UNIVERSIDAD DE DARMSTADT
B. FRÖMMER, E. HERGENRÖTHER, B. MEYER

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

3 DE MAYO
DE 2023

VISUAL INFORMÁTICA SOSE 2023

TAREA 2

2.1 Hacia la tercera dimensión

En Moodle encontrarás un archivo `Cube.h`, que deberás integrar en tu proyecto. Sólo contiene nuevos datos de vértices e índices para mostrar un cubo 3D coloreado.
Utilice esto para reemplazar su antigua geometría (ya sea la casa del 1er proyecto práctico, o sus iniciales hechas por usted mismo). Tienes que ajustar tus punteros de atributos de vértices en consecuencia para que la geometría tridimensional se muestre correctamente. También tienes que actualizar el sombreador de vértices (`vertex.glsl`) para que se procesen las tres coordenadas de la posición del vértice. Encontrarás los shaders en la carpeta assets/shaders. Sin embargo, los shaders adaptados sólo se toman (y se copian en el directorio de construcción) si se recarga el proyecto CMake (en CLion: Herramientas/CMake/Reload).
Si todo funciona, el cubo se renderiza en el centro de la ventana, pero frontalmente desde un lado, por lo que aún no es realmente 3D.

2.2 Transformaciones

Para obtener una primera impresión tridimensional del cubo, tenemos que rotarlo un poco. Como debería saberse por la clase, todas las transformaciones de un objeto se almacenan en una matriz de transformación acumulada. Nuestro framework proporciona la clase `Transform para` este propósito, que gestiona una matriz correspondiente y ya ha implementado varios métodos de transformación. La clase se puede encontrar en la subcarpeta src/Framework/SceneElements y le gustaría ser incluida una vez en el `Scene.h`.
Por lo tanto, crea un objeto de transformación para tu cubo y rótalo alrededor de dos ejes. El método `rotate` espera un cuaternión para la rotación (difícil), o un vector tridimensional (`glm::vec3(float x, float y, float z)`) con tres ángulos de Euler en radianes alrededor de los cuales rotar en los ejes X, Y y Z (que luego se transforma internamente en un cuaternión).
Puedes rotar tu cubo una vez en `Scene::init()`, o por fotograma en `Scene::render()`. El parámetro `float dt` ayuda a definir la velocidad de rotación independientemente de la tasa de fotogramas respectiva.

Sin embargo, la definición de la transformación es sólo el primer paso. Todavía tiene que transferirse al sombreador para poder aplicarse allí a los vértices individuales.
El framework soporta esta transferencia de datos con el método

```
Transform cubeTrans = nuevo Transform;
...
m_shader->setUniform("nameOfVariable", cubeTrans->getMatrix() , false);
```

La variable "nameOfVariable" (¡por favor usa un nombre significativo!) debe ser definida como `uniforme` en el shader y contiene la matriz respectiva de la transformación "cubeTrans". Utilice la matriz para transformar las posiciones de los vértices en consecuencia.

Nota: En una versión anterior del framework, había un error en Transform.cpp que necesita ser corregido: la matriz de rotación debe ser inicializada en la línea 16 con `m_rotation(glm::vec3(0,0,0))`, no con `m_rotation()`.

2.3 Mi propio robot

Ahora queremos utilizar nuestro cubo para construir un robot sencillo. El resultado puede parecerse a la imagen de la derecha. El robot debe tener un torso, una cabeza, dos piernas, dos brazos superiores y dos brazos inferiores. Cada parte del cuerpo debe tener su propia matriz de transformación, pero siempre utilizaremos la misma geometría.
En general, el robot debe ser escalable en tamaño, debe ser posible moverlo (para colocarlo después en cualquier escena) y debe ser posible rotarlo. Además, los componentes individuales del robot deben poder girarse en una dirección razonable (para animar secuencias de movimiento sencillas).

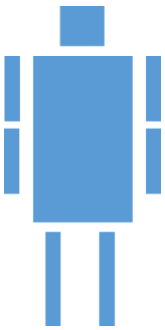

Figura 1: Ejemplo de robot

2.3.1 Gráfico de escenas

Piensa primero en el aspecto que debe tener el gráfico de la escena correspondiente. Haz un pequeño dibujo, esto simplificará considerablemente la siguiente implementación. Sin embargo, ¡ten en cuenta el orden en que se ejecutan las transformaciones del grafo de escena!

2.3.2 Borrar fondo

En cuanto empiece a animar sus objetos por fotogramas, notará que "manchan" el fondo. Estrictamente hablando, la imagen antigua del último fotograma no se borra y simplemente se dibuja "encima". Este problema es fácil de solucionar: Llama al método OpenGL `glClear(GL_COLOR_BUFFER_BIT)`.
También necesitas `glClearColor(0.0f, 0.0f, 0.0f, 1.0f)`. ¿Qué hacen los dos métodos en cada caso? Y como conclusión: ¿Cuándo hay que llamar a cada método?

2.3.3 Prueba de profundidad

Antes de dibujar varios objetos en la escena, otra advertencia: para la percepción humana, es natural que los objetos más cercanos al observador oscurezcan a otros objetos más lejanos. Este principio no está activado per se en OpenGL, hasta ahora la oclusión viene determinada por el orden en que se dibujan los objetos. Esto se remedia con estas líneas en `Scene::init()`:

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_GREATER);
glClearDepth(0.0);
```

Para poder ejecutar de nuevo la prueba de profundidad en cada fotograma (y no obtener un "emborronamiento" de los valores de profundidad, similar al de los valores de color), es necesario ampliar `glClear()` a `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`.

Nota: El ojo entrenado puede notar que estamos probando en la "dirección equivocada" aquí. Por defecto, OpenGL utiliza un sistema de coordenadas a la derecha, la cámara mira a lo largo del eje Z negativo. Por desgracia, ¡aún no tenemos cámara! Por lo tanto, los objetos que definimos ya están en Coordenadas Normalizadas de Dispositivo (ver conferencia posterior). Para más detalles, consulte `https://learnopengl.com/Advanced-OpenGL/Depth-testing` y `https://learnopengl.com/Getting-started/Coordinate-Systems`.
Este problema desaparece en cuanto añadimos una cámara en la siguiente hoja de ruta.

2.3.4 Gráfico de escenas

Transforme el gráfico de escena grabado. Crear una matriz de transformación separada para cada grupo de transformación en el gráfico. Cargue la matriz para cada parte del cuerpo en el sombreador (como se conoce de la tarea 2.2) y llame al comando dibujar para la geometría del cubo.
Para la implementación del gráfico de escena, sólo necesitas multiplicar las matrices de todos los grupos de transformación que van a actuar sobre una geometría en el orden correcto antes de cargarlas. En aras de la simplicidad, limítate por el momento al posicionamiento y escalado correctos de las geometrías individuales.

2.3.5 Rotaciones

Ahora, además de las traslaciones y escalados para el posicionamiento inicial, hay que integrar las rotaciones según las dependencias del gráfico de la escena. Las patas del robot deben poder "girar" hacia delante y hacia atrás para realizar una animación de marcha. Hay que tener en cuenta que no basta con una simple rotación alrededor del centro del objeto de la pierna, sino que la articulación de rotación debe estar situada en la cadera. Para ello, el framework ya proporciona el método `rotateAroundPoint(glm::vec3 point, glm::vec3 angles)`, que funciona según el conocido principio de primero mover el objeto al punto cero, luego rotarlo y finalmente volver a moverlo a la posición original.
Además de las piernas, los brazos también deben poder girar. La particularidad en este caso vuelve a ser el gráfico de escena, que garantiza que los brazos inferiores se muevan automáticamente con los superiores, pero que también puedan tener su propia rotación.
El resultado podría ser, por ejemplo, el siguiente:

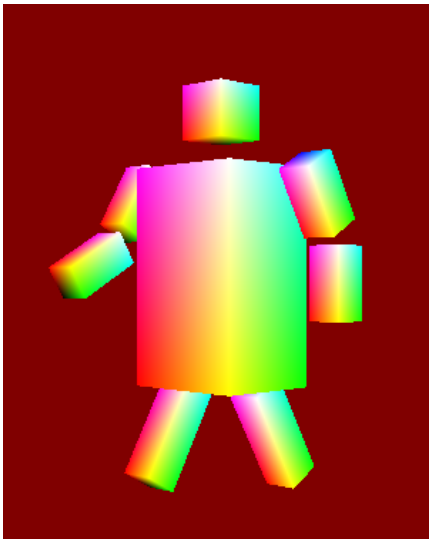

Figura 2: Robot acabado en 3D

2.3.6 Animaciones

Además de las transformaciones, se pueden implementar más animaciones en el método `render()`. Para ello, cargue una variable float directamente en el fragment shader y utilícela para cambiar la representación del color por fotograma. Para evitar saltos bruscos en los colores, son adecuados los cálculos con `sen` o `cos`. El tipo exacto de animación depende completamente de ti.