Simulación de Océano

TECNICAS DE GRAFICOS POR COMPUTADORA

Introducción a la simulación de un océano en tiempo real. Conceptos básicos para la animación de grandes volúmenes de agua. Reflexión, refracción y óptica del océano. Shaders y técnicas de optimización.

CHALLENGE ACCEPTED

Docentes

BARBAGALLO, LEANDRO

LEONE, MATÍAS

Alumnos

PRETE, MARTÍN

PRETE, PEDRO

Contenido

INVESTIGACIÓN PREVIA

Jerry Tessendorf y todo lo que (no) hicimos.

En 1999, Jerry Tessendorf escribió un paper que es considerado por muchos el documento introductorio de referencia para métodos de simulación, animación y render de grandes ambientes oceánicos. En el paper se habla de varios conceptos y técnicas ya existentes, pero se les da enfoques más simples para que puedan ser aplicados en el desarrollo de cada aplicación.

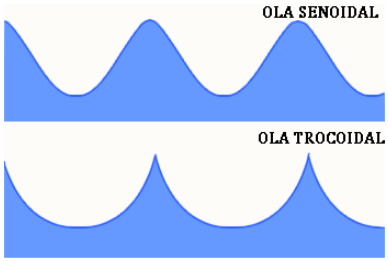
Ampliamente explica: algoritmos de generación/animación de la superficie del agua; el proceso de reflexión/refracción del agua; el color del agua bajo fundamentos ópticos; una introducción a iluminación compleja y algunos temas visuales adicionales.

El océano, como explica Tessendorf y para nuestros propósitos, consiste cuatro componentes principales:

- La superficie del agua.
- El aire.
- El sol.
- El agua bajo la superficie.

Entre los diferentes algoritmos para generación de olas, Jerry menciona primero a la *Grestner Wave*.

Franz Gerstner presentó a principio del 1800 una teoría basada en una ola cuyo perfil tenía forma de trocoide, en inglés se la denominó *Gerstner Wave.*



La *teoría trocoidal* combina una representación geométrica relativamente adecuada, con un tratamiento matemático sencillo, pero con limitaciones en su desarrollo. Se usa mucho en estudios de sensibilidad y movimiento de buques, y otros temas de construcción naval.

En general los Oceanógrafos tienden a considerar el *modelo de Gerstner* poco realista en cuanto al movimiento del océano. En su lugar, prefieren el uso de *modelos estadísticos* en combinación con observación experimental. En *los modelos estadísticos*, la altura de la ola se considera un valor aleatorio, en función de la posición sobre el plano del océano. También se basan en descomponer las diferentes alturas de las olas en una suma de senos y cosenos. Lo importante de esta descomposición es que las amplitudes de las olas descompuestas tienen propiedades matemáticas y estadísticas que permiten generar modelos similares. Para esto se usa la *Fast Fourier Transform* (FFT) que es un método que permite evaluar rápidamente este tipo de sumas.

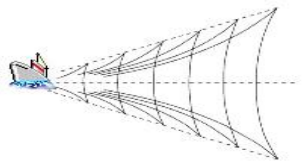
En éste trabajo se decidió optar animar las olas usando una transformación senoidal simple y *Perlin Noise* superficial para darle más realismo.

Tessendorf además explica la óptica de ondas superficiales mencionando la reflexión y transmisión (refracción) y el *Término de Fresnel* con gran detalle matemático. Conceptos similares fueron aplicados al trabajo, pero en un nivel más simple.

Otros conceptos explicados o mencionados, que si bien son interesantes no se puede o tiene pensado implementar son:

La ecuación de *Navier-Stokes (NSE)*, que se usa para describir el movimiento de fluidos viscosos incompresibles.

Kelvin Wedge o *Cuña de Kelvin* que explica matemáticamente el efecto de olas que producen las embarcaciones cuando se mueven sobre la superficie del agua.



Otros papers y artículos muy buenos, como *"Deep-Water Animation and Rendering"* de *Jensen and Golias,* y varias fuentes de información adicionales se encuentran en la sección "REFERENCIAS" al final del documento.

SHADERS

Ventajas y Desventajas de usarlo

Ventajas

El uso de *Shaders* acelera muchas operaciones de cálculo ya que se realiza directamente dentro de la GPU con un lenguaje para GPU, por lo que muchas de las funciones usadas están directamente asociadas a una instrucción de hardware.

Además los *Shaders* agregan paralelismo, ya que cada vértice se procesa como una unidad, independiente de los otros vértices y lo mismo para los pixeles. Esto permite ejecutar en paralelo el procesamiento de varios vértices/pixeles.

Desventajas

En tiempo de programación, no hay que olvidar que programar un *Shader* es una tarea extra, ya que hay que programar o migrar un módulo completo del código original al *Shader*.

Es un camino de ida, en el sentido que no se pueden sacar datos de vuelta para el CPU de una forma fácil u óptima.

La solución a éste último problema es implementar un método que simule la aplicación del *Shader* pero del lado del CPU, éste método será usado como guía y será aplicado a sólo unos pocos vértices bajo demanda.

CPU Shader

La implementación consiste en un método que recibe una posición como parámetro y se le aplican las transformaciones necesarias (emulando las acciones que realiza el *Shader*) para obtener de esta forma una posición transformado bastante parecido a su contraparte que se obtuvo como salida cuándo el vértice fue desplazado en el GPU.

En el caso de éste trabajo se puede ver la solución implementada en el método `AplicarCPUShader`, que emula los cálculos de `MovimientoOceano` del *Shader* pero en el CPU.

SUPERFICIE DEL AGUA

Básicamente, un plano

La superficie del agua es solo un plano generado con muchos vértices al cargar el proyecto. Pero estos vértices son los fundamentos sobre los que se realizará el resto de las acciones. Primero serán desplazados, para generar el movimiento de las olas y luego pintados para reflejar o refractar la luz recibida.

Movimiento

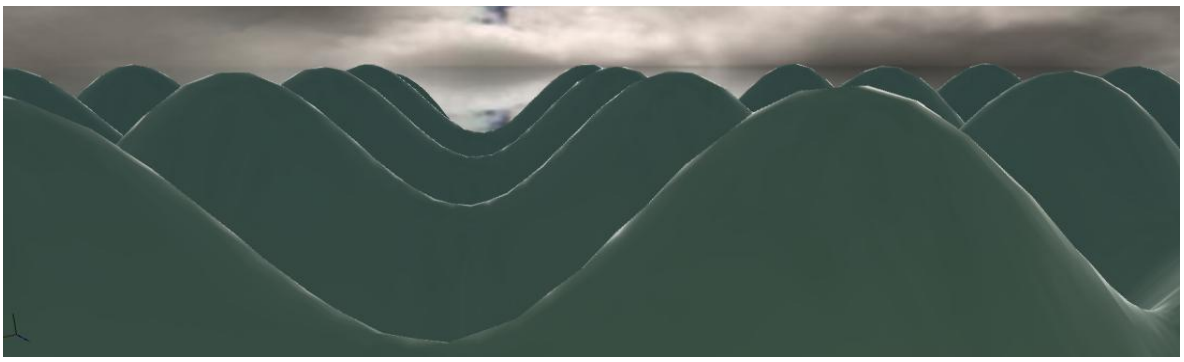
El océano cuenta con 2 movimientos básicos, un movimiento superficial que afecta en menor medida la altura del océano pero que genera ondulaciones visibles las cuales le dan realismo al agua, de lo contrario seria simplemente lisa; y un movimiento de mayor magnitud que es la ola propiamente dicha.

El movimiento superficial es obtenido mediante interpolación de *Heightmaps* generados en base a dos *Perlin Noise* (explicado más adelante), mientras que el movimiento del oleaje esta basado en ondas sinusoidales. Para ello creamos un *Vertex Shader,* en el cual aplicamos a cada vértice una función para calcular la altura que debería tener dicho vértice, esto se conoce también como *displacement shader*.

La función de desplazamiento más grande, es la multiplicación de senos y cosenos, utilizando como ángulo las coordenadas de textura del vértice, modificado por la frecuencia elegida, y luego por la amplitud establecida.

Frecuencia y amplitud normales:

Máxima frecuencia y amplitud:



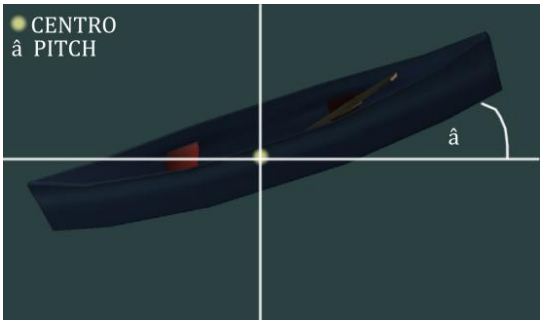
Quiero mover el bote

Para adaptar el modelo del bote al movimiento del mar se requiere llevar a cabo una serie de transformaciones, que permitan que el bote adopte una forma coherente sobre la superficie del océano.

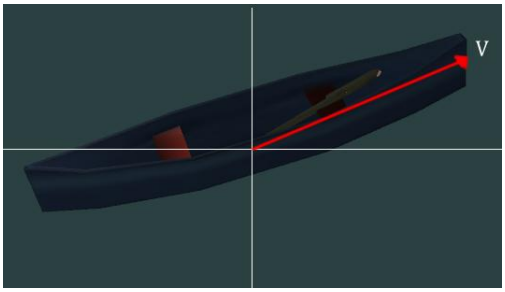
Primero consideramos al barco como un punto en el espacio. Para calcular la posición real del barco puntual con respecto a la superficie del océano que ha sido desplazada, usamos él método que emula la transformación del *Shader* en el CPU y obtenemos la posición correcta (llamaremos a esta posición V1).

Con esta aproximación inicial, el barco solamente sube y baja siendo desplazado sobre el eje Y. Este movimiento no tiene en cuenta que el barco no es un punto sino es una estructura con volumen.

Considerando al barco como un cuerpo real, pero con una dimensión predominante sobre las otras dos para simplificar los cálculos, vamos a conseguir el ángulo que debe ser rotado el modelo para que se adapte a la superficie oceánica sobre la que se encuentra, esta rotación sobre el eje lateral se denomina *Pitch*.



Para calcular el valor de la rotación, debemos buscar un punto en el extremo del barco. Conociendo las dimensiones por ser un *TgcMesh*, podemos usar *su Bounding Box* para estimar el largo que tiene (L).



El nuevo punto se encuentra a una distancia L/2 del centro, llamaremos a este punto V2.

Finalmente crearemos un tercer vector que tenga origen en V1 y fin en V2. Llamaremos a este vector V.

Con la posición actual del barco y el vector V obtenido, solo nos queda armar la matriz de rotación.

Para ahorra tiempo se usó el método CalcularMatriz presente en el ejemplo DemoShader brindado por la cátedra.

RUIDO

Ken Perlin nos hace ruido

Según Wikipedia

> ***Ken Perlin es un profesor en el Departamento de Ciencias de la Computación en la Universidad de Nueva York. Está involucrado con el desarrollo de técnicas como el Perlin Noise, Hipertextura, personajes interactivos de animación en tiempo real, y el zoom de interfaces de usuario entre otras cosas.***

El señor Perlin nos interesa particularmente por su invención en 1985 del ruido que lleva su nombre, **Perlin Noise** o **Ruido Perlin**. Para crear una función de *Perlin Noise* se necesitan dos cosas: una función ruido y una función de interpolación.

Función Ruido

Una *función ruido* o *noise function*, es fundamentalmente un generador de números pseudo aleatorio, que tiene la propiedad de generar el mismo número ante un valor de entrada (*semilla* o *seed*) determinado.

Una función cualquiera, como por ejemplo `y=cos(x)` siempre devuelve el mismo valor de `y` para un valor concreto de `x`. Si utilizamos esa función para generar un modelo geométrico, podemos volver a obtener el mismo modelo posteriormente utilizando las mismas variables. Sin embargo la función no puede ser usada en la representación de un perfil de dunas o montañas porque presenta una repetición periódica, un patrón, que le resta naturalidad. Las dunas del desierto se asemejan unas a otras pero no son todas iguales.

Una función aleatoria que no sea controlable tampoco sirve, porque no permite restituir o volver a obtener unos valores previos, ya que cada vez devolverá valores diferentes.

Una *función perlin* permite mezclar la ventaja de la generación aleatoria (ruido) con el control de una función clásica. Además se pueden obtener valores intermedios entre dos valores dados, dotando de continuidad a estas funciones. Si aumentamos la escala de representación necesitaremos obtener estos puntos intermedios para mantener la resolución de las imágenes generadas.

Interpolación

Una vez que se tiene la función ruido, es necesario suavizar la curva de valores obtenidos para evitar pendientes muy bruscas que generan *picos* y valles muy pronunciados. En general estos detalles no deseados ya que generalmente no son realistas salvo casos muy particulares.

Al obtener imágenes a diferentes escalas es necesario mantener la resolución de las mismas incorporando nuevos puntos. La función debe permitir realizar esta operación para asegurar una calidad razonable en la representación.

Cada *función perlin* genera datos diferentes que se pueden adaptar en mejor o peor medida a la creación de los modelos deseados.

Octavas

Para componer la función final se suman los resultados de las funciones ruido, cuyas frecuencias van doblándose correlativamente.

La frecuencia es la inversa de la longitud de onda. La ecuación $f=1/\lambda$ da noción de la periodicidad de la onda, mientras que la amplitud nos da el tamaño (altura de nuestra onda).

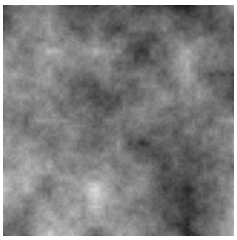
Cada una de estas funciones que doblan la frecuencia anterior se denominan *octavas*.

Suavizado del Ruido

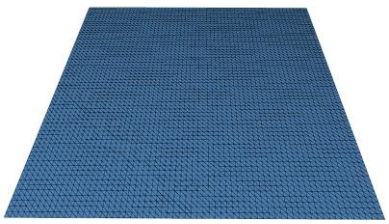
Además de la interpolación, para obtener un mejor efecto, se realiza el suavizado del ruido generado para que sea aún menos aleatorio. En lugar de tomar solamente un valor de la función ruido, se toma el promedio del valor, tomando en cuenta los valores de los puntos vecinos.

Texturas y Heightmaps

El *Perlin Noise* es muy bueno para la generación de texturas y terrenos. Un *Heightmap* es una buena forma de almacenar los valores obtenidos del *Perlin Noise*. Suponiendo los valores normalizados en un rango de valores determinados (a escala de grises por ejemplo) se puede acceder la información como un *Heightmap* previamente creado sin necesidad de tener que generar el ruido cada vez que lo necesitemos.



Heightmap de Perlin Noise



Plano

Plano desplazado según el Perlin Noise.

Como otra ventaja, es más fácil acceder a esta información en forma de textura cuándo nos encontramos trabajando con Shaders, dónde hacer una búsqueda dentro de las texturas para obtener información (*texture lookup* o *VTF: Vertex Texture Fetch*) es muy común.

Aplicación del concepto a este trabajo

El *Perlin Noise* lo usaremos para realizar las perturbaciones superficiales del océano. Son las pequeñas olas que forman parte de la superficie.

Tomamos el plano base que tenemos para nuestro océano y desplazamos sus vértices para que tome una forma aleatoria pero que todavía parezca lo suficientemente natural como para simular el movimiento natural que existe en grandes cuerpos de agua que no están estáticos.

Generación de Perlin Noise en tiempo Real

Si bien el *Perlin Noise* nos permite tener perturbaciones aleatorias para la superficie del agua, calcularlo todo el tiempo y en tiempo real es un costo importante para el CPU.

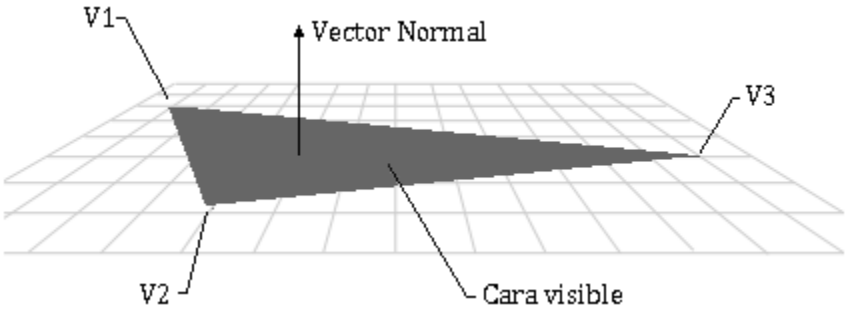
Para solucionar esto, generamos dos *Perlin Noise* al momento del inicio de la aplicación y los cargamos en texturas diferentes. Estos *Hightmaps* son pasados al *Vertex Shader* para desplazar los vértices a su altura correspondiente, consiguiendo una superficie irregular.

Para animar todo esto, realizamos la interpolación entre ambas texturas en base al tiempo transcurrido. Con esta acción las posiciones de los vértices cambian de una forma suave entre los valores que le corresponderían si se desplazaran con cada uno de los *Hightmaps* de ruido. De esta forma sacamos provecho del poder de procesamiento del GPU y ahorramos preciosos ciclos del CPU.

NORMALES

Calculo de normales

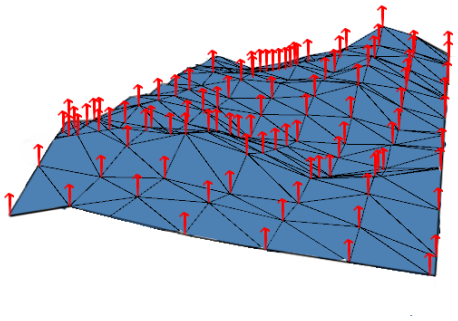
Si tenemos los tres vértices del triángulo, V1, V2 yV3, suponiéndolos en sentido anti horario, se puede obtener la dirección de la normal con la siguiente fórmula N= (V2 - V1) x (V3 - V1).



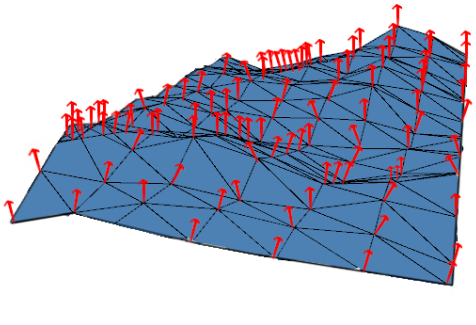
Normalmap y Normales pre-calculadas

Supongamos el Plano XZ, es un plano generado por muchos triángulos como el del ejemplo anterior. Recordando que los triángulos siempre residen en un único plano, lo que significa que sus 3 vértices son coplanares, todos los vértices pertenecen al Plano XZ, por lo que la normal es igual en todos los vértices, y sabemos que la normal del Plano XZ es N=(0,1,0)

Pero nosotros no estamos trabajando sobre un plano, dónde todas las normales son iguales, sino que cada vértice ha sido desplazado usando un *Heightmap* calculado a partir del *Perlin Noise* generado.



Normales incorrectas.



Normales correctas.

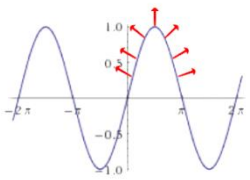
En este caso, los valores de las normales pueden haber sido calculados al mismo tiempo que se generó el *Heightmap* original. Inicialmente en este trabajo, se calculaba una matriz de vértices normales que correspondía con cada uno de las posiciones de la matriz de ruido generada. Esa matriz de ruido era a su vez convertida en una textura donde cada vector era representado por un

pixel de un color en particular, obteniendo de esta forma una imagen que contiene la información llamada *Normalmap*.

Ahora si podemos asignar a cada vértice desplazado usando el *Heightmap* una normal correspondiente, buscando en la coordenada de textura del *Normalmap*.

Probelmas del Normalmap

Una vez asignadas las normales al conjunto de vértices que forman el Océano, nos encontramos con otro problema. Los vértices no son solamente desplazados por un *Heightmap*, sino que es la interpolación de dos *Heightmaps* lo que da el movimiento superficial.



Normales de la función seno.

Además, se aplicó también otro desplazamiento con forma sinusoidal en el *Shader*, en este punto las normales están lejos de ser exactas.

Cálculo de normales en tiempo real

Nos encontramos con el problema de que el *Normalmap* no fue suficiente para el cálculo de las normales. Tenemos por un lado la interpolación de *Perlin Noises* y por otro lado el desplazamiento adicional con funciones seno y coseno. Llevando a ambos desplazamientos al *Shader*, no queda otra que mover la lógica del cálculo de normales al *Shader* e ir calculando las normales por vértice.

Más solo que vértice en el Shader

Ahora estamos en el *Shader*, al ponernos a calcular las normales nos encontramos con otro problema. El vértice con el que estamos trabajando está solo, sin conocer a nadie más.

El paralelismo es una ventaja de los *Shaders*, pero también aumenta el nivel de complejidad en algunos casos. El problema del cálculo de la normal ahora tiene más incógnitas, ya que no conocemos ninguno de los otros vértices del triángulo.

Para solucionar este problema se siguió la siguiente lógica:

> Tenemos 1 vértice. Necesitamos 3 vértices. ~~Inventamos~~ Fabricamos 2 vértices. Calculamos.

Se dispone de un vértice de trabajo que llamaremos $\mathrm{V1}$.

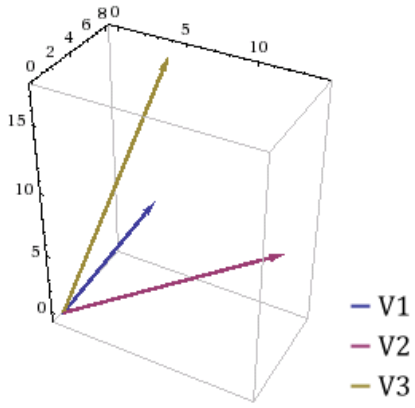
Además en el *Shader* está definida la función $\mathrm{MovimientoOceano}$ qué es la encargada de transformar un vértice a su posición final desplazandolo de acuerdo al movimiento sinusoidal primero y luego desplazandolo una vez más con el valor del *Perlin Noise* interpolado. Éste desplazamiento sólo se realiza en dirección de Y, por lo que los valores de X, Z resulantes serán los mismos que al comienzo, pero se usan para hacer *texture fetch* que nos permite aplicar el *Perlin Noise*. Para fines prácticos llamaremos a la función MovimientoOceano $\mathrm{M(x,z)}$.

Búsqueda de vecinos o como fabricar vértices

Tenemos un vértice y necesitamos dos, la forma de generar los otros dos vértices es más simple de lo que parece.

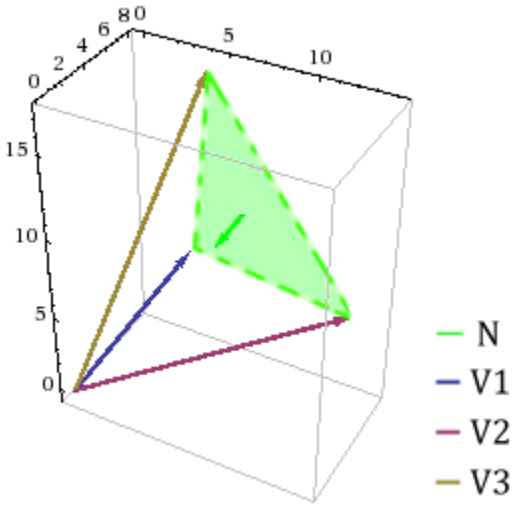
Sabemos que para una posición P(x,y,z), al aplicarle M(x,z) obtenemos P'(x,y',z).

Entonces podemos fabricar dos vértices más que pertenecen al conjunto de vértices que fueron desplazadas por el movimiento del océano M(x+Δ, z) y M(x , z+Δ).



Si los Δ usadas para fabricar los nuevos V2 y V3 no se alejan mucho del vértice original V1, podemos asumir que éstos pertenecen al mismo triángulo y como ya mencionamos antes, todos los vértices de un triángulo pertenecen al mismo plano.

Finalmente tenemos los 3 vértices (V1, V2, V3) necesarios para el cálculo de la normal.



Los últimos cálculos corresponden a la formula descripta al comienzo.

Tomamos el vector que forma la primer arista del triángulo haciendo (V2 - V1), en la figura es la arista inferior. Luego, con el segundo vértice encontrado operamos de igual manera (V3 - V1) obteniendo el vector para la segunda arista, en la figura es la de la izquierda.

Finalmente, sabiendo que el producto vectorial da como resultado un vector normal al plano que contiene los vectores iniciales, conseguimos la normal N.

Implementación

El código para el cálculo de normales se encuentra dentro del *Shader*, en particular en la función:

```
float3 GenerarNormal(float3 pos)
```

ENVIROMENT MAP

Reflejos de todo el mundo, Enviroment Map.

Environment mapping es una técnica que simula el resultado del seguimiento de rayos en un ambiente. Generalmente es el método más rápido para renderizar superficies que reflejan la luz. Para incrementar la velocidad de renderizado, se realizan los cálculos de reflexión de rayos en cada vértice directamente en el *Shader*, haciendo uso del poder de procesamiento del GPU y eliminando la necesidad de hacer el cálculo por pixel.

Existen varios tipos de *Enviroment mapping* tales como: *HEALPix mapping*, *Sphere mapping*, *Cube mappin*g. En el trabajo práctico usamos ésta última.

Cube map y reflexión global

La reflexión de un *Cube map* lo consideramos como *reflexión global*, lo que sería objetos considerados infinitamente lejos. Como la distancia a la superficie reflectada tiende a infinito, la parte de la superficie que refleja solo depende del vector reflexión de la superficie a la que estamos viendo.

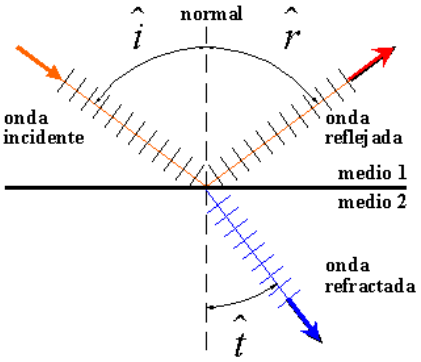
El *Skydome* de nuestro trabajo es un buen ejemplo de esto. El vector reflexión es calculado en el *Vertex Shader*, y luego tomado en el *Pixel Shader* para hacer un *cube map lookup* de la textura que corresponde al cielo (que es un *cube map*). Por suerte, para calcular en que parte del cubo esta incidiendo el haz reflejado existe una instrucción básica en HLSL que ayuda a realizar esto: $\mathrm{texCUBE}(\mathrm{Cubemap}, \mathrm{R})$. El resto de los cálculos pueden verse dentro del *Pixel Shader*.

Océano sin movimiento reflejando el cielo:

Nociones básicas de óptica

Reflexión y refracción

La *reflexión* es el cambio de dirección de un rayo o una onda que ocurre en la superficie de separación entre dos medios.



La *refracción* es el cambio de dirección que experimenta una onda al pasar de un medio material a otro. Solo se produce si la onda incide oblicuamente sobre la superficie de separación de los dos medios y si estos tienen índices de refracción distintos.

Ley de Snell

La ley de la refracción fue enunciada en 1621 por *Willebrord Snel*. La *Ley de Snell* es una fórmula utilizada para calcular el ángulo de refracción de la luz al atravesar la superficie de separación entre dos medios. Sin embargo, la *Ley de Snell* no tiene en cuenta la reflexión de la luz, solo el ángulo de la onda refractada.

Fresnel

Como se ve el agua depende en gran parte del ángulo con el que la miramos. Cuándo estamos mirando directamente al agua, y si es lo suficientemente clara, podemos ver a través de ella. Pero si miramos más hacia el horizonte, el agua comienza a comportarse como un espejo.

Augustin-Jean Fresnel fue un físico francés que estudió el comportamiento de la luz y contribuyó a la teoría de óptica ondulatoria. Nos interesa porque estudió el comportamiento de la luz al desplazarse entre medios que tienen índices de refracción distintos. Las *Ecuaciones de Fresnel* dan el grado de reflectancia y trasmitancia en el borde entre dos medios de densidad diferente. La expresión matemática es algo complicada, pero si se desea ver el término completo se puede consultar la referencia.

Fresnel tomo lo que había hecho *Snel* y lo amplio, no solo relacionando ángulos e índices de refracción, sino que bajo unos escalones más y se metió a estudiar la relación entre las amplitudes de longitudes de onda al atravesar diferentes medios.

Básicamente, la onda tiene dos componentes: uno paralelo y uno perpendicular. Como variables entran en juego la amplitud de la onda incidente, la amplitud de las ondas reflectada y transmitida (refractada), así como también los ángulos entre la normal, la superficie, y los haces de luz incidentes, reflejados y refractados.

Matemática de lado, el índice de refracción del agua (en diferentes ambientes) ya han sido calculados y tabulados. Por lo tanto es posible pre calcularlo y guardarlo en una textura uni-dimensional.

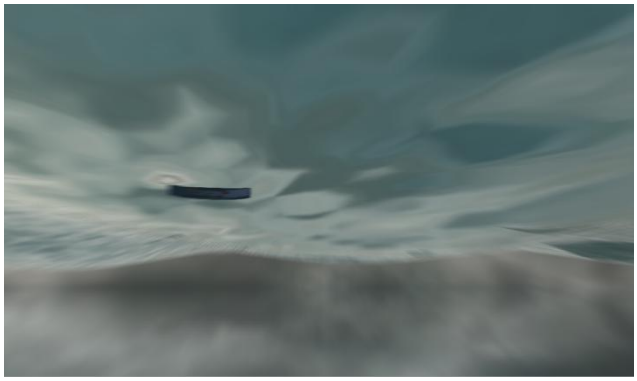
Ahora a lo importante, el *Coeficiente de Fresnel* nos dará el peso para interpolar entre la imagen reflejada y la refractada por la superficie del agua. Esto se puede ver dentro del *Pixel Shader* del océano.

BAJO EL AGUA

Ya que se han descripto las técnicas principales de modelado sobre el agua, queda pendiente una parte importante de una simulación completa: el ambiente submarino.

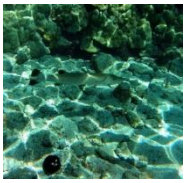
Refracción, Caustics, Godrays, Blurring

La *Refracción* es el cambio de dirección que experimenta una onda al pasar de un medio material a otro. Esto es importante tenerlo en cuenta para modificar la forma en que se ven objetos que están abajo del agua y, sobretodo en nuestro trabajo práctico, para cuando la cámara esta sumergida y se mira hacia la superficie, la forma en que se refracta el cielo.



Blurring es una técnica de esfumado con la cual se logra un efecto de vista nublada en la cámara. El trabajo lo tiene implementado como efecto de visión borrosa al introducirse al agua, este esfumado se logra en el post procesado, al interpolar pixeles próximos.

Godrays o *Crepuscular Rays* es un efecto atmosférico en el que rayos de luz parecen provenir de un punto particular en el cielo.



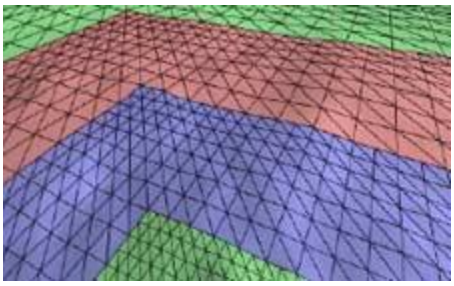
Caustics se le llama a la red de haces de luz formados por la reflexión/refracción de objetos curvos.

Por falta de tiempo *Godrays* y *Caustics* no fueron implementados.

OPTIMIZACION

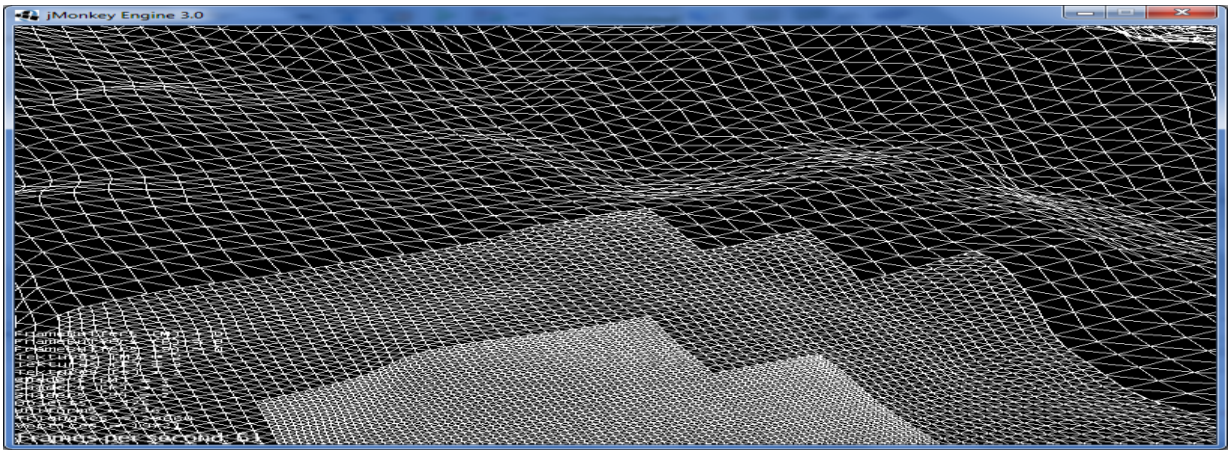
Rende rizar el océano completo es una operación muy costosa, por lo que se deben utilizar técnicas de optimización para lograr un nivel aceptable de FPS s sin perder calidad.

Level of Detail (LOD)

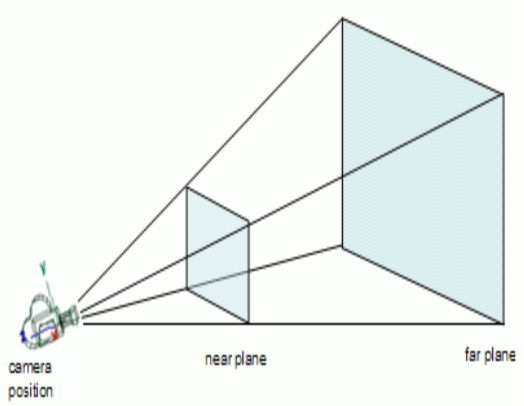


En la vida real cuando alguien esta a centímetros de un árbol uno puede apreciar la corteza del mismo. Pero si uno se encuentra a 100 metros de este no es posible distinguir mucho mas allá del tamaño, color y tipo de árbol, mucho menos que corteza tiene. *LOD* considera este hecho y lo utiliza disminuyendo la cantidad de vértices que se grafican cuanto mayor sea la distancia de la cámara, disminuyendo así el nivel de detalle pero ahorrando recursos y permitiendo mejorar la performance.

Para realizar esto se elige una longitud, esa longitud determinara la *franja de nivel de detalle*. La primera franja comienza desde la posición de la cámara hasta la longitud elegida. Los vértices dentro de esta franja son graficados en su totalidad ya que están próximos a la cámara y no queremos perder detalle alguno. La segunda franja comienza con los vértices que se encuentran a una distancia mayor a la longitud hasta una distancia de 2 veces la longitud original, como los mismos están mas alejados no necesitamos tanto nivel de detalle y la forma de disminuirlo es saltear de la matriz de vértices una columna y una fila por medio. De esta forma se obtienen triángulos que son el doble del tamaño original. La tercera franja comienza desde 2 veces la longitud original (el fin de la franja anterior) hasta el infinito. Se pueden colocar cuantas franjas se desee, pero es importante no bajar mas de un determinado nivel de detalle. En esta franja al igual que la anterior vamos a bajar el nivel de detalle aumentando el tamaño de los triángulos, pero a diferencia del anterior esta ves vamos a saltar de la matriz de vértices 2 filas y 2 columnas por medio, obteniendo un triangulo del triple del tamaño original.



Frustum Culling



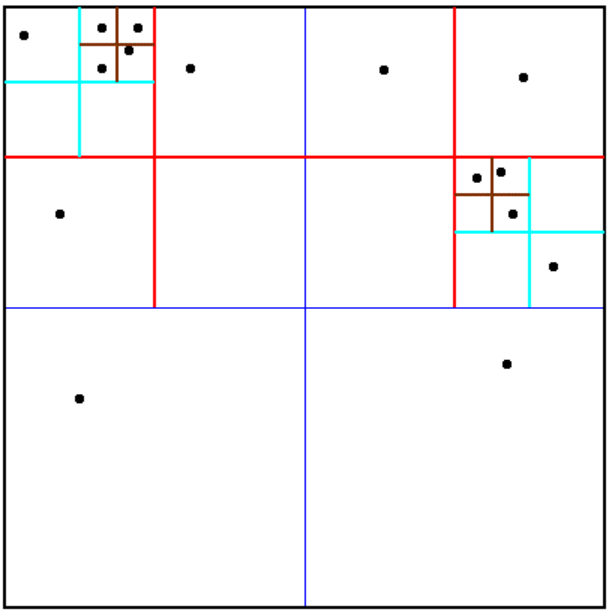
Frustum es el volumen que contiene todo lo potencialmente visible (puede haber oclusión) y es representado con la forma de una pirámide truncada.

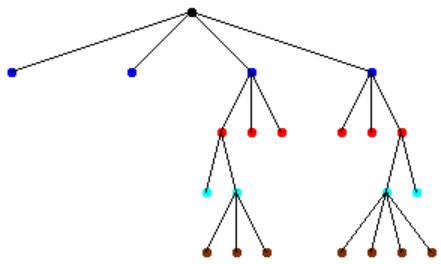
El objetivo de *Frustum Culling* es renderizar solamente los objetos que se encuentran dentro de *Frsutum*, logrando así ahorrar procesamiento.

Detectar vértice por vértice si este se encuentra dentro del *Frustum* es poco performante. Para solucionar esto utilizamos un *QuadTree*.

El *QuadTree* es un árbol de orden 4 que nos permite dividir el terreno en cuadrantes. A su vez cada cuadrante es dividido en 4 y así sucesivamente n veces.

Cuando se inicializa el programa se dividen los vértices a lo largo de la estructura del *QuadTree* de acuerdo a su posición. A la hora de buscar un objeto dentro del *Frustum* en lugar de ver objeto por objeto nos fijamos por cada cuadrante, si este esta completamente afuera del *Frustum*, no lo renderizamos, si este esta completamente dentro del *Frustum* lo renderizamos, si esta parcialmente dentro del *Frustum*, realizamos la misma operación para cada uno de sus hijos. Así obtenemos de forma rápida y sin gran cantidad de procesamiento cuales son los vértices que se encuentran dentro del *Frustum* para ser renderizados.





En el trabajo, dada la gran cantidad de vértices que forman el Océano, éstos fueron divididos usando este tipo de estructura para optimizar el rendimiento.

REFERENCIAS

Jerry Tessendorf. “Simulating Ocean Water”. http://graphics.ucsd.edu/courses/rendering/2005/jdewall/tessendorf.pdf

Wikipedia. “Ken Perlin”. http://es.wikipedia.org/wiki/Ken_Perlin

Ken Perlin. Página Personal. http://mrl.nyu.edu/perlin/

Microsoft Windows Dev Center. HLSL. http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

Claes Johanson. “Real-time water rendering - introducing the projected grid concept” http://fileadmin.cs.lth.se/graphics/theses/projects/projgrid/projgrid-hq.pdf

Wolfram MathWorld . “Normal Vector”. http://mathworld.wolfram.com/NormalVector.html

Georgia State University. “HyperPhysics”. http://hyperphysics.phy-astr.gsu.edu/hbasees/hframe.html

Wikipedia. “Coeficientes de Fresnel”. http://es.wikipedia.org/w/index.php?title=Coeficientes_de_Fresnel

Lasse Staff Jensen and Robert Golias. "Deep-Water Animation and Rendering". http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm

Virtual Terrain Project. "Terrain LOD". http://vterrain.org/LOD/Implementations/