DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico

Módulo de computación gráfica

4to bimestre 2024 Introducción a la Visualización y Simulación Interactivas e Inmersivas

Grupo: 1
Corrector: Emmanuel Iarussi

Integrante	LU	Correo electrónico
Klimkowski, Victoria	1390/21	02vicky02@gmail.com
Laks, Joaquin	425/22	laksjoaquin@gmail.com
Pages, Julieta	1691/21	julib.pages@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1 Introducción

El objetivo de este trabajo es renderizar un modelo 3D representado como malla de triángulos, utilizando rasterización. Para ello usamos la herramienta WebGL que provee un entorno cómodo para el trabajo con objetos 3D.

Dado un proyecto, debimos completar una serie de funciones para conseguir que el modelo se visualice correctamente necesitamos implementar varias funciones:

- **GetModelViewProjection** - calculará la matriz de vizualización del modelo a través de la matriz de proyección y las translaciones y rotaciones deseadas. El resultado de esta función va a ser la matriz que haga todas las transformaciones que hagan falta sobre las posiciones de los vértices.

Y dentro de la clase MeshDrawer:

- **constructor** - crea el programa que vamos a usar para renderizar nuestros modelos, compila los shaders y hace las inicializaciones necesarias para el resto de las funciones.
- **setMesh** - carga el modelo seleccionado por el usuario en el buffer de posiciones para que después el método draw pueda usarlo desde ahí.
- **draw** - pasa los valores que hagan falta al shader para que se pueda renderizar el objeto.
- **swapYZ** - establece la variable uniforme 'swap' según lo indicado por el usuario. El resultado será que se vean swapeados los ejes y y z.
- **setTexture** - carga la textura indicada por el usuario al buffer que corresponde, y además establecerá los parámetros necesarios para que funcione correctamente.
- **showTexture** - activa o desactiva la textura.

La clase usa un vertex shader y un fragment shader que modificamos nosotros.

2 `GetModelViewProjection`

La función devuelve la matriz de transformación por la que vamos a multiplicar todos los vértices del modelo. Para poder hacer traslaciones usamos coordenadas homogéneas, así que la matriz que retrona es de 4×4.

Recibimos como entrada una matriz de proyección, los datos de la traslación, y los ángulos de rotación en el eje X y el eje Y.

Dado un ángulo θ de rotación en el eje X, sabemos que podemos expresar la transformación como una matriz de *cabeceo*, en coordenadas homogéneas nos queda:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dado un ángulo ϕ de rotación en el eje Y, podemos hacer algo similar con una matriz de *guiñada* como:

$$R_y = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Aprovechando las coordenadas homogéneas, podemos representar una traslación en $\Delta = (\Delta_x, \Delta_y, \Delta_z)$ como:

$$T = \begin{bmatrix} 1 & 0 & 0 & \Delta_x \\ 0 & 1 & 0 & \Delta_y \\ 0 & 0 & 1 & \Delta_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ahora, usando la matriz de proyección P que viene por parámetro, nuestra transformación final que tiene en cuenta todos estos factores nos queda:

$$M = PTR_yR_x$$

Que es la matriz que devuelve la función.

3 Clase `MeshDrawer`

La clase se ocupa de cargar mallas tridimencionales y texturas, y de interactuar con WebGL para renderizarlas.

3.1 Constructor

En el constructor de la clase nos encargamos de compilar los shaders, inicializar el programa y guardar las variables que van a ser necesarias para los otros métodos.

Tanto para el vertex shader como para el fragment shader debimos: crear los shaders (`vs` y `fs` respectivamente), cargarles el código fuente del que deberán leer (`meshVS` y `meshFS`), y compilarlos. Agregamos también un chequeo de que no haya habido errores en la compilación. Después creamos el programa que es la unión de estos dos shaders compilados.

Para poder acceder más tarde a las variables uniformes y los atributos de los shaders, necesitamos sus ubicaciones, que funcionan como un puntero para decirle al programa qué variable queremos modificar. Entonces para cada uniforme y atributo que se declara en los shaders, usamos las funciones `gl.getUniformLocation` y `gl.getAttributeLocation` para tener sus ubicaciones y las guardamos en variables de la clase.

También hay que inicializar los buffers que vamos a usar para darle valores a los atributos. Vamos a hablar de cómo se usan los buffers en los métodos correspondientes, pero en esta función los inicializamos com `gl.createBuffer`.

Finalmente, inicializamos `showTextureLocation`, para expresar que comenzamos queriendo mostrar la textura que se seleccione.

3.2 Métodos

3.2.1 `setMesh(newVertPos, texCoords)`

Este método actualiza los datos de la malla cargada por el usuario. Recibe un arreglo de las nuevas posiciones de los vértices, `newVertPos`, donde cada vértice está representado por tres valores (x, y, z), y también un arreglo de coordenadas de textura, `texCoords`, con dos valores (u, v) por vértice.

Esta función entonces debe cargar en el buffer de posiciones de los vértices, `vertbuffer` y en el buffer de coordenadas de textura, `textureCoordsBuffer`, los nuevos valores que deben tomar.

En WebGL, para editar un buffer primero hay que *bindearlo* a uno de los buffers de WebGL y después pasarle la información al buffer enlazado. En esta función queremos usar los buffers para guardar información sobre los vértices, entonces tenemos que usar `gl.ARRAY_BUFFER`. Tanto para nuestro buffer de las coordenadas de texturas como el de posiciones de vértices, primero los bindeamos a `ARRAY_BUFFER` y después cargamos la información de los parámetros.

En draw también vamos a necesitar saber la cantidad de triángulos que hay en el objeto. Lo calculamos en esta función dividiendo por 3 la cantidad de vértices y lo guardamos en una variable de la clase.

3.2.2 `swapYZ(swap)`

Este método controla si las coordenadas y y z de los vértices deben intercambiarse al renderizar según el booleano que toma como parámetro.

Entonces, indicamos cuál es el programa a modificar y luego cargamos en la variable uniforme que estará en `swapLocation` el valor indicado (que debería ser true o false).

Para que se consiga el efecto deseado, además debimos modificar el vertex shader para que incluya una uniforme swapYZ para almacenar esta información, y para que chequee su valor y decida acorde a ello las posiciones que deberán tomar las coordenadas y y z.

3.2.3 `draw(trans)`

Este método dibuja la malla en la pantalla.

Para ello empezamos seleccionando el programa de shaders, y seteando en el uniforme `mvp` la matriz de transformación (la retornada por `GetModelViewProjection`).

Luego vamos a querer recargar los atributos de cada vértice (como ya mencionamos anteriormente, estos son la posición y coordenadas de textura). Tanto `vertbuffer` como `textureCoordsBuffer` ya van a tener cargados los datos del nuevo modelo a visualizar. Entonces, escribimos los datos de cada uno de los buffers en los atributos `pos` y `textureCoord` reespectivamente.

Esto se logra primero bindeando el buffer correspondiente a `ARRAY_BUFFER` para indicar a cada vértice que lea el contenido de ahí. Una vez hecho esto debemos a su vez bindear el buffer en `ARRAY_BUFFER` con el atributo que corresponda, indicandole a WebGL de qué manera debe leer los datos. Por último, habilitamos que los shaders puedan usar los datos asociados al atributo al renderizar.

Una vez hecho todo esto, le decimos a WebGL que renderice nuestro programa, pasándole la primitiva que tiene que usar (triángulos) y cuántas tiene que dibujar, que lo calculamos en `setMesh`.

3.2.4 `setTexture(img)`

Este método establece que la textura pasada como parámetro se aplicará a la malla.

Entonces bindeamos el uniforme de textura con `TEXTURE_2D`, y luego cargamos la imagen en el target marcado por `TEXTURE_2D`.

Tuvimos problemas con esta función, hasta que nos dimos cuenta que `TEXTURE_MIN_FILTER` por defecto tiene valor `NEAREST_MIPMAP_LINEAR`, lo que espera que la textura tenga mipmaps, que no es el caso. Entonces debimos establecer a mano que use `LINEAR` para que en cambio muestree usando interponlación lineal, y con eso se solucionó. Para `TEXTURE_MAG_FILTER` al principio hicimos lo mismo hasta que nos dimos cuenta que ya tiene por defecto establecido `LINEAR`.

Agregamos también dos líneas que establecen `TEXTURE_WRAP_S` y `TEXTURE_WRAP_T` en `CLAMP_TO_EDGE`. Esto establece que en caso de que los píxeles que estén fuera del rango normalizado [0, 1] se rellenan con el color del borde de la textura. Por defecto tienen en cambio el valor `REPEAT`, pero este puede romperse si las texturas no son múltiplo de 2. Aunque este NO es el caso para ninguna de las texturas con las que probamos, nos pareció bien dejarlo a manera de que si se quisiera pasar otra textura nueva, evitar encontrarse con el problema.

Una vez cargada la textura, el fragment shader va a poder usarla en su variable uniforme de tipo `sampler2D`. Como cargamos para cada vértice la posición (u, v) de la textura que le corresponde, en el shader podemos decir que el color de ese punto es el color de la textura sampleado en ese (u, v). El renderizado después va a interpolar ese sampleado para todas las posiciones entre los vértices de los triángulos.

3.2.5 `showTexture(show)`

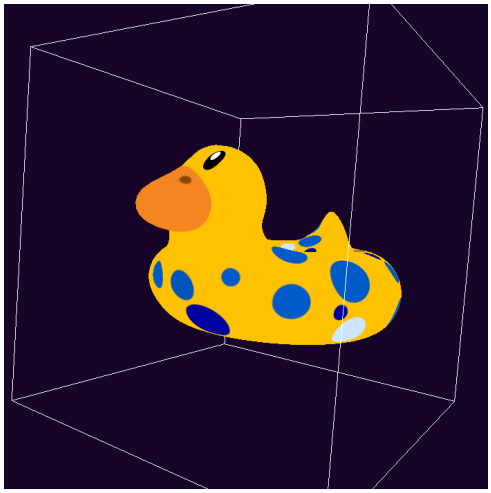
Este método controla si la textura debe mostrarse o no al renderizar según el booleano por parámetro.

Para esto, debimos establecer en el fragment shader un nuevo uniforme useTexture para almacenar si el usuario desea o no ver la textura, y luego agregar un chequeo del valor de este atributo para decidir si dejar en `gl_FragColor` la textura cargada o no.

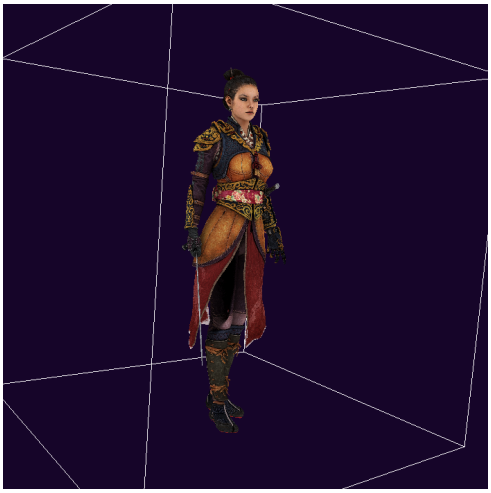
En caso de querer usar la textura, usamos un uniforme nuevo `uSampler`, que es un sampler que tomará las coordenadas de textura y sampleará el color de la textura en ese punto.

En caso contrario, le cargamos algún color a mano.

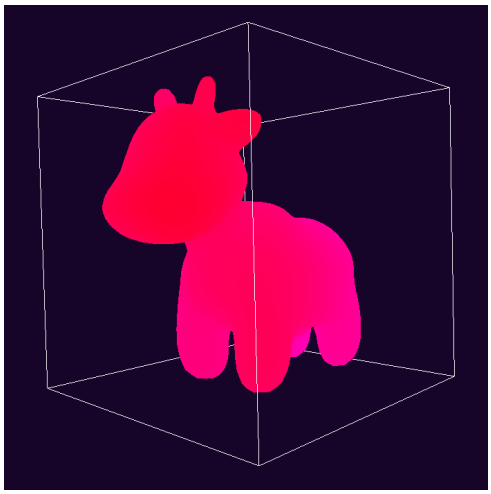
4 Resultados



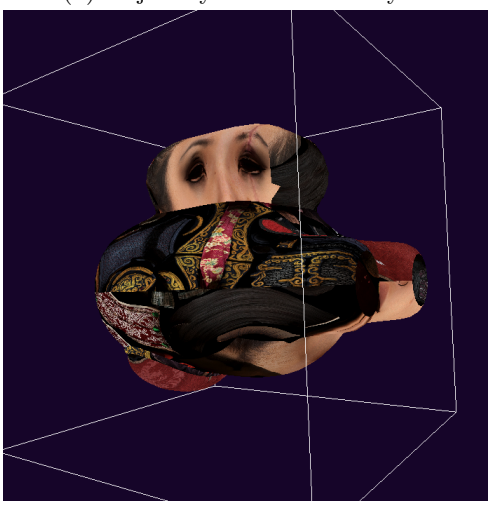
(a) Objeto duck con textura duck.



(b) Objeto nyra con textura nyra.



(c) Objeto cow con textura desactivada.



(d) Objeto among con textura nyra.

Figure 1: Renderizaciones.