UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNLVIRTUAL

Manipulación de objetos en 3D

Unidad 1
Conceptos de 3D

Docente
Jaime Fili

Autores
Jaime Fili, Alejandro Vairoli

CONTENIDOS

CONTENIDOS 1

1. ¿Qué significa 3D? 2

2. Sistema de coordenadas 2

2.1. Tamaño 3

3. Matrices de transformación 4

3.1. Matriz de mundo o modelo 4

3.2. Matriz de vista 5

3.3. Matriz de proyección 6

4. Triángulos 7

Referencias 10

1. ¿Qué significa 3D?

En la actualidad, el término es empleado de formas tan variadas y con objetivos tan dispares que, si nos hacen esta pregunta, probablemente no podamos brindar una única respuesta, sino varias y en función de un determinado contexto.

Para nosotros, *3D* es un entorno de tres dimensiones (por convención, *X,Y,Z*) que es representado en una superficie 2D.

Si bien sabemos que han estudiado los modelos 3D en otras materias, el objetivo de esta unidad es repasar el tema brevemente en el lenguaje adoptado por esta asignatura.

Este entorno 3D es virtual y sólo existe a nivel de datos; no es posible verlo, ya que nuestros equipos son 2D y, por lo tanto, incapaces de reproducir contenido 3D real. En lugar de esto, disponemos de *proyección,* que es una técnica que nos permite observar el entorno 3D con las limitaciones de nuestros equipos. Esto es, obtener una imagen o proyección del entorno 3D sobre una vista 2D (pantalla). Este concepto es muy importante y está presente en múltiples áreas dentro del mundo 3D.

Por definición, este entorno no es un *lugar* real; no tiene tamaño ni posiciones, límites ni referencias; no existe arriba y abajo, ni derecha o izquierda. Cada API (Interfaz de Programación de Aplicaciones) se encarga de normalizar la disposición de los ejes, con el objetivo de lograr un entorno menos abstracto y brindar algún punto de partida utilizable. No obstante, otros aspectos necesitan ser definidos por nosotros mismos al comienzo de cada proyecto.

XNA, como cualquier otra API, determina algunas características del entorno y nos ofrece herramientas de programación para facilitarnos su utilización.

Todo lo que aprendieron en otro entorno se aplica también, en mayor o menor medida, a XNA, pero con la visión propia de los creadores. Lo mismo ocurre a la inversa.

2. Sistema de coordenadas

Como dijimos anteriormente, cada API define su propio entorno 3D de acuerdo a sus propios intereses, pero existen ciertos criterios que están acotados a sólo un grupo. Uno de ellos es el *sistema de coordenadas.*

XNA utiliza el sistema de coordenadas conocido como Right-Hand, que especifica cómo estarán dispuestos los tres ejes de nuestro espacio virtual 3D.

Observemos en el siguiente gráfico cómo se presenta el sistema desde nuestra perspectiva, sentados frente a la máquina:

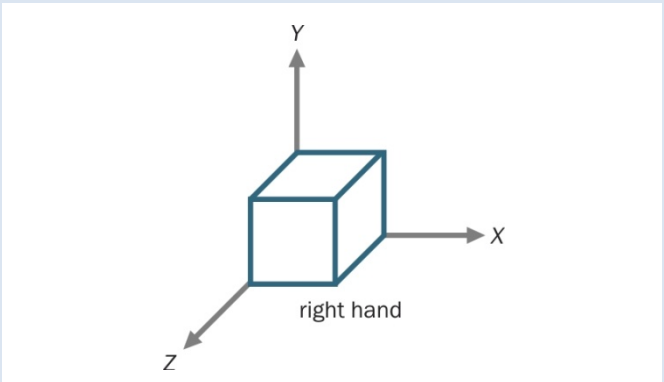

Figura 1. Sistema de coordenadas Right-Hand de XNA.

- El eje *Y* es el vertical y se incrementa (o apunta) hacia arriba.
- El eje *X* es el horizontal y apunta hacia la derecha.
- El eje *Z* es la profundidad y apunta hacia el frente.

Cabe destacar que la elección del sistema de mano derecha por XNA generó mucha controversia, ya que DirectX utiliza el sistema de mano izquierda y XNA es –básicamente– una encapsulación de DirectX sobre .Net.

Más abajo, en el apartado Referencias, recomendamos un artículo muy bueno extraído de Wikipedia, que explica los dos sistemas de coordenadas más adoptados por las API de videojuegos. Les sugerimos su lectura, ya que contiene información que les resultará de mucha utilidad.

2.1. Tamaño

Es un elemento que tenemos que considerar seriamente al momento de iniciar un desarrollo. Por ejemplo:

¿Qué significa un punto en la coordenada (0,0,1)?

- ¿Que está a un centímetro en *Z*? ¿Que está a un metro?
- ¿Que está a una pulgada?

La respuesta correcta es que está en todas y en ninguna, o mejor dicho, donde se nos ocurra, ya que podemos establecer que ese valor *1* equivale a 10 centímetros. Así, un personaje que tenga 1,70 m de alto, medirá 17.0 en la coordenada *Y*.

Puede ocurrir que nuestro juego sea de naves espaciales y tengamos un crucero de 300 m de ancho. Esto no implica que el modelo tenga 3000.0 en la coordenada *X*. Por el contrario, en este caso quizás convenga establecer que cada punto en el entorno 3D equivalga a 10 metros, por lo que nuestro modelo medirá 30.0 en la coordenada *X*.

Como último caso podemos suponer que tenemos un juego donde los personajes no poseen ninguna medida que pueda ser considerada real o relevante, ya que son virtuales o microscópicos. Es decir, donde el tamaño no es importante. De todas maneras, siempre deberemos tener en cuenta la necesidad de relacionar esos personajes en el entorno 3D, ya sea para desarrollar modelos, chequear colisiones o que los modelos no salgan de nuestro mundo. En cualquier caso, lo más importante es establecer las relaciones con todos los elementos 3D.

Sea como sea, esta especie de *convención* que adoptemos, que será interna y para el desarrollo puntual del videojuego en cuestión, no es algo que se refleje directamente en el código, sino en todo el contenido generado. Nos permitirá hacer del entorno 3D algo menos abstracto, más semejante a una realidad, lo cual nos resultará útil, especialmente al equipo no técnico.

En este sentido, les aconsejamos establecer estas coordenadas al comenzar el desarrollo del videojuego en cuestión, haciéndolo lo más natural o más cercano a la realidad posible, sin que esto resulte contraproducente para el desarrollo. Asimismo, no olviden asentarlo en algún documento para que esté al alcance de todos los miembros del equipo. Esto, sin dudas, les ahorrará dolores de cabeza en el futuro.

3. Matrices de transformación

¿Cómo llevamos toda la información del punto anterior directamente a nuestro código? Mediante el uso de las *matrices de transformación.*

3.1. Matriz de mundo o modelo

Cuando desarrollemos nuestros modelos u objetos a utilizar en el entorno 3D, ya sea de forma programática o mediante una herramienta, dichos objetos estarán en coordenadas locales, tendrán una coordenada (0,0,0), o un origen de mundo, y todos los vértices estarán referenciados a este origen local.

Al momento de ubicar este elemento en el mundo 3D, necesitaremos trasladar el objeto de coordenadas locales a coordenada de mundo, mediante la generación de una matriz que contenga transformaciones del objeto (traslación / rotación / escala). Esta *matriz de mundo o modelo* será utilizada cada vez que precisemos trasladar el objeto al mundo, ya sea al realizar el render, chequear colisiones, etc.

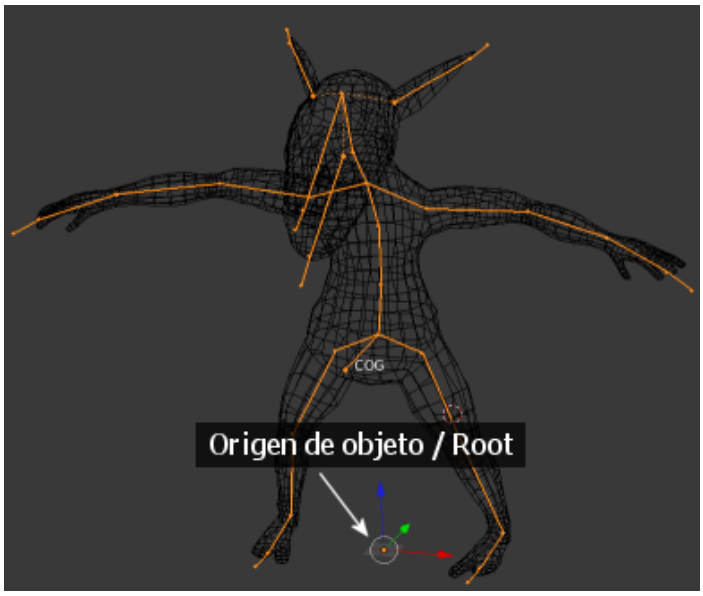

Figura 2. Ejemplo de modelo con coordenadas locales.

En el siguiente código es posible observar la forma en que generamos la matriz de mundo para llevar el modelo de coordenadas locales a coordenadas de mundo:

```
            // Matriz de rotación
            Matrix rotacion = Matrix.CreateFromYawPitchRoll (yaw, pitch,
roll);

            // Matriz de escala
            Matrix escala = Matrix.CreateScale (escalaModelo);

            // Matriz de traslación
            Matrix traslacion = Matrix.CreateTranslation
(posicionModelo);

            // Genero la matriz final
            Matrix world = rotacion * escala * traslacion;
```

3.2. Matriz de vista

En algún momento necesitaremos definir qué parte vamos a ver de ese espacio virtual infinito que denominamos entorno 3D. Este proceso es realizado por la *matriz de vista,* cuya función es transformar las coordenadas de mundo en coordenadas de vista.

La siguiente imagen representa el objeto y la cámara en espacio de mundo:

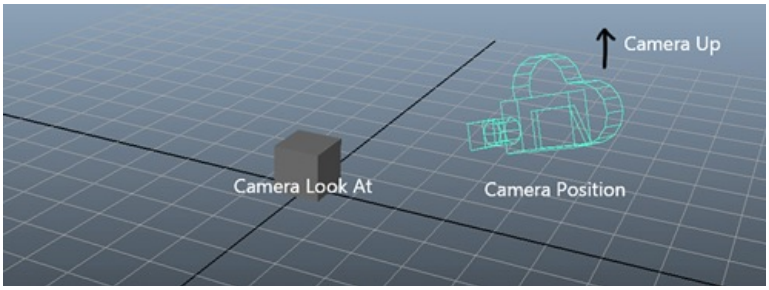

Figura 3. Objeto y cámara en espacio de mundo.

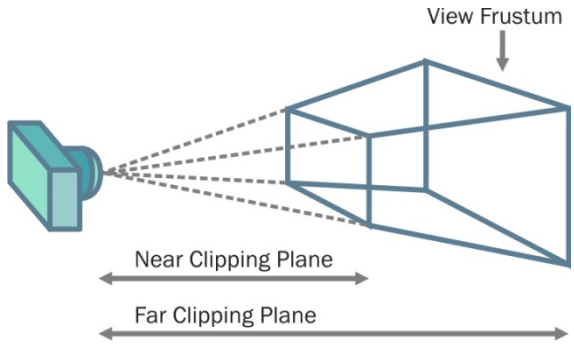

Figura 4. Cámara en espacio de vista.

Para generar la matriz de vista utilizaremos de modo habitual un método *LookAt,* que está presente en prácticamente todos los entornos. La idea es indicarle dónde está la cámara, dónde se encuentra la dirección *arriba* y el lugar hacia el que apunta. Luego, el método se encargará de generar la matriz.

A continuación, presentamos la forma de generar la matriz de vista, basada en información de la cámara:

```
Matrix Vista = Matrix.CreateLookAt(posicionCamara, objetivo,
vectorArriba);
```

3.2.1 Frustum

El **frustum** es una porción de una figura geométrica (usualmente un cono o una pirámide) comprendida entre dos planos paralelos. Las intersecciones del sólido con un plano cortante son las *bases*. El *eje* es el mismo que el del objeto original, si este lo tuviera. Si el eje es perpendicular a la base el sólido es *recto* y en caso contrario, *oblicuo*.

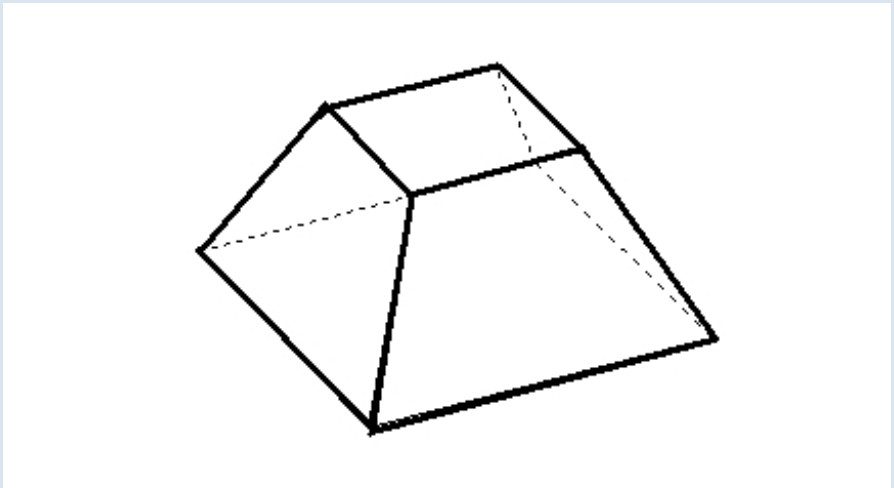

Figura 5. Representación geométrica del frustum.

Se usa como representación del volumen de visualización de una cámara (ya sea de vídeo o de fotografía). En este caso se usa un frustum piramidal con seis planos, los cuales son:

- Plano de recorte lejano (far).
- Plano de recorte cercano (near).
- Plano superior (top).
- Plano inferior (bottom).
- Plano izquierdo (left).
- Plano derecho (right).

En el vértice de la pirámide original se sitúa el observador, el plano de recorte cercano define la mínima distancia de enfoque y el plano de recorte lejano es la máxima distancia de enfoque.

En el ámbito del diseño 3D se utiliza ésta figura geométrica para calcular la parte del escenario virtual que ve la cámara u observador imaginario.

3.2.2 Field of View

El campo de visión (conocido como **field of view** o **field of vision** y abreviado **FOV**) es la extensión del mundo observable que se puede ver en un momento dado.

En los videojuegos se refiere a la parte que se ve del mundo del juego que depende del método de escala utilizada. Esto quiere decir que el FOV puede cambiar dependiendo de la relación de aspecto de la resolución de monitor. Esto es determinado por el método de escala de la imagen usada en el videojuego.

3.3. Matriz de proyección

Una vez aplicadas las anteriores transformaciones, tendremos todo el contenido en coordenadas de vista.

No obstante, aún resta definir cómo transformaremos esas coordenadas 3D en una representación 2D que pueda ser trasladada a nuestro monitor. Esta es la función de la matriz de proyección: proyectar el entorno 3D sobre una superficie 2D.

Nuevamente, todos los entornos disponen de al menos un método para realizar este proceso de forma sencilla.

En XNA contamos con varios. El siguiente es el más usado:

```
Matrix Proyeccion = Matrix.CreatePerspectiveFieldOfView(fov, aspectRatio,
nearPlaneDistance, farPlaneDistance);
```

A continuación, explicaremos brevemente los parámetros.

El primero, conocido como *FOV* (Field of view o campo de visión), es habitualmente un valor cercano a 0.75, pero dependerá del videojuego. Valores mayores generan una sensación de mayor profundidad. Es como si colocásemos un lente de mayor amplitud: captaríamos más área, pero la misma se deformaría progresivamente.

El segundo valor es el *AspectRatio,* el cual debe coincidir con el del viewport que estemos usando para que sea renderizado de forma correcta. Podemos obtenerlo dividiendo la resolución horizontal por la vertical.

El valor *nearPlaneDistance* es la distancia del primer plano paralelo al plano de la cámara que será tomado en cuenta para renderizar. Debe ser un valor mayor a cero; generalmente es 0.1.

El último parámetro es *farPlaneDistance,* que es la distancia del último plano paralelo al plano de la cámara que será considerado como límite de los objetos a renderizar. Así, todo lo que se encuentre por detrás de este plano no será renderizado. Este valor tiene mucha relación con el entorno, lo que queremos mostrar y el valor de FOV; también se puede acortar por motivos de performance. Como sugerencia, pueden utilizar el valor 1000.0 para realizar todas las pruebas.

No pretendemos explicar aquí la lógica que subyace en el uso de matrices, sino más bien el uso práctico de las mismas. En el apartado Referencias les sugerimos algunas lecturas para comprenderlas mejor. Es probable que no les quede totalmente claro la forma de utilizar las matrices, pero estas dudas desaparecerán cuando las veamos en acción, directamente en el código.

4. Triángulos

Una vez que hemos resuelto el tema del espacio virtual mediante el uso de las diferentes matrices, ¿cómo mostramos algo en pantalla?

Realmente no tenemos muchas posibilidades. La Unidad de Procesamiento Visual (GPU por sus siglas en inglés) no aporta demasiado al respecto, ya que sólo puede hacer una cosa: mostrar *triángulos.* El punto es que lo hace extremadamente bien y no existe nada que lo haga mejor.

Así, todo lo que mostremos en pantalla deberá de alguna forma u otra estar representado mediante triángulos, aún los sprites. No obstante, veremos que la GPU también es muy buena enviando píxeles a la pantalla, pero es un proceso posterior y una parte funcional del render de los triángulos. Cabe destacar que también existen métodos alternativos, como el copiado masivo de memoria, pero son atajos funcionales y no el propósito real de la GPU.

La forma de representar un triángulo (que podemos llamar también, de forma más amplia, *polígono*) es a través de tres vértices en el espacio, o sea, tres puntos con coordenadas *X,Y,Z:*

```
Vector3 vertice = new Vector3(x, y, z);
```

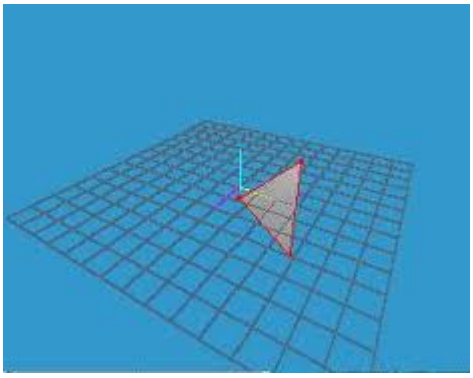

Figura 6. Imagen de un triángulo en el espacio.

Podemos considerar al triángulo como la unidad de nuestros elementos 3D a ser representados y que los mismos siempre estarán formados por vértices 3D en un determinado orden.

La agrupación de estos triángulos dará como resultado una *malla,* la cual es –básicamente– una lista de vértices en grupos de tres, o bien una lista de vértices con una lista de índices que especifica cómo se compone cada triángulo.

Finalmente, una lista de mallas dará como resultado un *modelo,* que es una unidad en sí misma y lo que comúnmente utilicemos, provistos por los artistas.

Desde luego, un modelo no está compuesto solamente por polígonos. A medida que avanzan los videojuegos, los modelos se tornan cada vez más complejos y llevan, además, información sobre cómo mapear una textura (o varias) sobre el polígono, huesos de deformación, color, normales de triángulo para calcular información, entre otros contenidos, según cómo sea renderizado el modelo y con qué propósito.

Así, por ejemplo, un modelo que represente un proyectil que nunca se verá de cerca puede solamente contener información sobre la posición de los vértices y el color. Por el contrario, un personaje complejo tendrá muchísima información, incluyendo un shader (que ya veremos más adelante), el cual realizará el render tomando en cuenta la información del modelo.

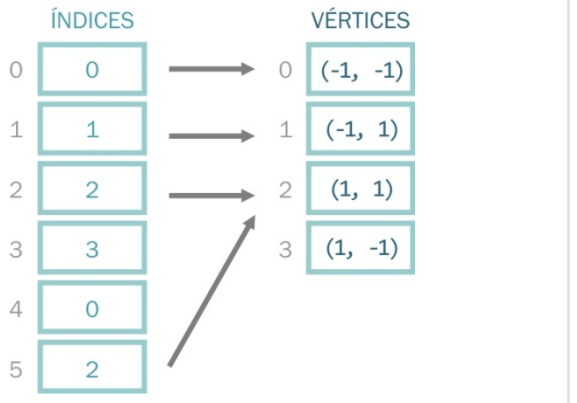

Figura 7. Imagen de una malla compuesta de vértices e índices.

Esta imagen puede representarse en código de la siguiente manera:

```
Vector3[] vertice = new Vector3[4];
int[] indices = new int[6];
```

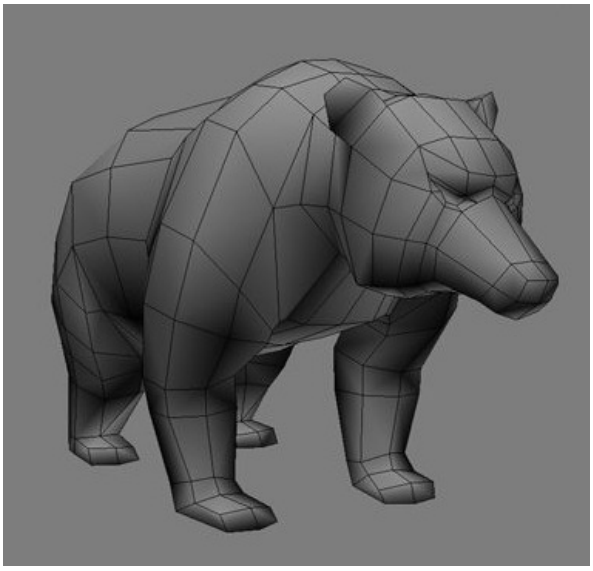

Figura 8. Modelo 3D compuesto de polígonos. (Generalmente, por una cuestión de claridad, en los editores se muestran polígonos de cuatro lados en lugar de los triángulos).

Referencias

Accediendo a los siguientes links podrán profundizar algunos temas que vimos en esta unidad.

API
http://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones

Field of View (FOV)
http://en.wikipedia.org/wiki/Field_of_view

Frustum
http://es.wikipedia.org/wiki/Frustum

Método CreatePerspectiveFieldOfView()
http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.matrix.createperspectivefieldofview.aspx

Sistema de coordenadas de la mano derecha / Right-Hand
http://en.wikipedia.org/wiki/Right-hand_rule

Exportación de modelos de 3D max a Unity
http://digitum.um.es/xmlui/bitstream/10201/22719/1/Memoria_PFC_vFinal_Fernando_Matarrubia.pdf

Herramientas de diseño 3D
http://es.wikipedia.org/wiki/Gr%C3%A1ficos_3D_por_computadora

Hidden surface determination
http://en.wikipedia.org/wiki/Occlusion_culling