Ciencia de la Computación

Computación Gráfica

Docente Manuel Eduardo Loaiza Fernandez

Proyecto Final

Entregado el 01/07/2024

Mauricio Carazas Segovia
mauricio.carazas@ucsp.edu.pe

Semestre VII

2024-1

"Los alumnos declaran haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo"

_____ _____

Proyecto Final de Computación Gráfica

Introducción y Motivación del Proyecto

El presente proyecto de computación gráfica tiene como objetivo explorar y aplicar conceptos avanzados de gráficos por computadora a través de la implementación de dos proyectos distintos: un cubo de Rubik con un solucionador y una representación 2D del cubo de Rubik en unas esferas ubicadas en anillos que representan el movimiento de las caras del cubo. La motivación principal detrás de este proyecto es profundizar en el conocimiento de técnicas de renderizado, manejo de shaders, y animación, así como adquirir experiencia práctica en la programación de gráficos en 3D usando OpenGL. Este proyecto combina la complejidad matemática con la creatividad visual, permitiendo una inmersión completa en el campo de los gráficos computacionales.

Descripción del Proyecto Final de Programación

Herramientas

Para el desarrollo del proyecto se utilizaron las siguientes herramientas:

- **Lenguaje de programación**: C++ es un lenguaje de programación de propósito general conocido por su rendimiento y control sobre los recursos del hardware.
- **API gráfica**: OpenGL es una API estándar de la industria para gráficos 2D y 3D, utilizada para interactuar con unidades de procesamiento gráfico (GPU) para lograr renderizado de gráficos de alta eficiencia.
- **Entorno de desarrollo**: Visual Studio Code es un editor de código fuente ligero y potente, con soporte para depuración, control de versiones, y una rica colección de extensiones.
- **Librerías adicionales**:
    - **GLFW**: una biblioteca que permite la creación de ventanas y el manejo de eventos de entrada.
    - **GLAD**: una biblioteca que carga punteros a funciones de OpenGL.
    - **GLM**: una biblioteca matemática para gráficos que facilita las operaciones con vectores y matrices.

Matemática

El proyecto involucra diversas operaciones matemáticas, tales como transformaciones geométricas (traslaciones, rotaciones y escalados), cálculos de normales para iluminación, y la utilización de matrices de proyección y vista para la cámara. Además, se aplicaron algoritmos de álgebra lineal para manipular las posiciones y orientaciones de los objetos en el espacio tridimensional.

Transformaciones Geométricas

Las transformaciones geométricas son esenciales para posicionar y orientar los objetos en la escena 3D. Las transformaciones básicas incluyen:

- **Traslación**: Desplaza un objeto a lo largo de los ejes X, Y o Z.
- **Rotación**: Gira un objeto alrededor de un eje específico.
- **Escalado**: Cambia el tamaño de un objeto en una o más direcciones.

El siguiente ejemplo muestra cómo se define una matriz de rotación en OpenGL:

```
void convert_matrix_rotation_y(float angle){
    convert_matrix_identity(4);
    (*data)[0][0]=cos(angle * PI / 180.0);
    (*data)[0][2]=sin(angle * PI / 180.0);
    (*data)[2][0]=-sin(angle * PI / 180.0);
    (*data)[2][2]=cos(angle * PI / 180.0);
}
```

Animación

La animación de la representación 2D del cubo de Rubik en el video muestra cómo las caras del cubo se representan mediante esferas dispuestas en anillos. Estas esferas giran y se desplazan de manera que simulan los movimientos de las capas del cubo de Rubik. Cada movimiento de las capas del cubo se refleja en la rotación y traslación de las esferas en los anillos correspondientes.

Descripción Detallada

La animación se compone de varios elementos clave:

1. **Esferas en Anillos**: Las esferas están organizadas en anillos concéntricos, cada uno representando una capa del cubo de Rubik. Cada anillo contiene un conjunto de esferas de diferentes colores, correspondientes a las piezas del cubo.
2. **Rotación de Anillos:** Los anillos de esferas giran alrededor de un punto central, simulando las rotaciones de las capas del cubo de Rubik. Esta rotación se realiza en el plano 2D, y cada anillo puede girar independientemente.
3. **Movimiento Sincrónico:** Las esferas en los anillos se mueven de manera sincronizada para reflejar los movimientos de las capas del cubo. Esto incluye tanto las rotaciones horizontales como verticales.
4. **Interacción del Usuario:** La animación permite la interacción del usuario mediante el teclado, donde se pueden activar los movimientos de las capas del cubo, y estos se reflejan en los movimientos de las esferas.

Implementación del Código

La implementación del código para la animación de las esferas en anillos se realiza a través de la clase CuboEsferas. A continuación se describe cómo se implementa esta funcionalidad en el código:

Clase CuboEsferas

La clase CuboEsferas es responsable de generar y manejar las esferas que representan las caras del cubo de Rubik en 2D. Utiliza una serie de funciones para calcular las posiciones y rotaciones de las esferas.

```
struct CuboEsferas {
    std::vector<Esfera> esferas;
    Vertex center;
    float angle;

    CuboEsferas(Vertex c, float r, float g, float b, float angleDeg)
: center(c), angle(angleDeg) {
        generateCuboEsferas(r, g, b);
    }

    void generateCuboEsferas(float r, float g, float b) {
        float radius = 0.02f;
        float angleRad = angle * M_PI / 180.0f;

        auto rotate = [&](float x, float y) {
            float s = sin(angleRad);
            float c = cos(angleRad);
            float xnew = c * (x - center.x) - s * (y - center.y) +
center.x;
            float ynew = s * (x - center.x) + c * (y - center.y) +
center.y;
            return std::make_pair(xnew, ynew);
        };

        esferas.push_back(Esfera(radius, r, g, b, center.x,
center.y, center.z));

        float spaceY = 0.12;
        auto [x, y] = rotate(center.x, center.y + spaceY);
        esferas.push_back(Esfera(radius, r, g, b, x, y, center.z));
        std::tie(x, y) = rotate(center.x, center.y - spaceY);
        esferas.push_back(Esfera(radius, r, g, b, x, y, center.z));

        float spaceX = 0.18;
        std::tie(x, y) = rotate(center.x + spaceX, center.y - 0.02);
        esferas.push_back(Esfera(radius, r, g, b, x, y, center.z));
```

```
        std::tie(x, y) = rotate(center.x - spaceX, center.y - 0.02);
        esferas.push_back(Esfera(radius, r, g, b, x, y, center.z));
    }
};
```

Grafo de Escena

El grafo de escena utilizado en el proyecto organiza los objetos en una jerarquía, permitiendo la aplicación de transformaciones de manera estructurada. Cada nodo del grafo representa un objeto o una transformación, facilitando la gestión y renderizado de escenas complejas. Este enfoque es crucial para manejar la complejidad del cubo de Rubik y la representación de las esferas en anillos.

Para mejorar la visualización y permitir la observación simultánea del cubo de Rubik y su representación 2D, se utiliza la técnica de `viewport` para dividir la pantalla en dos mitades. La implementación del viewport en OpenGL se realiza mediante la función `glViewport`, que define la parte de la ventana donde se renderiza la escena. La primera mitad de la pantalla muestra el cubo de Rubik en 3D, mientras que la segunda mitad muestra los anillos y esferas que representan las caras del cubo en 2D. Este método permite que ambos componentes de la animación sean visibles al mismo tiempo, proporcionando una visualización completa y educativa de los movimientos y transformaciones.

Shaders

Se desarrollaron shaders personalizados para el proyecto, incluyendo vertex y fragment shaders. Estos shaders permiten realizar cálculos de iluminación, aplicando modelos como Phong y Blinn-Phong para simular efectos realistas de luz y sombra sobre los objetos. Los shaders también manejan el mapeo de texturas para aplicar colores y texturas a las superficies de los objetos.

```
const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "layout (location = 1) in vec3 aColor;\n"
    "layout (location = 2) in vec2 aTexCoord;\n"
    "out vec3 ourColor;\n"
    "out vec2 TexCoord;\n"
    "uniform mat4 view;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = view * vec4(aPos, 1.0);\n"
    "   ourColor = aColor;\n"
    "   TexCoord = aTexCoord;\n"
    "}\0";
```

```
const char *fragmentShaderSource = "#version 330 core\n"
    "out vec4 FragColor;\n"
    "in vec3 ourColor;\n"
    "in vec2 TexCoord;\n"
    "uniform sampler2D texture1;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(ourColor, 1.0f) * texture(texture1,
TexCoord);\n"
    "}\n\0";
```

Funcionalidades de su Programa

Keyboard

El programa permite la interacción a través del teclado, ofreciendo funcionalidades como:

- Rotación y traslación de la cámara.
- Movimientos y giros del cubo de Rubik.
- Control de la animación de las esferas.

Guía de Funcionamiento

Las teclas asignadas para controlar el cubo de Rubik y la cámara son las siguientes:

- **Rotación y Traslación de la Cámara**:
    - **W**: Mover la cámara hacia adelante.
    - **S**: Mover la cámara hacia atrás.
    - **A**: Mover la cámara hacia la izquierda.
    - **D**: Mover la cámara hacia la derecha.
    - **Flecha Arriba**: Inclinar la cámara hacia abajo.
    - **Flecha Abajo**: Inclinar la cámara hacia arriba.
    - **Flecha Izquierda**: Girar la cámara hacia la izquierda.
    - **Flecha Derecha**: Girar la cámara hacia la derecha.
- **Movimientos y Giros del Cubo de Rubik**:
    - **R**: Rotar la capa derecha del cubo.
    - **L**: Rotar la capa izquierda del cubo.
    - **U**: Rotar la capa superior del cubo.
    - **D**: Rotar la capa inferior del cubo.
    - **F**: Rotar la capa frontal del cubo.
    - **B**: Rotar la capa trasera del cubo.
- **Control de la Animación de las Esferas**: La animación de las esferas se controla de manera sincronizada con los movimientos del cubo de Rubik, reflejando visualmente las rotaciones y giros de las capas del cubo.

Movimiento de Cámara

La cámara se puede mover libremente en el espacio 3D utilizando las teclas WASD para traslación y el mouse para rotación, proporcionando una vista interactiva y dinámica de la escena. A continuación se detalla cómo se controla la cámara:

- **W**: Mover la cámara hacia adelante.
- **S**: Mover la cámara hacia atrás.
- **A**: Mover la cámara hacia la izquierda.
- **D**: Mover la cámara hacia la derecha.
- **Mouse**: Rotar la cámara en cualquier dirección.

Iluminación

El sistema de iluminación incluye múltiples fuentes de luz, como luces direccionales, puntuales y spotlights. Los shaders calculan la iluminación en tiempo real, proporcionando efectos de sombreado y reflexión precisos.

```
const char *fragmentShaderSource = "#version 330 core\n"
    "out vec4 FragColor;\n"
    "in vec3 ourColor;\n"
    "in vec2 TexCoord;\n"
    "uniform sampler2D texture1;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(ourColor, 1.0f) * texture(texture1,
TexCoord);\n"
    "}\n\0";
```

Experimentos y Resultados

Se realizaron varios experimentos para evaluar el rendimiento y la calidad visual del proyecto. Entre los resultados destacables se encuentran:

- La correcta implementación de la solución del cubo de Rubik.
- La animación fluida de las esferas representando las caras del cubo.
- La iluminación realista aplicada a ambas escenas.

Los experimentos incluyeron la prueba de diferentes ángulos y velocidades de rotación para las esferas y el cubo de Rubik, así como la evaluación de la interacción del usuario mediante el teclado y el mouse.

Problemas Encontrados en la Implementación

Durante la implementación, se encontraron varios problemas, tales como:

1. **Integración de Ambos Proyectos**: Aunque ambos proyectos, el cubo de Rubik en 3D y la representación 2D con esferas, funcionaban bien de manera individual, integrar ambos en un solo programa presentó varios problemas. La mezcla de las dos animaciones en una sola aplicación causó conflictos en la gestión de recursos y en la ejecución simultánea de los gráficos.
2. **Problemas de Iluminación**: Al integrar ambos proyectos, surgieron problemas de iluminación que resultaron en que algunas escenas no se renderizaran correctamente. La configuración de las fuentes de luz y los shaders no se comportaba como se esperaba cuando ambas escenas se mostraban simultáneamente. Esto causó que en algunos casos no se graficara nada en la pantalla, requiriendo una revisión y ajuste de los parámetros de iluminación y los shaders utilizados.
3. **Sincronización de las Animaciones**: Lograr una transición suave entre los movimientos del cubo de Rubik y las rotaciones de las esferas en la representación 2D requirió ajustar cuidadosamente las velocidades de animación y los intervalos de tiempo. Asegurar que ambas animaciones estuvieran sincronizadas fue un desafío considerable.

Conclusiones

El proyecto permitió aplicar y consolidar conocimientos avanzados en computación gráfica, resultando en la creación de dos escenas complejas y visualmente atractivas. A través de este proyecto, se adquirió una comprensión más profunda de la programación de shaders, manejo de cámaras y animación de objetos en 3D. La implementación del cubo de Rubik y su representación 2D en esferas proporcionó una experiencia educativa enriquecedora, combinando teoría matemática con aplicaciones prácticas en gráficos por computadora.

Próximos Pasos

Los próximos pasos para mejorar el proyecto incluyen:

- Implementar más efectos de iluminación y sombreado, como sombras dinámicas y reflejos.
- Añadir más opciones de interacción para el usuario, incluyendo controles más precisos y modos de visualización alternativos.

Referencias

- LearnOpenGL
- OpenGL Programming Guide
- GLFW Documentation
- GLM Mathematics Library
- C++ Reference