Reporte de Práctica de Laboratorio de Computación Gráfica e Interacción Humano-Computadora

Información de la práctica

Nombre del alumno:	Martínez Pérez Brian Erik		
Número de Cuenta:	319049792	Práctica número:	4
Título de la práctica:	Modelado Jerárquico en OpenGL y Cámara Sintética		
Fecha de entrega:	15 de septiembre de 2025		

Introducción: (descripción inicial sobre la práctica)

Esta práctica, titulada "Modelado Jerárquico en OpenGL y Cámara Sintética", se centra en la aplicación de los principios fundamentales de la computación gráfica 3D para la creación de un modelo articulado. La base de este trabajo es el uso de la biblioteca OpenGL, junto con la biblioteca GLM para el manejo de matrices y vectores. El objetivo principal es construir una estructura jerárquica, donde cada componente (como los segmentos de un brazo) se define en relación con su "padre" para que las transformaciones, como rotación y traslación, se propaguen de manera coherente a lo largo del modelo completo.

Para lograr esto, la práctica implementa conceptos clave del pipeline gráfico, incluyendo el uso de Vertex Array Objects (VAO) y Vertex Buffer Objects (VBO) para gestionar los datos de los vértices de los objetos, y el uso de shaders (vertex y fragment shaders) para controlar la representación visual de la geometría. La manipulación de los modelos y la cámara se realiza a través de matrices de transformación (modelo, vista y proyección), que permiten no solo la articulación del objeto sino también la simulación de una cámara artificial para explorar la escena desde diferentes perspectivas. Al final de la práctica, se obtiene un modelo 3D funcional, manipulable a través de entradas de teclado, que demuestra la aplicación del modelado jerárquico y el control de la cámara en un entorno de gráficos en tiempo real.

Resumen Teórico: (introducción sobre los fundamentos teóricos en los que se basa la práctica y cuáles son los objetivos teóricos que persigue.)

La práctica se basa en los fundamentos de la computación gráfica 3D en tiempo real, utilizando OpenGL como su principal biblioteca de programación. El objetivo es que el estudiante comprenda y aplique el pipeline gráfico de OpenGL, que es la secuencia de etapas que sigue la tarjeta gráfica (GPU) para convertir los datos de un modelo 3D en una imagen 2D que se muestra en pantalla.

Para lograr este fin, la práctica se enfoca en dos conceptos teóricos principales. El primero es el modelado jerárquico, una técnica esencial para la construcción de objetos complejos y articulados. En este enfoque, los objetos se definen como una estructura de "padre-hijo", donde las transformaciones de cada parte (rotación, traslación) dependen de las transformaciones de la parte anterior. Esto permite simular movimientos realistas y coordinados, como los de un brazo robótico, de forma más eficiente.

El segundo concepto es la cámara sintética, que simula la posición del observador en la escena 3D. Esto se logra mediante la matriz de vista, que define la posición de la cámara, y la matriz de proyección, que determina cómo se proyecta la escena 3D en el plano 2D de la pantalla. El código utiliza una proyección de perspectiva para crear una sensación de profundidad, haciendo que los objetos más lejanos se vean más pequeños.

En conjunto, los objetivos teóricos de la práctica son dominar el uso de la biblioteca GLM para el manejo de vectores y matrices de transformación, entender la función de los shaders en el pipeline programable y, finalmente, integrar todos estos conocimientos para crear una aplicación interactiva que represente un modelo articulado con una cámara manipulable.

Descripción de la práctica: (En esta sección se deben describir todos los pasos ejecutados durante la sesión práctica realizada en el laboratorio.)

El código de modelado jerárquico se encuentra en los archivos de origen, además el archivo “Shader.h” en el apartado de Archivos de encabezado, y en archivos de origen también colocamos la carpeta que contenía los archivos “core.frag”, “core.vs”.

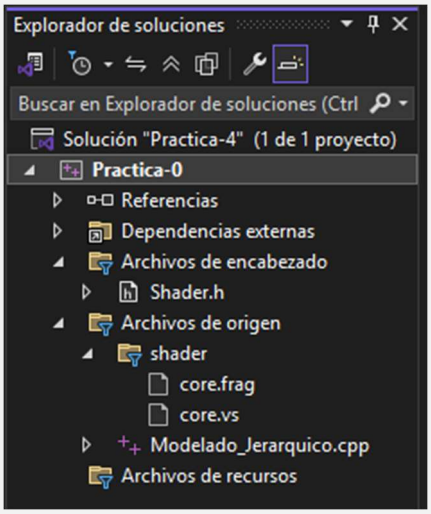


Imagen 1.1 – Ubicación de los archivos utilizados en la práctica.

Declaramos los pivotes utilizados, cada articulación tiene su ángulo de rotación, para mantener un movimiento natural en el modelado de la mano.

```
//For model
float   hombro = 0.0f;
float   codo = 0.0f;
float   muneca = 0.0f;
float   dedo1art1 = 0.0f;
float   dedo1art2 = 0.0f;
float   dedo2art1 = 0.0f;
float   dedo2art2 = 0.0f;
float   dedo3art1 = 0.0f;
float   dedo3art2 = 0.0f;
float   dedo4art1 = 0.0f;
float   dedo4art2 = 0.0f;
float   dedo5art1 = 0.0f;
float   dedo5art2 = 0.0f;
```

Imagen 1.2 – código para la declaración de los pivotes.

La matriz de vista representa la posición y la orientación de la cámara virtual en el mundo. Se utiliza para simular el movimiento de la cámara, las matrices temporales son fundamentales para guardar las transformaciones de las partes "padre" de un modelo. Esto permite que las partes "hijo" hereden las transformaciones de sus padres.

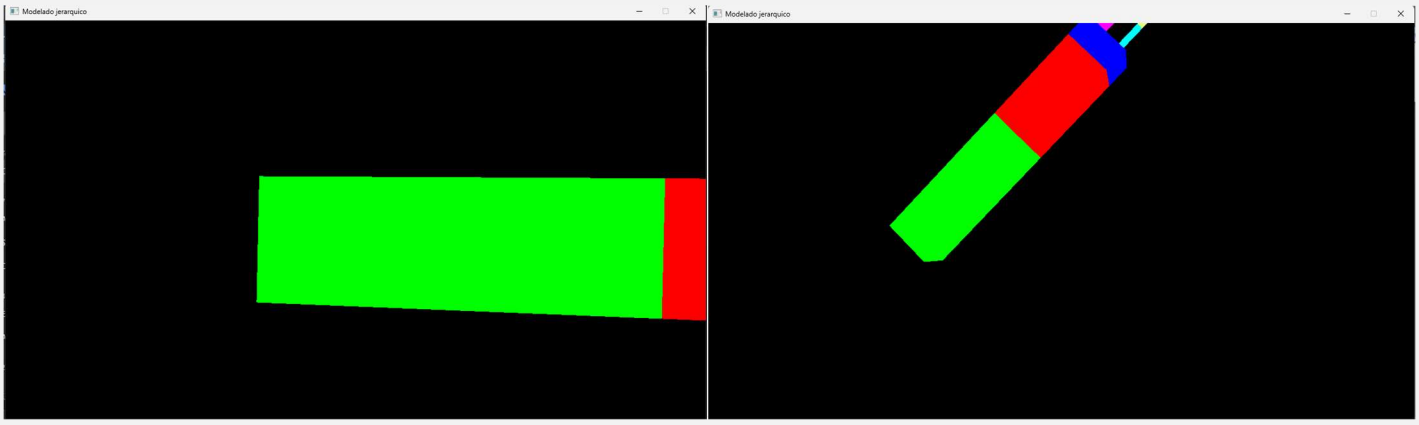
```
ourShader.Use();
glm::mat4 model=glm::mat4(1);
glm::mat4 view=glm::mat4(1);
glm::mat4 modelTemp = glm::mat4(1.0f); //Temp
glm::mat4 modelTemp2 = glm::mat4(1.0f); //Temp
```

Imagen 1.3 – matrices utilizadas para la manipulación de nuestros modelos.

En la creación de nuestro primer objeto, el cual será el padre de todos los demás objetos que generemos dentro del código, las características que tendrá es que rota en el eje Z, traslada 1.5 unidades en X, escala a (3,1,1) y su color verde. Dentro de esta practica notamos que el color lo agregamos con la función de la librería GLM, y ya no directamente en la definición de los vértices.

```
//Model hombro
model = glm::rotate(model, glm::radians(hombro), glm::vec3(0.0f, 0.0, 1.0f));
modelTemp = model = glm::translate(model, glm::vec3(1.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(3.0f, 1.0f, 1.0f));
color = glm::vec3(0.0f, 1.0f, 0.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 2.1 – código para generar el hombro.



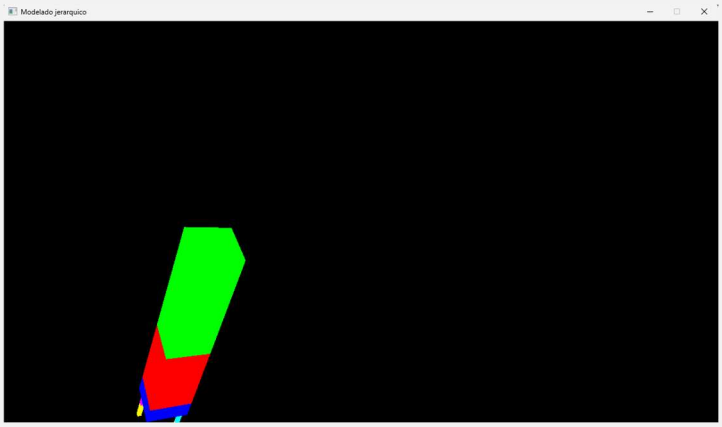
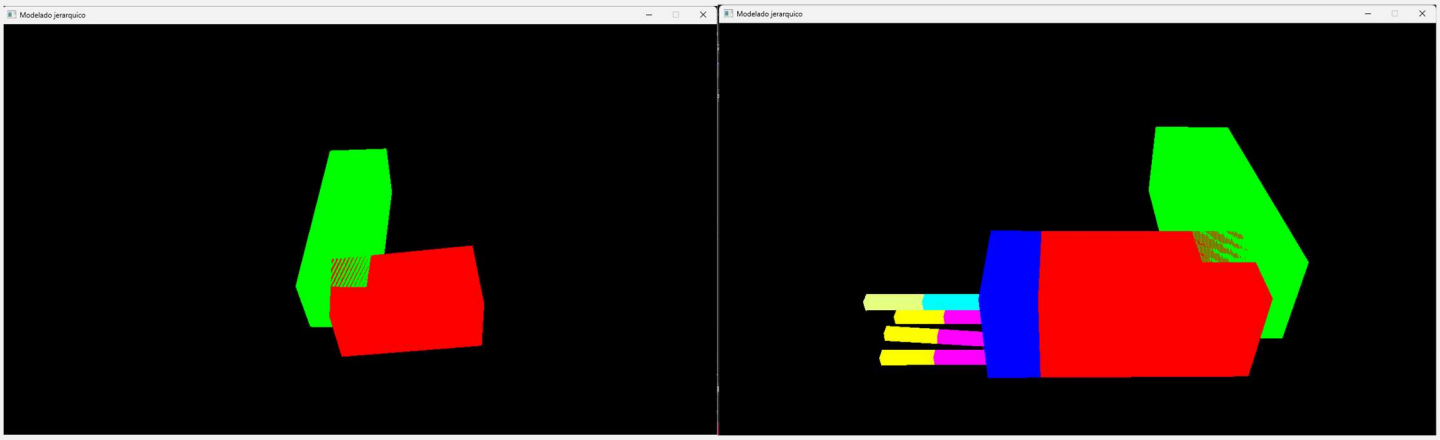


Imagen 2.2 – capturas de la creación del hombro.

Aplicando el concepto de modelado jerárquico, creamos el “codo” de nuestro brazo, el cual será el hijo de “hombro”. Para diferenciarlos colocamos un traslada desde el hombro, rotación en el eje Y, escala a (2,1,1) y un color rojo.

```
//Model codo
model = glm::translate(modelTemp, glm::vec3(1.5f, 0.0f, 0.0f));
model = glm::rotate(model, glm::radians(codo), glm::vec3(0.0f, 1.0, 0.0f));
modelTemp = model = glm::translate(model, glm::vec3(1.0f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(2.0f, 1.0f, 1.0f));
color = glm::vec3(1.0f, 0.0f, 0.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 3.1 – Código para la generación del codo.



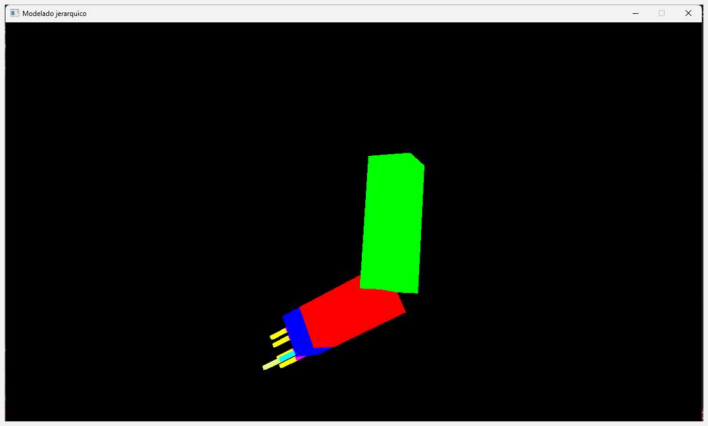
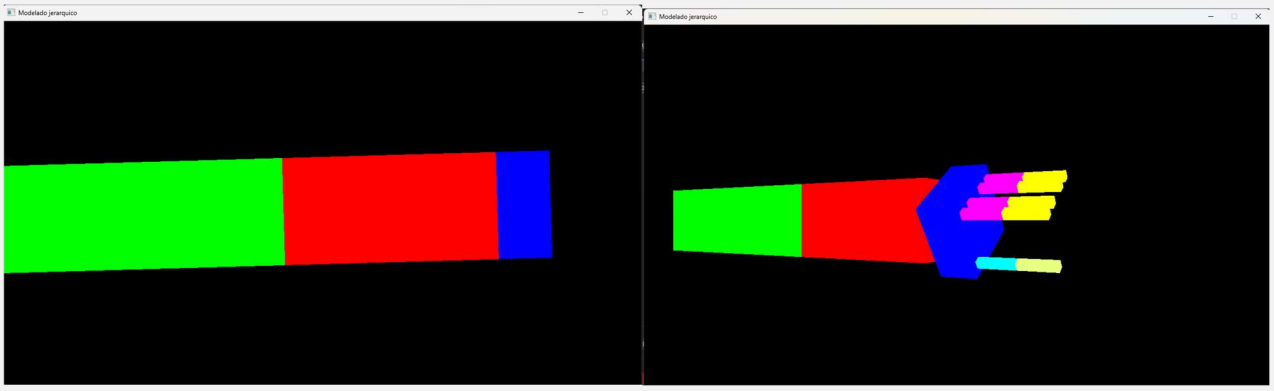


Imagen 3.2 – capturas del objeto “codo” generado.

Ya tenemos el hombro y el codo, seguiría crear el movimiento de la muñeca, respetando la jerarquía de modelado, “muñeca” seria el hijo de “codo”. Dando las propiedades de Traslado desde el codo, rotación en el eje X, escala a (0.5,1,1) y color azul.

```
//Model muneca
model = glm::translate(modelTemp, glm::vec3(1.0f, 0.0f, 0.0f));
model = glm::rotate(model, glm::radians(muneca), glm::vec3(1.0f, 0.0, 0.0f));
modelTemp = model = glm::translate(model, glm::vec3(0.25f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 1.0f, 1.0f));
color = glm::vec3(0.0f, 0.0f, 1.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 4.1 – Código generado para crear la “muñeca”.



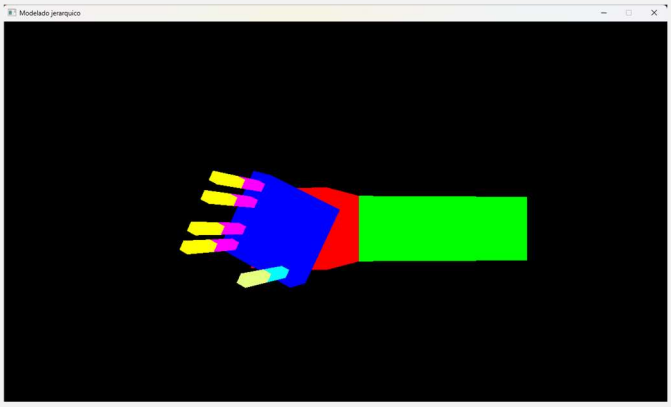


Imagen 4.2 – capturas del objeto “muñeca”.

En el último nivel de nuestra jerarquía vamos a tener a los dedos de la mano, cada dedo creado y las 2 articulaciones que se divide, serán hijos de “muñeca”. De manera general la posición relativa, se traslada desde la muñeca en Y/Z. Se rota en Z, se traslada y escala a un tamaño pequeño. La elección es de colores distintos para cada segmento.

```
//Model dedo 1 art1
model = glm::translate(modelTemp, glm::vec3(0.0f, 0.35f, 0.375f));
model = glm::rotate(model, glm::radians(dedo1art1), glm::vec3(0.0f, 0.0, 1.0f));
modelTemp2 = model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 0.0f, 1.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

//Model dedo 1 art2
model = glm::translate(modelTemp2, glm::vec3(0.0f, 0.0f, 0.0f));
model = glm::rotate(model, glm::radians(dedo1art2), glm::vec3(0.0f, 0.0, 1.0f));
model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 1.0f, 0.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 5.1 – Código para la generación del dedo 1

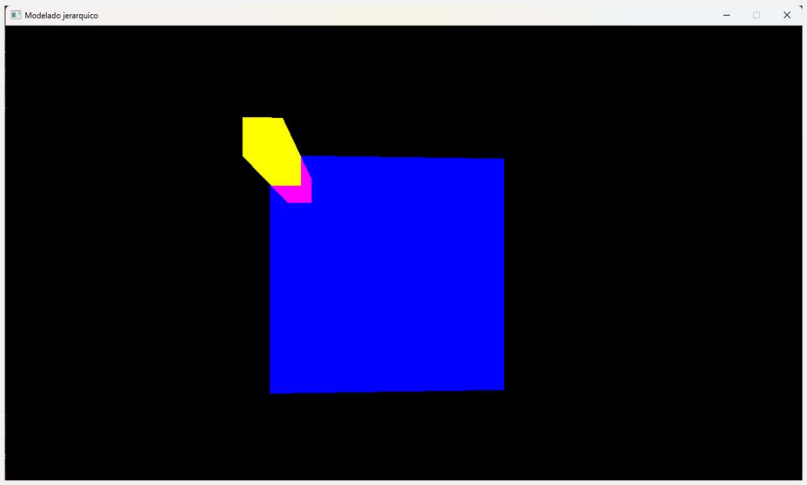


Imagen 5.2 – captura del objeto dedo 1 generado.

```
//Model dedo 2 art1
model = glm::translate(modelTemp, glm::vec3(0.0f, 0.35f, 0.17f));
model = glm::rotate(model, glm::radians(dedo2art1), glm::vec3(0.0f, 0.0, 1.0f));
modelTemp2 = model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 0.0f, 1.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

//Model dedo 2 art2
model = glm::translate(modelTemp2, glm::vec3(0.0f, 0.0f, 0.0f));
model = glm::rotate(model, glm::radians(dedo2art2), glm::vec3(0.0f, 0.0, 1.0f));
model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 1.0f, 0.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 5.3 – Código para la generación del dedo 2

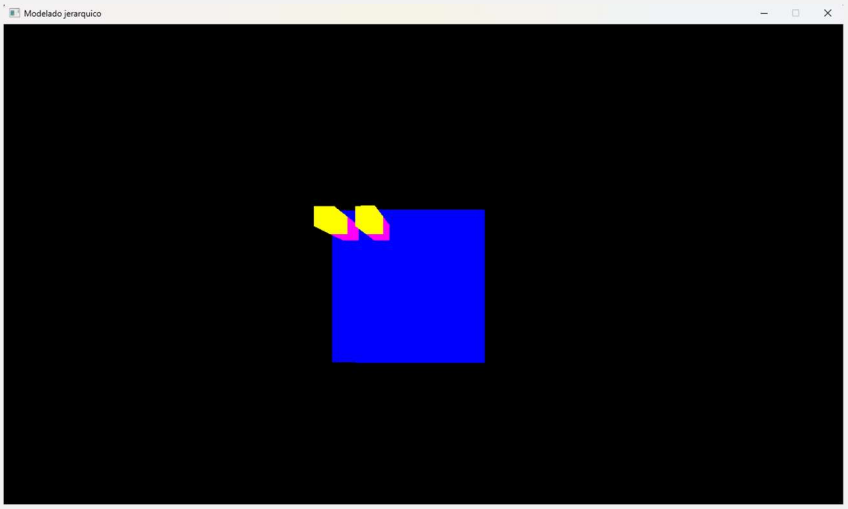


Imagen 5.4 – captura del objeto dedo 2 generado.

```
//Model dedo 3 art1
model = glm::translate(modelTemp, glm::vec3(0.0f, 0.35f, -0.17f));
model = glm::rotate(model, glm::radians(dedo3art1), glm::vec3(0.0f, 0.0, 1.0f));
modelTemp2 = model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 0.0f, 1.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

//Model dedo 3 art2
model = glm::translate(modelTemp2, glm::vec3(0.0f, 0.0f, 0.0f));
model = glm::rotate(model, glm::radians(dedo3art2), glm::vec3(0.0f, 0.0, 1.0f));
model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 1.0f, 0.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 5.5 – Código para la generación del dedo 3

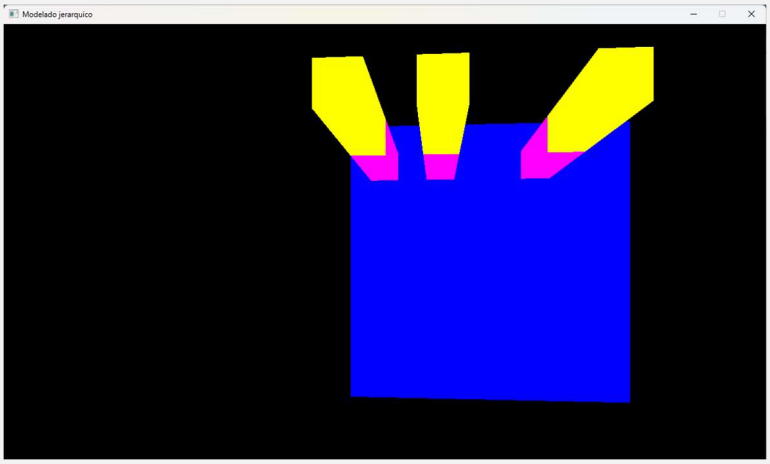


Imagen 5.6 – captura del objeto dedo 3 generado.

```
//Model dedo 4 art1
model = glm::translate(modelTemp, glm::vec3(0.0f, 0.35f, -0.375f));
model = glm::rotate(model, glm::radians(dedo4art1), glm::vec3(0.0f, 0.0, 1.0f));
modelTemp2 = model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 0.0f, 1.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

//Model dedo 4 art2
model = glm::translate(modelTemp2, glm::vec3(0.0f, 0.0f, 0.0f));
model = glm::rotate(model, glm::radians(dedo4art2), glm::vec3(0.0f, 0.0, 1.0f));
model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(1.0f, 1.0f, 0.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 5.7 – Código para la generación del dedo 4.

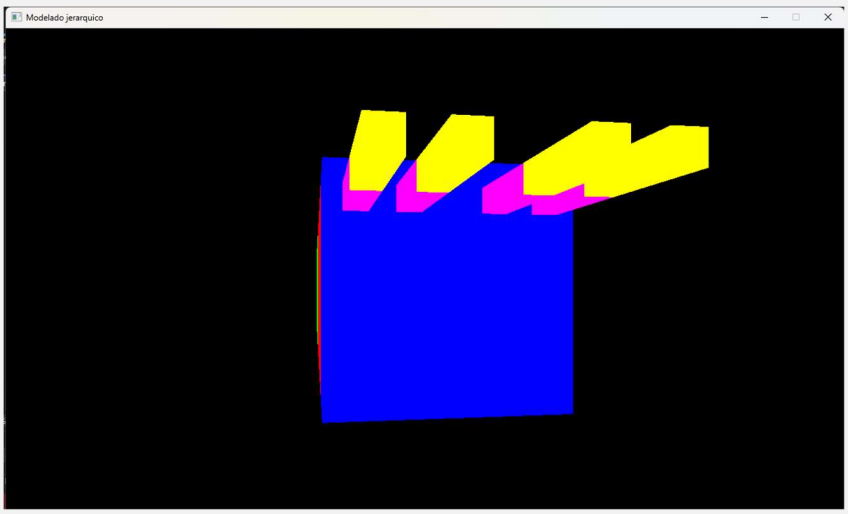


Imagen 5.8 – captura del objeto dedo 4 generado.

```
//Model dedo 5 art1
model = glm::translate(modelTemp, glm::vec3(0.0f, -0.35f, 0.375f));
model = glm::rotate(model, glm::radians(dedo5art1), glm::vec3(0.0f, 1.0, 0.0f));
modelTemp2 = model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(0.0f, 1.0f, 1.0f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

//Model dedo 5 art2
model = glm::translate(modelTemp2, glm::vec3(0.0f, 0.0f, 0.0f));
model = glm::rotate(model, glm::radians(dedo5art2), glm::vec3(0.0f, 1.0, 0.0f));
model = glm::translate(model, glm::vec3(0.5f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f, 0.1f, 0.1f));
color = glm::vec3(.9f, 1.0f, 0.5f);
glUniform3fv(uniformColor, 1, glm::value_ptr(color));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Imagen 5.9 – Código para la generación del dedo 5

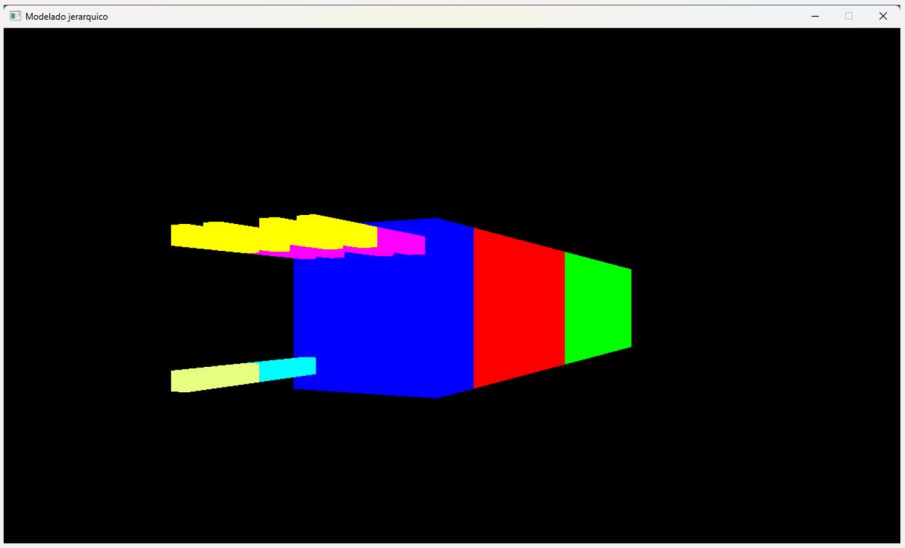


Imagen 5.10 – captura del objeto dedo 5 generado.

Agregamos dos articulaciones por cada dedo, por lo que el movimiento de cada articulación lo podemos realizar con la tecla de predefinida en el código.

```
void Inputs(GLFWwindow* window) {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)  //GLFW_RELEASE
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        movX += 0.008f;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        movX -= 0.008f;
    if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
        movY += 0.008f;
    if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
        movY -= 0.008f;
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        movZ -= 0.08f;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        movZ += 0.08f;
    if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS)
        rot += 0.18f;
    if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS)
        rot -= 0.18f;
    if (glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS)
        hombro += 0.18f;
    if (glfwGetKey(window, GLFW_KEY_F) == GLFW_PRESS)
        hombro -= 0.18f;
    if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS)
        codo += 0.18f;
    if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS)
        codo -= 0.18f;
    if (glfwGetKey(window, GLFW_KEY_Y) == GLFW_PRESS)
        muneca += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_H) == GLFW_PRESS)
        muneca -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_U) == GLFW_PRESS)
        dedo1art1 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS)
        dedo1art1 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_I) == GLFW_PRESS)
```

```
        dedo1art1 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS)
        dedo1art1 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_I) == GLFW_PRESS)
        dedo1art2 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_K) == GLFW_PRESS)
        dedo1art2 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_O) == GLFW_PRESS)
        dedo2art1 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS)
        dedo2art1 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS)
        dedo2art2 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_X) == GLFW_PRESS)
        dedo2art2 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS)
        dedo3art1 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_V) == GLFW_PRESS)
        dedo3art1 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS)
        dedo3art2 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_N) == GLFW_PRESS)
        dedo3art2 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS)
        dedo4art1 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_2) == GLFW_PRESS)
        dedo4art1 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_3) == GLFW_PRESS)
        dedo4art2 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_4) == GLFW_PRESS)
        dedo4art2 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_5) == GLFW_PRESS)
        dedoSart1 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_6) == GLFW_PRESS)
        dedoSart1 -= 0.018f;
    if (glfwGetKey(window, GLFW_KEY_7) == GLFW_PRESS)
        dedoSart2 += 0.018f;
    if (glfwGetKey(window, GLFW_KEY_8) == GLFW_PRESS)
        dedoSart2 -= 0.018f;
}
```

Imagen 6.1 – Código utilizado para las entradas de teclado.

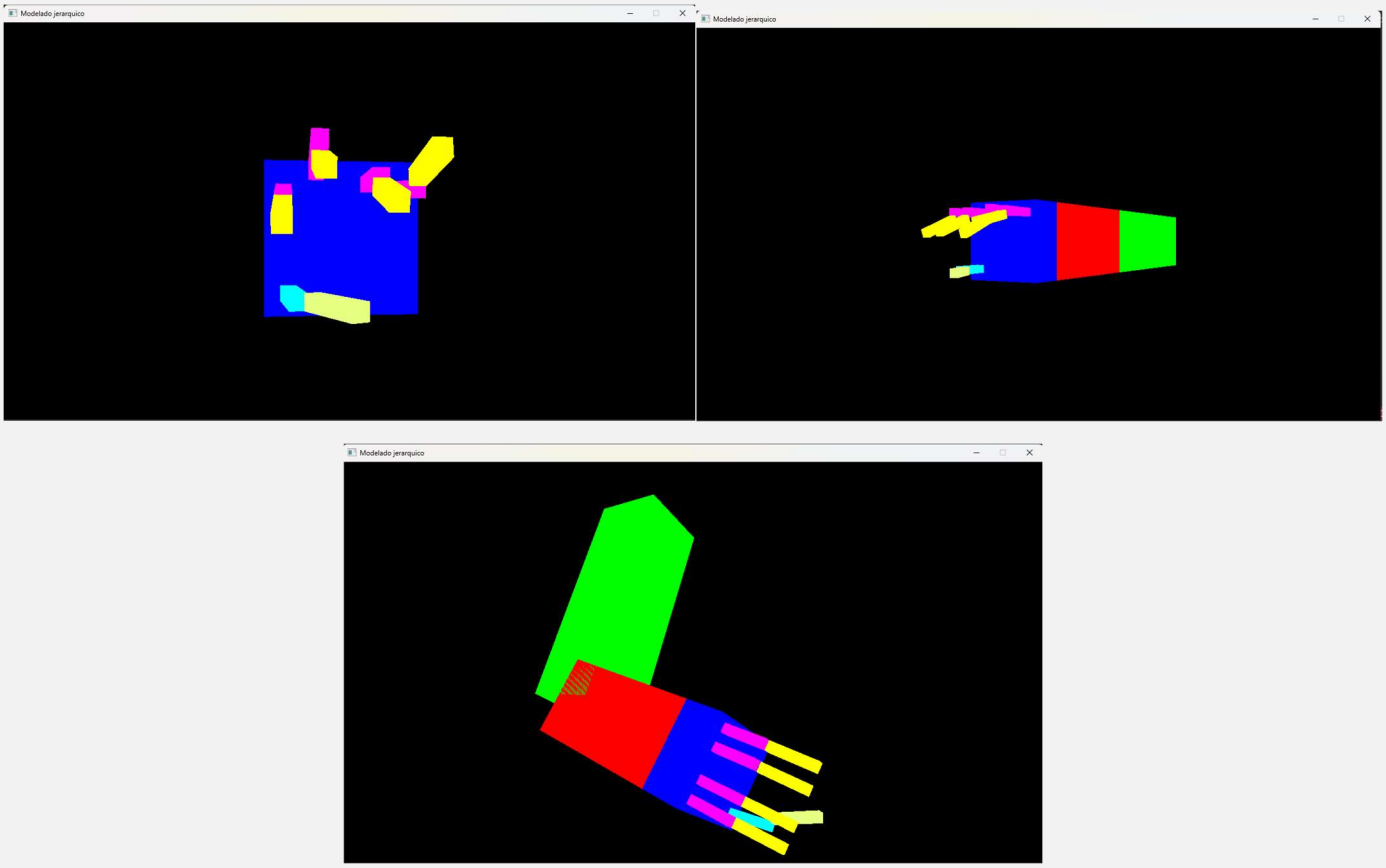


Imagen 6.2 – capturas del movimiento independiente de cada uno de los dedos.

La jerarquía de objetos, quedo de la siguiente manera:

Hombro (Verde)
└── Codo (Rojo)
    └── Muñeca (Azul)
        ├── Dedo1 (Magenta/Amarillo)
        ├── Dedo2 (Magenta/Amarillo)
        ├── Dedo3 (Magenta/Amarillo)
        ├── Dedo4 (Magenta/Amarillo)
        └── Dedo5 (Cyan/Amarillo claro)

Resultados: (explicar lo que se logró o aquello que no lograron realizar o comprender)

El resultado de esta práctica fue la implementación de un modelo jerárquico articulado, en este caso un brazo robótico, que se compone de múltiples partes (hombro, codo, muñeca y dedos) que se mueven de manera interdependiente. El código C++ creado cumple con el dominio de las matrices de transformación (traslación, rotación y escalado) para posicionar y orientar cada segmento del modelo con precisión.

Además del modelado, se pudo simular con éxito una cámara sintética usando la matriz de vista, lo que permitió al usuario manipular la perspectiva y el punto de vista de la escena en tiempo real a través de las entradas del teclado. Esto, combinado con una proyección de perspectiva, crea un entorno 3D convincente. El programa también demostró un manejo correcto de los shaders y los buffers (VBO y VAO) para renderizar eficientemente la geometría en la GPU. El código muestra la aplicación de los conocimientos previos descritos en el documento de la práctica, logrando así todos los objetivos teóricos propuestos. No se encontraron dificultades en la implementación, por lo que los conceptos principales se comprendieron y aplicaron de manera efectiva.

Conclusiones:

A través de esta práctica, hemos logrado aplicar los principios fundamentales del modelado jerárquico para crear y animar un brazo robótico en un entorno 3D. Las matrices de transformación fueron esenciales para establecer las relaciones de "padre-hijo" entre las diferentes partes del brazo, asegurando que cada segmento se moviera de manera coherente en relación con la articulación anterior. Este enfoque nos permitió controlar movimientos complejos, como la flexión del codo y la rotación de la muñeca, simplemente manipulando unas pocas variables. Adicionalmente, la implementación de una cámara sintética proporcionó la flexibilidad necesaria para visualizar el modelo desde múltiples ángulos y distancias. En su conjunto, esta práctica no solo demostró el poder de las matrices de transformación y los shaders en OpenGL, sino que también solidificó la comprensión del pipeline gráfico y la importancia del modelado jerárquico para la creación de modelos 3D articulados y dinámicos.

Bibliografía consultada:

Khronos Group. (2023). OpenGL Reference Pages. https://www.opengl.org/documentation/

Joey de Vries. (2023). LearnOpenGL: Model loading. https://learnopengl.com/Model-Loading/Model-Loading

GLM. (2023). OpenGL Mathematics (GLM) documentation. https://glm.g-truc.net/0.9.9/index.html