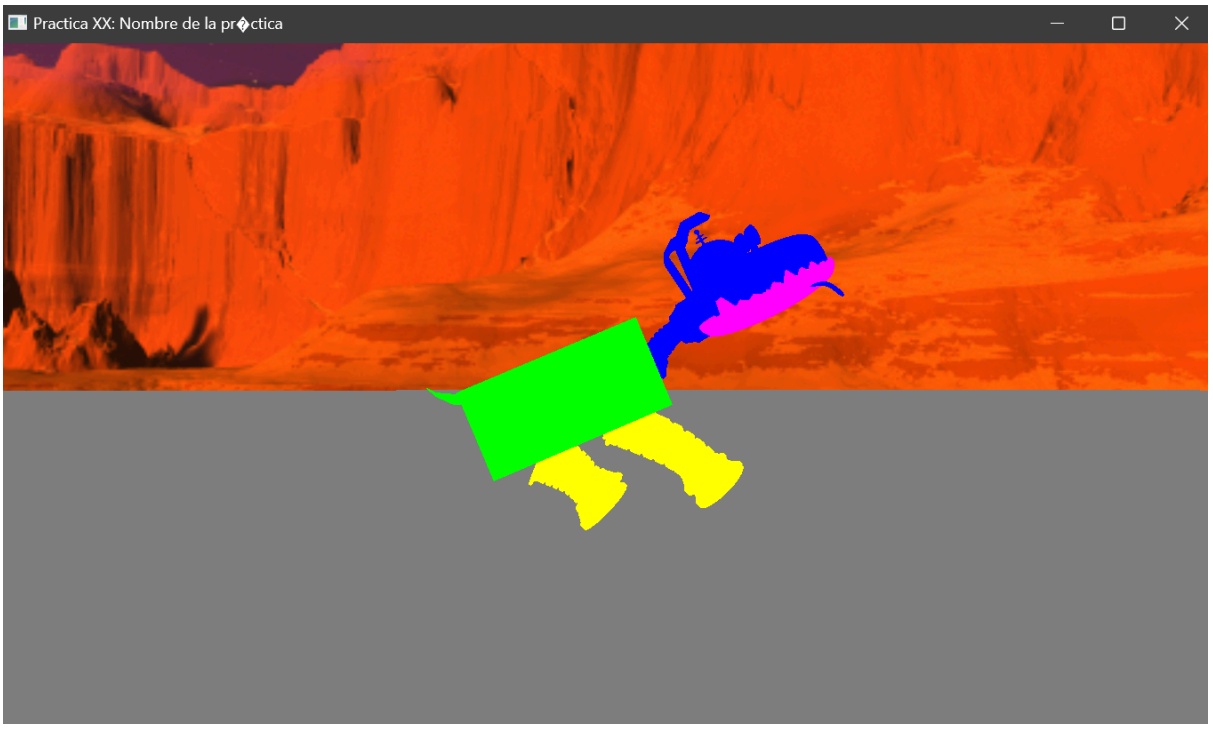
El límite de movimiento que se establece para cada articulación es de -10 grados a 45 grados. Es decir, las articulaciones pueden moverse en el rango de -10 grados a 45 grados. Si el ángulo de articulación alcanza estos límites, no se permite un movimiento adicional en esa dirección.

Problemas presentados

No se presentaron problemas en este ejercicio.

```
132     //Estableciendo los límites de rotación, asignando las teclas
133     if (key == GLFW_KEY_T)
134     {
135         if (theWindow->articulacion1 < 45.0) {
136             theWindow->articulacion1 += 5.0;    //pata delantera izquierda hacia adelante
137         }
138     }
139     if (key == GLFW_KEY_F)
140     {
141         if (theWindow->articulacion1 > -10.0) {
142             theWindow->articulacion1 -= 5.0;    //pata delantera izquierda hacia atrás
143         }
144     }
145     if (key == GLFW_KEY_Y)
146     {
147         if (theWindow->articulacion2 < 45.0) {
148             theWindow->articulacion2 += 5.0;    //pata delantera derecha hacia adelante
149         }
150     }
151     if (key == GLFW_KEY_G)
152     {
153         if (theWindow->articulacion2 > -10.0) {
154             theWindow->articulacion2 -= 5.0;    //pata delantera derecha hacia atrás
155         }
156     }
157     if (key == GLFW_KEY_U)
158     {
159         if (theWindow->articulacion3 < 45.0) {
160             theWindow->articulacion3 += 5.0;    //pata trasera izquierda hacia adelante
161         }
162     }
163     if (key == GLFW_KEY_H)
164     {
165         if (theWindow->articulacion3 > -10.0) {
166             theWindow->articulacion3 -= 5.0;    //pata trasera izquierda hacia atrás
167         }
168     }
169     if (key == GLFW_KEY_I)
170     {
171         if (theWindow->articulacion4 < 45.0) {
172             theWindow->articulacion4 += 5.0;    //pata trasera derecha hacia adelante
173         }
174     }
175     if (key == GLFW_KEY_J)
176     {
177         if (theWindow->articulacion4 > -10.0) {
178             theWindow->articulacion4 -= 5.0;    //pata trasera derecha hacia atrás
179         }
180     }
```

```
181     if (key == GLFW_KEY_O)
182     {
183         if (theWindow->articulacion5 < 30.0) {
184             theWindow->articulacion5 += 5.0;    //cabeza hacia arriba
185         }
186     }
187     if (key == GLFW_KEY_K)
188     {
189         if (theWindow->articulacion5 > -15.0) {
190             theWindow->articulacion5 -= 5.0;    //cabeza hacia abajo
191         }
192     }
193     if (key == GLFW_KEY_P)
194     {
195         if (theWindow->articulacion6 < 13.0) {
196             theWindow->articulacion6 += 5.0;    //mandíbula hacia arriba
197         }
198     }
199     if (key == GLFW_KEY_L)
200     {
201         if (theWindow->articulacion6 > -10.0) {
202             theWindow->articulacion6 -= 5.0;    //mandíbula hacia abajo
203         }
204     }
```



Conclusión

Puedo concluir que la comprensión y aplicación de conceptos de jerarquía en modelos 3D es fundamental para lograr animaciones realistas y dinámicas en entornos virtuales. Al establecer relaciones entre las diferentes partes de un modelo, como las extremidades y el cuerpo principal, podemos simular movimientos naturales y coordinados, lo que mejora significativamente la apariencia y la interactividad de la escena.

Además, el control de movimiento mediante teclas específicas del teclado, como se demostró en el código, permite una interacción más intuitiva y directa con los modelos 3D, lo que puede ser útil para aplicaciones de simulación, juegos y animación. La capacidad de establecer límites de movimiento asegura que las acciones del usuario se mantengan dentro de rangos realistas, lo que evita comportamientos incoherentes o antinaturales en la animación.