Tópicos de Física

Proyecto: Shaders Orientados a 2D

Ingeniería en Producción Multimedia

Romina Verdugo Corral

189085

Unity Shaders

El renderizado en Unity se hace con Materials, Shaders y Textures.

Hay una relación cercana entre Materiales, Shaders y las Texturas en Unity.

- **Materials** son definiciones acerca de cómo la superficie debería ser renderizada, incluyendo referencias a texturas utilizadas, información del tiling (suelo de baldosas), tines de color y más. Las opciones disponibles para un material depende en qué shader del materia está utilizando.
- **Shaders** son scripts pequeños que contienen los cálculos de matemáticas y algoritmos para calcular el color de cada pixel renderizado, basándose en el input de iluminación y la configuración del Material.
- **Textures** son imágenes bitmap. Un material contiene referencias a texturas, para que el shader del Material puede utilizar las texturas mientras calcula el color de la superficie de un objeto. Adicionalmente a el color básico (albedo) de la superficie de un objeto, las texturas pueden representar otros aspectos de la superficie de un material tal como su reflectividad o rugosidad.

Un material especifica un shader en ser utilizado, y el shader utilizado determina qué opciones están disponibles en el material. Un shader especifica una o más variables de textura que espera utilizar, y el inspector del Material de Unity le permite a usted asignar sus propios assets de textura a estas variables de textura.

Para la mayoría del renderizado normal - entendiendo por tal personajes, escenario, entorno, objetos sólidos y transparentes, superficies suaves y duras etc., el Standard Shader por lo usual es la mejor decisión. Este es un shader altamente personalizado el cual es capaz de renderizar muchos tipos de superficie en una manera bastante real.

Hay otras situaciones dónde un diferente shader integrado, o incluso un shader escrito personalizado podría ser apropiado - tal como líquidos, follaje, vidrio de refracción, efectos de partícula, caricaturesco, ilustrativo u otros efectos artísticos, u otros efectos especiales como visión nocturna, visión de calor, o visión de rayos x, etc.

Quizás una de las tareas complejas que puede repercutir en el rendimiento de nuestro videojuego sería la representación de las luces sobre cada una de las superficies, en esta ocasión abordaremos el tema de las luces. A decir verdad, no soy un experto en el tema de lograr una iluminación perfecta, en mi caso prefiero

luces en tiempo real, aunque estas sean de iluminación por vértice, pero entiendo que a veces se puede llegar a una solución intermedia, en mi caso particular no logró un resultado convincente, porque quiero una combinación de luces fijas con luces dinámicas y aun no logro entender bien los light pobres.

El asunto es que por cada luz que ilumine un objeto, este realiza el cálculo una, otra y otra vez por pase de la gráfica, cada pase significa un draw call. Unity nos permite limitar la cantidad de luces que se calcula por pixel, pasado ese límite las siguientes luces se calculan por vértice.
Un diseñador de niveles sabe esto, por eso es que juega bastante con reemplazar luces, las simplifica aumenta la intensidad de una buscando simplificar la escena buscando un equilibrio para que la escena quede bien iluminada sacrificando muy poco rendimiento.

Afortunadamente existe una alternativa y es pre renderizar esas luces en un papel tapiz que luego se le impregna a la maya y da la sensación de ser iluminada, esta técnica se llama mapeo de luces que viene acompañada de una sonda de luz, que completa la ilusión de iluminación en tiempo real.

Los light probes vendrían siendo una red creada para el transporte de luz, fueron creadas para mantener la ilusión de luz dinámica sobre nuestro personaje, ya que el escenario fue precalculado sin él. Esta red red de luz solo afectaría a nuestro personaje y enemigos, calculando el resto de efectos faltantes como la luz sobre nuestro personaje y la sombra proyectada por él. Para hacerlo efectivo hay que distribuir esta red sobre los caminos posibles que nuestro personaje pudiera transitar, esto vendría siendo algo así como para renderizar como se hacía en biohazard, solo que la cámara seria móvil y no estaría prefijada.

Iluminación por vértices en vez de iluminación por píxel.
Unity automatiza el modo en que se representa la luz dependiendo de la distancia sobre todo para luces cónicas o luces puntuales. también elije que luces serán calculadas en un mismo objeto dándole prioridad a las que tiene mayor intensidad.

pero puedes tener en consideración mediante scripts o manualmente que luces son importantes y qué luces son solo para dar ambiente o para enfatizar algo mediante diseño de niveles. las luces por vértice son más fáciles de calcular que las luces por pixel.

Existen otras técnicas que habrá que analizar según el contexto, a veces puede servir un hornado de luces en combinación con luces procedurales que solo iluminan los objetos dinámicos o el prerenderizado de alguna escena como en

resident evil y solo apoyado por colliders, o hacer falsas iluminaciones con un mapa esférico o prescindir totalmente de este usando una estética cartoon.

Como se puede apreciar puede haber distintos enfoques, no solo se limita a las opciones, se puede inventar métodos alternativos, a veces incluso programando falsas luces usando shaders. etc.

Lo importante es poder equilibrar el uso de espacio en disco, memoria y calculos, evitar recargarse solo a un lado de la balanza, a la vez que hay que conservar lo mejor posible la estética.

Desde que comenzó la “revolución 3D” en el ámbito de los juegos de computadora, allá por mediados de la década de los 90’, la tendencia de la tecnología aplicada a este rubro ha sido trasladar el trabajo de procesamiento de gráficos tridimensionales, desde la CPU hacia la tarjeta de video.

En primer lugar fue el filtro de las texturas, para lo cual se crearon chips especialmente dedicados para realizar esta tarea. Así nacieron las famosas placas aceleradoras 3D, que incorporaban dichos chips y un cantidad de memoria propia en la misma tarjeta. Luego, con la salida del GeForce 256 de NVIDIA, el procesador gráfico pasó a encargarse de lo que, hasta ese momento, realizaba la CPU. Estamos hablando de la función de Transformación e Iluminación (Transform & Lighting), utilizada para llevar a cabo los cálculos de geometría y de iluminación general de una escena en 3D. Hubo una versión mejorada de este motor, a la que se llamó de Segunda Generación. Ésta vino incluida a partir de la GeForce 2 y la gama Radeon de ATI, avanzando un poco más en cuanto a materia gráfica.

El gran cambio se dio a partir de la incorporación de los Píxel shaders y Vertex shaders. Esto permitió a los programadores una mayor libertad a la hora de diseñar gráficos en tres dimensiones, ya que puede tratarse a cada píxel y cada vértice por separado. De esta manera, los efectos especiales y de iluminación puede crearse mucho más detalladamente, sucediendo lo mismo con la geometría de los objetos.

Los Shaders

Los shaders usan especificamente OpenGL ES Shading Language. Los detalles de la forma de trabajar de los shaders estan fuera del alcance de este articulo, como tambien la sintaxis del lenguaje de los shaders sin embargo la version corta es que hay 2 tipos de shaders (funciones que se ejecutan en la GPU) que necesitas escribir. shaders de vertices y shaders de fragmentos. Estos son pasados al WebGL como una cadena y compilados para ejecutarse en el GPU.

Shaders de Vértices

La responsabilidad de los Shaders de Vértices es asignar un valor a una variable especial gl_Position para crear los valores del espacio de trabajo (valores entre -1 y +1) en toda la zona del canvas. En nuestro Shader de Vértices de abajo estamos recibiendo valores de un atributo que definiremos como aVertexPosition. Estamos entonces multiplicando esa posición por dos matrices 4x4 que definimos como uProjectionMatrix y uModelMatrix e igualando gl_Position al resultado.

Shaders de Fragmentos

Cada vez que el Shader de Vértices escribe de 1 a 3 valores al gl_Position este también dibujará un punto, una linea o un triángulo. Mientras éste esta dibujando, llamará al Shader de Fragmentos y preguntará: ¿De qué color debería dibujar este pixel? En este caso, simplemente retornaremos blanco cada vez.
gl_FragColor es una variable built-in de GL que es usada para el color del fragmento.
Vertex shader

Un vertex shader es una función que recibe como parámetro un vértice. Sólo trabaja con un vértice a la vez, y no puede eliminarlo, sólo transformarlo. Para ello, modifica propiedades del mismo para que repercutan en la geometría del objeto al que pertenece. Con ésto se puede lograr ciertos efectos específicos, como los que tienen que ver con la deformación en tiempo real de un elemento; por ejemplo, el movimiento de una ola. Donde toma una gran importancia es en el tratamiento de las superficies curvas, y su avance se vio reflejado en los videojuegos más avanzados de la actualidad. Particularmente, en el diseño de los personajes y sus expresiones corporales.

Por otro lado, un Pixel Shader no interviene en el proceso de la definición del "esqueleto" de la escena (conjunto de vértice, wireframe), sino que forma parte de la segunda etapa: la rasterización o render (paso a 2D del universo 3D). Allí es donde se aplican las texturas y se tratan los píxeles que forman parte de ellas. Básicamente, un Pixel Shader especifica el color de un píxel. Este tratamiento individual de los píxeles permite que se realicen cálculos principalmente relacionados con la iluminación en tiempo real, con la posibilidad de iluminar cada pixel por separado. Así es como se lograron crear los fabulosos efectos de este estilo que se pueden apreciar en videojuegos como Doom 3, Far Cry y Half Life 2, por mencionar sólo los más conocidos. La particularidad de los píxel shaders es que, a diferencia de los vertex shaders, requieren de un soporte de hardware compatible. En otras palabras, un juego programado para hacer uso de píxel shaders requiere si o si de una tarjeta de video con capacidad para manipularlos.

Ha habido una evolución lógica de los Shaders en estos últimos años. Este progreso tiene que ver, principalmente, con cuestiones internas de programación. Parámetros como la cantidad de registros disponibles, el número de instrucciones permitidas por programa y la incorporación de instrucciones aritméticas más complejas, entre otros, aumentando la flexibilidad a la hora de programarlos. Tanto OpenGL como DirectX han utilizado sus propios mecanismos de revisión y puesta al día de sus sistemas de shading. Mientras OpenGL se ha basando en el uso de extensiones que se han ido ampliando, DirectX ha definido diferentes modelos de shading (Shader Models), cada uno de estos modelos permite la realización de nuevos efectos y un mayor aprovechamiento del hardware.

Versión de DirectX	Shader Models soportados	Características Shader Models
DirectX 8.0	Vertex Shader 1.0, Pixel Shader 1.1	Primera generación de shaders, punto fijo, rango limitado, pequeños programas asm (fp:8 - vp:128), sin control de flujo.
DirectX 8.1	Vertex Shader 1.1, Pixel Shader 1.3 y 1.4	Primera generación de shaders, punto fijo, rango limitado, pequeños programas asm (fp:8 - vp:128), sin control de flujo.
DirectX 9.0	Vertex Shaders 2.0, Pixel Shader 2.0	Segunda generación de shaders, operaciones en coma flotante, cierto control de flujo, programas mas largos HLSL (fp:96 - vp:256)
DirectX 9.0c	Vertex Shader 3.0, Pixel Shader 3.0	Tercera generación de shaders, control dinámico de flujo, programas largos HLSL (fp:65k - vp:65k)

DirectX 10	Vertex Shader 4.0, Pixel Shader 4.0	Cuarta generación de shaders, control dinámico de flujo, programas largos HLSL (fp:99k - vp:65k), mayor número de registros y texturas.

}

Referencias Bibliográficas

Lammers, K. (2015). *Unity shaders and effects cookbook*. Packt Publishing Ltd.

Pranckevičius, A., & Dude, R. (2018). Physically based shading in Unity. In *Game Developer's Conference*.

Ouazzani, I. (2017). *Manual de creación de videojuego con Unity 3D* (Master's thesis).

Zucconi, A., & Lammers, K. (2016). *Unity 5. x Shaders and Effects Cookbook*. Packt Publishing Ltd.

Dean, J. (2016). *Mastering Unity Shaders and Effects*. Packt Publishing Ltd.

Ciborro Montes, A. (2019). Renderizado de funciones de distancia mediante Raymarching en Unity.

Halladay, K. (2019). Writing Shaders in Unity. In *Practical Shader Development* (pp. 299-325). Apress, Berkeley, CA.

Iseki, F., Tate, A., Mizumaki, D., & Suzuki, K. (2017). OpenSimulator and Unity as a Shared Development Environment. *Journal of Tokyo University of Information Sciences*, *21*(1).

Thorn, A., Doran, J. P., Zucconi, A., & Palacios, J. (2019). *Complete Unity 2018 Game Development: Explore techniques to build 2D/3D applications using real-world examples*. Packt Publishing Ltd.

Garrido Suárez, C. D. (2019). *Sistema de posicionamiento y rotación 3D virtual* (Doctoral dissertation).

Mateus Romero, F. (2015). Diseņo, desarrollo, modelado y animación de un juego con Unity3D.

Ferrando Monzón, G. (2016). *GameBrush. Plugin de generación de formas abstractas y de alteración visual del entorno para Unity3D* (Doctoral dissertation).

Doppioslash, C. (2018). Your First Unity Shader. In *Physically Based Shader Development for Unity 2017* (pp. 17-32). Apress, Berkeley, CA.

Doppioslash, C. (2018). Your First Unity Lighting Shader. In *Physically Based Shader Development for Unity 2017* (pp. 51-64). Apress, Berkeley, CA.

Doppioslash, C. (2018). How Shader Development Works. In *Physically Based Shader Development for Unity 2017* (pp. 3-16). Apress, Berkeley, CA.

Alapont Granero, S. (2015). *Scares for sale: diseño y desarrollo de un videojuego en Unity 3D. Audio e introducción a Leap Motion* (Doctoral dissertation).