Universidad Rey Juan Carlos

Gráficas y Visualización 3D

Práctica 7: *Matrices de Proyección*

Katia Leal Algara

Agustín Santos Méndez

1. **Ortogonal**: “Example 5-1.html”, hasta ahora estamos representando triángulos planos, pero para dibujar en tres dimensiones debemos introducir una nueva matriz. Se llama la **matriz de proyección**. En nuestras prácticas vamos a utilizar dos tipos de proyección: ortográfica y en perspectiva.

   Vamos a representar un triángulo en perspectiva ortográfica. Visualmente apenas vamos a ver ninguna diferencia con lo que ya teníamos, pero en nuestro código hemos introducido una nueva matriz.

   Observa que hemos definido una nueva variable llamada “uPMatrix”. Al igual que nuestra anterior matriz, esta variable es de tipo uniform. También es importante destacar el orden de la multiplicación de las matrices. Observa que, si leemos el código de derecha a izquierda, primero se calculan las coordenadas del punto, posteriormente le aplicamos la transformada definida por la matriz **ModelView** y finalmente la de proyección.

   En este ejemplo vemos una representación muy parecida a la que ya teníamos en los ejemplos anteriores. Eso ha sido así ya que siempre hemos tenido figuras planas y nos hemos movido por el plano “Z = 0”.

   Otro de los cambios que hemos realizado es la preparación y conexión de la matriz de proyección. Queremos llamar la atención sobre dos puntos de interés. Por una parte, la matriz de proyección se suele construir una sola vez. Normalmente en nuestros ejemplos no vamos a cambiar el tipo de proyección (en los juegos si suele ser habitual). Para evitar realizar operaciones innecesarias, solo se almacena esa variable una sola vez. En nuestro código hemos elegido la función “getUniforms”.

Otro de los aspectos que queremos destacar es el uso de la función "**mat4.ortho**":

mat4.ortho(left, right, bottom, top, near, far, out)

Generates a orthogonal projection matrix with the given bounds

Parameters:

Name	Type	Description
`left`	number	Left bound of the frustum
`right`	number	Right bound of the frustum
`bottom`	number	Bottom bound of the frustum
`top`	number	Top bound of the frustum
`near`	number	Near bound of the frustum
`far`	number	Far bound of the frustum
`out`	mat4	mat4 frustum matrix will be written into

Returns: *out,* type mat4

Ejercicios:

a) Nuestro triángulo se está moviendo en el eje "X". Intenta que se mueva en el eje "Z" haciendo que el valor de "Z" esté entre -10 y 10. ¿Notas algún cambio? ¿Tiene sentido lo que ves? ¿Debería hacerse más pequeño cuando se aleja y más grande cuando se acerca?

b) ¿Dónde tienes la cámara? ¿Cómo puedes saberlo?

c) Haz lo mismo que antes pero ahora haz que el rango del valor de "Z" sea entre -12 y 12. ¿Has notado algo diferente? ¿Tiene sentido?

d) Cambia el tamaño del canvas. Pon algo que sea más alargado (por ejemplo, 800x200). Observa que el triángulo sigue teniendo la misma forma. Eso es debido a que en nuestro código se calcula un "ratio". Encuentra el punto en el que se realiza esa operación. Intenta cambiar el código por "var ratio = 1.0;" y juega con el tamaño del canvas. ¿Qué observas?

2. **Perspectiva**: “Example 5-2.html”, El siguiente paso es cambiar nuestra proyección por una perspectiva. Ahora podemos ver que el triángulo realmente se hace grande o pequeño cuando se mueve en el eje Z. Según el triángulo se aleja de nosotros se ve más pequeño. Antes, con una matriz de tipo ortográfica eso no sucedía. ¿Sabrías explicar la razón de ese comportamiento? Este código se parece bastante al anterior. De hecho comparte el mismo código de shaders. La principal diferencia es cómo construye la matriz de proyección.

mat4.perspective(fovy, aspect, near, far, out)

Generates a orthogonal projection matrix with the given bounds

Parameters:

Name	Type	Description
`fovy`	number	the field-of-view in degree
`aspect`	number	the aspect ratio of the field of view (the aspect ratio of the viewport, typically width/height)
`near`	number	the minimal distance from the viewer which is still drawn (0.1 world units)
`far`	number	the maximum distance from the viewer which is still drawn (100.0 world units)
`out`	mat4	mat4 frustum matrix will be written into

Returns: *out,* type mat4

Ejercicios:

a) Nuestro triángulo ya se está moviendo en el eje “Z”, pero ¿Podrías decir el rango de valores que está tomando? ¿Tiene sentido que tenga un rango negativo?
b) ¿Dónde tienes la cámara? ¿Cómo puedes saberlo?
c) Intenta que el triángulo se mueva en el eje “Z” entre -6 y 6. ¿Has notado algo diferente? ¿Tiene sentido?
d) Dibuja varios triángulos moviéndose por los ejes “X” e “Y” pero sin moverse de la coordenada “Z”. Haz que cada triángulo tenga una coordenada “Z” diferente. Tendrás que ver varios triángulos de diferentes tamaños moviéndose por la pantalla.

e) Cambia los movimientos del ejercicio anterior para que también se muevan en profundidad.
f) Cambia el tamaño del canvas. Pon algo que sea más alargado (por ejemplo, 800x200). Observa que el triángulo sigue teniendo la misma forma. Eso es debido a que en nuestro código se calcula un “ratio”. Encuentra el punto en el que se realiza esa operación. Intenta cambiar el código por “var ratio = 1.0;” y juega con el tamaño del canvas. ¿Qué observas?

3. **Dibujando un cubo**: “Example 5-3.html”, el siguiente paso es introducir un modelo algo más complejo. Vamos a representar un cubo utilizando el código del ejemplo anterior. En este ejemplo solo cambiamos el modelo. El resto sigue siendo muy similar.

   En este ejemplo hemos guardado las coordenadas del cubo en un array y hemos utilizado índices (Elements). Debido al número de puntos, es más cómodo guardarlo como indica el ejemplo. Con los colores sucede algo similar.

   Notad también que hemos activado el test de profundidad mediante “gl.Enable(gl.DEPTH_TEST);”. Por lo tanto, si está habilitado, hace comparaciones de profundidad y actualiza el buffer de profundidad. Tenga en cuenta que incluso si el búfer de profundidad existe y la máscara de profundidad no es cero, el buffer de profundidad no se actualiza si el test de profundidad está deshabilitado. Por otra parte, gl.depthFunc(gl.LEQUAL); especifica la función de comparación de profundidad, la cual establece las condiciones bajo las cuales se dibujará el píxel. Normalmente, el test de profundidad se realiza tras finalizar la ejecución del Fragment Shader, con la salida del mismo. En este caso, el test se supera si el valor entrante es menor o igual (LEQUAL) que el valor del buffer de profundidad.

   Este código tiene cambios significativos con respecto a los ejemplos anteriores, aunque casi todo lo utilizado ya lo conocemos. Hemos utilizado índices con la misma técnica que utilizamos en el Example 2-4. La forma de dibujar es similar (usando drawElements). También hemos empaquetado todos los datos del cubo en un solo buffer mezclando coordenadas y colores. Vimos algo parecido en los Examples 2-*. Quizás el único cambio llamativo es la utilización de un control de profundidad para distinguir las partes vistas de las ocultas. Ahora cuando hacemos un clear, también le indicamos a WebGL que debe borrar el buffer de profundidad.

Ejercicios:

a) Dibuja varios cubos rotando en diferentes posiciones.
b) Simula un sistema solar representando los planetas como cubos (reutiliza el modelo que hemos construido en el ejemplo). Haz un cubo más grande en el centro y varios cubos más pequeños orbitando alrededor de éste.

4. **Dibujando figuras**: “Example 5-4.html”, el último ejercicio consiste en dibujar varias figuras en movimiento. De nuevo, no tendremos que modificar los shaders. Solo tocaremos la construcción de las matrices y la forma de dibujar los buffers. En este código no existen conceptos nuevos. Simplemente lo hemos juntado todo para que se genere algo más complejo.

   Ejercicios:

   a) Dibuja alguna otra figura sencilla pero que sea diferente a las utilizadas.
   b) Dibuja una o varias esferas.