Realidad-Virtual-Teoria-frecuent...

mike_

Realidad Virtual

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería
Universidad de Huelva

Etapas del proceso de renderizado

- VertexShader: se ejecuta sobre cada vértice de forma independiente. Utiliza atributos de entrada asociados a cada vértice y genera como salida la posición del vértice en coordenadas clip.
- Ensamblado de primitivas: es una etapa no programable. Recibe la indicación de las primitivas a dibujar y los atributos de los vértices que las forman, generados por el VertexShader. Además, detecta si están ocultos o fuera del área de dibujo.
- Teselado: recibe los datos de un parche y genera nuevas primitivas. Se divide en:
    - Control de teselado (TCS): recibe los vértices del patch y genera los valores de los vértices de las primitivas que se crearán.
    - Generación de primitivas (TPG): genera las primitivas en un espacio de coordenadas llamado espacio de parámetros, sus coordenadas se denominan u y v, que van de 0 a 1.
    - Evaluación de teselado (TES): transforma las coordenadas del espacio de parámetros a coordenadas de posición de los vértices (clip).
- GeometryShader: es una etapa opcional. Recibe primitivas ya ensambladas y permite realizar modificaciones sobre ellas.
- Rasterización e interpolación: es una etapa no programable. Estudia qué vértices quedan dentro del clipping volumen, desechando aquellos que estén fuera, y adaptando las primitivas que estén parcialmente fuera a través de nuevos vértices. Después, se identifican qué pixeles forman parte de la primitiva a dibujar según la ubicación del centro del píxel.
- FragmentShader: se ejecuta por cada píxel dentro de la primitiva, recibiendo los valores interpolados para el píxel. Genera el color asociado al píxel.
- Operaciones de raster: combina los diferentes fragmentos para generar la versión final de la imagen.

El proceso de renderizado genera al final el contenido de la imagen en una estructura que conoceremos como Color Buffer.

Creación de un programa gráfico

En un programa gráfico se distinguen dos partes principales:

- Las ejecutadas por la CPU, que utilizan funciones de OpenGL en C/C++.
- Las ejecutadas por la tarjeta gráfica, los shaders. Escritos en GLSL, serán cargados al programa.

Desde la parte ejecutada por la CPU, se crean los shaders y se les asigna el código en modo texto. Después se compilan y quedan listos para ser añadidos al programa gráfico. Serán obligatorios tanto el VertexShader como el FragmentShader para poder instalar el programa en el pipeline de renderizado.

Tipos de proyecciones

La proyección convierte el volumen visible del modelo en el volumen clip. Según como lo haga:

- Proyección ortográfica: consiste en transformar un volumen de forma rectangular en el volumen clip. Supone una transformación de escalas muy sencilla, pero provocaría que los objetos se vean del mismo tamaño estén cerca o lejos.

- Proyección en perspectiva: consiste en transformar el espacio que se ve desde el observador con un cierto ángulo de visión en el volumen clip. El volumen tendría una forma similar a un tronco de pirámide y provocaría que los objetos más cercanos se vean de mayor tamaño que los más lejanos.

Iluminación

El modelo de iluminación más utilizado es el modelo de Phong, que consiste en considerar la luz como la suma de tres efectos diferentes:

- Luz ambiental: procede de todas direcciones.
- Luz difusa: proviene de una determinada dirección y al chocar con un objeto se dispersa en todas las direcciones. La intensidad de esa dispersión depende del ángulo de incidencia.
- Luz especular: es una luz que proviene de una única dirección y se refleja con el mismo ángulo de incidencia al chocar con un objeto.

El color final de cada punto de un objeto se calcula como la suma de los efectos de color de cada tipo de luz.

- Color provocado por luz ambiental: es el producto de la luz ambiental por el color del material ante la luz ambiental.
- Color provocado por luz difusa: es el producto de la luz difusa por el color del material ante esa luz, multiplicados además por un factor de incidencia de la luz, siendo el coseno del ángulo formado por la dirección de la luz y el vector normal de la superficie.
- Color provocado por luz especular: es el producto de la luz especular por el color del material con esa luz, multiplicados por un factor especular. Este factor se calcula como el coseno del ángulo entre el vector posición del observador (del punto al observador) y el vector reflexión de la luz, elevado a un coeficiente de brillo.

Es importante tener en cuenta los distintos sistemas de coordenadas de los vectores que se utilizan. Para realizar todos los cálculos se transformarán todos los vectores a coordenadas de observador.

Los cálculos para obtener el color final se hacen en el FragmentShader, por tanto, es necesario indicarle mediante variables uniformes de entrada las direcciones de los tipos de luz y el color del material del objeto para cada tipo de luz.

Texturas

Una textura es la aplicación de una imagen a una primitiva geométrica. Los puntos de las texturas son los texels.

Según cómo esté estructurada, hay varios tipos de texturas:

- Texturas 1D: es un vector de texels en una dimensión.
- Texturas 2D: matriz de textels.
- Texturas 3D: matriz tridimensional de texels (colección de imágenes).
- Texturas Cubemap: es un conjunto de 6 superficies cuadradas distribuidas como las caras de un cubo.

Para trabajar con texturas, se definen objetos texturas que definen buffers de datos que se almacenan en la tarjeta gráfica. Cada objeto textura se identifica mediante un número.

saboteas a tu propia persona?
cómo?? escríbelo **aquí** y táchalo

manual de instrucciones: escribe sin filtros y una vez acabes, táchalo (si lo compartes en redes mencionándonos, te llevas 10 coins por tu cara bonita)

DESFÓGATE CON WUOLAH

Se reservan/crean los identificadores de texturas mediante la función glGenTextures, indicándole el número de identificadores que se deseen y un array donde los almacene. Después, se define el tipo de textura que tendrá un identificador a través de la función glBindTexture. Por último, para cargar una textura a un identificador, se utiliza la función glTexImage_D según el tipo de textura.

También se puede asignar las texturas copiando los valores del color buffer con glCopyTexImage_D, indicando desde qué posición tiene que copiar. Otra forma puede ser cargar sólo un trozo de una textura sobre la ya cargada con glSubTexImage_D, indicando desde qué posición empieza a sobrescribir.

Los puntos en las texturas se identifican con coordenadas (s, t, r, q) que parten desde la esquina inferior izquierda. Para aplicar una textura sobre una primitiva, se calculan las coordenadas de texturas correspondientes a cada píxel. Dado que los tamaños de la primitiva y la textura pueden ser diferentes, se filtrará la textura para aumentarla o encogerla. El proceso de filtrado se puede configurar con las funciones glTexParameter.

En caso de que los tamaños de la primitiva y la textura sean muy diferentes, podría suceder que a dos píxeles consecutivos se le asocien texels alejados, produciendo aliasing. Una solución serían los Mipmaps, que consisten en tener versiones de la textura en tamaños inferiores.

Vertex Buffer Object

Un VBO es una estructura de datos que puede almacenar en la memoria de la GPU diferentes tipos de información, según qué tipo de buffer se le asigna al activarlo como, por ejemplo: almacenar atributos de los vértices o almacenar una lista de índices a vértices.

Se crean, desde el código para la CPU, a través de la función glGenBuffers, donde se le pasa el número de buffers a crear y los identificadores de los buffers. Ya creado un VBO, para poder utilizarlo es necesario activarlo con la función glBindBuffer, donde se le indica qué tipo de buffer será. Por último, para almacenar los datos se utiliza la función glBufferData, indicándole el tipo de buffer a llenar y dónde están los datos que va a guardar.

Vertex Array Object

Cuando ya estén creados los VBOs y almacenados los datos de los vértices, el siguiente paso es crear los Vertex Array Objects. Éstos son estructuras de datos que definen el mapeo entre los atributos (almacenados en los VBOs) y los nombres de las entradas de los shaders.

A través de la función glGenVertexArrays podemos crear un cierto número de VAOs y obtener sus identificadores. Para usar un VAO, se activa primero con la función glBindVertexArray.

Para definir un atributo perteneciente a un VAO se usa la función glVertexAttribPointer, indicándole el índice, cuantos datos habrá por atributo y el tipo que será.

Para asignar la posición de una variable de entrada, se utiliza la función glBindAttribLocation, indicándole el shader, la posición del atributo y el nombre de la variable.

Generación de sombras

Para poder añadir sombras a un dibujo, es necesario conocer si cada píxel de la imagen se encuentra en sombra o no, ya que, si está en sombra, sólo será iluminado por la luz ambiental.

El algoritmo básico para generar sombras es el ShadowMap, que consiste en generar previamente una imagen de la escena desde el punto de vista del foco de luz y, en lugar de llevarla a la pantalla, se almacena en un Frame Buffer, guardando como textura la profundidad de los píxeles.

Para generar esa imagen, es necesario tener la matriz LightView que transforma de coordenadas de modelo a coordenadas de observador, siendo éste la luz, y la matriz LightProjection, que hará la transformación a coordenadas clip. Según el origen de la luz, si es un foco, se debe configurar como una proyección en perspectiva, pero si son luces direccionales, se calculará como una proyección ortográfica.

Para saber si un punto del modelo está en sombra, a través sus coordenadas clip en el sistema de la luz, se obtiene la distancia (profundidad) hasta el origen de luz. Después, tomando la imagen generada previamente desde el punto de vista de la luz, si la textura de la imagen, en el píxel donde se situaría el punto, tiene un valor menor (una profundidad menor), significa que hay un objeto más cercano a la luz impidiendo que le llegue esa luz y, por tanto, el punto estará en sombra.

Sin embargo, el principal problema de este método es el aliasing. Ya que calculamos la sombra a partir de una imagen (textura) generada desde la luz, un punto en la textura puede abarcar en el modelo a muchos más puntos, quedando un pixelado en los límites de la sombra muy evidente.

Shader de Geometría

Es un shader opcional en el proceso de renderizado. Se ejecuta por cada primitiva generada en la etapa de ensambado y su objetivo es realizar cambios sobre ella. Algunos de estos cambios pueden ser:

- Eliminar primitivas.
- Incorporar más nivel de detalle sustituyendo una primitiva por un número mayor de primitivas e interpolando los valores para los nuevos vértices.
- Introducir nuevas propiedades en los vértices, calculadas en base a la información geométrica. No puede realizarse en el VertexShader ya que ahí no se trata información de vértices vecinos.
- Sustituir la primitiva por otra diferente.

Tiene como entrada un array que almacena la información de cada vértice de la primitiva. Para cada vértice, la estructura recibida indica: su posición, el tamaño del punto y distancia a los planos clip (si esta función está activada).

Su salida predefinida son las nuevas primitivas.

Su funcionamiento se basa en dos funciones predefinidas:

- EmitVertex: genera los vértices de las nuevas primitivas. Se guarda como vértice el valor que esté en la salida del shader (el valor de gl_position).
- EndPrimitive: indica que los vértices generados anteriormente forman una primitiva. Los siguientes vértices que puedan generarse pertenecerán a otra primitiva.

Vertex Shader

Es la primera etapa del proceso de renderizado. Se ejecuta sobre cada vértice por separado utilizando atributos de entrada asociados al vértice como la posición, la normal, coordenadas de textura … También puede utilizar variables uniformes para compartir valores de entrada comunes a todos los vértices.

Tiene además dos entradas predefinidas que indican el índice del vértice en los arrays de atributos y el número de instancia del vértice, por defecto es 0.

Genera como salida, al menos, la posición del vértice en coordenadas normalizadas (clip). Puede generar también otros valores que se quieran utilizar en siguientes etapas.

Ejemplo de Vertex Shader que transforma la posición del vértice a coordenadas clip:

```
#version 400
in vec3 Pos;
uniform mat4 MVP;
void main() {
  gl_position = MVP * vec4(Pos, 1.0);
}
```

Fragment Shader

Se ejecuta por cada píxel dentro de la primitiva, recibiendo los valores interpolados para el píxel y genera el color asociado al píxel.

Sus entradas predefinidas serían:

- Las coordenadas del píxel.
- El índice de la primitiva que está asociada al píxel.
- Una variable que indica si la primitiva está generada en su cara frontal o en su cara trasera.

Una salida que tiene ya definida es la profundidad asociada al píxel, que se almacenará en el buffer de profundidad.

Aunque el objetivo principal de este shader sea calcular el color del píxel, no existe una variable de salida predefina para este dato, por lo que se toma como salida definida para el color aquella salida que esté definida en la posición 0.

Ejemplo de Fragment Shader que dibuja de color rojo todos los píxeles de la primitiva:

```
#version 400
out vec4 Color;
void main() {
  Color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Tipos de primitivas

Las primitivas incluidas en OpenGL son:

- **Punto**, compuesto por un único vértice.
- **Línea**, que une dos vértices.
- **Triángulo**, formado por tres vértices. Es la forma de primitiva más utilizada.

- Parche. No tienen un significado geométrico definido. Es un tipo especial de primitiva que se utiliza en los shaders de teselado, donde se convierte la información del parche en un conjunto de primitivas geométricas.

Salvo las primitivas de tipo parche, que se convierten en otras primitivas tras la etapa de teselado, todas pueden ser tratadas directamente por el Fragment Shader, a no ser que estén definidas como primitivas de adyacencia. Estas primitivas son aquellas que comparten aristas con otras primitivas, pero una de ellas se observa desde su cara frontal y la otra desde su cara posterior.

Para distinguir entre la cara frontal y posterior de una primitiva, se obtiene el vector normal de la primitiva. Este es un vector perpendicular a la superficie de la primitiva, y si su dirección en el eje z es positiva, entonces el triángulo está viéndose desde su cara frontal, si es negativa, desde su cara posterior.

De forma más simple, la cara que se ve es la frontal si los vértices que componen la primitiva, en el momento de representarse, siguen un recorrido en sentido antihorario.

Teselado

Es una etapa opcional del proceso de renderizado, pero si un programa contiene esta etapa, sólo se pueden renderizar primitivas de tipo parche. Esta etapa se divide en tres fases:

- Control de teselado (TCS): recibe los vértices del parche y con su información genera los valores de los vértices de las primitivas que se crearán.
- Generación de primitivas (TPG): recibe una lista con los vértices, obtenidos de la fase anterior, de la primitiva que va a crear. Las primitivas que genera las sitúa en un espacio de coordenadas llamado espacio de parámetros, donde sus coordenadas se denominan u y v, que van de 0 a 1.
- Evaluación de teselado (TES): recibe las coordenadas de cada vértice (tessCoord) en el espacio de parámetros y los atributos de los parches (gl_in) y los transforma a coordenadas de posición de los vértices (clipping volume).

El nivel de teselado se puede configurar como variables de entrada uniforme del TPG que indiquen el número de puntos de control que tendrá una superficie. Pero también puede calcularse internamente según la profundidad por ejemplo (indicando en la entrada los niveles máximo y mínimo de profundidad).

Frame Buffer Object

El proceso de renderizado genera el contenido de la imagen en el ColorBuffer. Si además se encuentra activada la opción de "test de profundidad", se almacena la información sobre la coordenada z (profundidad) en una estructura paralela llamada DepthBuffer. Existe otro buffer, el StencilBuffer, que se utiliza como buffer auxiliar para otro tipo de información configurable.

Un FrameBuffer Object (FBO) es una estructura de datos que describe la salida del proceso de renderizado y contiene enlaces al ColorBuffer, DepthBuffer y StencilBuffer. Por defecto, OpenGL crea un FBO con identificador 0 que asocia a la pantalla.

Para crear un FBO se utiliza la función glGenFramebuffers, indicándole el número de FBOs a crear y el array donde están los identificadores. Para activar uno, se utiliza glBindFramebuffer.

Para dirigir el proceso de renderizado a memoria, se crea un FBO y los buffers que estén vinculados a él (Color y Depth). Para vincular estos buffers al FBO, se utiliza glFramebufferTexture2D si el DepthBuffer está cargado como textura, si no, es necesario cargarlo mediante un RenderBuffer. El tamaño de los buffers debe ser el mismo. Una vez vinculados, se activa el FBO. Por último, se indica que el FBO será el buffer de dibujo mediante glDrawBuffers. Después hay que devolver la salida del renderizado a la pantalla.

Subrutinas de GLSL

Una subrutina es un mecanismo que permite seleccionar una función basado en el valor de una variable uniforme. Esto sucede en tiempo de ejecución sin necesidad de modificar el programa ni de instrucciones if.

Para crear una subrutina se utiliza la instrucción subroutine seguida de la declaración del tipo de función. Después, para crear una variable uniforme que contenga un puntero a una subrutina se utiliza la instrucción subroutine uniform subrutina variable.

Para crear funciones que desarrolla una subrutina, se incluye la instrucción subroutine (subrutinas) y a continuación la declaración de la función.

La subrutina se utiliza en el código como si fuera una función.

La selección de la subrutina en el programa OpenGL se realiza asignando el valor a la variable uniforme. Primero se obtiene la posición de la función con glGetSubroutineIndex y, después, a la variable uniforme de tipo subrutina se le asigna el valor obtenido mediante glUniformSubroutine_.

Transformación de coordenadas

Generalmente, las coordenadas de los vértices se expresan inicialmente en un sistema de coordenadas propio del objeto al que pertenecen. Cuando se añaden más objetos al modelo, es necesario que todos los objetos estén ubicados en un mismo sistema de coordenadas, que llamaremos coordenadas del modelo. Estando todas las figuras representadas con el mismo sistema de coordenadas, ya se puede realizar la proyección creando el volumen clip del modelo desde un punto de vista que será el observador.

Para realizar las transformaciones entre los diferentes sistemas de coordenadas, se utilizan las siguientes matrices:

- Model: esta matriz se utiliza para transformar las coordenadas locales de un objeto a coordenadas del modelo. Sitúa el objeto en una posición del modelo.
- View: esta matriz se utiliza para transformar el sistema de coordenadas del modelo a otro sistema de coordenadas en el que el observador es el origen de coordenadas, lo transforma a coordenadas de observador.
- Projection: esta matriz, a partir de las coordenadas de observador, obtiene las coordenadas en el clipping volumen.

Según como sea la proyección, la matriz será de una forma u otra. Tipos de proyección.

Transform Feedback

Una de las formas para generar escenas con figuras no rígidas es el Transform Feedback. Este consiste en modificar los atributos de los vértices en función de los valores generados en un renderizado anterior, lo que requiere almacenar los datos generados en la memoria de la GPU y ser capaz de leerlos en futuros renderizados.

La técnica utilizada se conoce como buffer “ping-pong” porque realiza el renderizado en dos pasadas. En la primera se ejecuta sólo el VertexShader, y en la segunda se realiza el proceso completo, tomando como entrada el buffer generado en la primera pasada.

Se definen dos Transform Feedback Object (TFO) que describen los buffers a utilizar como salida y entrada del VertexShader. Estos se crean con glGenTransformFeedbacks, indicando el número de buffers y el array donde están los identificadores, en este caso serían 2. Se activan con glBindTransformFeedback y se enlazan tantos buffers como se quieran tener ubicados con glBindBufferBase (un buffer por cada valor que se quiera conservar).

Después, se define la relación entre las salidas de los shaders y los buffers que forman el TBO con glTransformFeedbackVaryings.

Durante el renderizado hay que iniciar el proceso de transform feedback con la función glBeginTransformFeedback.

Cuando se lancen las primitivas de dibujo, las salidas del VertexShader se almacenarán en los buffers indicador del TFO. Para finalizar el proceso de transform feedback se utiliza el comando glEndTransformFeedback.