

COMPUTACIÓN GRÁFICA AVANZADA

CURSO 2017

Obligatorio 2

Alumnos:

Marzio CUELLO - C.I: 4.805.879-0

Camilo SATUT - C.I: 4.496.575-1

Bruno SENA - C.I: 4.748.816-6

22/5/2017

Índice

1. Introducción	2
1.1. Definición del Problema	2
2. Análisis del Problema	2
3. Diseño de la Solución	2
3.1. Arquitectura	2
4. Implementación	3
4.1. Bibliotecas	3
4.2. Código Desarrollado por Terceros	3
4.3. Modelos	4
4.3.1. Modelos simples	4
4.3.2. Modelos con mapas de normales	4
4.3.3. Modelos con instanciación	5
4.4. Terreno	5
4.5. Iluminación	6
4.6. Agua	8
4.7. Lluvia	11
4.8. Pasto	12
4.9. Post-procesado	14
4.9.1. Fast Approximate Anti-Aliasing	14
4.9.2. Profundidad de campo	16
4.9.3. Corrección del color	16
4.9.4. Niebla	16
4.9.5. Viñeta	17
4.10. Desenfoque de movimiento	17
5. Concusión	18
6. Trabajo Futuro	18

1. Introducción

1.1. Definición del Problema

Se pidió desarrollar una aplicación gráfica que utilizara las capacidades del Pipeline Gráfico Programable. Se propuso desarrollar una escena de un bosque, enfocando la aplicación en el dibujado de la vegetación.

2. Análisis del Problema

El planteo inicial corresponde a, lo que es básicamente el desarrollo de un motor gráfico básico utilizando la API OpenGL, para facilitar el acceso y manipulación de los componentes de la escena se utilizarán técnicas de la programación orientada a objetos.

3. Diseño de la Solución

3.1. Arquitectura

Los límites tecnológicos (cantidad de memoria, velocidad de procesamiento) implican que muchos de los objetos cargados en la escena deban componerse de los mismos vértices. Por tanto, la escena se compondrá de entidades que poseen una matriz de transformación y un modelo. Muchas entidades podrán compartir el mismo modelo, solucionando el problema anterior. Sin embargo, por problemas tecnológicos (binding de shaders) se separará el dibujado de cada tipo de modelo, el siguiente diagrama ilustra lo anterior:

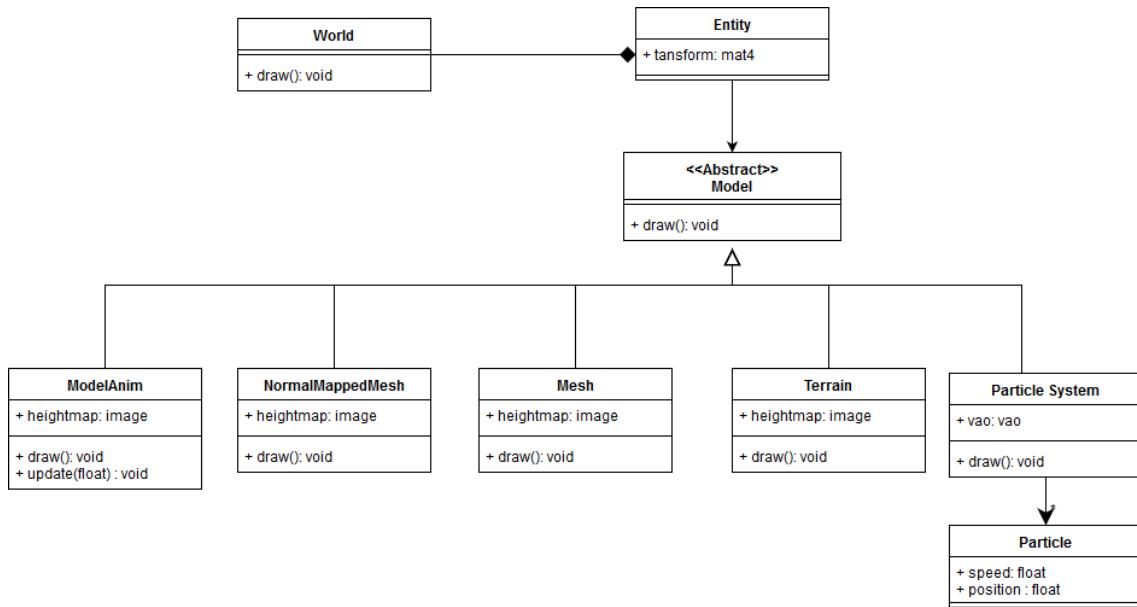


Figura 1: Diagrama UML de parte de la aplicación

4. Implementación

4.1. Bibliotecas

Para utilizar OpenGL se utilizó glew, que permite utilizar de forma más sencilla las funciones para el uso de shaders. Además se utilizó la biblioteca estándar de SDL 2.0, junto con ImGUI para la adición de menús de selección gráficos, GLM para el uso de funciones matemáticas y FreeImage para la carga de imágenes.

4.2. Código Desarrollado por Terceros

- Se adaptó código desarrollado por Karl Zylinski para generar animaciones. [1]
- Se adaptaron las implementaciones de la clase Particle y ParticleSystem desarrolladas por Usman Shahid para crear la lluvia. [3]
- Se tomó la idea de utilizar un geometry shader para convertir puntos en billboards para dibujar el pasto de Michal Bubnar, de su página "Megabyte Softworks". [4]

4.3. Modelos

4.3.1. Modelos simples

La implementación de los modelos simples (es decir, un conjunto de vértices con coordenadas de texturas y normales) se realizó de la misma manera que en el curso de Introducción a la Computación Gráfica con una diferencia. Dado que la optimización es crucial para las aplicaciones de tiempo real, se optó por utilizar Vertex Array Objects que no son más que una agrupación de buffers en memoria de video, estos buffers son escritos una única vez en VRAM, en vez de escribirlos una vez por fotograma calculado. Cabe destacar, que fue necesario re-indexar los modelos contenidos en los archivos Waveform pues se generaban artefactos extraños en iluminación y texturizado.

4.3.2. Modelos con mapas de normales

La técnica de mapa de normales es comúnmente utilizada en las aplicaciones de tiempo real ya que reducen drásticamente la cantidad de primitivas a dibujar y por tanto contribuyen a una mejora en rendimiento sin perjudicar la iluminación de los modelos aunque sí su geometría.

Consiste en realizar la lectura de un valor codificado de normales desde una textura en el fragment (o píxel) shader, y simplemente calcular la iluminación con este valor. Sin embargo, es necesaria una conversión, ya que las normales en la textura se encuentran en espacio tangente al modelo (es decir, en el plano del triángulo que lo compone). Es necesario, entonces, generar una matriz de transformación de espacios. Para convertir un valor de normal dado a espacio de mundo (es decir, al de la escena) es necesario calcular una matriz que se compone de “colgar” la normal original, la bitangente y la tangente a la superficie; para ello es necesario precomputar estos vectores en tiempo de carga y luego generar la matriz en el fragment shader.

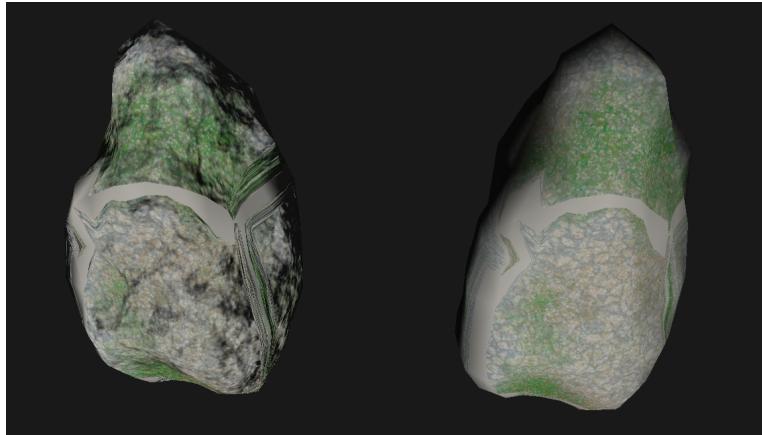


Figura 2: Comparación entre un modelo simple y uno con mapa de normales (ambos con los artefactos mencionados anteriormente)

4.3.3. Modelos con instanciación

En varias ocasiones es útil poder reutilizar la geometría de un modelo, cambiando únicamente su posición. En este trabajo se utilizó la capacidad de instanciar la geometría de OpenGL para, dado un modelo en una posición del mundo, poder dibujarlo en distintas posiciones con una única llamada a la API, mediante `glDrawArraysInstanced` (o bien `glDrawElementsInstanced`). Esto reduce el overhead que pueda producirse por hacer muchas llamadas a la API para dibujar una misma geometría, reduciendo la cantidad de datos que se pasan entre CPU y GPU, aligerando así el tiempo de renderizado. La utilidad que se le encontró a esta técnica en el proyecto fue poder colocar objetos que aparecían repetidas veces en el terreno en distintas posiciones. Se utiliza un archivo XML donde se detallan la cantidad de instancias deseadas y la posición relativa a la primer instancia donde se quieren dibujar las mismas.

4.4. Terreno

El terreno es generado a partir de un mapa de alturas (una imagen de intensidades que representan la variación de altura en cada punto), cada píxel del mapa de alturas se corresponderá con uno de los vértices que forman la malla de la superficie. En cuanto a las normales de estos vértices, se computarán como un promedio ponderado del vector director con origen en el vértice y fin en los vértices vecinos.

Además, es necesario generar una textura para el color difuso del terreno. Si bien es posible utilizar una única imagen para ello sucede que ésta deberá tener un tamaño muy grande para

percibirse con una calidad aceptable. Para solucionar el problema se utilizó tiling de texturas, es una técnica que consiste en que las coordenadas de lectura de las texturas pueden tener valores mayores a $(1, 1)$, simplemente se repite la textura. Nuevamente, esto genera terrenos con una buena calidad en la textura pero la monotonía de los tiles perjudica el resultado. Por ello, finalmente, se utilizó la técnica de mapa de mezclas. Consiste en utilizar una nueva textura, que cubra a todo el terreno. El cálculo del color se realizará mezclando cuatro texturas que utilizan tiling, y no es más que un promedio ponderado de los colores RGB y $1 - \text{RGB}$. Esto genera una escena más creíble que si solo se utilizara tiling.

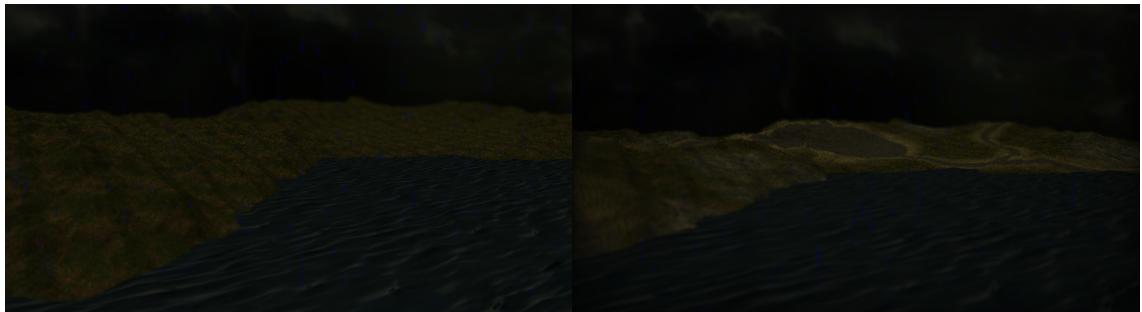


Figura 3: Terreno generado con mapa de alturas usando tiling y usando blendmapping respectivamente

4.5. Iluminación

Se cuenta con 2 fuentes de luz direccionales, una que imita al sol y otra a la luna.

Se cuenta además con una maquina de estados donde cada estado corresponde a un intervalo de tiempo en un día. Esta maquina de estados determina el comportamiento de ambas luces determinando dirección, intensidad y color, según el estado en que se encuentre.

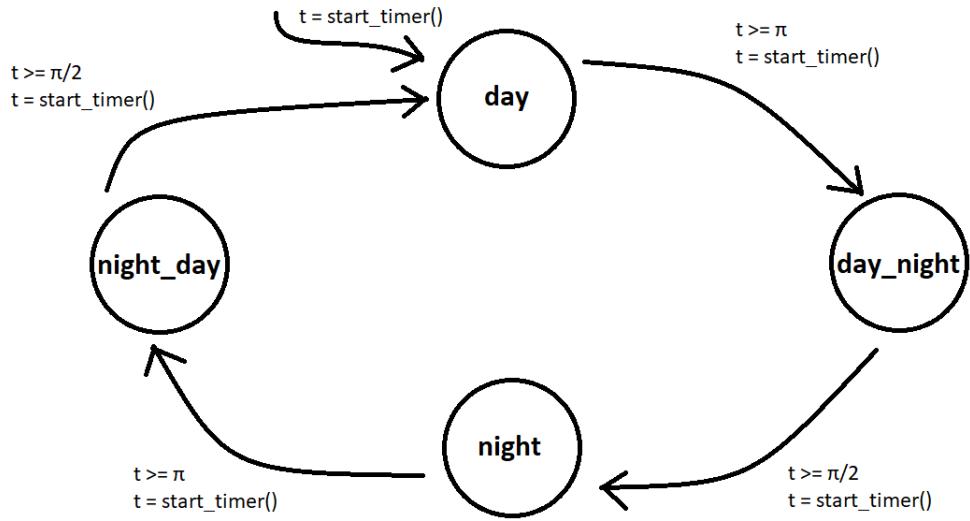


Figura 4: Maquina de estados para el ciclo día y noche

Si bien puede parecer que los valores de tiempo que inician los cambios de estado son anecdóticos, no fueron elegidos al azar. Para simular la trayectoria del sol, se decidió establecer una trayectoria que describiera una circunferencia en el plano xy y cómo las luces direccionales no tienen punto de origen o referencia, se tuvo que reflejar este comportamiento en la dirección de la luz. Es aquí que entran en juego los valores de tiempo, pues usando las propiedades del círculo trigonométrico es posible calcular la dirección en función del tiempo de la siguiente forma:

$$\vec{d} = (-\cos(t), -\sin(t), 0)$$

Y es de esta forma que se calcula la dirección en los estados de día y noche, notar que si t toma valores entre 0 y π el vector anterior describe media circunferencia en plano xy . Notar además que queremos que la dirección apunte al $\vec{0}$, por lo tanto cambiamos el signo.

Por otro lado, en el estado día, los valores de tiempo sirven para interpolar de forma lineal los colores de la luz. Eligiendo 2 colores, uno para el cenit a y otro para los horizontes b , se puede obtener el color de un cierto instante de tiempo mediante una combinación convexa de ambos:

$$\alpha(t) * a + (1 - \alpha(t)) * b$$

Donde $\alpha(t) = \sin^2(t)$ y $(1 - \alpha(t)) = \cos^2(t)$. Puede verse que cuando la luz esté más cerca del cenit ($\sin^2(t) \approx 1$), mayor peso tendrá el color a en la mezcla, mientras que en los horizontes

$(\cos^2(t) \approx 1)$ predomina b.

Los estados de day_night y night_day el tiempo se utiliza para calcular las intensidades de ambas luces, de modo de lograr una transición más suave:

$$\text{intensidad_sol} = \cos^2(t) \quad \text{intensidad_luna} = \sin^2(t) * 0,4$$

Notar que los anteriores valores son para el estado day_night, para night_day la intensidad del sol queda determinada por seno y la de la luna por coseno. Además la intensidad de la luna siempre va ponderada por un factor entre 0 y 1, de modo que en la noche la luz sea más tenue.

Por último, en los estados de transición entre día y noche se realiza un blending entre 2 texturas para el skybox, una para el día y otra para la noche. Dicha mezcla se ejecuta en el pixel shader del skybox y se calcula, de nuevo, como una combinación convexa, esta vez α es la intensidad del sol.

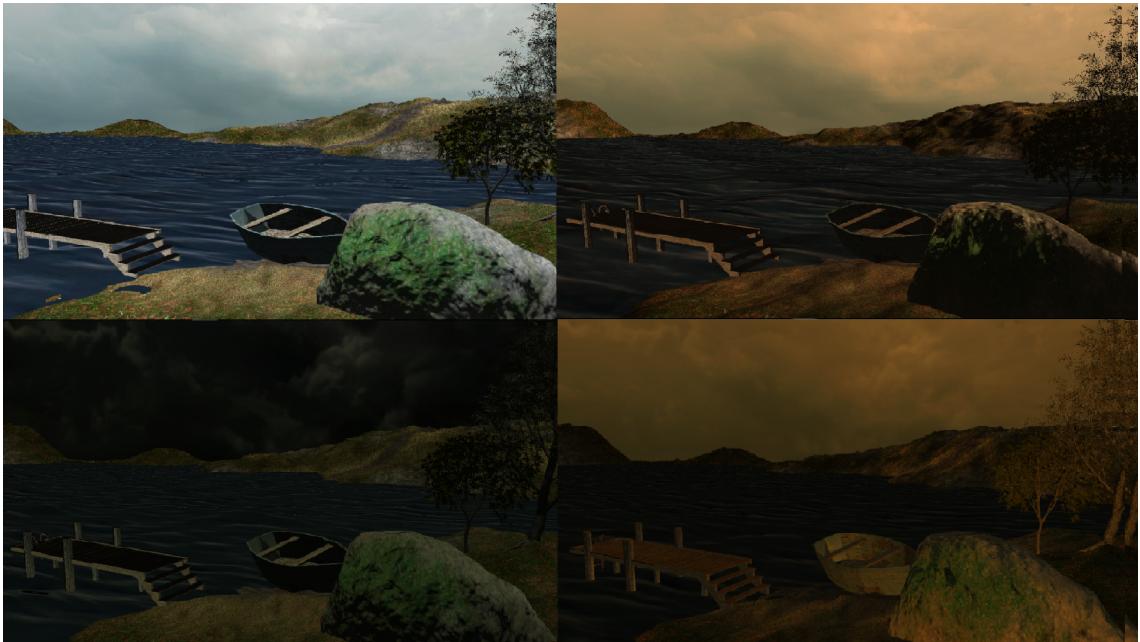


Figura 5: Imágenes de distintos tiempos del día

4.6. Agua

El agua consiste de un mesh de triángulos que forma un cuadrado. Dicho mesh se crea dándole una posición fija (x, y, z) a cada vértice.

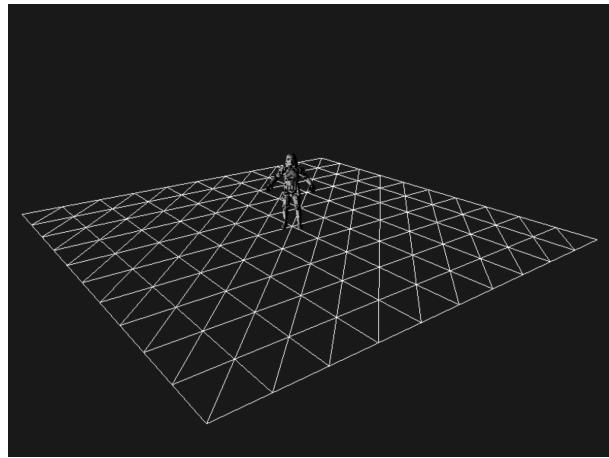


Figura 6: Mesh inicial con un modelo en el centro como referencia para el lector

De una forma similar a cómo se construye el terreno, las coordenadas en y finales de los vértices son determinadas por una función que se aplica en vertex shader. Dicha función tiene origen en las propiedades de la función seno y coseno que describen un movimiento ondulante.

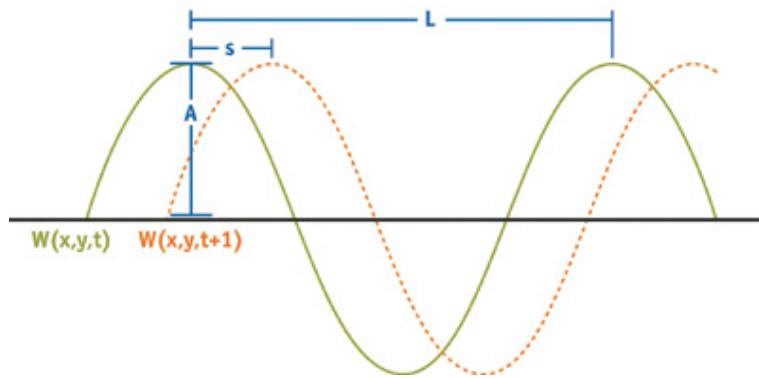


Figura 7: Parámetros de una onda sinusoidal

Dados los parámetros de una onda sinusoidal se puede definir la siguiente función que devuelve la altura de un vértice en base a sus coordenadas x y z , el tiempo y una amplitud, frecuencia, velocidad y dirección constantes:

$$y = W_i(x, z, t) = A_i \times \sin(D_i \cdot (x, z) \times w_i + t \times \phi_i)$$

Sin embargo, resulta evidente que una sola onda de este estilo representaría un movimiento ideal de fluido, algo que no ocurre en la realidad. Aquí entra en juego la propiedad de superposición

de ondas, una propiedad física que se cumple tanto en el medio líquido como en el aire. De esta forma podemos calcular la altura (coordenada en y) de los vértices del mesh, con la sumatoria de n ondas descritas por la ecuación antes mencionada.

$$y = \sum_{i=1}^n W_i(x, z, t)$$

Gracias a esto y eligiendo con astucia las constantes de cada onda, puede lograrse que las ondas del agua parezcan aleatorias.

La iluminación es uno de los aspectos más importantes a la hora de lograr un alto realismo, por lo tanto es necesario calcular correctamente las normales de cada vértice. Para esto también puede hacerse uso de una propiedad, esta vez, de matemática. Con lo descrito anteriormente, dado un instante de tiempo, se puede ver que se está trabajando con una función en R^3 , pues queremos los puntos que cumplen:

$$F_i(x, y, z) = y - W_i(x, z, t_j) = 0$$

Por lo tanto, para una sola onda la normal en el punto (x, y, z) es el gradiente de F evaluado en (x, z) , es decir:

$$\vec{N}_i = \nabla F_i = \left(-\frac{\delta W_i}{\delta x}, 1, -\frac{\delta W_i}{\delta z} \right)$$

Finalmente como el gradiente de la suma es la suma de los gradientes, la normal del punto luego aplicadas las n ondas es:

$$\vec{N} = \sum_{i=1}^n \nabla F_i$$

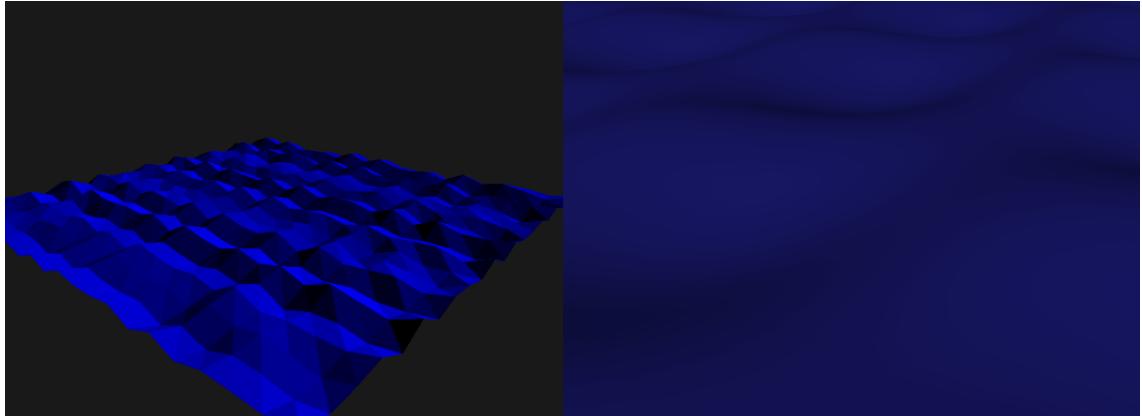


Figura 8: Normales por triangulo vs Normales calculadas

Si bien las ondas sinusoidales son una forma fácil y eficiente de modelar un fluido, su forma puede en ocasiones lograr efectos visualmente poco realistas, por lo que se decidió utilizar una variante de las mismas como se muestra en [2], logrando la siguiente forma:

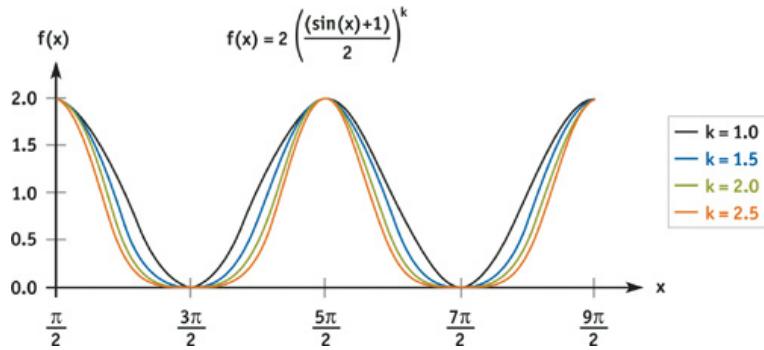


Figura 9: Nueva función para las ondas

Cabe mencionar que los cálculos de la normal para esta nueva función son análogos a los de la función anterior, puesto a que lo único que cambia son los W_i .

4.7. Lluvia

La lluvia se implementó como un sistema de partículas a partir de las clases antes mencionadas, por lo tanto, cada partícula posee posición, masa y velocidad. El sistema de partículas define una dirección constante que maraca la gravedad, de esta forma puede representarse un movimiento de caída libre. Para evitar generar partículas en todo el espacio, se crean solamente dentro de un cubo cuyo centro coincide con la posición de la cámara. De esta forma las partículas siguen al observador creando la ilusión de que hay partículas en toda la escena. Esto produce un mejor rendimiento sin comprometer el realismo de la escena. Por otro lado, la cantidad de partículas se mantiene constante, esto es, una vez inicializado el sistema de partículas, no se borran ni agregan más partículas. Una vez que una partícula sale del volumen antes mencionado, se actualiza su posición trasladándola al extremo opuesto del volumen.

Una vez que las partículas se mueven en la forma deseada es necesario determinar su geometría, se optó por dibujarlas como líneas de largo variable. Para hacer más eficiente el renderizado, en primera instancia, se dibujan las partículas como puntos y luego se utiliza el geometry shader para cambiar la primitiva a dibujar, emitiendo dos vértices por cada punto para lograr las líneas. Uno de los nuevos vértices es el punto original, mientras que el otro se dibuja en la misma posición con

un offset en sentido de la velocidad de la partícula, el offset depende de la masa de la partícula, por lo que una partícula de mayor masa tendrá una linea más larga.

Por último si bien las lineas que caen a alta velocidad son bastante convincentes, si la linea es suficientemente gruesa, la figura final parecerá un rectángulo y no una linea. Por otro lado, si las lineas son muy angostas, la figura será casi imperceptible. Para solucionar este problema se hizo uso de la función blend al momento de dibujar las partículas, dándole a uno de los vértices de cada linea transparencia total. Esto hace que la figura se mantenga visible con una apariencia más convincente.



Figura 10: Lineas sin blending vs lineas con blending

4.8. Pasto

Al generar el terreno, se guardan las posiciones donde se quiere que haya pastura. Para obtener estas posiciones, se lee el valor RGB de la textura que determina el terreno y, de la misma forma que se eligen los tiles para el tipo de terreno, se obtiene el la posición donde estará la unidad de pastura.

Las posiciones donde se quiere que haya pasto son enviadas a dibujar como GL_POINTS al vertex shader, el cual simplemente los procesa y se los manda al geometry shader que es donde se hace la verdadera magia.

Dada una posición para dibujar la pastura, y la posición de la cámara, el geometry shader se encarga de generar tres rectángulos (mediante triángulos claros, que es lo único que soporta) los cuales siempre están enfrentando a la cámara y están rotados levemente entre sí, y además

son deformados según la velocidad del viento, la cual es una constante predefinida. El fragment shader luego se encargará en su etapa de mapear las texturas de la pastura en cada uno de estos rectángulos.

Esta técnica es conocida como "Billboards", texturas que siempre miran a la cámara. Las siguientes imágenes y la técnica empleada, fue inspirada en el capítulo 7 de "GPU Gems" de Nvidia [5]

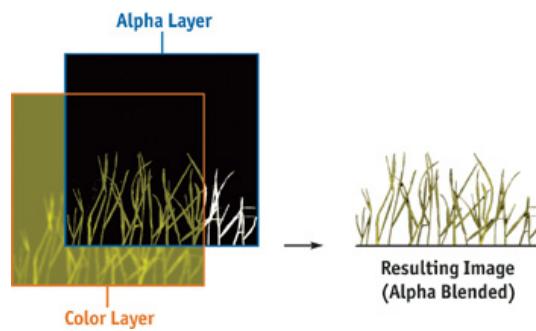


Figura 11: Texturas con alpha channel para hacer el blending entre las texturas.

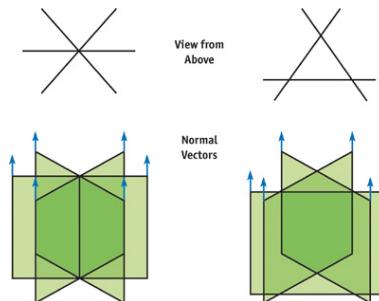


Figura 12: Los tres rectángulos generados en el punto donde irá la pastura.

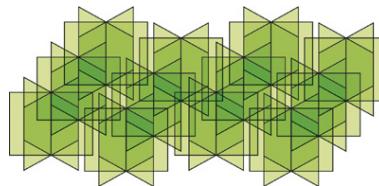


Figura 13: Al colocarlos uno al lado de otro, se aprovecha el alpha channel para dar aspecto de densidad en la pastura.

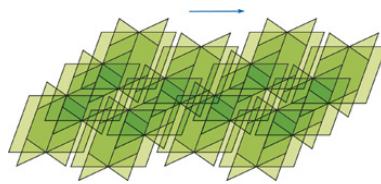


Figura 14: Se puede simular el efecto de ser movidos por el viento mediante un sesgo horizontal de los rectángulos generados.



Figura 15: Resultado final de dibujar la pastura mediante billboards en el terreno.

Esta técnica puede ser vista como una implementación del "Nivel de Detalle", ya que se procesan muchísimos menos datos que si se quisiera emplear un modelo 3D real de pastura u otro tipo de vegetación. Quitando el efecto del viento, puede ser utilizado para representar cualquier objeto estático. En este proyecto, se optó por usar el billboarding en la pastura, pero podría haber otro tipo de vegetación colocada a través de este método, por ejemplo árboles.

4.9. Post-procesado

Los efectos de post-procesado son muy utilizados en las aplicaciones de tiempo real, generan imágenes de mayor calidad que si no se utilizaran. Para implementarlos, se utiliza la capacidad de OpenGL de renderizar a texturas (conocida también como Frame Buffer Object). Luego, simplemente se dibuja un cuadrado en toda la pantalla, asignando coordenadas de textura que se correspondan con los píxeles finales.

4.9.1. Fast Approximate Anti-Aliasing

El uso de efectos de post-procesado implica un problema al utilizar métodos de anti-aliasing por múltiples muestras, ya que en realidad lo que se estará dibujando es el cuadrado mencionado

anteriormente. Por ello es responsabilidad de la aplicación el generar métodos de alisado de líneas, en este caso se optó por el uso de FXAA (Fast Approximate Anti-Aliasing). Consiste en un filtro que extrae de un píxel y cierta cantidad de vecinos adyacentes los valores de luminancia (valor que corresponde en escala de grises) y computar direcciones de muestreo basándose en ellos como se muestra en la figura siguiente. Luego, simplemente se promedian los colores obtenidos.

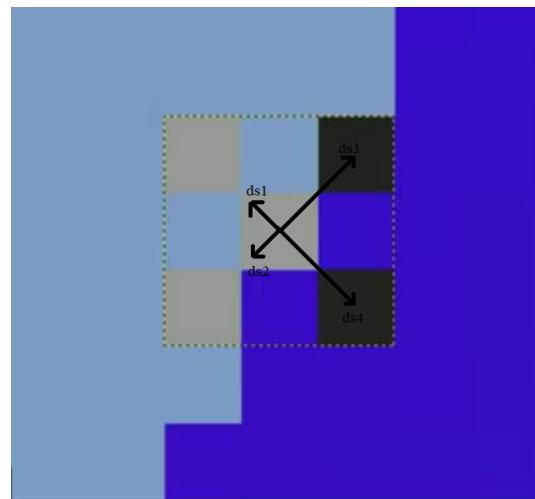


Figura 16: Se computan las direcciones de muestreo ponderadas por la diferencia entre los valores de luminocidad de los píxeles cercanos

Nótese que es un método aproximado, por lo que puede empobrecer la calidad de las texturas utilizadas si tiene betas o patrones particulares (pues se considerarán aristas y se las alisará).

La técnica genera resultados variados, aunque en el caso particular de esta aplicación mejora la calidad final de la imagen.

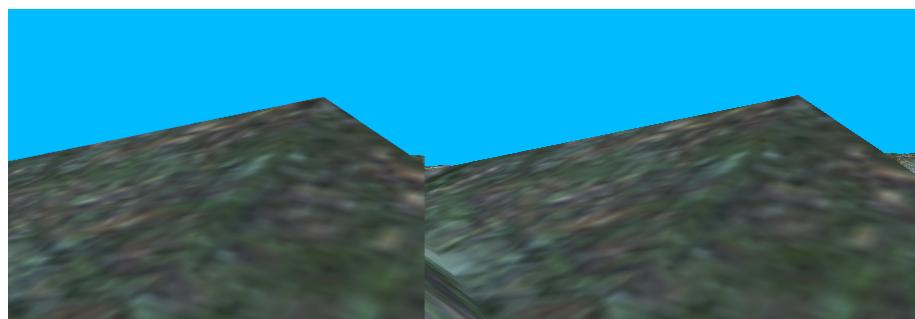


Figura 17: Dibujado con y sin FXAA respectivamente

4.9.2. Profundidad de campo

La profundidad de campo es un fenómeno que ocurre cuando la luz incidente atraviesa el lente de una cámara. En general, las aplicaciones de visualización de escenas lo implementan para agregar calidad al dibujado.

En este caso, su implementación se basó en el uso del Z-Buffer obtenido en el dibujado de la escena. Basta con realizar un filtro gaussiano de la imagen donde el radio del círculo dependerá directamente de la diferencia entre el valor de la profundidad en el centro de la pantalla y en el píxel procesado.

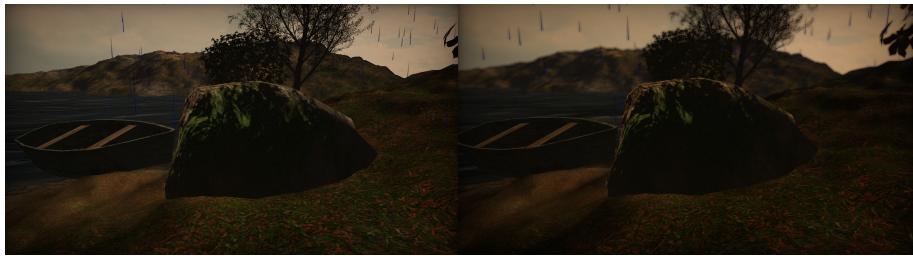


Figura 18: Dibujado sin y con profundidad de campo respectivamente

4.9.3. Corrección del color

Es necesario que los usuarios sean capaces de modificar los parámetros de brillo, contraste y gamma para la imagen producida. Su implementación es sencilla.

$$colorFinal = ((color + brillo) * contraste)^{\gamma}$$

4.9.4. Niebla

En este caso, se decidió implementar el efecto de niebla utilizando el post-procesado. Para añadir niebla al color final se obtiene la profundidad del píxel y luego se utiliza la siguiente función para calcular cuánto color se adicionará:

$$colorConNiebla = color + e^{-profundidad*modulacion}$$

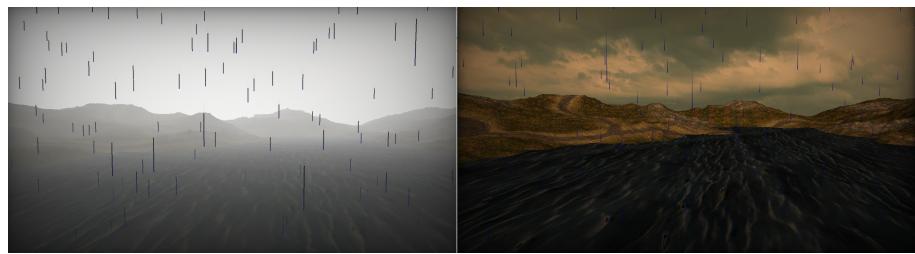


Figura 19: Dibujado con y sin niebla respectivamente

Donde la profundidad está linearizada (es decir, es un valor entre 0 y 1).

4.9.5. Viñeta

Otro de los ”desperfectos” de las cámaras fotográficas que es posible replicar es el de la viñeta, es básicamente, un halo negro en los bordes de la pantalla para integrar de manera más efectiva los colores.



Figura 20: Dibujado con y sin viñeta respectivamente

4.10. Desenfoque de movimiento

El desenfoque de movimiento también proviene de la fotografía debido a la velocidad de obturación y al movimiento que puede sufrir la cámara mientras los sensores se encuentran descubiertos. Para simularlo, se guardará copiará el FBO anterior a a una textura y luego se realiza el desenfoque combinando linealmente (según el movimiento de la cámara entre frames) los colores, es necesario aplicar difuminado al frame anterior.

5. Concusión

El obligatorio se trató de un primer acercamiento con los shaders, las dificultades se encontraron tanto en comprender su alcance a nivel conceptual, como es su uso en la práctica. Si bien se introdujeron algunos de los conceptos relacionados a los shaders en clase, el obligatorio requirió una cantidad considerable de tiempo destinada a la investigación, esto se debe a que el concepto de shader es demasiado amplio para cubrirlo todo en una sola clase. Además, GLSL ha cambiado mucho a lo largo del tiempo, muchas funciones y prácticas han sido deprecadas, por lo que fue necesario analizar varios ejemplos construidos con las últimas versiones.

Luego de este obligatorio el estudiante se encuentra en condiciones de pensar algoritmos e implementaciones en el contexto del uso de shaders. Si bien algunas etapas de shader no fueron utilizadas en el desarrollo de este obligatorio (por ejemplo tessellation shader), al finalizar este obligatorio, se cuenta con el enfoque mínimo necesario para pensar los objetos de una escena y su comportamiento, en base a sus shaders.

Los beneficios del uso de shaders son claros, al usar la GPU para realizar los cálculos varios vértices en paralelo, se logra un uso mucho más eficiente del poder de computo, esto se ve claramente en los tiempos de ejecución y la cantidad de marcos por segundos logrados sin la necesidad de implementar algoritmos de optimización, como por ejemplo frustum culling. Es por esto que los shaders resultan una excelente herramienta para el desarrollador gráfico, pues permite hacer uso del poder de la GPU sin la necesidad de entender el funcionamiento a bajo nivel del pipeline gráfico programable.

6. Trabajo Futuro

Las puntuaciones sobre las posibles extensiones de la aplicación son las siguientes:

- Implementar frustum culling para mejorar los tiempos de ejecución.
- Agregar SSAO (screen space ambient occlusion).
- Utilizar el Pipeline programable que ofrece OpenGL para la generación de mapas de sombras.
- Nivel de detalle adaptativo para el terreno, mediante el uso de técnicas de teselado, para aumentar la cantidad de polígonos del detalle del terreno y cambiar de texturas a medida

que se recorre el mismo.

Referencias

- [1] Implementación animaciones:

http://files.zylinski.se/skeletal_animation/

- [2] Paper sobre la simulación de agua:

http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch01.html

- [3] Implementación del sistema de partículas:

<https://github.com/usmanshahid/ParticleSystem>

- [4] Terrain Pt. 2 - Waving Grass:

<http://www.mbssoftworks.sk/index.php?page=tutorials&series=1&tutorial=32>

- [5] GPU Gems Chapter 7: Rendering Countless Blades of Waving Grass

http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch07.html