# Un Framework en GPU para Representar y Renderizar Materiales en Tiempo Real

Alumno: Rodrigo Guillermo Baravalle

Director: Dr. Claudio Augusto Delrieux
Co-Director: Ing. Cristian García Bauza

Julio de 2010



Tesina de Grado - Licenciatura en Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

# Agradecimientos

A mis directores, Claudio y Cristian. Sin sus desinteresados esfuerzos no hubiera sido posible la concreción del presente trabajo. Les agradezco la confianza que depositaron en mí y la posibilidad que me brindaron de ingresar al fascinante mundo de la Computación Gráfica, además de sus palabras de aliento constantes.

A mi familia, por haberme permitido cursar la carrera de grado sin presiones y con la suficiente libertad de elección.

A la Universidad, por mantener el nivel académico y brindarme la posibilidad de obtener los conocimientos necesarios para desempeñarme en mi actividad profesional.

A los profesores que guiaron mis estudios y despertaron el interés necesario para utilizarlos como una filosofía de vida.

A mis compañeros de facultad y amigos, por haber transitado el camino juntos.

A todas las personas que me brindaron su apoyo durante este trabajo.

A todos aquellos que de manera directa o indirecta, influyeron, hicieron su aporte o ayudaron a que este trabajo vea la luz.

# Resumen

Presentamos un framework para representar distintos tipos de materiales naturales, representados por medio de la composición de funciones simples. Se obtuvieron resultados visualmente satisfactorios, lo cual hace alentadora la profundización en el estudio del modelo. Se realizó una implementación para su renderizado en tiempo real, aprovechando la performance de las placas gráficas disponibles actualmente.

Se comienza analizando los antecedentes del problema, junto con una reseña de la teoría que sustenta el trabajo, seguido de la presentación del modelo matemático del framework. Luego se analizan tres casos de estudio, obteniéndose imágenes de los materiales marmol, madera y granito. Seguidamente se generalizan los resultados, abarcando otros materiales. Finalmente se explica una implementación del modelo.

# Índice general

# Índice de figuras

# Capítulo 1

# Introducción

*"¿Las computadoras hacen eso?"*
*Homero Simpson - Los Simpsons - "Homero va a la universidad"*

## 1.1. Texturas

Las texturas ven su aparición en computación gráfica debido a la necesidad de mayor realismo en las aplicaciones. Los detalles de los materiales presentes en la realidad difícilmente pueden representarse sólo diseñando la geometría de la misma, esto es, a través de vértices y sus conexiones. La cantidad de vértices que se necesitarían fácilmente sobrepasaría la imaginación. En otras palabras, la microestructura de los materiales de los objetos no puede ser modelada con los mismos elementos que la macroestructura de los objetos en la escena. La principal técnica que utiliza texturas recibe el nombre de *Texture Mapping*, la cual significa *mapear* una función en una superficie en 3D. Esta función puede tener una o varias dimensiones. En primera instancia, significa, "pegar" una imagen sobre una superficie como si fuese un calco.

Es innumerable la cantidad de aplicaciones que presentan las texturas; sólo por citar algunas: iluminación, sombras, transparencias, aplicaciones médicas, terrenos, juegos, simuladores, etc., lo cual posiciona a las texturas como una de las principales herramientas con la que cuenta el modelado en la computación gráfica, sobre todo debido a que, como se mencionó, presenta una capacidad para modelar detalles de las imágenes presentes en la realidad que de otra forma serían muy difíciles de alcanzar.

Han surgido nuevas alternativas a esta técnica, como es el caso de *Solid Texturing* [10]. Este último asocia a cada punto del espacio un valor (una textura de tres dimensiones). De esta forma, *cualquier* superficie puede tener una textura asociada sin necesidad de mapear puntos en la superficie con coordenadas de la textura. A pesar de la generalidad del método mencionado, en determinadas ocasiones la complejidad y el costo de tales texturas hacen que se opte por un modelo más sencillo. Los materiales que se pretenden modelar son generalmente representados con texturas, por lo tanto, esta herramienta es el principal elemento del presente trabajo.

## 1.2. Antecedentes

Un tópico de importancia con respecto a las texturas es su obtención. Si bien pueden utilizarse imágenes fotográficas como texturas en las aplicaciones gráficas, éstas presentan la desventaja de ser estáticas y de dimensión fija, lo cual despoja de flexibilidad al método, resultando en

un patrón repetitivo si es que la imagen fuese a utilizarse más de una vez, como generalmente ocurre al mapearse sobre superficies de gran tamaño. Por otro lado, estas imágenes presentan detalles sobre la iluminación del entorno en el cual fueron obtenidas (incluso, el "flash" de la cámara fotográfica), con lo cual, al ser utilizadas en un modelo iluminado, puede distinguirse que la interacción con la nueva fuente de luz no es la esperada. Este problema resulta de difícil solución.

Debido a las limitaciones mencionadas, a lo largo de los años se definieron métodos que buscan generar texturas utilizando modelos matemáticos, eliminando así las restricciones mencionadas, pues la generación puede ser introducida durante el proceso de renderización, contando con información como posición de la fuente de luz, distancia al observador, etc., permitiendo adaptar la textura a dichos datos. El modelo en el que se basa este trabajo es el modelo de *Fourier*. El mismo garantiza que cualquier imagen en dos dimensiones puede ser representada con diferentes texturas *seno* en distintas fases y amplitudes [14]. Estas funciones pueden obtenerse con el cálculo de la *Transformada de Fourier*. Gracias a este modelo, sabemos que la composición de funciones muy simples puede lograr resultados que muestran complejidad. Sin embargo la aplicación directa de este método resulta poco intuitiva y muy costosa. Ejemplos de otros modelos matemáticos menos generales, pueden observarse en el capítulo 4 de [2], o en [8]. Sin embargo, la mayoría de estos modelos falla a la hora de presentar sencillez de síntesis, además de mostrar escaso poder de control sobre los materiales representados, o bien el modelo sólo está diseñado para un material en particular.

Basados en estas consideraciones, se propone aquí un framework que permite modelar y sintetizar distintos tipos de materiales sin contar con tales restricciones. Es posible representar materiales existentes en la naturaleza, como madera, mármol, granito, arena, agua o vegetación; y otros que son producto del ser humano, por ejemplo telas o mosaicos. Los mismos son generados a partir de distintas funciones simples, las cuales reciben distintos *parámetros* que son propuestos en el modelo. A pesar de la sencillez en la definición del modelo matemático, los resultados presentan cierto nivel de realismo y de complejidad. Usuarios no familiarizados con los procesos subyacentes que dan origen a las texturas, pueden obtener igualmente los materiales que desean, debido a que el modelo muestra flexibilidad y facilidad de uso.

Siguiendo la creciente tendencia de computación intensiva en hardware paralelo, las texturas son sintetizadas en la *GPU*[1]. El diseño y la síntesis de los materiales tiene lugar en tiempo real, permitiendo su sencilla integración con aplicaciones gráficas existentes.

[1] Graphics Processing Unit

# Capítulo 2

# Teoría Subyacente

*"El segundo es después del primero"*
*Buzz Aldrin - Los Simpsons - "Homero en el espacio profundo"*

## 2.1. Resumen

A continuación se expone la teoría básica necesaria que encuadra el presente trabajo y permite entender su modelización e implementación. En primera instancia se presentan aspectos teóricos generales utilizados durante el desarrollo del mismo, como el mapeo de texturas, Perlin Noise y Spot Noise. Luego se explican tecnologías propias de la computación GPU, haciendo hincapié en el lenguaje de shading[1] utilizado en la implementación.

## 2.2. Texture Mapping

### 2.2.1. Introducción

*Texture Mapping* significa *mapear* una función en una superficie en 3D [7]. Esta función puede tener una o varias dimensiones. En primera instancia, significa, "pegar" una imagen sobre una superficie como si fuese un calco.

En su caso más usual, se desea mapear una superficie 2D en una 3D, como por ejemplo una pared. Así tenemos por un lado la imagen 2D que representa por ejemplo, ladrillos, y un modelo de una habitación. La imagen podría haber sido obtenida con una cámara fotográfica, o tal vez generada con un programa de edicion gráfica como por ejemplo CorelDRAW, PhotoShop, etc. Entonces se toman los vértices del polígono representando la superficie y se le asocian los texels de la imagen. Generalmente se hace esto con los extremos de la imagen, con lo cual los intermedios se obtienen por *interpolación*.

Se podría entonces resumir el problema del mapeo con la siguiente ecuación:

$$(u, v) = F(x, y, z)$$

es decir, encontrar $F$ tal que mapee cada punto del espacio a una coordenada de la textura. Por convención, el espacio del objeto es referido con las variables $x$,$y$ y $z$, mientras que el de la textura bidimensional con $u$ (coordenadas horizontales) y $v$ (coordenadas verticales).

[1] Lenguajes diseñados para trabajar sobre elementos gráficos en aplicaciones como color, normal, etc. http://en.wikipedia.org/wiki/Shading_language

### 2.2.2. Mapeos Usuales

Se exponen a continuación a modo de ejemplo dos de los mapeos más conocidos. En la Figura 2.1 pueden observarse, de izquierda a derecha, una textura, la textura con un mapeo cilíndrico aplicado, y la textura con un mapeo esférico aplicado.

Figura 2.1: Mapeos esférico y cilíndrico.

#### Cilíndrico

Se precisan dos variables para indicar un punto único en la superficie de un cilindro: el ángulo ($\theta$) y la altura (z). Los puntos tridimensionales de un cilindro se representan paramétricamente a partir de éstos como:

$$(r * cos(\theta), r * sen(\theta), h * z),$$

$$0 < \theta < 2\pi, 0 < z < 1, r : radio, h : altura.$$

Entonces, un punto de la textura resulta mapeado de la siguiente forma:

$$(u, v) = (\theta/2\pi, z), u, v \in [0, 1].$$

#### Esférico

Este caso es más complicado, como lo demuestran las ecuaciones. Una parte de la superficie de una esfera puede definirse paramétricamente, a partir de dos ángulos: $\theta$ y $\phi$, aquí tomamos como ejemplo ciertos límites para ambos:

$$(r * cos(\theta) * cos(\phi), r * sen(\theta) * sen(\phi), r * cos(\phi)),$$

$$0 < \theta < \pi/2, \pi/4 < \phi < \pi/2.$$

Resulta entonces el siguiente mapeo:

$$(u, v) = (\frac{\theta}{\pi/2}, \frac{(\pi/2) - \phi}{\pi/4}),$$

$$u, v \in [0, 1].$$

Generalmente las texturas están definidas entre 0 y 1, aunque también se pueden definir arbitrariamente. En cualquier caso, el comportamiento fuera del rango definido es variado. Podría estar definido de acuerdo a la aplicación, por ejemplo, se puede poner un borde a la textura y asignarle ese color a cualquier valor fuera de la misma, o bien podría ser el valor que "continuaría"

la textura, en el sentido que el texel más cercano dentro del rango es el valor que se devuelve. Una explicación completa y detallada sobre el uso de texture mapping en bibliotecas gráficas puede verse en [15]. A continuación se explica brevemente una técnica muy común utilizada junto a Texture Mapping, que permite añadir más detalle a la hora de aplicar el método, y que será mencionada como una posible continuación al presente trabajo.

## 2.3. Bump Mapping

### 2.3.1. Introducción

Siguiendo la línea expuesta previamente, la principal idea de este tipo de mapeo es representar un mayor nivel de detalle sin alterar la geometría. Previo a la aparición de esta técnica, las texturas no presentaban detalles de su microestructura ante los cambios en las posiciones de las fuentes de luz. Una de las principales características de las superficies es justamente, debido a cambios pequeños en su estructura, la aparición de sombras, lo cual deja al expuesto estas variaciones. James Blinn [1] introdujo entonces la técnica conocida como Bump Mapping, la cual utiliza un *mapa de alturas*, representando la profundidad a la cual se encuentra cada texel de la textura. A partir de este valor, y por medio de operaciones simples, es posible calcular la normal al mapa de alturas, la cual finalmente es añadida a la normal a la superficie en el modelo de la escena donde se encuentra la misma. Gracias a este cálculo es posible saber cómo interactúa la luz con cada texel en particular. En la Figura 2.2 puede observarse una esfera con bump mapping aplicado, lo cual permite apreciar cómo la iluminación influye en la textura. Una descripción detallada de bump mapping puede verse en el capítulo 6 del libro [14].

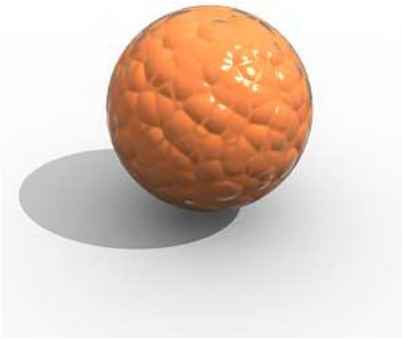


Figura 2.2: Bump Mapping aplicado a una esfera.

Las superficies de los objetos tienen un vector *normal* que es usado en los modelos de iluminación. Existe una técnica derivada llamada *Normal Mapping*, la cual consiste en *perturbar* estos vectores normales de tal modo que, cuando se ilumine el modelo, la superficie parezca tener otra geometría. Para ésto, se deben suplir estos nuevos vectores, diferentes de los originales. Estos son representados en un *mapa de normales*; cada *texel* del mapa de normales indica una perturbación a la normal de la superficie a la que se le pretende aplicar el mapeo. Es así que luego de un procedimiento normal de mapeo, lo que se asignará a la superficie en lugar de un color es una normal en ese punto. Con esto, a medida que la fuente de luz se mueve, se puede calcular en tiempo real la cantidad de luz que recibe cada punto.

### 2.3.2. Detalles

Con el vector normal y el vector que representa la orientación de la luz, podemos saber cuánto influye esta última en esa superficie. Esto se logra realizando el *producto escalar* de ambos vectores *normalizados*, lo cual representa una medida del ángulo entre ambos. Con este resultado, sabremos cuánta luz recibe ese texel de la superficie en particular. Por lo tanto, este procedimiento se realiza a nivel de texel, es así que cada punto en particular recibirá una cantidad de luz diferente, logrando representar los detalles pretendidos.

El mapa de normales puede guardarse en una textura *RGB*, así el canal rojo contiene la coordenada $x$ del vector, el canal verde la coordenada $y$, y el canal azul la coordenada $z$, como se observa en la Figura 2.3. Como se mencionó el mapa de normales generalmente se construye a partir de un mapa de alturas. Muchas aplicaciones gráficas realizan este proceso. Luego se puede utilizar el mapa como una textura común. En la Figura 2.4 se puede observar cómo la normal cambia para simular otra superficie. La normal perturbada es leída en el mapa de normales para cada texel.

Figura 2.3: Mapa de normales.

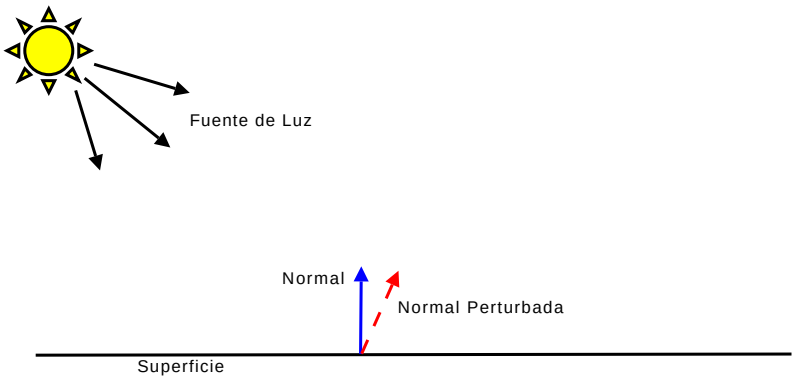



Figura 2.4: Esquema simplificado de Normal Mapping

Presentamos ahora el modelo teórico en el cual está basada la idea principal del trabajo. El mismo establece que por medio de funciones simples, es posible lograr representar cualquier imagen en dos dimensiones.

## 2.4. Transformada de Fourier

### 2.4.1. Introducción

La *Transformada de Fourier* permite analizar señales descomponiéndolas en señales más simples, más específicamete, *senos* y *cosenos*. Así, toda función con dominio en los reales puede ser aproximada como una sumatoria infinita de estas señales básicas. En el presente trabajo la misma es utilizada debido a propiedades útiles que posee, las cuales serán introducidas a continuación. En adición, ésta provee un marco teórico donde basar el modelo.

### 2.4.2. Transformada 1-D Continua

Se define $\forall u \in \mathbb{R}$ como:

$$F(u) = \int_{-\infty}^{\infty} I(x) e^{-i2\pi ux}\, dx,$$

donde $\forall x \in \mathbb{R}$:

$$I(x) = \int_{-\infty}^{\infty} F(u) e^{i2\pi ux}\, du.$$

Se dice que la función $I(x)$ tiene transformada $F(u)$ y que la *transformada inversa* de $F(u)$ es $I(x)$. Típicamente, I(x) es una función real, mientras que F(u) es una función compleja. Por consiguiente, F(u) puede ser separada en parte real e imaginaria:

$$F(u) = Re(u) + Im(u).$$

donde Re(u): parte real de F(u) y Im(u): parte imaginaria de F(u). Definimos su magnitud como:

$$|F(u)| = \sqrt{Re(u)^2 + Im(u)^2},$$

y su fase como

$$\phi(u) = tan^{-1}(Im(u)/Re(u)).$$

La magnitud de F(u) permite identificar cuáles funciones seno componen la señal original y la fase establece el desplazamiento de los mismos. Sumando todos los términos en sus magnitudes y fases correspondientes obtenemos la función original.

### 2.4.3. Transformada 2-D Continua

Similarmente, en dos dimensiones la transformada se escribe como:

$$F(u,v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x,y) e^{-i2\pi(ux+vy)}\, dx\, dy, \forall (u,v) \in \mathbb{R}^2,$$

donde:

$$I(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(u,v) e^{i2\pi(ux+vy)}\, du\, dv, \forall (x,y) \in \mathbb{R}^2.$$

### 2.4.4. Transformada Discreta Bidimensional

Las texturas presentes en un ordenador se almacenan en un arreglo discreto de una o más dimensiones. Por lo tanto, si se pretende aplicar la transformada a un objeto de este tipo, se debe definir la misma como una funcion discreta. Presentamos la definición de la transformada de Fourier bidimensional discreta, para una imagen bidimensional a escala de grises de dimensiones $N \times N$, la cual es otra imagen bidimensional. La textura resultante F se define en cada coordenada $(u, v)$ con $u, v \in \{0, 1, ..., N-1\}$ como:

$$F(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} I(x,y) e^{-i2\pi(ux+vy)},$$

siendo su par transformado (la imagen original):

$$I(x,y) = \sum_{u=0}^{N-1} \sum_{u=0}^{N-1} F(u,v) e^{i2\pi(ux+vy)}.$$

En la Figura 2.5 puede observarse la imagen original a la izquierda, y la magnitud de su transformada a la derecha[2], desplazada hacia el centro de la imagen, para su mayor comprensión.

Figura 2.5: Imagen y su Transformada de Fourier

Estas imágenes deben interpretarse entendiendo que representan un "mapa" de cuáles señales básicas componen la imagen. Así, en este caso, vemos que está formada mayormente por frecuencias bajas (cercanas al (0,5, 0,5)).

### 2.4.5. Propiedad de Separabilidad

Este tipo particular de transformada puede ser reescrita como:

$$F(u,v) = \sum_{x=0}^{N-1} F(x,v) e^{-i2\pi ux},$$

[2]Fuente: http://www.lip.uns.edu.ar/pdi/index33.htm

donde

$$F(x,v) = \sum_{y=0}^{N-1} I(x,y)e^{-i2\pi vy}.$$

Es decir, una transformada discreta bidimensional puede calcularse como una combinación de dos transformadas discretas de una dimensión. Esto permite calcular la transformada de manera más eficiente.

### 2.4.6. FFT

De la definición de la transformada se observa que el valor que asume en cada píxel es combinación de todos los píxeles de la imagen original. Por lo tanto, el cálculo de la misma resulta muy poco eficiente si es implementada directamente, siendo su complejidad $O(N^2)$. Sin embargo, en base a la propiedad de separabilidad de la misma se ha desarrollado el algoritmo *Fast Fourier Transform*, el cual posee complejidad $O(Nlog(N))$. No es el objetivo aquí describir en detalle el mismo, sino explicar que éste es utilizado en la implementación del cálculo de la transformada en determinadas bibliotecas, como se mostrará posteriormente.

### 2.4.7. Convolución

Se presenta aquí este operador que permite combinar dos funciones para obtener una tercera, la cual presenta características tanto de una como de otra. El mismo será aplicado en el tópico spot noise. Su definición matemática es:

$$f(t) \star g(t) = \int f(r)g(t-r)dr,$$

siendo en su caso discreto:

$$f(m) \star g(m) = \sum_n f(n)g(m-n).$$

El *teorema de convolución* establece que, para dos funciones $f$ y $g$ dadas, si denotamos con $F(f)$ y $F(g)$ a las transformadas de f y g, respectivamente:

$$F(f \star g) = F(f) * F(g).$$

Es decir, la transformada de la convolución es la multiplicación de las transformadas. A continuación se presenta un problema teórico que surge del incorrecto muestreo de señales por medio de dispositivos discretos. El problema será tenido en cuenta en el capítulo de implementación del modelo. La transformada de Fourier permite entender el problema y establece una de las soluciones al mismo en determinados casos particulares.

### 2.4.8. Aliasing

El término *aliasing* es propio de la teoría de procesamiento de señales. Una señal puede ser reconstruída a partir de un conjunto discreto de muestras de la misma, separadas por intervalos homogéneos. Este proceso es llamado *muestreo*. Un incorrecto muestreo puede producir la presencia de *artefactos* en la reconstrucción de la señal muestreada. En el caso de la computación gráfica, y más precisamente en el mapeo de texturas, un modelo de una textura debe ser representado por medio de muestras, las cuales deben poseer la capacidad de capturar toda la información que el modelo proporciona. Esta "información" que el modelo posee se conoce

como su *límite en banda*. Un modelo puede estar limitado en banda o bien puede no estarlo. El aliasing ocurre cuando las muestras ocurren en una frecuencia que no logra capturar toda la información que el modelo posee. La frecuencia mencionada recibe el nombre de *frecuencia de muestreo*. En la Figura 2.6 puede observarse cómo una señal incorrectamente muestreada puede ser interpretada como una de frecuencia más baja. En la Figura 2.7 se muestra el efecto en una imagen, lo cual ejemplifica el problema en dos dimensiones: a la izquierda de la imagen la pared de ladrillos está correctamente muestreada, mientras que a la derecha, la misma presenta ondas con frecuencia más baja no deseadas. Este tema será tratado en el capítulo de implementación del presente trabajo.

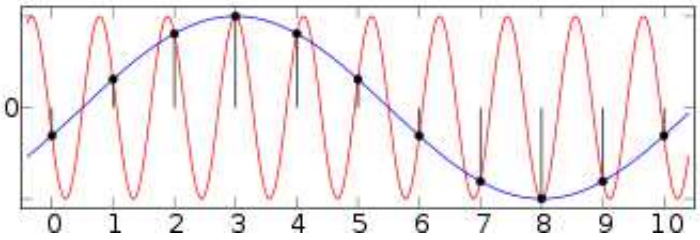



Figura 2.6: Aliasing en la teoría de señales.

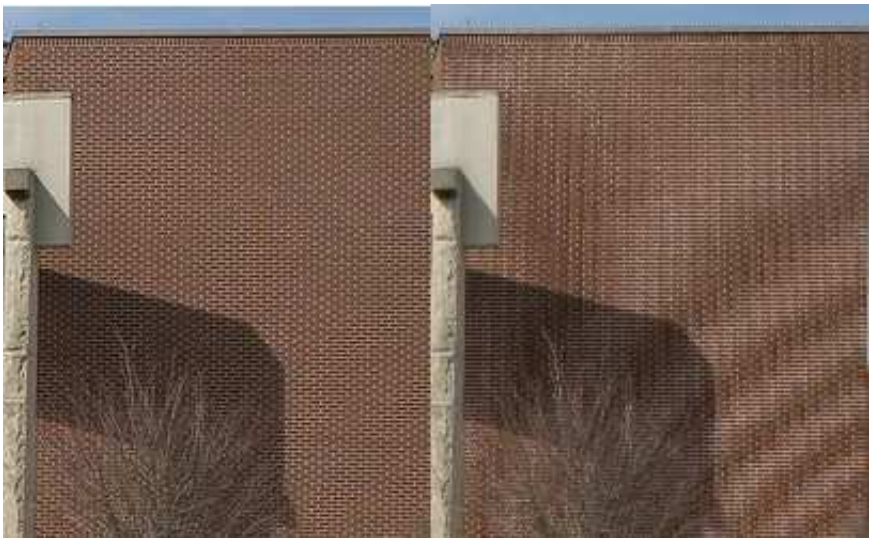


Figura 2.7: Aliasing en imágenes.

Utilizando el concepto de Transformada de Fourier, una función límitada en banda posee un máximo en su dominio de frecuencias. Por esta razón, un muestreo adecuado de la señal debe poder capturar esa frecuencia máxima. El *teorema del muestreo* establece que para que una señal pueda ser completamente reconstruída, sin pérdida de información, la frecuencia de muestreo debe ser más del doble de la frecuencia máxima de la señal. Una demostración de este teorema puede verse en el capítulo cuatro del libro [14]. La frecuencia máxima de la señal recibe el nombre de *Frecuencia de Nyquist*. En lugar de desaparecer, las frecuencias que no pueden ser

capturadas por el muestreo, aparecen como frecuencias más bajas, las cuales reciben el nombre de *alias* de las frecuencias que no pueden ser representadas.

Si se aumenta la frecuencia de muestreo, es posible capturar más información en la señal original. Lamentablemente, debido a cuestiones de espacio, y a la resolución limitada que presentan las imágenes, no siempre esto es posible. Por otro lado, si el modelo no está limitado en banda, nunca se logrará capturar toda la información, por mucho que se aumente dicha frecuencia. Es por esto que es necesario quitar las frecuencias más altas para poder representar de forma adecuada al modelo original, esta técnica es conocida como *filtrado pasabajos*, aunque esto no siempre es sencillo de llevar a cabo. El efecto de un filtrado pasabajos es el de suavizar la imagen, produciendo a veces una excesiva suavización. Por lo tanto, debe encontrarse un equilibrio en este proceso: deben eliminarse frecuencias excesivamente altas, cuidando de no suavizar demasiado la textura resultante.

Existen otros métodos muy comunes, como *supersampling*, el cual consiste en tomar muestras a un frecuencia mayor a la resolución que tendrá la imagen final. La señal es muestreada a intervalos más cortos, buscando capturar frecuencias que no eran representadas con la frecuencia de muestreo que tiene la resolución de la imagen, realizándose posteriormente un promedio entre los valores obtenidos. Sin embargo, esto no presenta una solución al problema, como fue expuesto previamente, sino que atenúa los errores que el aliasing podría producir.

Otro método es el conocido como *clamping*, el cual sólo puede aplicarse en el caso de síntesis de texturas por medio de agregado explícito de determinadas frecuencias (*spectral painting*). En este caso, es muy simple determinar cuándo una frecuencia supera el límite de Nyquist, en cuyo caso, la misma no es agregada a la textura final.

Tal vez uno de los métodos más conocidos es aquel que recibe el nombre de *Summed Area Table* [2, 14]. A partir de la imagen original, se construye otra, la cual en el texel $(0, 0)$ contiene el valor de la textura en el mismo texel, y, para todos los demás, el valor del texel $(i, j)$ es la suma de todos los valores de los texels dentro del rectángulo que se forma entre el origen de coordenadas y dicho texel. De esta forma resulta muy sencillo calcular la suma de todos los valores dentro de cualquier rectángulo arbitrario en la textura. Si ese rectángulo tiene las dimensiones que tendría un filtro lineal sobre la imagen, puede utilizarse el valor promedio entre los píxeles para realizar antialiasing sobre la textura. Lamentablemente, este método resulta muy costoso para un software en tiempo real. El método análogo es calcular la *integral* de la función que pretende sintetizarse, y utilizarlo en lugar del valor de la suma de todos los texels dentro de la región del filtro. Sin embargo, el cálculo de la integral de una función arbitraria no siempre resulta sencillo, y en muchos casos es impracticable.

La utilización de *ruido*[3] puede resultar útil en este problema. Este es utilizado para producir texturas más realistas que aquellas que presentan aliasing, debido a que las texturas naturales presentan características estocásticas que pueden ser capturadas por estas funciones. Por lo tanto, si no pudiera utilizarse otro método, la inclusión de ruido en determinadas ocasiones podrá producir mejores resultados que permitir que la textura presente alias.

En algunas ocasiones no podrá utilizarse ninguno de los métodos expuestos anteriormente, debido a veces a la complejidad que puede existir en un modelo procedural. En tales ocasiones, el programador se verá forzado a utilizar alguna estrategia simple, que intente disminuir el aliasing presente, sin embargo, tal vez no pueda eliminarlo completamente.

Como podrá observarse, el aliasing se presenta como un problema importante de solución no trivial asociado a la naturaleza misma de las texturas. Una vez presentados estos conceptos básicos sobre texturas en computación gráfica, se procede a desarrollar a continuación distintos

[3] función pseudoaleatoria

modelos de generación de texturas, los cuales serán utilizados para sintetizar los materiales que producirá el framework.

## 2.5. Texture Synthesis

### 2.5.1. Introducción

Debido a la importancia de la obtención de texturas, se han desarrollado diversas técnicas que intentan producir a las mismas utilizando diversos algoritmos. Estas técnicas reciben el nombre de *Texture Synthesis* (síntesis de texturas). Existen innumerables algoritmos que intentan resolver el problema de diversas maneras, entre ellos, modelos *fractales* [9], *image based texture synthesis* [3] que consiste en generar imágenes a partir de otras de menor tamaño y *cellular texturing* [2], que utiliza funciones base para combinarlas y producir otros resultados (similar operacionalmente a la propuesta de este trabajo). Aquí mencionaremos dos de ellos, que han sido utilizados a la hora de producir los materiales que están presentes en el framework: Perlin Noise y Spot Noise.

### 2.5.2. Perlin Noise

En [10] se presentan un conjunto de ideas que marcaron una época. Aún en nuestros días, éstas son utilizadas en el área de computación gráfica que ocupa el presente trabajo. Aquí nos proponemos presentar aquellas que fueron aplicadas para representar los distintos materiales que provee el framework.

#### Funcion *Noise()*

Esta primitiva es la principal herramienta con la que se cuenta a la hora de diseñar texturas realistas, en el sentido de que las mismas presentan un caracter *estocástico*, es decir pseudoaleatorio, como son los materiales presentes en la realidad. Sus principales características son:

- Invariancia estadística sobre *rotación.*
- Invariancia estadística sobre *traslación.*
- Límitada en banda (no existen detalles más allá de cierta frecuencia).

Esta función presenta la posibilidad de aplicación de rotación, escalamiento y desplazamiento (debido a sus propiedades), sin perder sus características primordiales. La propuesta en el artículo es la siguiente:

Se define un *reticulado entero* como el conjunto $\mathbb{Z}^3$. Asociamos a cada punto en ese conjunto un valor pseudo-aleatorio $d$ y un vector $(x, y, z)$. Entonces:

$$Noise : \mathbb{R}^3 \rightarrow \mathbb{R},$$

$$\forall (x, y, z) \in \mathbb{Z}^3, Noise((x, y, z)) = d_{(x,y,z)},$$

$\forall (x, y, z) \notin \mathbb{Z}^3, Noise((x, y, z))$ es el resultado de una interpolación entre los valores que asume la misma en los puntos del reticulado. Puede hacerse uso de interpolación cúbica, pero podría utilizarse interpolación lineal, por ejemplo, si se busca performance contra realismo.

Figura 2.8: Jarrón de mármol sintetizado con la función *Noise()*.

**Ejemplo de uso: Mármol**

Un ejemplo del uso de esta primitiva es el de producir texturas similares al mármol. Las funciones que siguen toman un punto del espacio $\mathbb{R}^3$ como entrada. En primer lugar, $Noise()$ se utiliza para crear otra primitiva: $Turbulence()$ la cual puede definirse de la siguiente forma:

```
function turbulence(p) {
    t = 0
    scale = 1
    while(scale > pixelsize) {
        t += abs( Noise(p/scale) * scale)
        scale /= 2
    }
    return t
}
```

Esta función agrega un caracter pseudo-aleatorio en distintas frecuencias presentes en la imagen final. Esto puede observarse dentro del loop *while*, donde se suma *Noise* en distintas escalas. Con esta primitiva, procedemos a definir una textura con una apariencia muy similar al mármol. El mismo presenta distintas *vetas* las cuales son modeladas aquí con una función *seno*.

```
function marble(p) {
    x = p.x + turbulence(p)
    return marbleColor(sin(x))
}
```

El código anterior puede ser implementado de manera trivial en un fragment shader (ver lenguaje Cg en la siguiente sección). En [10] también se menciona este concepto, pero con el nombre de *Pixel Stream Editor* (PSE).

La función marbleColor toma un escalar como argumento y devuelve el color del mármol que se pretende sintetizar. Así, si el argumento es 0 podría devolver un color blanco y un color

violeta en otro caso. De esta forma, cada mármol puede tener un *colormap* distinto asociado. Entre las ventajas de este proceso está la separación entre el modelado de la textura y la elección del color que presentará la misma.

Se ha demostrado entonces el poder que ofrece esta primitiva. De similar forma, puede modelarse fuego, nubes u otros fenómenos naturales. En este trabajo se utilizó la misma función base para producir el material mármol, pero realizando combinaciones entre distintas bases, como se mostrará en los casos de estudio del siguiente capítulo. La técnica que se presentará en la siguiente sección será utilizada para producir características estocásticas en los materiales que sintetizará el modelo.

### 2.5.3. Spot Noise

#### Introducción

Esta técnica es presentada en [13]. Su principal característica consiste en la utilización de un *spot* (una figura) como ser, un círculo, un cuadrado, un triángulo, etc., para producir texturas que tengan relación con dicho spot. Más específicamente, las texturas resultantes presentan apariciones de distintos tamaños en distintas posiciones de dicho spot. Estas texturas son sintetizadas por medio de una convolución del spot con ruido blanco[4].

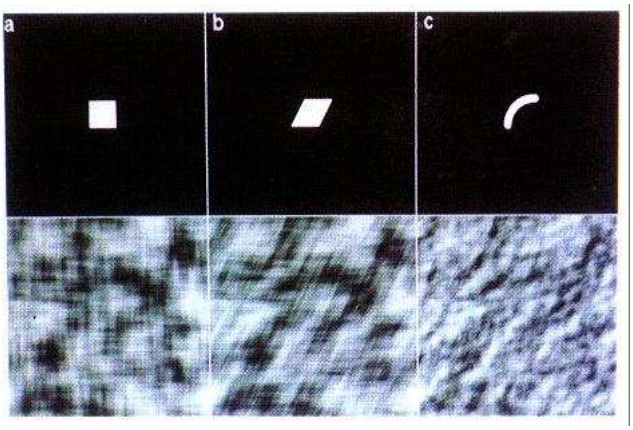



Figura 2.9: Spot Noise: distintos spots y textura resultante.

#### Definición

Spot Noise es el análogo en dos dimensiones de *shot noise*. Este último es una función que está formada por pulsos independientes en momentos aleatorios de distinta intensidad.

$$f(t) = \sum_i a_i h(t - t_i).$$

Cada $t_i$ es un valor aleatorio que toma la variable $t$. $h(x)$ es la función que produce los distintos pulsos. La transformada de $f(t)$ tiene una relación directa con la de $h(t)$:

$$P_f(\omega) = v < a_i^2 > S_h(\omega),$$

[4]http://es.wikipedia.org/wiki/Ruido_blanco

$$S_h(\omega) = |H(\omega)|^2 ,$$

siendo $v$ el promedio de repeticiones por unidad de tiempo y $a_i$ constantes que producen distintas "amplitudes". $H(\omega)$ es la transformada de $h(t)$.

Spot noise es definido de manera análoga como una sucesión de pulsos (spots) independientes en posiciones aleatorias, de distinto tamaño. Su definición matemática es la siguiente:

$$f(x) = \sum_i a_i h(x - x_i),$$

donde $x$ es un punto en el plano y $x_i$ es una posición aleatoria en el plano. Se tiene de igual manera, siendo $v$ el número de repeticiones promedio del spot por unidad de área:

$$P_f(k) = v < a_i^2 > S_h(k).$$

La última ecuación establece que la transformada del spot noise es igual a la transformada del spot, excepto por una constante. Por lo tanto se puede obtener spot noise a partir de la multiplicación de la transformada del spot con la adición de una randomización de la fase $\alpha_k$. Esto último se logra multiplicando con $W(k) = e^{i\alpha_k}$. La transformada de $W(k)$, $w(x)$ está uniformemente distribuida sobre todas las frecuencias. Por lo tanto $w(x)$ es ruido blanco, y como multiplicación en el dominio de las frecuencias es equivalente a convolución en el espacio de la textura (teorema de convolución), se desprende que spot noise puede ser sintetizado por medio de una convolución entre el spot y ruido blanco.

Visto de otro modo, spot noise puede ser sintetizado por medio del cálculo de la transformada del spot, y posterior multiplicación con la transformada del ruido blanco (que es ruido blanco), o bien por medio de una randomización de la fase de la transformada del spot. Estas consideraciones son muy fáciles de llevar a la práctica en la GPU gracias a las herramientas existentes hoy en día, las cuales son presentadas en la siguiente sección. Se procede entonces a presentar dos lenguajes que permiten codificar algoritmos para su posterior ejecución en la GPU.

## 2.6. Computación GPU

### 2.6.1. Introducción

La constante demanda de poder en las aplicaciones gráficas, motivada sobre todo por el auge de los videojuegos, han provocado que el poder de las unidades de hardware paralelo hayan sufrido una marcada evolución desde su aparición. Dicha evolución ha tenido como consecuencia también la rivalización y posterior superación por parte de las GPU's del poder de procesamiento con el que cuentan las CPU's[5] tradicionales[6]. En un principio, los lenguajes de shading probaron ser una herramienta útil para controlar el comportamiento y apariencia de objetos en una escena. Con el tiempo, y debido al mencionado poder de cómputo creciente que experimentaron las GPU's, junto al modelo de computación *paralelo* con el que cuentan las mismas, surgieron herramientas que permiten derivar todo el cálculo hacia el hardware secundario, conociéndose este modelo de computación como *GPGPU*[6]. La tendencia continúa en alza, y cabe preguntarse si con el tiempo, las placas gráficas acabarán por reemplazar a los CPU's existentes hoy en día. A continuación se presentan dos ejemplos de las tecnologías mencionadas. En primer lugar se

[5]Central Processing Unit

[6]General-Purpose Computing on Graphics Processing Units

discute el lenguaje de shading utilizado en el framework. En segundo lugar se muestra un ejemplo de una biblioteca de GPGPU muy conocida hoy en día, la cual fue utilizada para el cálculo de texturas offline.

### 2.6.2. El lenguaje Cg

#### ¿Qué es Cg?

La palabra Cg significa "C para gráficos", en alusión al bien conocido lenguaje de programación C. Esto implica que Cg es un lenguaje también, siendo su principal novedad que el mismo está diseñado para producir código *Assembler* de la placa gráfica de la computadora (GPU). El mismo nos permite controlar la apariencia, movimiento y forma de los objetos dibujados usando la GPU [5]. Cg no es un lenguaje de propósito general, sino que está completamente orientado a trabajar sobre elementos gráficos como vértices, píxeles, líneas, polígonos, vectores, etc.; su nombre está basado en C sólo por su sintaxis, lo cual le otorga un parecido al mismo. Este tipo particular de lenguajes reciben el nombre de *Shading languages* (lenguajes de Shading), aunque como se verá más adelante, el lenguaje permite realizar otras clases de operaciones.

La principal ventaja de usar la GPU consiste en la *velocidad*. La CPU (*Central Processing Unit*) delega las operaciones gráficas hacia el hardware gráfico programable, el cual realizará el procesamiento en un marco especializado, quedando así liberado el primero para realizar tareas más acordes con el procesamiento general de los programas, resultando entonces en una tarea conjunta entre CPU y GPU.

#### Cómo usar Cg en aplicaciones gráficas

Cg ofrece una biblioteca conocida como su *runtime*. Bibliotecas como *OpenGL* o *Direct3D* deben comunicarse con el mismo para obtener los servicios que necesitan. Como componente del runtime, Cg posee un *compilador*, el cual traduce el código Cg a alguno de los *Perfiles* existentes. Un perfil es una combinación de las capacidades que otorgan la biblioteca y la GPU que poseemos para ejecutar los programas. Los perfiles en definitiva establecen limitaciones a lo que podemos lograr con un programa Cg. Es así que al compilar para un perfil determinado debemos tener en cuenta las restricciones del mismo. Sin embargo, la rápida evolución del hardware gráfico hace que las restricciones sean cada vez menores.

Existen además dos bibliotecas que se encargan de tomar el programa traducido por el compilador Cg y pasarlo a la biblioteca gráfica que estamos utilizando: *cgGL* (OpenGL) y *cgD3D* (Direct3D). Este código traducido es tomado por alguna biblioteca gráfica la cual luego se encargará de que el mismo corra en la GPU.

Antes de la aparición de lenguajes como Cg, el programador que pretendía tener un cierto control sobre cómo renderizar una escena, debía programar directamente el assembler de la placa gráfica correspondiente. Esto supone una pérdida de generalidad, puesto que debe hacerse un programa distinto para cada placa que pretenda usarse, además de un conocimiento preciso sobre el juego de instrucciones y el funcionamiento de cada placa. Con Cg, el programador se abstrae de todo esto.

#### El pipeline Gráfico y Cg

Así como en la CPU, la GPU posee un *pipeline*, esto es, una secuencia ordenada de pasos por cada instrucción de procesamiento. La secuencia típica está ordenada de la siguiente manera:

- Procesamiento de Vértices: los vértices de los objetos de una escena llegan desde el programa. Este es el primer momento en el que interviene Cg. Se da lugar entonces a la transformación personalizada por el programador sobre los mismos. Este tipo de transformaciones son las usuales sobre los vértices: rotación, desplazamiento, etc.

- Ensamblamiento y Conversión Scan: aquí estos vértices transformados son unidos para formar líneas, puntos y triángulos, mediante la información que llega desde el programa junto con los vértices. Luego, se realizan los procesos de *clipping*: eliminación de partes de objetos que por su posición no aparecerán en pantalla; y *culling*: eliminación de superficies no visibles, puesto que están detrás de otras superficies desde el punto de vista de la cámara. Aquellas partes que pasan estos filtros, son luego *rasterizadas*, esto es, *asociadas* a posiciones en pantalla; puntos, líneas y triángulos pasan entonces a tener una posición determinada, este proceso se conoce como *conversión scan*. Es así que se generan los denominados *fragmentos*, los cuales pueden definirse como "candidatos a ser píxeles": aquellos fragmentos que pasen filtros posteriores se convertirán en píxeles que serán representados en pantalla. Puede haber numerosos fragmentos que compitan por ser un píxel. Es en este momento que Cg hace su segunda aparición, el programador decide entonces cómo quiere generar los fragmentos.

- Interpolación, aplicación de texturas y colores: en esta etapa, los fragmentos generados en la etapa previa son *interpolados* (por ejemplo el color de los mismos), y luego se le aplican operaciones matemáticas y de texturas.

- Operaciones de Raster: en esta última etapa se llevan a cabo una serie de tests sobre cada fragmento. Si alguno falla, el fragmento es descartado, de lo contrario el color del fragmento actualizará el color del píxel correspondiente. Un fragmento podría no sobrevivir a esta etapa porque debido a su profundidad queda detrás de otro fragmento.

En la GPU existen dos componentes llamados *procesador de vértices programable* y *procesador de fragmentos programable*, los cuales se encargan de correr los programas Cg escritos por el programador, respectivamente llamados *Vertex Shader* y *Fragment Shader*.

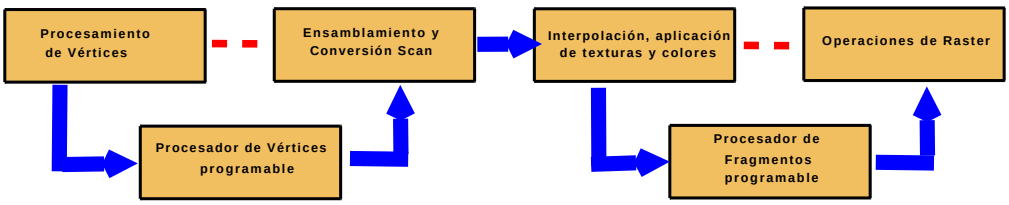



Figura 2.10: Pipeline de la GPU con shaders incluídos.

### Características de Cg - Generalidades

Presentamos un breve resumen de las pricipales características del lenguaje. Como se dijo antes existen dos tipos de programas en Cg, siendo sus nombres *Vertex Shader* y *Fragment Shader* respectivamente.

Se comienza analizando el primero. La función del programa es tomar los vértices que llegan desde la aplicación gráfica, transformarlos y pasarlos al siguiente eslabón del pipeline. De manera

simplificada, este shader recibe como entrada la posición del vértice en la escena, y terminada la ejecución devolverá el vértice con su posición modificada. También podría devolver el color y otros valores útiles que se puedan necesitar más adelante. Aquella información que reciba como entrada debe presentar una *semántica* asociada. Las *semánticas* hacen que un elemento del programa cobre una significación especial, como puede ser la posición, el color o una coordenada de una textura. Un ejemplo es:

```
float2 position : POSITION
```

Indicando en este caso que la variable position hace referencia a la posición. La semántica es la palabra en mayúsculas que aparece después de los dos puntos. Aquella información que devuelva el programa debe llevar la palabra clave *out*, la cual representa también una semántica. Es así que *out* indica que ese elemento representa una salida del programa. He aquí un ejemplo:

```
out float4 oPosition      : POSITION,
```

El Fragment Shader similarmente tiene parámetros de entrada y salida. Generalmente recibe el color interpolado por la etapa previa, también puede recibir información sobre texturas, codificada como coordenadas en las mismas. Aquí se presenta un ejemplo de las semánticas:

```
float4 color      : COLOR
float2 texCoord   : TEXCOORD0
```

Indicando que las variables *color* y *texCoord* son parámetros de entrada. El cero al final del nombre de la segunda semántica indica que nos referimos al primer par de coordenadas que se pueden recibir. Existen varios más, dependiendo del perfil para el que se compile.

Este shader a diferencia del primero, luego de realizar el procesamiento, sólo puede devolver un elemento, el *color* (excepto por algunos perfiles avanzados). Esta salida representa el color que, de pasar los diversos tests, tomará el píxel en pantalla.

Existe otro tipo de objetos de entrada, los cuales llevan la palabra *uniform*. Los mismos representan objetos que son recibidos por el shader desde la aplicación. El principal atractivo de este tipo de objetos es hacer interactivos a los shaders, es decir, que desde la aplicación el usuario puede configurar valores que serán utilizados en los shaders.

### Tipos de Datos

Presentes en Cg están los tipos de datos conocidos: *int*, *float*, y otros menos conocidos como *half* (punto flotante de menor precisión, el cual es usado por eficiencia, puesto que requiere menor cantidad de recursos), así como *arreglos*.

Al tener un enfoque gráfico, el lenguaje presenta tipos de datos de primer orden diferentes de lenguajes como C. Ejemplo de esto son los *vectores* y las *matrices*. Como se ha visto, Cg tiene palabras reservadas para estos tipos de datos.

```
float f;
int a[4];
float2 v;
float4x4 m;
```

Aquí se definen un flotante (f), un array de cuatro elementos enteros (a), un vector con dos componentes flotantes (v) y una matriz de 16 elementos de tipo flotante (m). La diferencia entre un arreglo y un vector es también la eficiencia. Un vector es en realidad un *packed array*.

Otros tipos de datos son los llamados *samplers*. Estos representan objetos externos a Cg, como por ejemplo una textura. Existen distintos tipos de samplers: *sampler1D*, *sampler2D* y

*sampler3D*, representando respectivamente, objetos de una, dos y tres dimensiones. También tipos especiales como *samplerCUBE* (utilizado para técnicas como *environment mapping*) y *samplerRECT* (no posee dimensiones que son potencias de dos). Un ejemplo de uso es el siguiente:

```
function f(     float2 texCoord : TEXCOORD0,
                uniform sampler2D decal)
{
        tex2D(decal, texCoord);
}
```

La rutina tex2D consulta el objeto decal, en las coordenadas especificadas en texCoord, las cuales llegan como parámetro al programa. Este constituye un ejemplo típico de *texture mapping*.

#### Más sobre el lenguaje

Existe una biblioteca estándar que contiene funciones muy útiles para el programador. Así presenta funciones para realizar cálculos usuales como máximos, mínimos, redondeos, seno, coseno, valores absolutos; y se le suman funciones para el trabajo con vectores, matrices y texturas, como *producto escalar*, *producto vectorial*, multiplicación de matrices, multiplicación de matriz y vector (y viceversa), obtención del valor de *texels* (como se ha visto).

En cuanto al lenguaje en sí, se utilizan *funciones* de la manera usual y con sintaxis similar a C. También posee una función *principal* que se comporta como *main* en C. La misma se especifica en la aplicación gráfica al crear una instancia de un programa Cg, la cual entonces representa el punto de entrada al programa. También se tienen estructuras de control usuales, como bucles *for* y *while*, aunque con determinadas restricciones dependiendo del perfil. Se presenta a continuación otro tipo de lenguaje de programación, el cual permite realizar cálculo de propósito general (GPGPU) en la GPU, a diferencia de Cg, que está diseñado para utilizarse únicamente con fines gráficos.

### 2.6.3. CUDA

#### Introducción

La capacidad de procesamiento de las placas gráficas ha ido evolucionado de manera vertiginosa hasta llegar al punto no sólo de rivalizar sino de superar en gran medida a los procesadores que se utilizan hoy en día. Las mismas disponen de un número de *cores* en aumento por lo cual aparecen como dispositivos con un alto procesamiento paralelo. Además, la velocidad de transmisión de datos en su propia memoria también es mucho mayor a la de un CPU tradicional. Esto ocurre porque la CPU debe dedicar parte de su poder a control de flujo y *caching* de datos, en contraposición la GPU está diseñada para resolver problemas con un alto contenido de paralelismo; se debe aplicar el mismo programa a cada dato por lo cual el control de flujo es menos necesario. Por estas razones, no es descabellado comenzar a pensar en utilizar la misma ya no como un dispositivo orientado únicamente a procesamiento de gráficos, sino uno de procesamiento general. Basado en este razonamiento, la empresa NVIDIA lanza al mercado CUDA (siglas en inglés de Compute Unified Device Architecture), una arquitectura de computación paralela de propósito general.

#### Generalidades

En esta metodología de programación, existen múltiples threads ejecutando en paralelo. Los threads son lógicamente agrupados en *blocks* y son enumerados con un índice de dos elementos,

así hablamos del thread (i,j) de un determinado block. Los threads dentro de un block se ejecutan en el mismo core. La cantidad de threads en un block queda limitada por los recursos de memoria del core de la GPU. Los blocks presentan la misma organización, siendo categorizados en *grids*. Por lo tanto también se habla del block (x,y) de un determinado grid. La memoria total de la GPU es la memoria de sus grids y es llamada memoria global. Cada thread tiene su memoria local, sólo visible en ese thread. A su vez, un block tiene su memoria local, la cual es compartida por todos los threads del mismo. Por último, todos los threads tienen acceso a la memoria *global*. Además, están presentes la memoria de texturas y la memoria constante, las cuales son de sólo lectura. Estas dos y la memoria global se preservan entre llamadas a *kernels* (ver siguiente sección).

### Cómo programar con CUDA

El software que provee CUDA permite a los programadores trabajar en un entorno C. En un futuro otros lenguajes como C++ y FORTRAN también serán soportados.

Existen dos formas de escribir programas CUDA: *C para CUDA* y la *API del driver de CUDA*. En el primer caso, se presenta como una extensión a dicho lenguaje junto con un *runtime*. El programador debe escribir funciones a ser ejecutadas en la GPU llamadas *kernels*. La principal diferencia con una función C convencional es que en lugar de ejecutarse secuencialmente, existen N threads paralelos ejecutando N instancias de dicha función. El creador del programa debe razonar el mismo como varios procesos a ejecutar en paralelo, los cuales unidos resuelven el problema.

He aquí un ejemplo de un kernel, el cual suma dos vectores:

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

(Aclaraciones: global se usa para definir el kernel y denota, el carácter global del mismo. La variable reservada threadIdx contiene el ID del thread dentro del block, que ejecuta la operación actual, la misma puede tener más de una dimensión, aunque en este caso tiene una sola y por lo tanto se accede al miembro x de la misma. Para entender el kernel, debe comprenderse que cada thread sumará un sólo elemento del vector final). Ejecutar este kernel es tan sencillo como una llamada en C, con una salvedad: debe especificarse la cantidad de blocks por grid (A) y la de threads por block (B), bajo la sintaxis:

$$<<< A, B >>>$$

```
int main() {
    int N = 10;
    float* A;
    float* B;
    float* C;
    VecAdd<<<1, N>>>(A, B, C);
}
```

El driver de CUDA es una API de un nivel de abstracción más bajo que el runtime. De hecho, el runtime está escrito sobre este API. Su idea principal es cargar assembler o código binario CUDA, y poder ejecutarlo. Este código se obtiene compilando un kernel. En este modo tenemos

más control sobre lo que ocurre en la GPU, pero también es más dificultosa la programación y el *debug* del código. Ambos modos proveen métodos para alocar memoria en la GPU, transferirla al CPU, manejar múltiples GPU's, etc. C para CUDA tiene la opción de emular dispositivos físicos en ausencia de ellos, lo cual puede resultar útil en caso de no estar los mismos presentes.

### NVCC

CUDA provee un compilador llamado *nvcc*. El mismo se encarga de traducir nuestro código hacia código objeto. Presenta una línea de comandos similar a otros compiladores C como gcc, para facilitar su uso. Normalmente, un fuente contiene código del host y del device mezclado. Por lo tanto, el esquema normal de trabajo del compilador consiste en separar código a ejecutar en el host del código a ejecutar en el device y traducir este último hacia assembler (*PTX*) o código objeto (*cubin*). El código del host encontrado en este proceso debe luego ser compilado con un compilador C o bien nvcc puede invocar al mismo y luego producir código objeto directamente.

### cufft

Esta biblioteca implementa algoritmos que permiten aplicar la transformada de Fourier directa e inversa a imágenes de una o más dimensiones. De este modo, podremos realizar la convolución de dos funciones aplicando sólo transformadas directas e inversas (por el teorema de convolución). Como su nombre lo indica, la biblioteca utiliza el algoritmo *fft* descripto en secciones previas. En este trabajo la misma será utilizada más adelante en el tópico *spot noise*.

Habiendo presentado la teoría que sustenta al framework, estamos en condiciones de presentar el modelo matemático del mismo. Como se verá, la mayoría de los conceptos mencionados han sido utilizado durante el desarrollo del framework.

# Capítulo 3

# Modelo Matemático

*"Primero, quiero asegurarles que esto no es una de esas pirámides fraudulentas de las que hablan. ¡No señor!. Nuestro modelo es el trapezoide, que garantiza a cada inversionista,* 800 % *de utilidad en su primer..."*
*Los Simpsons - "Me casé con Marge"*

## 3.1. Introducción

Se procede en este capítulo definir formalmente el modelo matemático que es utilizado para producir los materiales que se pretenden sintetizar. Existe un conjunto de funciones base que es sometido a operaciones usuales de texture mapping, para luego combinarse entre ellas y producir imágenes más complejas. Por otro lado, la combinación de diferentes texturas permite razonar a los materiales como imágenes compuestas por distintas *capas* que permiten modelar diferentes características de los mismos (pulimiento, desgaste de una superficie, colores, etc.). Cada operación de combinación permite construir materiales de una forma diferente, y en ciertas ocasiones texturas similares pueden ser obtenidas por medio de la aplicación en diferente orden de distintos conjuntos de operaciones, si bien este no es el caso más común. Por medio de esta formalización, será posible realizar una implementación sencilla y eficiente.

## 3.2. Esquema del Framework

A continuación se presentan las principales características sobre las cuales está basado el modelo, el cual puede observarse en la Figura 3.1. El mismo provee una *interfaz pública* la cual representa el nivel de abstracción más alto del framework. A través de ella será posible solicitar la síntesis de una determinada textura. La definición exacta de esta interfaz se ilustrará en los capítulos a seguir. Por ahora puede pensarse la misma como un conjunto de operaciones que permiten combinar determinadas texturas, por medio de parámetros que recibe.

Con estos parámetros, en el framework luego tiene lugar el proceso solicitado. Para este proceso, se propone aquí una *base de texturas*. Está formada por una cantidad fija de elementos (texturas) que intentan capturar las principales características morfológicas de los materiales que se buscan. Las texturas presentes en esta "base" representan por sí mismas una abstracción, respecto a las múltiples formas que presentan las texturas buscadas. A través de determinadas elecciones de estos elementos y operaciones entre ellos, se obtendrán diferentes materiales.

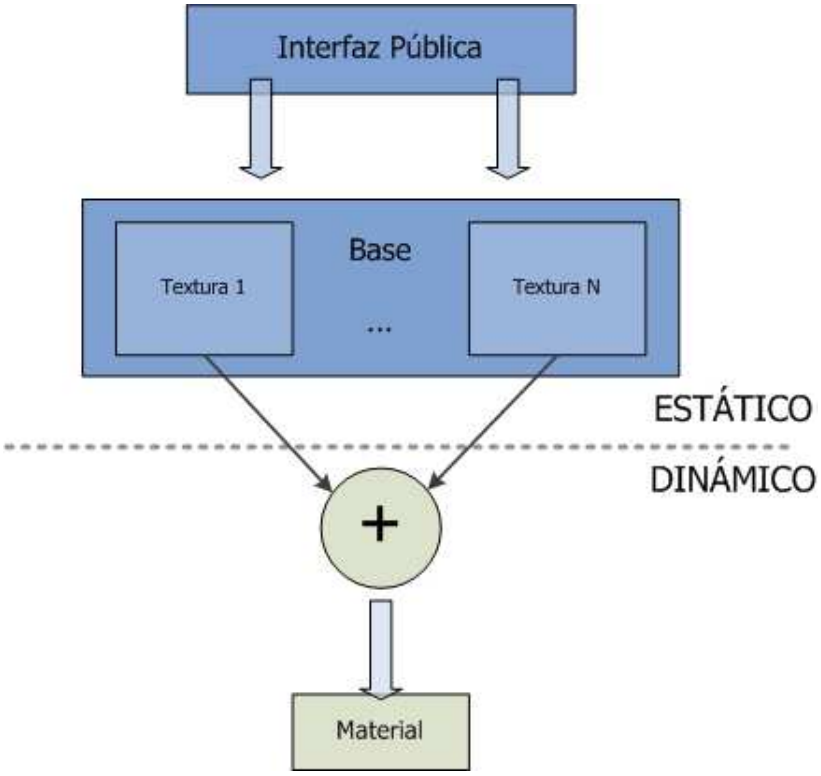



Figura 3.1: Idea principal del Framework

Se define entonces un *conjunto* de texturas que llamaremos *base*:

$$base = \{t_1, ..., t_n\},$$

$$t_i = \mathbb{R}^2 \rightarrow [0, 1], i \in [0, ..., n].$$

Si bien originalmente cada elemento de la base tiene como dominio el subconjunto del plano cartesiano $[0, 1] \times [0, 1]$, las coordenadas de estos sufren transformaciones que producen que estas se extiendan fuera de ese rango (las operaciones serán presentadas en la siguiente sección). El codominio de cada $t$ está definido en el $[0, 1]$ permitiendo que la textura resultante sea interpretada directamente como una imagen a escala de grises.

En un espacio vectorial, la combinación lineal de vectores de una base dada de ese espacio produce otros vectores (de hecho produce *todo* el espacio). Sin embargo aquí la idea de base debe entenderse abstractamente. Este concepto permite entender el proceso que realiza el framework, pero vale aclarar que no representa una base en el sentido matemático de la palabra.

Cada textura podrá someterse a operaciones usuales de texture mapping, más específicamente, rotación, escalamiento y desplazamiento, además de otras operaciones útiles que serán presentadas. Posteriormente, cada textura se combinará con otro u otros elementos de esa base, para producir otras texturas como resultado de esa combinación. La combinación en sí misma presentará diversas operaciones, cada una de las cuales tendrá un propósito específico, y servirá para producir distintos materiales.

De este proceso se obtendrá una textura en *escala de grises*. El framework permitirá luego la aplicación de un color *RGB*. Debido a que se produce una imagen a escala de grises, se presenta la

posibilidad de utilizar la textura resultante como un mapa de alturas (heightmap), para utilizarlo en la aplicación de bump mapping o displacement mapping. Trabajos posteriores podrán hacer uso de esta observación para implementar dichos algoritmos. Se detallan a continuación los distintos tipos de operaciones con los que cuenta el modelo.

## 3.3. Operaciones sobre la base

### 3.3.1. Tipos de operaciones

El flujo básico de procesamiento en el framework constará de las siguientes etapas ordenadas:

- Operaciones Unarias: las mismas se aplicarán a cada elemento de la base. Son las operaciones usuales de texture mapping, como fue previamente mencionado.
- Operaciones Especiales: cada material a estudiar ofrecerá operaciones útiles que serán tenidas en cuenta en este punto.
- Operaciones Binarias: representan la combinación de las texturas de la base, transformadas por las operaciones previas.

Se mencionan a continuación las operaciones unarias y binarias, puesto que las especiales serán expuestas más adelante en cada material.

### 3.3.2. Operaciones Unarias

Se detallan a continuación operaciones unarias sobre coordenadas. Dados $(u, v) \in [0, 1] \times [0, 1]$, se definen las siguientes operaciones:

- Traslación: desplazar las coordenadas $(u, v)$ por medio del vector desplazamiento $(x, y)$ con $x, y \in (-\infty, +\infty)$

  $traslacion(u, v, x, y) = (u + x, v + y)$.

- Escala: escalar las coordenadas $(u, v)$ por medio del vector escala $(x, y)$ con $x, y \in (0, +\infty)$

  $escala(u, v, x, y) = (u * x, v * y)$.

- Rotación: rotar las coordenadas $(u, v)$ respecto al centro de la textura (coordenadas (0.5,0.5)) un ángulo $\alpha$, donde $\alpha \in [0, 2\pi]$

$$(u', v') = (u, v) - (0{,}5, 0{,}5),$$

$$u'' = cos(\alpha) * u' - sin(\alpha) * v',$$

$$v'' = sin(\alpha) * v' + cos(\alpha) * v',$$

$$rotacion(u, v, \alpha) = (u'' + 0{,}5, v'' + 0{,}5).$$

Nótese que las coordenadas resultantes de las operaciones no están limitadas al segmento [0,1]. Estas son luego evaluadas en la textura correspondiente de la base. Por esta razón, la textura debe estar definida en esas coordenadas. El resultado de esa evaluación es asociado a las coordenadas $(u, v)$ originales.

Por lo tanto, una textura $t_i$ resulta transformada en otra $t'_i$, definida en el subconjunto del plano $[0,1] \times [0,1]$, donde:

$$t'_i(u,v) = t_i(Ops((u,v)), \forall (u,v) \in [0,1] \times [0,1].$$

*Ops* representa la composición de un subconjunto ordenado de operaciones unarias. Por ejemplo, a un elemento de la base se le podría desear aplicar una rotación y una escala ($(u,v)$ coordenadas en el espacio de la textura):

$$t'_i(u,v) = t_i(rotacion(escala(u,v,0{,}2,0{,}3),\pi)).$$

La textura resultante es la original escalada un 20 % en la dirección u, un 30 % en la dirección v y rotada un ángulo $\pi$ sobre su centro. La decisión de rotar la textura respecto al centro de la misma provoca que la textura gire sobre sí misma, en lugar de hacerlo sobre el (0,0). Adicionalmente, esto permite que se pueda rotar una textura y luego desplazarla, o desplazarla y luego rotarla, obteniendo el mismo resultado (ídem traslación). Por lo tanto, el orden de las operaciones unarias no cambiará el resultado final.

Las operaciones unarias se aplican sobre cada textura antes de cualquier otra operación, de manera independiente para cada textura. A cada elemento de la base se le puede aplicar cualquier subconjunto de ellas, en cualquier orden.

### 3.3.3. Operaciones Binarias

Se definen operaciones que permiten combinar de diferentes formas las texturas transformadas de la base entre sí. Un estudio completo sobre este tema puede verse en [12].

Sean $t_1, t_2 \in base, t_1 \neq t_2$, y sean

$$t'_1 = Esp_m(...(Esp_1(t_1 \circ Ops_1, arg_1^1), ...), arg_m^1),$$

$$t'_2 = Esp_m(...(Esp_1(t_1 \circ Ops_2, arg_1^2), ...), arg_m^2),$$

donde $Ops_i$ representan operaciones unarias como fue mencionado en la sección previa. Cada $Esp_i$ representa una operación especial. El framework presentará $m$ de ellas, que serán comunes a todos ("$\circ$" significa composición de funciones).

Estas operaciones especiales toman exactamente un argumento cada una, estando representado en las ecuaciones por $arg_k^i$, con $i \in \{1,2\}, k \in \{1,...,m\}$ el cual será posible controlar para sintetizar diferentes texturas. Los dominios de estos argumentos también variará para cada uno de ellos, por esta razón en esta instancia se dejan sin especificar.

Sea además $(u,v) \in [0,1] \times [0,1]$. Se definen las siguientes operaciones $\forall (u,v) \in [0,1] \times [0,1]$:

- Suma:
$$ADD(t'_1, t'_2)(u,v) = t'_1(u,v) + t'_2(u,v).$$
- Multiplicación:
$$MUL(t'_1, t'_2)(u,v) = t'_1(u,v) * t'_2(u,v).$$
- Resta:
$$SUB(t'_1, t'_2)(u,v) = t'_1(u,v) - t'_2(u,v).$$

- Interpolación Lineal (con peso):

$$LERP(t'_1, t'_2, \beta)(u, v) = (1 - \beta) * t'_1(u, v) + \beta * t'_2(u, v), \beta \in [0, 1].$$

- Superposición (se asume como transparente al valor 0):

$$SOBRE(t'_1, t'_2)(u, v) = \begin{cases} t'_1(u, v), t'_1(u, v) \neq 0, \\ t'_2(u, v), t'_1(u, v) = 0. \end{cases}$$

Se permiten hacer combinaciones entre todas las texturas de la base. Así, un material podría estar definido de la siguiente forma:

$M_1 = SUB(LERP(t_1, t_2, 0{,}5), t_3)$

Mientras que otro podría sintetizarse de la siguiente manera:

$M_2 = ADD(ADD(t_1, t_2), t_3)$

En general, se permitirán combinar texturas de la base de la siguiente forma:

$M = Op_1(t_1, Op_2(t_2, ..., args_2), args_1)$, donde $args_i$ con $i \in [1..n]$ representa el conjunto (posiblemente vacío) de argumentos para la operación $i$, y $n$ es el cardinal del conjunto de texturas de la base.

Además, cada textura podrá o no estar presente en la combinación final. De esta manera el material resultante podrá utilizar de 1 a $n$ texturas de la base. Nótese que cada textura puede combinarse una sóla vez.

## 3.4. Casos de estudio

Como fue mencionado con anterioridad, se analizará un material en particular en primera instancia. Luego se repetirá el proceso con otros materiales, permitiendo así una abstracción útil para sintetizar a todos ellos. Se analizarán en primer lugar mármoles sencillos, más específicamente, aquellos que presentan vetas en una dirección (horizontal o vertical).

### 3.4.1. Características Morfológicas del mármol

La primera decisión que debe tomarse es escoger una textura de la "base" que capture las principales características morfológicas del material que pretendemos sintetizar. Hasta aquí no hemos mencionado cuáles eran las texturas presentes en la base. Esto es así porque estas dependerán de cada material. Resulta intuitiva la elección de una textura muy sencilla: $sen(u)$, $u \in (-\infty, +\infty)$.

$$t_1(u, v) = sen(u).$$

Esta elección está basada en la elección de la misma textura utilizada en [10]. Esto da lugar a una textura a franjas alternadas blancas y negras en sentido vertical. Dependiendo de la escala habrá mayor o menor cantidad de estas franjas. El mármol que pretendemos sintetizar tiene generalmente diversas *vetas* de distintas frecuencias, con distintas orientaciones. Podemos entonces asociar una textura base $sen(u)$ con cada veta para controlarlas independientemente. Tenemos entonces:

$$t_i(u, v) = sen(u), i \in \{1, ..., m\}.$$

La coordenada $v$ de cada punto en la textura no es tenida en consideración en los cálculos. Por otro lado, al controlar las vetas de manera independiente, existe la libertad de modelar distintas propiedades en el material. Así, podemos modelar el pulimiento en una de las texturas, el desgaste de la superficie en otro, etc.

### 3.4.2. Parámetros de los mármoles

A continuación se presentan los parámetros con los cuales se permite generar distintos tipos de mármoles. En primer lugar se hace un resumen informal de cada uno de ellos, para luego proceder a su formalización. En la Figura 3.2 se muestra el significado de los parámetros en mármoles reales.

- Amplitud: en este tipo de material, se puede distinguir que, además de distintas frecuencias, las vetas presentan distinta amplitud. Una veta puede estar presente una determinada cantidad de veces, con mayor o menor amplitud. De esta forma podemos controlar el "ancho" de cada una de ellas. Más específicamente, el control es sobre las franjas blancas[1]. Así, mientras mayor es el valor de este parámetro, menor es la amplitud de las franjas blancas. Como se verá en el resto de los materiales, este razonamiento es válido para la mayoría de ellos. En la Figura 3.2 la imagen de la izquierda presenta una amplitud moderada, y la de la derecha muestra una amplitud menor, ya que las franjas blancas poseen un ancho mayor.

- Intensidad: se pretende variar la intensidad de cada veta, obteniendo texturas con mayor o menor oscuridad. De esta manera podemos controlar la "luminosidad" que aporta cada una de ellas. La imagen a la izquierda de la Figura 3.2 presenta menor intensidad que la de la derecha: la primera es más opaca.

- Turbulencia: se establece el control "nivel de ruido". Con este parámetro es posible "deformar" cada veta, lo cual produce texturas que no resultan excesivamente periódicas. Muchos mármoles constan de diferentes vetas que han sufrido un proceso de deformación de mayor o menor intensidad, lo cual produce su apariencia final. En la Figura 3.2, se observa un mayor nivel de turbulencia en las vetas en la imagen de la derecha; la imagen de la izquierda presenta vetas con estructura cercana a las vetas sin deformar.

Figura 3.2: Parámetros utilizados en el modelo explicados en mármoles reales.

Los primeros dos parámetros representan operaciones especiales. Sin embargo, el tercer parámetro representa una operación unaria, según el presente modelo. Esta operación unaria sí depende del orden, como se verá a continuación.

[1] Franjas blancas en la textura sin colores.

El parámetro turbulencia se implementará utilizando la idea de turbulencia presente en [10]. Con el propósito de establecer simpleza en esta instancia, teniendo en cuenta la performance del modelo, además de introducir una idea potencialmente útil, se utilizará sólo el primer término de la sumatoria de la definición en [10], pero tomando como textura *Noise()* a una textura resultante del método spot noise [13] que será utilizada en lugar de la textura que propone el autor. Como se obervará, no se presentarán en este momento grandes diferencias visuales por esta elección, debido a que la aplicación de este parámetro en este momento se hará dentro de unos valores muy acotados (pues estos mármoles no presentan mucha turbulencia). Más adelante se volverá sobre este punto.

Se define entonces turbulencia como:

$$turbulencia(u, v, c) = (u + c * Noise(u', v'), v), (u, v) \in \mathbb{R}^2, (u', v') \in [0, 1] \times [0, 1].$$

$(u', v')$ son las coordenadas que asume cada píxel sin que se apliquen las operaciones unarias. De esta forma se desplaza la coordenada $u$ una cantidad aleatoria controlada por $c$, si $c = 0$ la textura no presenta turbulencia, en caso contrario ésta es proporcional a $c$.

Volvemos a definir $t'_i = t_i \circ Ops_i$.

Las operaciones especiales se definen de la siguiente manera:

$$amplitud(x, a) = x^a, a \in [0, +\infty].$$

$$intensidad(x, i) = i * x, i \in [0, 1], x = t'_i(u, v), (u, v) \in [0, 1] \times [0, 1].$$

Se observa que $x \in [0, 1]$, puesto que es el valor que asume la textura de la base $t_i$. Podemos observar también, que luego de aplicadas las operaciones especiales, el valor devuelto por éstas continúa perteneciendo al rango $[0, 1]$.

La ecuación de cada veta resulta entonces definida de la siguiente manera [4]:

$$v_i = i_i \left( \frac{sen(u') + 1}{2} \right)^{a_i},$$

donde:

$i_i$ : intensidad,

$a_i$ : amplitud,

$(u, v)$ : coordenadas originales (sin operaciones unarias),

$(u', v')$ : coordenadas resultantes de aplicarles operaciones unarias a $(u, v)$.

Puede observarse en la ecuación que se introdujo un cambio de rango entre el valor que toma la textura base y las operaciones especiales, obteniendo valores que están presentes en el cerrado $[0, 1]$ (suma y división dentro de la expresión). Para este material, utilizaremos 3 texturas base, simulando respectivamente 3 vetas.

Se presentan algunos ejemplos resultado del soft que implementa el modelo anterior. En la Figura 3.3 se presentan tres ejemplos de texturas de mármol obtenidas (en las dos imágenes de la izquierda se utilizan tres texturas sen(u) y se suman, éstas presentan muy poca variación de frecuencia entre sí. En la imagen central, una de las vetas presenta más turbulencia que las otras. La tercer imagen utiliza una sola textura con turbulencia y poca amplitud).

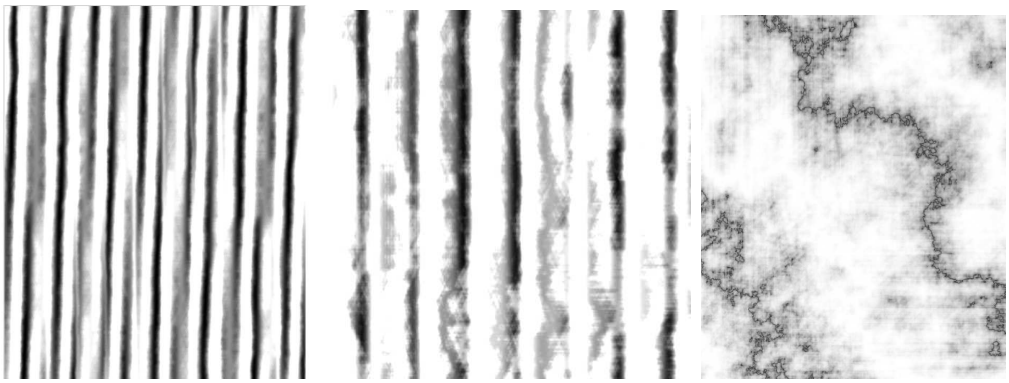


Figura 3.3: Mármoles Sintetizados.

Los resultados observados para una única textura son similares a los presentados en [10]. Con el agregado de otras vetas, es posible representar mármoles que presentan mayor detalle respecto a las diversas vetas que presentan los materiales reales. Se procede a analizar el segundo caso de estudio.

### 3.4.3. Madera

Se considerarán ahora algunos tipos de madera conocidos, intentando buscar similitudes con el caso anterior. De ser así, podríamos usar los mismos elementos para representar ambos materiales, resultando más sencilla su generalización y comparación con otras texturas.

### 3.4.4. Características Morfológicas de la madera

Algunos tipos de madera pueden ser representados de la misma forma que el mármol del caso de estudio anterior, debido a que presentan la misma estructura a franjas en una dirección con distinta apariencia entre ellas. Bastaría con usar un colormap distinto para producir una textura de madera en lugar de una de mármol.

Además de éstas, existen otras que están formadas por esferas concéntricas de similar amplitud. Cortes arbitrarios en esta estructura [8] producen las imágenes que pueden observarse en muebles, puertas o cualquier utensilio presente en la vida cotidiana. Una de sus principales características estructurales es la aparición de estas esferas, pero transformadas en círculos concéntricos con colores alternados. Debe agregarse que, debido a deformaciones propias durante el proceso de crecimiento de los árboles, estos círculos aparecen en forma de elipses con cierto nivel de turbulencia (similar a aquella presente en los mármoles vistos).

### 3.4.5. Parámetros de la madera

Se necesita hacer un breve análisis del concepto de turbulencia presentado en el caso de estudio anterior para entender la forma en que se trabajará con el presente material. Allí se establecía que ésta se producía al desplazar aleatoriamente (de acuerdo a una función de ruido) las coordenadas de las vetas del mármol. Es perfectamente posible controlar ese deplazamiento, utilizando otras funciones. Por ejemplo, en lugar de usar una función de ruido, se podría utilizar una función seno que desplace las coordenadas tomando la variable opuesta en el espacio de la textura. Lo que se producen son “tiras” de funciones seno en una dirección, como muestra el siguiente gráfico.

Figura 3.4: Tiras de funciones seno en una dirección de la textura

La ecuación de la textura que produce el gráfico es la siguiente:

$$sen(u + sen(v)).$$

se deduce que muchas funciones periódicas podrían ser representadas de esta manera, repitiéndose sobre una coordenada. Se observa que multiplicando por un real esta función seno que se comporta como turbulencia, éstas se acercan unas a otras, resultando la siguiente ecuación:

$$sen(u + \alpha sen(v)).$$

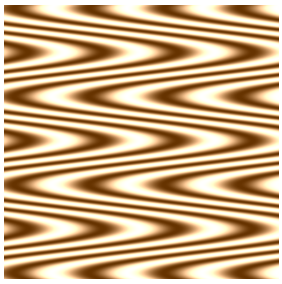


Figura 3.5: Funciones seno multiplicadas por un número real.

Lo cual presenta similitudes con las formas que se pueden observar en este material (en la Figura 3.5 $\alpha = 10$). Se concluye que esta función es un buen candidato para formar la base de este material. De este modo, la ecuación base para producir estas texturas es un caso más general que el se usó para producir texturas de mármol. Es decir, la diferencia radica en el agregado de $\alpha sen(v)$ dentro de la función.

El número real $\alpha$ será un parámetro de este material, y permitirá controlar la amplitud de la onda seno que está siendo representada en las tiras. Además de este, se agrega otro parámetro en la ecuación de la siguiente forma:

$$sen(\alpha u + \beta sen(v))$$

de esta forma $\alpha$ servirá para controlar la *frecuencia* de la onda seno representada. A medida que $\alpha$ se acerca a 0, menor frecuencia tiene la onda, y viceversa.

Con un parámetro más se puede dar una forma sutilmente diferente a esta estructura. El agregado de $sen(u)$ de la misma manera que se sumó $sen(v)$ produce mejores resultados. La Figura 3.6 muestra una textura con este nuevo parámetro (con $\beta = 2$).

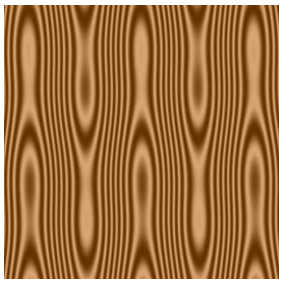


Figura 3.6: Tercer parámetro de la ecuación que sintetiza madera

Resumiendo, la ecuación de una textura de madera queda representada de la siguiente manera:

$$m = i_i \left( \frac{sen(\alpha u' + \beta sen(u') + \gamma sen(v')) + 1}{2} \right)^{a_i}.$$

Donde $(u', v')$ es el resultado de aplicarle operaciones unarias a $(u, v)$, y los demás parámetros son los presentados con anterioridad. Cabe destacar, que esta ecuación no es más que un caso más general de la ecuación utilizada para representar mármoles. Dicha ecuación es un caso particular de esta, seteando $\alpha = 1, \beta = 0, \gamma = 0$. En la Figura 3.7 pueden observarse algunas texturas generadas con esta ecuación, con diferentes colores ilustrativos.

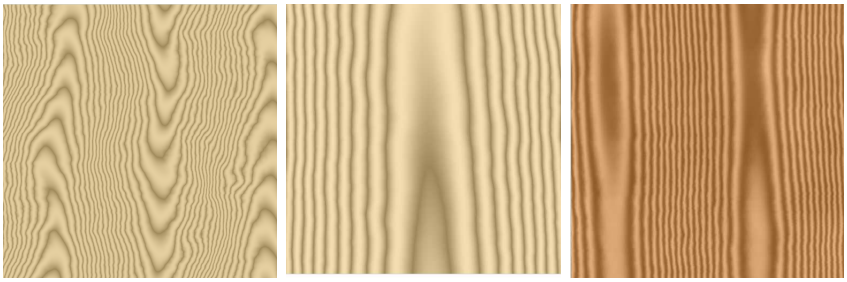


Figura 3.7: Diferentes texturas de madera

### 3.4.6. Granito

Se procede a analizar otro material de uso muy extendido en ámbitos urbanos: el granito. Pueden observarse estas texturas en diversos edificios públicos (por ejemplo granito rojo) así como en hogares (cocina, paredes, puertas, ventanas, etc.).

### 3.4.7. Características Morfológicas del granito

Este material está definido por poseer formas que presentan una morfología aleatoria, como puede observarse en las imágenes[2] de la Figura 3.8.

También puede distinguirse una combinación de unos pocos colores principales. En la imagen que se sitúa a la izquierda, puede observarse la aparición de tres de ellos: rojo, verde, y blanco. Esto en principio, podría modelarse con la combinación de tres texturas, cada una definida

[2]Fuente: http://es.wikipedia.org

Figura 3.8: Diferentes texturas reales de granito.

con uno de estos colores. Si bien en realidad, con menor cantidad de texturas y un colormap adecuado se produce el mismo efecto. Se mostrará este comportamiento en el desarrollo de este caso de estudio. Cabe destacar que al igual que en el mármol, el granito toma su forma final debido a procesos físicos que combinan diferentes materiales presentes en distintas capas de la superficie terrestre, con la diferencia de que en este caso, esos procesos físicos hacen que este material sufra transformaciones de mayor intensidad, ya que el granito necesita para formarse solidificarse lentamente y a una *presión* elevada[3].

### 3.4.8. Parámetros del granito

En este material se utilizará también la idea de turbulencia presentada en el caso de estudio anterior. En primer lugar, podemos tomar la ecuación del mármol del primer caso de estudio, y aplicarle una turbulencia mayor (siguiendo la idea mencionada previamente), buscando "deformar" en mayor medida las texturas producidas para dicho material. Utilizando entonces una textura resultante del método spot noise (tomando un spot sin una forma específica), se obtiene una imagen como la de la Figura 3.9.

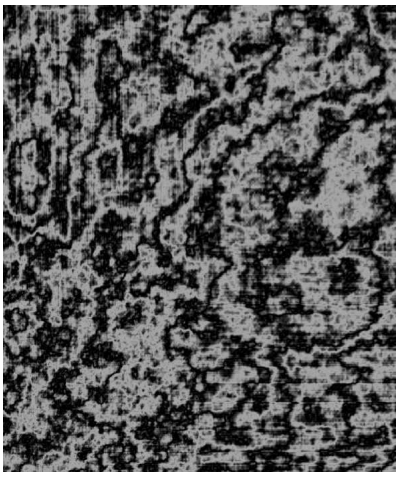


Figura 3.9: Turbulencia a partir de spot noise.

[3]Fuente: http://es.wikipedia.org/wiki/Granito

Se observa cierta similitud, en cuanto a las formas, con las texturas que se pretenden sintetizar. Esto parece indicar que la misma ecuación que describía al mármol es un buen candidato para representar texturas de granito. La única diferencia radica en la aplicación de una turbulencia mayor. Se muestra a continuación que dicha idea es adecuada, debido a los resultados obtenidos.

Las operaciones definidas para el mármol resultan útiles para distintos aspectos del presente material. El parámetro *amplitud* permite controlar la distancia existente entre las formas generadas en la imagen, permitiendo regular el "grosor" de las formas, mientras que con el parámetro *intensidad* se puede controlar el nivel de luminosidad, de manera análoga al mármol. En la Figura 3.10 puede observarse el efecto de la utilización del parámetro amplitud, la imagen de la izquierda de dicha figura presenta una amplitud menor que la de la derecha (esto puede observarse en una mayor aparición de color blanco en la imagen izquierda).

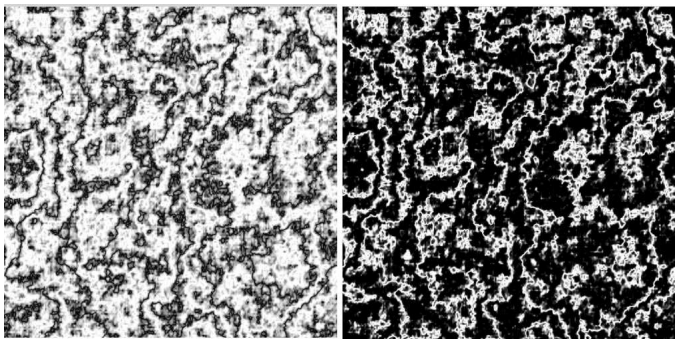


Figura 3.10: Efecto del parámetro amplitud en la generación de granito.

Las operaciones unarias siguen teniendo el mismo significado, y pueden ser utilizadas como fue descrito en secciones previas. En la Figura 3.11 se presentan algunos ejemplos de texturas junto a su textura sintetizada (utilizando colores ilustrativos, en cada par de imágenes, la textura situada a la izquierda es el material fotografiado, y la textura de la derecha es la sintetizada), mostrando que la elección de la textura base resulta correcta. Todas ellas se sintetizaron combinando de una a tres texturas (por medio de la operación binaria *LERP*) con las características mencionadas. En la textura de granito rojo, el color blanco presente representa el *pulido* de la misma. Se utilizó una textura de color blanco para simular este efecto. Podemos concluir entonces, que las texturas de mármol y granito que se pretendieron sintetizar, utilizan la misma textura base, difiriendo solamente en un mayor o menor uso de un parámetro: *turbulencia*.

Hasta aquí tenemos una única ecuación base que describe de manera adecuada tres materiales muy conocidos y diferentes entre sí. La ecuación resultó consecuencia del estudio de la formación de dichos materiales y de observaciones directas sobre las características morfológicas de los mismos. Por medio de parámetros que recibe la misma, es posible sintetizar cada una de las imágenes necesitadas. Cabe preguntarse si otras texturas podrán ser representadas por medio de dicha función. En el siguiente capítulo se describe la generalización del modelo, el cual es un intento de sintetizar más materiales a partir del framework presentado.

Izquierda: textura real. Derecha: material sintetizado

Figura 3.11: Granitos sintetizados.

# Capítulo 4

# Generalización

*"Por el alcohol, la causa y la solución de todos los problemas de la vida"*
*Homero Simpson - Los Simpsons - "Homero contra la prohibición"*

Se ha presentado el framework, desde su punto de vista matemático, mostrando su capacidad para representar determinados materiales. Se han visto hasta aquí tres de ellos, los cuales fueron presentados como casos de estudio: mármol, madera y granito. En este capítulo se pretende generalizar la capacidad de representación del modelo, extendiendo la misma hasta donde sea posible. Se comenzará analizando un tópico con posibles aplicaciones en la misma generalización. Luego se aplicará una generalización a los elementos de la base de texturas.

## 4.1. Turbulencia con distintos spots

Utilizando como función de ruido a distintas texturas resultantes del proceso de spot noise, generadas a partir del uso de diferentes spots, se generan texturas de granito con diferentes características, como puede observarse en la Figura 4.1. Si por ejemplo utilizamos un triángulo como spot en lugar de uno de forma arbitraria como en la sección anterior, al aplicar el parámetro turbulencia a una función base $sen(u)$, pueden distinguirse la aparición de triángulos de distintos tamaños, ubicados en posiciones aleatorias, que le dan forma a la textura seno original. Utilizando una elipse con su diámetro mayor en sentido vertical, la textura resultante presenta cierta tendencia a formar patrones en sentido vertical. En la Figura 4.1 se observan los spots con la textura que generan. Esta última observación resulta útil en el caso del granito puesto que existen algunos de ellos que presentan esa particularidad. Se muestra la imagen sintetizada del granito (como en la sección anterior) con este spot en la Figura 4.2.

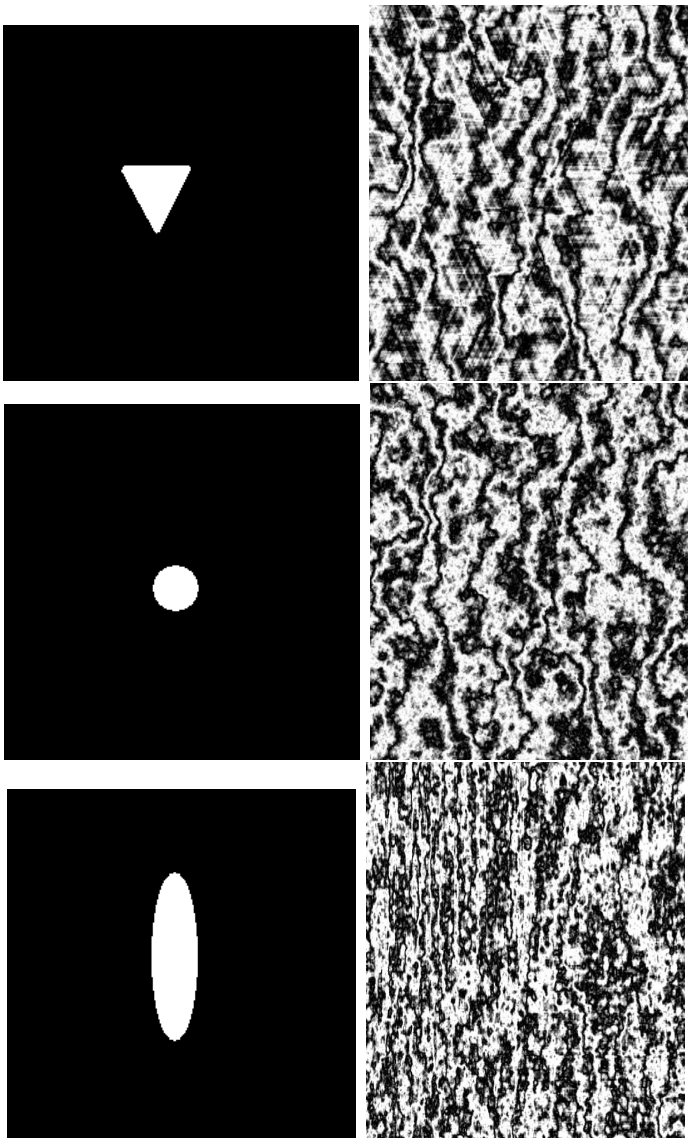


Figura 4.1: Texturas generadas a partir de distintos spots.

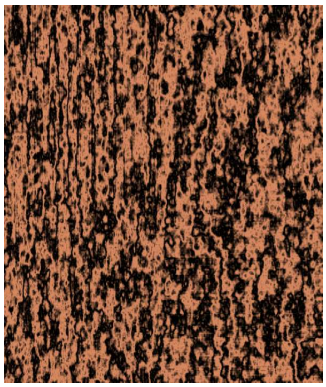


Figura 4.2: Granito utilizando una elipse en sentido vertical como spot.

La textura mencionada utilizada como función de desplazamiento de coordenadas, produce un corrimiento en sentido vertical de las mismas. Entonces, si se aplica la operación unaria *rotacion* sobre la textura de la base, sintetizando ondas senoidales con una orientación distinta a la que presenta la elipse en la textura utilizada como spot, el resultado no es el esperado. Este efecto indeseado puede observarse en la Figura 4.3. En dicha imagen, se aplica el parámetro turbulencia a una textura senoidal con un ángulo distinto a cero, notandose un desplazamiento de coordenadas que no produce una textura de granito. Esto provoca que la orientación del spot y de la textura de la base deba ser la misma. El presente modelo, como fue mostrado, permite orientar ambas texturas en el mismo sentido (con la operación mencionada). Cabe destacar que esto solamente ocurre en spots con una dirección predominante, así, no presenta inconvenientes un spot isotrópico como puede ser un círculo o uno con formas arbitrarias.

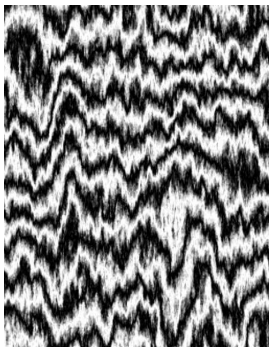


Figura 4.3: Efecto indeseado observado en el parámetro turbulencia aplicado a una textura senoidal rotada.

En el presente trabajo, se dejará libertad al autor de las texturas para que utilice distintos tipos de turbulencia, de tal forma que pueda capturar características como la presentada en los materiales que pretende sintetizar. Se procede a generalizar los elementos presentes en la base, lo cual permitirá la síntesis de otros materiales en el modelo.

## 4.2. Inclusión de elementos en la base

Los elementos de la base vistos hasta aquí estaban constituídos por funciones simples que capturaban las principales características morfológicas de los materiales que se pretendían representar. Para lograr mayor realismo, hasta aquí se utilizaba el parámetro "turbulencia", el cual introducía irregularidades en la textura final, buscando evitar la aparición de patrones que producen texturas excesivamente periódicas, siendo el efecto visible en mayor medida si se lo observa en una región amplia. Se procede a introducir en la base a distintas texturas resultantes del proceso de spot noise. Como se verá, esta inclusión otorga mayor flexibilidad para representar distintas texturas, además de permitir que las imágenes tengan una apariencia con un mayor grado de realismo. Esta última observación se basa en el hecho de que las texturas que están siendo añadidas a la base presentan un carácter estocástico que no poseían los elementos presentes en dicho conjunto, como ocurre con los materiales presentes en la realidad (esta afirmación podrá ser observada en las imágenes que se presentarán en las secciones a seguir). Además, dado que las texturas pueden ser generadas a partir de distintos spots, cuando se pretenda generar distintos materiales, se podrá introducir un spot específico que produzca una textura con determinadas características, permitiendo una mayor flexibilidad. Claro está, que la persona que

utilice un spot como parámetro, deberá tener cierta experiencia en el uso y funcionamiento de spot noise. Es por esto que el framework presenta ciertas texturas precargadas, que el usuario común utilizará habitualmente, y se deja abierta la posibilidad, en trabajos futuros, producir un spot variable para usuarios avanzados. El conjunto de texturas resultante del proceso de spot noise, se utiliza entonces tanto para generar turbulencia en las texturas, como para representar elementos en la base.

Existe sin embargo la limitación de que las texturas que son agregadas en este paso no pueden ser sometidas a operaciones unarias en el shader. Esto es así debido a que las texturas generadas en el proceso presentan una dimensión fija y no se extienden en todo el plano uv, por lo tanto no es posible la aplicación de rotación, desplazamiento, escala y turbulencia. Sin embargo, los resultados son satisfactorios y añaden valor a las texturas finales. Queda abierta la posibilidad en trabajos futuros de solucionar esta limitación.

Para incluir estos elementos en la base, debemos redefinir la misma como:

$$base = \{t_1, ..., t_n\},$$

$$t_i : A \rightarrow [0, 1], i \in [0, ..., n],$$
$$A \subseteq \mathbb{R}^2, [0, 1] \times [0, 1] \subseteq A.$$

Es decir, se abstrae el dominio de la función a un subconjunto del plano, siempre que ese subconjunto contenga al cerrado $[0, 1] \times [0, 1]$, permitiendo el muestreo dentro de ese subconjunto, necesario para las operaciones posteriores. De esta forma se contemplan los dos casos presentes en la base.

Si un elemento de la base posee como dominio a $A$, el cual es un subconjunto propio del plano cartesiano, entonces no podrán aplicarse operaciones unarias. Esto se traduce como :

$$A \subset \mathbb{R}^2 \Rightarrow t'(u, v) = t(u, v),$$

donde $t(u, v)$ es el elemento de la base y $t'(u, v)$ es el resultado de aplicarle operaciones unarias a ese elemento.

### 4.2.1. Casos de estudio con texturas de Spot Noise

Se procede a mostrar ejemplos de cómo los materiales de los casos de estudio, junto con los nuevos elementos de la base, poseen características estocásticas que resultan en un mayor realismo en las imágenes finales. En la Figura 4.6 se observa el resultado, se combinó en cada caso la función base que producía el material, por medio de la operación lerp, con una textura de spot noise. En la Figura 4.4 se ven los spots utilizados (el spot de la izquierda fue utilizado en las texturas de madera y granito. mientras que el de la derecha en la textura de mármol) y en la Figura 4.5 (la imagen de la izquierda fue utilizada en el material madera, mientras que la de la derecha en granito y mármol) se observan las texturas de spot noise utilizadas.

Figura 4.4: Spots utilizados para generar texturas de spot noise.

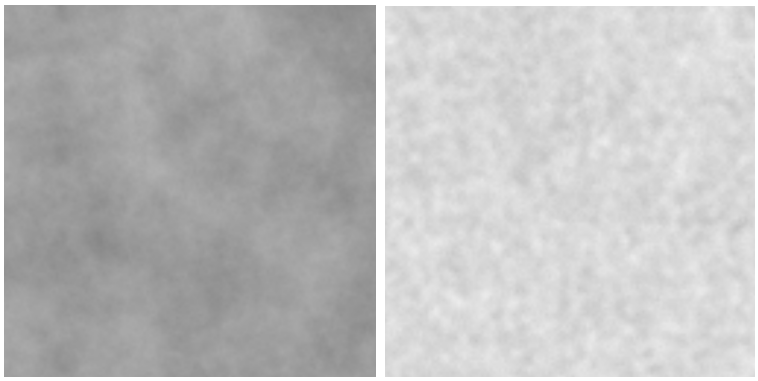


Figura 4.5: Texturas de spot noise utilizadas en la generalización.

Izquierda: sin spot noise. Derecha: con spot noise

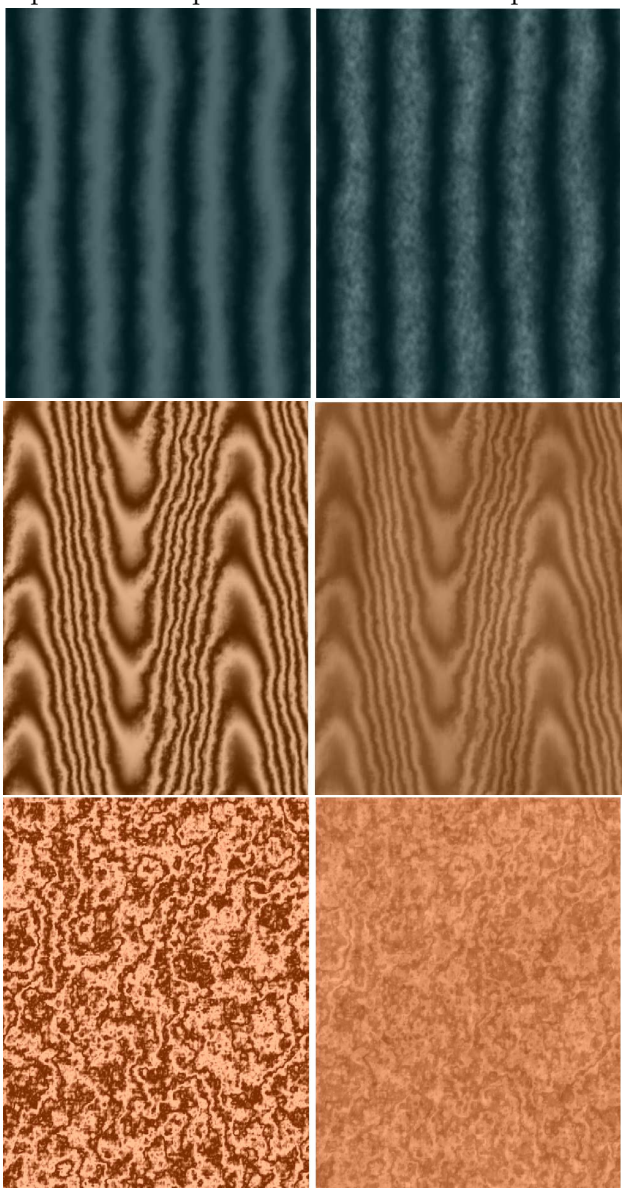


Figura 4.6: Materiales sin y con spot noise.

## 4.3. Otras funciones base

Es necesario que el framework posea mayor flexibilidad y capacidad a la hora de representar distintos materiales. Es comprensible, que no todos los materiales existentes pueden ser representados por el modelo; sin embargo, es posible una extensión sobre los materiales vistos. Una forma natural de alcanzar este objetivo, como se vio en la sección precedente, es introducir elementos en la base de tal forma de otorgar diversidad de representación en las texturas. Siguiendo esta idea, se procede a realizar una nueva generalización de los elementos de la base, incluyendo otras funciones matemáticas que resultan útiles. Las mismas están compuestas por patrones simples, que resultan útiles en el diseño de diversas formas, comunes a diferentes materiales. Por su generalidad, y simplicidad de generación, se eligieron las siguientes texturas como elementos de la base:

- $sin(\sqrt{u'^2 + v'^2})$,
- $sin(u') + sin(v')$,

$(u', v')$ son las coordenadas resultado de aplicar operaciones unarias a $(u, v)$. En la Figura 4.7 se observan ejemplos de la utilización de estas funciones como base. En la imagen a la izquierda, se observa la utilización de la primer función para producir agua, y en las otras dos imágenes se utiliza la segunda ecuación para producir telas.

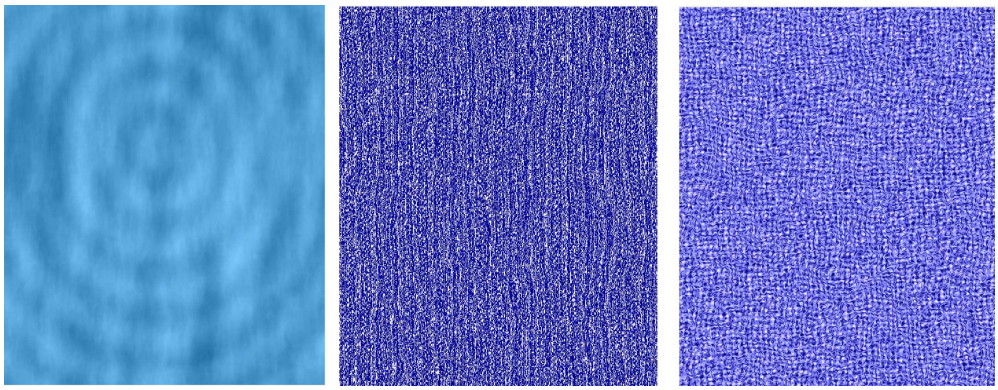


Figura 4.7: Materiales producidos utilizando otras funciones base.

La primer ecuación fue tomada de [8], la cual en dos dimensiones es una textura a círculos concéntricos equidistantes. Este patrón ayuda a producir texturas con determinadas formas circulares como telas de araña, cráteres, e inclusive madera (como se utiliza originalmente en el trabajo mencionado). La segunda provee a la base de un patrón regular a “cuadros”, resultando útil en el caso de, por ejemplo, modelar patrones en telas, o en determinadas formas regulares como rejas o alambrados.

La elección de las nuevas texturas base no deja de ser relativamente arbitraria. Las mismas sólo son un ejemplo de funciones que el modelo soporta. Entonces, si se avanza un paso más allá, es deseable que no se limite el modelo a determinadas funciones base. En la implementación del mismo, esta flexibilidad podría manifestarse en la capacidad de un usuario avanzado de producir sus propias funciones base, pudiendo añadirlas al código. Claro está que deberá ser capaz de implementar sus propios algoritmos de antialiasing. Sin embargo, el nivel de abstracción alcanzado en este razonamiento, no es el adecuado para usuarios del framework sin experiencia

en algoritmos de generación de texturas. Es por esto que se suplen estas funciones base, las cuales poseen capacidad para representar los materiales vistos hasta aquí y que serán útiles para la mayoría de los usuarios. De esta forma se abarca un amplio espectro de usuarios. Trabajos futuros podrán permitir dicha capacidad.

Hasta aquí, los pasos que fueron utilizados a la hora de sintetizar texturas fueron premeditados, es decir, el uso de determinadas texturas base y la aplicación de determinados parámetros tenían un fin específico (como producir un mármol determinado). Sin embargo nada impide combinar las texturas de manera aleatoria, y observar los resultados producidos, como se menciona en el capítulo cuatro de [2]. Esto resulta útil para generar texturas que podrían utilizarse en la decoración de escenas, como puede observarse en la Figura 4.8.

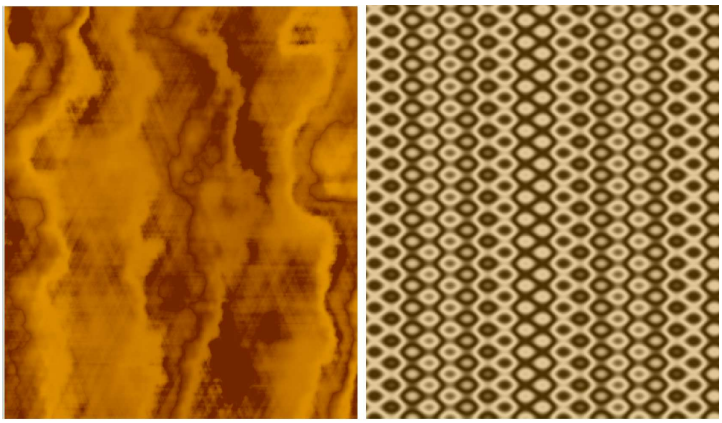


Figura 4.8: Texturas resultado de la combinación aleatoria de parámetros.

## 4.4. Otros materiales

El framework produce resultados satisfactorios en materiales con características aleatorias como agua, arena, césped, vegetación y otros que presentan patrones como telas, madera, o cortezas de palmeras; como lo muestran las siguientes imágenes. Sin embargo, no es posible representar características morfológicas específicas de los materiales, como puede ser la aparición de *nudos* en la madera, o diagramas de *Voronoi* (ver capítulo 4 de [2]). Esto es así debido a que esos materiales presentan características específicas como las mencionadas, las cuales no son compatibles con la generalidad del presente trabajo, el cual fue pensado para abarcar diversas texturas sin pretender representar justamente, características específicas de cada uno. Trabajos futuros pueden abordar las limitaciones mencionadas previamente, permitiendo sintetizar otros materiales con características específicas, extendiendo el modelo de tal forma de incluirlos en el mismo.

Es difícil determinar qué otros materiales no han sido sujeto del análisis en el modelo y sin embargo es perfectamente posible su síntesis en el framework, esto se debe en parte a la generalidad de las funciones base, que en teoría, pueden representar cualquier imagen, debido al uso de la función *seno* del modelo de Fourier. En la Figura 4.9 pueden verse ejemplos de otros materiales sintetizados en el framework. Se observan, de izquierda a derecha, nubes, cerámicos, vegetación y arena. Estos son sólo ejemplos de la capacidad del framework, por lo cual puede deducirse que el mismo puede representar diversos tipos de materiales.

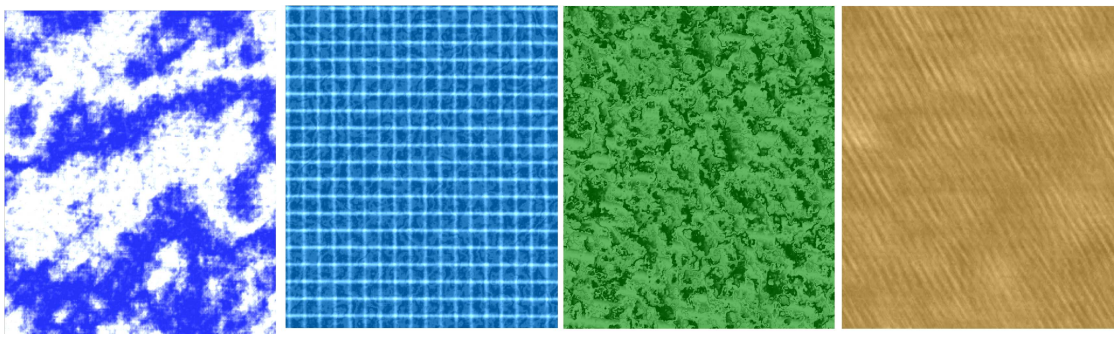


Figura 4.9: Otros materiales sintetizados.

## 4.5. Miscelánea

Para mostrar la flexibilidad del framework, se muestran otras tres imágenes obtenidas con relativa facilidad. Dos de ellas son de materiales estudiados y otra presenta la apariencia de la corteza de una palmera. En la Figura 4.10 se observan de izquierda a derecha, una imagen de mármol, la corteza de una palmera y una textura de granito. La textura de la derecha fue obtenida utilizando en gran medida el parámetro turbulencia, con tres texturas combinadas por medio del operador lerp.

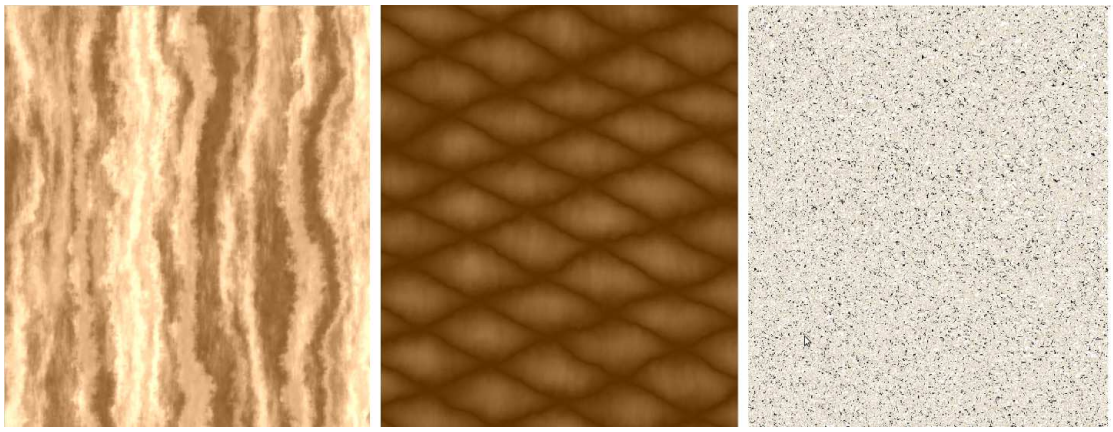


Figura 4.10: Otras texturas obtenidas.

# Capítulo 5

# Implementación

*"Digamos que con la ayuda de una pequeña . . . cajita . . . mágica"*
*Homero Simpson - Los Simpsons - "Homero va a la universidad"*

## 5.1. Introducción

Se eligió el entorno Qt para llevar a cabo la implementación. Su carácter multiplataforma fue la principal causa motivante de dicha elección. Por extensión natural se escogió un diseño orientado a objetos, el cual se integra de manera natural a este entorno de desarrollo. Se utilizó el lenguaje Cg para establecer la comunicación con la GPU y poder manipular las texturas en ella. El lenguaje de shader probó ser lo suficientemente flexible, permitiendo una implementación sencilla y eficiente.

## 5.2. Interfaz

La aplicación está constituída por un *ABM* [1] de texturas (ver el manual de usuario en el Apéndice), permitiendo ubicarlas en una escena 3D por medio de la especificación de cuatro vértices en el espacio de la textura que son mapeados a vértices en el espacio, la Figura 5.1 muestra los controles en la interfaz: cada fila representa un vértice del espacio donde se mapea un vértice de la textura.

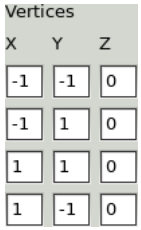



Figura 5.1: Parámetros de vértices en la interfaz.

Cada textura puede ser modificada permitiendo establecer los parámetros que la definen, mostrando un *preview* al usuario en tiempo real durante la modificación de los parámetros, lo cual representa una herramienta muy poderosa y flexible para el diseño de los materiales. Cada

[1] Alta - Baja - Modificación

material está compuesto de hasta tres texturas que se combinan por medio de las operaciones binarias mencionadas con anterioridad. En la Figura 5.2 se observan tres botones titulados "Textura 1" (2 y 3), que, de estar marcados establecen que esa textura no se utilizará en el material (en la figura las texturas 2 y 3 no serán utilizadas). El combo cuyos títulos comienzan con "Base" establece cuál base será utilizada para esa textura, y el combo restante, qué tipo de turbulencia será aplicado.

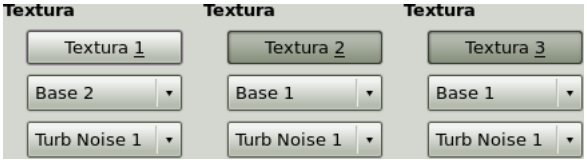



Figura 5.2: Parámetros de texturas en la interfaz.

Debajo de cada textura a utilizar se encuentran los parámetros propios de cada textura. El preview mencionado en realidad utiliza el mismo shader que es aplicado en la escena, por lo tanto las únicas variaciones que el usuario podrá observar son las transformaciones propias de aplicar texture mapping en OpenGL. Finalmente, es posible establecer un color *rgb* para cada material. Se cuenta con la capacidad de elegir un elemento diferente de la base para cada textura, estando conformada dicha base por las tres ecuaciones presentadas, además de una serie de texturas de spot noise precomputadas *offline*. Todas las operaciones tienen lugar en el espacio de la textura, independientemente de su ubicación final en la escena. En la Figura 5.3 pueden observarse capturas de pantalla del software mientras es sintetizada una textura (a la izquierda, la ubicación de la textura en la escena; a la derecha, la definición de los parámetros de la textura con el preview de la misma).

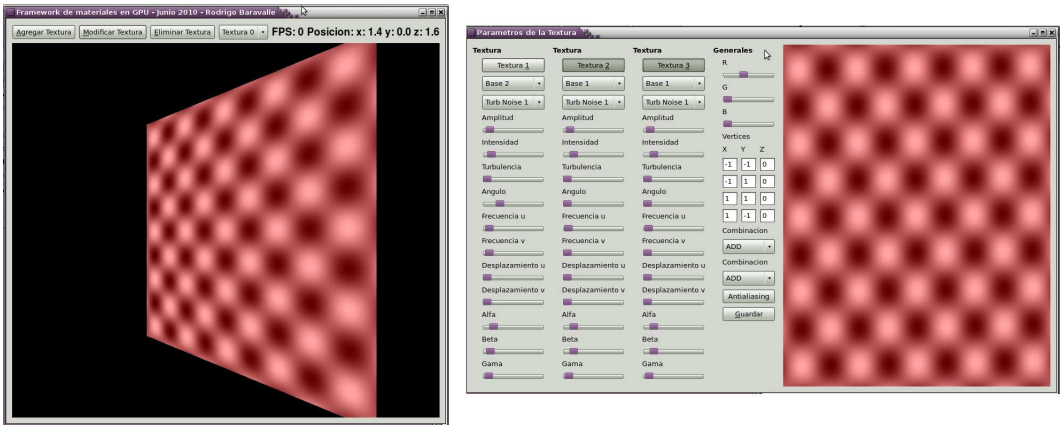



Figura 5.3: Capturas de pantalla del software desarrollado.

Con base en el concepto de simplicidad, se decidió abstraer al usuario de especificaciones propias de texture mapping, tales como, por ejemplo, si la textura presenta un borde, o si se utilizará mipmapping a la hora de construir las mismas, etc. Este tipo de extensiones en la interfaz resultarán muy sencillas de llevar a cabo, por lo tanto se definió la misma de tal forma de abarcar el mayor espectro posible de usuarios.

## 5.3. Funcionamiento

Básicamente, el flujo de datos que siguen los parámetros en la aplicación es el siguiente: el usuario especifica los mismos por medio de la interfaz, contando con un *preview* del material final. Los datos se envían desde la aplicación (Qt) hacia OpenGL, quien a su vez se encarga del paso hacia la GPU de los datos. Para esto, OpenGL se comunica con la GPU utilizando el runtime de Cg, especificando un programa de vértices y de fragmentos para el proceso, el cual define cómo manipular esos datos para lograr el resultado final, estableciendo cada uno de los parámetros que son definidos en los shaders.

Para mostrar la utilización del programa, se detalla cómo sintetizar un textura de madera: la función base que la define es $sin(\alpha u + \beta sin(u) + \gamma sin(v))$, donde $\alpha, \beta, \gamma$ son parámetros controlables por la interfaz del usuario. En la Figura 5.4 se observa cómo varía la textura a medida que se ejecuta cada paso. Los pasos que se siguieron son los siguientes:

- se utiliza el parámetro $\alpha$ para establecer la forma sinusoidal inicial de la estructura base de la textura (imagen de la izquierda de la Figura 5.4),
- se setea el parámetro $\beta$ a cero, produciendo funciones seno repetidas a lo largo de la coordenada $u$, con las formas que podemos observar en la segunda y tercer imagen (desde la izquierda) de la Figura 5.4.
- se establece el parámetro *turbulencia* con un valor cercano, pero distinto a cero, por ejemplo 0.1, y el color deseado *RGB* para que la textura de madera resulte como la imagen de la derecha de la figura mencionada.

Todos los pasos mencionados son controlados en la aplicación por medio del uso de sliders que permiten observar el proceso en tiempo real.

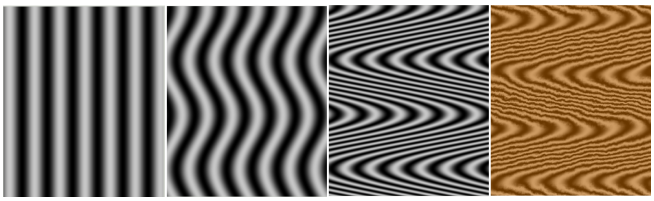


Figura 5.4: Pasos para la obtención de una textura de madera

## 5.4. Shaders

El shader con relevancia en la implementación es el *fragment shader*. El *vertex shader* representa un simple paso de datos en el pipeline de la GPU, aplicando una matriz *modelviewproj* [2] a los vértices que recibe como entrada, transformándolos en coordenadas en pantalla. Como fue mencionado, el fragment shader recibe los parámetros por medio del runtime de Cg, los cuales representan los datos del material que está siendo renderizado en ese momento. La implementación de las operaciones en el shader resulta muy sencilla. Por citar un ejemplo, la operación binaria de suma, es una suma en el fragment shader entre dos valores obtenidos de dos funciones base. Luego, este shader produce un valor entre 0 y 1 que representa el valor que asume el mapa de alturas (*heightmap* en inglés) en esas coordenadas. Este valor es utilizado para producir el

[2] Model - View - Projection

color final que el usuario eligió para el material. Finalmente, este shader produce un color en formato rgb, el cual, si sortea todos los tests mencionados en la teoría, se convertirá en un píxel en pantalla. Debido a que el resultado final, antes de la aplicación del color es un mapa de alturas, el framework puede ser extendido para representar bump mapping, normal mapping, displacement mapping, etc.

A continuación se muestra el código del vertex shader completo. En el mismo, la posición del vértice es transformada por la matriz mencionada para ubicarlo en la escena; por otro lado, las coordenadas de las texturas son pasadas al siguiente eslabón del pipeline sin modificaciones, debido a que las transformaciones tendrán lugar en el fragment shader.

Vertex Shader

```
void main(
 float4 position : POSITION,
 float2 texCoord0 : TEXCOORD0,

 uniform float4x4 modelViewProj,
 out float4 oPosition : POSITION,
 out float2 oCoord0  : TEXCOORD0)
{
    oPosition = mul(modelViewProj, position);
    oCoord0 = texCoord0;
}
```

En los fragmentos de código que siguen se presenta un resumen del fragment shader utilizado, resaltando sus características más importantes. El primer fragmento presenta la función *desplazar*, la cual dada una coordenada en el plano, desplaza la misma de acuerdo a los parámetros *dx* y *dy*.

```
float2 desplazar(float2 texCoord, float dx, float dy)
{
    return texCoord + float2(dx,dy);
}
```

La siguiente función permite combinar dos texturas de la base, a traves de un entero $i$ que representa una forma de combinación (ADD, SUB, MUL, etc.).

```
// combinacion de las texturas
float combina (float t1, float t2, int i)
{
  if (i == 0 )
      return t1+t2;
  else { if (i==1) return t1-t2;
  else {if(i==2) return lerp(t1,t2,0.5); else {
            if(i==3) return sobre(t1,t2);
            else return t1*t2;} }}

}
...
```

La función fbase devuelve el valor que toma la función base especificada en el parámetro $b$, en las coordenadas $(xf, yf)$. Puede apreciarse la implementacion trivial de la ecuacion utilizada para producir los materiales de los casos de estudio. Alpha, beta y gama son parámetros que toma la función, los cuales representan los parámetros de la ecuación presentada en los casos de estudio.

fbase

```
float fbase(int b, float xf, float yf, float alpha, float beta, float gama) {
    if(b == 1)
        return (sin(xf)+sin(yf)+2)/4;
    else {
        if(b==2) return (sin(alpha*xf + beta*sin(xf) + gama*sin(yf))+1)/2;
        else  {
            return (sin(sqrt(xf*xf + yf*yf))+1)/2;
        }
    }
}
```

Se presenta a continuación la función principal del shader. La función *main* es el punto de entrada al programa. Se puede observar la utilización de parámetros *uniform* (que representan parámetros que llegan desde la aplicación) que contienen valores de elementos en la interfaz de usuario, como color o intensidad de cada textura. También se observan texturas de spot noise precargadas, las cuales aparecen en el listado como *uniform sampler2D*. El shader retorna un color como valor, el cual es un color RGB.

```
#define CANT 3

float4 main(
        float2 coord0  : TEXCOORD0,
        uniform float base[CANT],
        uniform float tipo_turb[CANT],
        uniform float color[3],
        uniform sampler2D textura0,
        ...       ): COLOR
```

El cuerpo de la función main realiza el cómputo de la textura. En primera instancia se consulta el valor de las texturas de spot noise en las coordenadas actuales utilizando la función *tex2D*.

```
{
    float noise = tex2D(textura0, coord0);
    float noise2 = tex2D(textura1, coord0);
    float noise3 = tex2D(textura2, coord0);
    ...
```

El parámetro *tipo de turbulencia*, representado con el arreglo *tipo_turb*, permite seleccionar para cada una de las texturas que componen el material a sintetizar, cuál textura de spot noise será utilizada como turbulencia. Luego se multiplica la cantidad de turbulencia especificada en la interfaz de usuario por ese valor.

```
    if(tipo_turb[0] == 0)
            tturb1 = noise;
    else { if(tipo_turb[0] == 1) tturb1 = noise2;
            else tturb1 = noise3; }
    ...
    float turbulencia0 = turb[0] * tturb1;
    ...
```

A continuación se aplican operaciones unarias a las coordenadas actuales, si el elemento de la base seleccionado debiera ser sometido a las mismas (es decir, si el elemento de la base no

fuera una textura de spot noise). Luego se calcula el valor de la textura en las coordenadas transformadas (llamada a la función *fbase*).

```
    float b1,b2,b3;
    if(base[0] < CANT) {
        float2 c0 = rotar(coord0, angulos[0]);
        c0 = desplazar(c0, desplazamiento_u[0],desplazamiento_v[0]);
        c0 = c0 + float2(turbulencia0,turbulencia0);
        float xf = escala_u[0]*c0.x;
        float yf = escala_v[0]*c0.y;
        b1 = fbase(base[0],xf, yf, alpha[0], beta[0], gama[0]);
    }
    else {
        if(base[0] == CANT)
            b1 = noise;
        else { if(base[0] == CANT+1) b1 = noise2;
        else b1 = noise3; }
    }
    ...
```

Luego son aplicadas operaciones especiales (intensidad y amplitud) al valor obtenido en el paso previo, para cada textura. También se tiene en cuenta si la textura será utilizada o no. El último paso del proceso está representado por la aplicación de color. Este valor es el devuelto por el shader.

```
    float t1 = intensidad[0]*pow(b1,amplitud[0]);
    b1 = usa[0] == 1 ? t1 : 0;
    ...
    float altura; // valor en escala de grises
    ...
    col.xyz = altura+float3(color[0],color[1],color[2]);
    return col;
}
```

## 5.5. Cálculo offline de texturas de Spot Noise

Las texturas presentes en la base, resultantes del proceso de spot noise, fueron obtenidas gracias a un programa muy simple que se implementó en CUDA, siguiendo la teoría del método. Se mostró que una textura de spot noise puede obtenerse por medio de una convolución de la imagen del spot con ruido blanco. Por otro lado, el teorema de convolución establece que la transformada de la convolución de dos funciones es la multiplicación de las transformadas de esas funciones. CUDA cuenta con la biblioteca cufft la cual implementa Transformadas de Fourier de una manera eficiente utilizando la alta paralelización con que cuentan las GPU's. Consiguientemente, podemos utilizar esta biblioteca para realizar la transformada del spot, multiplicarla por ruido blanco y realizar la transformada inversa (utilizando la misma biblioteca), obteniendo la imagen deseada. Es de esta manera como se obtuvieron las texturas que se incluyeron en el modelo. En la sección "trabajos futuros" se mencionará una idea relacionada con este cálculo. Se presenta a modo ilustrativo el segmento del código que produce las tranformaciones mencionadas en CUDA.

En primer lugar, la función *ComplexMul* toma dos números complejos y los multiplica:

La función *ComplexPointwiseMulAndScale* toma dos arreglos a y b, cuyos elementos son números complejos y realiza la multiplicación elemento a elemento entre ellos, por medio de *ComplexMul*.

ComplexMul

```
static __device__ __host__ inline cufftComplex ComplexMul(
    cufftComplex a, cufftComplex b)
{
    cufftComplex c;
    c.x = (a.x * b.x - a.y * b.y)
            * (1.0f / (float)(tamanio));
    c.y = (a.x * b.y + a.y * b.x)
            * (1.0f / (float)(tamanio));
    return c;
}
```

ComplexPointwiseMulAndScale

```
static __global__ void ComplexPointwiseMulAndScale(
        cufftComplex* a, cufftComplex* b, int size)
{
    const int numThreads = blockDim.x * gridDim.x;
    const int threadID = blockIdx.x * blockDim.x +
                                      threadIdx.x;

    for (int i = threadID; i < size; i += numThreads)
        a[i] = ComplexMul(a[i], b[i]);
}
```

La entrada al programa es la función *spotNoise*. En primera instancia, la función *cufftPlan2d* inicializa el proceso para calcular una Transformada de Fourier. Esta función toma como parámetro las dimensiones de la imagen y debe llamarse obligatoriamente antes de realizar el cálculo. Luego, se copia a la memoria de la GPU la imagen (la cual en este caso está en el array *texturaCuda*). La variable *idata* representa la imagen en la placa gráfica.

```
void spotNoise(...)
{
    ...
    cufftResult res = cufftPlan2d(&plan, NX, NY, CUFFT_R2C);
    // copia : imagen -> device
    cudaMemcpy(idata, texturaCuda, sizeof(cufftReal)*tamanio,
                                cudaMemcpyHostToDevice);
```

Posteriormente, tiene lugar el cálculo de la transformada, por medio de la función *cufftExecR2C* (R2C significa *Real to Complex*, ya que la imagen de entrada contiene valores reales y su transformada es una función compleja). El mismo proceso es utilizado para realizar el cálculo de la transformada de ruido blanco.

```
    cufftExecR2C( plan, idata, odata );
    cufftPlan2d(&planNoise, NX, NY, CUFFT_R2C);
    // copia : ruido -> device
    cudaMemcpy(idataNoise, noiseCuda, sizeof(cufftReal)*tamanio,
                                cudaMemcpyHostToDevice);
    cufftExecR2C( planNoise, idataNoise, odataNoise );
```

Se utiliza entonces *ComplexPointwiseMulAndScale* (blocks por grid: 32, threads por block: 256), para multiplicar ambas transformadas, guardando el resultado en *odata*. Para obtener la

imagen deseada, sólo resta realizar la transformada inversa (C2R, *Complex to Real*) de *odata* y copiar el resultado a la CPU (en este caso, a texturaCuda).

```
    ComplexPointwiseMulAndScale<<<32, 256>>>(odata, odataNoise, tamanio);
    // en odata, la transformada que se necesita
    cufftHandle planInv;
    cufftPlan2d(&planInv, NX, NY, CUFFT_C2R);
    // tranformada inversa de fourier
    cufftExecC2R( planInv, odata, idata );
    cudaMemcpy(texturaCuda, idata, sizeof(cufftReal)*tamanio,
                                cudaMemcpyDeviceToHost);
    ...
}
```

## 5.6. Diagrama de Clases

En la Figura 5.5 se pueden observar las clases que componen la implementación en Qt y las relaciones existentes entre ellas. También se realiza una breve descripción de las mismas.

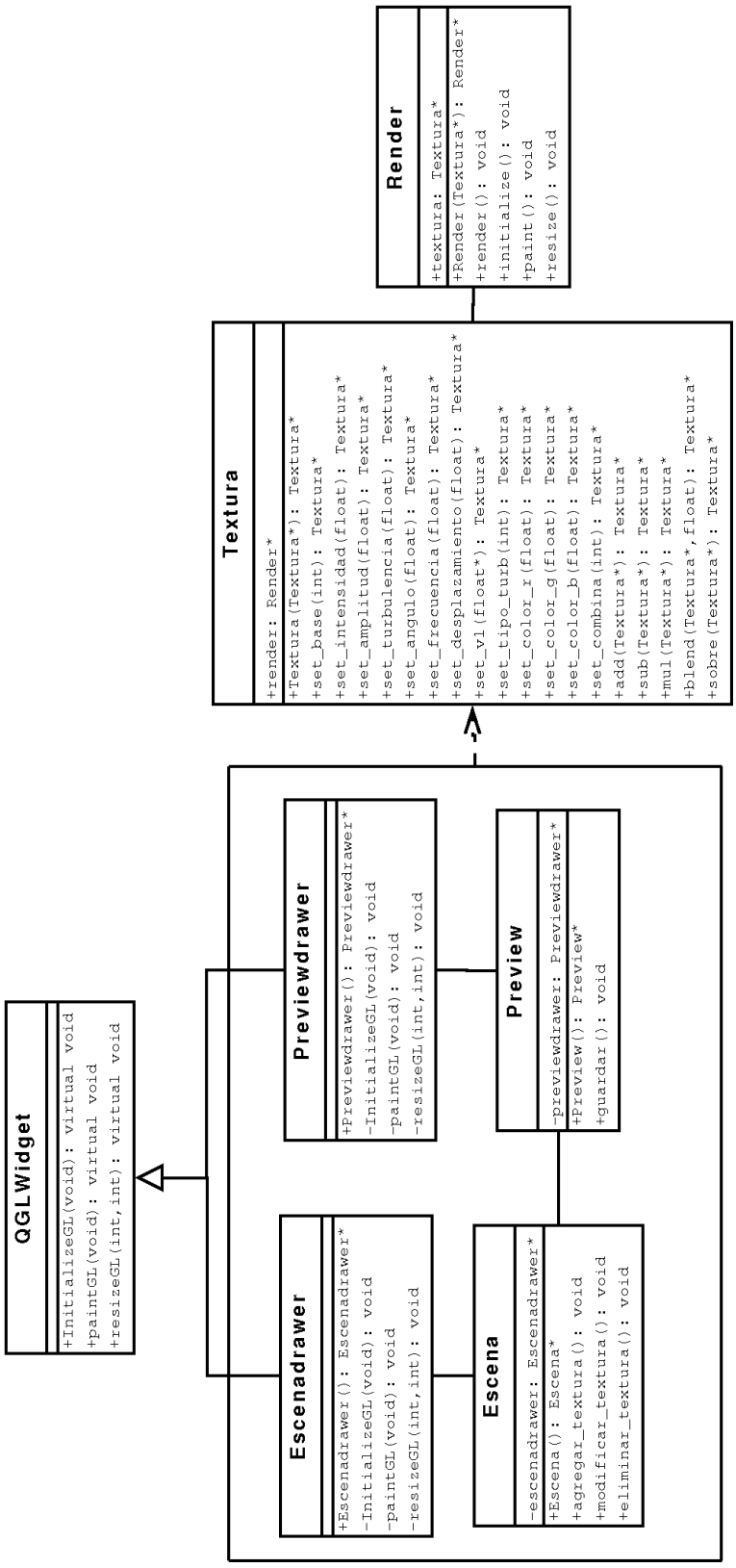



Figura 5.5: Diagrama de Clases

- Textura: constituye la clase central de la implementación. La misma encapsula la representación de un material. Su interfaz permite setear todos los atributos que fueron expuestos a lo largo del presente trabajo. Todos los demás componentes de la implementación utilizan instancias de esta clase para poder llevar a cabo su tarea.
- Render: encapsula la implementación. Si se desease por ejemplo utilizar CUDA en lugar de Cg, se debería modificar sólo esta clase. En estas dos primeras clases está basado el framework. Las clases que siguen son propias de la implementación en Qt, y de hecho, con sólo estas dos clases se pueden ver las texturas generadas en pantalla.
- Preview: esta clase toma como parámetro un objeto textura, permitiendo la manipulación de los parámetros de la misma. Posee un objeto previewdrawer que permite la comunicación con OpenGL con el fin de renderizar la textura, presentando un preview al usuario de la misma.
- Previewdrawer: esta clase *hereda* de la clase virtual de la biblioteca de Qt *QGLWidget*, reimplementando los métodos *initializeGL()*, *paintGL()* y *resizeGL()*, ofreciendo una preview de la textura final, permitiendo conocer de antemano la apariencia final del material al usuario.
- Escena: clase encargada de manejar el ABM de texturas. Contiene listas de texturas que representan los materiales presentes en la sesión[3]. Se comunica con el módulo preview para proveerle del material a ser previsualizado (una instancia de la clase Textura).
- Escenadrawer: al igual que Previewdrawer esta clase hereda de QGLWidget y reimplementa los mismos métodos. La diferencia radica en que esta clase renderiza en una escena 3D todas las texturas presentes en esa sesión, dichas texturas son el resultado del ABM de texturas.

Como puede observarse, el número de clases es reducido. Esto resulta fundamental en una implementación en la cual la performance es un punto crítico (debido a que las texturas son modeladas y renderizadas en tiempo real). Por otro lado, el diseño resulta suficientemente claro y modificable. Si quisiera utilizarse otro entorno de desarrollo, diferente al de Qt, la tarea no presentaría mayores dificultades, debido a la abstracción del material en la clase Textura, y a su sencilla integración con OpenGL, contando con la presencia del módulo Render, el cual libera a la aplicación de la comunicación con pantalla. El núcleo del framework está representado por la clase Textura y el fragment shader que la dibuja. Resulta sencillo utilizar dicha clase y realizar combinaciones entre instancias de la misma, para luego mostrarlas en pantalla, o integrarlas en otro proceso que se necesitase. El siguiente código muestra la sencilla utilización de dicha clase, instanciando dos texturas y sumandolas, para obtener un material y mostrarlo por pantalla:

```
int main(
    Textura* t1 = new Textura;
    Textura* t2 = new Textura;
    t1->set_base(TEX2)->set_intensidad(0.5)
      ->set_tipo_turb(1)->set_turbulencia(0.9);
    t1->add(t2);
    t1->render();
}
```

[3]Se considera *sesión* a la corrida actual del programa

## 5.7. Antialiasing

### 5.7.1. Introducción

Un tópico con importancia a la hora de diseñar texturas procedurales, está representado por la capacidad de éstas de producir aliasing en el resultado final. Como se ha explicado, el aliasing produce "artefactos" visibles en las texturas, resultante de un incorrecto muestreo en el modelo subyacente. En las ecuaciones presentes en la base puede observarse claramente, en determinados casos, la producción de estos indeseables artefactos si el problema no es tratado. En la Figura 5.6 puede observarse un ejemplo de una textura sin tratamiento de aliasing, (a la derecha de la imagen, aparecen círculos concéntricos). Para poder resolver el mismo, es útil conocer previamente en qué situaciones ocurre. De esta forma la solución puede incluir un comportamiento diferente basado en cada situación particular. Se procede entonces a identificarlas.

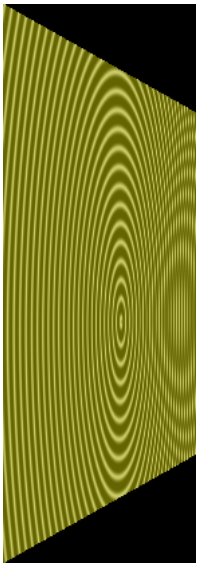


Figura 5.6: Aliasing en el modelo.

### 5.7.2. Discusión

En primer lugar, las texturas de spot noise presentes en la base poseen un carácter estocástico que hace que resulten poco propensas a sufrir tales fenómenos. De esta manera, se debe analizar sólo el caso de las texturas definidas por medio de una ecuación.

Las texturas mencionadas comenzarán a producir aliasing en el caso en que el intervalo de muestreo no resulte el adecuado, sin embargo, no resulta trivial determinar la frecuencia de Nyquist para las ecuaciones con las que se cuenta. El método ideal es el mencionado como *clamping* en la teoría. Sin embargo, debido a que la forma de generación de los materiales no utiliza directamente el método de *spectral painting*, no podemos optar por eliminar las frecuencias más altas, implementando de esa forma un filtro previo a la renderización.

Lamentablemente no existe aún una solución genérica para el problema del aliasing en texturas procedurales. Por lo tanto, lo recomendable es utilizar un método que se ajuste en cada caso, buscando atenuar lo mejor posible los artefactos producidos por el problema. En el caso del framework, sólo debemos tener en cuenta el intervalo de muestreo, y con base en el mismo, determinar una solución que intente resolver el problema sólo en los casos que sea estrictamente

necesario. Para resolver el problema entonces, debemos contar con una medida de la variación del muestreo en el shader. En la siguiente sección se discute la forma de obtener tal medida.

### 5.7.3. Determinando el tamaño del filtro

En Cg se cuenta con las primitivas ddx(a) y ddy(a), las cuales dada una variable a, devuelven un valor de cambio píxel a píxel (en las direcciones u y v respectivamente) respecto de esa variable. Con estos valores, podemos calcular fácilmente el intervalo de muestreo que está teniendo lugar en cada momento. Lamentablemente, dichas funciones no están disponibles en todos los perfiles, por lo cual se debe recurrir a otro método menos habitual, para que se tenga soporte en los perfiles más básicos.

Determinación del tamaño del filtro

```
glBindTexture(GL_TEXTURE_2D, texObjFilterMap);

#define LEVELS 9
GLubyte *filterMap = new GLubyte[256 * 256];
for (int level = 0; level < LEVELS; ++level) {
  int res = (1 << (LEVELS-1-level));
  for (int i = 0; i < 256*256 ; ++i)
    filterMap[i] = 16*level;
    glTexImage2D(GL_TEXTURE_2D, level, 1, res, res, 0,
                 GL_RED, GL_UNSIGNED_BYTE, filterMap);
}
```

Utilizando el método propuesto en [11], podemos hacer uso del esquema de *mipmapping* que ofrecen bibliotecas como OpenGL, para determinar el intervalo de muestreo basado en los momentos en que se aplica uno u otro mapa de dicha jerarquía. De esta manera, se define una textura de resolución $N \times N$ (en este caso, $256 \times 256$) píxeles. Si en el primer nivel, 0, se introduce en todos los pixeles que conforman el mismo el valor 0, en el segundo nivel, 1 ($128 \times 128$), se introduce el valor 1 de igual forma, hasta llegar al último nivel, 8, con solamente 1 píxel; tenemos un método que, cuando se consulta el valor de la textura en determinado píxel, devuelve el nivel de mipmapping utilizado por el hardware en esas coordenadas. Se utilizará esta idea, pero en su lugar se multiplicará por 16 el valor introducido en cada nivel para aprovechar al máximo la precisión de 8 bits ofrecida por las texturas. De esta forma, finalmente en cada nivel se está guardando el logaritmo en base 2 de la cantidad de texels presentes en cada nivel (por ejemplo si el nivel tiene 128 texels, se está introduciendo el valor $log_2 128 = 7$, multiplicado por 16).

En Cg, un parámetro sampler2D estará ligado a dicha textura. El valor del parámetro en las coordenadas actuales estará en el cerrado $[0, 1]$ . Multiplicando por 255 se obtiene el valor almacenado en la textura, el cual representa el logaritmo en base 2 de la cantidad de texels en el nivel. Solo resta obtener el espacio entre texels (el tamaño del filtro). Este resulta 1 entre la cantidad de texels: $1/2^{log_2(numTexels)}$. Equivalentemente, $2^{-log_2(numTexels)}$.

Determinación del filtro en Cg

```
float filterwidth(float2 uv)
{
  float log2width = (255./16.) * tex2D(filterMap, uv).x;
  return exp2(-log2width);
}
```

Este método resulta muy útil para eliminar la dependencia del shader respecto del perfil. Sin embargo, en un futuro cercano, debido a la rápida evolución de las placas gráficas, no se contará con este tipo de limitaciones. La performance del método es similar a la presentada por las funciones ddx y ddy. El único problema del método radica en el espacio en memoria que ocupa la textura. Pero, tampoco ésto presenta una problema de importancia, debido a la causa que se mencionó anteriormente.

### 5.7.4. Resolución

Una forma de intentar solucionar el problema es realizar supersampling en el fragment shader mismo. De esta manera, cada texel de la textura podría computarse como un promedio de los texels circundantes. La desventaja de este método es que no elimina el aliasing, ya que el mismo ocurre, pero en frecuencias más altas. Se considera costoso el cálculo de varios valores de texels, para luego realizar un promedio entre ellos, sin poder hacer desaparecer el problema. Por esto la solución no se encuentra satisfactoria.

Es posible razonar con toda validez que el problema del aliasing ocurre por la presencia de frecuencias altas en las texturas finales. Por lo tanto, podría intentarse eliminar las mismas durante el cálculo de las coordenadas luego de las operaciones unarias, entre ellas, el escalamiento. Pero este método resulta muy invasivo y generalmente produce un cambio total de la morfología de la textura. Por otro lado, no resulta trivial determinar cómo eliminar esas frecuencias más altas, sobre todo si se piensa que las ecuaciones base que definen cada imagen son muy diferentes entre sí. Por lo tanto este intento resulta impracticable. Sin embargo, permite observar que debido a que el usuario cuenta con controles en la interfaz para establecer distintos parámetros que determinan las frecuencias en cada textura, es posible que el mismo atenúe los efectos del aliasing disminuyendo valores excesivos en esos parámetros. De esta manera, el usuario puede reducir en parte los artefactos, y descansar en la idea de que el framework hará su trabajo en los casos que sea necesario.

Si se realizara el muestreo del modelo de manera estocástica, los artefactos producidos serían reemplazados por ruido, debido a que las ondas llamadas "alias" no podrían aparecer justamente por la forma en que se realiza el mismo. Como el modelo cuenta ya con funciones de ruido (las texturas existentes de spot noise), no resulta difícil cumplir con tal propósito. La operación de turbulencia presentada en capítulos previos, desplaza coordenadas previamente a su cálculo en las ecuaciones en las funciones de la base, lo cual es exactamente lo que se necesita en este punto. Para realizar la implementación en el shader, sólo se tiene que aplicar turbulencia en aquellos casos que sea necesario. Es buena idea aplicar la operación en base al valor que tiene el tamaño del filtro, utilizando el parámetro en mayor medida mientras dicho valor disminuye, ya que el aliasing se presentará con mayor notoriedad en los casos que el intervalo de muestreo sea menor. Si bien esto no resuelve el problema, presenta una solución eficiente y satisfactoria para la mayoría de los materiales que se pretenden sintetizar. Trabajos futuros podrán abordar el tema con mayor profundidad.

Luego de todas estas consideraciones, se concluye que si bien el problema no puede ser resuelto de manera genérica, se pueden reducir los efectos producidos si se utiliza el intervalo de muestreo para establecer en qué momento aplicar la función de ruido, buscando reemplazar el aliasing presente por ruido lo cual resulta más natural en los materiales que se pretenden sintetizar. El código muestra la sencilla implementación en Cg.

La variable start_alias establece el intervalo máximo de muestreo sin que se aplique antialiasing. Si la textura ya presentase turbulencia definida, no será necesario la utilización de este parámetro. La función *max* se utiliza debido a que es deseable siempre que sea posible un mayor

Tratamiento de aliasing

```
uniform sampler2D filtermap;
...
float fil = filterwidth(filtermap,coord0);
float start_alias = 0.5;
if(fil < start_alias) {
    float noise_level = (start_alias - fil);
    turbulencia0 = max(turb[0],noise_level) * tturb1;
    turbulencia1 = max(turb[1],noise_level) * tturb2;
    turbulencia2 = max(turb[2],noise_level) * tturb3;
}
```

nivel de turbulencia, para atenuar los efectos que presenta este problema. El valor 0.5 fue escogido luego de la experimentación con distintos materiales. Puede observarse en la Figura 5.7 un ejemplo de una textura con distintos start_alias. Las imágenes fueron obtenidas por medio del software que implementa el framework. La figura muestra una textura vista en perspectiva. En la imagen de la izquierda, $start\_alias = 0{,}3$, puede observarse que se produce alias en la parte más lejana de la textura. En la imagen central, $start\_alias = 0{,}5$, lo cual transforma alias en ruido de forma correcta. En la imagen de la derecha, $start\_alias = 0{,}8$. Se puede observar que el tratamiento de aliasing en esta última imagen ocurre en regiones innecesarias en la textura.

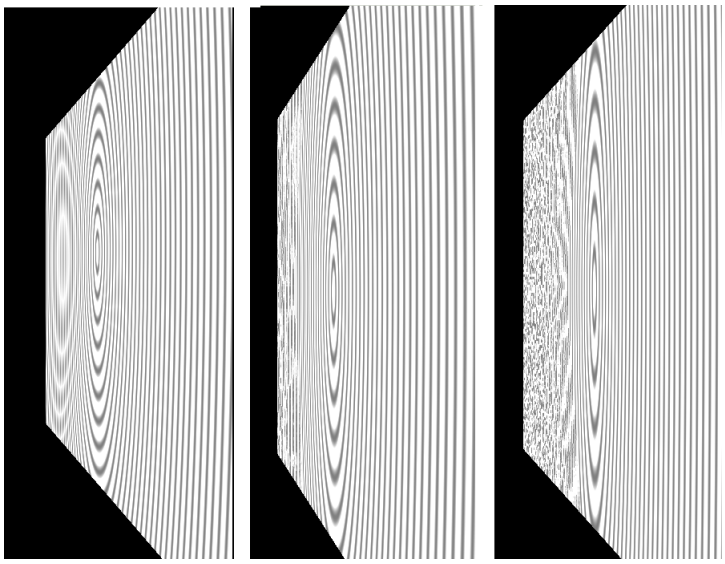


Figura 5.7: Distintos valores de la variable start_alias.

Además del antialiasing que se realizará automáticamente (si es solicitado, pues existe un parámetro para desactivarlo), el usuario cuenta con algunas herramientas para reducir los efectos indeseados. Si el material que se está sintetizando resultase visiblemente propenso a producir los mencionados artefactos, siempre se puede reducir su frecuencia en u y en v, reduciendo drásticamente los mismos, teniendo en cuenta que el diseñador del material tiene en la interfaz la posibilidad de hacerlo. Otra forma es reducir la intensidad de la textura final, debido a que justamente los cambios bruscos en la intensidad de una imagen (contraste) son los que producen altas frecuencias en su espectro. También se cuenta con la posibilidad de disminuir el parámetro

amplitud, el cual también es propenso a producir cierto contraste en la imagen final.

En la Figura 5.8 se comparan dos texturas, una con aliasing y la otra con el algoritmo descripto aplicado (en la figura, a la izquierda la textura sin la aplicación del algoritmo utilizado, a la derecha con antialiasing aplicado; las imágenes fueron tomadadas en el software implementado con cierta distancia entre el observador y la textura, magnificando los efectos del aliasing). La textura es la segunda listada en el apartado "Otras funciones base" del capítulo anterior. Puede observarse que el resultado es satisfactorio, a pesar de la aparición de ruido en la imagen, el cual es preferible al aliasing presente en la textura original.

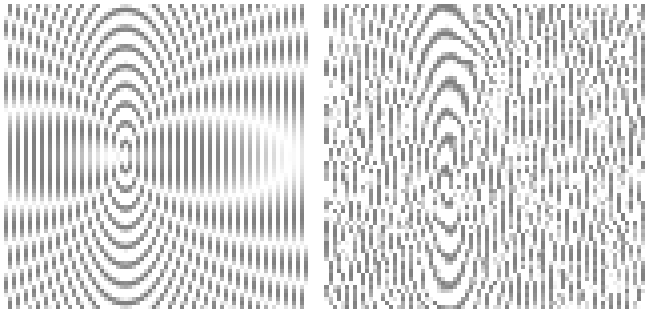


Figura 5.8: Antialiasing en el modelo.

## 5.8. Rendimiento

El presente trabajo ha sido diseñado con el objetivo de que los resultados sean aplicables en tiempo real. Con base en dicho objetivo, se ha hecho hincapié en la performance de la aplicación, en oposición a complejidad del diseño. Por este motivo, se han limitado la cantidad de accesos a texturas y los cómputos excesivos en el shader. No sorprende entonces, que el programa sea capaz de correr incluso en los perfiles básicos que ofrece el runtime de Cg.

Para observar en la práctica dichas aseveraciones, se añadió un contador de cuadros por segundo (*fps* en inglés) para observar si el rendimiento era acorde a las placas gráficas disponibles hoy en el mercado. Se utilizó la clase QTimer para llevar a cabo dicha implementación, dentro del módulo Escena. Se utilizaron resoluciones de hasta 1024x1024 texels en las texturas a ser sintetizadas.

Se utilizó una CPU Athlon 5400+ x64 Dual Core con 2gb de RAM y una GPU Nvidia GeForce n9500GT de 512 MB de RAM, lo cual representa una configuración standard para el año de lanzamiento del presente trabajo. Los valores observados son satisfactorios, y permiten interactuar en tiempo real en la escena mientras se realiza el modelado de los materiales. También se probó su correcto funcionamiento en placas nvidia GeForce de la línea 7000 con idénticos resultados. En todos los casos, los fps fueron mayores o iguales a 30, tomando este valor sólo en el caso de una textura que ocupe toda la pantalla. En general, los valores observados se ubican por encima de los 100 fps.

# Capítulo 6

# Conclusiones y Trabajos Futuros

*'Pero pronostico que en 100 años las computadoras serán dos veces más potentes, diezmil veces más grandes y tan costosas que sólo los cinco reyes más ricos de Europa las tendrán"*
*Profesor Frink - Los Simpsons - "Y dónde está el inmigrante"*

## 6.1. Conclusiones

El framework introducido presenta la ventaja de contar con un diseño e implementación sencillos, lo cual hace que la obtención de los materiales deseados sea fácil e intuitiva desde el punto de vista del usuario. Además, la calidad de las texturas resulta satisfactoria teniendo en cuenta lo escaso de los parámetros que el usuario debe introducir y las operaciones simples con las que se cuenta en el shader. Debido entonces a su simpleza, el modelo es muy eficiente en las placas gráficas actuales, y, teniendo en cuenta el crecimiento existente y las perspectivas a futuro respecto del hardware gráfico, no presentará problemas a la hora de migrar el mismo hacia hardware más avanzado.

El uso de spot noise para producir distintos tipos de turbulencia probó ser efectivo para modelar características de materiales, además de producir texturas estocásticas. Muchas veces esta asociación entre las características de un material y el spot utilizado no es obvia, sin embargo, en algunos casos como el visto para el granito, resulta de mucha utilidad y produce un resultado que de otra forma sería más complicado, o menos eficiente, o tal vez imposible de lograr sin cambiar la forma de generación de la textura. La utilización de distintas texturas en cada material, permite modelar otros atributos de los mismos, por ejemplo, el pulimiento de un granito.

Se ha presentado una ecuación que permite la representación adecuada de diversos materiales, como mármol, madera y granito. Esto demuestra el poder de un modelo basado en una ecuación que es capaz de capturar su morfología, permitiendo además razonar sobre el mismo, y extenderlo buscando capturar más características que se desea estén presentes en dicho modelo matemático.

Además se ha visto el poder y flexibilidad que presentan los lenguajes de shading a la hora de modelar todas las características que definen a un material. La elección de implementación en la GPU permite seguir la tendencia existente hoy en lo académico y en la industria sobre modelos paralelizables, lo cual garantiza que esta idea podrá ser continuada en el futuro.

## 6.2. Trabajos Futuros

Una de las principales continuaciones de este trabajo radica en la capacidad de extensión de las texturas de spot noise a todo el plano UV, eliminando la restricción que fue introducida en el modelo. Claro está que debe analizarse la factibilidad de esta extensión. Uno de los puntos fuertes para que esta extensión sea posible, es la utilización de GPGPU (CUDA u OpenCL) permitiendo el cálculo de spot noise dinámicamente, en coordenadas arbitrarias.

Tal vez el principal y más interesante trabajo a futuro es la extensión del modelo a tres dimensiones, lo cual representa la principal tendencia que siguen los materiales y texturas actualmente. La idea central de este razonamiento es la extensión de spot noise a tres dimensiones, si es que esto es posible y no se utilizara una versión ligeramente modificada del algoritmo que se vió en dos dimensiones. Las ecuaciones también deberían ser modificadas para capturar características morfológicas de los materiales, con una dimensión más. Sería interesante poder probar la ecuación principal del modelo, para investigar si se adapta a este nuevo marco de razonamiento. Los algoritmos de antialiasing, claro está, deberían ser redefinidos, como así las operaciones.

Con respecto a este último punto, las operaciones que pueden ser agregadas sólo dependen de la imaginación del diseñador. Por ejemplo, existe una operación presentada en el manual de Cg, llamada "twisting", la cual produce que los texels roten en un ángulo que depende de su distancia al centro. Así se pueden deformar texturas en forma de espiral alrededor del centro de la misma. Deberían analizarse operaciones existentes o, tal vez, diseñar otras, que sean útiles para modelar determinadas características de los materiales que se pretendan sintetizar.

Otro punto a tener en cuenta, es la posibilidad de que el usuario diseñe el spot, y el mismo se presente como un parámetro de entrada para el modelo. De esta forma, el usuario podría intentar capturar determinadas características de sus materiales. También contaría entonces con la posibilidad de diseñar distintos tipo de turbulencia, debido a que esa textura que fue ingresada puede ser utilizada para modificar ese parámetro. De esta forma la flexibilidad que presenta el framework se vería incrementada en gran medida, debido a la eliminación de la restricción de las texturas precomputadas offline. Debería analizarse la performance para poder garantizar su efectiva implementación.

El algoritmo de aliasing introducido es sólo una muestra de los muchos que podrían utilizarse. Es posible que existan algoritmos menos propensos a producir artefactos, tal vez a un costo que permita su utilización en el presente modelo. Cualquier algoritmo de antialiasing que se añada debería tal vez ser introducido luego de las otras extensiones al modelo discutidas en este sección, para de esta forma permitir un mejor aprovechamiento de dichas extensiones.

Respecto a la tecnología utilizada, la extensión obvia está dada en la utilización de un modelo de GPGPU, como CUDA. Esto permitiría aún una mayor flexibilidad en el diseño de las características del modelo, además de una mayor vida útil del mismo, al alinearlo con la tendencia que se sigue en el campo de las placas gráficas.

Es perfectamente posible el agregado de otros materiales que se deseen modelar. Tal vez la forma obvia de poder representar mayor variedad de materiales es la extensión o inclusión de otras funciones base. De esta forma se incluirían modelos de otros materiales capturados por otras ecuaciones. El agregado de operaciones es otro candidato para esta extensión, como se mencionó, texturas en forma de espiral podrían ser incluídas por medio de la operación mencionada previamente en esta sección. No es difícil entonces pensar que otras operaciones podrían llevar a la representación de imágenes que hoy resulta muy difícil o imposible en el presente modelo.

El framework presenta una fácil extensión debido a su capacidad de producir texturas representadas por mapas en escala de grises. Esto permite que métodos como bump mapping tengan ya su entrada preparada para poder funcionar. En el método mencionado por ejemplo, se utilizaría la salida del framework para producir el mapa de normales, el cual podría ser luego utilizado para modelar aún más características de los materiales, como por ejemplo el desgaste de una superficie, representado por una mayor o menor aplicación del mapa de normales.

Por último, debido a lo presentado respecto a las funciones de turbulencia, existe la posibilidad de realizar una generalización del concepto. Podría pensarse que, dadas dos funciones cualesquiera sobre el plano, una podría utilizarse para producir turbulencia en la otra. Como se ha visto, incluso con funciones sencillas esta es una buena idea. Así, una determinada forma observada en funciones de una o dos dimensiones (como la cresta de las funciones seno, que presentaba similitudes con formas de madera) podría intentar desplazar coordenadas en otra función. De esta manera tal vez podrían alcanzarse otros materiales no tratados en este trabajo.

# Capítulo 7

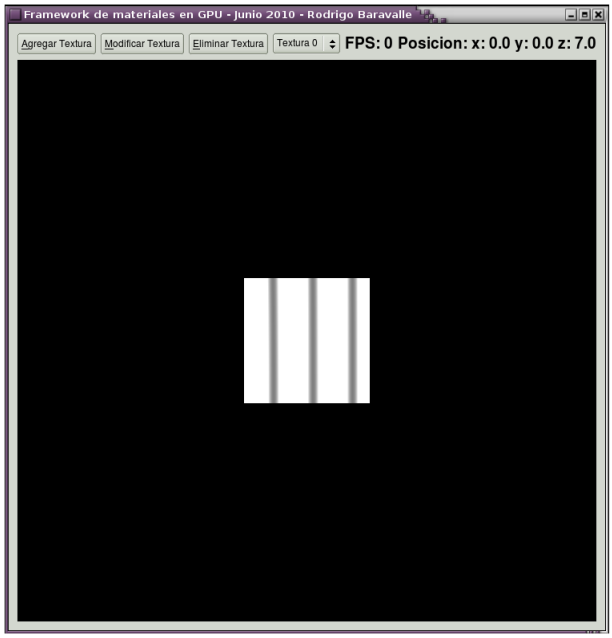
# Apéndice: Manual de Usuario

*"¿Qué tan difícil puede ser? Seguramente el manual indicará qué palanca es el acelerador y cuál el desacelerador"*
*Monty Burns - Los Simpsons - "Homero Smithers"*

## Introducción

Este documento explica la forma de uso del framework. Se desarrollan las dos pantallas presentes en el programa a partir de las cuales se pueden obtener los materiales deseados.

## Escena



La primer pantalla consiste en un *ABM*[1] de texturas. Se permite modificar texturas existentes, agregar o eliminar una de ellas. En esta implementación se limitó la cantidad de las mismas a 10.

En esta pantalla, que representa la *escena* principal, pueden observarse los siguientes elementos constituyentes:

- Botón Agregar Textura.
- Botón Modificar Textura.
- Botón Eliminar Textura.
- Combo de Texturas.
- FPS : cantidad de FPS[2], desactivado por defecto.
- Posición (del observador en el espacio).

El botón "Agregar Textura" agregará una textura en las coordenadas por default (con los valores por default). Dichas coordenadas están situadas sobre el plano $z = 0$. La modificación y la eliminación operan sobre la textura seleccionada en el combo que está a la derecha del botón eliminar textura. Dicho combo entonces, es utilizado para establecer la textura actual.

En todo momento, el observador se encuentra mirando hacia el origen de coordenadas $(0, 0, 0)$. La posición actual del observador es controlable y en todo momento puede observarse en esta pantalla. Al comienzo del programa, el observador se encuentra en la posición $(x, y, z) = (0, 0, 7)$, mirando al origen de coordenadas (intuitivamente, hacia "abajo"). Para controlar la posicion del observador, deben utilizarse las teclas:

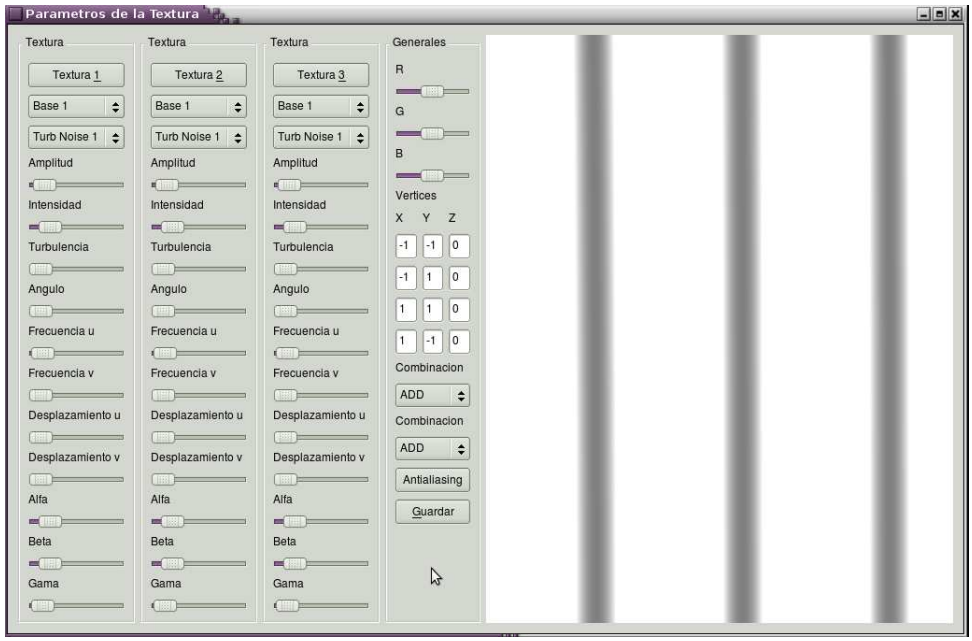
- w : aumentar Z
- s : disminuir Z
- a : disminuir X
- d : aumentar X
- g : aumentar Y
- h : disminuir Y

De esta forma, a medida que el observador cambia de posición, la escena se actualiza (el observador siempre se encuentra mirando al origen de coordenadas) y su posición se ve reflejada en la interfaz, mostrando las coordenadas $(x, y, z)$ actuales del mismo en el espacio.

## Parámetros

Es en esta pantalla donde tiene lugar el proceso de *síntesis* de la textura. Es posible acceder a esta pantalla de dos maneras: haciendo click en "Agregar Textura", en cuyo caso se editará una nueva textura o en "Modificar Textura", donde podrán editarse los valores de la textura seleccionada en el combo. Cada textura cuenta con una serie de parámetros que permiten modificar

[1] Alta-Baja-Modificación
[2] Frames Per Second - cuadros por segundo.




sus caracteristicas. Cada textura a su vez esta compuesta de 3 capas distintas con propiedades independientes, las cuales pueden combinarse entre si.

A continuación se listan los elementos constituyentes de esta pantalla:

- Textura : existen tres columnas iguales que definen cada una de las *capas* de la textura. Cada capa tiene propiedades que son controladas de manera independiente.
- Generales: son propiedades generales de la textura.

Este último punto a su vez se divide en :

- Color RGB[3],
- Vértices en el espacio sobre los que se mapeará la textura: cada fila de input's está compuesta de una coordenada (x,y,z) sobre la que se mapeará el vértice de la textura correspondiente. La primer fila corresponde al vértice $(0, 0)$ en el espacio de la textura, la segunda fila al vértice $(0, 1)$, la tercer fila al $(1, 1)$ y la cuarta al $(1, 0)$, de esta manera los vértices son mapeados en el sentido de las agujas del reloj. En esta implementación, éste es el único mapeo provisto.
- funciones de *Combinación*: el primer combo establece cómo se combinarán las dos primeras texturas, en caso de usarse. Las operaciones posibles son: suma, resta, interpolación lineal (lerp: toma el valor promedio de ambas texturas), sobre (la primer textura se ubica encima de la segunda, tomando como valor transparente en la primer textura aquellos valores cercanos al cero, lo cual corresponde al color negro) y multiplicación. El segundo combo se aplica al valor que surge de la combinación entre las dos primeras texturas y la tercer textura, de esta forma la precedencia es (T1 op1 T2) op2 T3.

[3]Red - Green - Blue: Rojo - Verde - Azul

- Antialiasing : establece si a esta textura se le aplica antialiasing o si no. En muchos casos el antialiasing produce un efecto no deseado en la textura, y por lo tanto puede desactivarse.
- Guardar : cierra la ventana y vuelve a la pantalla anterior.

El primer punto (textura) contiene los parámetros de cada una de las capas que conforman la textura final. A continuación se enumeran los mismos:

- Textura: establece si esta textura se usa o no. Por defecto están todas las capas activadas.
- Base: combo que establece cual textura se utilizará en esta capa. En esta implementación existen 6 posibilidades. Las textura "Base 1" representa círculos concéntricos (aumentar frecuencia v para observar esto). "Base 2" representa un mapa a cuadros (utilizar el mismo parámetro para notarlo) y "Base 3" es una textura especial en la cual se pueden utilizar alfa, beta y gama. En cualquier otro elemento de este combo, esos tres sliders no tienen efecto. Además, existen tres texturas útiles para representar fenómenos naturales. Las mismas comienzan con el nombre "Spot Noise" (para observarlas con mayor detalle, utilizar los sliders intensidad y amplitud, puesto que presentan tendencia a unificarse los valores, visualizándose una textura blanca). Estas últimas texturas, dejan sin efecto todos los sliders excepto los dos mencionados.
- Tipo Turbulencia: establece cómo se deforma la textura al utilizar el parámetro turbulencia que se presenta más adelante.
- Amplitud: permite controlar la separación entre las formas que definen a la textura actual. Así, por ejemplo, en una textua a círculos concéntricos, permite establecer el grado de separación entre los círculos. Mientras más a la derecha está el slider, más separación habrá entre las formas.
- Intensidad: controla el nivel de luminosidad de esta capa.
- Turbulencia: controla el nivel de "ruido" en la textura. Distintos tipos de turbulencia se logran con el combo "Tipo Turbulencia".
- Angulo: permite rotar la textura.
- Frecuencia u: controla el grado de repetición horizontal de la textura.
- Frecuencia v: controla el grado de repetición vertical de la textura.
- Desplazamiento u: permite desplazar la textura horizontalmente.
- Desplazamiento v: permite desplazar la textura verticalmente.
- Alfa: controla la frecuencia de la textura (horizontalmente[4]) en el caso de elegirse "Base 3"
- Beta: establece variaciones a la textura en forma horizontal.
- Gama: permite deformar a la textura en sentido horizontal.

[4]horizontal y vertical aquí representa la orientación de la textura cuando el ángulo es cero

## Ejemplos

Para mostrar la utilización del programa, se muestran dos ejemplos de cómo sintetizar texturas. Primero detallaremos cómo sintetizar una textura de madera, para lo cual deben seguirse los siguientes pasos:

- Al comenzar el programa, se hace click en el botón "Modificar Textura" lo cual permite modificar la textura que se está visualizando en la escena.
- Hacer click en los botones "Textura 2" y "Textura 3" para que se utilice sólo la primer capa. Los siguientes pasos trabajan sobre la primer capa.
- Hacer click en el combo de bases, eligiendo "Base 3".
- Setear el parámetro Beta a cero (cero es llevar el slider hasta la izquierda por completo).
- Modificar el parámetro Alfa, aumentando la frecuencia de las barras verticales.
- Mover el slider "Frecuencia v" sutilmente hacia la derecha.
- Setear el parámetro Gama hasta donde se desee.
- Mover levemente el slider *turbulencia*, para que la textura se deforme sutilmente.
- establecer el color deseado *RGB* (R en el centro del slider, G menor a R, B = 0).
- Opcional: disminuir el parametro intensidad (para eliminar el color blanco excesivo).
- Opcional: hacer click en Textura 2. y elegir como Base un elemento del combo que empiece con las palabras "Spot. . . ". Luego hacer click en el primer botón de combinación, eligiendo "LERP" en lugar de "ADD".

Se obtiene una imagen como la de la Figura 7.1.
En el segundo ejemplo se sintetiza una textura de granito:

- Al comenzar el programa, se hace click en el botón "Modificar Textura" lo cual permite modificar la textura que se está visualizando en la escena.
- Hacer click en los botones "Textura 2" y "Textura 3" para que se utilice sólo la primer capa. Los siguientes pasos trabajan sobre la primer capa.
- Mover el slider "Frecuencia u" hacia la derecha, aumentando la frecuencia de las barras vertcales.
- Mover el slider *turbulencia* hacia la derecha (casi hasta el final), para que la textura se deforme en gran medida.
- establecer el color deseado *RGB* (R en el centro del slider, G menor a R, B = 0).
- disminuir el parametro intensidad (para eliminar el color blanco excesivo).

Se obtiene una imagen como la de la Figura 7.2.

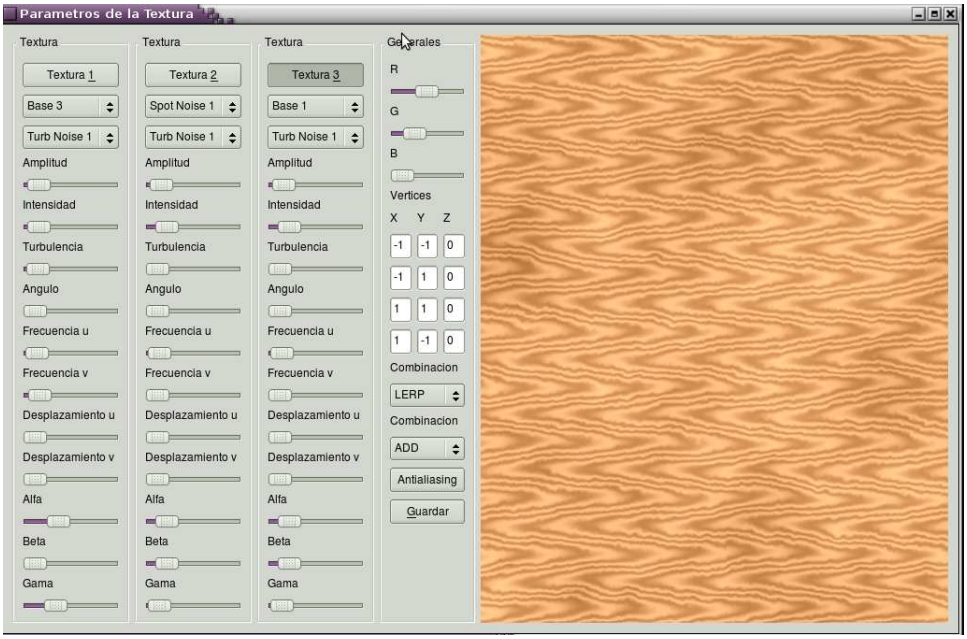



Figura 7.1: Ejemplo de textura de madera sintetizada

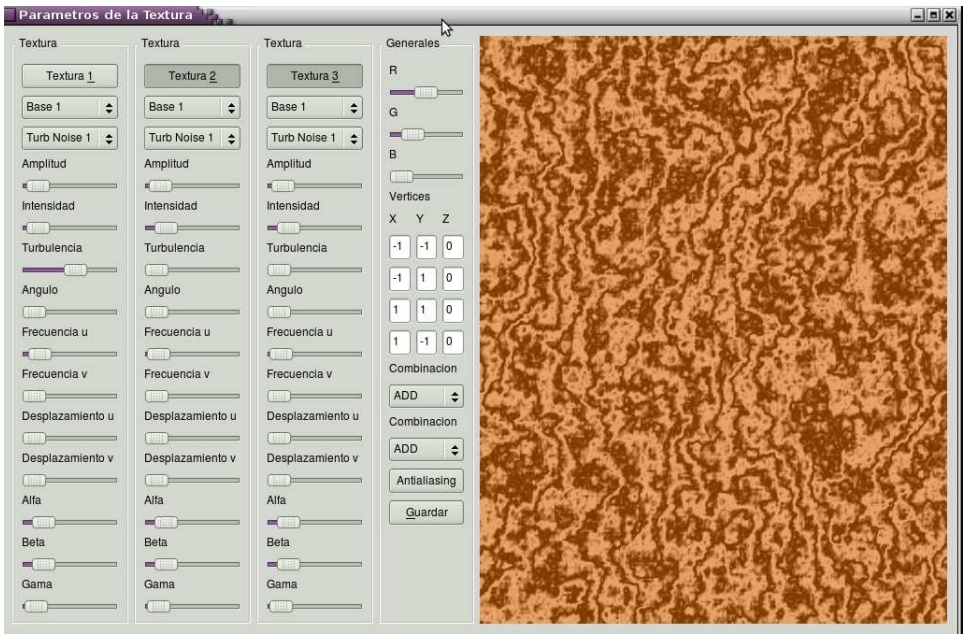



Figura 7.2: Ejemplo de textura de granito sintetizada

# Bibliografía


[1] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM.

[2] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[3] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. *Proceedings of SIGGRAPH 2001*, pages 341–346, August 2001.

[4] Randima Fernando. *Effective Water Simulation from Physical Models.*, chapter 1. Pearson Higher Education, 2004.

[5] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[6] Mark Harris and Kavid Luebke. GPGPU Tutorial. Supercomputing 2006 Conference, 2006.

[7] Paul S Heckbert. Survey of texture mapping. *IEEE Comput. Graph. Appl.*, 6(11):56–67, 1986.

[8] Laurent Lefebvre and Pierre Poulin. Analysis and synthesis of structural textures. In *Graphics Interface 2000 Proceedings*, pages 77–86, 2000.

[9] O. Peitgen, H. Jrgens, and D. Saupe. *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag, 1992.

[10] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[11] Matt Pharr. *Fast Filter-Width Estimates with Texture Maps*, chapter 25. Pearson Higher Education, 2004.

[12] Thomas Porter and Tom Duff. Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 253–259, New York, NY, USA, 1984. ACM.

[13] Jarke J. Van Wijk. Spot noise texture synthesis for data visualization. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 309–318, New York, NY, USA, 1991. ACM.

[14] Alan Watt and Mark Watt. *Advanced animation and rendering techniques.* ACM, New York, NY, USA, 1991.

[15] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.