Simulación de Materiales Naturales mediante Texturas Volumétricas Procedurales

Índice de contenido

1. Introducción

1.1. Motivación

Hoy en día los gráficos 3D de la mayoría de videojuegos y de películas de animación emplean todo tipo de técnicas para adaptarse a una creciente demanda de realismo virtual.

La mayoría de texturas basadas en imágenes o fotografías estáticas no aportan suficiente variedad para cubrir el suelo de madera de una habitación grande o un pasillo. Eventualmente alguien acaba por darse cuenta del patrón de repetición de las texturas.

Últimamente la potencia de las *GPU* de los ordenadores ha ido en aumento y con ello la capacidad de visualizar modelos y materiales cada vez más complejos. Ahora un ordenador personal e incluso un móvil puede disponer de un procesador gráfico lo suficientemente potente como para generar texturas procedurales en tiempo real.

Es importante producir software que utilice estas nuevas tecnologías para tener unos resultados más interesantes de cara a la sociedad actual.

1.2. Objetivos

El objetivo de este proyecto es diseñar e implementar un programa de *shader* capaz de generar una textura volumétrica que simula la madera del tronco de los árboles.

El objetivo inicial es que el *shader* pueda ejecutarse en tiempo real, si no es capaz de cumplir los mínimos en tiempo real, se diseñará como un *shader* de *renderizado* en diferido.

Entre los objetivos implícitos para desarrollar este proyecto cuentan los siguientes: adquirir conocimientos relativamente recientes sobre las tecnologías existentes relacionadas con el tema de las texturas procedurales 3D y aplicarlos de forma conveniente; elegir una herramienta de desarrollo adecuada para implementar el *shader* y aprender a usarla.

1.3. Planificación

El proceso de desarrollo del proyecto ha supuesto un total de 323 horas.

Los procesos de gestión han consumido 33h, entre planificación (3h) y reuniones (30h).

El proceso de aprendizaje ha requerido 87h. Existen muchos conocimientos específicos necesarios para poder terminar el proyecto.

El proceso de desarrollo del shader ha necesitado un mínimo de 157h, repartidas entre el diseño (57h), implementación (70h) y pruebas y corrección de errores (30h).

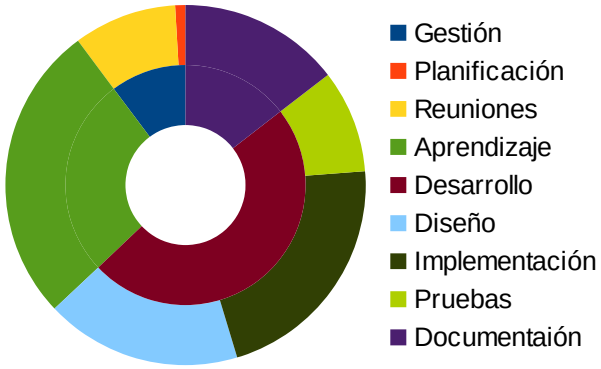

Figura 1: Diagrama de tiempo

El proceso de redacción de la documentación lleva consumidas más de 46h en total.

1.4. Contenido de la Memoria

Este documento recoge los detalles más importantes en lo referente al proyecto sobre el que trata.

En el siguiente capítulo comienza con una introducción a las tecnologías actuales para el desarrollo de texturas procedurales 3D.

Antes de presentar el diseño del proyecto hay un capítulo para explicar una serie de conceptos clave necesarios para entender los contenidos de la memoria.

A continuación se detalla el diseño y la implementación del shader. La estructura del capítulo de implementación está adaptada para seguir un esquema parecido al capítulo del diseño.

Después de analizar los resultados, hay una lista de referencias bibliográficas y un pequeño glosario.

2. Estado del Arte

En este capítulo se detalla el estado del arte en lo referente a *texturas volumétricas*. Se emplea un vocabulario muy especializado. Para saber más sobre ciertos nombres véase el glosario adjunto al final del documento.

2.1. Texturas

Las *texturas* son imágenes que se emplean en el mapeado de texturas, método por el que una o varias imágenes (*texturas*) se aplican a gráficos generados por ordenador o a la superficie de un *modelo 3D*. Esto aporta información del color, y de otros efectos de textura en la superficie de los mismos [1]. Entre dichos efectos, se pueden citar: la rugosidad, la especularidad, la reflectividad, la emisividad, etc.

2.1.1. Texturas Rasterizadas

Este tipo de *texturas* son las más utilizadas. Se parte de un mapa de bits o *raster* y se mapean una o varias de estas imágenes, total o parcialmente a la superficie del *modelo 3D*.

Un *raster* es una matriz de puntos que representan píxeles o *texels*. Normalmente es una matriz 2D con una distribución rectangular de sus puntos [2]. Cada punto contiene la información del color correspondiente a cada punto, aunque se puede tratar como otro tipo de información.

El mapeado de texturas es el proceso de conversión de las coordenadas UV de los puntos en la *textura* 2D a las coordenadas XYZ de los puntos en la superficie del *modelo 3D*. Esto se puede realizar mediante una asignación manual de los puntos correspondientes a los vértices del modelo, una asignación similar realizada mediante un software de desarrollo plano o un tipo de *proyección de textura*.

Existen varios **tipos de proyección geométrica**. El más sencillo consiste en una proyección planar de la textura en dirección perpendicular al plano sobre la superficie del *modelo* en la que se desa aplicar. Por otra parte, existen proyecciones con corrección de perspectiva, denominadas proyecciones cónicas, que simulan la proyección de una imagen desde un foco situado en un punto del espacio.

De igual forma pueden realizarse distintos tipos de proyecciones dependiendo del volumen primitivo utilizado como *frustum* de la proyecciñon. La proyección ortoédrica o paralela, proyecta imágenes desde las 6 caras (pueden ser la misma o distintas). La proyección esférica usa la superficie curva de una esfera que envuelve al *modelo*. Las proyecciones cilíndrica y cónica son una combinación de proyecciones planares para las bases y una proyección curva.

2.1.2. Texturas Volumétricas

"*Una **textura volumétrica** es una textura en 3 dimensiones. Se puede construir como una secuencia ordenada de texturas 2D (mapas de bits) o como una textura procedural 3D. Cuando se realiza un corte a un objeto con una textura volumétrica, las nuevas superficies también disponen de una textura precisa*" Traducido de *ComputerDesktopEncyclopedia*.

El primer método describe una matriz 3D de puntos llamados voxels. Requiere una gran cantidad de memoria, ya que se trata de una lista de elementos de tipo *raster*. Este método se usa sobre todo para *renderizar* volúmenes complicados o que requieren gran precisión. Se utiliza por ejemplo en la *tomografía axial computerizada* (*TAC*), en la visualización de fluidos, o para visualizar cualquier volumen al que se le puedan realizar cortes que no están predefinidos.

El segundo método es la utilización de una textura procedural 3D, sirve para visualizar materiales (sólidos y fluidos) con texturas sencillas que no requieren gran precisión. El resultado de un mismo volumen con una textura generada por un mismo algoritmo puede diferir según el motor que se utiliza. Se puede alcanzar un mayor nivel de complejidad cuando se utilizan varios materiales.

2.1.3. Texturas Procedurales

Una *textura procedural* es una imagen creada por medio de un algoritmo. Normalmente se utilizan para crear una representación realista de una superficie o un volumen de elementos naturales tales como madera, mármol, granito, metal, piedra, etc. y, posteriormente, mapear la *textura* (2D ó 3D) al *modelo 3D* [3].

Habitualmente el aspecto natural de estas *texturas* se logra mediante el uso de funciones pseudoaleatorias que simulan el componente aleatorio que presentan los materiales naturales.

2.2. Ruido de Perlin

Perlin (1985) describió un lenguaje completo para generar *texturas procedurales* y estableció los cimientos para el tipo más popular de *texturas procedurales* actuales [4].

"*En el ruido de Perlin se determina el ruido en cada punto del espacio como un gradiente pseudoaleatorio. Se divide el espacio en una matriz 3D compuesta de cubos idénticos de lado la unidad donde todos los vértices tienen todas sus coordenadas con valores enteros. En cada cubo se interpola a partir de los valores de sus 8 vértices para obtener un valor final. El componente pseudoaleatorio se calcula creando un índice a partir de las coordenadas de cada punto de la matriz que corresponde a un elemento de un vector de vectores unitarios de dirección pseudoaleatoria. El conjunto de gradientes consiste en los 12 vectores que van desde el centro del cubo a sus aristas. La función interpoladora es una curva polinómica de quinto grado, que permite una derivada continua de la función de ruido.*" Traducción de *State of the Art in Procedural Noise Functions*.

2.2.1. Implementación de Zucker

Matt Zucker hace una buena observación respecto a la complejidad computacional del ruido de Perlin. Se trata de la siguiente: "*Si se quiere extender la función de ruido a n dimensiones, es necesario tener en cuenta* 2^n *puntos de la cuadrícula y realizar* $2^{(n-1)}$ *sumas ponderadas. La implementación que se presenta aquí tiene una complejidad computacional de* $O(2^{(n-1)})$*, lo que significa que se va a ralentizar mucho si quieres utilizar ruido de 5, 6 ó más dimensiones*" Matt Zucker [5].

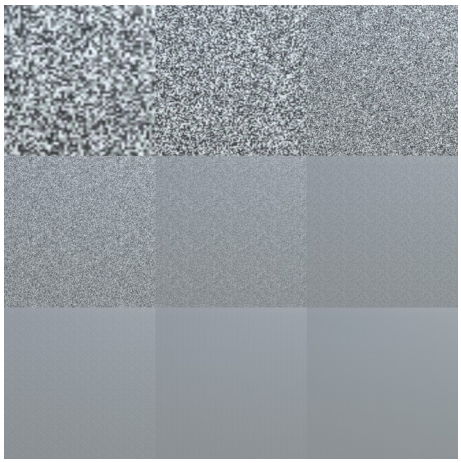

Figura 2: Ruido blanco. Comparativa de niveles de aumento desde x4 (arriba a la izquierda) hasta x1/64 (abajo a la derecha).

Explica que el rendimiento del algoritmo depende en gran medida del número de dimensiones en el que se calcula. Para *n* dimensiones en cada punto hay que considerar 2^n puntos de la matriz y realizar $2^{(n-1)}$ sumas ponderadas. En resumen, se trata de un algoritmo de complejidad $O(2^{(n-1)})$.

También propone algunas mejoras para agilizar el algoritmo. La mejora esencial consiste en aprovechar una de las características del ruido de Perlin (y del ruido en general). El ruido es localmente variable, pero globalmente plano. Propone repetir el patrón pseudoaleatorio de modo que no se note, ya que cuando se está lo suficientemente alejado como para percibir el patrón, no resulta importante porque el ruido es casi completamente plano (Ver Figura 2).

El "truco", según lo denomina Zucker, consiste en tener un vector de gradientes pseudoaleatorios precalculados y otro con los índices del mismo. Recomienda utilizar valores del 0 al 255 para indexar 256 elementos. Se emplea una función que transforma las coordenadas de un punto en un índice para el vector de gradientes.

Este "truco" lo menciona Perlin en su página web y ya lo utilizaba en su implementación original, Zucker sólo explica por qué Perlin lo emplea y los fundamentos en los que se basa al hacerlo.

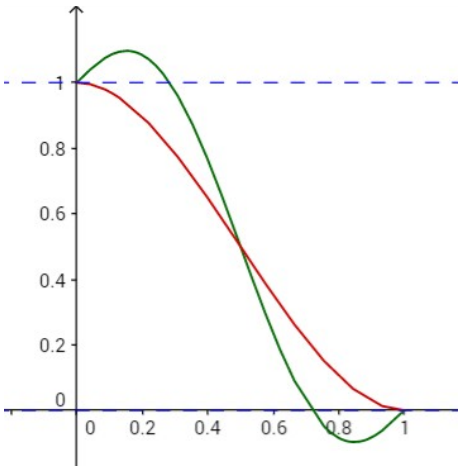

Figura 3: Al asignar pendiente 0 en los extremos, la función no se sale del rango.

Matt Zucker también simplifica el cálculo del polinomio de quinto grado bajo la siguiente premisa: En lugar de calcular una curva que mantenga las derivadas de la función de ruido, calcula una interpolación denominada "*ease-in-out*" que cumple que la derivada en sus extremos es 0. La curva es una interpolación polinómica de grado 3, lo cual facilita el cálculo. Si el valor de ruido pseudoaleatorio es un número real en el intervalo [0,1], esta curva también tiene todos sus valores acotados en dicho intervalo (Ver Figura 3).

Como cada cubo tiene de longitud de sus lados la unidad, el dominio de la función siempre se puede expresar como [0,1]. Esto permite precalcular el polinomio, en lugar de calcularlo 7 veces por cubo.

Con esta mejora, se pasa de tener que resolver un sistema de 6x6 7 veces por cada cubo visible a resolver uno de 4x4 una vez y utilizar una lerp 7 veces por cada cubo visible.

3. Conceptos Previos al Diseño

Antes de leer el diseño es recomendable tener unos conocimientos mínimos referentes a shaders e interpolaciones.

3.1. Shaders

Un *shader* es un programa informático que se emplea para producir los niveles apropiados de color en una imagen [6]. Actualmente también se utilizan para agregar efectos especiales o posprocesamiento de vídeo. Este programa, normalmente, se ejecuta en una unidad de *hardware* llamada *GPU* (*Graphics Processing Unit*). En su defecto puede ser emulada por la *CPU* (*Central Processing Unit*), pero con un rendimiento menor.

Clásicamente se dividen en tres categorías: *shaders* 2D, 3D y otros [6]. Esta división correspondía a una división física en las GPU antiguas, aunque en las actuales ya no se realiza esta separación. El tipo de *shader* que se va a utilizar en este proyecto corresponde a la categoría de *shaders* 3D.

El modelo clásico de *shader* se denomina *vertex shader*. A lo largo del tiempo con cada actualización de la capa de gráficos (G*raphics Layer* o *GL*) los *modelos de shader* han ido evolucionando. Con *OpenGL* 3.2 y *Direct3D* 10 se añadió el modelo *geometry shader*. Actualmente el modelo más reciente añadido es el denominado *tesellation shader* que ha sido introducido con *OpenGL* 4.0 y *Direct3D* 11. Para este proyecto es suficiente con emplear el modelo clásico de forma que es compatible con cualquier *GPU* actual.

3.1.1. Vertex Shader / Fragment Shader

El *vertex shader* es el tipo de *shader* 3D más arraigado y común. El programa se ejecuta una vez por cada vértice de cada *modelo 3D* en la *escena*. Su función es transformar la posición de cada vértice de un espacio virtual en 3D a un punto 2D perteneciente a la pantalla con la que se visualiza [6].

Para entender el funcionamiento de un *vertex shader* hay que revisar el concepto de *tubería de renderizado* (*Graphics Pipeline*). Esto es la secuencia de pasos que se siguen para crear una imagen *raster* 2D a partir de una *escena* 3D[7].

El **primer paso** es pasar las coordenadas de cada modelo del sistema local al global. Los *modelos 3D* se agrupan en la escena formando una estructura de árbol. La *escena* (o mundo) es la raíz del árbol y en cada nodo hay un objeto virtual 3D con una matriz de transformación asociada. Las hojas normalmente contienen la mayor parte de los *modelos geométricos*. Para calcular la matriz de transformación global de un nodo o una hoja hay que multiplicar todas las matrices desde la raíz hasta el nodo.

El **segundo paso** es pasar del sistema de coordenadas global al sistema de coordenadas local de la cámara. A menudo las cámaras son objetos pertenecientes al mismo árbol de objetos. Por lo tanto, se emplea el mismo proceso pero el camino inverso (desde el nodo de la cámara hasta la raíz) para calcular la matriz de transformación al sistema local de la cámara.

El **tercer paso** es aplicar la matriz de transformación de la *perspectiva* de la cámara (si no se trata de una proyección ortogonal).

En el **cuarto paso** se descartan todos los vértices que se encuentran fuera del *frustum* de la cámara, excepto aquellos que pertenecen a una arista que se encuentra dentro. Esto permite reducir la complejidad de la escena en los próximos pasos.

El **quinto paso** es calcular para cada vértice el color del material, la cantidad de luz que recibe y emite y las normales asociadas a cada superficie de aquellas a las que pertenece. Es el último paso del *vertex shader*. Toda esta información se almacena y pasa de forma ordenada a otro programa llamado *fragment shader*.

El **último paso** es calcular a cada *píxel* que color le corresponde. El *fragment shader* tiene la información de los vértices que corresponden a la superficie del fragmento que se está *renderizando*. Normalmente para muchos de estos datos (color, textura, transparencia, etc.) se hace una interpolación para calcular sus valores específicos en el punto que correspondería al píxel que se está calculando.

Estos son todos los pasos de la *pipeline* de un *vertex*/*fragment shader*. Por supuesto otros *modelos de shader* más nuevos disponen de una *pipeline* ligeramente diferente o más complicada.

3.1.2. Shading Language

El *shading language* es el lenguaje de programación orientado a gráficos y adaptado para programar los efectos del *shader* [8]. Resulta ser muy especializado y tiene tipos específicos para identificar puntos, vectores, colores, matrices etc. Hay dos grandes grupos según el tipo de proceso de *renderizado* se utilice el *shader*: el proceso puede ser *en tiempo real* o *en diferido*.

El proceso de ***renderizado en diferido*** suele ser mucho más exhaustivo y con resultados mucho mejores visualmente. Este tipo se utiliza únicamente para ilustraciones y producción de fotogramas de películas o aquellos casos en los que se dispone de bastante tiempo para *renderizar* cada imagen. En algunas ocasiones el proceso es tan intensivo que se emplean *clusters* de ordenadores interconectados para repartirse la carga de trabajo de *renderizar* cada imagen.

El proceso de ***renderizado en tiempo real*** no dispone de tanto tiempo ya que normalmente la *tasa de refresco* que necesita una imagen para no causar artefactos visuales es superior a 24fps (fotogramas por segundo). De modo que los *shaders* de este tipo requieren ser menos exigentes en sus resultados visuales y estar programados con un código bastante optimizado. Este tipo se usa más frecuentemente porque permite ver los resultados en el momento con una GPU normal. Además para trabajar con los modelos 3D que se utilizan en películas y en imágenes, normalmente, es necesario poder visualizarlo primero en tiempo real. Los editores de modelado, pintura y animación 3D y los videojuegos emplean este tipo de *shader*.

3.2. Interpolaciones

En el ámbito del análisis numérico, se denomina *interpolación* a la obtención de nuevos puntos partiendo del conocimiento de un conjunto discreto de puntos [9].

Para todos los ejemplos de esta sección se va a utilizar el mismo conjunto de puntos:

x	f(x)
0	0,24
1	-0,25
2	1,0
3	0,48
4	-0,87

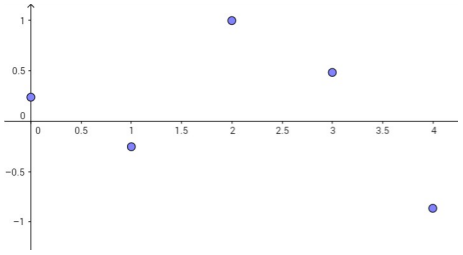

Figura 4: Conjunto de puntos a partir de los datos del ejemplo. Gráfica obtenida con GeoGebra.

3.2.1. Interpolaciones Lineales

El término "***Lerp***" viene del inglés "***L****inear* ***inter****polation*" (*Inter*polación *L*ineal). Se trata de un tipo de *interpolación* polinómica de primer *grado*. La *interpolación* se calcula entre cada par de puntos adyacentes $(A(x_a,y_a),B(x_b,y_b))$ de acuerdo con la siguiente ecuación:

$y=y_a+(y_b-y_a)\frac{x-x_a}{x_b-x_a}$ en el punto (x,y) [10]

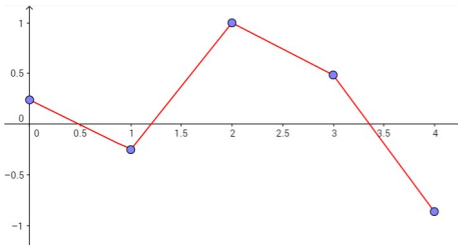

Figura 5: Interpolación lineal (rojo) del conjunto de puntos (azul).

Al interpolar un conjunto de puntos, hay que calcular una función segmentada. Los extremos de cada tramo $[x_a,x_b]$ deben ser adyacentes, es decir que cumplir que no existe ningún x_k perteneciente al dominio de los datos y que además pertenece al intervalo $[x_a,x_b]$.

Al unir dos tramos $[x_a,x_b]$ y $[x_b,x_c]$ que comparten un punto en común tenemos que elegir entre la imagen de dos funciones en dicho punto (x_b). En este caso se puede elegir cualquiera de las funciones de ambos tramos, ya que ambas coinciden en dicho punto.

La interpolación lineal de un conjunto de puntos tiene una característica más en los puntos comunes: son puntos de discontinuidad en su primera derivada. Esto quiere decir que en esos puntos la función presenta cambios bruscos en su pendiente (Ver Figura 5).

3.2.2. Interpolaciones Polinómicas

La interpolación lineal es un caso particular de una interpolación polinómica, concretamente una que utiliza un polinomio de grado 1.

Unas de las funciones matemáticas más sencillas son las *funciones polinómicas* (su expresión es un polinomio). Un polinomio es una expresión que consiste en una suma de variables y sus coeficientes [11]. Consiste en un sumatorio de elementos de la forma $a_i x^i$ siendo $i\in\mathbb{R}:i\geq 0$ un entero no negativo y $a_i\in\mathbb{R}$ un coeficiente real (puede ser 0).

Las interpolaciones polinómicas pueden cumplir todas las características que se les exijan en cada punto del conjunto de datos. El polinomio se calcula mediante una serie de ecuaciones (una por cada restricción).

Para que la interpolación $p(x)$ pase por el punto (x_a,y_a), se debe cumplir que $p(x_a)=y_a$.

Para que la pendiente en ($x=x_a$) sea m_a, se debe cumplir que $p'(x_a)=m_a$.

Y esto se puede aplicar para derivadas sucesivas.

No es necesario aportar ni toda ni la misma información en cada punto del dominio de los datos. Incluso es posible omitir totalmente la información perteneciente a un punto.

Con una ecuación se requiere un polinomio de grado 0, por lo tanto esta información no puede ser de una derivada, o si no el sistema no tiene solución. Con cada ecuación que se utiliza, se incrementa el grado del polinomio. Para un sistema de 5 ecuaciones, como el que se necesita para calcular un polinomio que pase por los 5 punto del conjunto de datos, se necesita un polinomio de grado 4. De la misma manera, para *n* ecuaciones, se necesita un polinomio de grado ($n-1$).

El sistema de ecuaciones del ejemplo es el siguiente:

$$p(0)=y_0=0{,}24$$
$$p(1)=y_1=-0{,}25$$
$$p(2)=y_2=1$$
$$p(3)=y_3=0{,}48$$
$$p(4)=y_4=-0{,}87$$

Se necesita un polinomio de grado 4, donde los coeficientes ($a_0..a_4$) aún no están determinados:

$$p(x)=a_0+a_1x+a_2x^2+a_3x^3+a_4x^4$$
$$a_0=y_0=0{,}24$$
$$a_0+a_1+a_2+a_3+a_4=y_1=-0{,}25$$
$$a_0+2a_1+2^2a_2+2^3a_3+2^4a_4=y_2=1$$
$$a_0+3a_1+3^2a_2+3^3a_3+3^4a_4=y_3=0{,}48$$
$$a_0+4a_1+4^2a_2+4^3a_3+4^4a_4=y_4=-0{,}87$$

Se puede representar en forma de ecuación matricial:

$$\begin{pmatrix}1&0&0&0&0\\1&1&1&1&1\\1&2&2^2&2^3&2^4\\1&3&3^2&3^3&3^4\\1&4&4^2&4^3&4^4\end{pmatrix}\begin{pmatrix}a_0\\a_1\\a_2\\a_3\\a_4\end{pmatrix}=\begin{pmatrix}y_0\\y_1\\y_2\\y_3\\y_4\end{pmatrix}$$

Para calcular los coeficientes, se puede resolver la ecuación matricial:

$$\begin{pmatrix}a_0\\a_1\\a_2\\a_3\\a_4\end{pmatrix}=\begin{pmatrix}1&0&0&0&0\\1&1&1&1&1\\1&2&2^2&2^3&2^4\\1&3&3^2&3^3&3^4\\1&4&4^2&4^3&4^4\end{pmatrix}^{-1}\begin{pmatrix}y_0\\y_1\\y_2\\y_3\\y_4\end{pmatrix}\approx\begin{pmatrix}0{,}24\\-3{,}65\\4{,}67\\-1{,}7\\0{,}19\end{pmatrix}$$

$$p(x)=0{,}24-3{,}65x+4{,}67x^2-1{,}7x^3+0{,}19x^4$$

En el espacio 2D y 3D las interpolaciones polinómicas se calculan multiplicando los polinomios correspondientes a cada eje:

$$p(x)=a_0+a_1x+a_2x^2$$
$$q(y)=b_0+b_1y+b_2y^2$$
$$r(z)=c_0+c_1z+c_2z^2$$

$$pq(x,y)=p(x)q(y)=$$
$$a_0b_0+a_0b_1y+a_0b_2y^2$$
$$+a_1b_0x+a_1b_1xy+a_1b_2xy^2$$
$$+a_2b_0x^2+a_2b_1x^2y+a_2b_2x^2y^2$$

$$
\begin{aligned}
pqr(x,y,z) = p(x)q(y)r(z) = \\
& a_0 b_0 c_0 && + a_0 b_0 c_1 z && + a_0 b_0 c_2 z^2 \\
+\ & a_0 b_1 c_0 y && + a_0 b_1 c_1 y z && + a_0 b_1 c_2 y z^2 \\
+\ & a_0 b_2 c_0 y^2 && + a_0 b_2 c_1 y^2 z && + a_0 b_2 c_2 y^2 z^2 \\
+\ & a_1 b_0 c_0 x && + a_1 b_0 c_1 x z && + a_1 b_0 c_2 x z^2 \\
+\ & a_1 b_1 c_0 x y && + a_1 b_1 c_1 x y z && + a_1 b_1 c_2 x y z^2 \\
+\ & a_1 b_2 c_0 x y^2 && + a_1 b_2 c_1 x y^2 z && + a_1 b_2 c_2 x y^2 z^2 \\
+\ & a_2 b_0 c_0 x^2 && + a_2 b_0 c_1 x^2 z && + a_2 b_0 c_2 x^2 z^2 \\
+\ & a_2 b_1 c_0 x^2 y && + a_2 b_1 c_1 x^2 y z && + a_2 b_1 c_2 x^2 y z^2 \\
+\ & a_2 b_2 c_0 x^2 y^2 && + a_2 b_2 c_1 x^2 y^2 z && + a_2 b_2 c_2 x^2 y^2 z^2
\end{aligned}
$$

[1]

3.2.3. Splines

En matemáticas se denomina "***Spline***" a una *función segmentada*. Cada *tramo* es una *función polinómica* que conecta dos puntos de datos. La función en conjunto no presenta cambios bruscos de *pendiente* [12].

Habitualmente cada *tramo* está definido por una *función polinómica* de *grado* 3 ó 5. Según si en cada *tramo* se dispone de 4 ó 6 datos respectivamente. Estas interpolaciones se denominan *interpolación cúbica* e *interpolación quíntica* respectivamente.

$$c(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$
$$q(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5$$

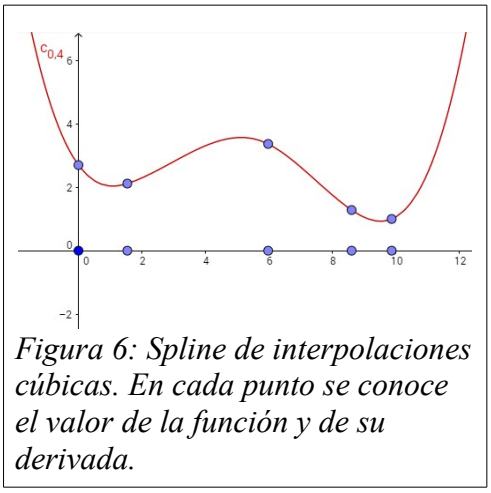

Figura 6: Spline de interpolaciones cúbicas. En cada punto se conoce el valor de la función y de su derivada.

Los datos de una *interpolación cúbica* corresponden al valor de la función ($f(x_a)$ y $f(x_b)$) y su *derivada* ($f^I(x_a)$ y $f^I(x_b)$) en ambos extremos del tramo $[x_a, x_b]$, mientras que los de la *interpolación quíntica* incluyen la segunda *derivada* ($f^{II}(x_a)$ y $f^{II}(x_b)$).

1 Los colores rojo, verde y azul, indican los ejes X, Y, Z respectivamente.

4. Diseño

El problema a resolver se trata de diseñar un *shader* capaz de generar proceduralmente una textura 3D que aplicada sobre cualquier modelo le da un aspecto de estar hecho de madera. La madera dispone de una gama de tonos que varía según su procedencia. El tronco de un árbol, al seccionarlo de forma perpendicular al eje de crecimiento (vertical), muestra una serie de anillos con forma parecida a coronas circulares con mayor o menor grado de distorsión.

4.1. Textura volumétrica

Para empezar el diseño, se necesita un *shader* que produzca el efecto de unos anillos cilíndricos alrededor de un eje. Para simular los anillos de un árbol se parte del eje Y (vertical).

Los anillos son una serie de coronas cilíndricas que parten del eje, su radio aumenta a medida que se alejan del eje, pero no lo hace su grosor.

Partiendo de un sistema de coordenadas cartesianas se pasa a un sistema mixto: una mezcla de coordenadas cartesianas y polares. En el eje Y se utilizan las coordenadas cartesianas, mientras que en el plano XZ se utilizan las coordenadas polares.

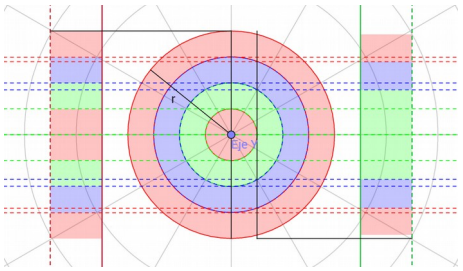

Figura 7: Secciones de los anillos. En los planos (x=0), (y=0) y (x=1) de izquierda a derecha.

$$\mathbb{R}^3 \to \mathbb{R}\times[0,\infty)\times[0,2\pi)$$
$$(x,y,z)\to(y,r,\alpha)$$

El dato del ángulo no es necesario para representar los anillos. De este modo, se puede omitir, sólo es necesario calcular la distancia al eje Y (el radio).

$$(x,y,z)\to(y,r=\sqrt{x^2+z^2})$$

Este *shader* también tiene la responsabilidad de establecer el color de cada punto del volumen. El color depende solamente del radio, el grosor de los anillos y los colores de la zona más clara y más oscura. Tanto el grosor, como los colores son valores constantes, de modo que la única variable es el radio.

Como una primera aproximación se puede visualizar como una *lerp* que genera un degradado de color a partir de los colores de ambos extremos. El dominio de la variable t es [0, 1), de modo que cuando t=0, el color resultante es el más claro y cuando t=1, el color resultante es el más oscuro.

$$C(t)=C_0(1-t)+C_1 t$$

A partir de esta aproximación se distorsiona el dominio de t para que el color más parecido a C_0 (color interior más claro) llene más área que el similar a C_1 (color exterior más oscuro).

Para diseñar esta función de distorsión del degradado $g:[0,1]\to[0,1]$ hay que cumplir lo siguiente:

- El valor de la función en t=0 es 0 y en t=1 es 1: $g(0)=0\wedge g(1)=1$
- La función es continua en todo su dominio [0, 1]
- La pendiente en t=0 es 0: $g^I(0)=0$

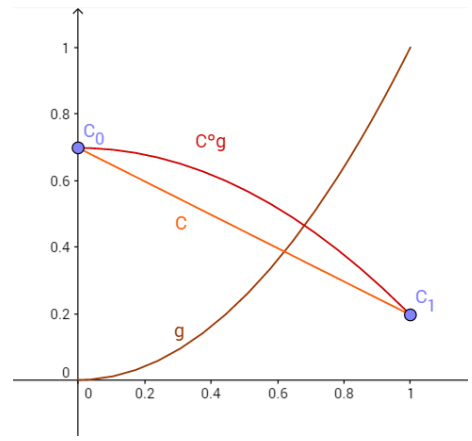

Figura 8: Superposición de las funciones C (naranja), g (marrón) y $C\circ g$ (rojo) en el intervalo [0,1]. Las funciones C y $C\circ g$ generan un degradado lineal y cuadrático respectivamente entre C_0 y C_1.

Se dispone de 3 datos, que sirven para calcular un polinomio de segundo grado:

$$g(t\in[0,1])=a_0+a_1t+a_2t^2$$

También se necesita la primera derivada:

$$g^I(t)=a_1+2a_2t$$

El sistema de tres ecuaciones y tres incógnitas a resolver es el siguiente:

$$\begin{cases} g(0)=a_0=0 \\ g(1)=a_0+a_1+a_2=1 \\ g^I(0)=a_1=0 \end{cases}\to\begin{cases} a_0=0 \\ a_1=0 \\ a_2=1 \end{cases}$$

Tras resolver el sistema se obtiene $g(t)=t^2$.

La composición $C\circ g$ da como resultado una función acorde con el efecto que se busca (Ver Figura 8).

El último paso es utilizar la función $C\circ g$ en todo el dominio del radio $r\in[0,\infty)$ repitiéndose en tramos del grosor indicado.

Se necesita una función $w:[0,\infty)\to[0,1]$ que repita su comportamiento con un periodo T. Dicha función debe tener un comportamiento lineal creciente con valores desde 0 hasta 1. Existe un punto de discontinuidad cada vez que se cambia de anillo, cuando $x=kT$ siendo $k\in\mathbb{Z}$.

$$\nexists\lim_{x\to kT} w(x)\Rightarrow\begin{cases}\lim\limits_{x\to(kT)^+} w(x)=0\\ \lim\limits_{x\to(kT)^-} w(x)=1\end{cases}$$

La función $f(x)=x\,mod\,1$, siendo mod la operación módulo (resto de la división entera), cumple con esta característica para $T=1$. El dividendo es *x*, la recta $y=x$ pasa por los puntos (0,0) y (1,1). En lugar de pasar por el punto (1,1), se requiere que la recta del dividendo pase por el punto $(T,1)$.

La función debe ser de la forma $w(x)=d(x)\,mod\,1$, siendo $d(x)$ el dividendo que cumple:

- *d* es una función lineal: Función lineal: $d(x)=a_0+a_1x$
- pasa por el punto (0,0): $d(0)=a_0=0$
- pasa por el punto (T,1): $d(T)=a_0+a_1T=1$

$$\left.\begin{matrix}a_0=0\\ a_0+a_1T=1\end{matrix}\right\}\Rightarrow a_1=T^{-1}\Rightarrow d(x)=T^{-1}x\Rightarrow w(x)=\frac{x}{T}\,mod\,1$$

Finalmente, si la salida de w se utiliza como entrada de $C\circ g$, tenemos la función $C\circ g\circ w:[0,\infty)\to Dom(C)$ que calcula el color para cada punto del radio.

4.2. Ruido de Perlin

Para darle un aspecto más natural a la madera se puede emplear el ruido, concretamente el que ha sido diseñado por Ken Perlin. Existen varias maneras en las que se puede utilizar para lograrlo. Para decidir cual, primero hay que entender sus propiedades.

Tanto la función de ruido 3D $N(x,y,z)$, como su primera derivada $N^I(x,y,z)$ son continuas en todo el dominio $\mathbb{R}^3$. La imagen de N se encuentra siempre en el intervalo $[0,1]$. El ruido de Perlin se construye como una onda que es la suma de varias octavas. Cada una proviene de la misma función generatriz, pero reduciendo su amplitud y ampliando la frecuencia.

La manera más sencilla de utilizarlo es empleando su salida como componentes de color y generar directamente una textura 3D. El problema es que eso se parece más a una textura de granito que a la madera.

Por lo tanto debe emplearse como una distorsión, partir de un patrón de base como los anillos y distorsionarlo para dotarlo de un aspecto más natural. Si se emplea como distorsión hay que tener dos factores en cuenta: el primero es el dato que se va a distorsionar con la salida de la función de ruido, y el segundo es la entrada de la misma.

La distorsión se puede aplicar en cualquier punto de la cadena de composición de funciones para calcular el color resultante. El ruido puede emplearse para distorsionar el valor final de color de cada punto, para desplazar el punto antes de empezar a calcular el color o para realizar una distorsión en cualquier punto intermedio de la cadena.

Independientemente del punto de aplicación hay que elegir los datos de entrada. Pueden ser las coordenadas cartesianas, las coordenadas polares, las coordenadas en cualquier mezcla de sistemas de coordenadas o sólo un subconjunto de las posibles coordenadas.

Distorsionar el color difícilmente puede dar como resultado unos anillos realistas. Aplicar la distorsión después de calcular el periodo de los anillos, da como resultado unos anillos en forma de corona cilíndrica totalmente regular, pero con la distribución del color ligeramente distorsionada. Aplicarla al radio presentaría unos anillos bastante verosímiles, pero no habría forma de ver pliegues entre éstos. Por eliminación el lugar adecuado para aplicar la distorsión es al comienzo de la cadena, generando unos anillos que también pueden presentar pliegues.

En este punto de la cadena la entrada más sencilla de lograr y que no requiere repetir cálculos son las coordenadas cartesianas.

La salida de la función de ruido debe producir 3 valores independientes, de modo que la distorsión en cada eje no es la misma. La distorsión en cada eje será un valor comprendido en el intervalo [0,1]: $N:\mathbb{R}^3\rightarrow[0,1]^3$

Para calcular el ruido de Perlin es necesario dividir el espacio en cubos con la longitud de su lado la unidad. Cada cubo está identificado por 3 enteros y las coordenadas de los puntos en el interior de cada cubo son 3 reales en el intervalo [0,1).

$\mathbb{R}^3\rightarrow(\mathbb{Z}\times[0,1))^3$
$(x,y,z)\rightarrow(\lfloor x\rfloor,x-\lfloor x\rfloor,\lfloor y\rfloor,y-\lfloor y\rfloor,\lfloor z\rfloor,z-\lfloor z\rfloor)$

Cada vértice compartido por 8 cubos tiene un identificador único: $(\lfloor x\rfloor,\lfloor y\rfloor,\lfloor z\rfloor)$

Es necesario definir una tabla $G:\mathbb{Z}^3\rightarrow[0,1]^3$ que dado un identificador de un cubo, devuelve 3 valores aleatorios. Este resultado corresponde al de una octava con la misma entrada:

$(x,y,z)\in\mathbb{Z}^3 \Rightarrow o(x,y,z)=G(x,y,z)$

Para el resto de valores del dominio de $o:\mathbb{R}^3\to[0,1]^3$, en cada cubo se debe calcular una interpolación tricúbica de los valores de sus 8 vértices. La interpolación tricúbica consiste en calcular una interpolación cúbica $i_{AB}:[0,1]\to[o(A),o(B)]$ eje por eje de cada par de datos (AB), con los resultados del primer eje, se obtienen los valores de los extremos del segundo.

Por ejemplo para $(x,y,z)\in(0,1)^3$, $o(x,y,z)$ se calcula de la siguiente manera:

$$\left.\begin{array}{l}\left.\begin{array}{l}o(0,0,0)\\o(1,0,0)\end{array}\right\}i_{(0,0,0)(1,0,0)}(x)=o(x,0,0)\\\left.\begin{array}{l}o(0,1,0)\\o(1,1,0)\end{array}\right\}i_{(0,1,0)(1,1,0)}(x)=o(x,1,0)\end{array}\right\}i_{(x,0,0)(x,1,0)}(y)=o(x,y,0)$$

$$\left.\begin{array}{l}\left.\begin{array}{l}o(0,0,0)\\o(1,0,0)\end{array}\right\}i_{(0,0,0)(1,0,0)}(x)=o(x,0,0)\\\left.\begin{array}{l}o(0,1,0)\\o(1,1,0)\end{array}\right\}i_{(0,1,0)(1,1,0)}(x)=o(x,1,0)\end{array}\right\}i_{(x,0,0)(x,1,0)}(y)=o(x,y,1)$$

$$\left.\begin{array}{l}i_{(x,0,0)(x,1,0)}(y)=o(x,y,0)\\i_{(x,0,0)(x,1,0)}(y)=o(x,y,1)\end{array}\right\}i_{(x,y,0)(x,y,1)}(z)=o(x,y,z)$$

La interpolación $i_{AB}:[0,1]\to[0,1]^3$ se calcula mediante la composición $l_{AB}\circ c$, siendo:

$l_{AB}:[0,1]\to[0,1]^3$ la interpolación lineal entre A y B $l_{AB}(t)=o(A)(1-t)+o(B)t$;

y $c(t)=3t^2-2t^3$ la curva polinómica de tercer grado que cumple $c(0)=0\wedge c(1)=1\wedge c^I(0)=0\wedge c^I(1)=0$.

Finalmente se calcula la suma de *n* octavas calculando por cada una de ellas un factor que multiplica cada coordenada antes de calcular el valor y luego divide al resultado. Dicho factor comienza siendo 1 y con cada octava duplica su valor. Tras calcular la suma hay que normalizar el resultado, para que la imagen no quede fuera del rango [0,1].

$$N(x,y,z)=\frac{\sum_{i=0}^{n}\frac{o(2^i(x,y,z))}{2^i}}{\sum_{i=0}^{n}2^{-i}}=\frac{\sum_{i=0}^{n}2^{-i}o(2^i(x,y,z))}{\frac{2^{n+1}-1}{2^n}}=\frac{2^n}{2^{n+1}-1}\sum_{i=0}^{n}2^{-i}o(2^i(x,y,z))$$

5. Implementación

Existe una amplia gama de tecnologías que permiten implementar un *shader* que reúna las condiciones del diseño. Entre ellas cuentan cualquier entorno de desarrollo que trabaje con una API como *Direct3D*, *OpenGL* o *WebGL*.

5.1. Unity

Unity es un *motor de juego* multiplataforma desarrollado por *Unity Technologies*. Se usa para desarrollar videojuegos para PC, videoconsolas, dispositivos móviles, páginas web, etc. [13]

La motivación principal de utilizar un *motor de juego* es que permite al desarrollador concentrarse en un elemento en concreto, en este caso los *shaders*, mientras se dispone de material suficiente para visualizar e interactuar con los resultados durante el proceso de desarrollo.

Concretamente *Unity* dispone de una edición personal gratuita y permite distribuir los resultados en una amplia gama de plataformas.

5.2. ShaderLab

El código de un programa *shader* en *Unity* está escrito en el lenguaje *Cg*/*HLSL*.

```
Shader "MyShader" {
    Properties {
        _MyTexture ("My Texture", 2D) = "white" { }
        // Place other properties like colors or vectors here as well
    }
    SubShader {
        // here goes your
        // - Surface Shader or
        // - Vertex and Fragment Shader or
        // - Fixed Function Shader
    }
    SubShader {
        // Place a simpler "fallback" version of the SubShader above
        // that can run on older graphics cards here
    }
}
```

El fragmento de código anterior[2] muestra el formato de un *shader* escrito para *Unity* en *ShaderLab*. En la primera línea se especifica el nombre del shader. A continuación hay un bloque de código precedido por la palabra clave "*Properties*", dentro se indican las propiedades con las que puedes interactuar desde el editor de unity.

2 Fragmento obtenido de: https://docs.unity3d.com/Manual/ShadersOverview.html

Después va una serie de bloques "*SubShader*", cada uno contiene el código de un *shader* escrito en *Cg*/*HLSL*. Cuando se ejecuta el *shader*, Unity intenta ejecutar el primer *subshader*. Si no compila o causa un error en tiempo de ejecución debido a un error de programación o una incompatibilidad con el sistema operativo o la GPU, Unity lo volverá a intentar con el segundo *subshader*.

5.3. Propiedades

El código de la implementación incluye un gran número de propiedades, de forma que se puede alterar en gran medida el resultado:

```
Properties {
   _VolTexScale ("Volumetric Texture Scale", Vector) = (1, 1, 1, 10)
   _NoiseGridSize ("Noise Grid Size", Vector) = (0.2, 1, 1, 10)
   _DistorFactor ("Distortion Factor", Vector) = (1, 1, 1, 0.5)
   _Texture2DImageSize ("Texture 2D Image Size", Vector) = (1, 1, 1,
256)
   _InnerColor ("Inner Color", Color) = (0.75,0.5,0.25,1)
   _OuterColor ("Outer Color", Color) = (0.5,0.25,0,1)
   _Noise ("Random Noise Source", 2D) = "" {}
   _RecDepth ("Recursion Depth", Range(1, 8)) = 3
}
```

Hay cuatro propiedades del tipo "Vector", este tipo es un vector 4D, cada uno de los 4 elementos que contiene es un numero real. A menudo para agilizar el *renderizado* los reales se utilizan con el tipo *fixed* (coma fija), en lugar de *float* (coma flotante). Los cuatro vectores representan factores de escala, cada uno tiene cuatro componentes (x,y,z,w), las tres primeras representan la escala de sus respectivos ejes XYZ, mientras que la cuarta multiplica a todos los ejes a la vez, la escala 3D resultante es (wx,wy,wz).

- `_VolTexScale`: Es la escala de la textura volumétrica. Afecta a las coordenadas antes de calcular la textura volumétrica, pero no afecta al ruido.
- `_NoiseGridSize`: Es la escala que se aplica antes de calcular el ruido. Afecta al número de cubos que caben dentro del modelo. Al aplicar factores de escala diferentes, los cubos en el sistema global de coordenadas pueden verse deformados.
- `_DistorFactor`: Es el factor de escala aplicado al vector de distorsión una vez calculado.

```
fixed2 dis = volTexCoords.xz;   // (dx, dz)
dis *= dis;                     // (dx^2, dz^2)
fixed radius2 = dis.x+dis.y;
fixed radius = sqrt(radius2);
fixed value = valueFromRadius(radius);
value = easeIn(value);
```

!!br0ken!!

Las propiedades `_InnerColor` y `_OuterColor` son, respectivamente, los colores de la madera en la zona más interior y más exterior de los anillos. Las componentes de los colores vienen dadas también por vectores 4D (r,g,b,a). Todas las componentes son números reales pertenecientes al intervalo [0,1]. En las tres primeras un 1 significa presencia total de rojo, verde y azul respectivamente; un 0 significa ausencia total. La cuarta componente es la opacidad, siendo total cuando $a=1$ y totalmente transparente cuando $a=0$.

La propiedad `_Noise` es un *raster* que se emplea para introducir valores pseudoaleatorios en el *shader*. El *raster* incluido por defecto es una imagen de ruido pseudoaleatorio de 256×256, en cada *pixel* hay una cantidad pseudoaleatoria de rojo, verde y azul.

La propiedad `_RecDepth` es un entero en el rango [1,8], indica la cantidad de octavas incluidas al calcular el ruido de Perlin.

5.4. Vertex Shader

Para generar la textura se emplea un vertex shader y un fragment shader. El código del vertex shader ha sido simplificado para mostrar una imagen sencilla sin reflejos, efectos de luz, niebla ni desenfoque. El programa del vertex shader se ejecuta una vez por cada vértice en cada fotograma.

```
// VERTEX
v2f vert (appdata v) {
   v2f o;
// o.uv = v.uv;
   o.pos = v.vertex;
   // World -> Model -> Viewport -> Perspective
   o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
// UNITY_TRANSFER_FOG(o,o.vertex);
   return o;
}
```

La función vert tiene como entrada un objeto de tipo appdata y como salida uno de tipo v2f.

El tipo appdata es la información de cada vértice proveniente del motor gráfico de Unity:

```
struct appdata {
   float4 vertex : POSITION;
// float2 uv : TEXCOORD0;
};
```

Contiene la información de la posición del vértice y sus coordenadas (u,v) asociadas. Las coordenadas (u,v) se utilizan para el mapeado de texturas. Estas estructuras forman parte de de un código generado automáticamente por Unity, pero el código comentado (precedido por "//") no es necesario para el programa y no se compila.

La estructura v2f (Vertex to Fragment) es una estructura auxiliar para pasar datos de la salida del vertex shader a la entrada del fragment shader:

```
struct v2f {
// float2 uv : TEXCOORD0;
// UNITY_FOG_COORDS(1)
   float4 vertex : SV_POSITION;
   float4 pos : V_POSITION;
};
```

Las dos primeras lineas comentadas pueden ser ignoradas, ya que corresponden a dos características que no se utilizan en este shader: el mapeado de texturas y la niebla.

El registro ***vertex*** contiene la posición del vértice transformada por la *matriz MVP* (*Model → Viewport → Perspective*), es decir, las coordenadas del punto en el sistema de coordenadas local de la cámara con la corrección de perspectiva cónica.

El registro pos es específico para este shader y contiene la posición del vértice en el sistema de coordenadas local del modelo sin modificar.

5.5. Fragment Shader

A diferencia del vertex shader, el fragment shader se ejecuta una vez por cada píxel del viewport. Algunos píxeles corresponden a triángulos que forman la superficie de los modelos. De los vértices de cada triángulo se calcula automáticamente una interpolación en el punto correspondiente al píxel que se está renderizando. En este caso los únicos datos que se interpolan automáticamente son los que pertenecen a los objetos de tipo v2f.

Una vez se ha calculado la interpolación el nuevo objeto v2f sirve de entrada para la función frag:

```
fixed4 frag (v2f i) : SV_Target {
// ...
   fixed4 color;
   color = lerp(_InnerColor, _OuterColor, value);
   return color;
}
```

La salida de tipo fixed4 es el color del *píxel*. Se calcula mediante una *lerp* entre el color interior y el exterior. Lo que cambia es la variable *value* que depende de todos los cálculos que se hacen hasta el final del fragment shader.

5.6. Textura Volumétrica

La variable *value* proviene del radio, según se indica en el diseño.

```
fixed2 dis = volTexCoords.xz;   // (dx, dz)
dis *= dis;                     // (dx^2, dz^2)
fixed radius2 = dis.x+dis.y;
fixed radius = sqrt(radius2);
fixed value = valueFromRadius(radius);
value = easeIn(value);
```

Esta sección de código calcula la variable `value` a partir de la distancia al eje Y. Empezando con el cambio de sistema de coordenadas.

Primero se calcula $\vec{dis}=(x,z)$, es un vector 2D que pertenece al plano XZ. Las componentes (x,z) son las mismas que las de `volTexCoords`, el origen de éstas coordenadas se encuentra en el centro del modelo.

La segunda operación puede resultar confusa, la notación `dis *= dis;` es lo mismo que la asignación `dis = dis * dis;` no se trata ni de un producto escalar ni de un producto vectorial, es un producto componente a componente $(x,z)*(x,z)=(x^2,z^2)$.

La tercera operación calcula el cuadrado del radio como la suma de ambas componentes de `dis`: $radius^2=x^2+y^2$.

Se calcula la raíz cuadrada y se obtiene el radio $radius=\sqrt{x^2+z^2}$.

Aplicar la función periódica `valueFromRadius`, que es la homóloga a la función $w(x)=d(x)\,mod\,1$ del diseño.

```
float valueFromRadius(float radius) {
   return radius % 1;
}
```

En lugar de utilizar $d(x)=T^{-1}x$, se queda $d(x)=x$ ya que la otra transformación se produce en un paso anterior al calcular `volTexCoords`.

Al resultado se le aplica la función `easeIn`, que es la homóloga de la función $g(t)=t^2$ del diseño.

```
float easeIn(float x) {
   if (x<=0) return 0;
   if (x>=1) return 1;
   return x*x;
}
```

Las dos primeras líneas son para controlar posibles excepciones del dominio y aportar modularidad a la función, pero en este caso se pueden omitir, ya que la variable `x` sale de la función `valueFromRadius` que ya se encarga de dejar el resultado en el rango[0,1].

La variable `volTexCoords` se calcula en este fragmento de código:

```
fixed3 volTexCoords = i.pos;
volTexCoords *= _VolTexScale.xyz * _VolTexScale.www;
volTexCoords += distor;
```

Se parte de la posición (x,y,z) del punto a *renderizar* en el *sistema de coordenadas local* del modelo. Habitualmente el punto (0,0,0) se encuentra en el centro de ese sistema.

Se aplica la escala _VolTexScale y luego la distorsión.

5.7. Ruido de Perlin

En el diseño se explica que los datos pseudoaleatorios provienen de una tabla $G:\mathbb{R}^3\rightarrow[0,1]^3$. El problema es que este shader no acepta tablas 3D ni utilizar generadores de números pseudoaleatorios. En su lugar se puede utilizar el "truco" de Perlin que menciona Zucker. [5]

Partiendo de una lista de 256 números del 0 al 255, se utiliza un proceso externo para desordenarla de forma aleatoria. Y obtienes la tabla $F:\mathbb{Z}\cap[0,255]\rightarrow\mathbb{Z}\cap[0,255]$

Esta tabla se utiliza para convertir las coordenadas $(x,y,z)\in\mathbb{Z}^3$ en un indice $j\in\mathbb{Z}\cap[0,255]$. El proceso es el siguiente:

1. Se comienza con el índice $j=x$.
2. Se utiliza la operación módulo $j\leftarrow j\,mod\,256$ para que $j\in\mathbb{Z}\cap[0,255]$.
3. Se obtiene el j-ésimo elemento de F $j\leftarrow F(j)$ y se le asigna a j.
4. Si ya se han utilizado todas las coordenadas, terminar.
5. Añadir a j la siguiente coordenada ($j\leftarrow j+y$ ó $j\leftarrow j+z$)
6. Repetir desde el paso 2.

$$j=F((F((F(x\,mod\,256)+y)\,mod\,256)+z)\,mod\,256)$$

Una vez calculado el índice, se utiliza para buscar el j-ésimo elemento de la tabla de gradientes aleatorios, que en este caso es $G:\mathbb{Z}\cap[0,255]\rightarrow[0,1]^3$, en lugar de $G:\mathbb{R}^3\rightarrow[0,1]^3$.

La manera más sencilla de incluir estas tablas en el shader de Unity es mediante un raster. La tabla F puede ser una imagen de 256×1 en blanco y negro, mientras que G puede ser una imagen del mismo tamaño a color.

En esta implementación en lugar de utilizar un raster de 256×1, se utiliza uno de 256×256. Esto aporta un mayor nivel de aleatoriedad, pero el algoritmo es similar.

La función que sustituye a $F:\mathbb{Z}\cap[0,255]\to\mathbb{Z}\cap[0,255]$ es:

`tex2Dlod: sampler2D x fixed4 → fixed4`

```
fixed4 getPixel (uint3 indices) { // uint for speed with modulus
   // indices : Z[0 , 256)
   fixed3 fIndices = (fixed3(indices)/256.0f)%1; // R[0 , 1)
   fixed4 pix;
   pix = tex2Dlod( _Noise, fixed4( fIndices.xy, 0, 0 ) );
   pix = tex2Dlod( _Noise, fixed4( (pix.rg + fIndices.yz)%1, 0, 0 ));
   pix = tex2Dlod( _Noise, fixed4( (pix.rg + fIndices.zx)%1, 0, 0 ));
   return pix;
}
```

`tex2Dlod` es una función de librería pero en este *shader* se utiliza con `_Noise` como primer argumento constante y el vector del segundo argumento siempre tiene sus dos últimas componentes nulas: (x,y,0,0).

A todos los efectos se comporta como si fuera: $F:[0,1)^2\to[0,1]^4$. El resultado es un color (r,g,b,a), pero a la hora de desplazar los índices sólo se utilizan sus 2 primeras componentes (r,g)

La primera línea después de los comentarios sirve para ajustar el dominio $\mathbb{Z}\cap[0,255]\to[0,1)$.

Esta función se utiliza 8 veces para obtener los valores pseudoaleatorios de los vértices del cubo actual:

```
fixed3 noise3(fixed3 noiseCoords) {
   qr3 qr = divmod3(noiseCoords,
_Texture2DImageSize.xyz*_Texture2DImageSize.www); //Random noise
texture size
   noiseCoords = qr.remainder;  // R[0 , 256)
   qr = divmod3(noiseCoords, fixed3(1, 1, 1));
   uint3 pixIndices = qr.quotient; // Z[0 , 256)
   noiseCoords = qr.remainder; // R[0 , 1)

   fixed3 a000 = getPixel( pixIndices + uint3( 0, 0, 0 ) );
   fixed3 a001 = getPixel( pixIndices + uint3( 0, 0, 1 ) );
   fixed3 a010 = getPixel( pixIndices + uint3( 0, 1, 0 ) );
   fixed3 a011 = getPixel( pixIndices + uint3( 0, 1, 1 ) );
   fixed3 a100 = getPixel( pixIndices + uint3( 1, 0, 0 ) );
   fixed3 a101 = getPixel( pixIndices + uint3( 1, 0, 1 ) );
   fixed3 a110 = getPixel( pixIndices + uint3( 1, 1, 0 ) );
   fixed3 a111 = getPixel( pixIndices + uint3( 1, 1, 1 ) );

   noiseCoords.z = easeInOut(noiseCoords.z);
   fixed3 a00  = lerp( a000, a001, noiseCoords.z );
   fixed3 a01  = lerp( a010, a011, noiseCoords.z );
   fixed3 a10  = lerp( a100, a101, noiseCoords.z );
   fixed3 a11  = lerp( a110, a111, noiseCoords.z );

   noiseCoords.y = easeInOut(noiseCoords.y);
   fixed3 a0   = lerp( a00,  a01,  noiseCoords.y );
   fixed3 a1   = lerp( a10,  a11,  noiseCoords.y );

   noiseCoords.x = easeInOut(noiseCoords.x);
   return lerp( a0, a1, noiseCoords.x );
}
```

La función divmod3 en conjunto con qr3 se utilizan para separar el cociente de la división entera y el resto. En la primera línea se acota el dominio de `noiseCoords` $\mathbb{R}^3\to[0,256)$. En la segunda línea se separan: `pixIndices` $\in\mathbb{Z}\cap[0,256)$ y `noiseCoords` $\in[0,1)$.

`pixIndices` se utiliza como argumento de `getPixel` para obtener los 8 valores, mientras `noiseCoords` se utiliza como variable para la interpolación lerp.

Esta es la función que se emplea para separar el cociente y el resto:

```
struct qr3 {
    fixed3 quotient;
    fixed3 remainder;
};

qr3 divmod3(fixed3 dend, fixed3 dsor) {
    qr3 qr;
    qr.quotient = floor(dend/dsor);
    qr.remainder = dend-qr.quotient*dsor;
    return qr;
}
```

$$q(dend, dsor) = \left\lfloor \frac{dend}{dsor} \right\rfloor$$
$$r(dend, dsor) = dend - q(dend, dsor)\, dsor$$

La función `easeInOut` es la curva cúbica $c(t) = 3t^2 - 2t^3$:

```
float easeInOut(float x) {
    if (x<=0) return 0;
    if (x>=1) return 1;
    return (3-2*x)*x*x;
}
```

La función noise3 genera una sola octava de ruido 3D, para calcular el ruido de Pérlin hay que combinar varias.

Como el shader no acepta bucles `for`, hay que poner todas las iteraciones fuera del bucle:

```
fixed4 octaves(fixed3 noiseCoords, uint depth) {
   fixed4 color = float4(0,0,0,1);
   // FOR

   uint it = 1;
   uint factor = 128;
   uint total = factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   if (depth<=it++) return color/total; // 1

   noiseCoords *= 2;
   factor /= 2;
   total += factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   if (depth<=it++) return color/total; // 2

   noiseCoords *= 2;
   factor /= 2;
   total += factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   if (depth<=it++) return color/total; // 3

   noiseCoords *= 2;
   factor /= 2;
   total += factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   if (depth<=it++) return color/total; // 4

   noiseCoords *= 2;
   factor /= 2;
   total += factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   if (depth<=it++) return color/total; // 5

   noiseCoords *= 2;
   factor /= 2;
   total += factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   if (depth<=it++) return color/total; // 6

   noiseCoords *= 2;
   factor /= 2;
   total += factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   if (depth<=it) return color/total; // 7

   noiseCoords *= 2;
   factor /= 2;
   total += factor;
   color.xyz += noise3(noiseCoords).xyz*factor;
   return color/total; // 8
}
```

Este código es la implementación de: $N(x,y,z)=\frac{2^n}{2^{n+1}-1}\sum_{i=0}^{n} 2^{-i} o(2^i(x,y,z))$

Finalmente, el fragmento de la implementación que calcula la distorsión en el punto (x,y,z):

```
fixed3 noiseCoords = i.pos;
noiseCoords *= _NoiseGridSize.xyz * _NoiseGridSize.www;
fixed3 noise = octaves(noiseCoords, _RecDepth);
fixed3 distor  = noise-0.5;
distor        *= _DistorFactor.xyz * _DistorFactor.www;
```

Se parte de las coordenadas en el sistema local del modelo y se aplica la escala para las coordenadas del ruido (`_NoiseGridSize`).

Se calcula el ruido de Perlin y se cambia el rango de distorsión $[0,1] \rightarrow [-0.5, 0.5]$.

Antes de aplicar la distorsión a las coordenadas de la textura volumétrica, se aplica la escala de la distorsión (`_DistorFactor`).

6. Resultados y Líneas Futuras

Las texturas volumétricas mezcladas con ruido de Perlin 3D son capaces de generar una textura de madera verosímil.

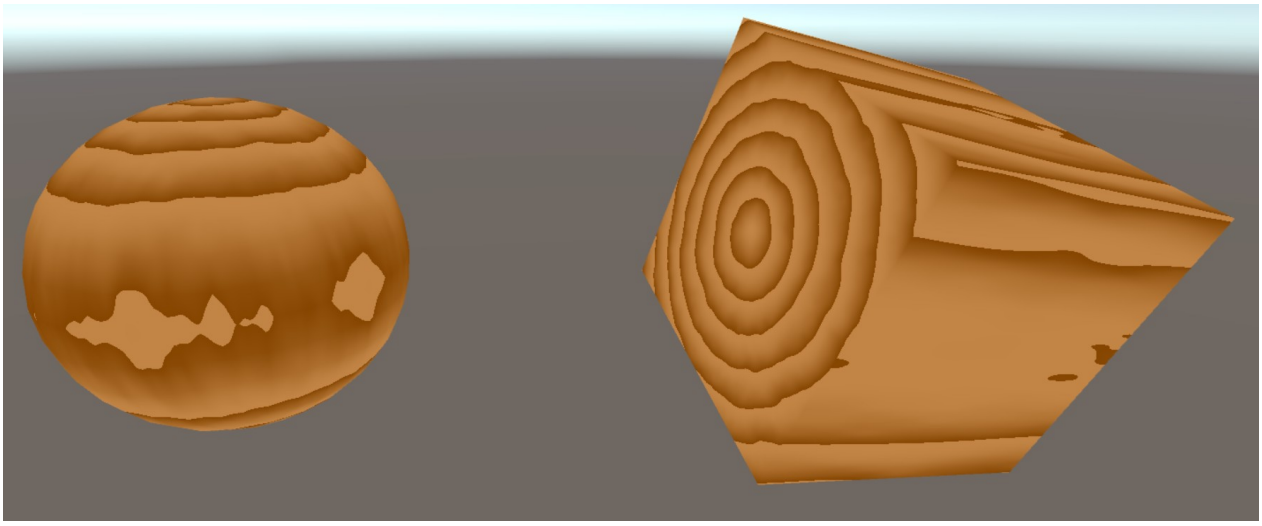

Figura 9: Captura de pantalla durante la renderización en tiempo real. Muestra la textura volumétrica generada con la misma semilla en un cubo, y en una esfera.

En mejoras futuras, además de permitir cambios sencillos en la fuerza del ruido o en el color, se pueden diseñar nuevas formas de interpolación para el color de los anillos. Incluso es posible añadir más puntos de color.

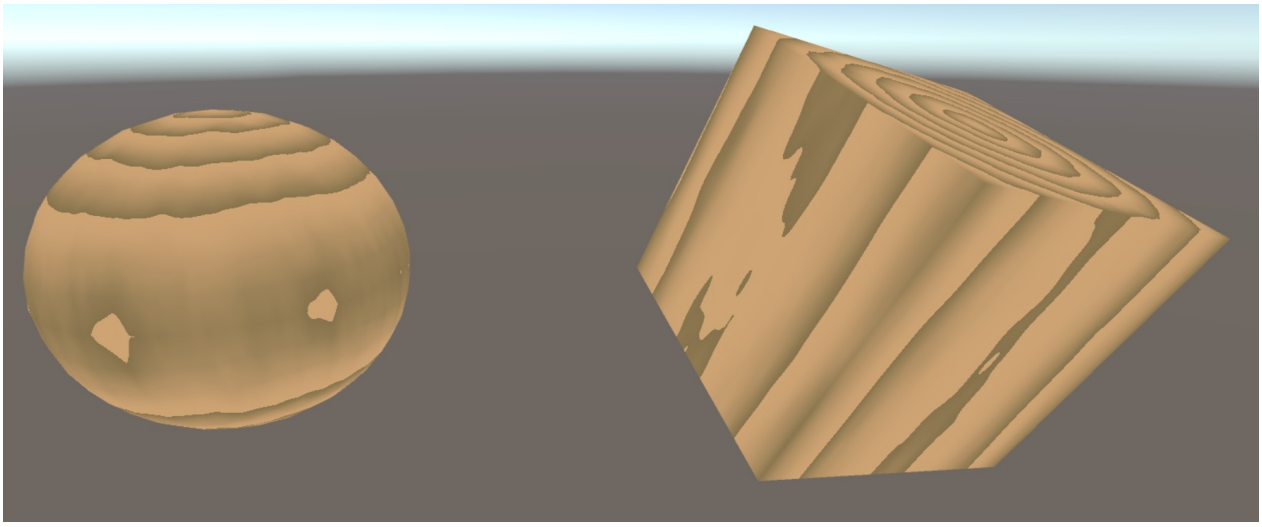

Figura 10: Captura de pantalla durante la renderización en tiempo real. Muestra la textura volumétrica generada con la misma semilla en un cubo, y en una esfera.

Una buena manera de explotar el uso de este tipo de texturas puede ser crear una biblioteca de materiales que simulen el tronco de árboles específicos.

Para mejorar el realismo del shader se pueden implementar efectos basados en la iluminación: sombras, reflejos, rugosidad, efecto de madera mojada, etc.

Para ir un paso más hacia el realismo habría que diseñar una manera de incluir estructuras de árboles generadas mediante L-systems. Añadir ramas y nudos dentro del tronco.

7. Bibliografía

Bibliografía

1: Wikipedia contributors. (2016, 7 de Agosto). Texture mapping. Disponible en https://en.wikipedia.org/w/index.php?title=Texture_mapping&oldid=733437514
2: Wikipedia contributors. (2016, 6 de Octubre). Raster graphics. Disponible en https://en.wikipedia.org/w/index.php?title=Raster_graphics&oldid=742903226
3: Wikipedia contributors. (2016, 12 de Junio). Procedural texture. Disponible en https://en.wikipedia.org/w/index.php?title=Procedural_texture&oldid=724874417
4: Ebert, David S. - Musgrave, F. Kenton - Peachey, Darwyn - Perlin, Ken - Worley, Steven. (2003). Texturing & Modeling: A Procedural Approach. En Texturing & Modeling: A Procedural Approach. (Third Edition, Morgan Kaufman Series in Computer Graphics and Geometric Modeling, pág. 7). Morgan Kaufmann Publishers. San Francisco, CA.
5: Matt Zucker. (2001, 28 de Febrero). The Perlin noise math FAQ. Disponible en http://web.archive.org/web/20160714001306/http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html
6: Wikipedia contributors. (2016, 2 de Octubre). Shader. Disponible en https://en.wikipedia.org/w/index.php?title=Shader&oldid=742248413
7: Wikipedia contributors. (2016, 2 de Octubre). Graphics pipeline. Disponible en https://en.wikipedia.org/w/index.php?title=Graphics_pipeline&oldid=742222271
8: Wikipedia contributors. (2016, 5 de Septiembre). Shading language. Disponible en https://en.wikipedia.org/w/index.php?title=Shading_language&oldid=737794742
9: Wikipedia contributors. (2016, 9 de Mayo). Interpolación. Disponible en https://es.wikipedia.org/w/index.php?title=Interpolaci%C3%B3n&oldid=90965161
10: Wikipedia contributors. (2016, 22 de Septiembre). Linear interpolation. Disponible en https://en.wikipedia.org/w/index.php?title=Linear_interpolation&oldid=740703388
11: Wikipedia contributors. (2016, 17 de Octubre). Polynomial. Disponible en https://en.wikipedia.org/w/index.php?title=Polynomial&oldid=744812455
12: Wikipedia contributors. (2016, 25 de Octubre). Spline (mathematics). Disponible en https://en.wikipedia.org/w/index.php?title=Spline_(mathematics)&oldid=746142589
13: Wikipedia contributors. (2016, 28 de Octubre). Unity (game engine). Disponible en https://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=746599907
14: Wikipedia contributors. (2016, 11 de Octubre). 3D modelling. Disponible en https://en.wikipedia.org/w/index.php?title=3D_modeling&oldid=743783676
15: Wikipedia contributors. (2016, 1 de Noviembre). Rendering (computer graphics). Disponible en https://en.wikipedia.org/w/index.php?title=Rendering_(computer_graphics)&oldid=747257450
16: Wikipedia contributors. (2016, 21 de Julio). Texel (graphics). Disponible en https://en.wikipedia.org/w/index.php?title=Texel_(graphics)&oldid=730951195

8. Glosario

En este anexo se explican más en detalle algunos términos necesarios para comprender el contenido de este documento. También puede utilizarse como guía de traducción inglés-español/español-inglés de los términos técnicos.

3D model	Los ***modelos tridimensionales (3D)*** representan un cuerpo físico usando un conjunto de vértices conectados mediante aristas y triángulos o superficies curvas. Los *modelos 3D* pueden construirse a mano, mediante algoritmos (*modelado procedural*) o escaneados. Su superficie puede definirse con más detalle *mapeando texturas* [14].
Cluster	Conjunto de ordenadores que trabajan de forma paralela con un mismo objetivo. En ocasiones comparten uno o varios recursos (red, almacenamiento, memoria, etc.).
Fragment shader	Un Fragment shader o Pixel shader calcula el color y otros atributos de un fragmento (Término técnico que se refiera a un *Píxel*) [6].
Frustum	Un tronco o sección de pirámide o de cono situada normalmente entre dos planos de corte.
Función polinómica	Ver *Polinomio* abajo.
Función segmentada	Función matemática definida por tramos de su dominio.
GPU	Unidad de procesamiento de gráficos (**G**raphics **P**rocessing **U**nit).
Grado (de un polinomio)	Exponente más alto de la variable del polinomio con un coeficiente no nulo.
Graphics Pipeline	***Graphics Pipeline*** o *Rendering Pipeline* (*tubería de gráficos* o *tubería de renderizado*) se refiere a la secuencia de pasos que se siguen para crear un *raster* 2D a partir de una *escena* 3D[7].
Mapa de bits	Ver *Raster* abajo.
Mapeado de texturas	Ver Texture mapping abajo.
Modelo 3D	Ver *3D model* arriba.
Modelo de shader	Ver *Shader model* Error: no se encontró el origen de la referencia.
Pixel shader	Ver *Fragment shader* arriba.
Polinomio	Un polinomio es una expresión que consiste en una suma de variables y sus coeficientes [11].
Raster	Un ***raster*** es una matriz de puntos que representan *píxeles* o *texels*. Normalmente es una matriz 2D con una distribución rectangular de sus puntos [2]. Cada punto contiene la información del color correspondiente a cada punto, aunque se puede tratar como otro tipo de información.
Rendering	Ver Renderizado abajo.
Renderizado	Proceso de generar una imagen a partir de un modelo 2D o 3D [15].
Renderizado en diferido	Tipo de renderizado exhaustivo. Se utiliza para producir imágenes de alta calidad.

Renderizado en tiempo real	Tipo de renderizado que produce varias imágenes por segundo.
Shader	Un ***shader*** es un programa informático que se emplea para producir los niveles apropiados de color en una imagen [6].
Shading language	Lenguaje de programación para programar *shaders*.
Sombreador	Ver *Shader* arriba.
TAC	**T**omografía **A**xial **C**omputerizada.
Texel	Un ***texel*** (elemento o *píxel* de *textura*) es la unidad fundamental de una *textura* [16]. Ver *Textura* abajo.
Textura	Una ***textura*** se refiere a una imagen que se va a emplear para mapearla a un *objeto virtual*. Ver *Texture mapping* abajo.
Textura procedural	Textura generada mediante un algoritmo.
Textura volumétrica	
Texture mapping	El ***mapeado de texturas*** se refiere al método por el que una o varias imágenes (texturas) se aplican en gráficos generados por ordenador o la superficie de un modelo 3D. Esto aporta información de color, y otros efectos de textura en la superficie de los mismos [1].
Tomografía axial computerizada	***T**omografía **A**xial **C**omputerizada* (***TAC***). Ver *TAC* arriba.
Tubería de renderizado	Ver *Graphics pipeline* arriba.
Vertex shader	Es el tipo de *shader* 3D más común. Su función principal consiste en transformar los vértices de un modelo de su sistema de coordenadas local al de la cámara y aplicar la corrección de perspectiva si es necesario [6].