Universidad Zaragoza

Trabajo Fin de Grado

Kubex, desarrollo de un motor gráfico 3D basado en cubos

Autor

Víctor Arellano Vicente

Director

Eduardo Mena Nieto

Grado en Ingeniería Informática
Escuela de Ingeniería y Arquitectura
2016

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

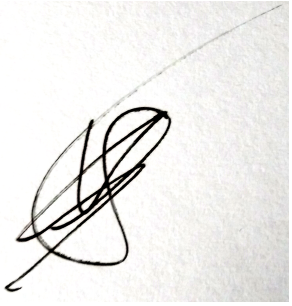
D./Dª. Víctor Arellano Vicente,

con nº de DNI 78754388-L en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Kubex, desarrollo de un motor gráfico 3D basado en cubos.,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 22 / 04 / 2016



Fdo: Víctor Arellano Vicente

RESUMEN

Un nuevo género de videojuegos basados en cubos e inspirados en el famoso título Minecraft (Mojang AB, 2009) está irrumpiendo en el mercado, creciendo rápidamente en popularidad. No obstante, no existen aún motores gráficos *open-source* populares especializados en este género. Además, los motores gráficos tradicionales no son capaces de explotar suficientemente las características que un mundo dividido en cubos ofrece, y no permiten desarrollar este tipo de videojuegos con facilidad. Por esta razón, los títulos de este género están forzados a implementar su propio motor gráfico cuyo desarrollo es, dado el amplio coste en tiempo y recursos que la creación de un motor gráfico complejo conlleva, relevado a un segundo plano en la mayor parte de los casos. Esto causa un estándar de calidad gráfica en este género realmente bajo, muy por debajo del estándar de industria actual.

En este proyecto se implementa, usando tan solo Java y OpenGL, desde cero y en código abierto, un motor gráfico orientado a cubos con la calidad gráfica y la eficiencia como máximas prioridades. Sobre el mismo se ha desarrollado, asimismo, un videojuego basado en la exploración y creación de estructuras, con todas las características que un título de este género posee.

Es nuestra intención que, dada la naturaleza de código abierto de este título, cualquier programador interesado en este género pueda basarse en este proyecto para resolver sus dudas, obtener partes de código para implementar funcionalidades que necesite o, incluso, extender fácilmente este proyecto a un prototipo de mayor jugabilidad, centrándose únicamente en el desarrollo del mismo y despreocupándose del motor básico, ya implementado.

Agradecimientos

A mi familia, por su apoyo.

A mis amigos, por sus críticas.

Y a Internet, todos esos miles de desconocidos que, dedicando su tiempo y esfuerzo a dejar conocimiento al alcance de todos, sin saberlo, han hecho esto posible.

TABLA DE CONTENIDOS

1. INTRODUCCIÓN

El popular título Minecraft (Mojang AB, 2009) protagonizó una auténtica revolución en el mundo de los videojuegos, al crear de la nada un género absolutamente nuevo, basado en mundos formados por cubos. La absoluta libertad que permitía, dando rienda suelta a la imaginación de los jugadores, lo catapultó a un éxito inmediato, pese a sus gráficos de aspecto *retro* (ver figura 0 a continuación), en desuso en esos tiempos. Tras él han surgido numerosos títulos, todos compartiendo esa estética alejada de los nuevos estándares de industria en cuanto a gráficos se refiere. Un motor gráfico basado en cubos y orientado a la calidad gráfica es pues un desafío inexplorado e interesante, que vamos a afrontar en este proyecto.

Figura 0 – Minecraft, *in game*

Crear un motor gráfico es una tarea ardua y compleja, que requiere conocimiento de múltiples campos de la Informática. Es por ello que normalmente se desaconseja desarrollarlos, recomendando en su lugar el uso de alguno de los muchos motores ya existentes con la famosa frase *"write games, not engines"*.

En el género de videojuegos de cubos, no obstante, esto no se aplica. De los motores gráficos más populares en la actualidad ninguno está optimizado para este género, forzando a los desarrolladores a enfrentar, de forma obligada, el desafío de programar su propio motor gráfico, careciendo la mayoría del interés, conocimientos o tiempo necesario para ello. Muchos proyectos de este tipo son abandonados en este punto por frustración, y la mayoría de los que logra desarrollar un motor prototipo básico detienen ahí la evolución del mismo, comenzando a implementar la parte que ellos realmente deseaban desde un principio: Un videojuego.

Existe mucha información en la red sobre el desarrollo de juegos de este tipo, pero toda parece acabar cuando se trata el tema gráfico. A partir de este punto solo quedan meros apuntes genéricos, ya que los pocos proyectos existentes que se han centrado en la

implementación de los mismos se mantienen en la oscuridad, sin liberar su código ni dar explicación alguna de las metodologías que han seguido para cada caso.

Este proyecto tiene como objetivo afrontar el desafío del desarrollo de esta área tan olvidada en este género, a la par que de intentar en lo posible servir de ayuda a todos los que, después de mí, tengan este mismo propósito.

1.1. Objetivo y alcance

Se implementará un motor gráfico y lógico basado en cubos (así como una versión jugable del mismo), con todas las características comunes a los videojuegos de este género (mapas procedurales infinitos, terreno dinámico modificable en tiempo real, iluminación dinámica independiente al número de luces, motor físico, movimiento de fluidos, carga y guardado del mundo en mapa, ciclos de día y noche, etc.), a la par que algunas características gráficas no tan comunes (sombras dinámicas, *atmospheric scattering* o agua realista, entre otras).

1.2. Metodología y herramientas

Todo este proyecto ha sido realizado desde cero y al nivel más bajo posible, en Java [15] y OpenGL 3.2 [14], requiriéndose una tarjeta gráfica que lo soporte para poder ejecutar el prototipo. Al no ser posible acceder a las librerías de OpenGL desde Java directamente, se ha hecho uso de la librería LWJGL (**L**ight**W**eight **J**ava **G**raphics **L**ibrary)[1], que actúa tan solo de intermediaria entre ambos, ofreciendo únicamente funciones análogas a las de OpenGL.

Dada la dificultad de tratar con texto en OpenGL, se ha hecho uso de la librería SlickUtil[2] para ese único propósito. Asimismo, dada la necesidad de decodificar archivos PNG a una cadena de bytes a la hora de subirlos a la GPU, se ha usado la librería PNGDecoder[3].

El uso de Java en este proyecto también tiene el propósito de desmentir las muchas afirmaciones que defienden que este lenguaje no es válido para el desarrollo de videojuegos en tiempo real dado su recolector de basura y su condición de lenguaje interpretado, o que no es válido para la realización de juegos con gráficos avanzados. Java es un lenguaje perfectamente capaz para todas estas tareas, y la única razón por la que no es usado para ellas es por ese prejuicio falso sobre su lentitud, tan extendido hoy dia. Esto ha ocasionado, al alejar a posibles desarrolladores de esta área, una gran falta de herramientas especializadas y documentación sobre estos temas en comparación con otros lenguajes, como C++.

Durante el desarrollo, se ha utilizado un sistema de control de versiones ubicado en GitHub[4].

[1] *LWJGL 2*. [Citado el 18/06/2016] https://www.lwjgl.org/

[2] Slick-Util. [Citado el 18/06/2016] http://slick.ninjacave.com/slick-util/

[3] Loading PNG images with TWL's PNGDecoder. [Citado el 18/06/2016] http://wiki.lwjgl.org/wiki/Loading_PNG_images_with_TWL's_PNGDecoder

[4] https://github.com/Ivelate/Kubex

Todo este motor se ha implementado utilizando un computador *AMD Athlon 64 X2 Dual Core 2'8GHz, 4GB RAM, ATI Radeon 4800*. Es un equipo de muchos años de antigüedad, y podemos considerar que solo llega a los requisitos mínimos para la ejecución de este proyecto a calidad gráfica máxima. Para alcanzar los requisitos recomendados se aconseja contar con una tarjeta gráfica de gama superior, como, idóneamente, una de la familia GTX 600 en adelante.

1.3. Contenidos de la memoria

Esta memoria se divide en dos partes claramente diferenciadas. Primero, se detallarán todas las partes lógicas del motor (la estructura de datos en las que se dividen los cubos, generación procedural de mundos, estructura multihilo implementada, gestión del mundo dinámico modificable en tiempo real, motor físico, optimizaciones de eficiencia y gestión de datos en disco), para detallar a continuación las partes gráficas (Iluminación, sombras en tiempo real, agua realista, *Deferred Rendering*, optimizaciones de eficiencia gráfica, etc.).

Tras ello, quedará la conclusión y la bibliografía, seguida de cuatro anexos: Un manual de usuario, un anexo en el que detallamos que procedimientos hemos seguido para elaborar el *render* de agua, un anexo en el que detallamos como hemos realizado el algoritmo de sombras y, por último, un anexo con fotografías extras, con objeto de no aumentar más aún el volumen de la memoria.

1.4. Licencia usada

Se ha optado por el uso, en todo el código del proyecto, de la licencia menos restrictiva de CreativeCommons, ***Attribution 4.0 International***[5]. Cualquier persona que lo desee podrá usar cualquier parte de este trabajo para lo que desee, sea comercial o no, y podrá añadirle la licencia que considere oportuno. La única restricción será la obligación de mencionar al autor de este proyecto si lo hace.

[5]Creative Commons, Attribution 4.0 International. [Citado el 18/06/2016] https://creativecommons.org/licenses/by/4.0/

2. LÓGICA DEL SISTEMA

Se detallan a continuación todas las consideraciones tomadas en el desarrollo de la parte lógica del motor, ejecutadas sobre la CPU del ordenador y usando la memoria RAM. Toda optimización detallada más adelante estará, por tanto, orientada tanto a relajar la carga computacional del microprocesador como a reducir la cantidad de memoria usada. Debemos considerar que trabajamos con Java, con lo que es posible que la memoria asignada a la ejecución de este motor sea limitada, siendo doblemente importante aplicar las optimizaciones al respecto que sean pertinentes.

Los juegos de cubos poseen algunas características que los hacen únicos con respecto a otro tipo de géneros, siendo esta la razón por la que un motor gráfico genérico no está bien adaptado para la creación de los mismos. Una longitud infinita de mundo, la posibilidad de simplificar los mapas procedurales dada la ausencia de detalles de tamaño menor que un cubo, u optimizaciones posibles en el motor de físicas al asumir que no existen obstáculos intermedios entre bloque y bloque son algunas de las posibles razones.

2.1. Cubos

Todo nuestro mundo se basa en cubos de $1m^3$ de tamaño. Cada cubo en el mundo está identificado por un byte, reduciendo así el gasto de memoria (y disco) necesario para mantenerlos. El número máximo de los mismos, por tanto, estará limitado a 256 tipos de cubos diferentes. Todos los cubos existentes hasta el momento en el juego prototipo desarrollado pueden apreciarse en la figura 1. De forma natural aparecerán tan solo el tocón de árbol y las hojas (esquina superior izquierda), la arcilla, la hierba y la nieve (Esquina inferior derecha), el agua (centro, fila inferior) y la vegetación, tanto verde como amarilla. El bloque en la parte superior derecha se trata de un bloque indestructible, que aparece en la altura cero del mundo con objeto de impedir al jugador seguir picando y caer al vacío. El resto de estos son cubos sólidos estéticos y usados para construir, a excepción del cubo de luz, el cubo de cristal y el cubo de TNT, siendo estos, respectivamente, el tercero, segundo y primero por la izquierda en la fila inferior.

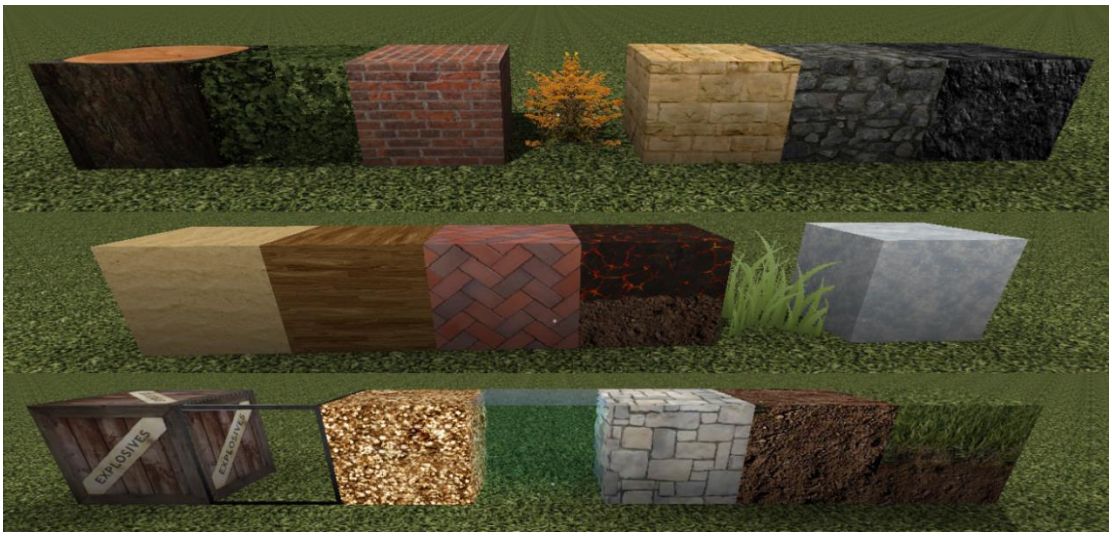


Figura 1 - Todos los cubos existentes

Los cubos se basan en un sistema de propiedades, abstrayendo cada cubo de su identidad real y solo interaccionando con el mismo en función de cada propiedad. Gracias a ello, podemos añadir nuevos cubos fácilmente, al solo tener que preocuparnos por asignarles cada propiedad requerida. En concreto, estas son:

- *getCubeName*: Obtiene el nombre concreto del cubo. Usado al seleccionarlo.
- *isSolid*: Define si el jugador puede atravesar este cubo andando. Un ejemplo de cubos no solidos son el agua, el aire o la vegetación.
- *isOpaque*: Define si la luz puede atravesar el cubo.
- *canSeeTrough*: Define si el cubo es semitransparente. Si lo es, tendremos que dibujar también los cubos que se encuentren tras este, ya que no podemos asegurar que los esté cubriendo.
- *isPartnerGrouped*: Solo para cubos semitransparentes. Define si los cubos transparentes dibujan las caras en contacto con otros cubos transparentes del mismo tipo. Un ejemplo de cubo que no cumple esta propiedad son las hojas, y uno que sí, el cristal.
- *isCrossSectional*: Define si el cubo, en vez de dibujarse como cubo, se dibuja en forma de cruz. Usado para la vegetación.
- *isDrawable*: Define si el cubo se puede dibujar. Podría haber cubos invisibles que cumplan determinadas funciones, aunque por ahora el único cubo existente que cumple esta propiedad es el aire.
- *isLiquid*: Define si el cubo se comporta como un líquido, permitiendo al jugador nadar y bucear en él, además de no tener una altura fija, sino variante en función de su nivel, que marca la cantidad de líquido en cada cubo. Por cada líquido, para cada nivel deseado deberá haber un cubo extra. Por ejemplo, en caso del agua, si queremos que pueda ir desde nivel siete (cubo lleno de agua) a nivel cero (solo una mínima cantidad), deberemos tener ocho cubos dedicados a ella. El nivel de cada uno, y el nivel máximo se podrán acceder con las propiedades *getLiquidLevel* y *getLiquidMaxLevel*.
- *getLightProduced*: Define la cantidad de luz producida por el cubo. En caso de cubos no luminosos, este valor será cero. Un cubo puede producir hasta 15 unidades de luz.
- *getUpTex*, *getLatTex*, *getDownTex*: Indica el identificador de la textura superior, lateral e inferior del cubo, respectivamente. Ese identificador de textura es el marcado por la clase *FileLoader* en la carga de cada imagen.
- *occludesNaturalLight*: Define si el cubo impide que pasen rayos de luz natural por él. Ello no impide la propagación de luz indirecta a través del mismo, con lo que al llegar a éste la luz natural comenzará gradualmente a disminuir.

Todas estas propiedades serán accedidas de forma estática mediante la clase *BlockLibrary*, aportando el identificador del cubo deseado.

2.2. Agrupaciones de cubos

Cargar de disco el mundo cubo a cubo o renderizar el mundo cubo a cubo no es una idea viable. Desde el punto de vista gráfico, toda llamada de dibujo a OpenGL gasta muchos ciclos, en los que la CPU debe sincronizarse con la GPU, con lo que hay que procurar minimizarlas. Desde el punto de vista lógico, cargar y guardar cada cubo de forma separada en disco conllevaría un gran gasto de tiempo abriendo o cerrando ficheros, y buscando índices concretos en los mismos. En Mínecraft, el título líder en este género, se llegan a tener unos 115.605.504 cubos diferentes en pantalla al mismo tiempo. Tomando una aproximación individualizada para cada cubo, esta cifra sería sencillamente inalcanzable. Por tanto, agrupamos los cubos en "grupos de cubos", llamados *chunks*. Ahora se dibujarán, cargarán y guardarán grupos grandes de cubos simultáneamente, minimizando las llamadas de dibujo a OpenGL y los cambios de estado de archivos en disco. Esta aproximación tiene desventajas, como una carga en pantalla menos suave (los chunks se cargarán en forma de grandes masas de cubos simultáneos, apreciables en la distancia) y un *culling* grafico[6] menos efectivo (dibujando cubo a cubo, podíamos analizar si éste estaba dentro del área de visualización o no antes de dibujarlo. Para un chunk, eso no es posible: Solo podremos aplicar *culling* cuando la totalidad del chunk esté fuera del área de dibujo. Si una mínima parte se encuentra dentro, deberemos dibujarlo entero). No obstante, estos pequeños inconvenientes son compensados con creces por las ventajas que esta estructuración aporta.

Para este proyecto, se ha observado que un tamaño de chunk de 32x32x32 cubos da buenos resultados, siendo este el usado en el prototipo desarrollado. No obstante, un tamaño de 16x16x16 también podría ser aceptable. El hecho de dividir el mundo en chunks hace la carga dinámica del mismo mucho más sencilla. Al cambiar el jugador de chunk, se descargarán los que sobrepasen esa distancia de *render* y se mandarán peticiones de carga para los chunks que acaban de acercarse a menos de esa distancia.

Un ejemplo gráfico de un solo chunk dibujado en pantalla puede verse en la figura 2. Se puede apreciar tierra, agua hasta el nivel del mar y, a partir del mismo, bloques de hierba, vegetación y un cúmulo de bloques formando un árbol.

[6] No ordenar dibujar polígonos no visibles desde el punto de vista actual, reduciendo la carga computacional de la GPU. Más información: Culling Explained. Crytek. [Citado el 18/06/2016] http://docs.cryengine.com/display/SDKDOC4/Culling+Explained

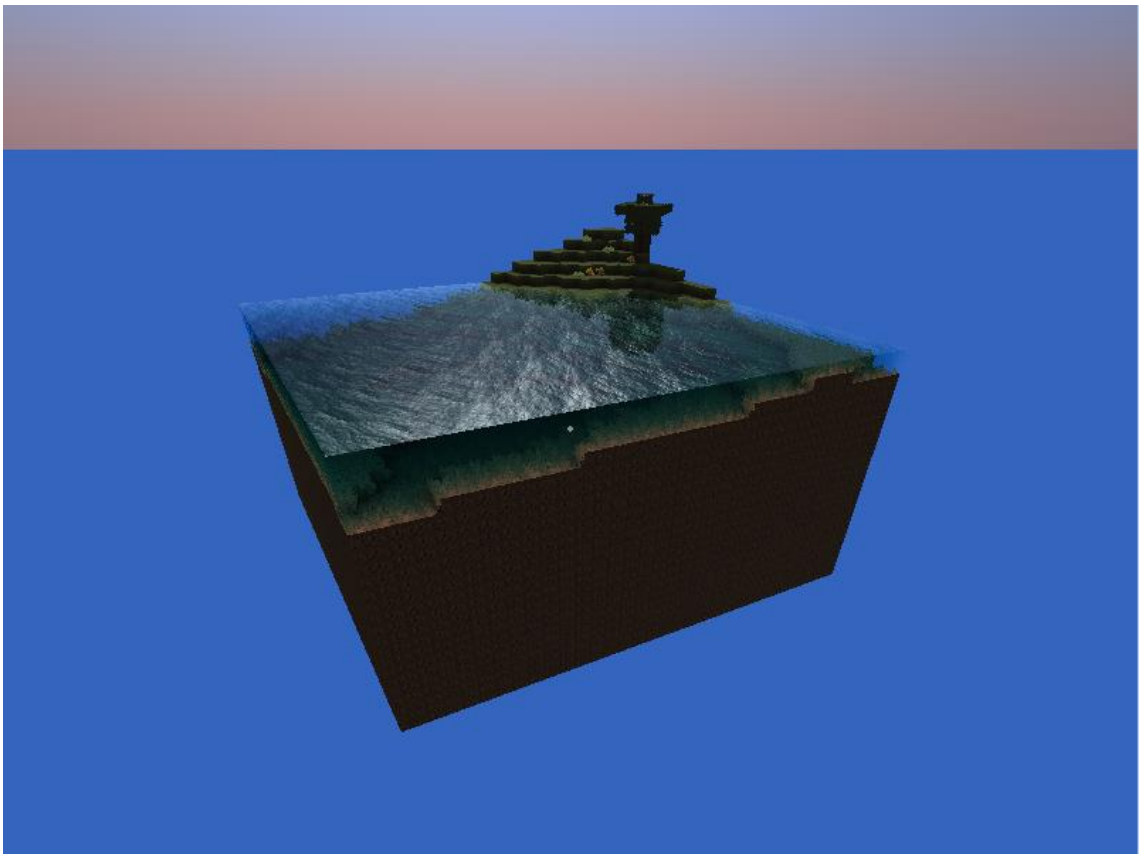


Figura 2 - Chunk de terreno en solitario

2.3. Generación procedural de mundos

En este proyecto nos basamos en la generación procedural de mundos infinitos, en contraposición con la carga de mapas ya existentes, siendo los cubos, o *vóxeles*, unos excelentes candidatos para la aplicación de estas técnicas dada su poca complejidad en contraposición a otros tipos de escenarios.

2.3.1. Modos de generación

Dada nuestra necesidad de crear mundos infinitos, la única posibilidad factible que tenemos es que estos sean creados en tiempo real por este propio sistema, en función de la posición del jugador en un momento dado. Por tanto, necesitamos algún método que nos garantice una generación realista, determinista y rápida para cualquier punto concreto del terreno. De todos los posibles métodos existentes, nos hemos decantado por el uso de ruido *Simplex*, una versión actualizada y más eficaz del ruido *Perlin*, más conocido. Se usa, en concreto, una implementación eficiente del mismo, disponible online[7]. A pesar de ser este algoritmo una optimización del citado ruido *Perlin*[8], comparte con el mismo las mismas bases.

Estos algoritmos requieren de una semilla inicial, usada para generar valores de ruido blanco pseudo aleatorios, que después interpolarán, y mezclarán en varias octavas, para

[7] A speed-improved simplex noise algorithm for 2D, 3D and 4D in Java. Stefan Gustavson [Citado el 18/04/2016] http://webstaff.itn.liu.se/~stegu/simplexnoise/SimplexNoise.java

[8] PerlinNoise [Citado el 18/04/2016] http://freespace.virgin.net/hugo.elias/models/m_perlin.html

obtener el resultado final para un punto solicitado. Partiendo de una misma semilla inicial, todo *sampleo* para un mismo punto nos dará, en todo momento, el mismo resultado, cumpliendo así la propiedad determinista requerida. Por tanto, cada punto del mundo se generará de forma idéntica independientemente del momento o condición del programa en el que sea requerida su carga, y todo mundo usando la misma semilla será idéntico. En nuestra implementación, cada instancia de ruido Simplex existente usará tres octavas, lo que nos proporcionará un terreno suficientemente detallado para este caso puntual.

El ruido Simplex puede ser aplicado tanto en dos dimensiones como en tres. La aplicación más común para generar terrenos usa el ruido en dos dimensiones para generar un *heightmap*, que, multiplicado por la altura máxima deseada (el ruido genera valores entre cero y uno) nos informará de hasta que altura llegan los bloques del terreno en cada punto (x,z) requerido. Este modo de generación es rápido y realista, pero no es capaz de generar cuevas, o acantilados. El tipo de cubo colocado variará también con la altura: El bloque más alto será de hierba si está a una altura baja, o de nieve si es alta. Los bloques inferiores a éste serán todos de tierra. Todo bloque por encima de la altura marcada será aire, salvo si se encuentra por debajo del nivel del mar, siendo agua. El ruido de tres dimensiones, por su parte, se usa como una función de densidad en cada punto (x,y,z) del terreno, en la que se considerará un resultado por debajo de un valor prefijado (por ejemplo, cero) como terreno sólido y un valor por encima como aire. Esto sí genera cuevas o acantilados en el terreno, pero es un orden de magnitud más lento que la anterior aproximación. En este proyecto se hacen uso de ambos modos dependiendo del tipo de mapa, siendo esto detallado en la siguiente sección.

En el caso de ruido 2D, a pesar de las propiedades favorables para la generación de terrenos que éste posee, *samplear* un solo ruido resultará en un terreno muy repetitivo, realista pero sin variaciones significativas. Por ello, hacemos uso de tres funciones de ruido diferentes:

- *mapBase*: Ruido suave y de variación lenta. Crea praderas suaves. Un ejemplo de mapa generado usando tan solo esta función puede verse en la figura 3a.
- *mapElevation*: Ruido de variación rápida. Crea montañas. Un ejemplo de mapa generado usando tan solo esta función puede verse en la figura 3b.
- *elevationCoef*: Ruido irregular de variación lenta. Especifica cómo de relevante es *mapElevation* en cada punto.

La forma de combinar estos ruidos tiene el formato:

$$mapBase + (mapElevation * elevationCoeff)$$

Con algunas variaciones. Esto generará un mapa más complejo, con praderas suaves en algunas zonas, y montañas escarpadas en otras (ver figura 4). Usamos esta técnica en el tipo de mapa *Islands* y en el tipo de mapa *SnowyMountains.*

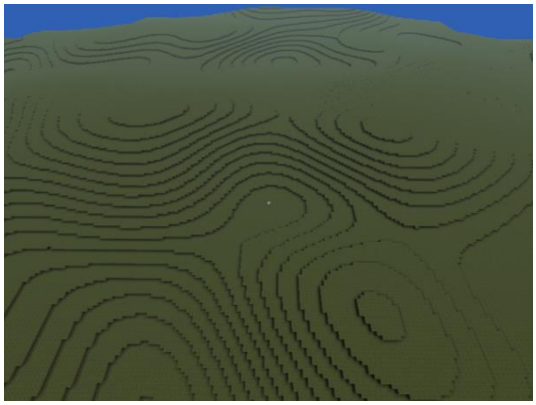


Figura 3a: Mapa generado usando la función *mapBase*

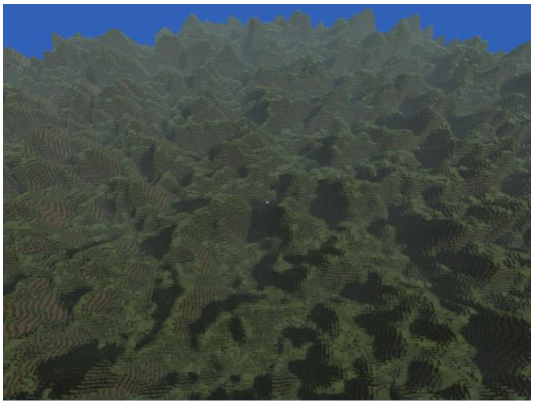


Figura 3b: Mapa generado usando la función *mapElevation*

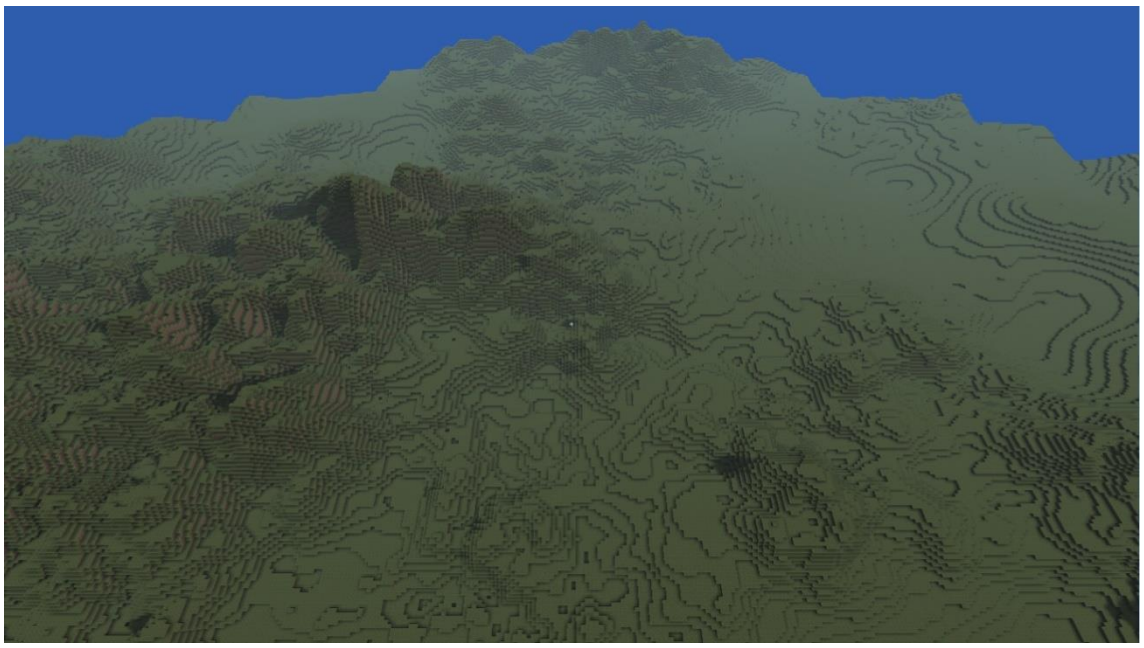


Figura 4 - Mapa generado combinando *mapBase* y *mapElevation* mediante *elevationCoeff*

Generamos, basándonos en lo explicado anteriormente, seis tipos diferentes de mapas:

- *Islands*: Genera islas interconectadas, con praderas y montañas. Un ejemplo de este mapa puede apreciarse en la figura 31 del Anexo 4.
- *Snowy Mountains*: Resultado de multiplicar por un índice mayor el valor del ruido con respecto a lo hecho en *Islands* al calcular la altitud de cada punto. Genera un mapa con variaciones de altitud más significativas, con montañas muy altas y extensas, y algunas praderas bajas con lagos. Un ejemplo gráfico se encuentra en la figura 32 del Anexo 4.
- *Plains*: Solo usa el ruido *mapBase*, así que solo contiene praderas, y océanos poco profundos. Ver figura 33 del Anexo 4.

- *Buggy caves*: Usa ruido 3D (sumándole a la densidad un valor exponencial en función de la altura, lo que lo hace generar menos terreno a medida que la altura sube) y ruido 2D para controlar la altitud de cada grupo de bloques. Genera un mapa de cuevas con algunos errores en el terreno intencionales, que le dan un buen aspecto. No genera agua. Una escena renderizada usando este mapa se puede apreciar en la figura 34 del Anexo 4.
- *Floating world*: Usa ruido 3D, sumándole a la densidad un valor que solo disminuye al estar cerca del suelo o al estar a una altura determinada (recordemos que, en nuestra implementación, una densidad menor que cero es terreno). Esto genera un mapa relativamente suave, con islas flotantes que arrojan sombras en el terreno. No genera agua. Dos imágenes de un mapa creado usando este parámetro se pueden ver en las figuras 35 y 36 del Anexo 4.
- *Underwater ruins*: Mediante operaciones con el ruido 3D, genera un entramado de cuevas. Tras ello, se aplica agua hasta cierta altura, con lo que se forman unas islas con unos océanos profundos. Se generan bloques de luz bajo el agua de forma aleatoria, con lo que el jugador podrá explorar las cuevas submarinas buceando. Una imagen mostrando la superficie de este mapa se puede encontrar en la figura 37 del Anexo 4. Una imagen subacuática del mismo se aprecia en la figura 38 del Anexo 4.

Para todos estos mapas, se hará una segunda fase de generación para cada chunk, para la que se esperará a que todos los chunks colindantes a él (sus chunks vecinos) se hayan añadido a escena. Esta segunda fase servirá para añadir árboles, generados también proceduralmente, o vegetación. La razón por la que se espera a una segunda generación es que, si algún vecino no está añadido aún y el árbol generado tenía (por ejemplo) una rama que pasara por ese chunk que aún no existe, el árbol no podría ser generado completamente, y aparecería cortado en el terreno. Tanto los árboles como la vegetación requerirán de un cubo de aire inmediatamente superior a un cubo de hierba, e iluminación natural suficiente. Un árbol aparecerá de media cada 200 cubos, vegetación verde cada 10 y plantas amarillas cada 50. La vegetación solo comenzará a ser generada a partir de un bloque por encima del nivel del mar.

La altura de los arboles será aleatoria, entre tres y nueve bloques. Con una probabilidad proporcional a su altura, el árbol podrá hacer crecer una rama en una dirección aleatoria, de longitud menor que dos tercios de la altura del árbol. Tras ello, el árbol se llenará de hojas.

2.3.2. Mundos infinitos

La posibilidad de la existencia de mundos infinitos genera otro problema: Al alejarnos del origen, comienzan a aparecer problemas con la precisión de los *floats*, lo que produce errores gráficos y lógicos (movimiento entrecortado, deformaciones en el terreno, fallos en el motor de colisiones, fallos en la generación de terreno), como se puede apreciar en las figuras 5a y 5b. La solución será cambiar todos los *floats* del jugador y del sistema de *Simplex Noise* por *doubles* (arreglando así el movimiento y las colisiones). No obstante, OpenGL no puede usar *doubles*, así que para arreglar los errores gráficos deberemos crear un modelo de

rendering basado en la posición del jugador como origen, manipulando manualmente cada vector traslación de cada matriz de modelo en función de la posición del jugador antes de poder enviarlas a la tarjeta gráfica. La matriz vista, por su parte, tendrá siempre una traslación igual a cero.

No obstante, el mundo no es totalmente infinito. A pesar de poder solventar el problema de los *floats*, el *overflow* de los *int* es inevitable llegado a algún punto. A pesar de la posibilidad de cambiar cada *int* del programa por un *long* (O por un *BigInteger* si se desea un mundo verdaderamente ilimitado), se ha considerado que una anchura de mapa de 4.294.967 km, con un área 36.162.995 veces mayor que la de la tierra, es suficiente en este caso concreto. Este límite puede apreciarse en la figura 6, que muestra a esa distancia un corte abrupto en el terreno, con una ausencia absoluta de fallos gráficos.

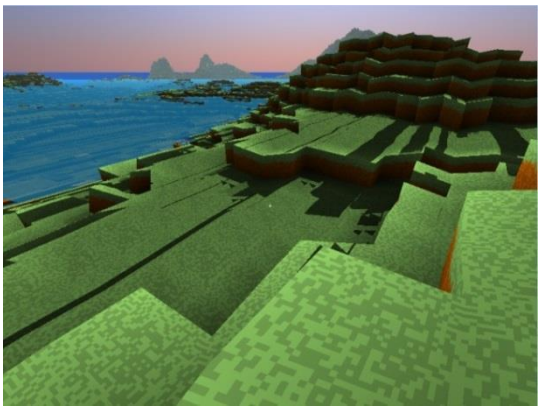


Figura 5a - Antigua distorsión gráfica a 10.000km del origen

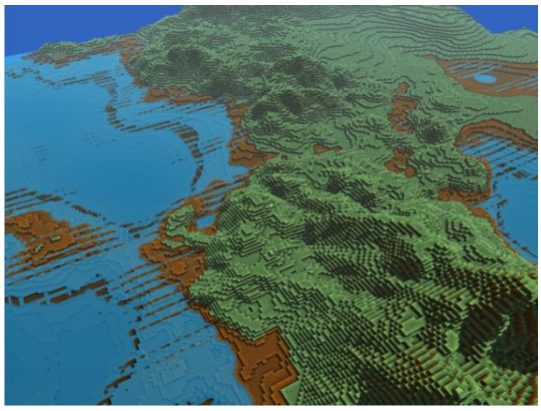


Figura 5b - Antigua distorsión gráfica a 100km del origen

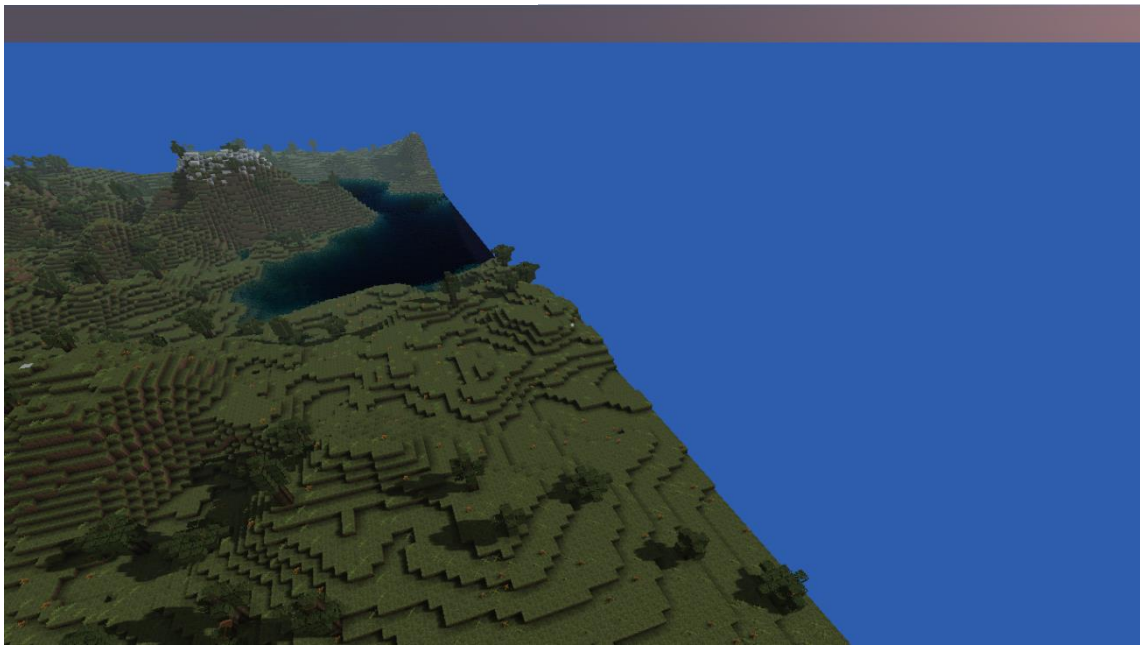


Figura 6 - Límite del mundo actualmente, a 2.147.484 km del origen

2.4. Estructura multihilo

En un motor gráfico en tiempo real como éste, una ejecución suave y sin bloqueos debido a cálculos complejos es imprescindible. Al estar creando la ilusión de movimiento en pantalla al cambiar numerosas veces por segundo la imagen que esta muestra, cualquier bloqueo inesperado, por corto que sea, hará que la imagen actual permanezca en pantalla más tiempo del normal, rompiendo la ilusión y causando una sensación desagradable en el usuario. Por tanto, todo el hilo principal encargado de calcular el movimiento del jugador, pintar la pantalla, etc. Debe ser lo más ligero posible.

Por ello, se han separado las dos tareas más computacionalmente costosas (generar los chunks mediante ruido y generar el buffer de vértices que debe ser enviado a la tarjeta gráfica cada vez que un chunk cambia) en dos hilos diferentes. Asimismo, se ha creado otro hilo aparte para las operaciones de Input / output ya que, a pesar de no ser estas tan computacionalmente caras como los dos casos anteriores, generan numerosos bloqueos indeseados al ser el disco duro mucho más lento que la memoria RAM. Se detalla en profundidad cada hilo a continuación:

- *ChunkGenerator*: El hilo principal guarda peticiones de añadido de chunks en un *stack*. Este hilo los inicializa, generándolos con ruido si no existen o cargándolos de disco si ya se encuentran guardados en el mismo. Tras ello, añadirá el chunk ya inicializado a una cola de espera del hilo principal, para ser añadido al almacén de chunks definitivo.
- *ChunkUpdater*: Para cada chunk que se le proporcione, en función de los contenidos del mismo generará un buffer de vértices en punto flotante correspondiente a la representación gráfica de ese chunk en un mundo 3D. Este buffer tan solo tendrá que ser enviado a la tarjeta gráfica por el hilo principal. Por defecto, generará primero los buffers de los chunks más cercanos al jugador, logrando así una carga de mapa centrada en éste.
- *ChunkStorer*: Se encarga de guardar chunks en disco, evitándole esa tarea al hilo principal, e interaccionando con la clase *FileManager*. A pesar de que computacionalmente no tendrá que hacer labores muy complejas (comprimir chunks y guardarlos), se mantendrá la mayor parte del tiempo esperando en bloqueos de disco, tarea que le debemos evitar al hilo principal.
- Hilo Principal: Se encarga de hacer todas las llamadas pertinentes a OpenGL, que solo pueden ser realizadas desde éste, como dibujar cada chunk en pantalla o enviar los buffers de vértices de los mismos, aportados por *ChunkUpdater*, a la tarjeta gráfica. Asimismo, gestiona el motor de físicas, eventos, el movimiento del jugador y genera peticiones de borrado o añadido de chunks.

Incluso en computadores modestos con tan solo uno o dos núcleos esta estructura es positiva, al permitir a los demás hilos seguirse ejecutando mientras alguno de ellos permanece en algún bloqueo. Las tareas en las que se han dividido exigen una mínima cantidad de sincronizaciones, al estar completamente separadas desde el punto de vista lógico.

2.5. Mundo dinámico

Lo que hace a este tipo de motores únicos es la posibilidad de modificar cualquier parte del mapa en tiempo real, en contraposición a los habituales proyectos existentes en los que el mapa es un ente completamente estático y pre generado. Esta es la razón por la que el hilo *ChunkUpdater* existe: Si el mundo jamás cambiara, podríamos directamente cargar ese buffer de vértices del disco, o similares. De este modo, no obstante, tenemos que recrear ese buffer cada vez que un cubo cambie en el chunk, para después sobrescribir el contenido guardado en el vbo[9] de la tarjeta gráfica por el nuevo recién generado.

Esto también hace posible la carga dinámica de mundo, al solo tener que preocuparnos por cargar los chunks más cercanos al jugador. Los más lejanos serán borrados, y para volver a cargarlos solo tendremos que tratarlos como un chunk que acaba de cambiar, y en función de sus cubos se generará un buffer de vértices de forma limpia y rápida.

A pesar de que el contenido de cada chunk puede cambiar en cualquier momento, seleccionamos como tipo de almacenamiento de memoria la propiedad GL_STATIC_DRAW, que asume que el contenido del buffer no va a cambiar, aportando una mayor velocidad de dibujo y una menor velocidad de actualización de contenido. La razón para ello se basa en que muchos chunks del mundo jamás serán modificados por el jugador y, los que los sean, lo serán solo a nuestra escala temporal (por ejemplo, una actualización por segundo) que, aunque para nosotros es un instante breve, para una computadora es una cantidad de tiempo realmente alta. Se recomienda que la actualización periódica de cubos (agua expandiéndose, TNT explotando, etc.) se realice con valor temporal suficientemente bajo como para que al jugador no le sea molesto, pero extremadamente alto desde el punto de vista de la computadora. Por defecto, en nuestro prototipo, este valor ha sido configurado a 0.3 segundos.

2.6. Motor físico

Se implementa un motor físico básico para el jugador, en el que se gestiona su movimiento, colisiones con el mundo, gravedad, etc. En concreto, se gestionan tan solo una serie de propiedades variables, siendo tras ello toda variación de posición el resultado de movimientos en torno a los tres ejes. Las propiedades son:

- *grounded*: Si el jugador está tocando tierra. Esto activa la capacidad de saltar, por ejemplo. Si esta variable es falsa, se incrementará la velocidad vertical, hasta colisionar en el eje Y con algún cubo, momento en que la velocidad vertical es puesta a cero y *grounded* pasa a ser verdadero.
- *flying*: Si el jugador está volando. El modo volar se activa por defecto pulsando la tecla SHIFT (ver Anexo 1), y se desactiva volviéndola a pulsar. Si el jugador está volando, su velocidad se incrementará significativamente y podrá saltar sin estar tocando tierra.
- *climbing*: Si el jugador está trepando. Esta propiedad se activa mientras se intente avanzar hacia un muro, y si la velocidad vertical es mayor que -1m/s. El jugador

[9] Buffer de OpenGL en el que se insertan los datos para cada vértice a dibujar. En nuestra implementación, cada chunk posee uno diferente.

comenzará a trepar por ese bloque, subiendo a 2/3 de su velocidad normal. La gravedad no afecta al personaje si *climbing* es verdadero.

- *underwater*: Si el jugador está bajo el agua. Bajo el agua, la velocidad desciende, la velocidad de caída se limita y se activa la posibilidad de saltar sin estar tocando tierra (nadar hacia arriba)

Considerando estas propiedades, el movimiento se calculará trigonométricamente en función del ángulo de visión en el plano XZ y una velocidad módulo, que será aumentada al volar y reducida al nadar. Si el jugador no está tocando tierra (*grounded*)**,** la gravedad comenzará a afectarle, agregándole una velocidad creciente en el eje Y en función de la gravedad a la que el mundo se encuentre configurada. Sabiendo el tiempo pasado desde el anterior *frame* de movimiento, podremos obtener la distancia total a recorrer. Se realizará el movimiento en cada eje de forma separada y por partes, comprobando la propiedad *isSolid* de cada cubo por el que vaya a pasar cualquier parte del cuerpo del jugador, teniendo en cuenta su tamaño y altura y deteniendo el movimiento en cuanto un obstáculo se encuentre, sea cual sea nuestra velocidad.

Esta posición x, y, z del jugador, sumándole la altura de los ojos del mismo, será la utilizada como centro del mundo, restándola manualmente a la traslación de las matrices modelo de los chunks a la hora de dibujar.

En el prototipo desarrollado contamos por defecto con un mundo de gravedad 15m/s² y un personaje de anchura 0.8m y altura 1.8m, con los ojos posicionados a una altura de 1.65m y una velocidad módulo de 5m/s. Estos valores son totalmente modulares, pudiendo ser alterados libremente en cualquier implementación que se desee realizar sobre este motor.

En cuanto al agua, su sistema físico de propagación se basa en unas reglas sencillas:

- Si el bloque inferior es aire, agua o vegetación, crear abajo un bloque de agua de nivel máximo.
- Si el bloque inferior es sólido, modificar los cubos a los lados de este, si no son sólidos, plantas o líquidos por un cubo de líquido con un nivel una unidad menor que el nivel del líquido del cubo actual. Si ese nivel fuera a ser menor que cero, no hacer nada.

Un ejemplo de este sistema de propagación puede apreciarse en la figura 7, que muestra agua fluyendo en una estructura de canales construida por el jugador.

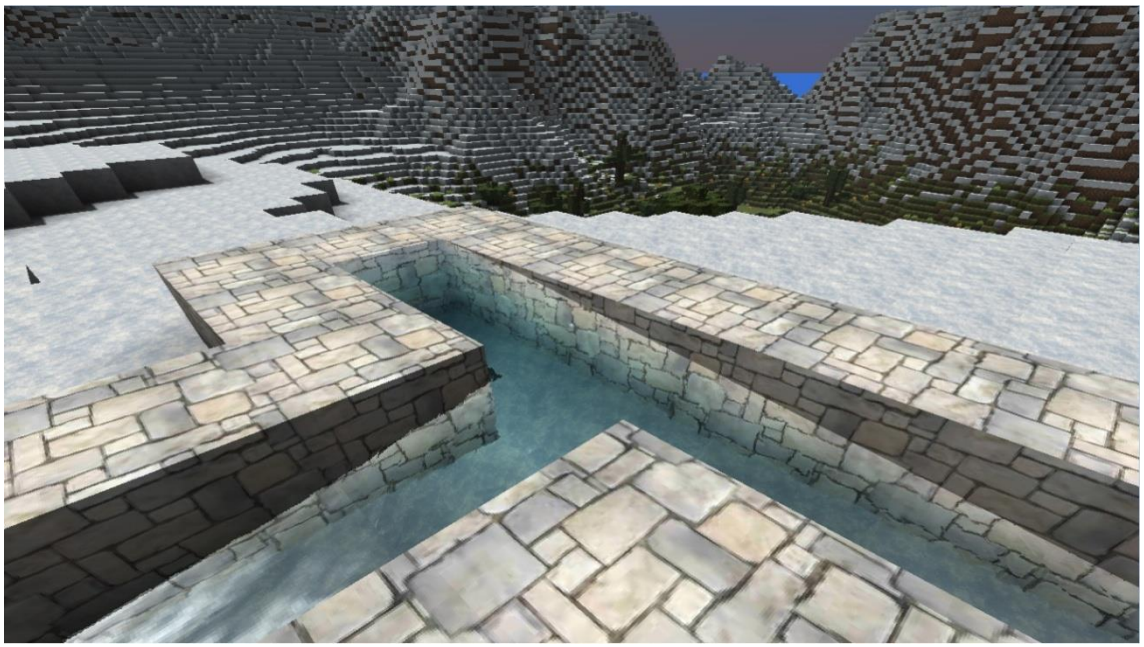


Figura7- Flujo de agua

2.7. Optimizaciones lógicas (mejora de eficiencia)

Se desea mantener la mayor cantidad posible de cubos en pantalla al mismo tiempo. Además, se desea que la carga de mundo sea todo lo rápida que sea posible, y que no haya ninguna clase de bloqueos durante la ejecución de este motor. En el caso de los bloqueos, aunque pueda parecer que nos hemos librado de la gran mayoría gracias a la estructura multihilo, queda un gran problema por solucionar. Java es un lenguaje que cuenta con un recolector de basura. En aplicaciones de otro tipo, la ejecución del mismo no es apreciable, pero en aplicaciones en tiempo real, éste para brevemente la ejecución del hilo principal a su paso el suficiente tiempo como para que sea percibido por el jugador, resultando en una experiencia molesta. Como no poseemos control directo sobre el recolector de basura ni podemos indicarle en que momentos o no pasar, no nos queda otra opción que intentar hacer que su ejecución sea lo más ligera posible, evitando interferir con el normal funcionamiento del programa. Para ello, la única opción que tenemos es tratar de minimizar, en todo momento, la cantidad de objetos a ser destruidos. Para ello, usaremos *Pooling*.

El *Pooling*[10]es un patrón de diseño que permite reciclar objetos ya inútiles, para ser reutilizados más tarde. La mayor cantidad de memoria destruida es causada por el borrado y la creación automática de chunks cada vez que el jugador se mueve: Cada chunk contiene dos arrays de 32x32x32 bytes, uno para almacenar los cubos y otro para almacenar la luz de los mismos, sumando ambos un total de 64KB de memoria. La opción más lógica, por tanto, será gestionar la creación y destrucción de estas listas mediante una Pool, lo que permitirá que la memoria que antes usaba un chunk sea reutilizada por otro creado posteriormente. Esto soluciona dos problemas: Elimina una enorme cantidad de trabajo del recolector de basura y

[10] Game Programming Patterns: Object Pool [Citado el 18/06/2016] http://gameprogrammingpatterns.com/object-pool.html

acelera la creación de chunks, dado que no es ya necesario reservar dinámicamente una gran cantidad de memoria por cada nueva inicialización. La desventaja que este método posee es que al reciclar un array es posible que éste no esté inicializado a cero, pudiendo causar problemas en arquitecturas que den por hecho este estado inicial, o un gasto de tiempo en recorrer cada array para reinicializarlo. Por ello esta *Pool*, llamada *ByteArrayPool*, almacenará una lista de arrays "sucios" y una lista de arrays a cero, devolviendo en cada caso la más necesaria para cada ocasión. Además, el hilo con menos trabajo (*ChunkGenerator*) se encargará de llenar arrays de ceros en su tiempo libre.

Esta aproximación será tomada también para los buffers de *floats,* usados para guardar los vértices a ser dibujados para cada chunk antes de ser enviados a la tarjeta gráfica, y para buffers de bytes, usados para cargar imágenes.

Otro gran problema es el alto gasto de memoria que tal cantidad de chunks provocan, sobre todo considerando que la memoria asignada a java suele ser relativamente baja en comparación a la memoria total del computador. En concreto, en un mundo con una distancia de render de, por ejemplo, 10 (Siendo la usada en Minecraft), la cantidad de chunks en pantalla al mismo tiempo es de 3528, dando un total de gasto de memoria, solo en arrays de luz y cubos, de 220'5MB. En otro lenguaje de programación con acceso a toda la memoria RAM del computador, esta cifra sería aceptable. Sin embargo, en el caso de Java, la memoria que se asigna es limitada. Debemos, por tanto, intentar comprimir de alguna forma el contenido de los chunks existentes, sin empeorar excesivamente el tiempo de computación. Para ello, nos aprovechamos de que hay una gran cantidad de chunks formados exclusivamente por un tipo de cubo: Aire o tierra. De todos estos, el predominante es el aire, ya que muchos mundos no suelen sobrepasar los cuatro chunks de altura, y la altura máxima del mundo, es de, por defecto, ocho chunks. Es decir, que la mitad del mapa está compuesta por arrays vacíos.

Abstraeremos por tanto estos arrays, encapsulándolos en una clase genérica. Cuando detectemos que todo el chunk está formado por el mismo cubo, o que toda la luz del chunk es idéntica, reciclaremos ese array y lo cambiaremos por un solo byte constante, que será el valor devuelto cuando se intente obtener cualquier cubo o luz del chunk. Con esto, reduciremos el consumo de memoria a, aproximadamente, un cuarto de la usada anteriormente.

2.8. Gestión de datos en disco

Aunque dedicar un fichero en disco para cada chunk es la opción más sencilla existente, se trata de una práctica muy poco eficiente. Como ya se ha descrito anteriormente, usando una distancia de render razonable como 10 chunks tendremos 3528 chunks al mismo tiempo en pantalla. Sólo éstos, considerando que el jugador no se ha movido en ningún momento y son los únicos generados, conllevarían ya la creación de 3528 ficheros. Cuando el jugador comenzara a moverse y llegara al punto en el que se deben descargar chunks lejanos y generar nuevos cercanos, sería necesario escribir un total de 168 ficheros nuevos, y cargar otros tantos. Esta es una práctica realmente problemática, por muchas razones.

- Cuanto más alto es el número de ficheros en una carpeta, más costoso es encontrar un fichero en particular en la misma. Si guardáramos el identificador de cada chunk en el nombre de su fichero correspondiente, deberíamos dedicar una gran cantidad de tiempo para tan solo descubrir si el mismo existe ya en disco, o no.
- El mero hecho de abrir un fichero para leerlo o escribirlo conlleva múltiples llamadas computacionalmente caras al sistema operativo. Aunque la apertura de un solo fichero particular en un momento puntual puede no parecer extremadamente costosa, la apertura de (como se ha detallado antes) 168 ficheros para guardar archivos lo más rápido posible conllevaría una gran cantidad de bloqueos y tiempo desaprovechado.

Ambas razones hacen a esta práctica inviable, más aun al tratarse de un programa en tiempo real.

Está claro, por tanto, que el problema radica en el número de archivos. Al igual que las razones que nos llevaron a agrupar cubos en chunks, éstas nos deberán llevar a crear agrupaciones grandes de chunks en ficheros especializados. Esta aproximación, llamada ficheros de región[11], es la que han tomado muchos juegos de este estilo.

Un fichero de región agrupa en su interior grupos grandes de chunks. En concreto, en nuestra implementación se ha observado que un tamaño de 8x8x8 chunks ofrece buenos resultados. Necesitaremos para esto alguna forma de escribir y leer en partes no secuenciales del archivo, tareas para las cuales la clase *RandomAccessFile* nos ofrecerá todas las utilidades que necesitemos.

Sería problemático, no obstante, volver a tomar en este momento la aproximación más intuitiva, creando grandes ficheros de región vacíos y asignar, mediante alguna fórmula, una posición en el archivo a cada chunk que requiera ser escrito en el mismo. Esta aproximación, teniendo en cuenta que cada chunk posee un total de 32KB de datos de cubos, generaría ficheros de región con un tamaño, cada uno, de 16MB, estando la mayor parte vacíos. Un mapa normal, sin alejarnos en exceso del origen, podría ocupar fácilmente unos 200Mb. Esto es completamente inadmisible.

Una buena solución pasará por no añadir ningún chunk al fichero hasta que no sea necesario. Cuando lo sea, se añadirá al final del mismo. Aplicando técnicas de compresión como *RunlengthEncoding*[12] y ZLIB (existente por defecto en las librerías de Java) será posible reducir el tamaño de cada chunk significativamente. De hecho, siendo el mapa tan homogéneo, una compresión como *RunlengthEncoding* reducirá el tamaño de cada chunk, de media, a una veinteava parte del original. Tras aplicar estas compresiones, el tamaño final de los datos comprimidos, a usar a la hora de leerlos, no es ya posible de calcular. Por tanto, dedicaremos tres bytes antes de comenzar a escribir los datos comprimidos para almacenar el tamaño en bytes de datos que el chunk ocupa en disco, y el bit más significativo para

[11]Creating a Region File System for a Voxel Game. Benjamin Arnold [Citado el 18/06/2016] https://www.seedofandromeda.com/blogs/1-creating-a-region-file-system-for-a-voxel-game

[12]Run-length encoding. Wikipedia [Citado el 19/06/2016] https://es.wikipedia.org/wiki/Run-length_encoding

comprobar si el chunk ya ha pasado la segunda generación (uno) o no (cero), con objeto de aplicársela al cargarlo si las condiciones son correctas (todos los vecinos se han añadido).

Como los chunks ya no tienen un tamaño fijo y pueden ser añadidos en cualquier lugar del fichero, comportándose ya éste como una especie de lista dinámica, dividiremos el mismo en sectores de tamaño fijo, de forma que si un chunk se expande o reduce tenga cierto margen hasta que haya que reescribir el fichero entero por falta de espacio (si un chunk comprimido ocupaba 500b y al reescribirlo ocupa 600b, y el sector ocupa 512b, estos datos ya no cabrán en un sector, sino en dos. No obstante, el siguiente sector puede estar ya en uso, con lo que la solución será reescribir el archivo, ampliando el tamaño de este chunk a dos sectores). Usamos por tanto en esta implementación un tamaño de sector de 1024 bytes, cifra que ofrece una buena relación entre espacio malgastado y número de reescrituras de archivo.

Además, necesitaremos alguna clase de índice al principio del fichero para poder localizar la posición de cada chunk. Reservaremos para ello cuatro bytes por cada posible chunk (un total de 2048 bytes) al principio del fichero. El primer byte marcará el tamaño en sectores que el chunk comprimido ocupa, y los últimos tres bytes, el sector de comienzo del mismo. Un tamaño de chunk igual a cero significará que el éste no existe aún en el fichero. Además, con objeto de aumentar aún más la compresión, si un chunk está formado enteramente por un tipo de cubo (por ejemplo, solo aire), se guardará con un tamaño de -1 en este índice, usando el último byte reservado para la posición para indicar, en vez, que código de cubo es el repetido en la totalidad del chunk. De esta forma, los chunks vacíos, o los llenos de materiales homogéneos, no ocuparán ni un solo sector de espacio en el fichero, quedando comprimidos dentro del propio índice.

Por último, hemos aplicado técnicas para acelerar aún más el proceso de Input / Output. Guardamos una lista bidimensional de ficheros de región abiertos, con un tamaño igual al máximo de ficheros de región que pueden ser abarcados por la distancia de *render* actual. Así minimizaremos la apertura y cierre de ficheros, ocurriendo solo en algunos puntos al movernos por el mapa. Asimismo, para cada fichero de región abierto cachearemos su índice entero, guardándolo en RAM. Cada lectura del índice se podrá realizar ahora en la RAM, a costa de un mayor gasto de memoria, pero dividiendo las lecturas de ficheros a la mitad en el peor caso (chunk existe en el fichero) y a cero en el mejor (chunk no existe aún en el fichero, o está formado en su totalidad por el mismo cubo). Esto nos permitirá obtener velocidades de carga de mundo desde disco más altas que la propia velocidad de generación procedural de ese mundo usando funciones de ruido.

Todos estos archivos se guardarán dentro de una carpeta cuyo nombre será igual al nombre que se le ha dado al mapa creado, con el nombre ***f_<x>_<y>_<z>.kxw***. Cada chunk comprobará a que fichero de sectores pertenece aplicando, la formula

$$(x, y, z) = (floor(chunkx/8), floor(chunky/8), floor(chunkz/8))$$

Por su parte, los datos generales para el mundo (semilla del mundo, posición del jugador, momento del día, tipo de mapa, etc.) se guardarán en un archivo estándar de texto llamado settings.txt

Un ejemplo de esta técnica de almacenamiento en disco puede apreciarse a continuación en la figura 8, en la que suponemos un tamaño de región de dos chunks para que la explicación resulte más simple. El chunk cero indica, con su tamaño -1, que es uniforme, y no ocupara un sector en el archivo. Como vemos, en su bit de ubicación menos significativo hay un dos, con lo que será un chunk con todos los cubos de id=2. En caso del chunk uno, vemos que ocupa un sector de espacio y está en el sector 0x00 00 00. Al posicionarnos en ese sector y combinar los bytes ocho, nueve y diez tendremos, en el bit más significativo, si los árboles y vegetación del chunk han sido ya generados, y en los demás bits la longitud en bytes del chunk en disco. Leemos esos bytes comenzando desde el byte 11 y descomprimimos lo obtenido usando ZLIB y, tras ello, RLE. Tendremos ya los bytes del chunk, que podremos insertar en un array tridimensional.

0	1	2	3	4	5	6	7
-1	0	0	2	1	0	0	0
8	9	10	11	...		1035	
Tamaño en bytes de chunk 1			Datos chunk 1 (Comprimidos con RLE y ZLIB)				

Figura 8 - Ejemplo de chunks guardados con este método, suponiendo un tamaño de sector de 2 chunks.

3. RENDERIZACIÓN

Se detallan a continuación todas las consideraciones tomadas en el desarrollo de la parte gráfica del motor, ejecutadas sobre la GPU de nuestro computador. Los algoritmos detallados a continuación exigen una tarjeta gráfica moderadamente potente y con una memoria gráfica elevada. Sin embargo se han incluido en el menú opciones para desactivar alguna de estas características gráficas con objeto de lograr una ejecución correcta en ordenadores más modestos.

Existen en la parte gráfica también ventajas a la hora de desarrollar nuestro propio motor gráfico para este género en contraposición a usar uno ya existente. Al saber que todos los cubos miden lo mismo, están posicionados secuencialmente y están alineados con los ejes, no nos será necesario indicar para cada vértice enviado a la GPU que coordenada de que textura cada cubo usa, al poder deducirlas simplemente con la posición del mismo. Además, podremos precalcular iluminación dinámica, como se detalla más adelante, o renderizar agua en múltiples alturas y direcciones de visualización, situaciones no existentes en otros géneros y para las que, por lo tanto, los motores gráficos genéricos no están diseñados.

3.1. Iluminación

Un sistema de iluminación completo debe contener tanto la iluminación causada por el sol, creando ciclos de día y noche, como la iluminación causada por cada uno de los posibles cubos de luz de la escena, causando luz artificial. Esta iluminación será computada a nivel de *shader*, manteniendo en cada punto la que más predomine. Mientras que la iluminación artificial será constante, la luz natural nocturna brillará a un nivel del 15% de la luz diurna.

No existe un nivel mínimo de luz, con lo que en las cuevas más profundas, donde ni tan siquiera un atisbo de luz indirecta solar llegue, todo será negro.

3.1.1. Ciclos de día y noche

Se buscaba generar un cielo diurno relativamente fotorrealista actualizable en tiempo real y de generación relativamente sencilla, basándonos tan solo en un vector de dirección de vista. Se ha decidido, por cumplir todos esos puntos, implementar el paper "A Practical Analytic Model for Daylight", de A. J. Preetham [1]. En concreto, se ha extendido y trasladado a Java la implementación del mismo propuesta en [2], con algunas diferencias.

A pesar de la recomendación del autor de usar un tetraedro o un cubo para plasmar el cielo, en nuestro caso este cambia en tiempo real. Teniendo ello en cuenta, y aprovechando el uso de *DeferredShading*[13] podemos renderizarlo sin el uso de ninguna clase de geometría extra, shader especializado o llamada de dibujo. Para ello, comprobaremos la profundidad de la escena en cada pixel. Si esta es igual a uno, asumiremos que no hay nada tapando el cielo y lo dibujaremos aportándole al método la dirección de visualización en coordenadas de mundo, precalculada según el modelo presentado por Crytek [16].

[13] Técnica de render detallada en el punto 3.4

La turbiedad del cielo ha sido elegida mediante prueba y error, con un valor final de 2'7 en nuestra implementación. Asimismo, consideramos una latitud, longitud y día del año constante en toda la ejecución, con objeto de evitar ángulos poco estéticos del sol a la hora de arrojar sombras. Este día, en recuerdo del día en el que este algoritmo fue implementado, será el 25 de Agosto del 2015. En cuanto a la latitud y longitud, también constante, se ha elegido una latitud de 0 y una longitud, al no ser relevante a excepción de para el cálculo de la hora del día, exactamente igual a la ubicación de un punto de mi ciudad, -1.630753. Un ejemplo del color del cielo al atardecer puede ser encontrado en la figura 9a.

El cielo nocturno es, por su parte, una imagen circular obtenida de Internet, que se samplea trigonométricamente a partir de la dirección de la visión. Consideraremos una dirección de visión dirigida por debajo del horizonte como no cielo, y pintaremos el pixel asociado del color de fondo del mundo, un azul dependiente de la luz del día. El color nocturno será atenuado en función del momento del día en que nos encontremos, comenzando a aparecer gradualmente en un *zenith*[14] igual a 1.1, aumentando en intensidad hasta el *zenith* 1.94, en el que alcanzará su máximo. Un ejemplo del color del cielo a medianoche puede apreciarse en la figura 9b.

El color que este algoritmo arroja del cielo en horarios nocturnos es incorrecto. Por tanto, marcaremos el color de día como puramente negro a partir del *zenith* 1.94, mismo momento en el que el cielo nocturno alcanza su máxima intensidad. En otras latitudes o días del año esta cifra podría no ser lo suficientemente baja para que estos errores no aparecieran, siendo esta la razón por la que es doblemente importante conservar el día del año, latitud y longitud completamente iguales.

El color final del cielo será igual a la suma del color nocturno y el color de día (la luz es aditiva).

Se creará además otra variable, llamada *daylightAmount*, que reducirá suavemente su valor al aproximarse el anochecer, pasando de un valor de uno al mediodía a un valor de 0.45 tras la puesta de sol. Este valor será multiplicado a todos los cubos a los que se vaya a aplicar iluminación natural, haciendo la luz nocturna notablemente más tenue que la diurna (considerando que, además de este oscurecimiento, de noche todo el mundo está en sombra, lo que lo oscurece aún más).

[14] Medición usada en astronomía para simbolizar la altura del sol. Con valor 0, el sol se encuentra en la cúspide del cielo. Con valor pi/2 el sol se encuentra a la altura del horizonte, simbolizando todo valor mayor que ese un sol aún más bajo y, por tanto, la noche.

Figura 9a - Cielo diurno

Figura 9b - Cielo nocturno

3.1.2. Propagación de luz dinámica en un entorno *voxel*

Prácticamente todos los proyectos 3D existentes tienen un problema con la cantidad de luces en la escena, ya que a mayor cantidad de luces, mayor es la complejidad de los cálculos en la tarjeta gráfica. Aunque el uso de *DeferredShading* disminuye este coste, sigue siendo significativo.

En este motor gráfico, sin embargo, todo cubo puede ser una fuente de luz en un momento dado. Incluso aun creando los mapas sin fuentes de luz, nada impide al jugador poner las que considere oportuno. No podemos, por tanto, tomar una aproximación en la que la cantidad de luces puedan ralentizar el juego.

La solución a este problema es precalcular la luz en cada cubo de cada chunk, y guardarla en otro array de bytes. Cada vértice subido a la tarjeta gráfica tendrá por tanto un valor precalculado de luz artificial y de luz natural, resultante de interpolar la luz de todos los bloques colindantes al mismo, con lo que el trabajo de calcular la iluminación ya estará hecho de antemano. Como deseamos dos tipos de iluminación, natural (para luz propagada por el sol) y artificial (luz propagada por bloques), pero no queremos gastar más de un byte por bloque, deberemos compactarlas. Por tanto, la luz natural ocupará los primeros cuatro bits del byte y la artificial los cuatro últimos. Esto nos proporcionará una intensidad de luz comprendida entre los valores 15 y cero. Este valor entero será traducido a un valor de luz normalizado entre cero y uno de forma linear, dividiéndolo entre 15. A pesar de que la luz decae exponencialmente, en este caso concreto una disminución lineal de la intensidad da también resultados aceptables, al tardar las luces más distancia en comenzar a perder intensidad, perdiéndola más rápidamente al final.

La luz será precalculada mediante un autómata celular. Por cada bloque, toda luz se extenderá a los bloques colindantes mientras estos no sean opacos, perdiendo un nivel de intensidad. Por tanto, la máxima distancia que un cubo de luz podrá iluminar será de 15 bloques, suponiendo una intensidad inicial también máxima, de 15. La luz natural tendrá una propiedad extra además de la anterior: Mientras la intensidad sea máxima (15) y el bloque inferior no sea opaco ni impida el paso de rayos de sol (*occludesNaturalLight*), la luz podrá propagarse hacia abajo sin perder intensidad, simulando los rayos de sol. Todas las

propagaciones / borrados de luz existentes en el proyecto son versiones optimizadas de estos dos principios.

Cuando un chunk es inicializado, su valor de luz es cero para todos los cubos. El chunk analizará la luz de los bloques colindantes a él desde cada chunk vecino en cada dirección, y extenderá la luz desde ellos a él de esta forma. Tras ello, analizará si en su interior existe algún bloque fuente de luz, extendiéndola dentro de él mismo y potencialmente a algún chunk vecino.

En el interior de los *shaders*, se comprobará la luz artificial (constante) y la luz natural (afectada por sombras y por luminosidad del día, *daylightAmount*). La iluminación elegida será la que tenga mayor valor de ambas, razón por las que las luces en una casa, por ejemplo, no se harán aparentes hasta que se haga de noche.

Se muestra en la figura 10 un ejemplo de este sistema de propagación de luz. Podemos ver que, dada la corta longitud de los muros, algo de luz indirecta logra "escapar" por las esquinas, con mucha menor intensidad. El color de fondo no es completamente negro al encontrarnos en el exterior, y ser la luz nocturna levemente luminosa. En caso de habernos encontrado dentro de una cueva, por ejemplo, la única luz existente sería la arrojada por nuestro cubo de luz.

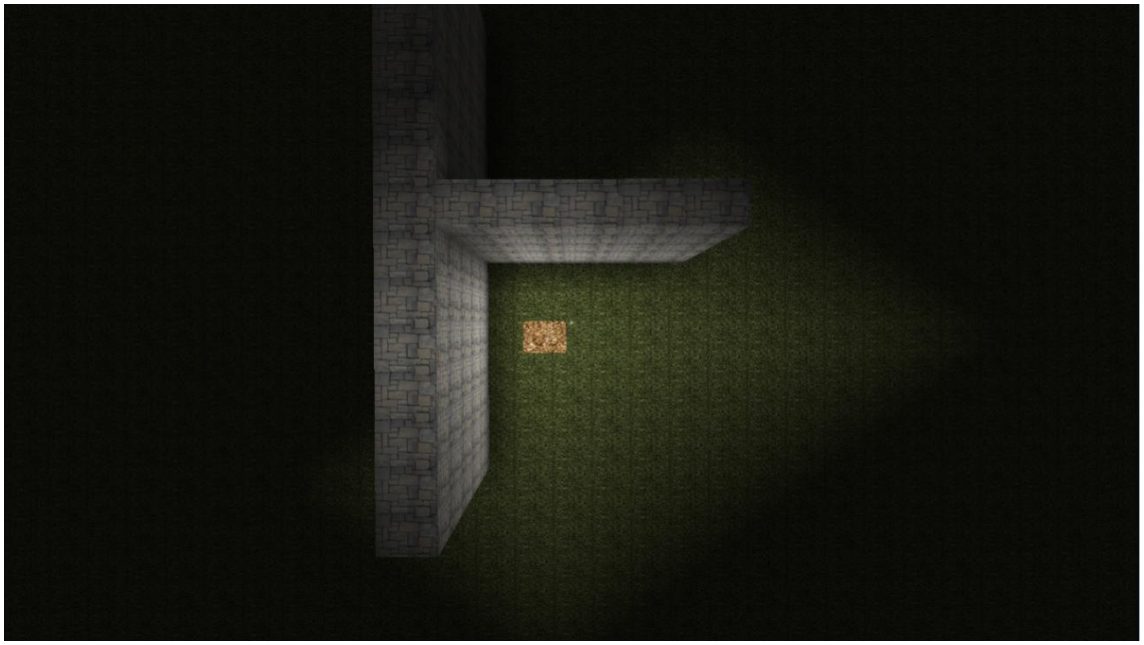


Figura10 - Propagación de la luz usando un autómata celular, en una construcción con muros.

3.2. Sombras

Para calcular sombras, la técnica más popular existente es el Shadow Mapping[15]. Se basa en redibujar la escena desde el punto de vista de la luz deseada (con matrices de perspectiva para luces puntuales y con matrices ortogonales para luces direccionales, como el sol), guardando solo el *depth buffer* en una textura aparte. Al dibujar la escena de forma normal, se transformará cada pixel al espacio basado en el punto de vista de esa luz de nuevo, y se comprobará ahí su profundidad. Si esa profundidad es igual que la profundidad guardada para ese punto en la textura de sombras, el pixel estará iluminado por esa luz. Si, sin embargo, la profundidad es mayor que la guardada, el pixel estará detrás de algún otro objeto tapándole la luz, resultando sombreado.

Esta técnica posee un gran problema, entre otros. Al estar guardando toda la información sobre las sombras de toda la escena en una textura de resolución limitada, múltiples pixeles del terreno caerán, a la hora de la transformación espacial, en el mismo pixel de la textura de sombras, causando una resolución extremadamente baja (tan baja, de hecho, que resulta inaplicable). Si intentamos lograr sombras de alta resolución, deberemos reducir el área de la matriz de proyección de la luz. Haciendo que esta matriz solo abarque un área cercana al jugador, lograremos unas sombras bien definidas durante unos metros. Mirando más lejos las sombras simplemente desaparecerán, al no tener información de luz a la que acceder al haber comprimido el área de la matriz de proyección de la luz a una zona pequeña, y no a toda la escena. Esta desaparición de sombras, aun pareciendo una mala práctica, ha sido (y continúa siendo) la aplicada en muchos motores gráficos.

Se ha implementado uno de los algoritmos de sombras más usados a nivel profesional en los actuales motores gráficos modernos. Se descartaba su uso hasta hace relativamente poco tiempo por la complejidad de su implementación y su alto coste computacional y en memoria, ya que hace necesario redibujar las sombras de la escena en varias ocasiones, así como reservar una gran cantidad de memoria gráfica para texturas de profundidad. A cambio, contaremos con sombras detalladas en todo el mapa, sea cual sea su distancia, opción imposible usando algoritmos de sombras estándar. Su nombre es *Cascaded Shadow Mapping* [10], [11].

Un ejemplo del aspecto de las sombras en la escena puede apreciarse en la figura 11. Se pueden apreciar sombras de alta calidad en la proximidad a la cámara, donde las hojas del árbol cercano permiten pasar algunos rayos de sol. A lo lejos se puede ver que los árboles proyectan también sombras detalladas, logrando sombrear con éxito y alta calidad a una escena entera, lo que no es posible usando *Shadow Mapping* simplemente.

Una explicación detallada del desarrollo de este algoritmo se encuentra en el Anexo 3.

[15] Tutorial 16: Shadow Mapping. [Citado el 18/06/2016]
http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

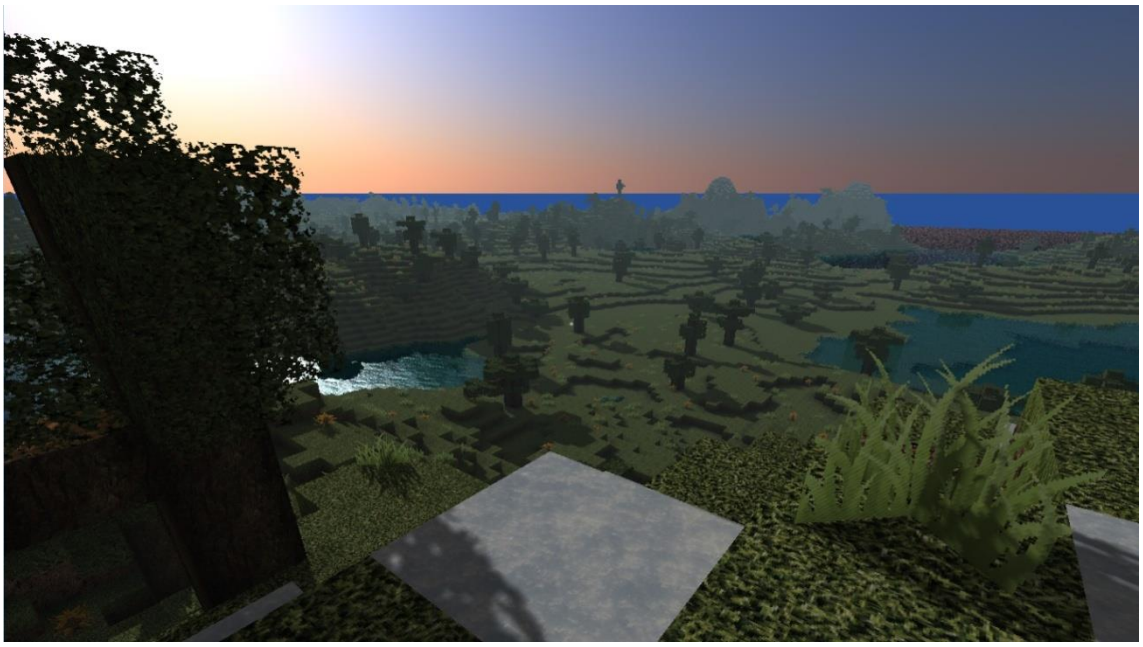


Figura 11 - Sombras al anochecer

3.3. Agua fotorrealista

Renderizar agua fotorrealista en tiempo real es una tarea extremadamente compleja, que no ha podido ser acontecida hasta hace relativamente poco tiempo. Aun así, por su complejidad y, ante todo, por estar trabajando con un sistema basado en matrices de proyección a pantalla en vez de con un trazador de rayos, los métodos que se suelen proponer, aunque simulan los fenómenos y características de este líquido, no lo hacen de forma realista. De hecho, al contrario que en muchas áreas, aún no existe un estándar para enfrentarnos a este desafío, causando que el agua varíe radicalmente entre proyectos, tanto en métodos como en calidad y eficiencia.

En concreto, en los motores gráficos basados en cubos existentes en el mercado no existe prácticamente ningún ejemplo que intente generar un agua relativamente realista, basándose muchos en meras aproximaciones de la misma o, en caso del título Minecraft, en una capa azul semi-transparente. La razón para este descuido de la estética del medio líquido en este tipo de motores se basa en tres inconvenientes que complican incluso más la ya compleja tarea del renderizado de agua foto realista: El agua no está siempre en la misma altura (se pueden poner cubos de agua por encima del nivel del mar), el agua puede ser vista desde más direcciones que verticalmente (en el caso de un cubo de agua suspendido en el aire, o en una ladera de la montaña) y se pueden dar casos en los que, en un pixel de visión, un rayo entre y salga de cubos de agua secuenciales en repetidas ocasiones, haciendo casi imposible calcular la distancia recorrida bajo el agua del rayo de luz.

El método presentado a continuación es el resultado de la lectura de numerosos *papers* diferentes sobre el tema, a la par que la aplicación de algunas técnicas poco usadas para este propósito con objeto de intentar enfrentar los tres problemas que un motor de cubos añade a esta área.

Los seres humanos distinguimos agua de forma instintiva. No obstante ésta es un material con unas propiedades muy sutiles, que bajo algunas circunstancias causarían, de carecer nosotros de esa percepción afinada hacia la misma, que no pudiéramos llegar a apreciarla. Tomemos por ejemplo un vaso lleno de agua, transparente y calmada. La única diferencia con respecto a un vaso vacío es la refracción que esta causa sobre los colores del fondo. En caso de ser el fondo de un color homogéneo, podremos seguir distinguiéndola, al estar el ojo preparado para detectar cualquier mínima variación que esta cause (por ejemplo, la tensión superficial). Esto no ocurre con prácticamente ningún otro material existente. Es por ello que, aunque para (por ejemplo) renderizar arcilla nos bastaría con cubrir sus propiedades más básicas y el jugador la reconocería como tal al instante, para el caso del agua deberemos intentar mimetizar la mayor parte de sus propiedades si queremos causar esa misma respuesta en la mente de los que observen nuestro proyecto en funcionamiento.

Entre las propiedades del agua, estas seis son las más notables:

- Reflexión
- Refracción
- Extinción / *Scattering* de la luz en función de la distancia
- Perturbaciones (oleaje)
- Reflejos de luz
- *Fresnel*

En la práctica, podemos no cumplir una de estas propiedades sin que existan problemas. Cumplir solo cuatro de ellas o menos, no obstante, romperá la ilusión de realismo.

Tras implementar todas estas propiedades en nuestros bloques de agua, obtendremos imágenes como las ilustradas en las figuras 12 y 13. Una explicación detallada de la implementación de cada propiedad del agua puede verse en el Anexo 2.

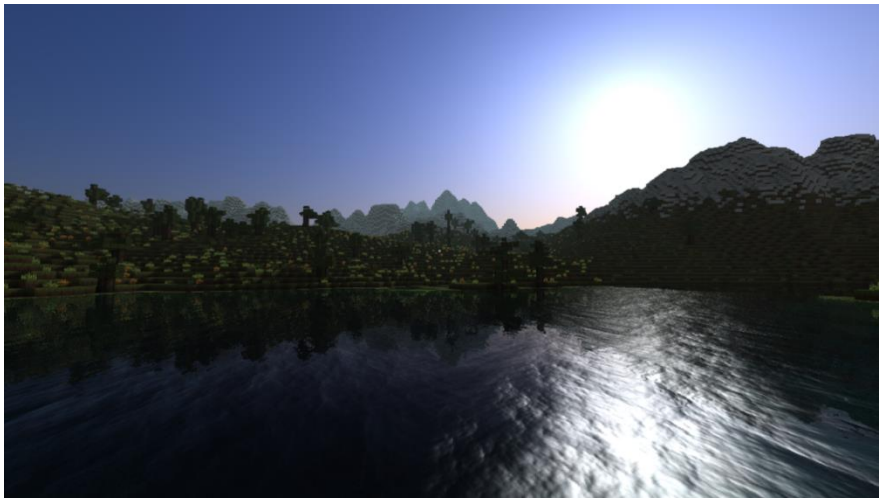


Figura 12- Agua final, producto de combinar todas estas propiedades

Figura 13 - Visión subacuática

3.4. *Deferred Shading*

El *Deferred Shading* [13] es un paradigma de programación gráfica de popularidad creciente orientado a GPUs de alta gama[16], que cambia el orden natural de dibujo en pantalla. Por defecto, sin usar *Deferred Shading*, se suelen agrupar todas las funcionalidades de dibujo en un *shader* haciendo que dibuje los pertinentes polígonos en pantalla, aplicándoles las operaciones necesarias. Para *shaders* básicos esto no produce problema alguno, pero en *shaders* muy complejos computacionalmente se va a perder mucho tiempo calculando el color de pixeles que serán después ocultos por otros polígonos para los que habrá también que calcular su color. En una escena compleja este escenario puede ocurrir numerosas veces por pixel, en claro detrimento de nuestra eficiencia gráfica al estar la GPU derrochando tiempo de cálculo en pixeles que jamás van a ser vistos.

Deferred Shading aboga por usar un *shader* simple para calcular la imagen producida por los polígonos y después aplicar las operaciones gráficas complejas sobre esa imagen producida por el primer *shader*. Así, garantizaremos que cada pixel sobre el que esas operaciones van a ser aplicadas es el pixel final de la escena, que no va a ser ocultado por ningún otro pixel más cercano a la cámara. No obstante, al trabajar sólo sobre una imagen a color no tendremos toda la información que teníamos en el primer *shader* (la posición de cada pixel en el mundo, su profundidad, su normal, etc.). El principal inconveniente de esta técnica es que, para poder acceder a ellas, deberemos guardarlas también en texturas aparte, consumiendo una gran cantidad de memoria gráfica.

[16]Forward Rendering vs. Deferred Rendering. Brent Owens. [Citado el 18/06/2016] http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

En este proyecto se usa un *Deferred Shading* de tres pasadas, necesitando tres órdenes de dibujo sucesivas para obtener la imagen final a dibujar en pantalla. En concreto, se distribuye de esta forma:

- Primera pasada: Muchos *shaders* simples generan texturas con datos a ser usados en pasadas posteriores, como podemos ver en la figura 14. En concreto:
    - Se generan las texturas de profundidad (de 16 bits) para cada *shadow map.*
    - Se generan las texturas de profundidad (de 16 bits) para cada capa de agua por cada pixel, generando también una textura que albergue la normal de la primera capa de agua encontrada (rgb).
    - Se genera la textura de color inicial (rgb) tras dibujar todos los polígonos. Contendrá solo el color de cada cubo en cada punto, sin sombras ni iluminación de ningún tipo. Se genera también una textura extra para guardar la profundidad de cada pixel en pantalla (el *depth buffer,* 16 bits), a partir de la cual la posición completa puede ser recuperada sin necesidad de guardarla en otra textura, como explica el paper de Crytek [16]. Se genera asimismo otra textura para guardar tanto la iluminación (en los canales rg) como la normal de cada pixel (en los canales ba, comprimida guardando solo dos de sus componentes[17], podremos reconstruirla al saber que la normal de los pixeles dibujados siempre apunta a pantalla al estar *GL_CULL_FACE*[18] activado).

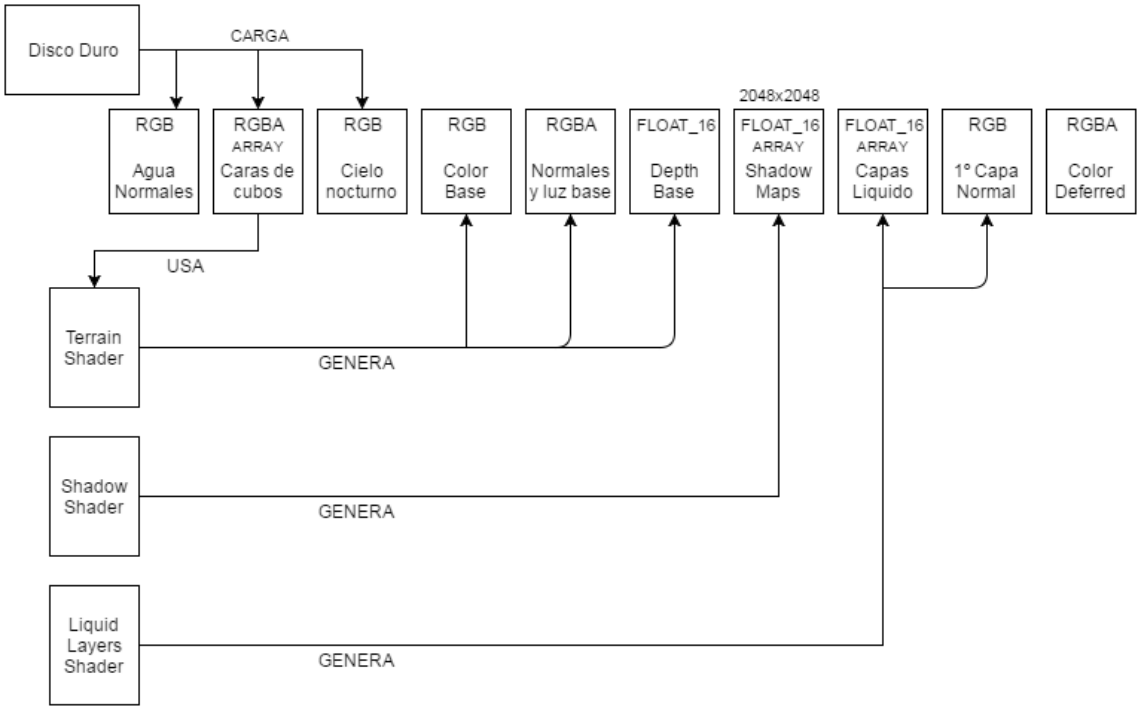


Figura 14 - Interacción de los shaders de primera pasada con las texturas guardadas en memoria gráfica

[17] Compact Normal Storage for Small G-buffers [Citado el 20/04/2016] http://aras-p.info/texts/CompactNormalStorage.html

[18] Parámetro de OpenGL que, al activarse, dibuja solo los triángulos cuya normal apunta hacia la dirección de visión. Reduce la carga gráfica en un 50% ya que, normalmente, los demás triángulos se trataban de caras tapadas de objetos, que no iban a poder ser vistos de igual manera. Es un modo de *culling*.

- Segunda pasada: Aplica sombras sobre la textura de color inicial y las capas de agua existentes, si las hay. Además, calcula la absorción / *scattering* en cada pixel que posea capas de agua intermedias. Esta nueva textura de color generada se guardará en una imagen rgba, guardando en el canal alfa información importante (si el pixel está o no bajo el agua y que cantidad de iluminación especular se le aplica en caso de estarlo). El uso de texturas en esta pasada por parte del *shader* puede verse en la figura 15 a continuación.

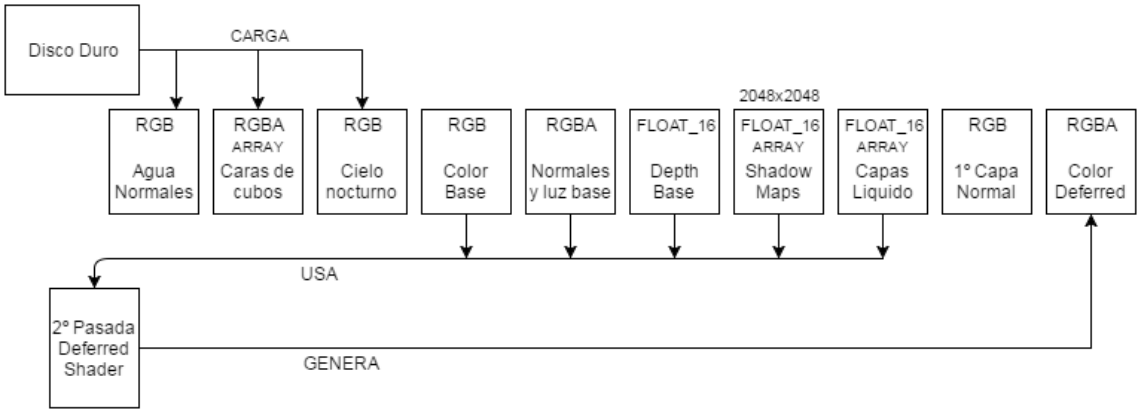


Figura 15 - Interacción de los shaders de segunda pasada con las texturas guardadas en memoria gráfica

- Tercera pasada: En esta pasada se dibujará ya directamente en pantalla.
    - En caso de no estar bajo el agua, calcula reflejos (usando *Screen Space Ray Marching*), *fresnel*, refracción e iluminación especular sobre las superficies de agua, si las hay. Asimismo, genera el cielo usando el algoritmo de A.J. Preetham mencionado anteriormente en todos los pixeles que no sean de terreno, así como en los reflejos que no colisionen con ningún dato en pantalla. Mostramos este caso en la figura 16.
    - En caso de estar bajo el agua, calcula tanto la refracción como el *fresnel* de la superficie.

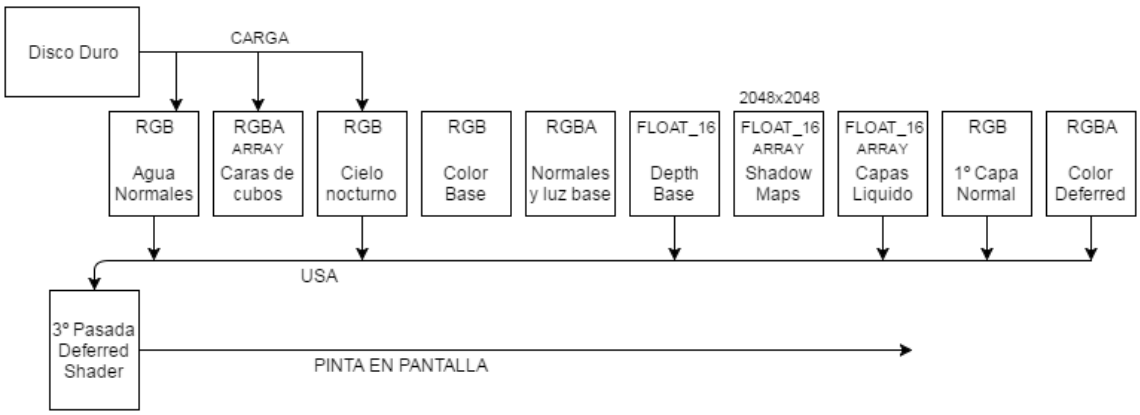


Figura16 - Interacción de los shaders de tercera pasada con las texturas guardadas en memoria gráfica

3.5. Otras mejoras gráficas

Además de todas las técnicas gráficas complejas implementadas, se detallan aquí otras que, pese a ser mucho más simples, mejoran también significativamente el aspecto de la escena.

3.5.1. *Mip Mapping*

El *Mip Mapping* [12] crea, para cada textura, subtexturas de la misma en menor resolución, hasta llegar a una textura 1x1. OpenGL aplicará una subtextura u otra (o una mezcla de varias) en función de la distancia existente hasta la textura a renderizar. Esto evita ruido y *aliasing* en la distancia, añadiendo un *antialiasing* de forma "natural", sin necesidad de aplicar supermuestreo[19], al emborronar ya la imagen de antemano.

El problema con el Mip Mapping es que, en caso de usar atlas de imágenes (almacenar varias imágenes en una grande, aproximación ampliamente utilizada en motores voxel de este tipo, agrupando todos los voxeles en una imagen grande y pintando uno u otro en función de la circunstancia) y alejarnos, se comenzarán a ver líneas entre cubo y cubo. Esto es causado porque, al calcular los mipmaps, openGL está mezclando el color de unos cubos con el de otros. No existe una solución limpia a este problema, con lo que ha sido necesario pasar cada textura de cubo a una imagen propia, y cargarlas todas usando un *TextureArray2D*. Esto implica que para arquitecturas más antiguas, que no soporten esta estructura de datos, este proyecto no funcionará.

OpenGL permite asignar a cada textura que filtro aplicarle en la distancia y en la cercanía. No nos es un inconveniente mantener un estilo ligeramente retro, al ser este un motor de cubos. En la cercanía, por tanto, filtraremos usando GL_NEAREST, con el que se distinguirá cada pixel de la textura al acercarnos a un cubo (la alternativa, GL_LINEAR, lo emborronaría). En la distancia el filtro será GL_NEAREST_MIPMAP_LINEAR. La parte MIPMAP_LINEAR indicará a OpenGL fundir varias subtexturas de la imagen en función de la distancia, en vez de cambiar de una a otra directamente, lo que causa artefactos visuales. Esta opción, evidentemente, será más cara computacionalmente.

3.5.2. *Anisotropic Filtering*

El filtrado anisotrópico[20]es la versión avanzada del *Mip Mapping*. En *Mip Mapping*, al ver imágenes desde ángulos oblicuos, estas aparecen emborronadas debido a la disminución de la calidad en la frecuencia vertical, mientras que la horizontal se mantiene igual. En un filtrado anisotrópico las subtexturas generadas se reescalan para cada eje en función de diversos grados de inclinación, causando un mayor gasto en memoria gráfica pero una mayor calidad de imagen. Dado el gasto extra que esta mejora supone, se permite activarla o desactivarla desde el menú principal.

[19] Técnicas de Antialiasing [Citado el 20/04/2016]
http://acacia.ual.es/profesor/LIRIBARNE/AIG/antialiasing/tecnicas.html

[20] Filtrado Anisotrópico. Wikipedia. [Citado el 18/06/2016]
https://es.wikipedia.org/wiki/Filtrado_anisotr%C3%B3pico

Configuraremos OpenGL para que, si este filtrado es activado, use la máxima calidad del mismo que la tarjeta gráfica soporte, marcado por la variable de entorno GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT.

3.5.3. *Atmospheric Scattering*

En la vida real, los objetos en la distancia parecen azulados. Esto se ocasiona por el Scattering de Rayleigh [17]. A pesar de que computacionalmente implementar algoritmos que usen este principio es demasiado costoso, se puede aplicar una ligera niebla, exponencial en la distancia, de un color azulado. En esta implementación concreta se usará un color de niebla azulado-blanquecino de color (0.6,0.74,0.8), que se irá oscureciendo suavemente con la llegada del anochecer, alcanzando por la noche un valor oscuro de (0.06,0.074,0.08) .

En concreto, con objeto de dar a esta aproximación un aspecto lo más agradable posible (asumiendo su inexactitud) se aplicará la siguiente fórmula:

$$fog = e^{-0.00001*distance^2}\,;fog \geq 0.2\,;fog \leq 1$$

$$finalColor = mix(fogColor, originalColor, fog)$$

Cortamos el valor *fog* a 0.2 para que ninguna parte del terreno, esté lo lejos que esté, sea nunca completamente tapada. Basándonos en esta ecuación, a una distancia de aproximadamente 263 metros la niebla tendrá la misma relevancia en la mezcla que el color original, aumentando exponencialmente a partir de ahí, hasta alcanzar el valor 0.2 a los 401 metros.

3.5.4. *Ambient Occlusion*

La oclusión ambiental[21] es una técnica gráfica sutil que añade una gran mejora visual a la escena. Se basa en el principio de que a las esquinas llega menos luz que a otros puntos de los muros, al tener esta menos área disponible desde la que proceder. Esta es una propiedad que los humanos no solemos apreciar a simple vista, pero que nuestra vista extraña si no está, notando un gran aumento de realismo al comparar imágenes con y sin esta mejora activada (ver Anexo 4, figuras 27, 28, 29, 30).

Se aplicará obteniendo para cada vértice de la escena la luz de sus cuatro cubos colindantes en el plano en el que este vértice se encuentre, y dividiendo la luz acumulada entre cuatro. En casos normales se conseguirá luz dinámica, mientras que en las esquinas, al tener los cubos sólidos un valor de luz igual a cero, se conseguirá oclusión ambiental de una forma sencilla.

[21] Ambient Occlusion for Minecraft-like worlds. M. Lysenko. [Citado el 18/06/2016] https://0fps.net/2013/07/03/ambient-occlusion-for-minecraft-like-worlds/

3.6. Optimizaciones gráficas (mejora de eficiencia)

Con objeto de poder dibujar el juego en pantalla tantas veces por segundo como sea posible, se deben incluir algunas optimizaciones gráficas a nivel de cubos, para reducir lo más posible la cantidad de geometría a renderizar.

- Renderizar caras de cubos sólo si no son adyacentes a otro cubo: Si un cubo está rodeado de cubos no transparentes, no nos sirve de nada dibujarlo porque jamás va a poder ser visto, al estar siendo tapado por los otros cubos. Podemos reducir este planteamiento a nivel de caras de cubos: Si una cara de un cubo es adyacente a otro cubo solido no transparente, ni esta cara ni la cara adyacente van a poder verse jamás, así que evitando enviarlas a la tarjeta gráfica ahorraremos un gran espacio en memoria gráfica y aceleraremos enormemente el tiempo de dibujo de la escena. Esta técnica es llamada *culling*.

  En concreto, para cubos no líquidos, solo se dibujará una cara de un cubo si su cubo adyacente es semitransparente o transparente, o si, en caso de que sea transparente, no cumple la propiedad *isPartnerGrouped***,** que activaba este *culling* de forma forzosa. Para cubos líquidos, solo se dibujará la cara si el cubo colindante es transparente o semitransparente, y además este no es un líquido. De esta forma, se dibujarán caras de cubos mirando hacia el agua, pero no se dibujarán capas de agua inútiles pegadas a esos cubos, que a fin de cuentas no van a suponer una variación de distancia recorrida por el rayo de luz.

- Usar un *VBO* por chunk: Cada chunk posee su propio *VertexBufferObject* [14], que actualiza contando cuantos triángulos líquidos y sólidos posee, dibujando uno u otro según se especifique. Este *VBO* será borrado si se detecta que el chunk no dibuja nada en pantalla, creándose de nuevo cuando esto si ocurra. Esto puede ocurrir tanto debido al añadido de un cubo nuevo como debido al borrado de algún cubo en algún chunk colindante que haya desactivado el culling en algún cubo de este. Hay que considerar que los chunks subterráneos no dibujan nada a pesar de estar completamente llenos de cubos, al estar estos pegando con otros cubos y aplicársele culling a todos. Asimismo, si se detecta que el chunk no tiene nada que dibujar, la llamada de *glDrawArrays* ni siquiera se producirá.

- *Frustrum culling* a nivel de chunk: Cada vez que un chunk va a ser dibujado, se pasa su punto central a coordenadas de pantalla y se comprueba si alguna parte de su esfera envolvente se encuentra dentro de alguna parte del *frustrum* de visualización. Asimismo, se comprueba también si el chunk se encuentra tras la cámara, comprobando su coordenada z y el radio de su esfera envolvente. Si cualquiera de esas condiciones se cumplen, el chunk no necesitará ser dibujado en este *frame* concreto.

- Compresión de la normal de cada vértice en los datos del shader: Cada vértice subido a la tarjeta gráfica requerirá ciertos datos en punto flotante: La posición x, la posición y,

la posición z, la iluminación artificial, la iluminación natural, la textura a usar y la normal del vértice en cuestión sumando, en total, nueve datos por vértice. Esa cantidad es muy grande, pero podemos aprovechar que un cubo solo tiene seis normales posibles para comprimir estos datos. Además, sabemos que la textura a usar por el cubo va a ser forzadamente un valor entre cero y 255, con lo que la mayor parte de la capacidad del punto flotante usado para contenerla va a ser desaprovechada.

Como sabemos que solo hay seis normales, podemos distinguirlas con un número de cero a cinco y descomprimirlas en el shader. Por ejemplo, consideramos que cero es igual a la normal (1,0,0), que uno es igual a la normal (-1,0,0) , etc. Viendo pues que las normales se comprimen en un número de cero a cinco y que las texturas se limitan en un número de cero a 255, podemos comprimir esos dos datos juntos (para separarlos luego vía shader) con la fórmula:

$$índice\_normal * 1000 + índice\_textura$$

Usando así sólo un total de seis puntos flotantes por vértice a subir a la GPU.

4. CONCLUSIONES

Se ha implementado con éxito un motor gráfico y lógico completo para un mundo basado en cubos, cumpliendo todos nuestros propósitos marcados. Se ha superado ampliamente el estándar gráfico en este género, mostrando que un género de videojuegos basado en cubos no es un impedimento para desarrollar gráficos avanzados a todos los niveles posibles. Asimismo, podemos comprobar que Java es perfectamente capaz de ser usado para desarrollar aplicaciones gráficas en tiempo real, pese a tener implementado de base un colector de basuras. Es cierto que el colector de basuras causa, de no tomar las debidas precauciones, pequeños bloqueos que en aplicaciones de tiempo real son notables y molestos. La solución para esto, como se ha visto, es gestionar nosotros mediante *Pooling* la mayor cantidad de memoria que podamos, dejándole al colector de basuras un trabajo menor, lo que hace su ejecución absolutamente imperceptible.

Podemos ver las comparaciones gráficas entre nuestro motor y el videojuego líder en este género, Minecraft, en el Anexo 4, figuras 39 a 44, contrastando nuestros gráficos detallados con su aspecto *retro*, en el que se han basado la mayor parte de títulos de este género. Esperamos que este motor demuestre que esa estética no es la única posible en el género, a pesar de encontrarse esta tan extendida.

Entre todos los algoritmos desarrollados, destacamos:

- Algoritmo de Cascaded Shadow Mapping: Un algoritmo realmente avanzado, muy difícil de conseguir hacer funcionar correctamente y no implementado por ningún otro motor de cubos que hayamos podido encontrar, en los que las sombras, si existen, simplemente desaparecen al alejarse.
- Algoritmo de rendering del agua: El agua implementada es el resultado de numerosas lecturas de artículos científicos y de una implementación totalmente personal, sin basarse en ningún algoritmo preexistente. Mención especial requieren los reflejos en superficies acuáticas en entornos de cubos usando *Screen Space Ray Marching*, desmintiendo las afirmaciones existentes en múltiples lugares de la red que exponían que sencillamente no era posible realizar reflejos de agua a tantas alturas en tiempo real. Es cierto que este sistema tiene sus limitaciones (destacando el claro error visual en las esquinas del agua al inclinar la pantalla hacia abajo), pero dados los buenos resultados obtenidos en condiciones normales, y dadas las limitaciones de este sistema, se considera asumible.

De todos los problemas encontrados en la implementación, el más destacable de todos ha sido, paradójicamente, la propia dificultad de *debug* que OpenGL posee. Al no permitirse devolver datos a la CPU no es posible imprimir valores críticos en la consola, siendo la única opción posible imprimir esos datos en forma de colores por pantalla, a pesar de que la información obtenida de ese modo es extremadamente vaga. Incontables horas de tediosa solución de errores han sido causadas por este hecho, errores que, de otro modo, podrían haber sido solventados en muy poco tiempo. Mención especial requieren, de nuevo, los errores producidos por los algoritmos de sombras y el agua durante la implementación, siendo estos también, con diferencia, los más difíciles de depurar. Del tiempo de implementación de los mismos, de hecho, un alto porcentaje ha sido empleado en intentar encontrar la razón de

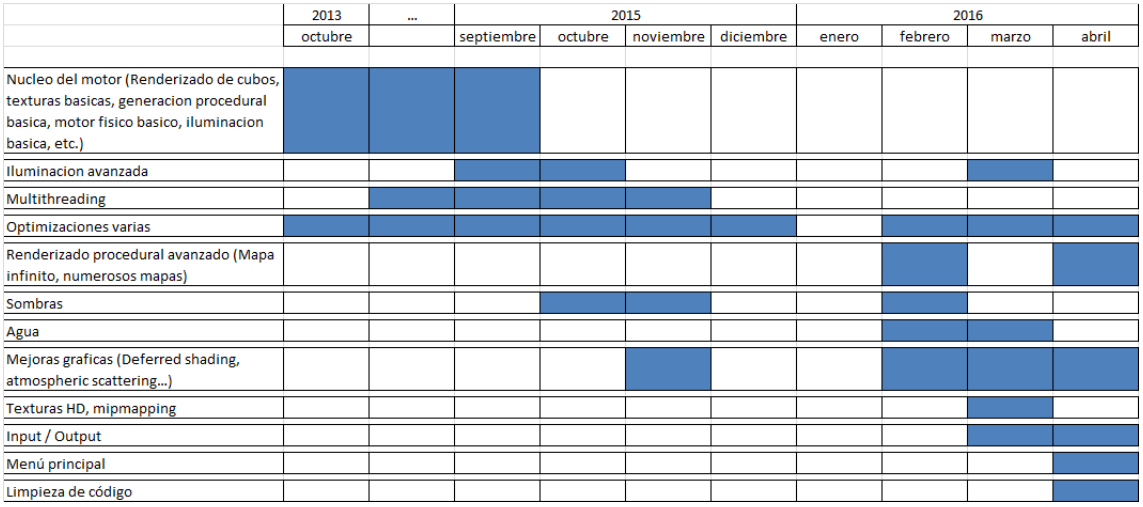
diversos errores y su depuración. Esta puede ser una de las razones por las que la mayor parte de los programadores son reticentes a implementar motores gráficos.

4.1. Cronograma

Al comenzar a implementar este proyecto a finales de 2013 con objeto de aprender 3D y no haber tenido en mente en esa época que este podría crecer tanto, llegando a ser de hecho mi proyecto en el futuro, comencé esta implementación sin seguir ninguna clase de estructuración o diseño de código. De hecho, cuando comencé a implementar la parte de insertar y eliminar cubos, descubrí que esa tarea era imposible dada mi estructura de código del momento. Una gran cantidad del tiempo de este proyecto ha sido utilizado, por tanto, en reescribir y reestructurar código ya elaborado previamente con objeto de hacerlo suficientemente extensible.

Sumando el tiempo empleado en aprender 3D desde cero, las horas empleadas en reestructurar el código y mi poco o nulo conocimiento de esta área de la informática en 2013, además del tiempo empleado en debuguear errores en OpenGL y al gran alcance de este proyecto, podemos asumir una cantidad de horas de trabajo en este motor, desde sus orígenes, superior a 1000.

Se ilustra en el siguiente cronograma la división del trabajo desde que se aceptó este motor como proyecto. Anteriormente a ello, este fue desarrollado en momentos de tiempo libre y de forma muy inconstante, con lo que no se detallan.



4.2. Posibles ampliaciones

En proyectos de este tipo es siempre posible añadir más características. Ilustramos, para cada área del mismo, unos ejemplos:

- Mejora en el render de agua: Se podría considerar el añadido de un *blur* gaussiano a la superficie del agua, suavizando la imagen como ocurre en la vida real, aunque esto

añadiría una pasada extra de *Deferred Shading*. Se podrían implementar también cáusticas o brillos superficiales para los fondos marinos.

- Mejora en la generación de mundos: Se podría extender la generación de mundos para generar diferentes tipos de mapas en función de la posición en el mapa, en lugar de seleccionarlos vía menú. Mediante el uso de datos *long* se podrían generar mundos con límites de tamaño mucho mayores que los actuales, o incluso lograr mapas infinitos con el uso de clases sin límite de tamaño como *BigInteger*.
- Mejora en el motor físico: Podría extenderse el motor físico para que afectara también a los bloques, pudiendo añadirles propiedades físicas, o que fueran afectados por explosiones o la gravedad.
- Mejora en el rendering general: Se podría añadir *glare* al mirar al sol, niebla en el horizonte, partículas para las explosiones o viento que afectara a la vegetación, creando movimiento en las hojas.

La ampliación más evidente, no obstante, es crear usando este motor un juego más avanzado que el prototipo presentado, al tratar el actual solo de construcción de estructuras y exploración. En esta ampliación, no obstante, las posibles ideas y posibilidades son infinitas, y escapan al alcance de este proyecto.

4.3. Opinión personal

Desarrollar un motor gráfico propio desde cero es una tarea compleja, larga y plagada innumerables horas de *debug*. Implementar un motor gráfico no es implementar un videojuego, pudiendo ahora afirmar rotundamente y con conocimiento de causa a cualquiera que me manifieste su deseo de desarrollar uno, la ya mencionada frase de *"write games, not engines"*.

A pesar de que la implementación de un motor gráfico diste mucho de la programación de un videojuego, no está exenta de encanto. Este desarrollo me ha permitido profundizar con creces en mi conocimiento del área gráfica de la informática aplicada a tiempo real. Ahora sé cómo funcionan la mayor parte de las características gráficas de los juegos actuales, y puedo pararme a apreciarlas, en lugar de ignorarlas como estaba haciendo hasta este momento. Gracias a la investigación para este proyecto he aprendido mucho sobre la física de la luz en diversos medios, sobre la programación eficiente y avanzada de OpenGL y sobre el funcionamiento de una GPU.

Y, por último, así como Internet me ha enseñado y ayudado ampliamente a conseguir esta meta, espero honestamente que, gracias al código abierto, este proyecto pueda llegar a ayudar también a alguien algún día.

5. BIBLIOGRAFÍA

1. Preetham, A. J., Shirley, P., & Smits, B. (1999, July). A practical analytic model for daylight. *In Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (pp. 91-100). ACM Press/Addison-Wesley Publishing Co.
2. Nico Schertler**.** *Simulating a day's Sky*. [En línea] [Citado el 18/06/2016] https://nicoschertler.wordpress.com/2013/04/03/simulating-a-days-sky/
3. McGuire, M., & Mara, M. (2014). Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques*.
4. Pharr, M., & Fernando, R. (2005). *Generic Refraction Simulation*. *In GPU gems 2: Programming techniques for high-performance graphics and general-purpose computation*. Upper Saddle River, NJ: Addison-Wesley.
5. Everitt, C. (2001). Interactive order-independent transparency. *White paper, nVIDIA*, 2(6), 7.
6. Wojciech Toman. *Rendering Water as a Post-process Effect*. [En línea] [Citado el 18/06/2016] http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/rendering-water-as-a-post-process-effect-r2642
7. Mtnphil. *Water shader follow up*. [En línea] [Citado el 18/06/2016] https://mtnphil.wordpress.com/2012/09/15/water-shader-follow-up/
8. Pharr, M., & Fernando, R. (2005). Volume Rendering Techniques*. In GPU gems*. Upper Saddle River, NJ: Addison-Wesley.
9. Hecht, E., Dal Col, R., Talavera, R. W., & Pérez, J. M. G. (2000). *Óptica*. Addison Wesley. p.113-120
10. Dimitrov, R. (2007). Cascaded shadow maps. *Developer Documentation, NVIDIA Corp.*
11. Parallel-Split Shadow Maps on Programmable GPUs. (2007). *In GPU gems 3: Programming techniques for high-performance graphics and general-purpose computation. Boston*, MA: Addison-Wesley.
12. McReynolds, T., & Blythe, D. (2005). *Advanced graphics programming using OpenGL*. Elsevier.
13. Pharr, M., & Fernando, R. (2005). Deferred Shading in S.T.A.L.K.E.R. *In GPU gems 2: Programming techniques for high-performance graphics and general-purpose computation*. Upper Saddle River, NJ: Addison-Wesley.
14. Shreiner, D., & Bill The Khronos OpenGL ARB Working Group. (2009). *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education.
15. Croft, D. W. (2004). *Advanced Java game programming*. Apress.
16. Wenzel, C. (2006, July). Real-time atmospheric effects in games. *In ACM SIGGRAPH 2006 Courses* (pp. 113-128). ACM.
17. Hecht, E., Dal Col, R., Talavera, R. W., & Pérez, J. M. G. (2000). *Óptica*. Addison Wesley. p.86
18. Glassner, A. S. (1989). *An introduction to ray tracing*. Elsevier.

6. ANEXO 1: MANUAL DE USUARIO

Tan solo es necesario un archivo para la correcta ejecución de este programa, cuyo nombre dependerá del Sistema Operativo en el que se vaya a ejecutar. Seleccionamos el archivo:

Kubex_<nombreDeTuSO>.jar

Copiamos ese archivo a cualquier carpeta en la que nosotros tengamos derechos para crear archivos y carpetas nuevos (muy importante, ya que los mapas se guardarán ahí). Simplemente con esto el juego estará instalado. Para ejecutarlo, hacemos doble clic en el mismo, lo que mostrará una ventana como la mostrada en la ilustración 1.

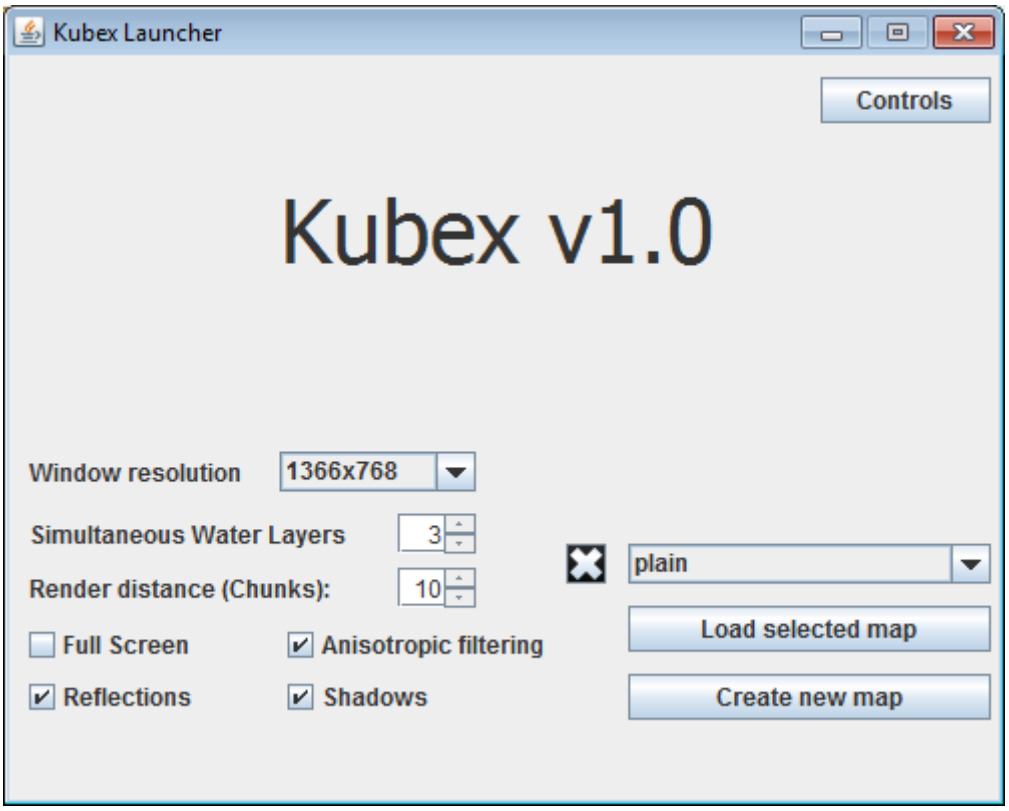


Ilustración 1 - Ventana de menú principal

En esta ventana se puede seleccionar la resolución de pantalla deseada, a elegir entre algunos tamaños predefinidos, incluyendo la posibilidad de ejecución a pantalla completa.

El *Spinner Simultaneous Water Layers* especifica cuantas capas de agua se dibujarán al mismo tiempo. Cuanto mayor sea el número, mayores superficies de agua separadas podrá atravesar cada rayo de visión, obteniendo una suma correcta del espacio que este rayo ha atravesado bajo el agua, con objeto de calcular el scattering y la absorción de la misma. En la práctica, un valor mayor de siete no es práctico: Solo es útil en cascadas en las que existen muchos cubos de agua separados, logrando con un número alto ver una imagen correcta. La mayor parte del tiempo, no obstante, solo va a causar un gasto innecesario de recursos. Para ordenadores modestos el valor se puede configurar como uno, aunque se podrán ver errores gráficos cada vez que un rayo deba salir de un cubo líquido ya que considerará que, tras entrar, toda la distancia a partir de ese punto es agua.

El Spinner *Render Distance* permite configurar la distancia de renderizado máxima. Considerando que cada chunk tiene por defecto una anchura de 32m, una distancia de render de 10 chunks equivaldrá a un radio de visión de unos 320 metros. 10 es un buen valor, aunque se puede subir hasta 30 (en la práctica, valores mayores que 20 resultan poco prácticos). En ordenadores modestos este valor puede bajarse hasta tres, aunque el mínimo recomendado, con objeto de mantener la experiencia de juego, es cinco.

Cada uno de los Combo Box existentes indican si se desea activar alguna característica gráfica (*Reflections* para reflejos, *Shadows* para sombras y *Anisotropic Filtering* para activar el filtrado anisotrópico de la GPU). En caso de ordenadores modernos, se recomienda activarlos. En caso de ordenadores más modestos se pueden desactivar, en orden de mayor a menor consumo gráfico, el agua, las sombras y, por último, el filtrado anisotrópico.

A la derecha tenemos un botón llamado *Controls*, que abrirá una ventana de información de controles como la detallada por la ilustración 2.

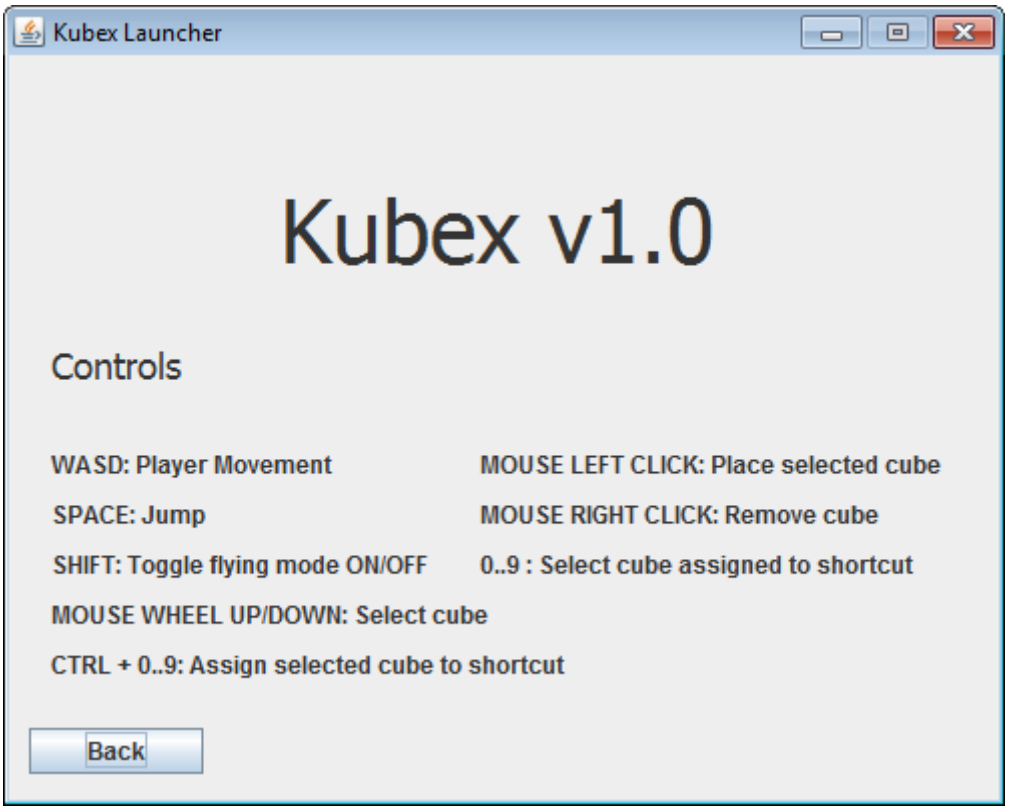


Ilustración 2 - Ventana de información de controles

Más abajo, tenemos una Drop Box con todos los mapas que hayamos creado. Seleccionando el que deseemos podemos elegir presionar la X (lo que lo eliminará), o pulsar el botón *Load Selected Map*, lo que comenzará el juego usando el mapa seleccionado.

Por último, tenemos el botón *Create new map*, que nos llevará a la ventana de creación de mapa, mostrada en la ilustración 3.

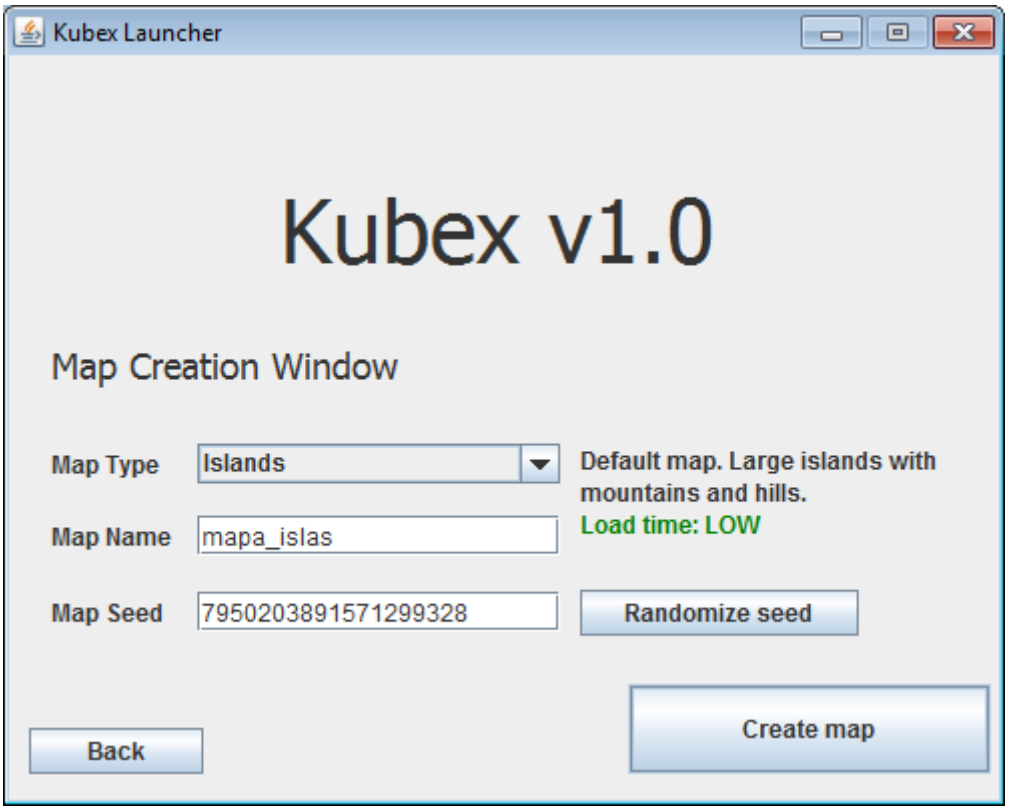


Ilustración 3 - Ventana de creación de mapa

Se permite aquí seleccionar el tipo de mapa deseado (mostrándose información de cada tipo de mapa a la derecha), ponerle un nombre al mapa (debe ser válido y no estar usado, o no permitirá crearlo) y usar una semilla para el mapa (dos mapas con la misma semilla son idénticos, así que se puede crear un clon de un mapa deseado, o sencillamente usar una semilla aleatoria pulsando el botón *Randomize seed)*.

Tras ello, pulsar en *Create Map* comenzará el juego usando el mapa recién creado. Se nos presentará una pantalla como la que la ilustración 4 nos muestra.

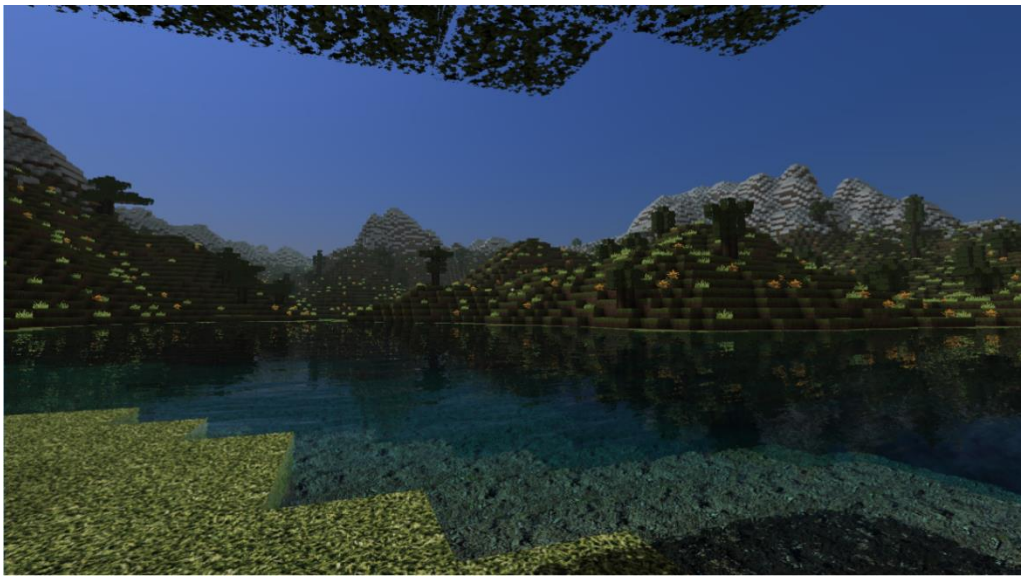


Ilustración 4- Pantalla de juego

La forma de interactuar con el mundo se basa en una combinación entre el teclado y el ratón. Mediante el teclado, podemos:

- Teclas WASD**:** Mover el personaje
- Tecla SHIFT**:** Activar o desactivar el modo de vuelo. En este modo la velocidad se ve incrementada, y se puede ascender pulsando SPACE. Nos seguimos viendo afectados por la gravedad, con lo que el vuelo se realizará mediante numerosos saltos aéreos.
- Tecla SPACE**:** Hace al personaje saltar si este se encuentra en el suelo. En casos especiales, como al estar volando o nadando, estar en el suelo dejará de ser un requisito.
- Teclas 0..9**:** Atajos de teclado. Selecciona como cubo actual el cubo asociado al atajo del número presionado. Al comienzo los atajos están configurados por defecto, pero pueden configurarse mediante el siguiente modo de control que detallamos.
- Tecla CTRL + 0..9**:** Asigna el cubo actual al atajo de teclado deseado. Así, si estamos construyendo usando unos pocos cubos, podemos configurarlos en atajos de teclado y así poder acelerar mucho nuestra tarea, al no tener que buscar cada cubo individualmente al desear cambiar el actual.
- Tecla P**:** Acelera el tiempo. El personaje seguirá moviéndose a la misma velocidad, pero el día pasará más rápido. Existen numerosas escalas de tiempo configuradas, pasando de la realista (una hora del juego equivale a una hora real) a la ultra rápida (Una hora en el juego equivale a 0'5 segundos en la realidad). Al crear un mapa nuevo, antes de que el jugador comience a configurar la escala temporal con estas teclas, una hora en el juego equivaldrá a 20 segundos en la realidad.
- Tecla O**:** Desacelera el tiempo.
- ESC**:** Cierra el juego, guardando todos los cambios en mapa y en configuraciones (escala temporal seleccionada, atajos de teclado configurados, posición del jugador, etc.)

Mediante el ratón, podremos:

- CLIC IZQUIERDO**:** Insertar el bloque seleccionado en la posición del mundo marcada por el punto central de la pantalla. El bloque no podrá ser insertado si nos encontramos demasiado cerca (dentro) de la posición elegida, o demasiado lejos.
- CLIC DERECHO**:** Eliminar el bloque seleccionado en la posición del mundo marcada por el punto central de la pantalla. Este no podrá ser eliminado si se encuentra demasiado lejos.
- RUEDA DEL RATÓN**:** Selecciona cubo. Al desplazar la rueda hacia arriba se elegirá el siguiente cubo en la lista y al rotarla hacia abajo, el anterior. La lista es cíclica: Cuando se haya sobrepasado el último cubo existente, se volverá al primero. El nombre del cubo actual seleccionado aparecerá en pantalla brevemente tras cada cambio en la selección.

Aunque el juego comenzará de día, la noche llegará en algún momento. La posición del sol es un buen indicativo del tiempo de día restante, o las sombras del terreno, que se volverán alargadas al atardecer. Tras el anochecer, el mundo se quedará en penumbra. Se recomienda al jugador posicionar luces (*Light Block*, por defecto la tecla 4 en los atajos de teclado), que iluminarán la zona en la que se coloquen. Estas luces también son útiles al explorar (o construir) cuevas, al poder estas iluminar zonas a las que el sol, ni de día, puede llegar.

El modo vuelo (tecla SHIFT) favorece la exploración rápida del terreno. El mundo es infinito, con lo que podremos viajar en la dirección que deseemos el tiempo que deseemos, encontrando en cada lugar paisajes diferentes a los visitados anteriormente. Si creamos alguna estructura que deseemos volver a ver habrá que ser, no obstante, cauteloso viajando, ya que un mundo infinito unido a un modo de vuelo rápido hacen muy posible perderse y no volver a encontrar los puntos en los que habíamos construido estructuras anteriormente. Se recomienda ir dejando marcas en el terreno al viajar que nos permitan guiarnos en la vuelta.

No existe daño en este videojuego, ni muertes, ni enemigos. El único propósito es construir y explorar, y las posibilidades son infinitas. No es necesaria ya ninguna explicación extra: Es el jugador, llevado por su imaginación, el que decida cuál va a ser su siguiente propósito, tarea en la que esta guía no puede ser ya de ayuda. Proporcionamos, no obstante, un último consejo: La TNT *explota*. Recomendamos cautela antes de intentar usarla como decoración en alguna estructura a la que tengamos aprecio.

7. ANEXO 2: RENDERING DE AGUA

Relegamos al anexo la parte más compleja y larga de todo el proyecto. El agua obtenida ha sido el resultado de intentar emular cada una de sus propiedades de la forma más realista posible, usando técnicas totalmente diferentes para cada una.

7.1. Reflexión

La mayor parte de los métodos de reflexión de agua existentes en la actualidad asumen una altura de agua constante (Lo cual es falso en el caso de los videojuegos de cubos), recomendando redibujar el mundo en esa altura aplicándole una matriz de simetría en el eje Y. De ser aplicada esta aproximación en este caso concreto, deberíamos redibujar el mundo una vez por cada altura de agua distinta en la pantalla, siendo esto computacionalmente imposible de lograr en tiempo real. Debemos tomar, pues, otra aproximación.

Se ha optado por usar una técnica nueva, basada en el *raytracing* [18] y usada en conjunción con *Deferred Shading*, que está aumentando mucho de popularidad en los últimos años. Su nombre es *Screen Space Ray Marching*[22]. Se basa en, sabiendo el color y profundidad de cada pixel en la pantalla, detectar el punto de colisión con el agua y trazar desde él el rayo reflejado, comprobando su colisión por cada pixel por el que este fuera a pasar. Este método, aunque es el único que permite solventar en tiempo real nuestro problema, no está exento de fallos, al solo permitir reflejar objetos que se estén viendo en pantalla en este momento (fallando para objetos fuera de la pantalla, u objetos siendo tapados por otros objetos), y al causar artefactos visuales al mover la cámara. Es inviable aplicarlo sin contar con algún otro sistema que nos dé un color de reflejo alternativo si el *Screen Space Ray Marching* falla para algún pixel. Por fortuna, el método de renderizado de cielo que hemos elegido permite su generación aportando solo una dirección de vista, que es justo lo que obtenemos al calcular el vector de dirección del rayo reflejado por el agua. Por tanto, podremos reflejar el cielo cada vez que este algoritmo falle en algún pixel. Esto, aunque seguirá causando errores visuales (como se aprecia en los bordes de la parte superior de la figura 17, donde reflejos del paisaje aparecen con el color del cielo al no estar siendo vistos en este momento), los minimizará lo suficiente como para que sean aceptables. En cuanto a los artefactos visuales, podemos confiar en que el oleaje será lo suficientemente elevado como para ocultarlos.

La implementación del *Screen Space Ray Marching* utilizada, salvo por algunos añadidos, se ha obtenido de [3], al ofrecerla ellos ya altamente optimizada. Se ha elegido una *Thickness* para cada pixel de 20 (suponiendo así un mundo compuesto por cubos de una profundidad igual a 20 metros, en vez de uno). La razón para esto es que, aunque se causen algunos errores en reflejos cercanos, los reflejos lejanos se podrán fundir perfectamente (por ejemplo, un árbol con una montaña detrás en la que, aunque el árbol tape reflejos de la montaña, podemos asumir un árbol más profundo y reflejar este en vez), sin renderizar trozos de cielo, quedando mucho más estéticos. Además, una anchura mayor ocultará algunos artefactos visuales. Este número ha sido obtenido por prueba y error. Podemos ver un ejemplo

[22]The future of screenspace reflections. Bartlomiej Wronski [En línea] [Citado el 18/06/2016] http://www.gamasutra.com/blogs/BartlomiejWronski/20140129/209609/The_future_of_screenspace_reflections.php

de este algoritmo en funcionamiento en la figura 18, en el que, incluso sin aplicar oleaje, los resultados no muestran ningún error apreciable.

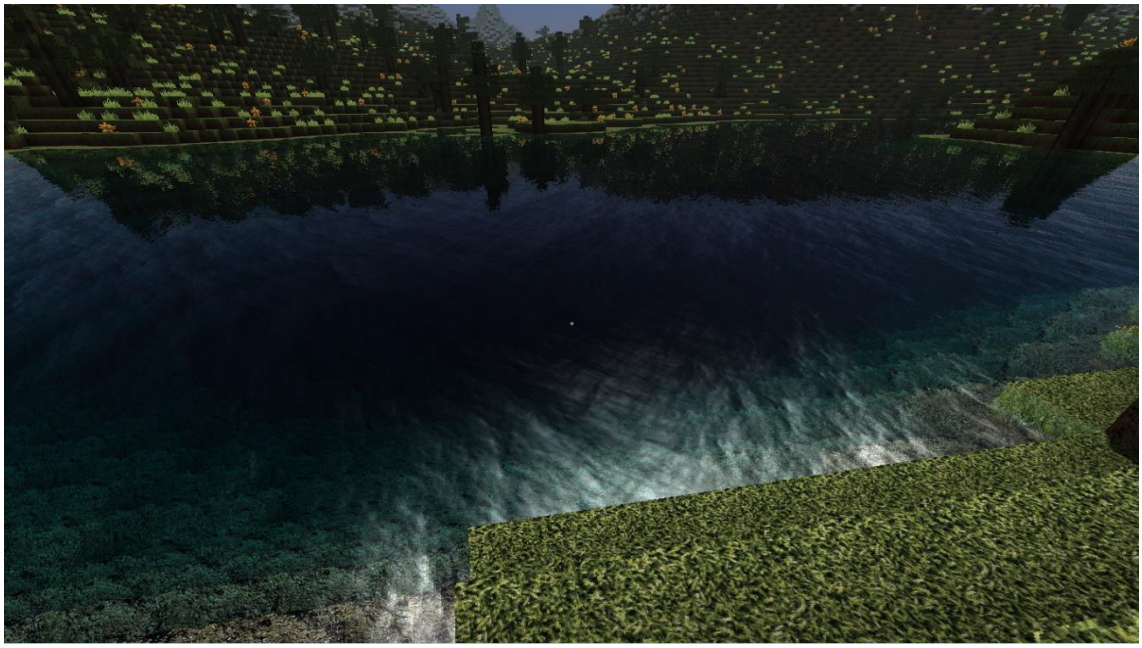


Figura 17 - Fallos de reflexión usando Screen Space Ray Marching por falta de información en pantalla

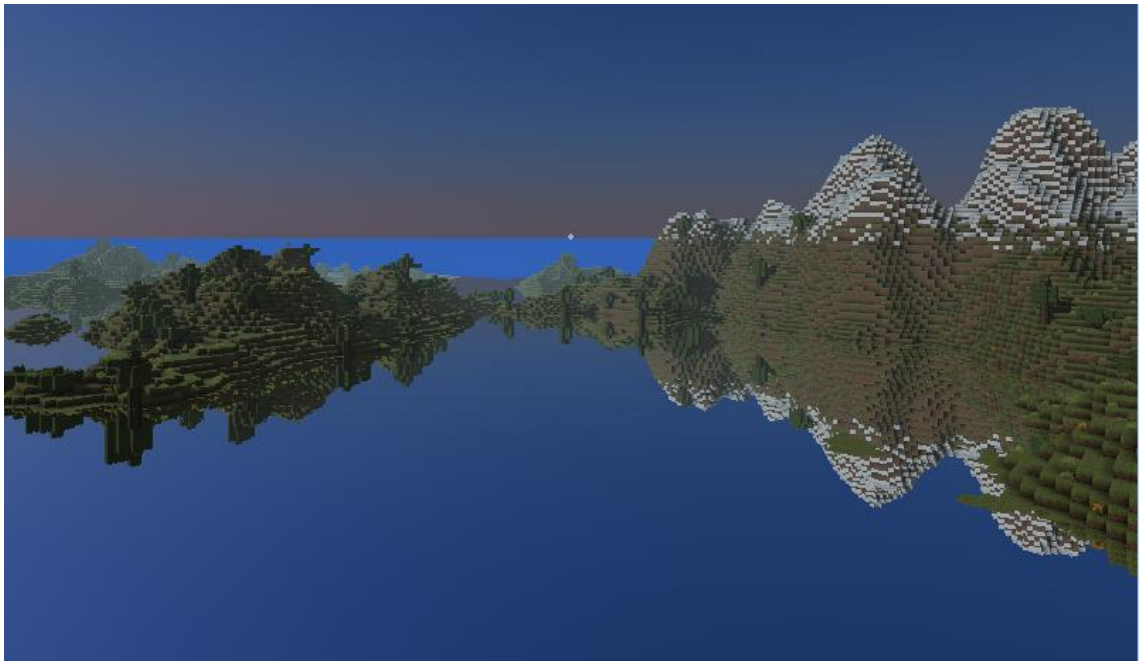


Figura 18 - Reflexión pura

7.2. Refracción

La refracción es un fenómeno prácticamente imposible de calcular correctamente sin usar un *raytracer*. Por tanto, usaremos una aproximación que, aunque no sea realista, consigue engañar a la mente jugador, siendo tomada como real. Usaremos el método propuesto en [4].

Guardado en la cuarta coordenada de la imagen aportada a nosotros por la segunda pasada del *Deferred Shading*, se encontrará un valor que indica si ese pixel de la imagen se encuentra tras una capa de agua. Gracias a ello, podremos modificar la posición en la cual *sampleamos* la imagen en función de la normal del agua en el punto de colisión. Si intentamos *samplear* un lugar de la imagen fuera del agua, ese valor nos indicará que la refracción es incorrecta y nos quedaremos con la imagen normal, sin perturbar. Esto, sumándolo al oleaje en movimiento, proporcionará una refracción razonable, que dará aspecto realista (aunque no lo sea). Un ejemplo de este método de refracción puede verse en la figura 19, en la que se puede apreciar que la refracción ha curvado las líneas rectas de los cubos bajo el agua. Este efecto, sutil al ser visto en una imagen, es mucho más apreciable al verse en movimiento.

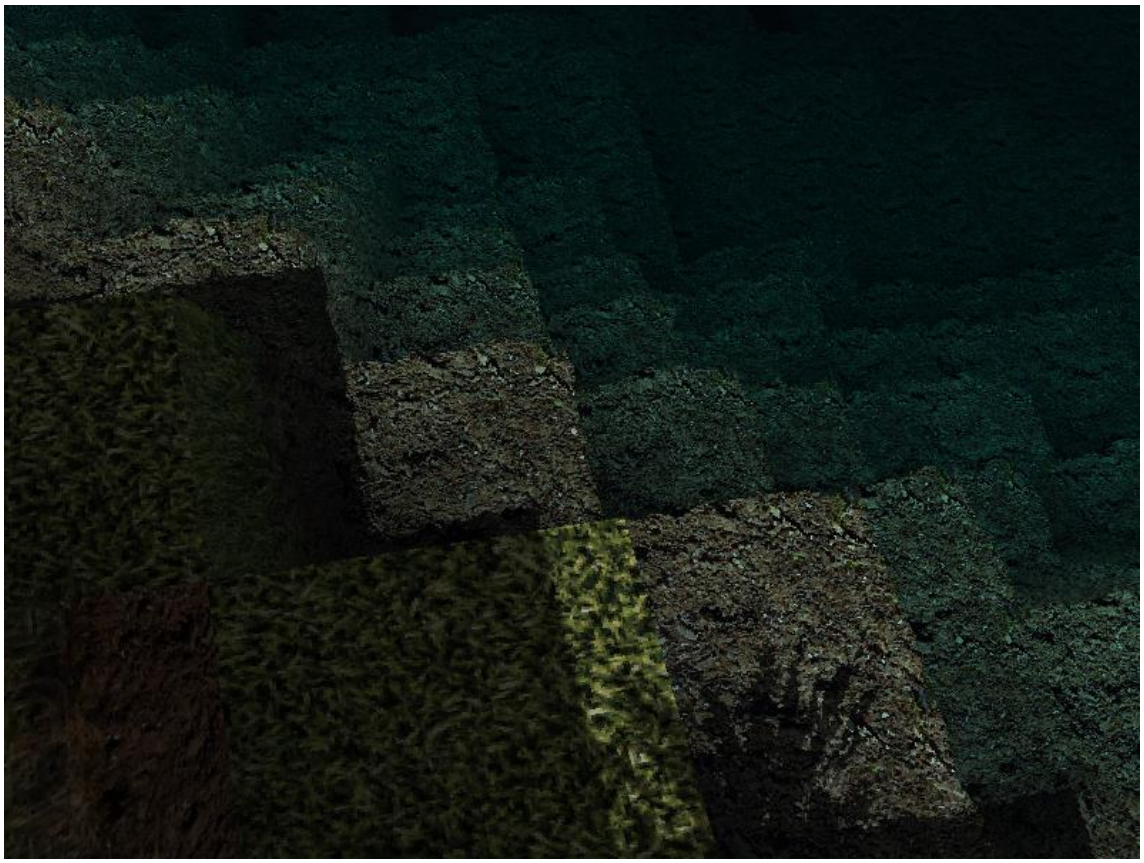


Figura 19 - Refracción acentuada para visibilidad.

7.3. Absorción / *Scattering* de la luz

Lo primero que necesitamos para calcular el valor de luz absorbida y el *scattering* aportado a la imagen es saber que distancia exacta el rayo de luz atraviesa bajo el agua. Como el rayo puede entrar y salir del agua en repetidas ocasiones, se implementa el algoritmo de Depth Peeling [5]. Este algoritmo cambia el orden natural de render, separando la escena por capas. En la primera se encontrará, en cada pixel, la profundidad de los polígonos más cercanos a la cámara. En la segunda se encontrarán la profundidad de los polígonos que, de no existir los de la primera capa, habrían sido los más cercanos a la cámara. Repitiendo este proceso para el número de capas especificado se obtiene un listado de polígonos por los que cada rayo de luz lanzado por cada pixel colisionaba, en orden. El rayo solo encontrará un

polígono de agua al entrar o salir de esta: Por lo tanto, para un pixel dado, una cantidad impar de capas de agua significará que, al final de su recorrido, el rayo de luz ha terminado entrando en el agua y no ha salido de la misma antes de colisionar contra el terreno, y una par que el rayo ha entrado y, al final de su recorrido, ha terminado saliendo (Cero capas en un pixel significarán que el rayo ni tan siquiera ha llegado a entrar al agua en su recorrido). Hay que tener en cuenta que por cada capa generada hay que hacer una llamada de dibujo extra, redibujar toda el agua de la pantalla y guardarla en una textura de profundidad con el tamaño de la pantalla entera. Es decir, cada capa extra añade un gran uso de memoria gráfica y de tiempo de computación. Tres es un número razonable (entrar en agua, salir y entrar en agua de mar, por ejemplo), aunque cinco o siete sería mejor aún al cubrir algunos casos especiales, si contamos con un ordenador que pueda soportarlo. Se puede configurar mediante el menú este parámetro, elevando el número en secuencias de dos hasta un total de 19 capas, aunque no suele valer la pena el gasto extra.

Al tener ya guardada la profundidad de cada capa de luz y la profundidad de cada pixel solido de la imagen final (guardados en una textura, sin aplicar sombreado ni iluminación, en el dibujo de la geometría del mundo), podemos calcular la distancia exacta hasta cada punto[23]. Calcular la distancia exacta recorrida en el agua, sabiendo el valor de profundidad de cada capa, será trivial. En caso de que un rayo de luz recorra todas las capas de agua reservadas (siendo la última siempre una entrada a agua), el sistema no podrá detectar si ha salido de esta o no en algún punto, con lo que asumirá que hay agua hasta la distancia del pixel sólido final.

Sabiendo la distancia exacta de agua por la que el rayo pasa, podremos calcular la absorción y el *scattering* de la luz que esta ocasiona. Tanto la absorción como el *scattering* dependen de la cantidad de minerales disueltos en el agua o la temperatura, así que hay muchos entre los que elegir. Los valores de vida media para cada canal de luz en el agua se han extraído de [6]. Nos hemos inspirado en [7] para las fórmulas de *scattering*, y los valores.

Se ha aplicado la función de fase para medios participativos asumiendo uniformidad de medio[24] en el caso de la absorción y out-scattering. Para el scattering, se ha aplicado una aproximación de la misma, en la que se mezcla, exponencialmente en función de la distancia y un coeficiente de scattering, un color de agua profunda azulado con el color extinto como tal. La razón de aplicar una mezcla es que el scattering causa dos fenómenos: Extingue color (*out scattering*) y lo añade al rayo (*in scattering*). La fórmula resultante es:

$$colorTrasAbsorcion = colorInicial * e^{-(ext\_rojo,ext\_verde,ext\_azul)*distancia}$$

$$colorFinal = mix(\ colorScattering, colorTrasAbsorcion\ , e^{-coef_{scattering}*distancia}$$

Los valores de extinción (siendo inversos al valor de la vida media, en metros, de cada canal de color en el agua) son (0.46,0.09,0.06) . Para el scattering usaremos un valor de 0.01 para mostrar un agua bastante limpia, aunque podría aumentarse en caso de agua sucia (por ejemplo, ríos arenosos). El color de scattering, o color de océano profundo, se ha elegido como

[23] Getting the true z value from the depth buffer. StackOverflow [Citado el 18/06/2016] http://stackoverflow.com/questions/6652253/getting-the-true-z-value-from-the-depth-buffer

[24] Light Transport in Participating Media [Citado el 20/04/2016] https://www.cs.dartmouth.edu/~wjarosz/publications/dissertation/chapter4.pdf

(0.05,0.05,0.1) para dar un aspecto relativamente oscuro a las masas de agua grandes. Ha sido elegido por prueba y error.

Esta extinción y scattering se puede apreciar en la figura 20. Nótese la rápida absorción del color rojo por parte del agua. En las zonas más lejanas, donde todo el color ha sido ya absorbido, solo quedará el color de scattering de fondo.

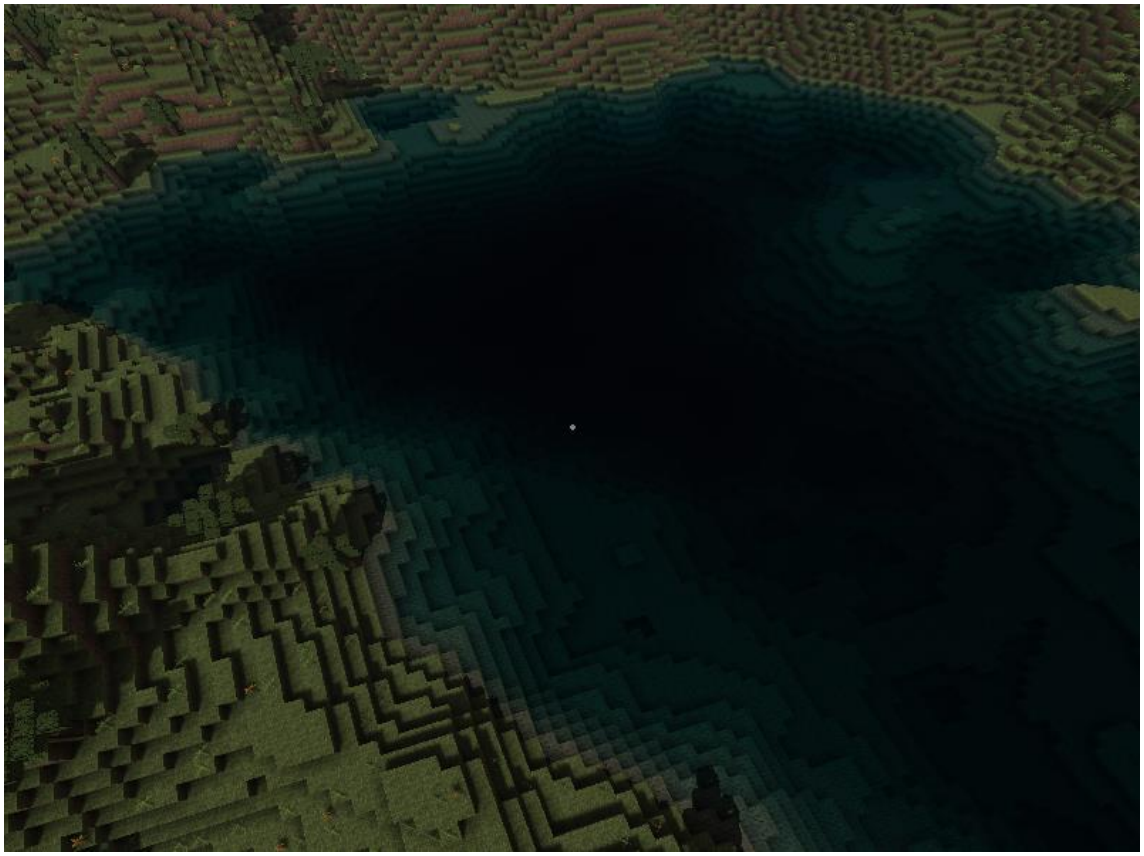


Figura 20 – Absorción / Scattering puro, sin reflejos.

7.4. Perturbaciones (oleaje)

Para aplicar este efecto se usa una textura de normales del agua extraída de Internet. Considerando una base en la que el agua tiene siempre una normal (0,1,0), en esta textura el canal rojo simularía la perturbación de la normal en el eje x, el canal verde en el eje z y el canal azul en el eje y, por defecto. Esta textura deberá ser normalizada a valores en el rango [-1,1][25]

El propio algoritmo de *Depth Peeling* guardará la normal de la primera capa en una textura adicional. Es esa normal la que consideramos y perturbamos en función al valor de esta textura. Al estar la textura considerando una base (0,1,0), deberemos efectuar un cambio de base. Se realizará de la misma forma en la que se obtienen los vectores unitarios de la cámara en un trazador de rayos: Se hará un producto vectorial entre la normal y un valor por defecto, por ejemplo (1,0,0). El resultado de este producto será nuestro vector unitario u. Tras ello, se

[25] OpenGL Water Tutorial 7: Normal Maps. ThinMatrix. [Citado el 18/06/2016] https://www.youtube.com/watch?v=7T5o4vZXAvI

hará el producto vectorial entre u y la normal, obteniendo el vector unitario v. La nueva normal tras la perturbación será igual, por tanto, a aplicar la proyección de cada canal de la imagen de perturbación sobre cada eje de esta nueva base, usando el producto escalar, y normalizar el vector resultante. Más concretamente:

$$normal_{perturbada} = (dot(u, textura_{pert}.r), dot(normal, textura_{pert}.b), dot(v, textura_{pert}.g))$$

Estas perturbaciones, no obstante, son demasiado intensas. Podemos multiplicar por un valor el componente rojo y verde de la imagen para suavizarlas. Por ejemplo, para los reflejos, los multiplicamos por 0.07. Para la refracción, sin embargo, al desearla más suave, los multiplicamos por 0.02. Para los reflejos especulares la multiplicamos por 0.3 como mínimo, al desear reflejos más intensos. En caso de agua cayendo, esta multiplicación se incrementará hasta incluso uno. El resultado de aplicar estas perturbaciones a una capa de agua puede apreciarse en la figura 21. Las alteraciones del ángulo de la normal del agua causan que el *fresnel* de la misma priorice la refracción en algunos puntos y la reflexión en otros, creando el efecto de olas.

Un oleaje quieto, sin embargo, no aporta nada. Necesitamos que se mueva para que dé un aspecto realista. Moveremos el oleaje en función de la corriente marcada por la gravedad, es decir, en la dirección en la que se maximice la dirección y negativa del flujo de agua[26]. La velocidad de la corriente rondará desde los 0.2 m/s (agua horizontal, no puede caer en ninguna dirección) a los 1.2 m/s (agua completamente vertical, cae directamente hacia la gravedad). La corriente se determinará por el sampleo de un punto u otro de la textura de normales, dado el tiempo y la posición, creando ilusión de movimiento.

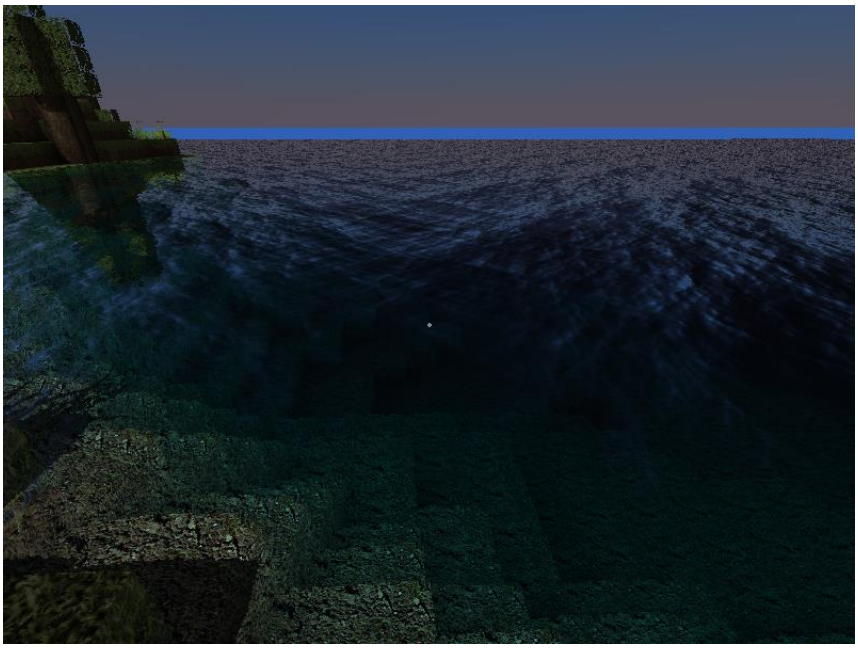


Figura 21 - Oleaje exagerado para visibilidad.

[26] Get water flow direction vector from water normal vector. StackOverflow [Citado el 18/06/2016] http://gamedev.stackexchange.com/questions/119086/get-water-flow-direction-vector-from-water-normal-vector

7.5. Reflejos de luz

Se aplica el modelo de *Blinn-Phong* [8] para crear reflejos en la superficie del agua. Este modelo crea reflejos de luz más realistas que *Phong* para luces situadas en el infinito, como el sol. Usando la normal especular calculada antes, se crearán reflejos en función de la posición del sol. El coeficiente de *Blinn-Phong* en nuestro caso será de 60, hallado por prueba y error.

Con objeto de que no pueda haber reflejos en zonas a las que el sol no llegue, se analizará en el cálculo de sombras si cada punto de agua está o no sombreado, guardando, si lo está, un valor en la cuarta coordenada (w) de la imagen de cero. Conforme el anochecer vaya llegando, la w por defecto para lugares con agua no sombreados (0.8) irá decreciendo también, hasta ser cero al anochecer. El valor final de la especular será igual al producto del resultado del blinn-phong por esa coordenada w, que puede apreciarse en la figura 22, a continuación.

Figura 22 - Reflejos en el agua

7.6. Coeficiente de *fresnel*

El coeficiente de *fresnel* [9] indica que cantidad de luz se refracta y que cantidad se refleja al pasar de un medio a otro. Estas ecuaciones son computacionalmente complejas, con

lo que usaremos una aproximación relativamente buena: La aproximación de Schlick[27]. Esta aproximación es fiel solo en un rango concreto de índices de refracción (de 1.4 a 2.2)[28]. En el caso de transiciones aire-agua, como la que estamos estudiando, obtenemos un índice de refracción entre los dos elementos de 1.33 que, aunque se encuentra ligeramente fuera del rango de fidelidad, sigue siendo correcto para un entorno no científico, al ser el error aún mínimo.

Calculando el *fresnel* con la normal usada para calcular los reflejos de luz (la menos suavizada) y mezclando el color reflejado y refractado con él, obtendremos unas olas de un aspecto bastante bueno y nuestro sistema de render de agua estará completo. Podemos apreciar el fresnel en la figura 23, a continuación. En las partes bajas de la imagen, el ángulo de incidencia a la superficie del agua es casi perpendicular a esta, con lo que la refracción prevalece. En los lugares más alejados los rayos de luz inciden de forma casi oblicua lo que, unido a la ausencia de color refractado en esa zona (al haber sido absorbido por la gran masa de agua), crea un reflejo muy definido.

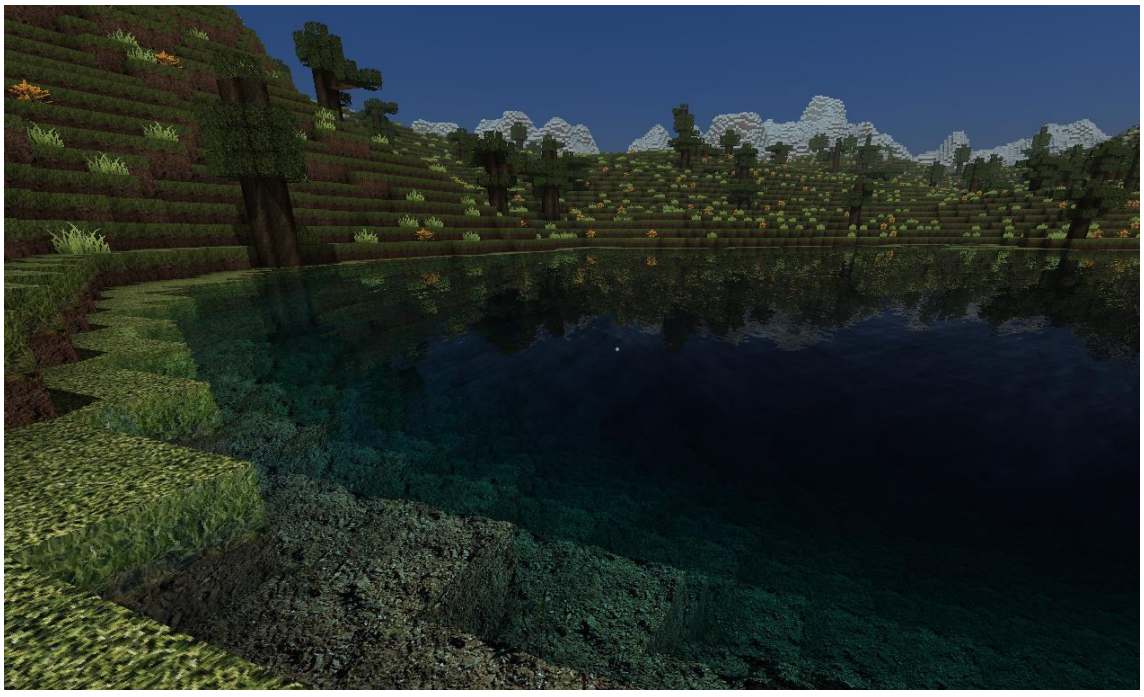


Figura 23 - Fresnel en masa de agua

7.7. Visión subacuática

Deberemos aplicar *shaders* especializados al estar dentro del agua. Todo el sistema de cálculo de distancias del rayo dentro del agua se invertirá, al comenzar este ya por defecto

[27] Schlick's approximation. Wikipedia. [Citado el 18/06/2016] https://en.wikipedia.org/wiki/Schlick%27s_approximation

[28] Memo on Fresnel equations. Sébastien Lagarde. [Citado el 18/06/2016] https://seblagarde.wordpress.com/2013/04/29/memo-on-fresnel-equations/

dentro de la misma: Un número impar de capas de agua significará que el rayo, al final de su recorrido, ha salido del agua, y un número par que ha terminado su recorrido dentro de ella.

Tras calcular la distancia que cada rayo ha pasado dentro del agua, le aplicaremos absorción y *scattering* como se hacía en el exterior. No obstante, modificaremos ligeramente los parámetros con objeto de darle un aspecto más atractivo:

- La extinción de color se reduce ligeramente, a (0'3,0'06,0'04)
- El coeficiente de *scattering* aumenta a 0.04. Esto dará un aspecto más nublado a los fondos marinos.
- El color de *scattering* ahora es más complejo. Podemos al estar dentro del agua aproximar relativamente mejor la función de fase, al conocer la iluminación exacta a nuestro alrededor en cada momento. Este color estará basado en la iluminación que rodee al jugador. Así, cerca de la superficie y en pleno día, el agua tendrá un color azul claro. En las profundidades marinas, donde nada de luz llega, el color de scattering será mucho más oscuro.

La superficie del agua es otro problema. *Screen Space Ray Marching* funciona solo para reflejar objetos ya visibles en la pantalla, pero bajo el agua, al haber pequeñas montañas, la superficie suele reflejar partes no existentes en la vista actual. La cantidad de errores es tan alta que se ha decidido no aplicar reflejos bajo el agua. De todas formas, esos reflejos suelen nublarse enormemente debido al *scattering* y la absorción del agua y además, al no estar los seres humanos preparados para vivir bajo el agua, ni siquiera los echamos en falta si no están. En caso de necesitar reflejar un rayo, sencillamente mostraremos el pixel original pero multiplicando en las ecuaciones de absorción y *scattering* la distancia recorrida por el rayo por dos, como ocurrirá si fuera reflejado (aunque se mostraría otra dirección). Esto da un resultado bastante vistoso.

A la hora de calcular el *fresnel* bajo el agua, la aproximación de *Schlick* ya no nos sirve, al existir un índice de refracción de 0.75. Aplicamos una fórmula ideada por nosotros mismos, que no tiene nada de científico pero aproxima más o menos la curva que produce el *fresnel* agua-aire, en todos sus puntos (posee nomenclatura común a la ecuación de *Schlick* original):

$$fresnel = R0 + (1 - R0) * (1.33 - cos\theta)^{10}$$

Bajo el agua, la refracción de la superficie será el doble de intensa que al mirar el agua por fuera. Se adopta esta aproximación para que la superficie del agua se distinga más claramente del mar en si en condiciones poco apreciables, como el cielo nocturno.

Todos estos algoritmos producen resultados como los apreciables en la figura 24. Nótese la reflexión absoluta a partir de cierto ángulo, en la que dada la imposibilidad de reflejar el suelo oceánico se opta por duplicar el scattering / absorción y mostrar las montañas del fondo. Nótese también como prácticamente toda la componente de luz roja del cubo de luz a la izquierda ha sido ya absorbida al llegar a nuestra posición, mostrando un cubo azulado.

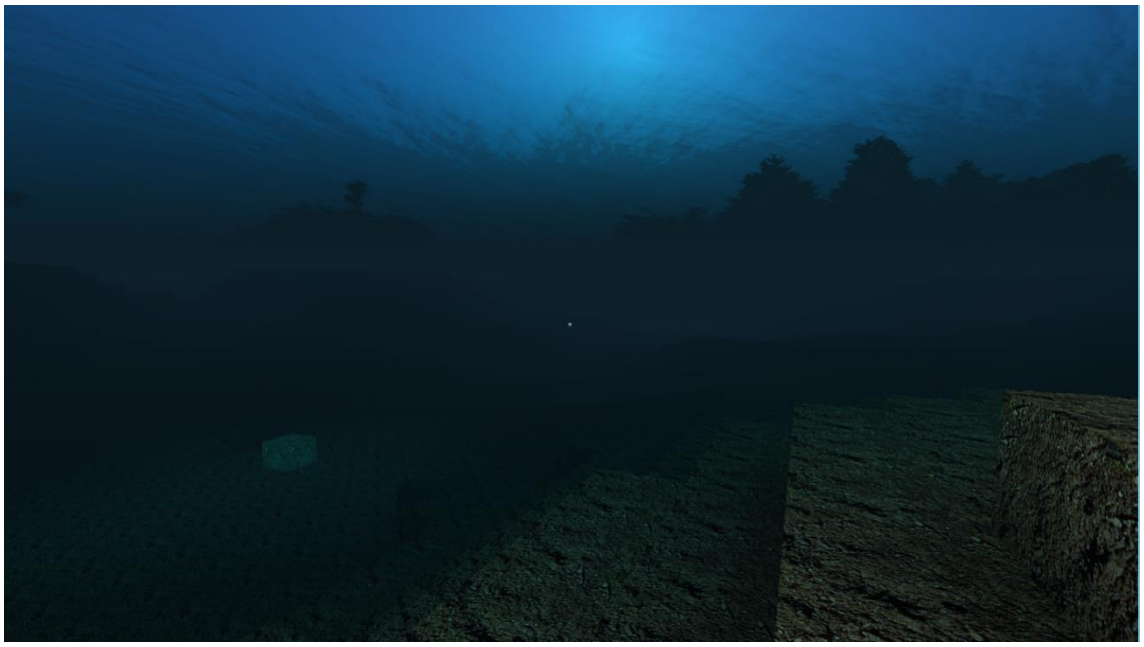


Figura 24 - Visión subacuática.

8. ANEXO 3: *CASCADED SHADOW MAPPING*

Cascaded shadow mapping se basa en una división logarítmica en numerosas partes del *frustrum*[29] de visión. No obstante, en [11] se propone una forma de división más efectiva, mezclando división lineal y logarítmica, aplicando en nuestra implementación una variación de esa aproximación, con cuatro divisiones en total. Esas partes separadas del frustrum (al comienzo del mismo más pequeñas, al final más grandes) serán, una a una, las que la matriz de proyección de la luz deba englobar. Para ello, en vez de guardar todas las sombras de la escena en una misma textura, se hará uso de tantas texturas como divisiones del frustrum de visión se hayan producido, lo que conllevará ese mismo número de renderizados de la escena. Al estar solo almacenando el valor del *depth buffer* de cada render sucesivo en una textura (usando shaders computacionalmente muy simples), redibujar la escena será mucho más barato que al hacerlo a texturas de color, en shaders complejos. Aun así, el coste de redibujar la escena tantas veces no es anecdótico, con lo que se ha añadido una opción en el menú para desactivar las sombras, para poder usar en equipos de gama baja. De hecho, en nuestra implementación, la única sombra que generaremos será la del sol.

Una aclaración de estos tecnicismos es simple. Sencillamente, deseamos partir el cono de visión en fragmentos y, en vez de guardar todas las sombras de la escena en una misma textura, dedicar una textura de sombras para cada fragmento partido del cono de visión. Al haberlo dividido de tal forma que las primeras secciones engloben un trozo pequeño de terreno, las sombras en ese lugar serán de alta calidad (pero abarcarán poco terreno). En divisiones sucesivas los trozos serán cada vez más grandes, siendo las sombras cada vez de peor calidad pero abarcando más terreno, hasta así llegar a la división final. Al mirar a lo lejos y ver las sombras dadas por las divisiones lejanas del frustrum ni tan siquiera notaremos que las estas tienen poca resolución, al estar perdiendo nosotros también agudeza visual de forma natural en función de la distancia. Para lugares muy cercanos, donde nuestra agudeza visual es máxima, si necesitaremos sombras de alta calidad, que nos serán proporcionadas por las primeras particiones. Dar para cada distancia de visión unas sombras con una calidad proporcional a nuestra agudeza visual es en lo que se basa este algoritmo.

Tras tener todas las texturas de sombras ya dibujadas, tendremos solo que dibujar la escena normalmente y, para cada pixel, determinar a qué distancia se encuentra del frustrum de visión y, por tanto, con que matrices tenemos que hacer la transformación al espacio de la luz y con qué textura tenemos que comparar.

El protocolo necesario para calcular cada sombra asociada a cada división del frustrum se hará como sigue:

- Se obtienen los ocho puntos que marcan los límites del pedazo de frustrum seleccionado
- Se pasan a coordenadas de mundo, con la matriz vista inversa

[29] El cono de visión que el usuario posee dentro del videojuego. Más información: Viewing Frustrum. Wikipedia [Citado el 21/04/2016] https://en.wikipedia.org/wiki/Viewing_frustum

- Se pasan a coordenadas de la luz, sin aplicar matriz de proyección por ahora
- Se construye una Bounding Box orientada con los ejes, y se hace que englobe a esos ocho puntos. Con esa Bounding Box se construirá la matriz de proyección ortográfica. La coordenada z inicial de esa matriz ortográfica deberá abarcar todo el posible mundo visible desde la misma, ya que no podemos solo calcular las sombras de un pedazo de mundo partiendo desde la altura que deseemos, sino que tenemos que partir desde la luz. Si no, algunas sombras podrían ser incorrectas al no estar considerando objetos que tapen la luz por encontrarse más atrás de lo esperado.
- Tras tener ya la matriz de proyección, se dibuja la escena usándola (junto a la matriz vista de la luz, claro está) y se guarda en una textura. En mi caso particular, estoy usando un TextureArray2D, siendo cada índice igual al número de partición del frustrum.
- Se repite para todas las particiones.
- Al dibujar la escena, como sabemos a qué distancia de frustrum hemos realizado cada partición, tan solo tendremos que fijarnos en el zbuffer de cada pixel, y mirar las sombras de una partición o de otra. Un ejemplo de esta selección puede apreciarse en la figura 25, simplificando cada partición del frustrum mediante colores.

En la figura 26 podemos apreciar la misma escena mostrada en la figura 25 sombreada de forma correcta. Las distancias están adaptadas para que las zonas de cambio de shadow map no sean muy aparentes, aunque pueden distinguirse poniendo atención. No se ha tenido que aplicar ningún algoritmo de blending entre zonas al estar las distancias adaptadas para no ser esto necesario, causando que no existan cambios de calidad muy bruscos entre cada cascada.

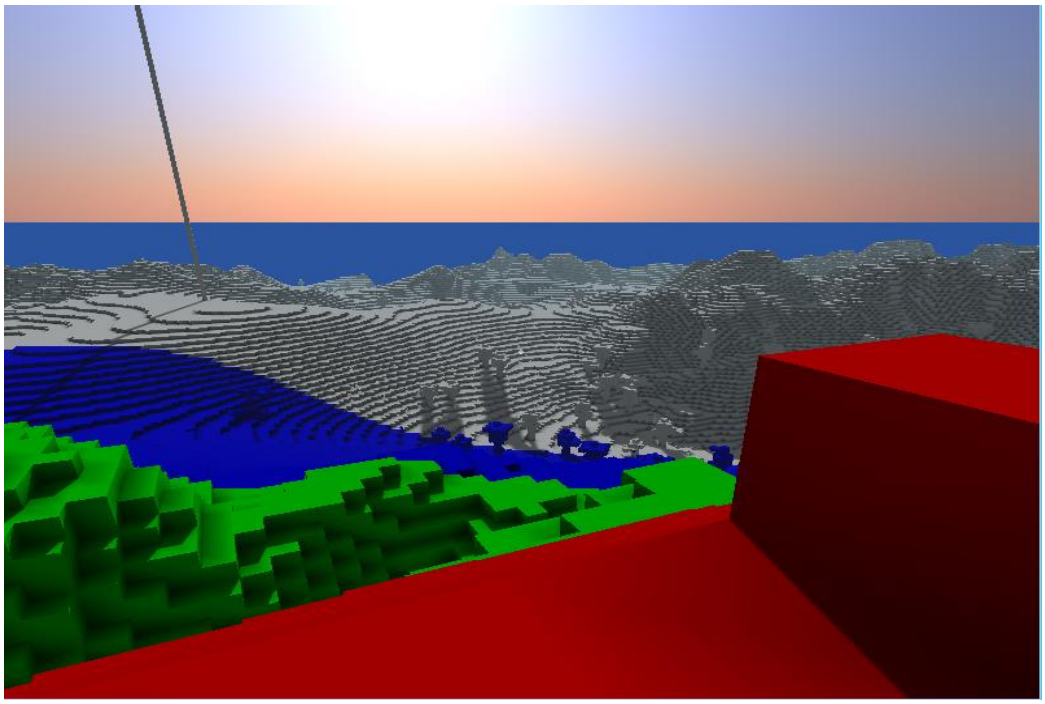


Figura 25 - División del frustrum de visión en distancias. (Marcadas por rojo, verde, azul, blanco).

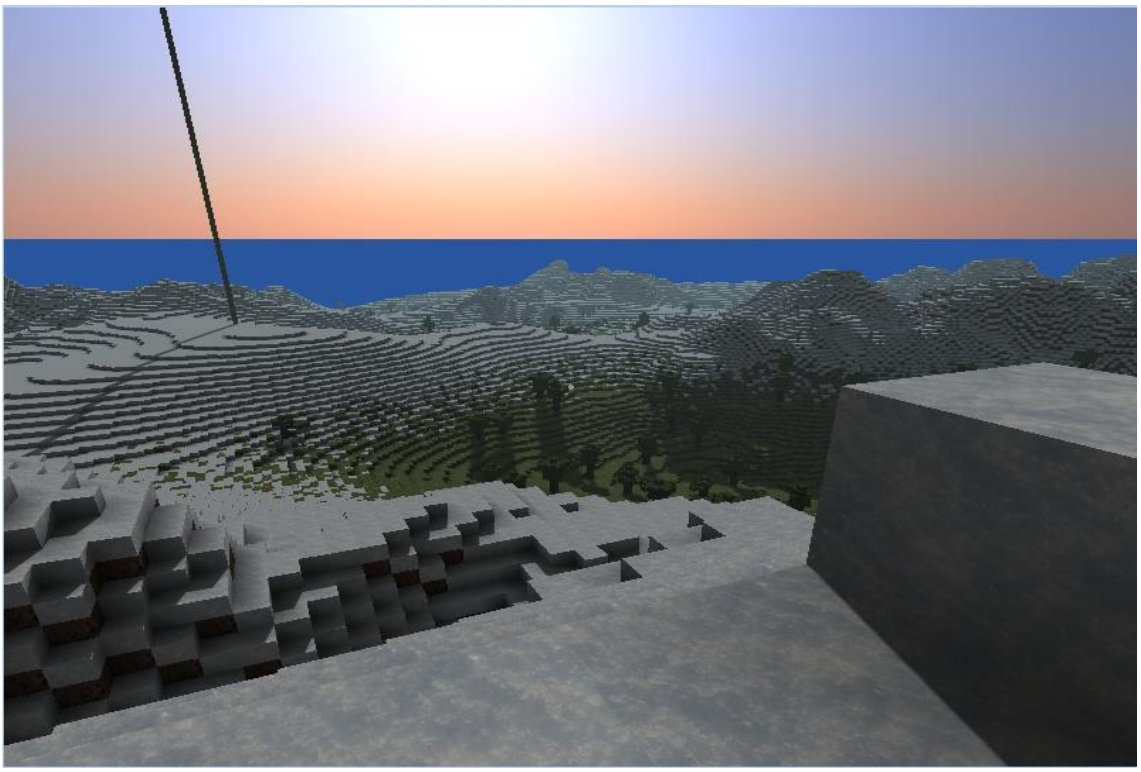


Figura 26 – Ejemplo de escena con sombras a varias distancias

Aun con esta mejora, se pueden ver dientes de sierra en las sombras (causados porque, aunque hemos mejorado la resolución, no la hemos aumentado lo suficiente en puntos inmediatamente cercanos como para alcanzar la excepcional agudeza visual que se tiene en distancias cortas, por ser un gasto de recursos enorme para cubrir poco terreno), unas líneas discontinuas en las sombras (*Shadow Acne*) y unos pequeños saltos al avanzar el día (causado porque la resolución va cambiando en función del movimiento del sol).

Los dientes de sierra se han arreglado parcialmente utilizando *PCF* y *Poisson Sampling*[30] con cuatro sampleos y una dispersión de milímetros. Con una dispersión más alta los dientes de sierra se suavizaban totalmente, pero los saltos por cambio de resolución empeoraban. Con esta configuración se ha intentado obtener la mejor solución posible para ambos casos, aunque los dientes de sierra siguen siendo visibles, aunque con mayor dificultad, y sigue habiendo pequeños saltos al avanzar el sol, aunque mucho menos aparentes. El resultado, en general, es aceptable.

Las líneas discontinuas se han arreglado añadiendo un offset a la profundidad de cada pixel, en función de su distancia a la cámara (con un offset fijo inicial, para evitar este mismo problema al acercarnos mucho a un bloque). Esto, no obstante, podía causar problemas cuando el sol se encontraba paralelo al polígono a sombrear, haciendo reaparecer el shadow acne. La solución ha sido añadir el offset en la dirección de la normal de cada polígono, en vez de en la profundidad. La fórmula usada es:

[30] Tutorial 16: Shadow Mapping. [Citado el 18/06/2016] http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

$$offset = normal * \max(mod(distanceVector) * 0.01\ , 0{,}015)$$

Ese offset mínimo de 0.015 se usa para evitar pequeños errores en polígonos extremadamente cercanos.

Con esto, el shadow acne se soluciona en todos los casos y, aunque se añade un pequeño bias, es tan diminuto que no es apreciable.

Con objeto de acelerar el cálculo de sombras, los polígonos en los que su producto escalar entre su normal y la dirección de la luz sea positivo (es decir, en los polígonos a los que la luz les da "por detrás") se considerarán completamente sombreados, eliminando errores. Además, para evitar fallos en los que una cara pasa de sombreada a luminosa repentinamente (cuando el sol cruza justo el ángulo necesario para comenzar a incidir en ella, por ejemplo), las sombras se suavizarán usando el producto escalar anteriormente descrito, haciendo que los polígonos pasen suavemente de estar sombreados a no estarlo. Por último, tras el anochecer comenzaremos a sombrear suavemente todos los polígonos a los que aún las sombras no les llegan (por ejemplo, torres altas), hasta estar todo el mapa en la oscuridad brevemente tras la puesta de sol.

Con objeto de conseguir sombras lo más detalladas posibles en la cercanía, usaremos el método de división de frustrum detallado en [11], pero dividiendo la distancia inicial de corte entre 4, la distancia del segundo corte entre 2.5 y la distancia del tercero entre 1.5. Esto causará peor definición de sombras en la lejanía, pero dado que esas sombras solo suelen ser relevantes en el anochecer o amanecer (mientras que las sombras cercanas lo son todo el tiempo), es un sacrificio razonable.

Dentro de nuestros shaders existirá un parámetro, llamado "shadowAttenuation", que tendrá un valor de uno si el objeto está recibiendo iluminación solar directa y de 0.3 si está completamente en sombra, con valores intermedios en los bordes debido al PCF y al PoissonSampling. Este valor se multiplicará a la luz natural del cubo, con lo que un cubo sombreado tendrá una iluminación del 30% con respecto a los cubos iluminados por el sol.

9. ANEXO 4: IMÁGENES EXTRAS

9.1. Mejoras gráficas ilustradas

Figura 27- Ambient Occlusion Desactivado

Figura 28 – Ambient Occlusion Activado

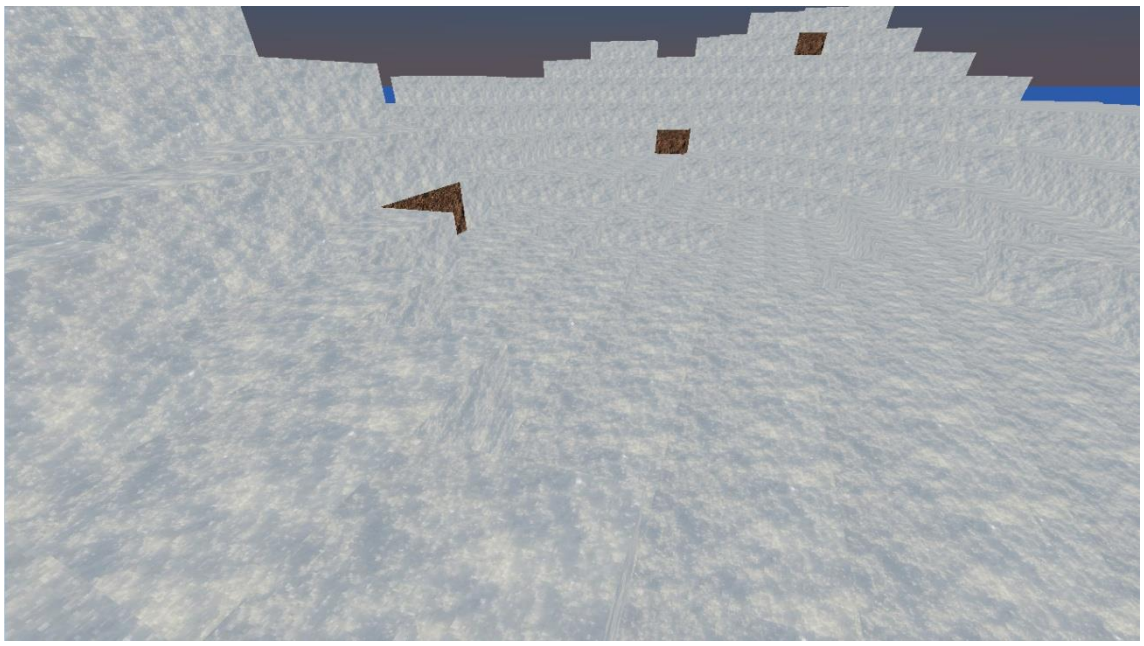


Figura 29 – Ambient Occlusion Desactivado

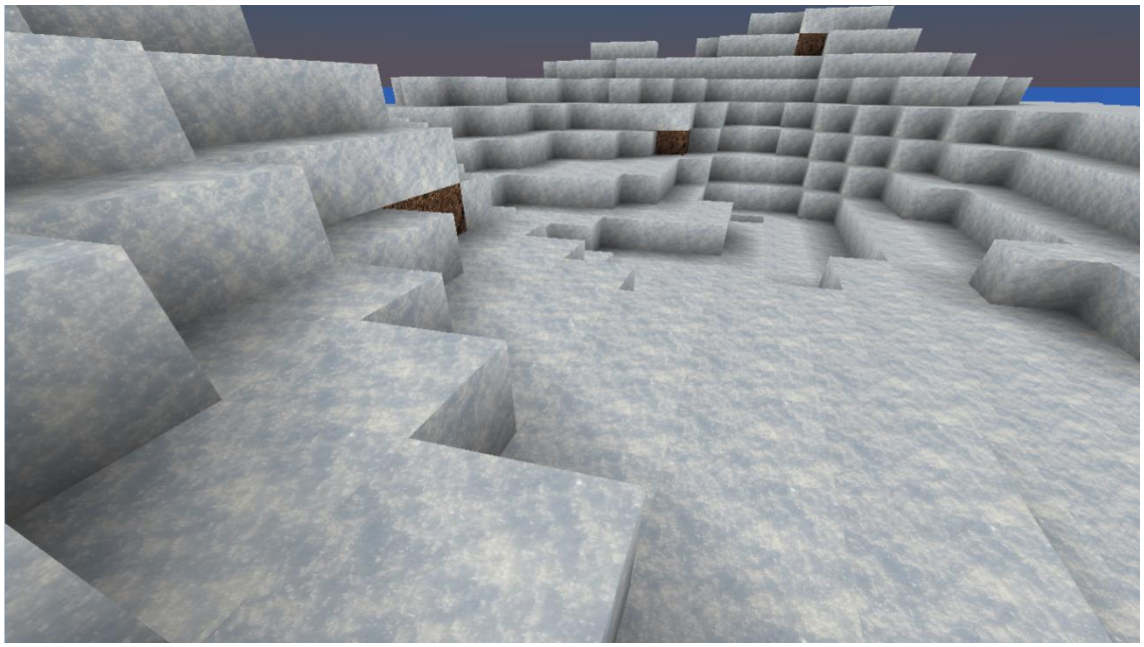


Figura 30 – Ambient Occlusion Activado

9.2. Mundos existentes

Figura 31 - Mapa uno: *Islands*

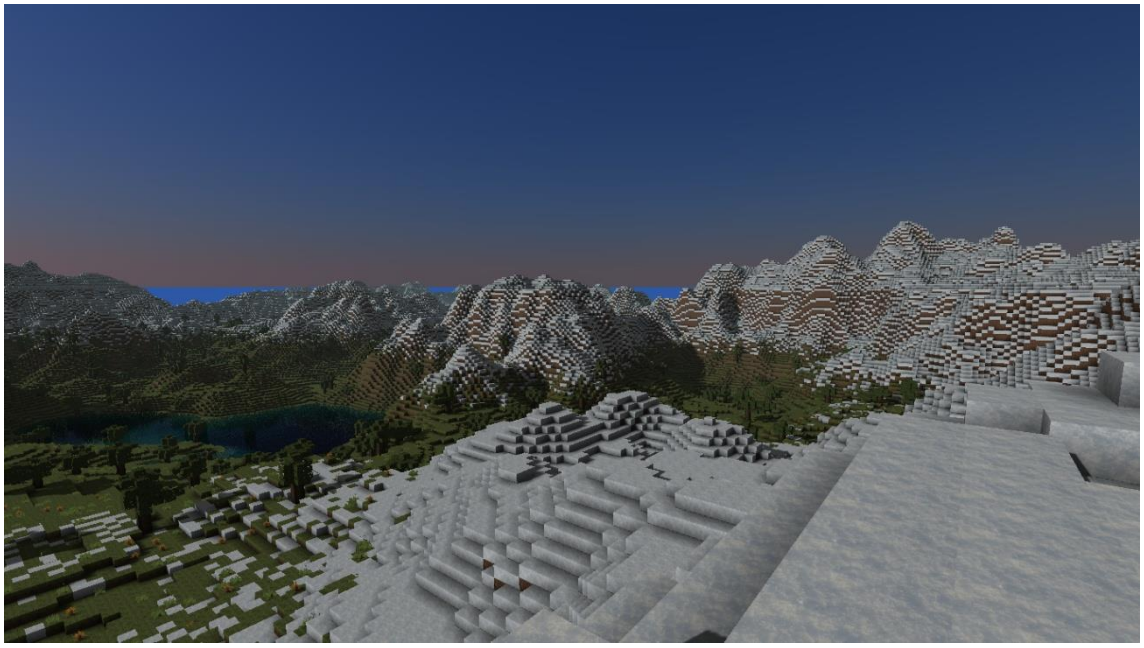


Figura 32 - Mapa dos: *Snowy Mountains*

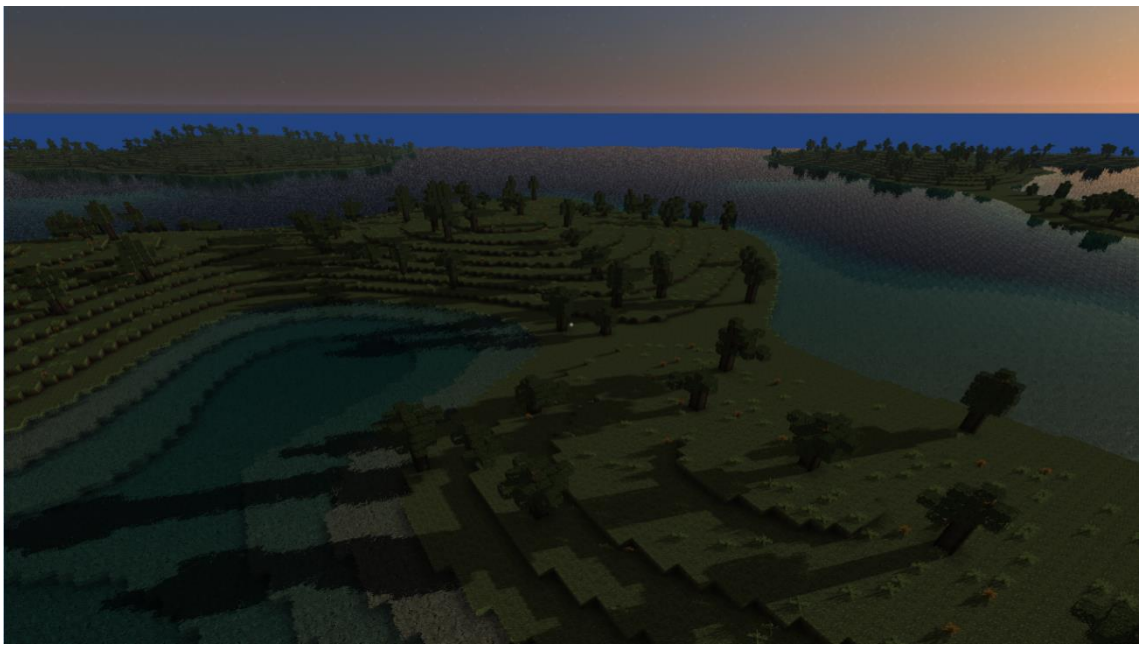


Figura 33 - Mapa tres: *Plains*

Figura 34 - Mapa cuatro: *Buggy Caves*

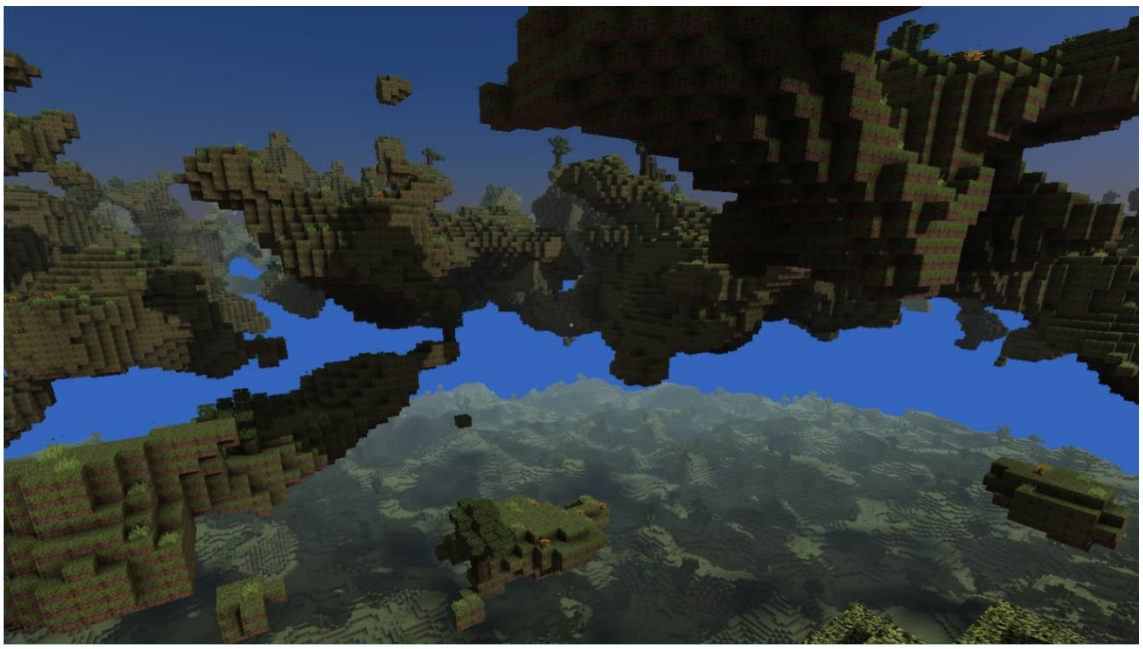


Figura 35 - Mapa cinco: *Floating Islands*

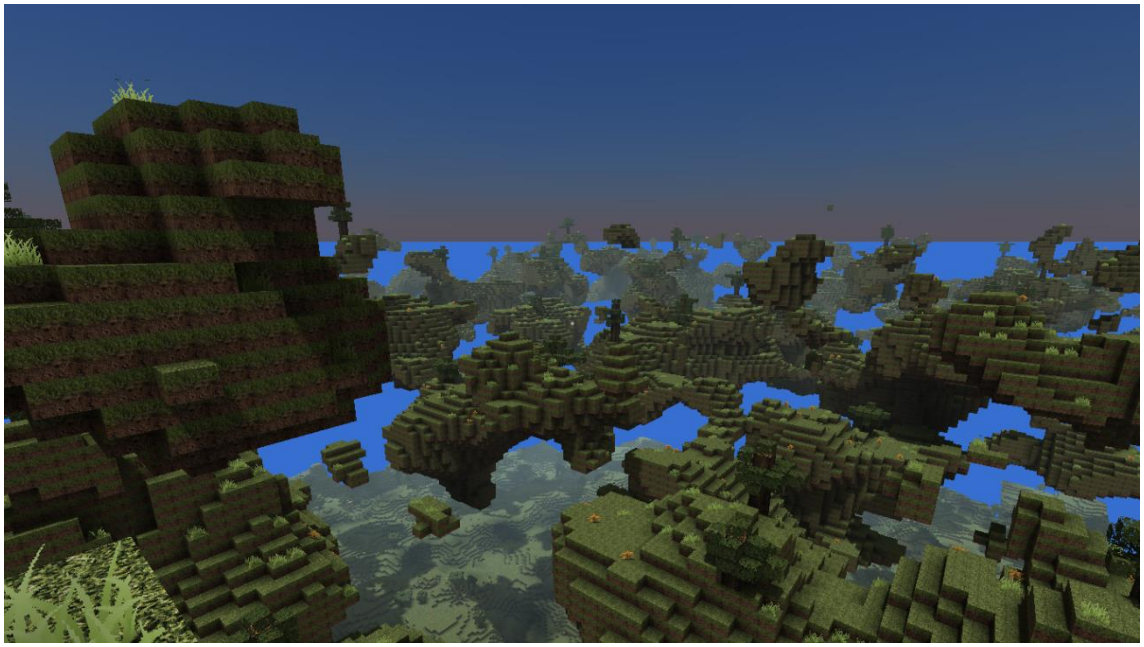


Figura 36- Mapa cinco: *Floating Islands*

Figura 37 - Mapa seis: *Underwater Ruins* (sobre el agua)

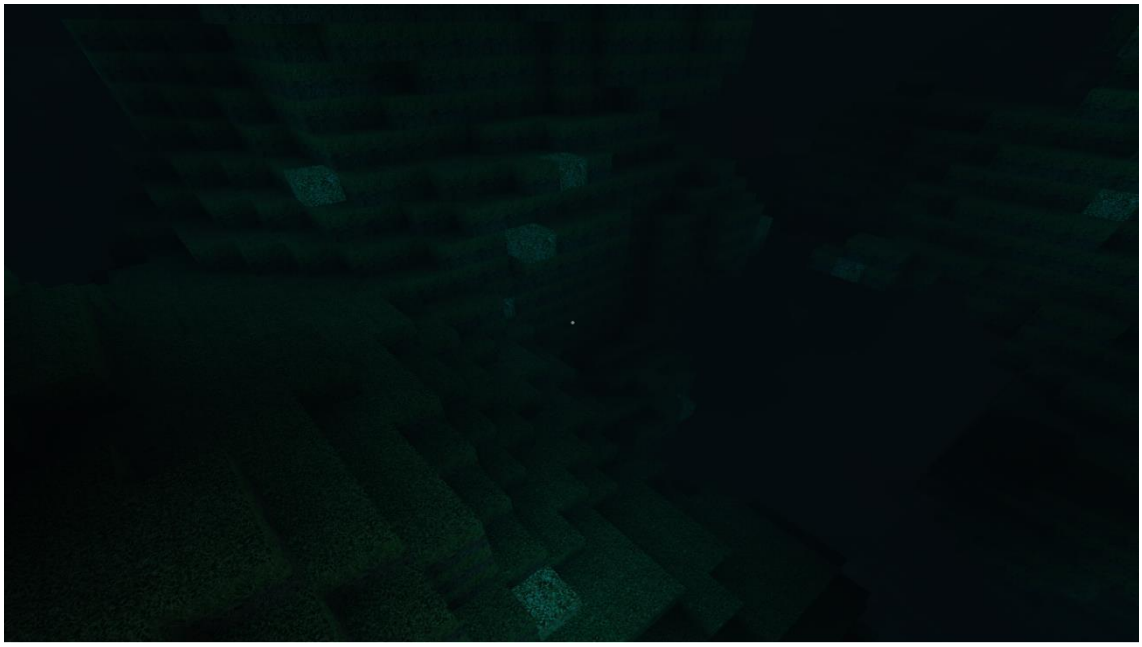


Figura 38 - Mapa seis: *Underwater Ruins* (bajo el agua)

9.3. Comparativa con Minecraft

Figura 39 - Minecraft: Planicies

Figura 40 - Kubex: Planicies

Figura 41 - Minecraft: Escena aérea

Figura 42 - Kubex: Escena aérea

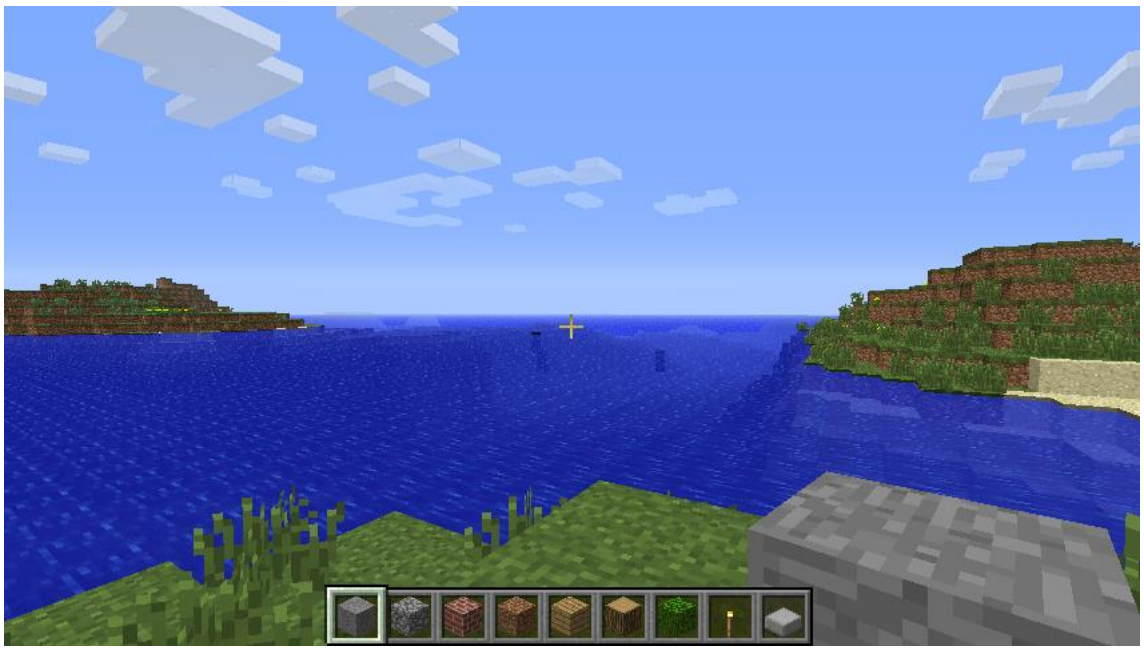


Figura 43 - Minecraft: Mar

Figura 44 - Kubex - Mar