DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Final

Editor de terreno

Noviembre 2021 — Fundamentos de la computación gráfica

Integrante	LU	Correo electrónico
Miguel Fainstein		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Contents

1 Introducción **3**

2 Desarrollo **3**

2.1 Terreno 3

2.2 Shadow mapping 4

2.3 Proyección del mouse 5

3 Conclusiones **5**

1 Introducción

En este informe cuento un poco como hice la implementación del editor de terreno, las dificultades que fueron surgiendo, y posibles extensiones que me quede con ganas de hacer. Originalmente la idea era, dada una imagen en blanco y negro (como por ejemplo *perlin noise* o algún otro modelo para generar ruido) implementar un renderizador que permita visualizarla en 3d, para luego agregarle cuestiones visuales como texturas, iluminación, sombreado, etc. Sin embargo, la idea fue mutando y terminé implementando un editor de terreno, donde se puede alterar la elevación de este utilizando el mouse. Como todas las cosas que fui implementando tuve que aprenderlas sobre la marcha de forma muy iterativa, agregando funcionalidades y reimplementando secciones enteras de código, y mi dominio de *webgl* era muy limitado al principio (y sigue siendo); el código quedó bastante desprolijo. Intenté usar nombres lo más declarativos posibles, pero no llegué a hacer una refactorización apropiada del código. Además esto requeriría reescribir un montón de cosas y creo que no vale la pena. De todas formas pido perdón de antemano y espero que este informe ayude a aclarar lo que no se entienda del código.

No utilicé ninguna biblioteca externa, así que se debería poder correr de una sin tener que instalar ni descargar ningún requerimiento.

2 Desarrollo

2.1 Terreno

La grilla que conforma el terreno es de 64x64, o $(2*64)^2$ triángulos en el plano $y = 0$. La calculo dinámicamente simplemente porque esto me permitió durante el desarrollo modificar este tamaño y ver que andaba mejor, ya que modificar la densidad de triángulos disminuye o aumenta la resolución del terreno. El shader principal de renderización de terreno es el más extenso pero el más sencillo. Simplemente consiste en, dado un mapa de altura (textura en blanco y negro) calcular la altura de cada vértice. Además implementa, como en el último TP de la materia, el modelo de iluminación de Blinn-Phong pero sin componente especular (simplemente porque dado que estoy queriendo renderizar montañas y valles me pareció que iba a quedar mejor sin, y me ahorro un poco de complejidad en el código). Para las texturas, utilicé un método que se llama *triplanar mapping*, la idea es leer de la textura en base a las combinaciones posibles de sus coordenadas y pesarlas con la pendiente del vértice en función del terreno. Esto es porque de lo contrario, si todos los fragmentos leyeran de la textura en base por ejemplo a su posición *x, z*, en las secciones donde tenemos montañas o valles muy empinados, muchos fragmentos utilizarían los mismos valores produciendo un efecto de estiramiento de la textura. Entonces leemos la textura en base a $(x, z), (x, y), (y, z)$ y las pesamos en base las coordenadas de la normal. Dividimos las coordenadas de la normal por la suma total para que no se sature. Se puede ver en la figura 1 las diferencias entre métodos.

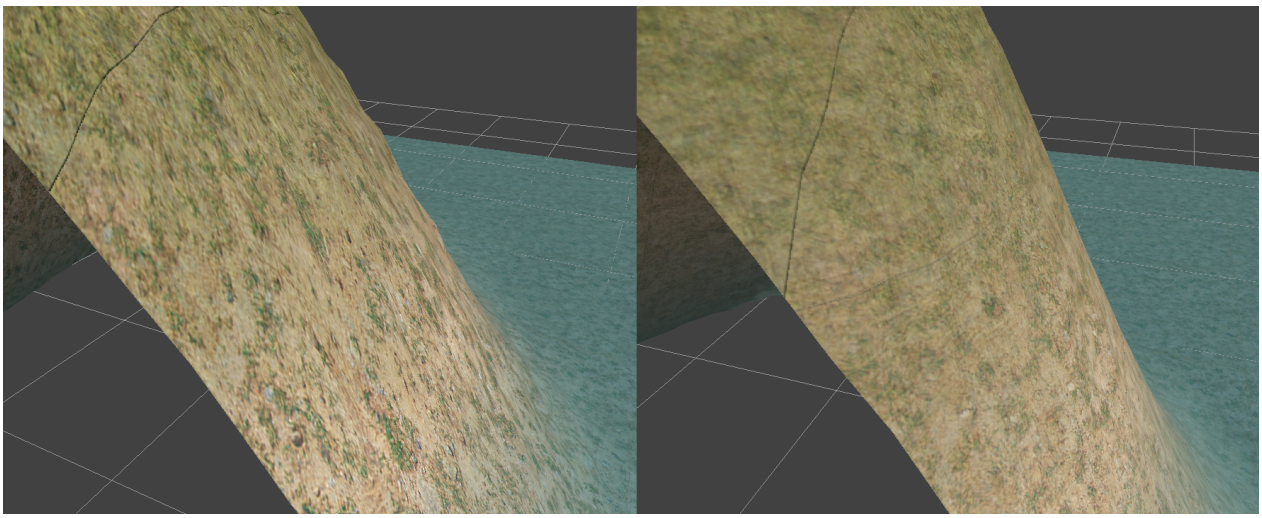

Figure 1: A la izquierda proyección de la textura usando plano (X, Z), a la derecha método de proyección tri-planar.

Es evidente que la textura sufre de ese "stretching" no deseado en el primer método, pero tal vez es más nítida que la segunda, donde ocurre un mayor blurreo debido a la suma entre los distintos mapeos. Esto sin embargo se podría solucionar siendo un poco más agresivo con los pesos.

Además quería que el terreno no tenga una textura constante, sino que dada la altura de cada fragmento se seleccione entre 4 posibles. Osea que si el fragmento tiene alturas bajas, utilice una textura de pasto por ejemplo, y si tiene alturas altas y se encuentra en una montaña utilice textura de nieve. Esto lo logré

armando texturas que tienen 4 cuadrantes con una textura en cada uno, y se calcula en el *vertex shader* el indice de sub-textura que le corresponde a cada fragmento.

Por último en el shader se implementa en base a una textura de sombras, la proyección de las mismas o *"Shadow mapping"* como se llama a este método. Voy a hablar más de esto en la siguiente sección.

2.2 Shadow mapping

El modelo de iluminación de Blinn-Phong está muy bueno por ser muy efectivo visualmente en relación a lo sencillo que es de implementar. Sin embargo, tiene sus limitaciones. Una de ellas es que los objetos no proyectan sombras sobre otros objetos o sobre si mismos, lo cual le puede quitar mucho dramatismo a una escena; y en especial cuando se quiere renderizar una montaña por ejemplo, la sombra ayuda muchísimo en enfatizar las dimensiones y las diferencias en altura. Podemos ver en figura 2 cuanta diferencia hace en comparación con el modelo de iluminación sin shadow mapping.

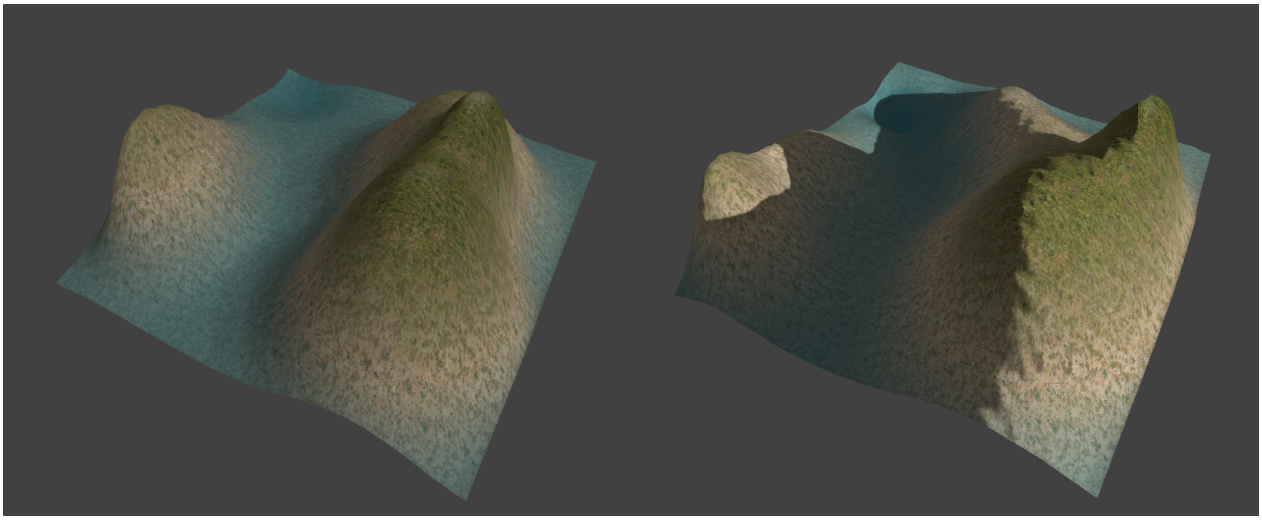

Figure 2: Escena similar, sin y con sombreado.

El método más común para implementar esto se llama *"shadow mapping"* o *"z-mapping"* y es bastante simple en concepto pero tiene sus sutilezas a la hora de implementarlo. Lo que vamos a hacer es renderizar a una textura la escena centrando la cámara en la posición de la fuente de luz; y pintaremos los fragmentos con su distancia en el eje Z a la fuente de luz. En la figura 3 podemos ver como queda esta textura.

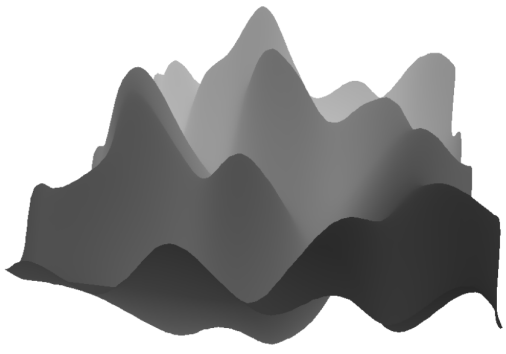

Figure 3: Mapa de profundidad desde posición de la fuente de luz.

Una vez que calculamos esta textura (que solo debe ser calculada una vez a menos que cambie la disposición de la malla de triángulos), lo único que debemos hacer es: dentro del shader que renderiza el terreno fijarse si la posición en la textura que le correspondería a cada fragmento tiene un color más oscuro que la distancia en el eje Z del fragmento con la fuente de luz. En cuyo caso significa que el fragmento esta siendo tapado y por lo tanto debe estar en sombra.

2.3 Proyección del mouse

En esta última sección cuento como implementé la proyección del cursor sobre la malla de triángulos para poder realizar la parte interactiva del editor. Resultó ser lo más complicado, porque al ser un problema medio especifico no encontré mucho material que me ayudase a hacer exactamente lo que quería, por lo que tuve que armar una solución propia que es medio rebuscada, pero tiene de ventaja que, a diferencia de algunas de las soluciones a problemas similares que encontré en internet, utiliza casi únicamente la GPU para las computaciones. Lo cual le da una muy buena performance.

Lo primero que hice fue, dadas las coordenadas del mouse sobre el canvas, calcular la dirección en el "espacio mundo" de este. Esto es porque como las coordenadas (x_m, y_m) del cursor viven en la proyección del mundo, en realidad no señalan a un solo punto (x_w, y_w, z_w), sino que a toda una recta que atraviesa el espacio mundo (con espacio mundo me refiero a las posiciones reales de los vértices, antes de hacer toda la conversión a espacio cámara, espacio homogéneo, etc.). Esta recta la podemos escribir como $\mathbf{d}t$ donde $\mathbf{d}$ es la dirección de la recta y t un escalar. Entonces yo lo que quiero es poder identificar todos los fragmentos que estén cerca de esta recta. Esto es lo que hago en el shader llamado "color pick". Para cada fragmento me fijo si existe algún t tal que $\|\mathbf{d}t - \mathbf{x}\| < \varepsilon$, donde $\mathbf{x}$ es la posición del fragmento en espacio mundo. Expandiendo la ecuación nos queda que $-\varepsilon^2 < at^2 + bt + c < \varepsilon^2$ donde $a = \mathbf{d}.\mathbf{d}$, $b = 2\mathbf{d} - \mathbf{x}$ y $c = \mathbf{x}.\mathbf{x}$.

Se puede ver entonces que, como a es siempre mayor a 0, no va a haber ningún t que cumpla esta condición si $\min_t\{at^2 + bt + c\} \geqslant \varepsilon^2$, y si derivamos e igualamos a 0 podemos ver que el mínimo se encuentra en $t_{min} = \frac{-b}{2a}$.

Luego renderizo la escena a una textura pero pintando cada fragmento de un color en base a su posición (x, z) en espacio mundo (para que no varíe cuando se mueve la cámara por ejemplo) si es que cumple la condición recién descrita, o sino lo pinto de azul. Una vez que hice esto, busco (usando CPU) en la textura a la que renderizamos el color correspondiente al fragmento seleccionado (si es que hay alguno).

Ahora viene la parte más polémica de la implementación, que visto en retrospectiva podría haber sido mucho más sencilla. Ya que obtuve la posición (x, z) del fragmento, tengo además la posición que le corresponde en el height-map, que uso para generar las alturas del terreno. Entonces dependiendo de si está queriendo "dibujar" o "borrar" debería modificar levemente los pixeles a cierta distancia del pixel correspondiente a la posición del mouse.

Esto lo quise hacer por CPU, pero resultó ser lentísimo en *Javascript*, pero ahora que estuve laburando bastante con *OpenCv* me doy cuenta que se podía resolver usando esta librería de forma muy sencilla y eficiente aplicandole una máscara a la textura. Pero bueno como no conocía esta solución, lo tuve que resolver usando la GPU, y manteniendo 2 texturas en simultaneo. Cada vez que se tiene que modificar el height-map se llama al shader que llamé "mask" donde se copia el último height-map y se le suma (o resta) un offset a cada fragmento, dependiendo de la distancia de este a la posición del cursor en el el mapa.

3 Conclusiones

Estoy bastante contento con como quedó el trabajo. Pero me quedé con ganas de agregar un montón de cosas, y más importante arreglar unas cuantas otras. La primera cosa que no me gusta como quedó es el hecho de que debido a como implemente las distintas partes, y en particular la parte del editor; este es **muy** dependiente de la posición estática de la cámara (se puede rotar, pero no realizar traslaciones) y del tamaño de la grilla del terreno (en espacio mundo va desde $(-1, -1, -1)$ a $(1, 1, 1)$ y no se puede agrandar). Y no son muy fáciles de solucionar sin reescribir bastante código. Por otro lado las sombras y la aplicación de texturas puede generar bastantes artifacts. Hay muchos que pude ir solucionando, pero otros quedaron.

Como cosas para agregar, me encantaría simular agua, y formar ríos y lagos; y agregar la parte de generación random con perlin noise como funcionalidad además del editor. Pero bueno quedaran para la próxima. Igualmente si vuelvo a intentar algo así, ni loco lo hago en *webgl* pelado. Aunque sea usaría una librería medio básica que abstraiga un poco de toda la burocracia que viene con los shaders, las texturas, los framebuffers, etc.