



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Implementación de un generador de terrenos utilizando Marching Cubes

Fundamentos de la Computación Gráfica
Segundo Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Martín Emiliano Lombardo	49/20	mlombardo9@gmail.com
Ignacio Ezequiel Vigilante	61/20	nachovigilante@gmail.com



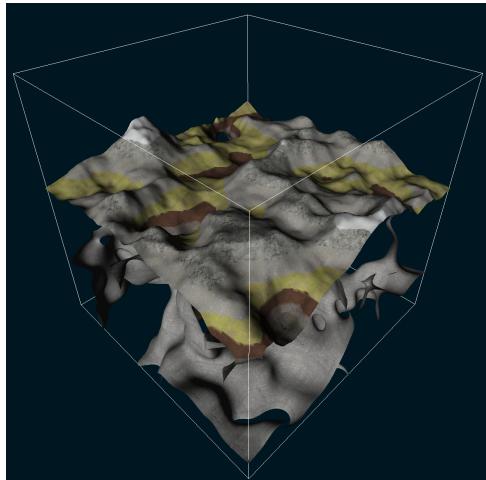
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
<http://www.fcen.uba.ar>

Resumen

Trabajo práctico final para fundamentos de la computación gráfica: generación de terrenos en 3D utilizando la técnica de Marching Cubes implementada en C++, compilada a WebAssembly. Renderizado utilizando WebGL y JavaScript.

Demo online en <https://tpfinal-fcg.netlify.app>.

Código fuente en <https://github.com/nachovigilante/TP-Final-FCG>.



Render final

Índice

1. Introducción	3
2. Generación de terreno	3
2.1. Generación de la elevación	3
2.2. Generación de las cuevas	4
2.3. Interpolación	4
2.4. Tipos de terreno	5
3. Algoritmo de Marching Cubes	5
4. Rendering	6
4.1. Chunks	6
4.2. Shaders	7
5. Conclusión	7

1. Introducción

El objetivo de este trabajo es utilizar el algoritmo de Marching Cubes para generar un terreno aleatorio que posea cuevas y "overhangs", a diferencia de otras formas de generación las cuales solo permiten utilizar ruido bidimensional e interpretarlo como un mapa de alturas. En conjunto a la generación del terreno a partir de una nube de puntos aleatoria, desarrollamos un renderizador para visualizarlo con una interfaz donde se encuentran los distintos parámetros para modificar la generación del mismo.

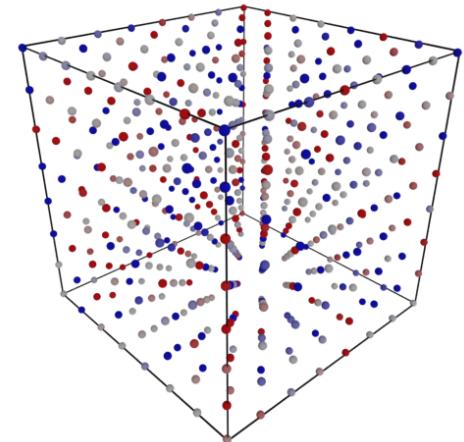
2. Generación de terreno

Antes de poder pensar en generar terreno, hay que definir el formato en que se va a representar. Utilizaremos el concepto de nube de puntos grillada, donde cada punto tendrá un valor en el intervalo $[-1, 1]$ y representa qué tan "relleno" está ese punto.

En la imagen a la derecha se puede observar un ejemplo, donde los puntos están separados de manera equidistante y su color indica su valor (de azul a rojo, qué tan lleno deberá estar cada punto).

Nuestro objetivo al generar terreno es establecer un valor para cada punto de la grilla y adjuntar ciertos datos para cada uno, como el color y la textura que le corresponderán al renderizarse.

Este es el formato que se utilizará en el [algoritmo de marching cubes](#).



La generación de terreno se separa en tres partes: la generación de elevación utilizando la misma técnica que con terrenos basados en heightmaps, la generación de cuevas y finalmente una interpolación entre ambas para el resultado final.

2.1. Generación de la elevación

Para generar la elevación del terreno utilizamos una variación de una técnica conocida: se obtiene una imagen en escala de grises donde cada píxel representa la altura del terreno en cada punto. Luego se genera una malla poligonal que levanta o extruye los puntos de un plano a la altura que corresponde en la imagen. En nuestra implementación, en vez de generar una imagen en 2D y extruir un plano, utilizamos la información de la altura para marcar en la grilla 3D todos los puntos por debajo de la altura como sólidos (1) y todos por encima como vacío (-1).

Para obtener los valores de las alturas que buscamos utilizamos ruido Perlin, en particular fBM ([Fractal Brownian Motion](#)) con parámetros elegidos a mano para obtener un resultado interesante.

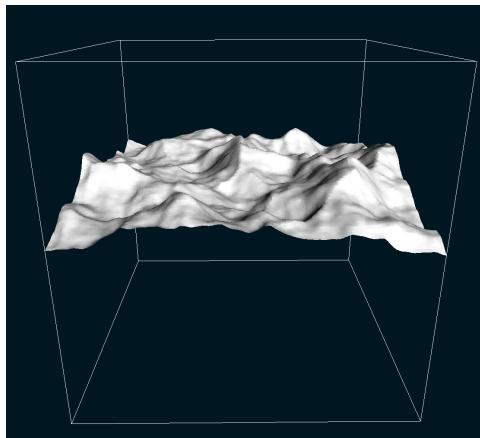


Figura 1: Render únicamente de la elevación

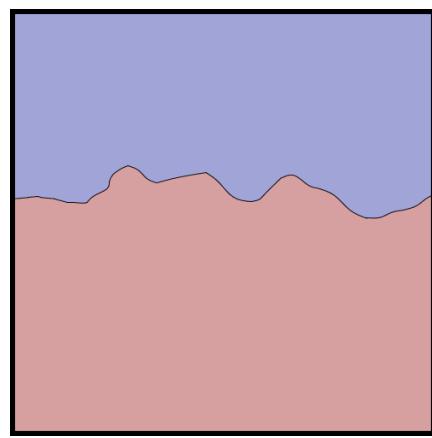


Figura 2: Ilustración en 2D (corte lateral de la versión 3D)

2.2. Generación de las cuevas

Para generar las cuevas utilizamos el mismo tipo de ruido que para la elevación pero en tres dimensiones. Para cada punto en la grilla en 3D obtenemos el valor del ruido directamente entre -1 y 1. Nuevamente ajustamos los valores a mano para encontrar un resultado que fuera satisfactorio.

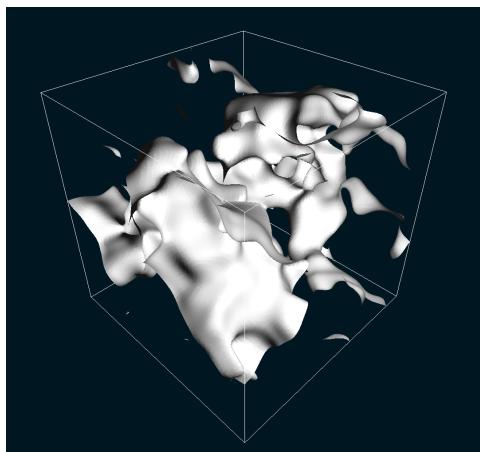


Figura 3: Render únicamente de las cuevas

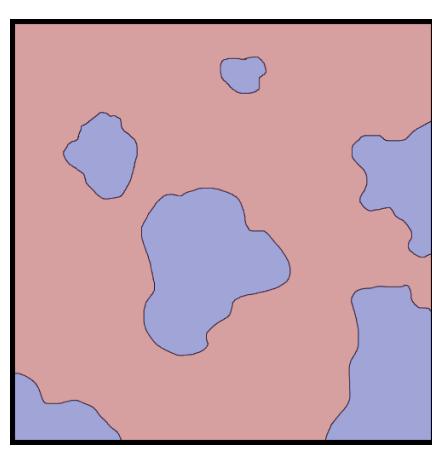


Figura 4: Ilustración en 2D (corte lateral de la versión 3D)

Aclaración: en las ilustraciones en 2D diferenciamos los colores como "sólido" y "vacío" utilizando un umbral (más adelante, el isolevel) para discriminarlos, pero en realidad los valores obtenidos por el ruido son suaves y no cambian abruptamente.

2.3. Interpolación

Ahora tenemos que combinar ambas generaciones manteniendo las cuevas por debajo y aire por encima de la elevación. Más precisamente, eliminar las partes sólidas de las cuevas por encima de la elevación y lograr que se mezclen de manera suave.

Para lograr esto, utilizamos la función [opSmoothUnion](#) creada por Inigo Quilez, que se utiliza para unir de manera suave superficies descriptas por funciones de distancia, pero también podemos aplicarla en nuestro caso.

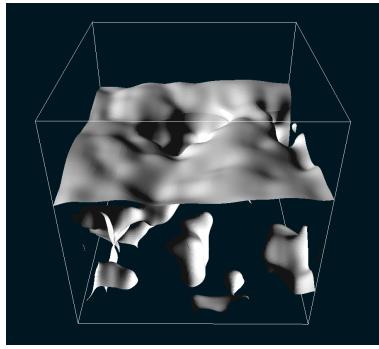


Figura 5: Render de la interpolación

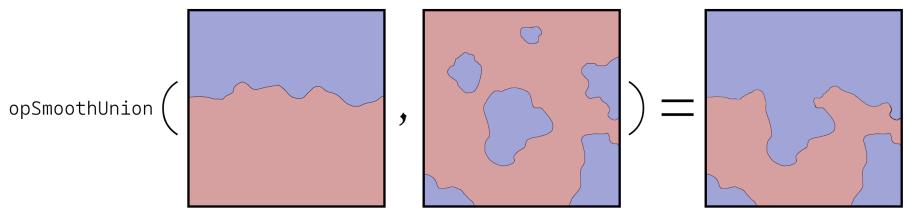


Figura 6: Ilustración en 2D de la interpolación

2.4. Tipos de terreno

Para poder diferenciar distintos tipos de terreno asignamos un índice a cada vértice. Este índice nos deja diferenciar entre terrenos como nieve, pasto, tierra, roca entre otros. Durante el rendering utilizamos esta información para dibujar texturas diferentes para cada terreno.

Para asignar el índice que corresponde a cada vértice utilizamos la altura del vértice, con umbrales elegidos a mano para cada tipo.

3. Algoritmo de Marching Cubes

La idea detrás de este algoritmo es, utilizando una nube de puntos grillada (en nuestro caso la generada previamente) y un valor arbitrario (llamado isolevel), generar una malla de triángulos donde los puntos que tengan un valor por encima del arbitrario elegido, estarán fuera de esta superficie, y los puntos con un valor por debajo estarán dentro. Para lograrlo, se itera toda la nube tomando de a 8 puntos que forman un cubo cada vez (de aquí el nombre "Marching cubes").

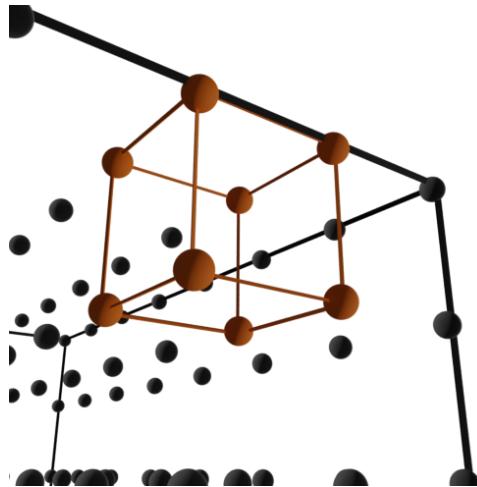


Figura 7: Iteración

En cada iteración es necesario decidir qué triángulos se generarán en ese cubo, y para este propósito el algoritmo cuenta con una lookup table con todas las configuraciones posibles de triángulos a generar en un único cubo.

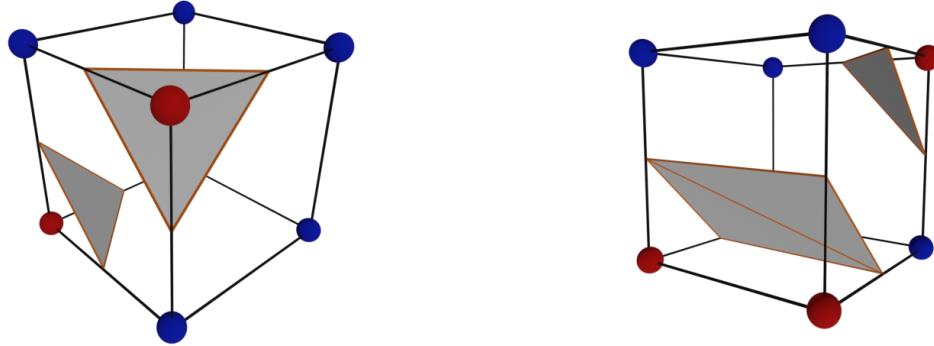


Figura 8: Ejemplos de configuraciones

Además de utilizar la configuración adecuada para cada cubo, es necesario interpolar la posición de cada vértice en base a los valores de los puntos, debido a que si esto no se realiza, los vértices de los triángulos siempre se posicionarían en la mitad de las aristas del cubo.

Para nuestra implementación, nos basamos en lo expuesto en el [artículo original de Paul Bourke](#) y realizamos una serie de modificaciones del mismo.

4. Rendering

4.1. Chunks

Para acelerar y simplificar la generación del terreno y su malla, dividimos el espacio de generación, que en teoría es infinito, en cubos (chunks) de 100 unidades de lado. Generamos una cantidad acotada de estos chunks, por defecto 5x5x5 chunks donde cada uno tiene una resolución de 25 vértices en cada dimensión. La generación de terreno y la malla se generan de a un chunk a la vez y van apareciendo a medida que se computan en otro thread.

Uno de los problemas de utilizar esta técnica es que genera artefactos gráficos o "seams" en las uniones de los chunks (las normales no toman información de los triángulos faltantes). Para remediar esto generamos una unidad más que el tamaño en cada dirección, pero no la incluimos en la malla final (para que no se solapen), sólo utilizamos la normal de los triángulos adicionales. Pese a nuestros esfuerzos, el efecto es minimizado pero no logramos quitarlo del todo.



Figura 9: Artefactos gráficos o "seams"



Figura 10: Artefactos gráficos o "seams" reducidos

4.2. Shaders

El vertex shader que utilizamos es simple, únicamente pasa las variables necesarias para el fragment shader, transformando los vectores a los espacios correspondientes para cada uno, aplicando diferentes transformaciones lineales. A diferencia de éste, el fragment shader es un poco más complejo. Para el renderer utilizamos un modelo de iluminación de Blinn-Phong con una luz estática direccional (sol) con respecto a la malla, por lo que debemos realizar todos los cálculos pertinentes para obtener tanto la componente difusa como la especular y la componente de ambiente; en este último shader.

Otra parte importante del proceso que se realiza en el fragment shader es el texturado, en el cual nos encontramos con una de las mayores desventajas de generar una malla de manera aleatoria: se vuelve imposible realizar el mapeo de las texturas por medio de un mapa UV. Por este motivo, si fuésemos a proyectar una textura de forma directa, ésta se estiraría y deformaría en muchas zonas.

Para solucionar este problema decidimos utilizar el mapeo triplanar de texturas. Esta forma de mapeo consiste en utilizar tres proyecciones de la textura (una por cada eje x, y, z) y realizar un promedio pesado de éstas utilizando las normales en cada punto. Así, logramos obtener una textura interpolada que sufre significativamente menos de deformaciones y estiramientos.



Figura 11: Mapeo en función de dos dimensiones



Figura 12: Mapeo triplanar

5. Conclusión

Este trabajo es únicamente una de las tantas aplicaciones posibles para el algoritmo de marching cubes. Éste, es una herramienta muy útil tanto para generación de terrenos donde queremos garantizar que existen varios puntos para distintas alturas (principalmente para videojuegos o animaciones), como para otras disciplinas como la medicina, el escaneo y representación de objetos en 3D. Por ejemplo, puede utilizarse para modelar en 3D editando la nube de puntos de manera interactiva, para generar mallas para objetos del mundo real a partir de nubes de puntos de scans de objetos en 3D, radiografías, funciones de distancia signadas (SDFs) y otros. Además de marching cubes existen otras técnicas similares para generar mallas a partir de nubes de puntos como marching tetrahedra, que utiliza una técnica parecida. Sin embargo, decidimos utilizar marching cubes por su simplicidad y por la cantidad de información disponible a la hora de buscar solución a posibles inconvenientes. Si bien la aplicación que se le da en este trabajo es con un fin visual, esta herramienta es utilizada en una amplia variedad de campos y nos resultó muy interesante experimentar con ella.