Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

Generación fractal de diversos tipos de terrenos basada en síntesis de ruido

Trabajo Especial de Grado presentado ante la Ilustre
Universidad Central de Venezuela
Por el Bachiller Juan Raúl Padrón Griffe
para optar al título de Licenciado en Computación

Tutor: Prof. Héctor Navarro

Caracas, 13 de Mayo de 2015

ACTA DEL VEREDICTO

Quienes suscriben, miembros del Jurado designado por el Consejo de la Escuela de Computación, para examinar el Trabajo Especial de Grado presentado por el **Br. Juan Raúl Padrón Griffe, C.I. 19.224.940,** titulado: **"Generación fractal de diversos tipos de terrenos basada en síntesis de ruido"** para obtar al título de de Licenciado en Computación, dejan constancia de lo siguiente:

Leído como fue, dicho trabajo por cada uno de los miembros del Jurado, se fijó el día 13 de Mayo de 2015 a las 15:00 horas, para que su autor lo defendiera en forma pública, lo que se hizo en el Centro de Computación Gráfica de la Escuela de Computación, en la Facultad de Ciencias de la Universidad Central de Venezuela, mediante una exposición oral de su contenido, luego de lo cual, respondió las preguntas formuladas por el Jurado y público en general. Finalizada la defensa pública del Trabajo Especial de Grado, el Jurado decidió aprobarlo.

En fe de lo cual se levanta la presente acta, en Caracas a los trece días del mes de Mayo de dos mil quince, dejándose también constancia de que actuó como Coordinador del Jurado, el Profesor Tutor Héctor Navarro.

Héctor Navarro, Tutor

Rhadamés Carmona

Eugenio Scalise

Agradecimientos

- A la Universidad Central de Venezuela, especialmente a la Escuela de Computación de la Facultad de Ciencias, por brindarme la oportunidad de formarme como Licenciado en Computación en esta casa de estudios, donde diferentes profesores me brindaron sus conocimientos y enseñanzas para ser profesional.
- Al Centro de Computación Gráfica y su cuerpo docente, por iniciarme en este apasionante campo, por proporcionarme los conocimientos y la colaboración necesaria para poder llevar a cabo este Trabajo Especial de Grado.
- A mi tutor, el Profesor Héctor Navarro, por brindarme la ayuda, los conocimientos y la orientación oportuna para poder elaborar este Trabajo Especial de Grado.
- Al Profesor Francisco Sans, por todos los consejos y apoyo prestados durante la elaboración de este trabajo.
- A mi cuñada Anaís Sánchez, por la ayuda y apoyo prestados en la revisión de la redacción de este documento.
- A mis padres, Beulah y Raúl, por ser mi ejemplo a seguir, por apoyarme, guiarme y motivarme en cada paso de mi vida. Sin ustedes, no hubiese sido posible.
- A mis hermanos y familiares, por su apoyo y cariño icondicional, especialmente en las situaciones dificiles.
- A mis amigos de la Universidad, por las experiencias y los momentos compartidos, que hicieron mi paso por esta casa de estudios una experiencia única.

Resumen

En las últimas décadas, los avances que se han desarrollado de los mundos virtuales han sido impresionantes y abismales, pasando de primitivos a visualmente complejos. Sin embargo, el proceso para constructirlos tradicionalmente se ha basado en el modelado manual, que es un proceso laborioso, repetitivo, tedioso y costoso, lo que lo hace una opción cada vez menos viable para satisfacer las expectativas de los usuarios. Una alternativa atractiva es el modelado procedimental, que construye el contenido por medio de un procedimiento o programa. Un campo activo de investigación basado en esta alternativa, es la generación automática de terrenos, específicamente la generación procedimental de diversos tipos de terrenos encontrados en la naturaleza o imaginarios, se presenta como un reto interesante e importante.

En el presente Trabajo Especial de Grado, se desarrolló un generador fractal de diversos tipos de terrenos, basado en una extensión de síntesis de ruido, que incorpora transformaciones las cuales permiten ampliar de forma significativa la capacidad y el potencial de la generación. Además, se implementó un visualizador tridimensional de mapas de altura adecuado para estudiar y explorar los mapas generados.

La propuesta planteada, construye diversos tipos de terrenos bajo un mismo enfoque, eficiente y extensible mediante la inclusión de nuevos algoritmos, funciones base y/o transformaciones. Adicionalmente, a través de las pruebas realizadas, fue posible estudiar y analizar detallamente los algoritmos, las funciones base y las transformaciones implementadas.

Palabras claves: modelado procedimental, generación de terrenos, fractales, síntesis de ruido, transformaciones, mapas de altura.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

Introducción

En las últimas décadas, los mundos virtuales han avanzado de forma impresionante de primitivos a visualmente avanzados y complejos. Sin embargo, el proceso para construirlos tradicionalmente se ha basado en el modelado manual, que se ha caracterizado por ser un proceso laborioso, repetitivo y tedioso. Además, la calidad del resultado depende principalmente de las habilidades del artista. Todos estos aspectos, son factores que dificultan y aumentan el costo para construir mundos virtuales mediante el modelado manual, presentando al mismo como una opción cada vez menos viable para satisfacer las expectativas de los usuarios.

Una alternativa atractiva para mejorar tanto la calidad como la velocidad de la construcción de mundos virtuales, es el modelado procedimental, el cual construye el contenido por medio de un procedimiento o programa. Este enfoque, tiene el potencial de reducir drásticamente el esfuerzo para modelar contenido, requiriendo poca intervención humana en la mayoría de los casos.

Un campo activo de investigación basado en esta alternativa, es la generación automática de terrenos, los cuales son utilizados en diversas aplicaciones tales como: planificación del uso de tierras, simuladores de vuelo, turismo virtual, películas y video juegos. En el mundo real, existe una diversidad impresionante de terrenos con formas y materiales distintos, desde cordilleras a través de cañones, hasta llanuras. A pesar de los avances en el campo, la generación procedimental de la diversidad de tipos de terrenos encontrados en la naturaleza o imaginarios, continua siendo un reto interesante e importante.

De acuerdo a lo planteado, se propone desarrollar una aplicación que genere diversos tipos de terrenos, junto con un Visualizador tridimensional de mapas de altura adecuado para estudiar y explorar los terrenos generados. El presente Trabajo Especial de Grado, aborda las principales técnicas para la generacion fractal de terrenos, centrándose en la extensión de síntesis de ruido propuesta por De Carpentier y Bidarra [5], con el propósito de implementar algoritmos, funciones base y transformaciones de manera tal de poder realizar pruebas sobre ellos.

El presente documento esta organizado en ocho capítulos. Los tres primeros,

dedicados exclusivamente al marco teórico, mientras que el resto se enfocan en los aspectos prácticos del Trabajo Especial de Grado.

En el capítulo 1 se presentan aspectos básicos, tales como una introducción a los fractales y las estructuras de datos para representar los terrenos, que sirven como base para entender el documento en su totalidad. El capítulo 2 presenta técnicas para la renderización de terrenos, relacionadas a la visualización y el texturizado, enfocándose más en los atributos proporcionados por los algoritmos de texturizado que en la eficiencia proporcionada por los algoritmos de visualización. El capítulo 3 presenta distintas técnicas basadas en la generación fractal de terrenos, con énfasis en la síntesis de ruido.

El capítulo 4 aborda los detalles relacionados al diseño de la implementación, como las herramientas de desarrollo utilizadas y el diagrama de clases de la aplicación. En los capítulos 5 y 6 se explica detalladamente la implementación del Visualizador y del Generador respectivamente. En el capítulo 7 se describen y analizan las pruebas elaboradas sobre la implementación. Finalmente, en el capítulo 8 se presentan las conclusiones derivadas del trabajo realizado y se proponen los trabajos futuros.

Capítulo 1

Terrenos

El presente capítulo, cubre una diversidad de tópicos que servirán como base para el entendimiento de los capítulos subsecuentes. Virtual Terrain Project [46] es un portal muy completo, donde se puede encontrar información valiosa y precisa relacionada con diferentes aspectos de los mundos virtuales (generación de terrenos artificiales, vegetación, renderización de mapas de altura, entre otros).

El capítulo se divide en cuatro secciones. La sección 1.1, es una introducción a los fractales, con interés especial en el movimiento browniano fracccional. En la sección 1.2, se presentan diferentes estructuras de datos para representar terrenos, indicando sus ventajas y desventajas. En la sección 1.3, se describe una clasificación general de las técnicas de generación de terrenos, indicando las fortalezas y debilidades. Finalmente, en la sección 1.4, se describe brevemente dos de los programas más populares en la actualidad para la generación de terrenos, destacando sus características principales.

1.1 Fractales

Mandelbrot [29] introduce la geometría fractal, que proporciona tanto una descripción como un modelo matemático para muchas de las formas complejas encontradas en la naturaleza. Formas irregulares tales como: costas, montañas y nubes; no se describen ni fácilmente ni precisamente mediante geometría Euclidiana. Mientras las formas Euclidianas son descritas normalmente mediante fórmulas algebraicas, los fractales son producto de un algoritmo o procedimiento.

Esta sección proporciona una introducción a la geometría fractal, desde una perspectiva adecuada para computación gráfica principalmente basada en el trabajo presentado por Ebert et al. [11]. Los fundamentos matemáticos son abarcados en detalle por Peitgen y Saupe en [37].

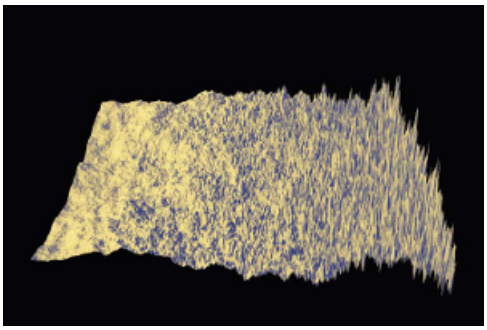


Figura 1.1: En este terreno la dimensión fractal varía de 2,0 (izquierda) a 3,0 (derecha). Imagen tomada de [11].

Musgrave [11] define un fractal como un objeto geométricamente complejo, donde la complejidad deriva de la repetición de una forma determinada sobre un rango de escalas (tamaños). Los fractales poseen dos propiedades particulares: la autosimilaridad *(self-similarity)* y la dimensión fractal. Un objeto es autosimilar cuando es idéntico o aproximadamente similar a las partes del mismo. Por lo tanto, su forma es invariante a la escala y en consecuencia preserva el detalle, a diferencia de las formas Euclidianas tradicionales, que son cada vez más lineales al trabajar con escalas más finas.

Las dimensiones Euclidianas son representadas por enteros: 0 dimensiones corresponde a un punto, 1 a una línea, 2 a un plano y 3 al espacio. La dimensión fractal extiende este concepto a números reales tales como 2,3; la parte entera de la dimensión fractal indica la dimensión Euclidiana subyacente (en este caso 2) y la parte fraccional denominada el incremento fractal (en este caso 0,3) define la complejidad. A medida que el incremento fractal aumenta, el fractal pasa de ocupar (localmente) la dimensión Euclidiana subyacente (en este caso un plano) a ocupar densamente alguna parte local de la próxima dimensión Euclidiana (en este caso el espacio). Intuitivamente, la dimensión fractal se puede concebir como una medida de la complejidad visual del fractal, donde al aumentar la dimensión fractal aumenta la complejidad visual del mismo. La figura 1.1 ilustra como la rugosidad de la superficie de un terreno varía con la dimensión fractal.

La autosimilaridad puede ser exacta o estadística. En la exacta, el fractal es idéntico en diferentes escalas, mientras que en la estadística preserva medidas numéricas o estadísticas. Un ejemplo de autosimilaridad exacta es el copo de nieve de Koch *(Koch snowflake)*, en la figura 1.2 se puede apreciar las primeras 5 etapas de la construcción del mismo. La dimensión fractal del copo de nieve de Koch es 1,26, la derivación es explicada en [37]. Entre los ejemplos de autosimaridad estadística se encuentran: helechos, costas y redes fluviales.

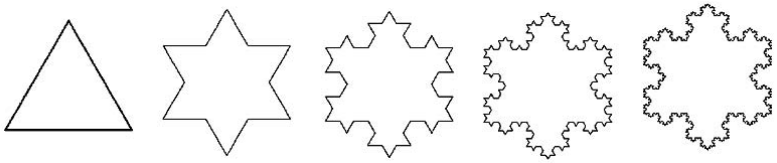


Figura 1.2: Las primeras 5 iteraciones del copo de nieve de Koch. Imágenes extraídas de [11].

Los fractales son caracterizados por una serie de parámetros: la función base, la dimensión fractal y la lacunaridad *(lacunarity)*. La función base es la forma subyacente repetida en el rango de escalas, los ruidos y ondas sinusoidales son ejemplos de funciones base. En el caso de la dimensión fractal, éste controla la rugosidad del fractal. Finalmente, la lacunaridad o resolución espacial no es más que la brecha entre las escalas que componen al fractal, es decir, por cuanto cambia la escala en cada iteración. Es importante destarcar, que en computación gráfica todos los fractales poseen un ancho de banda limitado (un rango de escalas limitado).

Uno de los modelos matemáticos más utilizados para fractales aleatorios encontrados en la naturaleza, es el movimiento browniano fraccional o movimiento browniano fractal *(fractional brownian motion)*, conocido como fBm por sus siglas en Inglés. fBm es una generalización del movimiento browniano.

El comportamiento del proceso es caracterizado por el exponente de Hurst H. Específicamente, cuando $H > \frac{1}{2}$ los incrementos están correlacionados positivamente y cuando $H < \frac{1}{2}$ negativamente. Cuando $H = \frac{1}{2}$, el proceso es un movimiento browniano donde los incrementos son independientes. La figura 1.3 muestra trazas del fBm para distintos valores de H.

Por otra parte, el espectro de potencia de fBm corresponde a una función $\frac{1}{f^{\beta}}$, donde f es la frecuencia y β es el exponente espectral. La relación matemática entre la dimensión fractal D_F, el exponente de Hurst H y el exponente espectral β en un fBm es:

$$D_F = D_E + 1 - H = D_E + \frac{3 - \beta}{2} \tag{1.1}$$

donde D_E es la dimensión Euclidiana, $0 < H < 1$, $D_E < D_F < D_E + 1$ y $1 < \beta < 3$. La explicación detallada de esta relación se puede encontrar en [37]. En la sección 3.1 se presentarán técnicas para generar fBm que asemejan terrenos, es importante considerar esta relación ya que las técnicas controlan la dimensión fractal mediante el parámetro H.

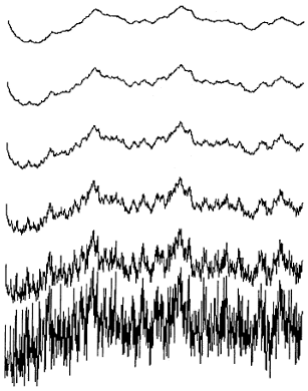


Figura 1.3: Trazas del fBm para H variando de 1,0 (arriba) a 0,0 (abajo) en incrementos de 0,2. Imagen tomada de [11].

1.2 Representación del terreno

Una de las decisiones más importantes al trabajar con terrenos virtuales es su representación. La selección de la estructura de datos influye en los algoritmos para generarlos, las características que pueden ser representadas y las herramientas disponibles para manipularlos.

En esta sección se analizan tres tipos de representación: el mapa de altura *(heightmap)*, la cuadrícula de vóxeles *(grid voxels)* y el mallado *(mesh)*. El análisis esta enfocado tanto en el potencial para la generación como para la visualización. Siendo la representación más popular en las aplicaciones, prácticamente todas las técnicas presentadas en este documento se basan en mapas de altura.

1.2.1 Mapa de altura

Un mapa de altura es una cuadrícula bidimensional donde cada casilla representa la altura en una localización. Los mapas de altura normalmente son almacenados como imágenes en escala de grises, donde cada píxel representa un valor de altura; colores oscuros representan elevaciones bajas mientras que colores brillantes representan elevaciones altas. La figura 1.4 ilustra un mapa de altura junto con la superficie del terreno que representa.

La estructura regular de los mapas de altura permite optimizar operaciones tales como: visualización, detección de colisiones, entre otros. La visualización de mapas de altura muy grandes en tiempo real es posible debido a los algoritmos basados en nivel de detalle, que serán explicados en mayor detalle en la sección 2.1. Además, al interpretar el mapa de altura como una imagen en escala

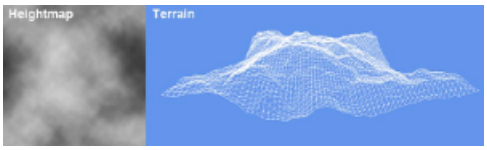


Figura 1.4: Mapa de altura (izquierda) y el terreno asociado (derecha). Imagen tomada de [24].

de grises, las técnicas de procesamiento digital de imágenes y visión artificial pueden ser utilizadas para construir, modificar, analizar y comprimir modelos de terrenos representados como mapas de altura. Por último, es importante acotar, que los Sistemas de Información Geográfica (SIG) utilizan mapas de altura para representar terrenos del mundo real; por lo tanto, no es de extrañar que exista una gran cantidad de terrenos disponibles para trabajar.

Por otra parte, los mapas de altura no pueden representar estructuras donde existan múltiples valores de altura para la misma localización; tales como: cuevas, superficies verticales, entre otros. Adicionalmente, los mapas de altura poseen una resolución uniforme y finita; en consecuencia, no existe una manera sencilla de manejar un terreno con distintos niveles de detalle locales.

1.2.2 Cuadrícula de vóxeles

Una cuadrícula de vóxeles es una cuadrícula tridimensional de píxeles volumétricos, seleccionando cuales vóxeles dibujar es posible construir formas tridimensionales arbitrarias. Esta cuadrícula permite representar cualquiera de las estructuras de terrenos tales como: cuevas, arcos naturales, entre otros.

Al igual que los mapas de altura, las cuadrículas de vóxeles poseen una resolución uniforme y finita pero operaciones como visualización y detección de colisiones consumen más memoria y tiempo de procesamiento. Sin técnicas de subdivisión espacial (tales como *octrees*), la cuadrícula de vóxeles normalmente desperdician mucha memoria (grandes porciones de la cuadrícula están vacías o muy profundas en el volumen).

1.2.3 Mallado

La superficie del terreno puede ser representada como un mallado arbitrario de primitivas 2D (usualmente polígonos) en el espacio 3D. Los mallados proporcionan la mayor flexibilidad para el modelado de terrenos, permitiendo representar superficies con geometría y topología arbitrarias. Además, soportan niveles de detalle variable, permitiendo más vértices en regiones con cambios pronunciados y relativamente pocos vértices en zonas planas; por ende, pueden

representar algunos modelos de terrenos más eficientes que las cuadrículas regulares.

La mayoría de herramientas de modelado y animación 3D están basadas en mallados, por lo tanto, hay una gran cantidad de herramientas disponibles para trabajar y una cantidad importante de usuarios acostumbrados a este paradigma. La gran limitación es la generación automática de mallados para representar terrenos, hasta donde conocemos no existen técnicas para este propósito.

1.3 Generación de terrenos

Las técnicas que producen terrenos, se pueden clasificar en tres categorías: basadas en medición, manuales y procedimentales. En esta sección serán descritas, considerando sus características principales: el realismo del resultado, el tiempo requerido, el control del proceso y la facilidad. Saunder [49] analiza estas categorías ponderando cada característica principal.

1.3.1 Técnicas basadas en medición

La medición, involucra la derivación de los datos de elevación, basados propiamente en las mediciones del mundo real que producen modelos de elevación digital; normalmente construidos a partir de técnicas de teledetección (detección remota) tales como imágenes satelitales y estudios topográficos.

Mapas de elevación de distintas partes del mundo (especialmente Estados Unidos) pueden ser descargados o solicitados de una variedad de proveedores de datos de Sistemas de Información Geográfica tales como: United States Geological Survey (USGS) [57] y Geo Community [22]. Los datos están disponibles en resoluciones de 30 y 60 metros por muestra, siendo los formatos más comunes DEM (Digital Elevation Model) y SDTS (Spatial Data Transfer Standard).

La principal ventaja de estas técnicas es producir terrenos altamente realistas con relativamente poco esfuerzo humano. La principal limitación es el control, ya que las aplicaciones están limitadas a los lugares y resoluciones disponibles en los repositorios.

1.3.2 Técnicas manuales

En la generación manual de terrenos, un artista modela su morfología manualmente, mediante programas de modelado 3D (Blender, Maya, entre otros), editores de imágenes (Photoshop, Gimp, entre otros) o editores de niveles especializados (CryENGINE Sandbox, UnrealEd, entre otros).

La principal fortaleza de la generación manual es el control prácticamente ilimitado del artista sobre el diseño y las características del terreno; esta a su vez, puede ser su principal desventaja, ya que modelar terrenos con características específicas requiere mucho tiempo y esfuerzo humano. Además, la calidad del resultado depende de las habilidades del artista, donde cada vez los ambientes virtuales son más extensos y detallados, por lo que el costo de generarlos manualmente es gradualmente menos viable.

1.3.3 Técnicas procedimentales

La generación procedimental abarca una amplia familia de técnicas que generan terrenos por medio de algún tipo de procedimiento o programa; estas técnicas tienen el potencial de reducir drásticamente el esfuerzo para modelar contenido, requiriendo poca intervención humana en la mayoría de los casos.

Smelik et al. [55] publican una revisión completa de técnicas procedimentales para generar características de los mundos virtuales: terrenos, vegetación, ríos, ciudades, entre otras; enfocada principalmente en el grado de control intuitivo y de interacción proporcionado por cada técnica. El artículo presenta una colección representativa de diversos enfoques para la generación procedimental de terrenos, que van desde los enfoques históricamente utilizados como fractales y simulación física hasta enfoques recientes como algoritmos evolutivos.

El presente documento esta centrado en la generación fractal de terrenos, que es el enfoque más popular encontrado en las aplicaciones, ya que las técnicas se caracterizan por ser eficientes, requerir poca intervención humana y ser relativamente fáciles de implementar. Por lo tanto, diversas técnicas de generación fractal serán explicadas en el capítulo 3.

La generación procedimental resalta por dos características esenciales: la amplificación de datos y la comprensión de datos; la primera, implica que un conjunto de parámetros o reglas sencillas produzcan una amplia variedad de modelos, y la segunda, se refiere a la posibilidad de representar modelos geométricos complejos de forma compacta mediante procedimientos y parámetros.

El control limitado proporcionado por la mayoría de las técnicas de este tipo, que requieren manipular reglas y parámetros complicados cuyos efectos son difíciles de predecir, representan su principal desventaja. Además, la mayoría de las técnicas, permiten modelar un rango limitado de tipos de terrenos.

1.4 Programas para la generación de terrenos

Esta sección pretende dar a conocer brevemente, dos de los programas más importantes para la generación de terrenos en la actualidad: Terragen y World Machine; proporcionando así, una idea general del tipo de programas disponibles para este propósito, destacando sus características principales con respecto a la generación y visualización de terrenos.

1.4.1 Terragen

Terragen [56] es un programa para Mac y Windows desarrollado para la visualización y animación de ambientes naturales realistas, desarrollado por Planetside Software; proporciona una versión de descarga gratuita, que puede ser utilizada únicamente con propósitos no comerciales, el resto de las versiones son comerciales. El renderizador es capaz de producir imágenes muy realistas incluyendo iluminación global, efectos atmosféricos, nubes, entre otros.

Este programa ha sido ampliamente utilizado en las industrias del cine y video juegos, permitiendo crear terrenos completamente procedimentales basados en síntesis de ruido que inclusive pueden abarcar un planeta entero o importar datos de terrenos del mundo real. Terragen posee su propio formato de mapa de altura TER, que es actualmente soportado por muchas aplicaciones, y permite importar terrenos de un rango amplio de formatos de imagen; así como exportar los mismos en forma de mallado o mapa de altura.

1.4.2 World Machine

World Machine [51] es un programa para Windows desarrollado por Stephen Schmitt, enfocado principalmente en la creación flexible de terrenos; incluye una versión básica de descarga gratuita, el resto de las versiones son comerciales.

World Machine proporciona generadores fractales poderosos, especialmente el generador de ruido de Perlin avanzado, que permite crear una variedad de tipos y estilos de terrenos. Además, ofrece herramientas de erosión para modelar terrenos que luzcan más naturales; proporciona una interfaz gráfica de usuario basada en una red o grafo, conectando una serie de operaciones que permiten modelar una gran variedad de combinaciones de efectos, y posee la opción de distintas vistas del terreno: la vista 3D en tiempo real, la vista de explorador y la vista de *layout*.

La vista $3D$ muestra la previsualización del terreno en el *viewport* actual. La vista de explorador es la visualización 3D en tiempo real del terreno, en donde se puede caminar, volar o manejar libremente sobre el terreno. La vista de *layout*

ofrece una navegación al estilo de Google Maps. World Machine permite importar y exportar mapas de altura de alta precisión en una variedad de formatos de imagen, así como exportar los mismos en forma mallado.

Capítulo 2

Renderización de terrenos

El componente clave de la renderización realista de escenas abiertas, es la renderización del terreno subyacente. Considerando que el terreno es representado mediante mapas de altura, la misma involucra la construcción de la superficie representada por el mapa de altura (visualización de terrenos o visualización tridimensional de mapas de altura), incluyendo el texturizado e iluminación de esta superficie.

La diversidad de materiales de los terrenos desde la perspectiva de colores, puede ser representada mediante texturas. La iluminación es vital en la percepción de formas, pero considerando que la iluminación realista no es un aspecto primordial del presente trabajo, se utilizará el modelo de iluminacion local de Phong [44].

Para una mayor comprensión, el capítulo se divide en tres secciones. En la sección 2.1 se presenta la visualización de terrenos de manera general e introductoria, tomando en cuenta que la visualización eficiente de terrenos extensos no es primordial para la investigación. En la sección 2.2 se presentan diferentes técnicas para el texturizado de terrenos. Las técnicas explicadas en esta sección tienen un enfoque procedimental, para el lector interesado Nicholson en [35] presenta una revisión concreta de técnicas para texturizado de terrenos con diversos enfoques. Por último, en la sección 2.3 se describe el problema del estiramiento de texturas y una técnica para solucionarlo.

2.1 Visualización de terrenos

La manera más sencilla de construir terrenos a partir de mapas de altura, es crear una cuadrícula con las mismas dimensiones del mapa de altura, donde cada valor del mapa de altura (píxel) determina la altura (coordenada y) del vértice correspondiente en la cuadrícula. A partir de esta cuadrícula, se construy-

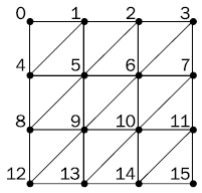


Figura 2.1: Triangulación de una cuadrícula bidimensional para visualización de terrenos de dimensiones 4×4. Imagen tomada de [24].

ye un mallado de triángulos para conectar los vértices basado en la triangulación mostrada en la figura 2.1.

En una cuadrícula, la distancia vertical y horizontal entre vértices vecinos es uniforme. La altura del terreno puede ser escalada normalizando los valores del mapa de altura y posteriormente multiplicando estos valores por un factor de escala. Una descripción detallada de este procedimiento denominado como visualización simple es presentada por Polack [45] para CPU y James [24] para GPU.

La visualización simple es un procedimiento sencillo que proporciona el mayor nivel de detalle posible, pero es computacionalmente prohibitivo para terrenos grandes. Por ejemplo, un mapa de altura de 4000×4000 píxeles constaría de 31.984.002 triángulos e inclusive para el hardware actual esta cantidad de triángulos tendría impacto sobre el rendimiento. Adicionalmente, los triángulos muy lejanos a la cámara serán más pequeños que un píxel, por lo tanto, generarían *aliasing* espacial, y debido a estas limitaciones, surgen los algoritmos basados en nivel de detalle que aprovechan la regularidad de los mapas de altura para aumentar la eficiencia. Estos algoritmos adaptan el nivel de detalle para optimizar la geometría en base a un criterio específico. El criterio más utilizado es la distancia a la cámara, donde regiones cercanas a la cámara poseen el nivel de detalle más alto y a medida que la distancia aumenta, progresivamente el nivel de detalle disminuye. La figura 2.2 ilustra el concepto.

Tomando en cuenta que el documento esta enfocado primordialmente a la generación fractal de terrenos, no se considera necesario un algoritmo basado en nivel de detalle complejo y potente, sino más bien uno sencillo y correcto. Por consiguiente, se asume este último como el algoritmo de visualización de terrenos, incluso a pesar de sus limitaciones, respecto al tamaño de los mapas de altura.

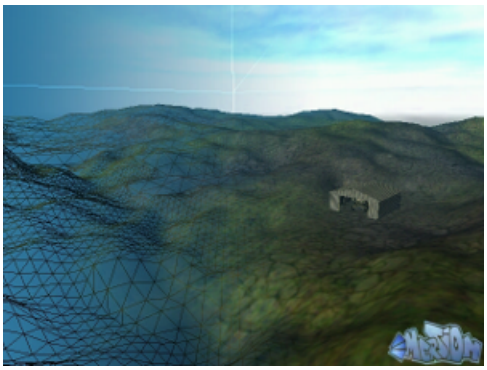


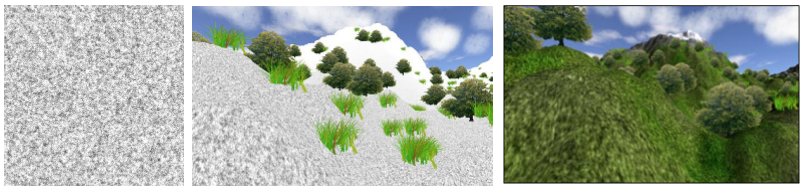
Figura 2.2: Terreno visualizado con la técnica Geomipmapping. Imagen tomada de [7].

2.2 Texturizado de terrenos

El texturizado de terrenos usualmente se hace mediante un método sencillo descrito por Polack [45] conocido como texturizado simple o global, estirando una textura determinada sobre todo el mapa de altura. Es decir, se aplica una proyección vertical ortográfica al mapa de altura (plano xz) sin considerar la altura de los vértices, donde se asignan como las coordenadas de textura s y t las correspondientes coordenadas geométricas x y z respectivamente. Este procedimiento para generar las coordenadas de textura es conocido como proyección planar simple. Es importante normalizar las coordenadas de la textura de manera tal que pertenezcan al rango $[0; 1]$.

Este enfoque puede presentar múltiples inconvenientes: la cantidad de memoria requerida, la dificultad para construir la textura y la carencia de detalle cuando el terreno es visto de cerca. Una alternativa para evitar aplicar texturas grandes conocida como *tiling*, consiste en la repetición de la textura a través del terreno (plano), sin huecos ni solapamientos. Esta alternativa soluciona el problema del tamaño de la textura, además mejora el resultado visual parcialmente, considerando que el terreno luce mejor pero aún carece de detalle cuando es visto de cerca; y por otra parte, al aumentar la repetición, también aumenta la periodicidad y la textura empieza a parpadear mientras se reduce en la distancia (fenómeno conocido como *strobing* [24]), mitigando el realismo.

James [24] describe una técnica para superar estos inconvenientes conocida como mapas de detalle *(detail maps)*. Un mapa de detalle es una imagen pequeña en escala de grises que contiene detalles tales como: bultos, grietas, rocas pequeñas, entre otros. La solución, consiste en mezclar la textura principal de poca repetición con un mapa de detalle de mayor repetición, mediante la multiplicación en regiones cercanas a la cámara, donde la textura principal aporta el color y el mapa de detalle las características. La figura 2.3 ilustra un mapa



(a) (b) (c)

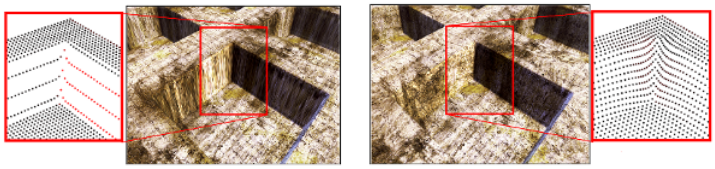
Figura 2.3: Texturizado incorporando mapas de detalle: (a) mapa de detalle, (b) mapa de detalle aplicado al terreno (c) mapa de detalle mezclado con la textura principal aplicado al terreno. Imágenes extraídas de [24].

de detalle aplicado a un terreno.

Bloom [3] introduce una técnica muy popular conocida como *texture splatting*, que mezcla un conjunto de texturas en el *framebuffer* para texturizar un terreno; la técnica dibuja el terreno múltiples veces. Cada pasada aplica una mezcla lineal entre la textura en cuestión y el *framebuffer* utilizando como peso el mapa alfa asociado a la textura (mezcla alfa), el resultado es almacenado en el *framebuffer*. Las texturas se mezclan de menor a mayor prioridad, considerando que una textura puede reemplazar la información de las texturas previamente mezcladas. Cada textura junto al peso de mezcla asociado es conocida como *splat*. Una explicación más clara e ilustrativa de la técnica se presenta en [19]; la principal limitación es el costo computacional de dibujar múltiples veces el terreno.

Nicholson [35] presenta una técnica para aplicar un conjunto de texturas de manera semi-automática, basado en un atributo proveniente del mapa de altura. El procedimiento en general comprende tres pasos: primero, se determina el atributo (calcularlo de ser necesario); segundo, se define un conjunto de texturas, cada una identificada por un rango de valores del atributo; y tercero, por cada vértice, se selecciona la textura basada en el atributo y se aplica un texturizado simple con esta textura. Para evitar las transiciones notables, Nicholson propone aplicar una mezcla lineal en la frontera entre las texturas involucradas, utilizando como peso el valor del atributo. El autor describe la técnica para dos atributos: altura y pendiente.

Ferraris y Gatzidis [13] plantean el texturizado basado en reglas *(rule-based texturing)*, una técnica derivada de *texture splatting* que agrupa varias características de las técnicas descritas en esta sección. Dado un conjunto de texturas y un conjunto de mapas de detalle asociado a los *splat*, la técnica aplica *texture splatting* de forma automática basado en un conjunto de reglas (reglas de color). La primera parte de las reglas de color (reglas de elevación), crea un *splat* por cada conjunto de alturas; a cada *splat* le asigna una textura, un mapa de detalle



(a) (b)

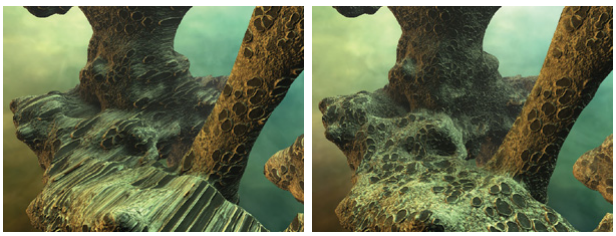
Figura 2.4: Problema de estiramiento de texturas, resaltando la resolución de las coordenadas de textura: (a) proyección planar simple, (b) mapeado correcto. Imágenes editadas de [31].

y un peso. La segunda parte de la regla (reglas de ángulo), son un conjunto de reglas adicionales que pueden modificar la información en un *splat* de acuerdo a la pendiente del vértice. Por ejemplo, si la pendiente es mayor a 30°, se puede mezclar el splat en cuestión con el *splat* más cercano.

2.3 Estiramiento de texturas

La proyección planar simple funciona correctamente desde un ángulo particular, pero genera distorsión desde otros. Al aplicar la proyección en terrenos con pendientes pronunciadas se producen artefactos como ilustra la figura 2.4. En esta la textura luce estirada y por esta razón el problema es conocido como estiramiento de texturas. La distorsión surge por la baja resolución de las coordenadas de textura en la zona afectada que conlleva a la repetición de las muestras.

Geiss [18] plantea una solución sencilla conocida como texturizado triplanar *(triplanar texturing)*, que considera tres proyecciones planares por cada uno de los ejes principales (x, y, z). En cada punto de la superficie aplica la proyección planar que genere la menor distorsión posible, tomando en cuenta que es necesario mezclar proyecciones en algunas regiones para evitar transiciones notables. Por ejemplo, si la normal de la superficie apunta principalmente en dirección $+x$ o $-x$, la menor distorsión sería la proyección planar al plano yz. Por lo tanto, Geiss propone aplicar las tres proyecciones planares asociadas a cada eje principal y mezclar las tres muestras basado en la normal de la superficie mediante una suma ponderada. La figura 2.5 ilustra el texturizado triplanar y como efectivamente elimina la distorsión por lo menos de manera visual.



(a) (b)

Figura 2.5: Comparación entre: (a) proyección planar simple, (b) texturizado triplanar. Imágenes extraídas de [18].

Capítulo 3

Generación fractal de terrenos

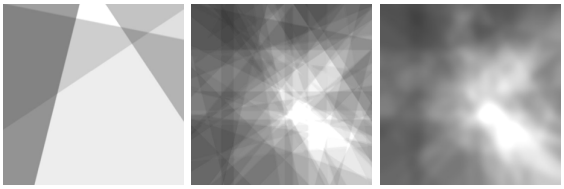
La base para la renderización de terrenos son los datos de elevación, representando la forma y la apariencia del terreno. La generación fractal constituye el enfoque más popular para la generación procedimental de terrenos, los fractales fueron presentados en la sección 1.1 a modo de introducción.

El capítulo estará divido en cuatro secciones. En la sección 3.1, se presentan los enfoques históricamente utilizados para generar terrenos, siendo el último el de mayor interés para el presente documento. Estos enfoques producen terrenos homogéneos que carecen de realismo, por ende surgen los modelos multifractales presentados en la sección 3.2, que generan terrenos heterogéneos con llanuras, piedemontes y montañas alpinas dentadas.

En la sección 3.3, se presentan trabajos más recientes orientados a proporcionar mayor interación y control intuitivo. Finalmente, en la sección 3.4 se presentan técnicas para construir ruidos, los cuales son utilizados como función base por el enfoque síntesis de ruido descrito en la sección 3.1.3.

3.1 Enfoques para la generación fractal de terrenos

Como se reporta en [29], el origen de las montañas basadas en fractales en computación gráfica, ocurrió cuando Mandelbrot reconoció la similitud entre la traza de un fBm en una dimensión y el horizonte de una cordillera con picos pronunciados; percatándose que si extendía este proceso a dos dimensiones, la superficie resultante debería ser similar a una escena montañosa.



(a) (b) (c)

Figura 3.1: Mapa de altura generado con el algoritmo de formación de fallas: (a) 4 iteraciones, (b) 64 iteraciones, (c) 64 iteraciones y filtro pasa bajo. Imágenes extraídas de [8].

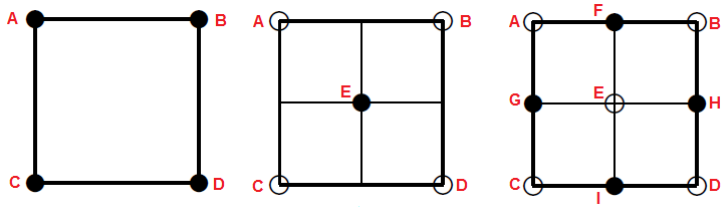
Los enfoques históricamente más utilizados para construir estas superficies son: formación de fallas basadas en la distribución de Poisson *(Poisson faulting)*, desplazamiento del punto medio *(midpoint displacement)* y síntesis de ruido *(noise synthesis)*.

3.1.1 Formación de fallas basadas en la distribución de Poisson

Mandelbrot [29] introduce la técnica formación de fallas basadas en la distribución de Poisson, que involucra desplazamientos aleatorios a un plano o esfera (fallas), basados en una distribución normal en intervalos acordes a una distribución de Poisson. Krten [25] introduce una variante denominada formación de fallas.

Shankel [53] describe el algoritmo de formación de fallas para generar terrenos con características como: mesas, acantilados y escarpes. Las fallas son generadas desplazando los valores del mapa de altura (por una cantidad determinada por el usuario) asociados a un lado de una línea seleccionada aleatoriamente. Este procedimiento es repetido hasta alcanzar el nivel de detalle deseado, donde en cada iteración se reduce linealmente la cantidad de desplazamiento. El proceso descrito genera diferencias notables entre vértices vecinos, similar a cortar un papel múltiples veces con una navaja. Para disminuir estas diferencias, el autor aplica un filtro paso bajo. La figura 3.1 ilustra el proceso.

Esta técnica tiene la gran ventaja de ser aplicable a esferas y por ende permite la creación de planetas. Por otra parte, la complejidad en tiempo depende del tamaño del mapa de altura y el número de iteraciones, requiriendo muchas iteraciones para obtener resultados interesantes.



(a) (b) (c)

Figura 3.2: Primera iteración del algoritmo de diamante-cuadrado: (a) cuadrícula inicial, (b) paso del diamante, (c) paso del cuadrado. Imágenes editadas de [36].

3.1.2 Desplazamiento del punto medio

Fournier et al. [16] publican la técnica desplazamiento del punto medio, que crea los mapas de altura mediante la subdivisión recursiva del mallado asociado al mapa de altura (teselación) y la perturbación aleatoria de los nuevos vértices. Diferentes esquemas de subdivisión han sido ideados para diferentes topologías de mallado. Los autores introducen dos esquemas: arista triangular *(triangular edge scheme)* para triángulos y diamante-cuadrado *(diamond-square scheme)* para cuadriláteros.

Shankel [54] describe el algoritmo de diamante-cuadrado para la generación de cordilleras, dividiéndolo en dos pasos: el paso del diamante y el paso del cuadrado; la figura 3.2 ilustra el procedimiento. La generación empieza con un cuadrilátero $ABCD$ con alturas asociadas a cada esquina, junto con un parámetro dh asociado al desplazamiento inicial. El paso del diamante, calcula la altura del punto medio del cuadrilátero (E), como el promedio de las alturas de cada esquina (A, B, C y D) más un desplazamiento aleatorio entre $-\frac{dh}{2}$ y $\frac{dh}{2}$; en el caso del paso del cuadrado, calcula la altura del punto medio de cada lado del cuadrilátero (F, G, H y I), como el promedio de las alturas de las esquinas y los puntos medios de los cuadriláteros adyacentes más un desplazamiento aleatorio entre $-\frac{dh}{2}$ y $\frac{dh}{2}$. Finalmente, se reduce el desplazamiento multiplicando dh por 2^{-H} (H es el exponente de Hurst y 2 es la lacunaridad) y se repite el procedimiento para cada subcuadrilátero: $AFEG$, $FBHE$, $GEIC$ y $EHDI$. Este procedimiento es repetido hasta alcanzar el nivel de detalle deseado.

El algoritmo de diamante-cuadrado es muy eficiente, ya que la complejidad en tiempo es lineal a las dimensiones del mapa de altura y la cantidad de cómputo por vértice es muy limitada. No obstante, el algoritmo opera únicamente con mapas de altura potencia de dos ($2^k \times 2^k$) y define una lacunaridad fija de 2. Por otra parte, los esquemas de subdivisión citados crean artefactos que lucen como pliegues (discontinuidades de pendiente) como ilustra Miller en [33].

3.1.3 Síntesis de ruido

Perlin [40] presenta un modelo fractal funcional como la suma de varias copias apropiadamente escaladas de un ruido de Perlin, donde tanto la lacunaridad como la dimensión fractal son fijas.

Saupe [50] presenta el modelo denominado reescalar y sumar *(rescale-and-add)*, que extiende el modelo de Perlin a un modelo fractal completo que permite controlar la dimensión fractal y la lacunaridad. Simultáneamente, Musgrave et al. [34] presentan un modelo denominado síntesis de ruido, que permite controlar localmente la dimensión fractal y otros parámetros de la generación. El artículo de Saupe esta enfocado en los fundamentos matemáticos, mientras la síntesis de ruido esta enfocada en las aplicaciones. Esta síntesis es matemáticamente representada como:

$$f(p) = \sum_{i=0}^{n-1} \frac{N(r^i p)}{r^{iH}} \tag{3.1}$$

donde N es la función base (ruido de Perlin), r es la lacunaridad, H es el exponente de Hurst y n es el número de bandas. La dimensión fractal es $3 - H$ basado en la relación entre H y la dimensión fractal expresada en la ecuación 1.1. El modelo de Perlin es un caso específico de síntesis de ruido con $r = 2$ y $H = 0{,}5$.

Cada término de la suma es denominado banda, donde el número de bandas afecta directamente el nivel de detalle y el tiempo de ejecución requerido, a mayor cantidad mayor nivel de detalle y consecuentemente mayor tiempo de ejecución. En la sección 3.4 se presentan diferentes ruidos que sirven como función base.

El enfoque funcional de síntesis de ruido donde cada punto es calculado independientemente de sus vecinos, implica una serie de ventajas importantes: modelo adecuado para ejecutarse en la GPU, control local de los parámetros y nivel de detalle adaptable *(antialiasing)*. Además, produce menos artefactos que las técnicas mencionadas.

3.2 Multifractales

Musgrave et al. [34] extienden el enfoque síntesis de ruido para producir terrenos heterogéneos. Los autores se basan en la observación, que en ciertas cordilleras hay un cambio importante en las mediciones estadísticas de la superficie a medida que uno se desplaza de los piedemontes hacia los picos; donde los piedemontes son más suaves debido a la acumulación de materiales, mientras que los picos son más dentados producto de procesos erosivos.

Este comportamiento puede ser caracterizado como un cambio de dimensión fractal en función de la altura; fractales que requieren múltiples valores de sus parámetros para caracterizarlos, son clasificados como fractales heterogéneos o multifractales. Por consiguiente, los fractales generados por las técnicas descritas en la sección 3.1, son clasificados como fractales homogéneos o monofractales.

Los autores plantean un modelo multifractal denominado estadísticas por altura o multifractal heterogéneo, que establece la dimension fractal en función de cuan cerca esta el punto a evaluar con respecto al nivel del mar, donde la rugosidad tiende a ser menor. El modelo es calculado con la siguiente ecuación:

$$f(p) = \sum_{i=0}^{n-1} g(i-1)\frac{N(r^i p) + o}{r^{iH}} \tag{3.2}$$

donde o controla la ubicación del "nivel del mar" y $g(j)$ para $j > 0$ representa el valor acumulado después de j iteraciones (el valor actual de la función).

Musgrave en [11] describe otro modelo multifractal denominado multifractal multiplicativo, que multiplica las bandas en vez de sumarlas. El modelo es expresado matemáticamente como:

$$f(p) = \prod_{i=0}^{n-1} \frac{N(r^i p) + o}{r^{iH}} \tag{3.3}$$

donde o controla la multifractalidad de la función, cuando es 0 la multifractalidad es máxima y a medida que aumenta, la función pasa de multifractal a monofractal hasta aproximar un plano (cuando es aproximadamente 100 o más). El rango de la salida es bastante impredecible, por lo que es necesario medirlo después de la creación para poder reescalarlo a un rango predecible.

3.3 Generación interactiva y controlada

Las técnicas basadas completamente en fractales, controlan el resultado por medio de parámetros que tienen incidencia global y no permiten especificar ni el tamaño ni la localización de las características. Además, predecir el efecto de cada parámetro requiere una comprensión de la base matemática subyacente y/o experimentos hasta alcanzar el efecto deseado.

Schneider et al. [52] presentan una técnica para la síntesis, edición y renderización de terrenos pseudo-infinitos en tiempo real (GPU), que exhiben un rango amplio de estructuras geológicas (tipos de terrenos). La técnica integra la síntesis en la renderización, por lo tanto, no requiere ningún tipo de representación poligonal ni fase de pre-procesamiento.

La generación del terreno esta basada en síntesis de ruido multifractal, definiendo la rugosidad dependiente de la altura y aplicando deformación del dominio (traslación y rotación) dependiendo de la rugosidad. La idea básica es aumentar la cantidad de rotación basado en la rugosidad, generando estructuras turbulentas similares a la lava fría. La generación de terrenos pseudo-infinitos es lograda mediante la repetición de la función base, sobre todo el dominio base $2D$.

El editor fractal permite proporcionar o pintar las funciones base representadas como imágenes en escala de grises. La función base resultante utilizada por el generador, es la composición de las distintas funciones base proporcionadas, permitiendo alcanzar un resultado particular tal como un terreno con 30 % desierto y 70 % cráteres. Además, permite modificar la forma general del terreno representado por un mapa de altura de baja frecuencia (boceto), utilizando herramientas de dibujo convencionales (elevación, hundimiento, entre otras). Internamente, las funciones base, los pesos y el boceto son almacenados en texturas $2D$.

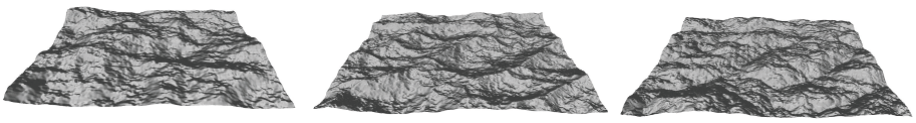
El renderizador, evalúa el mapa de altura de la porción visible del terreno mediante la síntesis de ruido multifractal, utilizando las texturas y los parámetros proporcionados por el usuario. Para determinar los vértices dentro del *viewport*, se utiliza una cuadrícula proyectada; esta técnica, proyecta una cuadrícula regular definida en espacio de imagen al dominio base y desplaza los vértices de acuerdo a las alturas evaluadas.

De Carpentier y Bidarra [5] presentan las brochas procedimentales *(procedural brushes)*, que ofrecen una transición continua entre el control local y la generación completamente automática mediante el tamaño de la brocha. De Carpentier, presenta la implementación detallada en [8] y publica el editor de terrenos Scape basado en brochas procedimentales [9].

El sistema basado en brochas, permite posicionar y activar una brocha con forma circular directamente sobre el terreno, mediante un dispositivo de entrada (tal como el ratón); este sistema es implementado en la GPU, logrando ediciones interactivas. Los autores describen distintos procedimientos asociados a la brocha tales como: elevación, hundimiento y distintas variantes de síntesis de ruido. Adicionalmente, adaptan la síntesis de ruido para modelar diversos tipos de terrenos, incorporando transformaciones en partes de la ecuación:

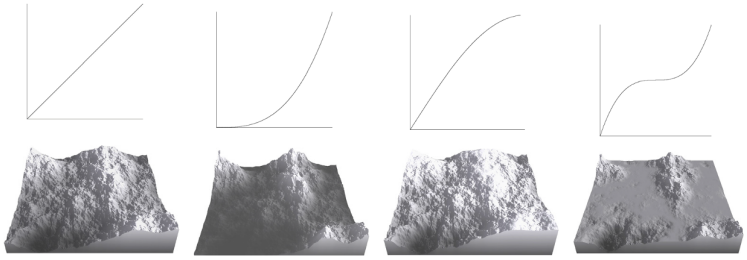
$$H(p) = T_{post}\left(\sum_{j=0}^{n-1} w^j T_{in}\left(N\left(T_{pre}(\lambda^j p), e_j\right)\right)\right) \tag{3.4}$$

donde $N(p, e)$ es el ruido de Perlin, p es la posición, e es la semilla, λ es la lacunaridad, w es la rugosidad (calculada como $\frac{1}{\lambda^H}$, donde H es el exponente de Hurst) y n es el número de bandas.



(a) (b) (c)

Figura 3.3: Transformaciones de síntesis de ruido: (a) síntesis de ruido básico ($T_{in}(h) = h$), (b) ruido de cresta *(ridged noise)* ($T_{in}(h) = 1 - abs(h)$), (c) ruido ondulante *(billowy noise)* ($T_{in}(h) = abs(h)$). Imágenes extraídas de [5].



(a) (b) (c) (d)

Figura 3.4: Deformación del rango utilizando las funciones *bias* y *gain*: (a) terreno original ($T_{post}(h) = h$), (b) efecto de glaciar ($T_{post}(h) = bias_b(h)$ para $b < \frac{1}{2}$), (c) efecto de cañón ($T_{post}(h) = bias_b(h)$ para $b > \frac{1}{2}$), (d) efecto de tierra media ($T_{post}(h) = gain_g(h)$ para $g < \frac{1}{2}$). En la parte superior se encuentra el mapeo y en la inferior el resultado. Imágenes extraídas de [8].

T_{pre} es un pre-procesamiento denominado deformación del dominio, T_{post} es un post-procesamiento denominado deformación del rango y T_{in} procesa la función base. T_{pre} puede ser definida como una distorsión del dominio, que desplaza p mediante un ruido. La figura 3.3 muestra transformaciones básicas asociadas a T_{in}. Las funciones *bias* y *gain* [38] son utilizadas frecuentemente como deformaciones del rango, como ilustra la figura 3.4.

Los autores presentan dos algoritmos novedosos: el ruido direccional y el ruido erosivo. El ruido direccional, comprime o estira la síntesis en un ángulo relativo a la dirección local del trazo de la brocha. El ruido erosivo, desplaza el punto de entrada para bandas subsecuentes en la dirección del gradiente local del ruido de Perlin (hacia el pico más cercano), estirando las características en las pendientes y comprimiendolas en los topes. La figura 3.5 ilustra el resultado del ruido erosivo y la distorsión del dominio, en la misma se puede apreciar como se aproxima el efecto erosivo de la lluvia, formando barrancos y valles suaves.

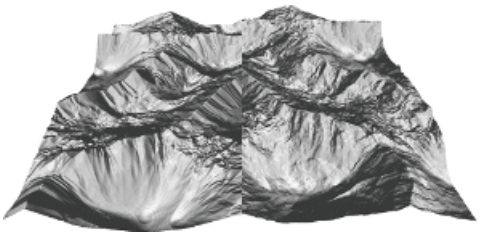


Figura 3.5: Ruido erosivo: original (mitad izquierda) y distorsión del dominio (mitad derecha). Imagen tomada de [5].

3.4 Ruido

El ruido es una primitiva estocástica, que permite agregar complejidad visual de manera controlada. La idea general, es construirlo como una versión de ruido blanco a la cual se le aplica un filtro paso bajo *(antialiasing)*. Sus propiedades más importantes son: ancho de banda limitado, estacionario (invariante a la traslación) e isotrópico (invariante a la rotación).

La sección esta enfocada en la generación procedimental de ruido de retículo *(lattice noise)*, tomando en cuenta que constituye el enfoque más popular en las aplicaciones. Una revisión completa de diferentes enfoques para la generación de ruido es presentada por Ebert et al. [11]. También, Lagae et al. [26] presentan una revisión más reciente, donde describen los últimos avances en generación procedimental de ruido.

El ruido de retículo es generado interpolando un retículo entero, cuyas coordenadas tienen asociadas uno o más números pseudoaleatorios. Dos tipos de ruido de retículo son presentados: el ruido de valor *(value noise)* y el ruido de gradiente *(gradient noise)*. Los ruidos de retículo, comúnmente producen patrones conocidos como artefactos de cuadrícula, que evidencian la estructura de la cuadrícula subyacente. La sección concluye con un análisis comparativo entre ambos tipos.

Peachey en [11] describe el ruido de valor, que asigna números pseudoaleatorios en el rango $[-1; 1]$ al retículo entero y construye el ruido aplicando un esquema de interpolación al mismo. La decisión clave es el esquema de interpolación, los esquemas utilizados van desde interpolación lineal hasta una variedad de técnicas de interpolación cúbica. La interpolación lineal no es adecuada ya que produce artefactos que lucen como cajas, donde se aprecian las celdas del retículo.

Peachey en [11] describe el ruido de gradiente, que asigna gradientes pseudoaleatorios (vectores) al retículo entero y utiliza estos gradientes para construir el ruido. Perlin [40] introduce conceptualmente la primera implementación de

este tipo, denominada el ruido de Perlin, posteriormente descrito por Perlin y Hoffert [38].

El procedimiento para evaluar el ruido de Perlin en un punto en el espacio p es:

1. Definir un gradiente pseudoaleatorio en cada uno de los 8 vértices vecinos más cercanos a p en el retículo entero.

2. Determinar la contribución de cada vecino con respecto al punto p, como el producto punto entre el gradiente y el vector formado desde el vértice en cuestión hacia el punto p.

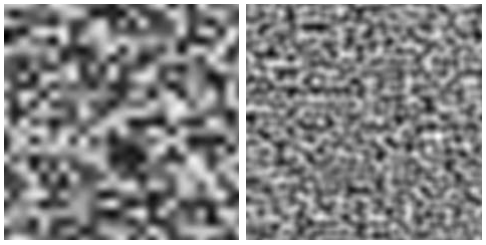
3. Mezclar la contribución de cada vecino mediante una interpolación trilineal, utilizando la función de mezcla $s(t) = 3t^2 - 2t^3$ como interpolante.

El gradiente pseudoaleatorio es obtenido evaluando una función hash, que aplica sucesivamente una permutación pseudoaleatoria a las coordenadas con la finalidad de descorrelacionar los índices en la tabla de gradientes G. Esta función hash evita patrones notables no deseados. La tabla G contiene vectores distribuidos uniformemente en la esfera unitaria. El procedimiento descrito, se puede generalizar a n dimensiones, tomando en cuenta los 2^n vecinos más cercanos en un retículo n-dimensional.

Perlin [42] introduce una versión mejorada de su ruido, que corrige dos defectos con respecto a la naturaleza del interpolante y el conjunto de gradientes. El interpolante produce discontinuidades de segundo orden en las aristas de la cuadrícula, ya que la segunda derivada $6 - 12t$ no es 0 ni en $t = 0$ ni $t = 1$. Estas discontinuidades son notables cuando una superficie desplazada por el ruido es sombreada, donde los efectos de iluminación varían con la normal de la superficie (operador de derivada). La solución planteada es reemplazar el interpolante $s(t) = 3t^2 - 2t^3$ por $s(t) = 6t^5 - 15t^4 + 10t^3$, que tiene primera y segunda derivada de valor 0 tanto en $t = 0$ como $t = 1$.

El defecto asociado con el conjunto de gradientes es la distribución irregular, donde gradientes muy cercanos se amontonan generando valores anormalmente altos; este defecto produce una apariencia de mancha. La solución planteada, es definir el conjunto de gradientes G con solo 12 gradientes, correspondientes a las direcciones del centro del cubo unitario al punto medio de sus aristas. Además de solucionar el problema de la distribución irregular, esta modificación permite evitar muchas multiplicaciones asociadas a la técnica. Por ejemplo, el producto punto entre (x, y, z) y $(1, 1, 0)$ puede ser calculado simplemente como $x + y$.

Gustavson [23] explica el ruido de Perlin de forma más detallada y clara. Perlin [43] describe una implementación en GPU del ruido de Perlin mejorado utilizando texturas 3D, aprovechando la aceleración de hardware de la interpolación trilineal. Green [20] plantea una implementación del ruido de Perlin



(a) (b)

Figura 3.6: Corte 2D de un: (a) ruido de valor, (b) ruido de gradiente. Imágenes extraídas de [11].

mejorado en sintaxis FX y CgFX, que coincide exactamente con la versión en CPU. Las tablas de gradientes y permutación son almacenadas en texturas.

En el *píxel shader*, se implementa un código similar a la versión en CPU, utilizando estas texturas para acceder a la información. McEwan et al. [30] presentan una implementación completamente autocontenida en GLSL (sin referencia a datos externos) del ruido de Perlin.

La figura 3.6 muestra el resultado del ruido de valor y del ruido de gradiente. El ruido de valor es más sencillo tanto conceptualmente como en implementación que el ruido de gradiente, no obstante, presenta más artefactos de cuadrícula. Además, el ruido de gradiente tiene más energía de alta frecuencia y a su vez menos energía de baja frecuencia, es decir, es un ruido de ancho de banda más estrecho.

Capítulo 4

Diseño

En este capítulo, se describirá el diseño de la aplicación implementada, la cual pretende generar diversos tipos de terrenos basado en la extensión de síntesis de ruido propuesta por De Carpentier y Bidarra [5], implementando los algoritmos, las transformaciones y las funciones base de manera tal de poder realizar pruebas sobre los mismos. El contenido se divide en tres secciones. Primero, en la sección 4.1 se explicarán los requerimientos tanto en hardware como software, necesarios para la implementación de la aplicación. Segundo, en la sección 4.2 se describe brevemente en qué consiste la aplicación. Por último, en la sección 4.3 se mostrará un diagrama de clases de la aplicación y la explicación en forma general de cada clase.

4.1 Herramientas y entorno de desarrollo

El desarrollo de la implementación se basa en el paradigma de programación orientada a objetos. El sistema debe cumplir con los siguientes requisitos:

Requisitos de hardware:

- Computadora personal (PC) convencional.
- Tarjeta gráfica que soporte al menos OpenGL 3.3 y GLSL 3.30.

Requisitos de software:

- Windows 7 como sistema operativo.
- Microsoft Visual Studio 2012 como ambiente de trabajo.
- C++ como lenguaje de programación.
- Freeglut 2.8.1 como biblioteca para el despliegue de gráficos 2D y 3D sobre OpenGL 3.3.

- GLEW 1.9.0 como biblioteca para el manejo de extensiones y shaders en OpenGL sobre GLSL 3.30.
- GLM 0.9.5.2 [6] como biblioteca matemática para software gráfico basado en la especificación GLSL.
- FreeImage 3.16.0 [17] como biblioteca para la carga y almacenamiento de imágenes en los formatos más populares.
- AntTweakBar 1.16 [2] como biblioteca para la interfaz gráfica de usuario.

4.2 Descripción de la aplicación

La aplicación esta compuesta por dos módulos principales: el Generador y el Visualizador. El Generador se encarga de producir un mapa de altura basado en un algoritmo y un conjunto de parámetros asociados con la finalidad de producir un terreno con características específicas. El visualizador se encarga de la visualización tridimensional de un mapa de altura, es decir, permite la navegación del terreno que representa al mapa de altura proporcionado o generado. Los distintos materiales (grama, arena, roca, entre otros) presentes en el terreno son representados mediante texturas proporcionadas por el usuario, incluyendo los mapas de detalle asociados.

La interfaz gráfica esta basada en tres barras de AntTweakBar: la principal, el generador y el visualizador. La barra principal se encarga de manejar las operaciones básicas de la aplicación: cargar, generar y guardar un mapa de altura e importar el conjunto de materiales (texturas y sus mapas de detalle asociados). Además, dispone de los parámetros estáticos asociados al Visualizador como la escala del terreno y la resolución del Skydome.

La barra del generador dispone de la configuración utilizada para generar un mapa de altura: las dimensiones del mapa de altura, el algoritmo utilizado, la lacunaridad, entre otros. En tanto que, la barra del visualizador dispone de la configuración dinámica utilizada por el Visualizador: la velocidad de la cámara, las opciones activadas por el usuario, el tipo de texturizado (basado en altura o basado en pendiente), entre otros.

El Visualizador incluye las siguientes características:

- Texturizado semi-automático de múltiples materiales basado en la altura o pendiente del terreno.
- Iluminación dinámica basada en el modelo de iluminación local de Phong.
- Navegación libre del terreno, integrando un mapa de navegación.

- El cielo, que permite una percepción más realista del ambiente. Incluyendo el modelado del Sol.

El Generador incluye las siguientes características:

- Generación de terrenos basada en la extensión de síntesis de ruido descrita en el trabajo elaborado por De Carpenter y Bidarra [5], explorando distintas transformaciones asociadas al pre-procesamiento y post-procesamiento del mapa de altura.
- Algoritmos de generación multifractal propuestos por Musgrave en [11] aplicados a la extensión propuesta.
- Algoritmo de generación basado en el gradiente local de la función base planteado por Quilez [47] aplicado a la extensión propuesta.
- Ruido de valor y ruido de Perlin como función base. Incluyendo el gradiente local asociado.

4.3 Diagrama de clases y estructuras de datos

En la figura 4.1 se mostrará el diagrama de clases de la aplicación con los atributos y métodos más importantes. No se colocaron todos los atributos y métodos para mantener el diagrama de clases compacto. A continuación se describe cada clase en forma general.

4.3.1 Clase *Shader*

La clase *Shader* permite cargar, compilar y manipular un *vertex shader* y un *fragment shader*. La clase permite activar o desactivar los *shaders* por medio de los métodos *Enable* y *Disable* respectivamente. El programa completo (*vertex shader* y *fragment shader*) es representado por el atributo *program*, cualquier error al momento de compilar los *shaders* o al enlazar el programa son notificados por consola. Las variables uniformes son modificadas mediante los métodos: *SetUniform* (*bool*, *int* y *float*), *SetUniformArray* (arreglo de *floats*), *SetUniformVector* (vector 3D), *SetUniformMatrix* (matriz 4×4), *SetUniformTexture (2D Texture)* y *SetUniformTextureArray (2D Texture Array)*. Por último, la clase permite verificar las variables uniformes activas y los atributos activos mediante los métodos *PrintUniforms* y *PrintAttributes* respectivamente.

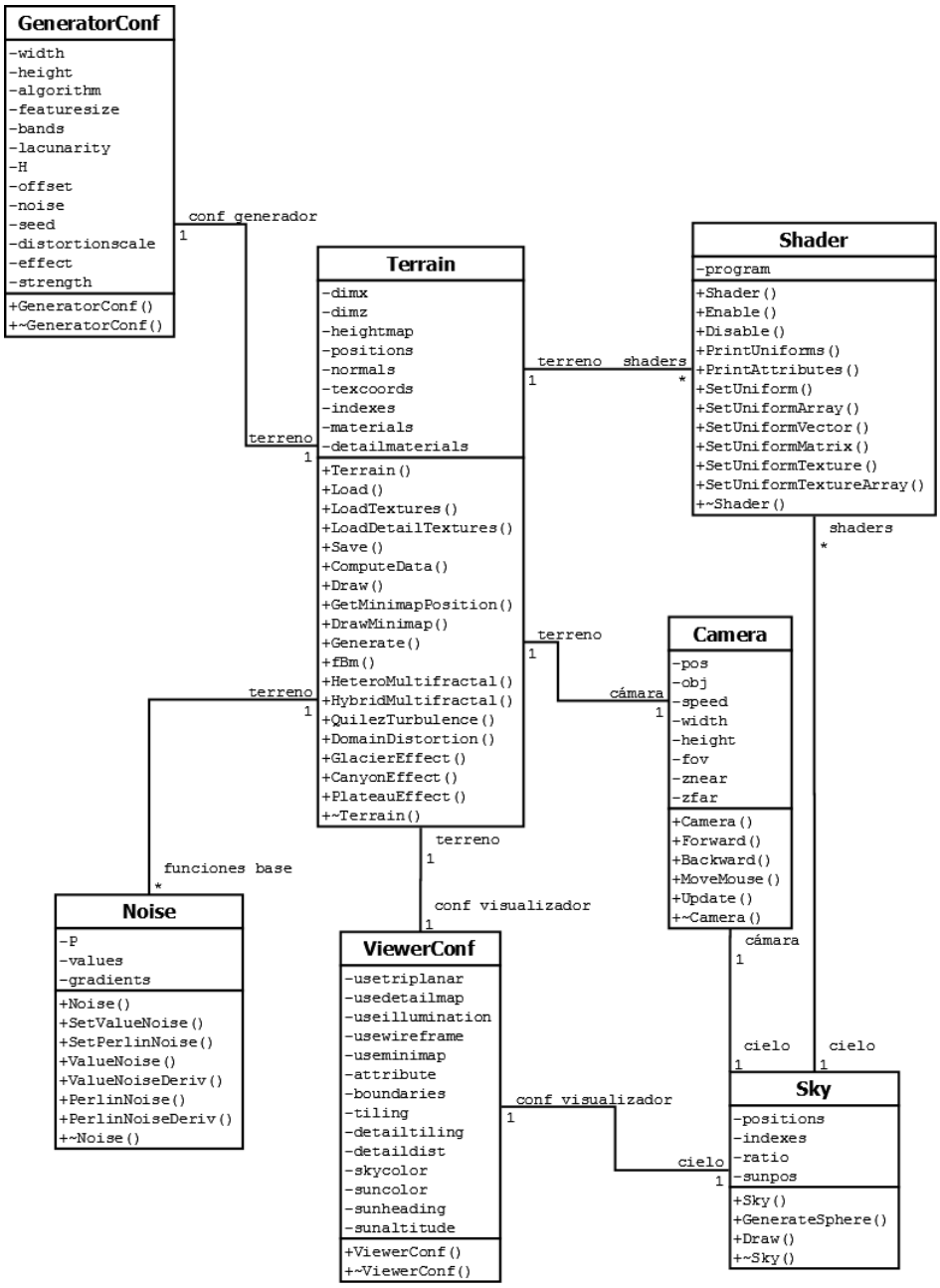


Figura 4.1: Diagrama de clases de la aplicación. Este diagrama fue elaborado con el programa Dia [10]

4.3.2 Clase *Camera*

La clase *Camera* se encarga de la navegación de escenas en primera persona en modo volar o libre. Las cámaras en OpenGL se definen mediante la función *gluLookAt*, que define una transformación de visualización basada en la posición y la orientación de la cámara (derivadas de los puntos *pos* y *obj*) y el vector *up*. *pos* y *obj* indican la posición de la cámara y la posición del punto de referencia respectivamente, a partir de estos puntos se calcula el vector de visión. En esta implementación, *up* siempre se define como $(0, 1, 0)$ y por lo tanto no es necesario almacenarlo.

Además, la clase define la proyección en perspectiva mediante la función *gluPerspective* basado en los parámetros: *fov*, *width*, *height*, *znear* y *zfar*; en tanto que, el usuario controla la cámara mediante los métodos: *Forward*, *Backward* y *MoveMouse*. *Forward* desplaza la posición de la cámara en la dirección del vector de visión (avanza) basado en la velocidad de la cámara *(speed)* cuando el usuario pulsa la tecla W. *Backward* es similar a *Forward* pero desplaza en la dirección opuesta al vector de visión (retrocede) cuando el usuario pulsa la tecla S. *MoveMouse* rota la cámara basado en el movimiento del mouse. Por último, el método *Update* define el *viewport*, actualiza la posición y la orientación de la cámara y aplica la proyección en perspectiva cada vez que la escena es dibujada.

4.3.3 Clase *ViewerConf*

La clase *ViewerConf* sirve como estructura de datos para almacenar las entradas proporcionadas por el usuario en la barra del visualizador. Específicamente, almacena las entradas para el texturizado (tipo de atributo, número de repeticiones de la textura, entre otros) y la iluminación (posición del Sol, color del cielo y color del Sol). Además, almacena las variables booleanas con las opciones que el usuario puede activar (proyección triplanar, mapas de detalle, mallado y mapa de navegación). Los métodos asociados a esta clase son métodos para acceder y modificar los valores de los atributos, estos métodos no fueron colocados en el diagrama de clases para mantenerlo compacto.

4.3.4 Clase *GeneratorConf*

Esta clase sirve como estructura de datos para almacenar las entradas proporcionadas por el usuario en la barra del generador. Específicamente, almacena las entradas asociadas al mapa de altura (dimensiones), a los parámetros de la generación (tipo de algoritmo de generación, lacunaridad, número de bandas, entre otros), a la función base (tipo de ruido y semilla) y a las transformaciones (escala de la distorsión, tipo de efecto e intensidad del efecto). Los métodos asociados a esta clase son métodos para acceder y modificar los valores de los atributos, estos métodos no fueron colocados en el diagrama de clases para man-

tenerlo compacto.

4.3.5 Clase *Noise*

La clase *Noise* se encarga de modelar las funciones base utilizadas por el Generador. Ésta, permite evaluar un ruido de valor $2D$ y un ruido de perlin $2D$ en un punto determinado, mediante los métodos *ValueNoise* y *PerlinNoise* respectivamente. Los valores y gradientes asociados a la cuadrícula (*values* y *gradients*), utilizados por el ruido de valor y el ruido de Perlin respectivamente, son generados mediante los métodos *SetValueNoise* y *SetPerlinNoise* respectivamente. P es el arreglo con la permutación utilizada en la construcción de ambas funciones base.

El ruido de valor $2D$ junto su gradiente local y el ruido de Perlin $2D$ junto su gradiente local son calculados mediante los métodos *ValueNoiseDeriv* y *PerlinNoiseDeriv* respectivamente. Esta clase será explicada en detalle en el capítulo 6.

4.3.6 Clase *Terrain*

La clase *Terrain* se encarga de incluir el terreno a la escena, el cual es representado mediante un mapa de altura almacenado en la matriz *heightmap*. La clase dibuja el terreno (visualización tridimensional del mapa de altura) en GPU basado en *shaders* mediante el método *Draw*, incluyendo el texturizado de múltiples materiales y la iluminación dinámica. La configuración dinámica asociada a la visualización es almacenada en un objeto de la clase *ViewerConf* y pasado como parámetro a *Draw*. El mallado del terreno y los atributos necesarios para la visualización son calculados mediante el método *ComputeData*; éste, calcula las posiciones, las normales y las coordenadas de textura por vértice y las almacena en los arreglos *positions*, *normals* y *texcoords* respectivamente; además, construye los índices que representan el mallado y los almacena en el arreglo *indexes*. Las texturas y los mapas de detalles asociados a los materiales son cargados en *materials* y *detailmaterials (Texture Arrays)* mediante los métodos *LoadTextures* y *LoadDetailTextures* respectivamente.

La clase *Terrain* dibuja el mapa de navegación mediante el método *DrawMinimap*. El mapa de navegación consiste en mostrar el mapa de altura (imagen) que representa el terreno junto con la posición y la orientación de la cámara en un momento determinado; la posición es expresada como un punto $2D$ con tamaño relativamente grande mientras la orientación es expresada dibujando la vista superior del *frustum* de visualización (plano superior del *frustum*). Una explicación detallada de como extraer la información de cada plano del *frustum* es presentada en [28]. Además, el usuario puede interactuar con el mapa de navegación y transportarse directamente a cualquier parte del terreno haciendo click en el mapa de navegación. Para obtener la posición en el terreno

correspondiente a la posición en el mapa de navegación, se utiliza el método *GetPositionMinimap*.

Los mapas de altura pueden ser cargados o generados mediante el método *Load* o *Generate* respectivamente. La configuración asociada a la generación del mapa de altura es almacenada en un objeto de la clase *GeneratorConf* y pasado por parámetro a *Generate*. Los algoritmos de generación disponibles son implementados en los métodos: *fBm*, *HeteroMultifractal*, *HybridMultifractal* y *QuilezTurbulence*. Los distintos tipos de transformaciones son implementados en los métodos: transformación de pre-procesamiento (*DomainDistortion*) y transformaciones de post-procesamiento (*GlacierEffect*, *CanyonEffect* y *PlateauEffect*). Esta clase será explicada en detalle en el capítulo 5 y 6.

4.3.7 Clase *Sky*

La clase *Sky* se encarga de incluir el cielo a la escena. El cielo es modelado mediante *Sky domes*, donde la esfera es construida a través del método *GenerateSphere*; este, calcula las posiciones de la esfera en el arreglo *positions* basado en coordenadas esféricas y construye los índices que representan el mallado en el arreglo *indexes*. Finalmente, el cielo es dibujado en GPU basado en *shaders* mediante el método *Draw*. La configuración dinámica asociada a la visualización es almacenada en un objeto de la clase *ViewerConf* y es pasado por parámetro al método *Draw*. Esta clase se explicará en detalle en el capítulo 5.

Capítulo 5

Implementación del Visualizador

En este apartado se explicará en detalle la implementación del Visualizador tridimensional de mapas de altura, diviendo el capítulo en cinco secciones. Primero, en la sección 5.1 se describirá el algoritmo para construir el mallado del terreno basado en un mapa de altura, incluyendo atributos de los vértices como las coordenadas de textura y las normales. Seguidamente, en la sección 5.2 se describirá la implementación del texturizado de múltiples materiales representados por texturas. Después, en la sección 5.3 se explicará el modelo de iluminación implementado para calcular la iluminación del terreno.

Posteriormente, en la sección 5.4 se describirá la implementación de la visualización tridimensional del terreno basada en *shaders*. Finalmente, en la sección 5.5 se explicará la implementación del cielo que permite una percepción más realista al momento de navegar el terreno.

5.1 Geometría del terreno

El mallado del terreno es calculado mediante el método *ComputeData* de la clase *Terrain*, mostrado en el código 5.1; éste construye el mallado similar al algoritmo de visualización simple descrito en la sección 2.1 basado en el mapa de altura almacenado en la matriz *heightmap*. Los atributos por vértice necesarios para la visualización del terreno son las posiciones, las coordenadas de textura y las normales; almacenadas en los arreglos *positions*, *texcoords* y *normals* respectivamente.

El algoritmo define una cuadrícula de vértices, donde cada vértice corresponde a un píxel del mapa de altura y la altura del vértice (coordenada y) es el valor del píxel. Es decir, el vértice que corresponde al píxel con coordenadas

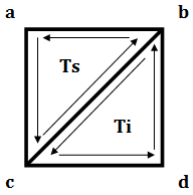


Figura 5.1: Definición de los dos triángulos (triángulo superior Ts y triángulo inferior Ti) por cada cuadrilátero del terreno, ilustrando la orientación en sentido contrario a las agujas del reloj al especificar los vértices de cada triángulo. La notación corresponde a la utilizada en la construcción del mallado en el código 5.1

(x, y) y valor h tendrá la posición (x, h, y). El terreno es escalado basado en dos parámetros: *cellsize* y *heightscale*. *cellsize* define la distancia entre cada vértice y *heightscale* define la altura máxima.

Las coordenadas de textura (u, v) son generadas normalizando las coordenadas del píxel (x, y) del mapa de altura. Tomando en cuenta que la implementación incluye proyección triplanar para el mapeo de las texturas, también se incluye la coordenada de textura asociada a la altura (coordenada y). El cálculo consiste en normalizar la altura del terreno (altura almacenada en *heightmap* por *heightscale*) dividiendo por el máximo entre el ancho y el alto del mapa de altura. La proyección triplanar será explicada en detalle en la sección 5.2.

Después de definir la posición y las coordenadas de textura de cada vértice, se construye el mallado de triángulos que conecta los vértices basado en la triangulación mostrada en la figura 2.1. El procedimiento consiste en definir los índices de los vértices de cada cuadrilátero y definir los triángulos que lo componen especificando los índices correspondientes en sentido contrario a las agujas del reloj *(counterclockwise)*. Estos índices son almacenados en el arreglo *indexes*. La figura 5.1 ilustra la notación y el orden utilizado en el código para calcular los índices de cada cuadrilátero.

Finalmente, el algoritmo determina las normales del terreno por cada vértice, calculando la normal de cada triángulo y agregando la misma a cada vértice perteneciente al mismo. Luego, se normaliza cada normal, promediando las influencias de cada triángulo conectado a cada vértice.

```
1  void Terrain::ComputeData(){
2
3    int x,z,i;
4    float s;
5
6    i = 0;
7    s = (float)max(dimx,dimz);
```

```
8
9     nvertexes = dimz*dimx;//Número de vértices
10    positions = new glm::vec3[nvertexes];
11    normals = new glm::vec3[nvertexes];
12    texcoords = new glm::vec3[nvertexes];
13
14    //Calcula los atributos por vértice
15    for(z = 0; z < dimz; z++) {
16      for(x = 0; x < dimx; x++) {
17
18        positions[i] = glm::vec3(cellsize*x,heightmap[z][x]*heightscale,↩
              cellsize*z);
19        normals[i] = glm::vec3(0.0f,0.0f,0.0f);
20        texcoords[i++] = glm::vec3((float)x/(dimx-1.0),heightmap[z][x]*↩
              heightscale/(s-1.0),(float)z/(dimz-1.0));
21      }
22    }
23
24    //Calcula los índices del mallado
25    int a,b,c,d;
26
27    i = 0;
28    nindexes = 6*(dimz-1)*(dimx-1);//Número de índices
29    indexes = new GLuint[nindexes];
30
31    for(z = 0; z < dimz-1; z++) {
32      for(x = 0; x < dimx-1; x++) {
33
34        //Esquinas del cuadrilátero
35        a = z*dimx+x;
36        b = a+1;
37        c = a+dimx;
38        d = c+1;
39
40        //Triangulo superior: (a,c,b)
41        indexes[i++] = a;
42        indexes[i++] = c;
43        indexes[i++] = b;
44
45        //Triangulo inferior: (b,c,d)
46        indexes[i++] = b;
47        indexes[i++] = c;
48        indexes[i++] = d;
49      }
50    }
51
52    //Calcula las normales
53    glm::vec3 va,vb,vc,res;
54
55    for(i = 0; i < (int)nindexes; i+=3){
56
57      va = positions[indexes[i]];
58      vb = positions[indexes[i+1]];
59      vc = positions[indexes[i+2]];
60
61      res = glm::normalize(glm::cross(vb-va,vc-va));
62
63      normals[indexes[i]] += res;
64      normals[indexes[i+1]] += res;
65      normals[indexes[i+2]] += res;
66    }
67
68    //Normaliza las normales
69    for(i = 0; i < (int)nvertexes;i++) normals[i] = glm::normalize(↩
          normals[i]);
70  }
```

Código 5.1: Algoritmo para generar el mallado del terreno (método

ComputeData de la clase *Terrain*)

5.2 Texturizado del terreno

El texturizado del terreno en esta implementación es automático basado en un conjunto de materiales (texturas) y un atributo del mapa de altura (altura o pendiente). Primero, en la sección 5.2.1 se describe el algoritmo de texturizado implementado basado en la técnica presentada por Nicholson [35] descrita en la sección 2.2. Luego, en la sección 5.2.2 se explica la implementación de los mapas de detalle con la finalidad de agregar detalles en las regiones cercanas a la cámara. Finalmente, en la sección 5.2.3 se describe la solución implementada basada en la propuesta elaborada por De Carpentier [8] en su editor Scape [9] para el problema de estiramiento de texturas descrito en la sección 2.3.

5.2.1 Algoritmo de texturizado

Dado un conjunto ordenado de materiales (texturas) y un atributo asociado al vértice del terreno (altura o pendiente), se define un rango de valores del atributo asociado a cada textura de manera tal que abarque el rango completo del atributo sin solapamientos y respetando el orden de las texturas. Los valores del atributo deben estar normalizados para poder trabajar ambos atributos de la misma forma. El usuario define el valor máximo de cada material excepto el último ya que por definición será 1,0 y a partir de esta información es posible determinar el rango de valores asociado a cada textura. Por ejemplo, si el usuario proporciona la siguiente información: arena (0,11), grama (0,27) y roca (0,58); el rango de valores de cada textura sería: arena $[0{,}0; 0{,}11]$, grama $(0{,}11; 0{,}27]$, roca $(0{,}27; 0{,}58]$ y nieve $(0{,}58; 1{,}0]$.

Luego, en cada vértice el algoritmo selecciona la textura a utilizar determinando a cual rango pertenece el valor del atributo. La misma se determina recorriendo cada textura en orden, comparando el valor del atributo con el valor máximo del rango, si es menor o igual selecciona esa textura y en caso contrario continua el recorrido. Para evitar transiciones notables entre los materiales, se aplica una mezcla lineal en la frontera entre las texturas involucradas utilizando el valor del atributo como el peso de la mezcla.

5.2.2 Mapas de detalle

Cada textura del conjunto de materiales tendrá asociado un mapa de detalle adecuado. La textura y su mapa de detalle asociado son mezclados a través de la siguiente ecuación:

$$col = tex + detcol - 0{,}5 \tag{5.1}$$

donde *col* es el color resultante, *tex* es el color de la textura y *detcol* es el color del mapa de detalle. La ecuación 5.1 permite agregar los detalles del mapa de detalle manteniendo el color de la textura. Si se multiplicará el mapa de detalle con la textura, efectivamente la textura tendría los detalles pero perdería brillo.

Es importante considerar, que un mapa de detalle debería poseer un valor promedio de 0,5. Por lo tanto, si el promedio es mucho mayor, la textura se iluminaría (mayor brillo), mientras si es mucho menor se oscurecería (menor brillo).

5.2.3 Proyección triplanar de texturas

La proyección triplanar, consiste en aplicar las tres proyecciones asociadas a cada eje principal, combinando los colores obtenidos de las mismas mediante una suma ponderada, donde el peso de la suma esta relacionado a la normal de la superficie. Esta técnica aplica un procedimiento similar al texturizado triplanar, difiriendo únicamente en el cálculo del peso. Una comparación cualitativa de ambas técnicas es presentada por De Carpentier en [9], donde la proyección triplanar produce transiciones más suaves.

La proyección triplanar es calculada con la siguiente ecuación:

$$C = (C_{yz}, C_{xz}, C_{xy}) \cdot \frac{N^2}{\|N^2\|} \tag{5.2}$$

donde C es el color resultante, N es la normal de la superficie y C_{yz}, C_{xz} y C_{xy} es la muestra obtenida de la textura (colores RGB) mediante una proyección al plano yz, xz y xy respectivamente.

5.3 Iluminación del terreno

El modelo de iluminación implementado es una adaptación del modelo de iluminación de Phong [44] para terrenos (sin el componente especular). El modelo es planteado en la siguiente ecuación descrita por Foley y Dam en [14]:

$$I = I_a K_a + I_d K_d N \cdot L \tag{5.3}$$

donde I_a e I_d es la intensidad de la luz ambiental y la intensidad de la fuente de luz respectivamente. K_a y K_d es el coeficiente de reflexión ambiental y el coeficiente de reflexión difusa respectivamente, que dependen del material. N y L representan la normal de la superficie y la dirección hacia la fuente de luz respectivamente. Para considerar color se calcula la ecuación 5.3 por cada canal de color.

En este trabajo el componente difuso corresponde a la luz solar y el componente ambiental a la luz del cielo. Además, el color del material producto del

texturizado representa a ambos coeficientes de reflexión (K_a y K_d) por lo que el resultado de la ecuación representará el color final del terreno (texturizado e iluminación).

5.4 Visualización del terreno

La visualización del terreno se ejecuta mediante el método *Draw* de la clase *Terrain* basado en *shaders*. Los datos necesarios para la visualización basada en shaders son almacenados en *buffers* en la GPU mediante el método *LoadGPU*. Los atributos por vértice calculados en CPU: *positions*, *normals* y *texcoords*; son almacenados en los *Vertex Buffer Objects (VBOs)*: *vbopos*, *vbonormal* y *vbotexcoord* respectivamente. La implementación utiliza visualización indexada por ende el arreglo *indexes* es almacenado en GPU en el *Index Buffer Object (IBO) ibo*. Finalmente, los *VBOs* y el *IBO* son encapsulados en un *Vertex Array Object (VAO)* en la variable *vao*. Los códigos 5.2 y 5.3 muestran el *vertex shader* y el *fragment shader* respectivamente.

El vertex shader utiliza tres atributos por vértice: *positionin* (la posición), *normalin* (la normal) y *texcoordsin* (las coordenadas de textura). Primero, transforma la posición del vértice a espacio de imagen multiplicando *positionin* por las matrices de transformación. Luego, envía al *fragment shader* la normal y las coordenadas de textura mediante las variables *normal* y *texcoords* respectivamente. Además, envía al *fragment shader* la altura normalizada del vértice y la pendiente mediante las variables *height* y *slope* respectivamente. Considerando que la escala del terreno ya fue aplicada en CPU, *height* es calculada normalizando la altura del vértice (coordenada y de *positionin*) basado en la altura máxima del terreno (variable uniforme *heightscale*), y *slope* es calculada como el complemento de la coordenada y de la normal.

Finalmente, se envía al *fragment shader* la dirección hacia Sol y la distancia entre el vértice y la cámara mediante las variables *sundir* y *dist* respectivamente. *sundir* se calcula en base a la posición del Sol en coordenadas de objeto (variable uniforme *sunpos*) y *positionin*. *dist* se determina transformando *positionin* en coordenadas de ojo multiplicando *positionin* por la matriz *ModelView*, luego se calcula la distancia del resultado considerando que la cámara en coordenadas de ojo siempre está ubicada en la posición $(0, 0, 0)$ apuntando a $(0, 0, -1)$.

El *fragment shader* incluye las siguientes características: texturizado, iluminación y corrección gamma. El texturizado es implementado basado en el algoritmo de texturizado explicado en la sección 5.2. Primero, se define el atributo a utilizar basado en la variable uniforme *attr*. Luego, se recorre cada material hasta que el valor máximo del rango del material en cuestión almacenado en la variable uniforme *boundaries* sea mayor o igual al valor del atributo. Para evitar transiciones bruscas se aplica una mezcla lineal en la frontera entre los

materiales involucrados, en esta implementación la frontera abarca 0,05 (valor determinado empíricamente).

La función *GetColor* obtiene el color del material seleccionado, incluyendo proyección triplanar y mapas de detalle. Si la variable uniforme *usedetailmap* es verdadera se mezcla la textura del material con su mapa de detalle asociado basado en la ecuación 5.1. El efecto del mapa de detalle es limitado en distancia basado en la variable uniforme *detaildist*, es decir, el mapa de detalle tendrá incidencia hasta la distancia *detaildist*. Para lograr este comportamiento se aplica una mezcla lineal entre el mapa de detalle con el color gris (valor promedio del mapa de detalle) utilizando como peso $\frac{dist}{detaildist}$, limitando el peso al rango $[0; 1]$.

Si la variable uniforme *usetriplanar* es verdadera se aplica la proyección triplanar para obtener la textura y el mapa de detalle (si *usedetailmap* es verdadero) mediante la función *TriplanarProjection*, que es una implementación de la ecuación 5.2. Las variables uniformes *tiling* y *detailtiling* definen la frecuencia (número de repeticiones) de la textura y el mapa de detalle respectivamente.

Las texturas y los mapas de detalles asociados a cada material, son cargadas en las variables *materials* y *detailmaterials (Texture Array)* mediante los métodos *LoadTextures* y *LoadDetailTextures* de la clase *Terrain* respectivamente. Un *Texture Array* es estructuralmente similar a una textura $3D$, pero no aplica un filtro entre las texturas y la tercera coordenada de textura es un índice entero que representa una de las imágenes almacenadas.

El modelo de iluminación implementado (si la variable uniforme *useillumination* es verdadera) esta basado en el modelo de iluminación descrito en la sección 5.3. La intensidad de la luz ambiental y la intensidad de la luz solar corresponden a las variables uniformes *skycolor* (color del cielo) y *suncolor* (color del Sol) respectivamente.

El *fragment shader* opera con datos lineales, no obstante, la respuesta del monitor es no lineal y en consecuencia los colores no serán desplegados correctamente (imágenes muy oscuras). La corrección gamma soluciona este problema al cancelar la no linealidad que caracteriza la respuesta del monitor mediante elevar el color del fragmento a la potencia $\frac{1}{gamma}$ con valor 2,2 (valor recomendado en la bibliografía). Una explicación detallada de la corrección gamma y su importancia es presentada por Gritz y d'Eon en [21].

```
1  #version 330
2
3  //Atributos por vértice
4  layout(location = 0) in vec3 positionin;
5  layout(location = 1) in vec3 normalin;
6  layout(location = 2) in vec3 texcoordsin;
7
```

```
8  //Matrices de transformación
9  uniform mat4 ProjectionMatrix,ModelViewMatrix;
10 uniform float heightscale;//Escala de la altura del terreno
11 uniform vec3 sunpos;//Posición del Sol
12
13 out vec3 normal;//Normal del vértice
14 out vec3 texcoords;//Coordenadas de textura
15 out float height;//Altura normalizada del vértice
16 out float slope;//Pendiente del vértice
17 out vec3 sundir;//Dirección hacia el Sol
18 out float dist;//Distancia entre la cámara y el vértice
19
20 void main()
21 {
22   gl_Position = ProjectionMatrix*ModelViewMatrix*vec4(positionin,1.0);
23
24   //Atributos por vértice
25   normal = normalin;
26   texcoords = texcoordsin;
27
28   //Atributos del texturizado
29   height = positionin.y/heightscale;//Altura escalada al rango [0,1]
30   slope = 1.0-normalin.y;
31
32   sundir = normalize(sunpos-positionin);//Dirección hacia el sol
33   //Distancia a la cámara
34   dist = length(ModelViewMatrix*vec4(positionin,1.0));
35 }
```

Código 5.2: Algoritmo para visualizar el terreno *(vertex shader)*

```
1  #version 330
2
3  //Texturas
4  uniform sampler2DArray materials;//Texturas
5  uniform sampler2DArray detailmaps;//Mapas de detalle
6
7  //Opciones para activar
8  uniform bool usetriplanar;//Proyección triplanar
9  uniform bool usedetailmap;//Aplicar mapas de detalle
10 uniform bool useillumination;//Iluminación
11
12 //Parámetros Texturizado
13 uniform int attr;//Tipo de atributo (altura, pendiente)
14 uniform int nmaterials;//Número de materiales
15 uniform float boundaries[4];//Valor máximo del rango de cada material
16 uniform float tiling;//Repetición de la textura
17 uniform float detailtiling;//Repetición del mapa de detalle
18 uniform float detaildist;//Distancia del efecto del mapa de detalle
19
20 //Parámetros Iluminación
21 uniform vec3 skycolor;//Color del cielo
22 uniform vec3 suncolor;//Color del Sol
23
24 in vec3 texcoords;//Coordenadas de textura del vértice
25 in vec3 normal;//Normal del vértice
26 in float height;//Altura normalizada del vértice
27 in float slope;//Pendiente del vértice
28 in vec3 sundir;//Dirección hacia el Sol
29 in float dist;//Distancia entre el vértice y la cámara
30
31 out vec4 color;//Color del fragmento
32
33 /*Calcula la proyección triplanar para el Array Texture y las
34 coordenadas de textura*/
```

```
35 vec3 TriplanarProjection(sampler2DArray sampler,vec3 coords,int index)↩
       {
36   vec3 cxy,cxz,cyz,weight;
37
38   //Proyecciones
39   cxy = texture(sampler,vec3(coords.xy,index)).rgb;
40   cxz = texture(sampler,vec3(coords.xz,index)).rgb;
41   cyz = texture(sampler,vec3(coords.zy,index)).rgb;
42
43   weight = normal*normal;
44   weight /= (weight.x+weight.y+weight.z);
45
46   return (cyz*weight.x+cxz*weight.y+cxy*weight.z);
47 }
48
49 /*Devuelve el color del material index+1, tomando en cuenta el mapa de
50 detalle asociado y la proyección triplanar (si la opciones estan
51 activadas)*/
52 vec3 GetColor(int index){
53
54   vec3 col,detcol;
55
56   //Aplica la textura asociada al material index+1
57   if(usetriplanar) //Proyección triplanar
58     col = TriplanarProjection(materials,tiling*texcoords,index);
59   else //Proyección planar estándar
60     col = texture(materials,vec3(tiling*texcoords.xz,index)).rgb;
61
62   //Aplica el mapa de detalle asociado al material index+1
63   if(usedetailmap){
64
65     if(usetriplanar) //Proyección triplanar
66       detcol = TriplanarProjection(detailmaps,detailtiling*texcoords,↩
             index);
67     else //Proyección planar estándar
68       detcol = texture(detailmaps,vec3(detailtiling*texcoords.xz,index↩
             )).rgb;
69
70     //Aplica el mapa de detalle hasta la distancia detaildist
71     detcol = mix(detcol,vec3(0.5),clamp(dist/detaildist,0,1));
72
73     //Mezcla el mapa de detalle con la textura
74     col = clamp(col+detcol-0.5,0.0,1.0);
75   }
76
77   return col;
78 }
79
80 void main()
81 {
82   vec3 col;//Color
83
84   //-----------------------------------Texturizado
85   float val,val2;
86   int i;
87
88   //Define el tipo de atributo del texturizado
89   if(attr == 0) val = height;//Texturizado basado en altura
90   if(attr == 1) val = slope;//Texturizado basado en pendiente
91
92   //Define el material en base al valor máximo del rango
93   for(i = 0; i < nmaterials-1;i++) if(val <= boundaries[i]) break;
94
95   if(i == 0){//Primer material
96
97     col = GetColor(0);
98
99   }else{
```

```
100
101     //Valor máximo del rango del material antecesor
102     val2 = boundaries[i-1];
103
104     if(val-val2 < 0.05){//Frontera (mezcla lineal)
105
106       col = mix(GetColor(i-1),GetColor(i),smoothstep(val2,val2+0.05,↩
              val));
107
108     }else{//Material i+1
109
110       col = GetColor(i);
111     }
112   }
113
114   //-----------------------------------Iluminación
115   if(useillumination){
116
117     vec3 light = skycolor+suncolor*max(dot(normal,sundir),0.0);
118     col *= light;
119   }
120
121   //-----------------------------------Corrección Gamma
122   col = pow(col,vec3(1.0/2.2));
123
124   color = vec4(col,1.0);
125 }
```

Código 5.3: Algoritmo para visualizar el terreno *(frament shader)*

5.5 Implementación del cielo

El cielo es modelado mediante *Sky Domes*. En la sección 5.5.1 se describe el algoritmo para construir la esfera o domo basado en el tutorial presentado por Medina [32], y en la sección 5.5.2 se explica la visualización del cielo basada en *shaders* que incorpora características como el Sol y la corrección gamma, basado en el *shader* presentado por Quilez en Shadertoy [48].

5.5.1 Generación de la esfera

La generación de la esfera se ejecuta mediante el método *GenerateSphere* de la clase *Sky*, mostrado en el código 5.4. La esfera se define en base al número de meridianos *(meridians)* y paralelos *(paralells)* que representan el número de divisiones verticales y horizontales respectivamente, estableciendo la resolución de la misma. Para describir las posiciones de los vértices en la esfera se utilizan las coordenadas esféricas considerando el sistema de coordenadas de OpenGL:

$$\begin{aligned} x &= rsin(\theta)sin(\phi) \\ y &= rcos(\phi) \\ z &= rcos(\theta)sin(\phi) \end{aligned} \tag{5.4}$$

donde r es el radio de la esfera, $0 \leq \theta \leq 2\pi$ es el ángulo azimutal y $0 \leq \phi \leq \pi$ es el ángulo polar. El algoritmo, construye una esfera unitaria recorriendo progresivamente cada ϕ y θ en incrementos asociados a *paralells* y *meridians* respectivamente, calculando las coordenadas cartesianas del vértice y almacenándolas en el arreglo *positions*. Luego, almacena en el arreglo *indexes* los índices de cada vértice de cada cuadrilátero que conforma la esfera, definiendo el mallado de la esfera.

El procedimiento para asignar los índices es similar al caso de los terrenos explicado en la sección 5.1, pero en este caso los índices se generarían hacia la izquierda ya que θ en la ecuación 5.4 define las coordenadas cartesianas en dirección *counterclockwise*. Para trabajar similar al caso mostrado en la figura 5.1, se recorre θ en decrementos (cambio de dirección a *clockwise*) asociados a *meridians* en vez de incrementos. Si se desea construir un domo en vez de una esfera se trabaja con $0 \leq \phi \leq \frac{\pi}{2}$ en vez de $0 \leq \phi \leq \pi$.

```
1  void Sky::GenerateSphere(int meridians,int parallels){
2
3    int i,j,c1,c2,a,b,c,d;
4    float theta,phi,dtheta,dphi;
5
6    dtheta = 2.0f*PI/meridians;//Incremento ángulo azimutal
7    dphi = PI/parallels;//Incremento ángulo polar
8    c1 = c2 = 0;//Índices (positions e indexes respectivamente)
9    parallels++;//Agrega el polo a la parte inferior de la esfera
10   nvertexes = meridians*parallels;//Número de vértices
11   nindexes = 6*meridians*(parallels-1);//Número de índices
12
13   positions = new glm::vec3[nvertexes];
14   indexes = new GLuint[nindexes];
15
16   for(i = 0,phi = 0.0f;i < parallels;i++,phi += dphi){
17
18     for(j = 0,theta = 0.0f;j < meridians; j++,theta -= dtheta){
19
20       //Calcula el vértice de la esfera
21       positions[c1++] = glm::vec3(sin(theta)*sin(phi),cos(phi),cos(↩
             theta)*sin(phi));
22
23       //Calcula el mallado
24       if(i < parallels-1){
25
26         a = i*meridians+j;
27         b = i*meridians+(j+1)%meridians;
28         c = a+meridians;
29         d = b+meridians;
30
31         //Triangulo superior: (a,c,b)
32         indexes[c2++] = a;
33         indexes[c2++] = c;
34         indexes[c2++] = b;
35
36         //Triangulo inferior: (b,c,d)
37         indexes[c2++] = b;
38         indexes[c2++] = c;
39         indexes[c2++] = d;
40       }
41     }
```

```
42    }
43  }
```

Código 5.4: Algoritmo para generar esferas (método *GenerateSphere* de la clase *Sky*)

5.5.2 Visualización del cielo

La visualización del cielo se ejecuta mediante el método *Draw* de la clase *Sky* utilizando *shaders*. Similar a la visualización del terreno, los datos necesarios (posiciones de los vértices) son cargados en la GPU mediante el método *LoadGPU*. Los códigos 5.5 y 5.6 muestran el *vertex shader* y el *fragment shader* respectivamente.

El *vertex shader* transforma la posición del vértice a espacio de imagen multiplicando *positionin* (atributo por vértice) por las matrices de transformación. Las transformaciones aplicadas al modelo, incluyen trasladar la esfera al centro del terreno y escalarla a un radio que abarque todo el terreno. Además, envía al *fragment shader* la distancia del vértice al Sol mediante la variable *dist*, que se calcula en base a la posición del Sol en coordenadas de objeto a través de la variable uniforme *sunpos*.

El *fragment shader* primero inicializa el color del fragmento al color del cielo proporcionado en la variable uniforme *skycolor*. Luego, modela el Sol mediante la funcion exponencial $e^{-\alpha d}$, donde d representa la distancia del Sol a la posición del vértice y α controla el radio del Sol, a mayor valor menor radio. Entonces, se agrega el Sol mezclando el color actual del fragmento con el color del Sol (variable uniforme *suncolor*) mediante una interpolación lineal utilizando como peso la función exponencial. Este procedimiento se aplica para tres radios de mayor a menor de manera tal de producir el resplandor de Sol en cierta medida (sol brillante y cierto brillo alrededor de él). Los radios fueron determinados empíricamente. Por último, aplica una corrección gamma similar al caso de la visualización del terreno.

```
1  #version 330
2
3  //Atributos por vértice
4  layout(location = 0) in vec3 positionin;
5
6  //Matrices de transformación
7  uniform mat4 ProjectionMatrix,ModelViewMatrix;
8  uniform vec3 sunpos;//Posición del Sol
9
10 out float dist;//Distancia entre el vértice y el Sol
11
12 void main()
13 {
14   gl_Position = ProjectionMatrix*ModelViewMatrix*vec4(positionin,1.0);
15
16   dist = distance(positionin,sunpos);//Distancia al Sol
```

```
}
```

Código 5.5: Algoritmo para visualizar el cielo *(vertex shader)*

```
#version 330

uniform vec3 skycolor, suncolor;//Color del cielo y Color del Sol

in float dist;//Distancia al Sol

out vec4 outcolor;//Color del fragmento

void main()
{
  vec3 color = skycolor;//Color del cielo constante

  //Agrega el Sol
  color = mix(color,suncolor,smoothstep(0.0,1.0,exp(-64.0*dist)));
  color = mix(color,suncolor,smoothstep(0.0,1.0,exp(-32.0*dist)));
  color = mix(color,suncolor,smoothstep(0.0,1.0,exp(-16.0*dist)));

  color = pow(color,vec3(1.0/2.2));//Corrección Gamma

  outcolor = vec4(color,1.0);
}
```

Código 5.6: Algoritmo para visualizar el cielo *(fragment shader)*

Capítulo 6

Implementación del Generador

En este capítulo se explicará la implementación del Generador de mapas de altura, constituido por algoritmos de generación de terrenos basados en el enfoque síntesis de ruido. En cada algoritmo, se implementa la extensión de síntesis de ruido propuesta por De Carpentier y Bidarra [5], donde se incorporan las transformaciones asociadas al pre-procesamiento T_{pre} y post-procesamiento T_{post}.

El capítulo se divide en tres secciones: en la sección 6.1, se explicará cada algoritmo de generación de terrenos implementado; seguidamente, en la sección 6.2 se describirán cada transformación disponible; y por último, en la sección 6.3 se explicarán las implementaciones de las funciones base incorporadas.

6.1 Algoritmos de generación de terrenos

La generación de los mapas de altura se ejecuta mediante el método *Generate* de la clase *Terrain*. Éste, se encarga de generar los valores asociados a la matriz *heightmap* (mapa de altura); al evaluar en cada vértice algunos de los algoritmos de generación disponibles, considerando las transformaciones involucradas.

Primero, se cargan en variables los parámetros asociados al Generador almacenados en el objeto *conf* de la clase *GeneratorConf*. Luego, se recorre cada vértice evaluando la altura con el algoritmo seleccionado, considerando el tamaño de las características y aplicando la transformación del dominio. Dado que el rango de valores generado es muy diverso, se calcula el mínimo y el máximo para poder normalizar la altura. Después, se recorren nuevamente los vértices normalizando la altura y se aplica la transformación del rango.

El Generador incluye cuatro algoritmos: en la sección 6.1.1, se explica el algoritmo básico para construir un fBm planteado por Musgrave en [11]; en las secciones 6.1.2 y 6.1.3 se describen detalladamente dos algoritmos multifractales planteados por Musgrave en [11], que extienden fBm procedimental para producir terrenos heterogéneos; y la última sección 6.1.4, explica el algoritmo basado en el gradiente local de la función base planteado por Quilez en [47], que extiende fBm procedimental para producir terrenos más complejos y naturales.

6.1.1 fBm procedimental

El algoritmo básico para calcular síntesis de ruido es definido en el método *fBm* de la clase *Terrain*, mostrado en el código 6.1. Este algoritmo calcula la siguiente ecuación:

$$H(p) = \sum_{i=0}^{n-1} \frac{N(\lambda^i p)}{\lambda^{iH}} \tag{6.1}$$

donde N es la función base (ruido de valor o ruido de Perlin), λ (*lacunarity* en el código) es la lacunaridad, H es el exponente de Hurst y n (*bands* en el código) define el número de bandas a calcular.

Para lograr un código compacto, se establece *nfunction* a apuntar al método correspondiente del objeto *noise* de la clase *Noise* para que calcule el tipo de ruido indicado en la variable *noisetype*.

```
1  float Terrain::fBm(glm::vec2 p, float lacunarity,float H,int bands,↩
       Noise *noise,int noisetype){
2
3    float value,amp,n;
4    //Tipo de funcion base
5    float (Noise::*nfunction)(glm::vec2) = (noisetype == 0) ? &Noise::↩
         ValueNoise: &Noise::PerlinNoise;
6
7    //Inicialización
8    value = 0.0f;
9
10   //Construcción del fractal
11   for(int i = 0; i < bands; i++){
12
13     n = (noise->*nfunction)(p);//Función base
14
15     //Disminuye la amplitud basado en la dimensión fractal
16     amp = pow(lacunarity,-i*H);
17
18     value += amp*n;//Acumula la altura
19
20     p *= lacunarity;//Incrementa la frecuencia
21   }
22
23   return value;
24 }
```

Código 6.1: Algoritmo básico de síntesis de ruido para generar terrenos (método *fBm* de la clase *Terrain*)

6.1.2 Multifractal heterogéneo

El multifractal heterogéneo es calculado mediante el método *HeteroMultifractal* de la clase *Terrain*, mostrado en el código 6.2. El objetivo de este algoritmo es lograr que regiones bajas sean más suaves producto de la acumulación de materiales, mientras regiones elevadas sean más rugosas producto de procesos erosivos.

Este método extiende al fBm procedimental, mediante escalar el incremento de cada banda por el valor actual de la función (altura actual). Por lo tanto, en regiones cercanas a la elevación 0 (denominado nivel del mar por Musgrave) los valores de las bandas de alta frecuencia serán amortiguados y por ende permanecerán suaves, mientras regiones elevadas no lo serán, por lo que al progresar las iteraciones serán más rugosas. El parámetro *offset* desplaza el nivel del mar, originalmente definido en la elevación 0. Por ejemplo, si *offset* es 0,3 el nuevo nivel del mar será la elevación $-0{,}3$.

```
1  float Terrain::HeteroMultifractal(glm::vec2 p,float lacunarity,float H↩
       ,int bands,float offset,Noise *noise,int noisetype){
2
3    float value,amp,n;
4    //Tipo de funcion base
5    float (Noise::*nfunction)(glm::vec2) = (noisetype == 0) ? &Noise::↩
         ValueNoise: &Noise::PerlinNoise;
6
7    //Primera banda no escalada
8    n = (noise->*nfunction)(p);
9
10   value = (n+offset);
11   p *= lacunarity;
12
13   // Construcción del fractal
14   for(int i = 1; i < bands; i++){
15
16     n = (noise->*nfunction)(p);//Función base
17
18     //Disminuye la amplitud basado en la dimensión fractal
19     amp = pow(lacunarity,-i*H);
20
21     /*Acumula la altura, el incremento es escalado por el valor actual
22     de la función*/
23     value += value*amp*(n+offset);
24
25     p *= lacunarity;//Incrementa la frecuencia
26   }
27
28   return value;
29 }
```

Código 6.2: Extensión multifractal de *fBm* procedimental para generar terrenos

heterogéneos (método *HeteroMultifractal* de la clase *Terrain*)

6.1.3 Multifractal híbrido

El multifractal híbrido es calculado mediante el método *HybridMultifractal* de la clase *Terrain*, mostrado en el código 6.3. El objetivo de este algoritmo es lograr que los valles sean suaves en todas sus elevaciones y no únicamente en las elevaciones cercanas al nivel del mar.

Este método extiende al multifractal heterogéneo, mediante escalar el peso que pondera el valor de cada banda por el valor de la banda previa. Por lo tanto, el peso disminuye monótonamente cada iteración, reduciendo el valor de las bandas de alta frecuencia y por ende reduciendo la rugosidad. El peso (w en el código) no debe ser mayor que 1 para evitar la divergencia de la suma a medida que el valor aumenta.

```
1  float Terrain::HybridMultifractal(glm::vec2 p,float lacunarity,float H↩
       ,int bands,float offset,Noise *noise,int noisetype){
2
3    float value,amp,n,w,aux;
4    //Tipo de funcion base
5    float (Noise::*nfunction)(glm::vec2) = (noisetype == 0) ? &Noise::↩
         ValueNoise: &Noise::PerlinNoise;
6
7    //Primera banda no escalada
8    n = (noise->*nfunction)(p);
9
10   w = value = (n+offset);
11   p *= lacunarity;
12
13   //Construcción del fractal
14   for(int i = 1; i < bands; i++){
15
16     if(w > 1.0) w = 1.0;//Previene la divergencia
17
18     n = (noise->*nfunction)(p);//Función base
19
20     //Disminuye la amplitud basado en la dimensión fractal
21     amp = pow(lacunarity,-i*H);
22
23     aux = amp*(n+offset);//Valor de la banda actual
24     value += w*aux;//Acumula la altura ponderada por el peso
25
26     /*Actualiza el peso al escalarlo por el valor de la banda previa
27     (el peso disminuye monótonamente)*/
28     w *= aux;
29     p *= lacunarity;//Incrementa la frecuencia
30   }
31
32   return value;
33 }
```

Código 6.3: Extensión del multifractal heterogéneo propuesta por Musgrave para generar terrenos más heterogéneos y más realistas (método *HybridMultifractal* de la clase *Terrain*)

6.1.4 Turbulencia de Quilez

La turbulencia de Quilez es calculada mediante el método *QuilezTurbulence* de la clase *Terrain*, mostrado en el código 6.4. Éste es una variante de *fBm* procedimental, donde se utiliza el gradiente local (derivadas parciales) de la función base para suprimir la rugosidad en pendientes pronunciadas detectadas en bandas previas. Aunque el autor indica que calcula el ruido de Perlin y su gradiente local, en realidad el cálculo corresponde a un ruido de valor y su gradiente local. Quilez implementa esta variante en el *shader* que publica en Shadertoy [48]

Para controlar la rugosidad dependiendo del gradiente local, se utiliza la longitud o magnitud del gradiente local acumulado (d en el código) para escalar el incremento de cada banda al dividir el incremento por uno más el cuadrado de esta longitud. Por lo tanto, cuando esta longitud sea grande (inclinación pronunciada) el valor será amortiguado tanto para la banda actual como las subsecuentes, produciendo un terreno más suave. En caso contrario, el valor no será amortiguado y en consecuencia el terreno será más rugoso.

```
1  float Terrain::QuilezTurbulence(glm::vec2 p,float lacunarity,float H,↩
       int bands,Noise *noise,int noisetype){
2
3    float value,amp;
4    glm::vec2 d;
5    glm::vec3 n;
6    //Tipo de funcion base
7    glm::vec3 (Noise::*nfunction)(glm::vec2) = (noisetype == 0) ? &Noise↩
         ::ValueNoiseDeriv: &Noise::PerlinNoiseDeriv;
8
9    //Inicialización
10   value = 0.0f;
11   d = glm::vec2(0.0f);
12
13   //Construcción del fractal
14   for(int i = 0; i < bands; i++){
15
16     n = (noise->*nfunction)(p);//Función base y su gradiente local
17
18     //Acumula las derivadas parciales de la función base
19     d += glm::vec2(n.y,n.z);
20
21     //Disminuye la amplitud basado en la dimensión fractal
22     amp = pow(lacunarity,-i*H);
23
24     /*Acumula la altura, el incremento es escalado por la longitud del
25     gradiente local*/
26     value += amp*n.x/(1.0f+glm::dot(d,d));
27
28     p *= lacunarity;//Incrementa la frecuencia
29   }
30
31   return value;
32 }
```

Código 6.4: Extensión de *fBm* procedimental propuesta por Quilez para generar terrenos más complejos y más naturales (método *QuilezTurbulence* de la clase *Terrain*)

6.2 Transformaciones

El Generador incluye dos tipos de transformaciones: pre-procesamiento T_{pre} y post-procesamiento T_{post}; T_{pre} transforma el punto de entrada p antes de evaluar el algoritmo de generación seleccionado; y T_{post} transforma la salida del algoritmo de generación seleccionado, esta salida es previamente normalizada antes de la transformación.

La transformación asociada al pre-procesamiento incluida en el Generador, es la distorsión del dominio. La distorsión del dominio desplaza el punto de entrada p mediante un ruido, expresado matemáticamente como:

$$T_{pre}(x, y) = \Big(x + \alpha N_1(x, y), y + \alpha N_2(x, y)\Big) \tag{6.2}$$

donde N_1 y N_2 son ruidos generados con semillas distintas. $0 \leq \alpha \leq 1$ controla la intensidad de la distorsión, a mayor valor mayor distorsión. La distorsión del dominio es calculada mediante el método *DomainDistortion* basado en la ecuación 6.2. El método considera el tipo de ruido utilizado: ruido de valor o ruido de Perlin.

El Generador incluye tres transformaciones asociadas al post-procesamiento, identificadas por las características que generan: efecto glaciar, efecto cañón y efecto meseta. El efecto glaciar, produce que las regiones bajas del terreno sean más bajas y planas, mientras que las regiones altas sean más empinadas; el efecto cañón, aplana tanto las regiones bajas como altas y aumenta la inclinación entre ambas produciendo acantilados; y el efecto meseta, produce mesetas en las elevaciones medias al aplanar las regiones bajas y altas, además aumenta la inclinación entre regiones bajas y las mesetas.

Estas transformaciones están inspiradas en las curvas preestablecidas en el dispositivo *Curves (Filter Device)* de World Machine [51], cuyas curvas están basadas en la función *bias* y *gain*.

El efecto glaciar es modelado mediante la función *bias* con $0 \leq b \leq 0{,}5$:

$$bias_b(t) = t^{\frac{log(b)}{log(0{,}5)}} \tag{6.3}$$

donde t va representar la altura normalizada del terreno y b va a controlar la intensidad del efecto. Cuando $b = 0{,}5$ no hay efecto y a medida que su valor disminuye, progresivamente el efecto se hace más intenso. El efecto glaciar es calculado mediante el método *GlacierEffect*.

El efecto cañón es modelado mediante la función *gain* con $0{,}5 \leq g \leq 1{,}0$:

$$gain_g(t) = \begin{cases} \frac{bias_{1-g}(2t)}{2} & : t < 0{,}5 \\ 1 - \frac{bias_{1-g}(2-2t)}{2} & : t \geq 0{,}5 \end{cases} \tag{6.4}$$

donde t va representar la altura normalizada del terreno y g va a controlar la intensidad del efecto. Cuando $g = 0{,}5$ no hay efecto y a medida que su valor aumenta, progresivamente el efecto se hace más intenso. El efecto cañón es calculado mediante el método *CanyonEffect*.

El efecto meseta es modelado mediante una variante de la función *gain* con $0{,}5 \leq g \leq 1{,}0$:

$$gain2_g(t) = \begin{cases} \frac{bias_{1-g}(2t)}{2} & : t < 0{,}5 \\ 1 - \frac{bias_g(2-2t)}{2} & : t \geq 0{,}5 \end{cases} \tag{6.5}$$

donde t va representar la altura normalizada del terreno y g va a controlar la intensidad del efecto. Cuando $g = 0{,}5$ no hay efecto y a medida que su valor aumenta, progresivamente el efecto se hace más intenso. El efecto meseta es calculado mediante el método *PlateauEffect*.

6.3 Funciones base

El Generador incluye dos tipos de funciones base: el ruido de valor y el ruido de Perlin. En las secciones 6.3.1 y 6.3.2 se explican los algoritmos para calcular cada tipo respectivamente, incluyendo el cálculo analítico del gradiente local, considerando que es necesario para la turbulencia de Quilez.

El cálculo analítico de las derivadas parciales (gradiente local) del ruido de valor y de Perlin son presentadas por Quilez en [47] y De Carpentier en [8] respectivamente. En este documento se calculó el gradiente analítico de ambas funciones base.

En el caso del ruido de Perlin, el procedimiento es más sencillo que el procedimiento presentado por De Carpentier y en consecuencia la ecuación resultante es una versión factorizada. Los detalles del procedimiento se encuentran en el apéndice A.

6.3.1 Ruido de valor

El ruido de valor es calculado mediante el método *ValueNoise* de la clase *Noise*, mostrado en el código 6.5. Éste calcula el ruido asociado al punto $p = (x, y)$, mediante la interpolación bilineal de los valores pseudoaleatorios asociados a los vértices más cercanos a p en la cuadrícula. La figura 6.1 ilustra la celda de la cuadrícula que contiene al punto p con la notación utilizada en el código.

Primero, se define las coordenadas (i, j) del vértice a mediante la función piso y las distancias u y v relativas al vértice a. A partir de estas coordenadas,

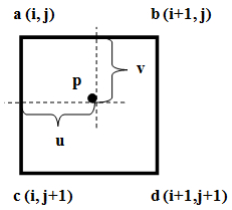


Figura 6.1: Celda en la cuadrícula de valores que contiene al punto p. u y v denotan la distancia relativa al vértice a en el eje x y en el eje y respectivamente.

se derivan las coordenadas de los otros vértices (b, c, d). Luego, se obtiene el valor de cada vértice almacenado en el arreglo *values*. Para evitar la correlación entre los índices de vértices vecinos, se aplica una permutación a cada coordenada basado en la permutación pseudoaleatoria de los índices almacenada en el arreglo P. Finalmente, se aplica la interpolación bilineal basada en la función *Lerp*, que calcula $f(t, a, b) = a + t(b - a)$. Para obtener una interpolación más suave, los pesos de la interpolación lineal son suavizados mediante la función *Fade*, que calcula $f(t) = 6t^5 - 15t^4 + 10t^3$.

Los valores pseudoaleatorios pertenencientes al rango $[-1; 1]$ son calculados mediante el método *SetValueNoise*, basado en la función *rand* de C++. La permutación pseudoaleatoria corresponde a la permutación utilizada por Perlin en [39].

```
1  float Noise::ValueNoise(glm::vec2 p){
2
3    int i,j;
4    float a,b,c,d,u,v,wu,wv;
5
6    /*Calcula las coordenadas del vértice a (esquina superior de la
7    celda) y calcula la distancia relativa al vértice a*/
8    i = (int)floor(p.x);
9    u= p.x-i;
10
11   j = (int)floor(p.y);
12   v = p.y-j;
13
14   //Repite los valores cada 256 índices en cada dimensión
15   i = i & 255;//i%256
16   j = j & 255;//j%256
17
18   //Obtiene los 4 valores asociados a cada vértice de la celda
19   a = values[P[i+P[j]]];
20   b = values[P[i+1+P[j]]];
21   c = values[P[i+P[j+1]]];
22   d = values[P[i+1+P[j+1]]];
23
24   //Aplica la función de mezcla de quinto orden
25   wu = Fade(u);
26   wv = Fade(v);
27
28   //Interpolación bilineal
```

```
29    return Lerp(wv,Lerp(wu,a,b),Lerp(wu,c,d));
30  }
```

Código 6.5: Algoritmo para generar ruido de valor (método *ValueNoise* de la clase *Noise*)

El ruido de valor junto su gradiente local, es calculado mediante el método *ValueNoiseDeriv* de la clase *Noise*, mostrado en el código 6.6. El método es similar a *ValueNoise*, pero incluye el cálculo analítico de las derivadas parciales basado en la ecuación A.6. Éste retorna un vector tridimensional, donde la coordenada x representa el valor, la coordenada y la derivada parcial con respecto a x y la coordenada z la derivada parcial con respecto a y. La derivada de la función de mezcla es calculada mediante la función *FadeD*, que calcula $f(t) = 30t^4 - 60t^3 + 30t^2$.

```
1   glm::vec3 Noise::ValueNoiseDeriv(glm::vec2 p){
2
3     int i,j;
4     float a,b,c,d,u,v,wu,wv,du,dv;
5     glm::vec3 res;
6
7     /*Calcula las coordenadas del vértice a (esquina superior de la
8     celda) y calcula la distancia relativa al vértice a*/
9     i = (int)floor(p.x);
10    u = p.x-i;
11
12    j = (int)floor(p.y);
13    v = p.y-j;
14
15    //Repite los valores cada 256 índices en cada dimensión
16    i = i & 255;//i%256
17    j = j & 255;//j%256
18
19    //Obtiene los 4 valores asociados a cada vértice de la celda
20    a = values[P[i+P[j]]];
21    b = values[P[i+1+P[j]]];
22    c = values[P[i+P[j+1]]];
23    d = values[P[i+1+P[j+1]]];
24
25    //Aplica la función de mezcla de quinto orden
26    wu = Fade(u);
27    wv = Fade(v);
28
29    //Derivada de la función de mezcla
30    du = FadeD(u);
31    dv = FadeD(v);
32
33    //Interpolación bilineal
34    res.x = a+(b-a)*wu+(c-a)*wv+(a-b-c+d)*wu*wv;//Valor
35    res.y = ((b-a)+(a-b-c+d)*wv)*du;//Derivada parcial con respecto a x
36    res.z = ((c-a)+(a-b-c+d)*wu)*dv;//Derivada parcial con respecto a y
37
38    return res;
39  }
```

Código 6.6: Algoritmo para generar ruido de valor junto su gradiente local (método *ValueNoiseDeriv* de la clase *Noise*)

6.3.2 Ruido de Perlin

El ruido de Perlin es calculado mediante el método *PerlinNoise* de la clase *Noise*, mostrado en el código 6.7. El mismo, aplica un procedimiento similar al ruido de valor, pero asigna un gradiente unitario pseudoaleatorio a cada vértice de la cuadrícula. Luego, define el valor asociado a los vértices de la celda que contiene al punto p, como el producto punto entre el gradiente asociado al vértice y el vector del vértice al punto p. Finalmente, estos valores son interpolados mediante una interpolación bilineal similar al caso del ruido de valor.

Los gradientes unitarios pseudoaleatorios almacenados en el arreglo *gradients* son calculados mediante el método *SetPerlinNoise*. Éste calcula el ángulo polar de forma pseudoaleatoria, basado en la función *rand* de C++ y luego calcula las coordenadas cartesianas del vector, basado en la definición de las coordenadas polares.

```
1  float Noise::PerlinNoise(glm::vec2 p){
2
3    int i,j;
4    float a,b,c,d,u,v,wu,wv;
5    glm::vec2 ga,gb,gc,gd;
6
7    /*Calcula las coordenadas del vértice a (esquina superior de la
8    celda) y calcula la distancia relativa al vértice a*/
9    i = (int)floor(p.x);
10   u = p.x-i;
11
12   j = (int)floor(p.y);
13   v = p.y-j;
14
15   //Repite los valores cada 256 indices en cada dimensión
16   i = i & 255;//i%256
17   j = j & 255;//i%256
18
19   //Obtiene los 4 gradientes asociados a cada vértice de la celda
20   ga = gradients[P[i+P[j]]];
21   gb = gradients[P[i+1+P[j]]];
22   gc = gradients[P[i+P[j+1]]];
23   gd = gradients[P[i+1+P[j+1]]];
24
25   /*Obtiene los 4 valores asociados a cada vértice de la celda como el
26   producto punto entre el gradiente determinado y el vector del
27   vértice al punto p*/
28   a = glm::dot(ga,glm::vec2(u,v));
29   b = glm::dot(gb,glm::vec2(u-1.0f,v));
30   c = glm::dot(gc,glm::vec2(u,v-1.0f));
31   d = glm::dot(gd,glm::vec2(u-1.0f,v-1.0f));
32
33   //Aplica la función de mezcla de quinto orden
34   wu = Fade(u);
35   wv = Fade(v);
36
37   //Interpolación Bilineal
38   return Lerp(wv,Lerp(wu,a,b),Lerp(wu,c,d));
39 }
```

Código 6.7: Algoritmo para generar ruido de Perlin (método *PerlinNoise* de la clase *Noise*)

El ruido de Perlin junto su gradiente local, es calculado mediante el método *PerlinNoiseDeriv* de la clase *Noise*, mostrado en el código 6.8. El mismo, es similar al método *PerlinNoise*, pero incluye el cálculo analítico de las derivadas parciales basado en la ecuación A.11. El método retorna un vector tridimensional, similar al caso de *PerlinNoiseDeriv*.

```
1  glm::vec3 Noise::PerlinNoiseDeriv(glm::vec2 p){
2
3    int i,j;
4    float u,v,wu,wv,du,dv,a,b,c,d;
5    glm::vec2 ga,gb,gc,gd;
6    glm::vec3 res;
7
8    /*Calcula las coordenadas del vértice a (esquina superior de la
9    celda) y calcula la distancia relativa al vértice a*/
10   i = (int)floor(p.x);
11   u = p.x-i;
12
13   j = (int)floor(p.y);
14   v = p.y-j;
15
16   //Repite los valores cada 256 indices en cada dimensión
17   i = i & 255;//i%256
18   j = j & 255;//j%256
19
20   //Obtiene los 4 gradientes asociados a cada vértice de la celda
21   ga = gradients[P[i+P[j]]];
22   gb = gradients[P[i+1+P[j]]];
23   gc = gradients[P[i+P[j+1]]];
24   gd = gradients[P[i+1+P[j+1]]];
25
26   /*Obtiene los 4 valores asociados a cada vértice de la celda como
27   el producto punto entre el gradiente determinado y el vector del
28   vértice al punto p*/
29   a = glm::dot(ga,glm::vec2(u,v));
30   b = glm::dot(gb,glm::vec2(u-1.0f,v));
31   c = glm::dot(gc,glm::vec2(u,v-1.0f));
32   d = glm::dot(gd,glm::vec2(u-1.0f,v-1.0f));
33
34   //Aplica la función de mezcla de quinto orden
35   wu = Fade(u);
36   wv = Fade(v);
37
38   //Derivada de la función de mezcla
39   du = FadeD(u);
40   dv = FadeD(v);
41
42   //Interpolación bilineal
43   res.x = a+(b-a)*wu+(c-a)*wv+(a-b-c+d)*wu*wv;//Valor
44   res.y = ga.x+(gb.x-ga.x)*wu+(gc.x-ga.x)*wv+(ga.x-gb.x-gc.x+gd.x)*wu*↩
         wv+
45   ((b-a)+(a-b-c+d)*wv)*du;//Derivada parcial con respecto a x
46   res.z = ga.y+(gb.y-ga.y)*wu+(gc.y-ga.y)*wv+(ga.y-gb.y-gc.y+gd.y)*wu*↩
         wv+
47   ((c-a)+(a-b-c+d)*wu)*dv;//Derivada parcial con respecto a y
48
49   return res;
50 }
```

Código 6.8: Algoritmo para generar ruido de Perlin junto su gradiente local (método *PerlinNoiseDeriv* de la clase *Noise*)

Capítulo 7

Pruebas y resultados

En este capítulo se presentarán y analizarán los resultados obtenidos para diferentes pruebas aplicadas, tanto a la implementación del Generador como a la implementación del Visualizador. Primero, en la sección 7.1 se describirán detalladamente los ambientes utilizados para realizar las pruebas. En la sección 7.2 se estudiarán los factores que afectan el desempeño del Generador. Después, en la sección 7.3 se analizarán detalladamente los algoritmos de generación implementados, estudiando la influencia de los parámetros sobre la forma del terreno generado. En la sección 7.4 se analizarán cómo las transformaciones implementadas afectan la forma del terreno generado.

En la sección 7.5 se estudiarán los factores que afectan el desempeño del Visualizador. Finalmente, en la sección 7.6 se revisará el Visualizador y sus características implementadas: visualización del terreno o visualización tridimensional del mapa de altura, texturizado basado en altura o pendiente de múltiples materiales, entre otras.

7.1 Ambiente de pruebas

Las pruebas se realizaron en dos ambientes diferentes, las especificaciones de cada uno fueron:

- Ambiente 1: Intel Core i7-2600K de 3,40 GHz, 16 GB de RAM, sistema operativo Windows 7 de 64 bits. Tarjeta de video Nvidia GeForce GTX 550 Ti.
- Ambiente 2: Intel Core 2 Quad Q9550 de 2,83 GHz, 3,25 GB de RAM, sistema operativo Windows XP de 32 bits. Tarjeta de video Nvidia GeForce 9400 GT. Microsoft Visual Studio 2010. Freeglut 2.8.0.

En la tabla 7.1 se pueden observar algunas de las especificaciones más importantes de ambas tarjetas de video.

Especificación	GTX 550 Ti	9400 GT
Reloj del procesador	1800 MHz	1400 MHz
Ancho de banda de la memoria	98,4 GB/s	12,8 GB/s
Tasa de relleno de texturas	28,8 GTexels/s	4,4 GTexels/s

Tabla 7.1: Especificaciones de las tarjetas de video.

7.2 Rendimiento del Generador

Para evaluar el rendimiento del Generador, se creó un proyecto en Visual Studio con las clases *Noise* y *Terrain*, donde se midió el tiempo de ejecución para las distintas pruebas a través de una clase *Timer*, basada en la implementación de Ahn [1]. La clase *Timer* utiliza las funciones *QueryPerformanceCounter* y *QueryPerformanceFrequency* proporcionadas por el API de Windows, que permiten medir el tiempo de ejecución para el segmento de código determinado con al menos un microsegundo de precisión. Mediante un *script* de Windows se ejecutó 100 veces cada prueba, escribiendo los resultados en un archivo, donde cada prueba fue seleccionada por el parámetro pasado al programa. Al culminar las pruebas, estos archivos fueron cargados y procesados en Excel.

Considerando que en las mediciones aplicadas en el ambiente 1 se detectó una desviación estándar muy alta, donde dependiendo del núcleo utilizado la medición variaba considerablemente, se decidió seleccionar un solo núcleo a través de la función *SetThreadAffinityMask*, seleccionando el núcleo 7 y el núcleo 3 para el ambiente 1 y el ambiente 2 respectivamente.

El rendimiento de cada algoritmo incluido en el Generador depende de tres factores primordiales: las dimensiones del mapa de altura, el número de bandas y la función base. Para analizar la influencia de las dimensiones del mapa de altura, se fijó el número de bandas a 10 y se midió cada algoritmo para diferentes dimensiones utilizando el ruido de Perlin.

En las tablas 7.2 y 7.3 se puede apreciar el tiempo de ejecución promedio obtenido en estas pruebas para el ambiente 1 y el ambiente 2 respectivamente. La figura 7.1 muestra dos gráficos (uno para cada ambiente) de los resultados obtenidos. Para todos los algoritmos y en ambos ambientes, se presenta un crecimiento lineal en el tiempo de ejecución; considerando que cada mapa de altura posee 4 veces más vertices que el mapa de altura de previo, y el tiempo de ejecución requerido es aproximadamente 4 veces mayor que en el mapa de altura previo. Para todas las dimensiones, el ambiente 1 presenta un rendimiento considerablemente superior al ambiente 2, donde para el mapa de altura más

grande (2048×2048) fBm y la turbulencia de Quilez son 171 % y 368 % más rapidos en el ambiente 1 con respecto al ambiente 2.

	Dimensiones del mapa de altura			
Algoritmo	256×256	512×512	1024×1024	2048×2048
fBm	20,85	80,96	324,60	1285,32
Multifractal heterogéneo	20,82	80,75	323,21	1280,84
Multifractal híbrido	43,24	170,03	674,67	2693,74
Turbulencia de Quilez	25,91	101,34	404,40	1605,22

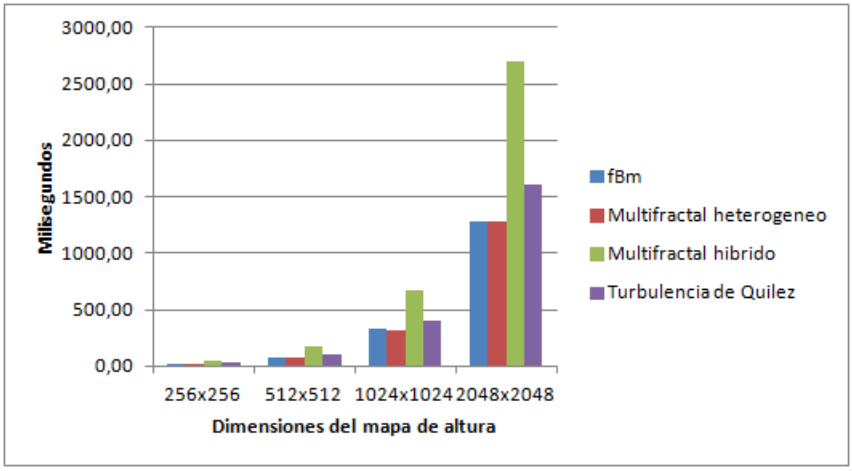
Tabla 7.2: Tiempo de ejecución promedio (ms) de los algoritmos de generación para diferentes dimensiones del mapa de altura (ambiente 1).

	Dimensiones del mapa de altura			
Algoritmo	256×256	512×512	1024×1024	2048×2048
fBm	54,31	218,15	871,79	3483,53
Multifractal heterogéneo	101,28	407,09	1622,79	6489,91
Multifractal híbrido	102,64	412,55	1642,50	6567,99
Turbulencia de Quilez	117,38	470,95	1879,03	7519,86

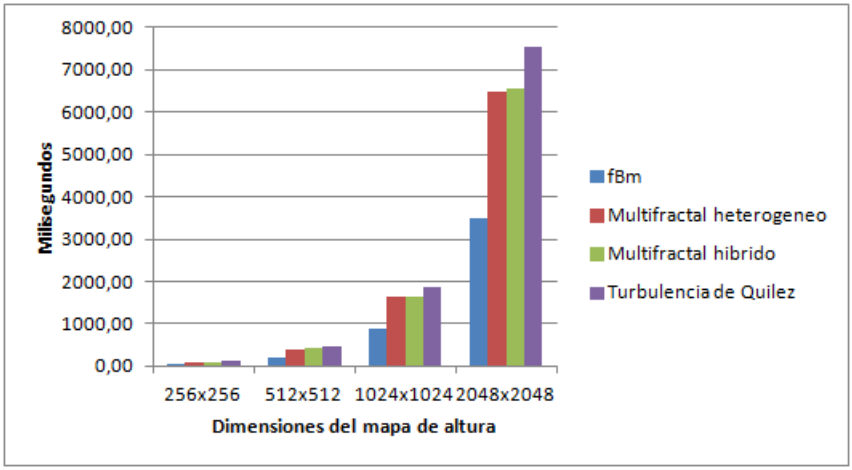
Tabla 7.3: Tiempo de ejecución promedio (ms) de los algoritmos de generación para diferentes dimensiones del mapa de altura (ambiente 2).

Al analizar los gráficos se pueden apreciar resultados que no son los esperados, como el hecho que los ambientes presentan tendencias distintas con respecto a los algoritmos. Por ejemplo, la turbulencia de Quilez en el ambiente 1 es considerablemente más rapida que el multifractal híbrido, mientras en el ambiente 2 es ligeramente más lenta. Tomando en cuenta que la turbulencia de Quilez es un algoritmo más complejo que el multifractal híbrido, se esperaría un tiempo de ejecución mayor.

Al obtener las mediciones en modo *Debug*, ambos ambientes presentan las mismas tendencias. El tiempo de ejecución del multifractal heterogéneo es ligeramente superior al multifractal híbrido, y a su vez fBm es ligeramente superior a ambos algoritmos, con una diferencia en milisegundos. Por otra parte, el tiempo de ejecución de la turbulencia de Quilez es considerablemente superior a los algoritmos mencionados, con una diferencia de segundos. Por lo tanto, los resultados inesperados como los obtenidos para la turbulencia de Quilez en el



(a)



(b)

Figura 7.1: Tiempo de ejecución de los algoritmos de generación para diferentes dimensiones del mapa de altura: (a) Ambiente 1 y (b) Ambiente 2.

ambiente 1, se deben a optimizaciones aplicadas en el modo *Release*.

Con la finalidad de medir el impacto de las transformaciones en los algoritmos, se midió fBm utilizando el ruido de Perlin con las transformaciones computacionalmente más costosas (distorsión del dominio y efecto cañón). Para el mapa de altura más grande (2048×2048), se presenta un incremento de 21 % y 22 % del tiempo de ejecución para el ambiente 1 y el ambiente 2 respectivamente.

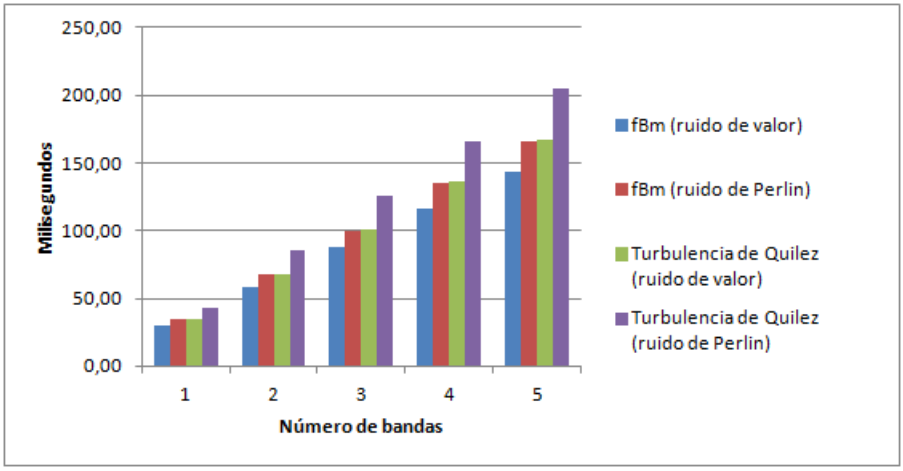
Para analizar el impacto del número de bandas y la función base, se fijó las dimensiones del mapa de altura a 1024×1024 y se midió fBm con ambas funciones base para diferentes número de bandas; en el caso del gradiente de las funciones base, se midió la turbulencia de Quilez en vez de fBm. Inicialmente, las pruebas para medir el rendimiento de las funciones base, consistían en evaluar la función determinada directamente. Sin embargo, al elaborar las pruebas resultó que el tiempo de ejecución era 0, demostrando que la precisión de la clase *Timer* no era suficiente.

En las tablas 7.4 y 7.5 se puede apreciar el tiempo de ejecución promedio obtenido en estas pruebas para el ambiente 1 y el ambiente 2 respectivamente. La figura 7.2 muestra dos gráficos asociados a cada tabla. Para todos los algoritmos y en ambos ambientes, se presenta un crecimiento lineal en el tiempo de ejecución. Para todos los números de bandas, el ambiente 1 presenta un rendimiento considerablemente superior al ambiente 2, donde para el número de bandas más grande (5) fBm y la turbulencia de Quilez con el ruido de Perlin son 165 % y 355 % más rapidos en el ambiente 1 con respecto al ambiente 2.

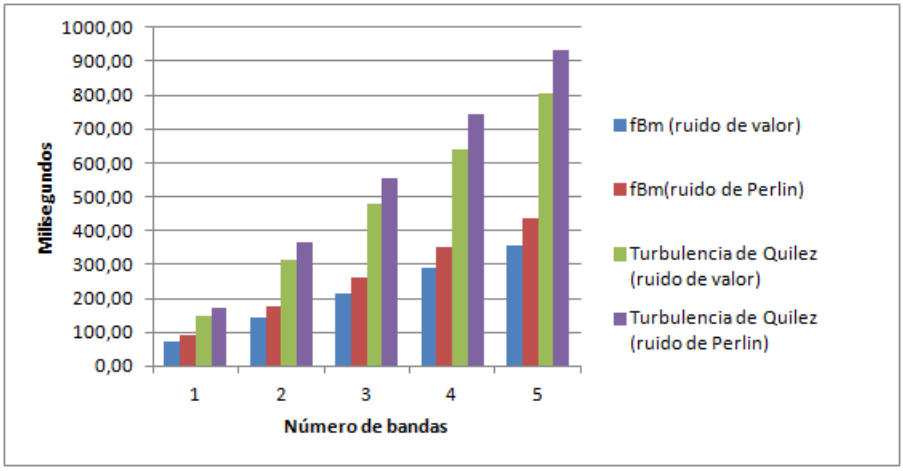
Analizando los gráficos, resultó inesperado el hecho de que en el ambiente 1 la diferencia entre las funciones base es relativamente pequeña, mientras en el ambiente 2 es notable (especialmente entre fBm y la turbulencia de Quilez utilizando el ruido de valor). Similar a las pruebas basadas en las dimensiones del mapa de altura, la discrepancia entre ambientes se deben a optimizaciones aplicadas en el modo *Release*.

	Número de bandas				
Algoritmo	1	2	3	4	5
fBm (ruido de valor)	30,52	58,60	87,65	115,93	143,69
fBm (ruido de Perlin)	34,27	68,15	99,89	134,57	165,31
Turbulencia de Quilez (ruido de valor)	34,81	68,22	101,38	136,51	166,83
Turbulencia de Quilez (ruido de Perlin)	43,45	84,98	125,58	166,24	205,17

Tabla 7.4: Tiempo de ejecución promedio (ms) de los algoritmos de generación para diferentes número de bandas (ambiente 1).



(a)



(b)

Figura 7.2: Tiempo de ejecución de las funciones base para diferentes número de bandas: (a) Ambiente 1 y (b) Ambiente 2.

	Número de bandas				
Algoritmo	1	2	3	4	5
fBm (ruido de valor)	73,48	144,56	216,53	287,97	356,46
fBm (ruido de Perlin)	89,77	176,41	263,13	349,98	438,36
Turbulencia de Quilez (ruido de valor)	148,77	313,26	477,30	642,08	805,30
Turbulencia de Quilez (ruido de Perlin)	173,30	364,57	553,69	744,83	933,09

Tabla 7.5: Tiempo de ejecución promedio (ms) de los algoritmos de generación para diferentes número de bandas (ambiente 2).

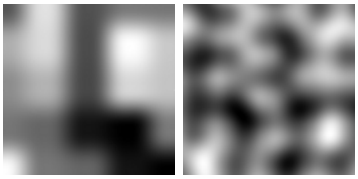
7.3 Algoritmos del Generador

Tomando en cuenta que *fBm* es el algoritmo básico del cual el resto de los algoritmos se extienden, primero se analizó este algoritmo y sus parámetros asociados. *fBm* se basa en cuatro parámetros fundamentales: la función base, el exponente de Hurst H, la lacunaridad λ y el número de bandas n.

Para estudiar la influencia de cada parámetro sobre el terreno generado, se definieron unos valores por defecto para cada parámetro, y en cada prueba se recolectaron los terrenos generados (capturas de pantalla) para diferentes valores del parámetro de interés. En el caso de la función base, se analizó de forma separada mediante el mapa de altura generado. Posteriormente fueron analizados el multifractal heterogéneo y el multifractal híbrido, ambos incorporan el parámetro desplazamiento o. Para estos algoritmos, las pruebas consistieron en estudiar la influencia de o, ya que el resto de los parámetros se comportan similar al caso de *fBm*. Finalmente, se analizó la turbulencia de Quilez que se basa en los mismos parámetros que *fBm*.

Todos los terrenos fueron renderizados desde la misma posición y orientación para una ventana de 800×600 píxeles, utilizando únicamente el factor asociado al componente difuso del modelo de iluminación de Phong y el mapa de navegación correspondiente; ubicando el Sol a una altura de 90° en dirección 0°. La escala del terreno, se definió con una altura máxima de 50 y un tamaño de celda de 1. Para todas las pruebas, se generaron mapas de altura de 256×256 píxeles, con los siguientes valores por defecto: ruido de Perlin como función base con semilla 17, 90 % como el tamaño de las características, $n = 8$, $\lambda = 2{,}0$ y $H = 1{,}0$.

La forma de la función base, determina en gran medida las cualidades visuales del terreno generado, por ende, la selección de la misma es una decisión crucial. El Generador incluye dos funciones base: el ruido de valor y el ruido de Perlin; la figura 7.3 muestra ambos ruidos. Los resultados ratifican los artefactos presentes en el ruido de valor, donde queda en evidencia las celdas de la cuadrícula.



(a) (b)

Figura 7.3: Comparación funciones base: (a) ruido de valor, (b) ruido de Perlin.

(a) (b) (c) (d) (e)

Figura 7.4: Ruido de Perlin para diferentes tamaños de las características: (a) 0 %, (b) 50 %, (c) 75 %, (d) 90 % y (e) 100 %.

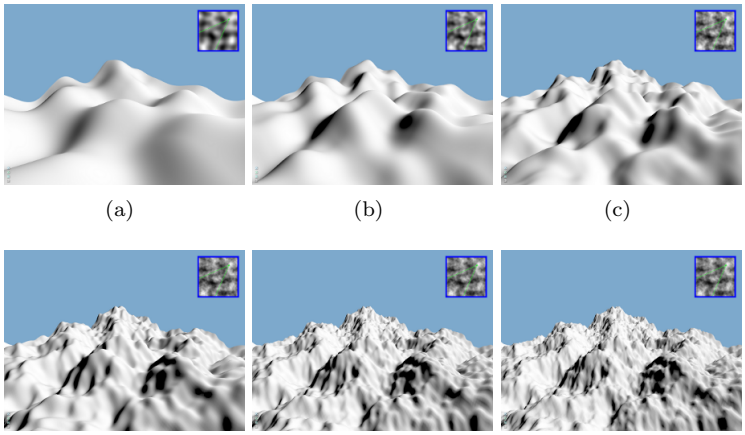
La figura 7.4 permite apreciar como el porcentaje del tamaño de las características afecta el ruido de Perlin, donde al aumentar el porcentaje, se produce una función base con menos características pero de mayor tamaño.

El número de bandas n, específica el número de iteraciones de la construcción. La figura 7.5 ilustra como n afecta el terreno generado; al incrementar n, aumenta la cantidad de nivel de detalle presente en el terreno y a su vez el tiempo de generación (incremento lineal como se demostró en la sección 7.2).

La cantidad de nivel de detalle, puede ser un desperdicio de cómputo. Por ejemplo, el cambio de $n = 5$ a $n = 6$ es prácticamente imperceptible, no obstante, implica un incremento en el tiempo de generación. Además, el exceso del mismo produce *aliasing* espacial. Para un *fBm* de ancho de banda limitado al máximo nivel de detalle representable por un píxel, Musgrave en [11] recomienda un n de aproximadamente 6 a 10, dependiendo de la resolución de la ventana.

La lacunaridad λ, específica la brecha entre las escalas que componen la construcción. La figura 7.6 permite observar como λ afecta el terreno generado; a mayor valor, es posible cubrir un rango de escalas con mayor rapidez, no obstante, en valores muy altos ($\lambda = 5{,}0$ y $\lambda = 10{,}0$) se puede observar cada escala individual del ruido de Perlin, mitigando la calidad visual.

λ normalmente se establece a 2,0, pero Worley en [11] recomienda un va-



(a) (b) (c)

(d) (e) (f)

Figura 7.5: fBm para diferentes número de bandas: (a) $n = 1$, (b) $n = 2$, (c) $n = 3$, (d) $n = 4$, (e) $n = 5$ y (f) $n = 6$.

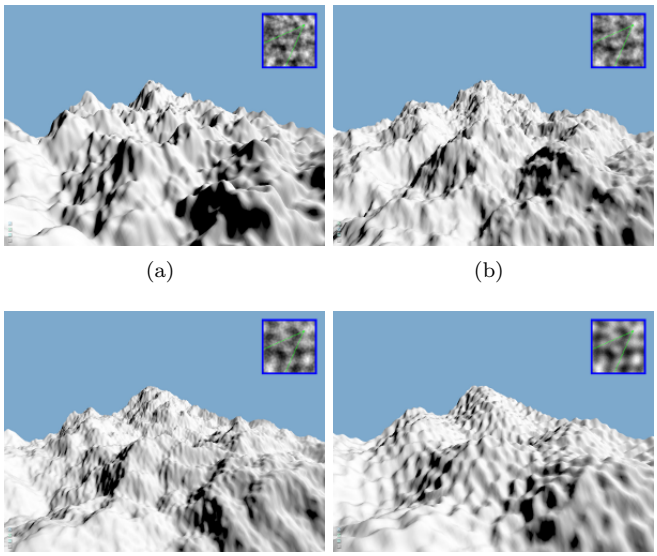
lor cercano con una gran cantidad de dígitos decimales tales como 1,985743 o 2,027473; logrando eliminar la periodicidad, es decir, la siguiente escala más fina, no será exactamente la mitad de la escala anterior.

El exponente de Hurst H, específica la dimensión fractal de la contrucción. La figura 7.7 muestra como H afecta la rugosidad del terreno generado; al aumentar H, produce terrenos más suaves con menos detalles (menor rugosidad). Valores menores o iguales a 0,5, producen resultados visualmente ruidosos.

El multifractal heterogéneo define la rugosidad del terreno en función de cuan cerca esta la posición determinada con respecto al nivel del mar, donde la rugosidad tiende a ser menor. El algoritmo incorpora el parámetro o, que permite desplazar el nivel del mar inicialmente ubicado en la elevación 0; la figura 7.8 permite ilustrar como o afecta el terreno generado.

Los resultados ratifican que las regiones cercanas al nivel del mar son más suaves que el resto. Por otra parte, evidencian un comportamiento extraño, especialmente cuando o es 0,0, donde regiones por debajo del nivel del mar son más rugosas.

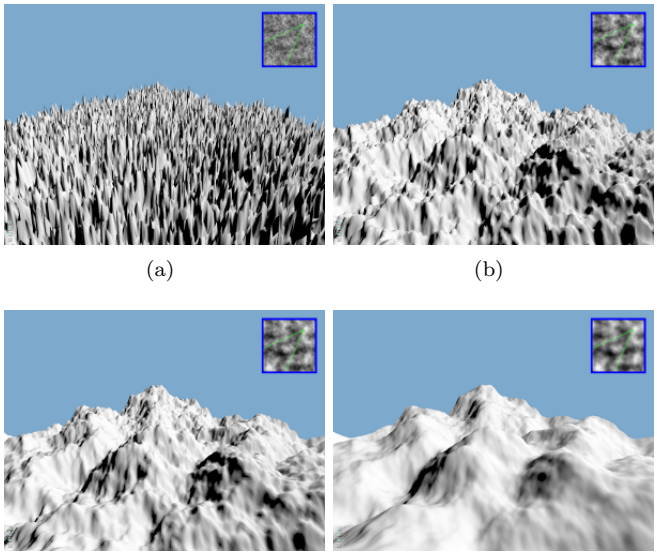
Inspeccionando el algoritmo cuidadosamente y recordando que el ruido de Perlin puede producir valores negativos, es posible notar que el comportamiento del algoritmo no es adecuado cuando el valor del ruido de Perlin desplazado es negativo, ya que no se garantiza que el valor asociado a las bandas sea monótonamente decreciente al avanzar en las iteraciones. Tampoco se puede garantizar cuando la ponderación de la banda sea mayor a 1, como lo indica Musgrave en



(a) (b)

(c) (d)

Figura 7.6: fBm para distintos valores de lacunaridad: (a) $\lambda = 1{,}5$, (b) $\lambda = 2{,}0$, (c) $\lambda = 5{,}0$ y (d) $\lambda = 10{,}0$.



(a) (b)

(c) (d)

Figura 7.7: fBm para distintos valores del exponente de Hurst: (a) $H = 0{,}0$, (b) $H = 0{,}7$, (c) $H = 1{,}0$ y (d) $H = 1{,}5$.

[11] y por lo que recomienda restringir el peso más alto a 1.

Cuando o es 0,5, se garantiza un rango positivo del ruido de Perlin desplazado debido a que el rango del mismo es $[-0{,}49; 0{,}49]$; en consecuencia, el comportamiento del algoritmo es adecuado, produciendo el terreno más interesante de la prueba. Valores superiores a los aplicados en la prueba, no tienen efecto y por lo tanto el resultado es similar a fBm.

El multifractal híbrido extiende al multifractal heterogéneo, produciendo una disminución monótona del peso cada iteración y en consecuencia, disminuyendo la rugosidad en todas las elevaciones y no únicamente las cercanas al nivel del mar. La figura 7.9 permite apreciar como o afecta el terreno generado. Para estas pruebas, se definió $H = 0{,}25$ como recomienda Musgrave en [11] considerando que la disminución del peso es pronunciada.

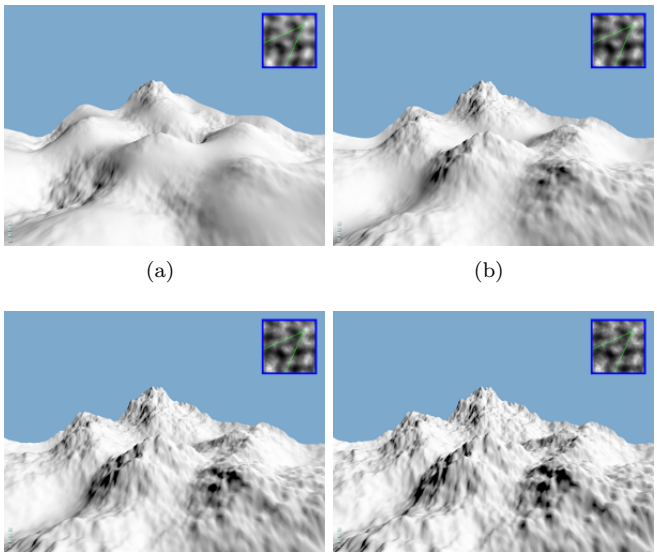
Cuando o es 0,0, el terreno es suave y carece de detalle; al incrementar su valor, aumenta la rugosidad global del terreno, siendo progresivamente mayor al encontrarse en una elevación mayor. Cuando o es 0,5, se garantiza el comportamiento adecuado del algoritmo y se genera el terreno más interesante de la prueba. Es importante destacar que valores superiores a los aplicados en la prueba, producen terrenos ruidosos. Además, valores del ruido de Perlin desplazado mayores a 1, no garantizan la disminución monótona del peso.

Finalmente, la turbulencia de Quilez define la rugosidad del terreno en función de la magnitud del gradiente local de la función base; cuando la magnitud es mayor, es decir, la pendiente es pronunciada, la rugosidad será menor. La figura 7.10 permite observar como H afecta la rugosidad del terreno generado; similar a fBm, al aumentar su valor, incrementa la rugosidad.

Los resultados muestran terrenos menos uniformes y de apariencia más natural; que presentan una variación importante de la rugosidad, donde se pueden apreciar tanto regiones suaves como regiones con detalles. Cuando H es 1,0, se genera el terreno más interesante de la prueba.

7.4 Transformaciones del Generador

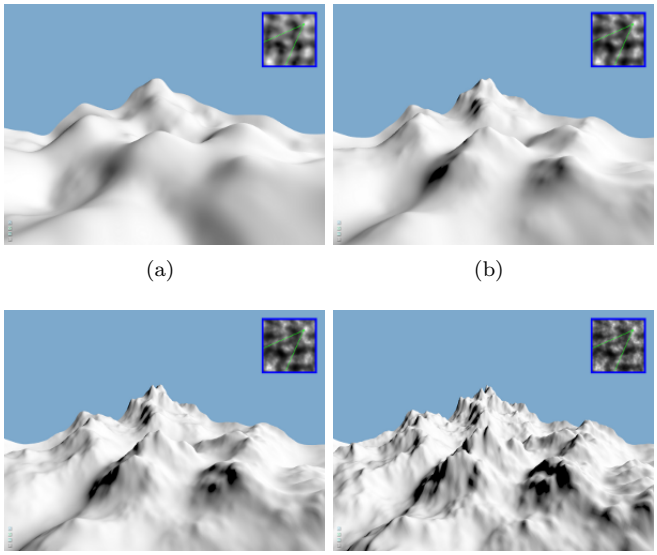
Para analizar la influencia de las transformaciones sobre el Generador, se llevarán a cabo pruebas similares a las realizadas a los algoritmos del Generador presentadas en la sección 7.3. En el caso de la renderización del terreno, se utilizará la misma configuración. En cuanto a la configuración del Generador, se utilizará: mapas de altura de 256×256 píxeles, el ruido de Perlin como función base con semilla 17 y fBm como algoritmo de generación con los parámetros por defecto (tamaño de las características del 90 %, $n = 8$, $H = 1{,}0$ y $\lambda = 2{,}0$).



(a) (b)

(c) (d)

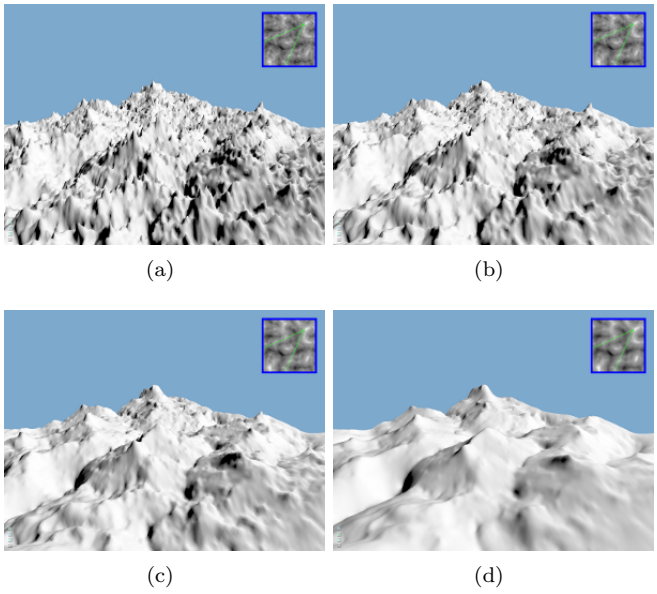
Figura 7.8: Multifractal heterogéneo para diferentes desplazamientos: (a) $o = 0{,}0$, (b) $o = 0{,}3$, (c) $o = 0{,}5$ y (d) $o = 0{,}7$.



(a) (b)

(c) (d)

Figura 7.9: Multifractal híbrido para diferentes desplazamientos: (a) $o = 0{,}0$, (b) $o = 0{,}3$, (c) $o = 0{,}5$ y (d) $o = 0{,}7$.



(a) (b)

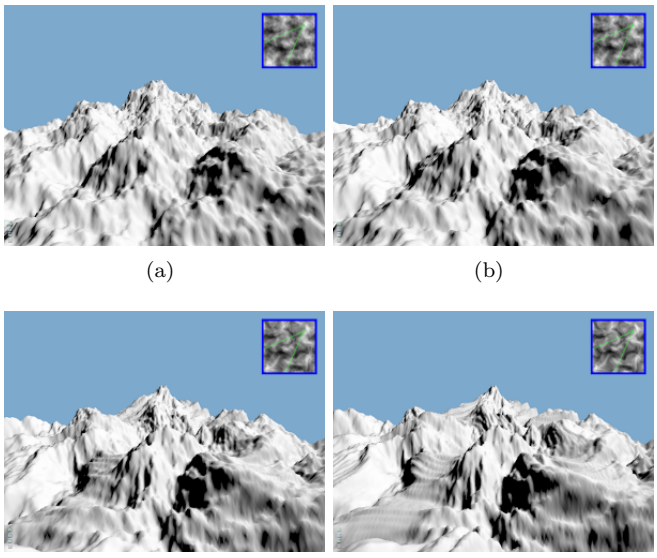
(c) (d)

Figura 7.10: Turbulencia de Quilez para distintos valores del exponente de Hurst: (a) $H = 0{,}5$, (b) $H = 0{,}7$, (c) $H = 1{,}0$ y (d) $H = 1{,}5$.

La transformación asociada al pre-procesamiento incluida en el Generador, es la distorsión del dominio; que incluye el parámetro α para controlar la intensidad de la distorsión. La figura 7.11 muestra como α afecta el terreno generado; al aumentar α, aumenta la distorsión presente en el terreno, produciendo formas más turbulentas donde se comprimen, rotan y estiran localmente las características. Cuando α es 50 %, se puede notar claramente el efecto de la distorsión.

Finalmente, el Generador incorpora tres transformaciones asociadas al post-procesamiento: el efecto glaciar, el efecto cañón y el efecto meseta; que incluyen el parámetro s para controlar la intensidad del efecto. Las figuras 7.12, 7.13 y 7.14 permiten apreciar como cada transformación afecta el terreno generado.

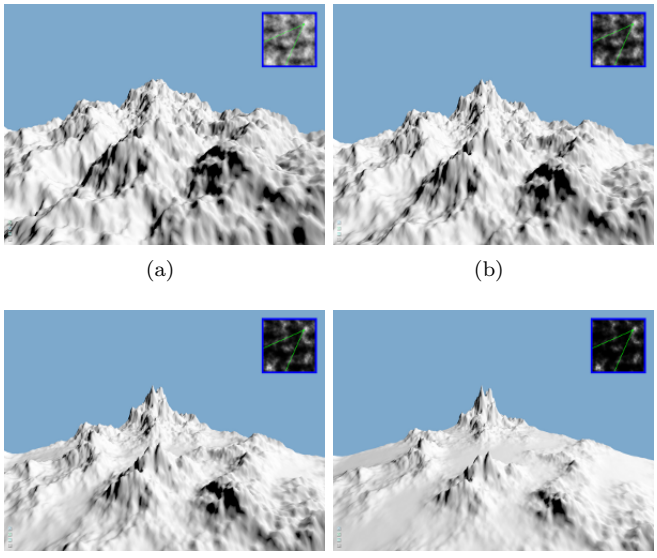
Los resultados ratifican el funcionamiento adecuado de las transformaciones, donde al aumentar s se hace progresivamente más notable el efecto de la transformación determinada sobre la forma del terreno generado. Cuando s es 75 %, se puede apreciar claramente las características de cada transformación.



(a) (b)

(c) (d)

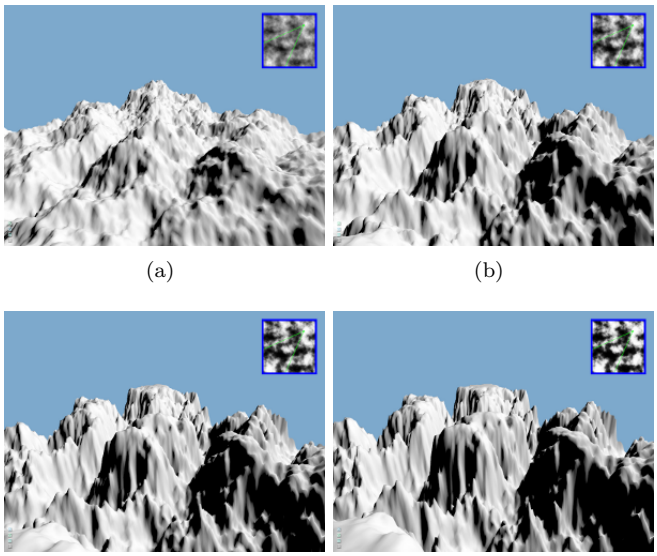
Figura 7.11: Distorsión del dominio para diferentes escalas de distorsión: (a) $\alpha = 0\,\%$ (sin distorsión), (b) $\alpha = 25\,\%$, (c) $\alpha = 50\,\%$ y (d) $\alpha = 75\,\%$.



(a) (b)

(c) (d)

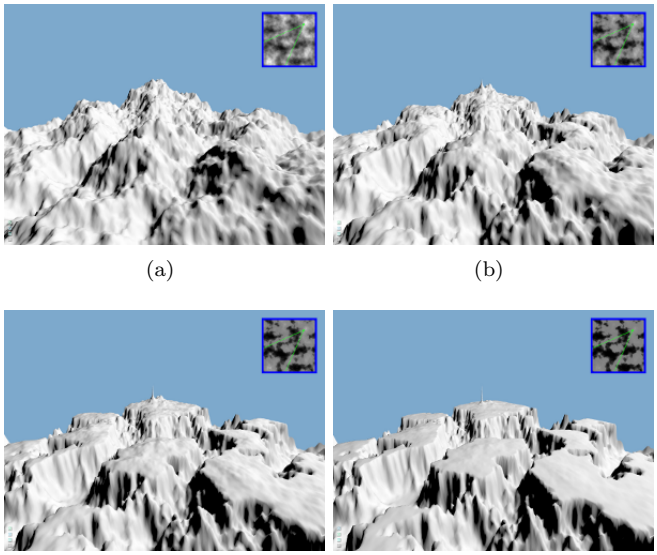
Figura 7.12: Efecto glaciar para distintas intensidades del efecto: (a) $s = 0\,\%$ (sin efecto), (b) $s = 50\,\%$, (c) $s = 75\,\%$ y (d) $s = 90\,\%$.



(a) (b)

(c) (d)

Figura 7.13: Efecto cañón para distintas intensidades del efecto: (a) $s = 0\,\%$ (sin efecto), (b) $s = 50\,\%$, (c) $s = 75\,\%$ y (d) $s = 90\,\%$.



(a) (b)

(c) (d)

Figura 7.14: Efecto meseta para distintas intensidades del efecto: (a) $s = 0\,\%$ (sin efecto), (b) $s = 50\,\%$, (c) $s = 75\,\%$ y (d) $s = 90\,\%$.

7.5 Rendimiento del Visualizador

Para evaluar el rendimiento del Visualizador, se midieron los FPS utilizando la función *glutGet* con parámetro *GLUT_ELAPSED_TIME* basado en el ejemplo publicado en [27]. Se desactivó la opción de renderización con sincronizado vertical *(vertical sync)*, que ajusta los FPS de la aplicación a la frecuencia de actualización del monitor (60 FPS para la mayoría de los monitores). Las pruebas consistieron en medir las primeras 100 muestras de los FPS e imprimir los resultados en un archivo. Estos archivos, posteriormente fueron cargados y procesados en Excel.

El rendimiento del Visualizador depende de tres factores: las dimensiones del mapa de altura, el tamaño de la pantalla y las opciones activadas asociadas al Visualizador. Considerando que el primero es el factor primordial, se midieron los FPS para distintas dimensiones del mapa de altura, utilizando la pantalla completa (1680×988 píxeles y 1680×1050 píxeles para el ambiente 1 y el ambiente 2 respectivamente) y activando todas las opciones excepto la del mallado.

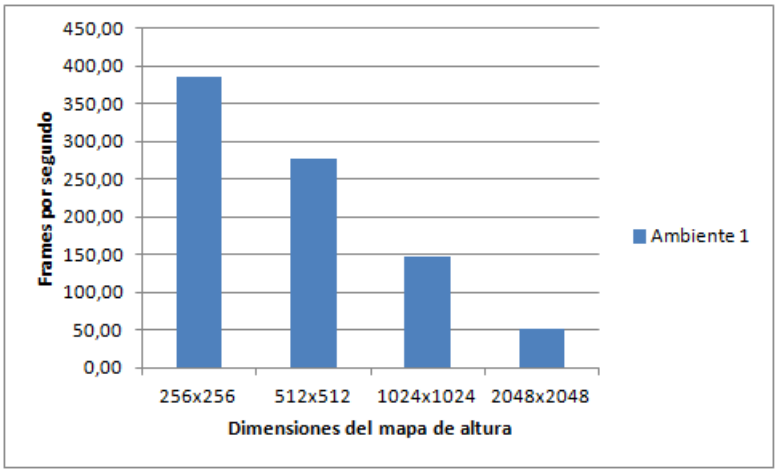
En todas las mediciones se utilizó un conjunto de dos materiales, ya que el algoritmo de texturizado implementado utilizaría a lo sumo dos materiales independientemente del número de materiales disponibles. Este caso ocurriría cuando se aplique una mezcla lineal en la frontera entre materiales.

En la tabla 7.6 se puede apreciar los FPS promedio obtenidos en estas pruebas para el ambiente 1 y el ambiente 2respectivamente. Además, la figura 7.15 muestra dos gráficos (uno para cada ambiente) de los resultados obtenidos; presentándose en ambos ambientes una tendencia similar pero a escalas diferentes, donde al aumentar las dimensiones del mapa de altura se produce una disminución gradual de los FPS. La disminución es gradual, considerando que cada mapa de altura altura posee 4 veces más vértices que el mapa de altura previo, pero los FPS se reducen menos de 4 veces con respecto al mapa de altura previo.

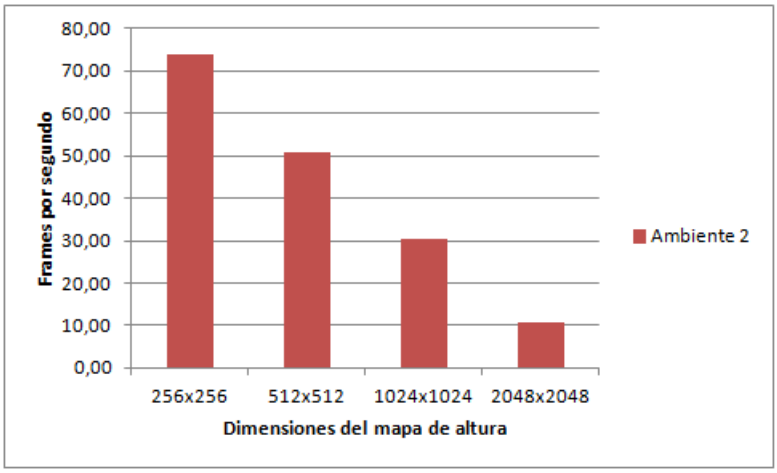
Para todas las dimensiones, el ambiente 1 presenta un rendimiento considerablemente superior al ambiente 2, donde para el mapa de altura más grande (2048×2048) el ambiente 1 produce 394 % más FPS que el ambiente2.

	Dimensiones del mapa de altura			
Ambiente	256×256	512×512	1024×1024	2048×2048
Ambiente 1	385,46	277,42	148,47	52,43
Ambiente 2	73,90	50,88	30,44	10,61

Tabla 7.6: FPS promedio del Visualizador para diferentes dimensiones del mapa de altura (ambiente 1 y ambiente 2).



(a)



(b)

Figura 7.15: FPS para diferentes dimensiones del mapa de altura: (a) Ambiente 1 y (b) Ambiente 2.

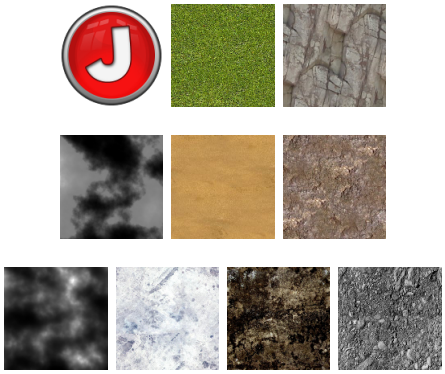


Figura 7.16: Datos de los escenarios de prueba del Visualizador.

Para medir el impacto del tamaño de la pantalla y las opciones activadas, se midieron los FPS para el mapa de altura más grande (2048×2048) con los mismos parámetros mencionados anteriormente, difiriendo únicamente en el factor determinado. Con una pantalla de 800×600 píxeles, se presenta un incremento de 4 % y 13 % para el ambiente 1 y el ambiente 2 respectivamente. Sin ninguna opción activada, se presenta un incremento de 4 % y 16 % para el ambiente 1 y el ambiente 2 respectivamente.

7.6 Visualizador y sus características

Para verificar el funcionamiento correcto del Visualizador y sus características implementadas, se plantearon tres escenarios de prueba representando terrenos diversos tanto en formas como en materiales. En la figura 7.16 se puede apreciar los datos utilizados en cada escenario, que incluyen el mapa de altura y las texturas. La última imagen del tercer escenario corresponde al mapa de detalle.

En todos los escenarios, se activaron las opciones asociadas a la proyección triplanar, la iluminación y el mapa de navegación. Además, se definió el color del Sol a 253, 220 y 48 que corresponde al valor del canal rojo, verde y azul respectivamente. La figura 7.17 muestra distintas capturas de pantalla de los diferentes escenarios.

El mapa de altura del primer escenario, corresponde a un icono de una J de 256×256 píxeles encontrado con la busqueda de imágenes de Google. En el caso de las texturas, corresponden a texturas utilizadas por el programa Scape [9].

En el primer escenario, se aplicó un texturizado basado en altura con $[0{,}0; 0{,}26]$ y $(0{,}26; 1{,}0]$ como el rango de valores de los materiales grama y piedra respec-

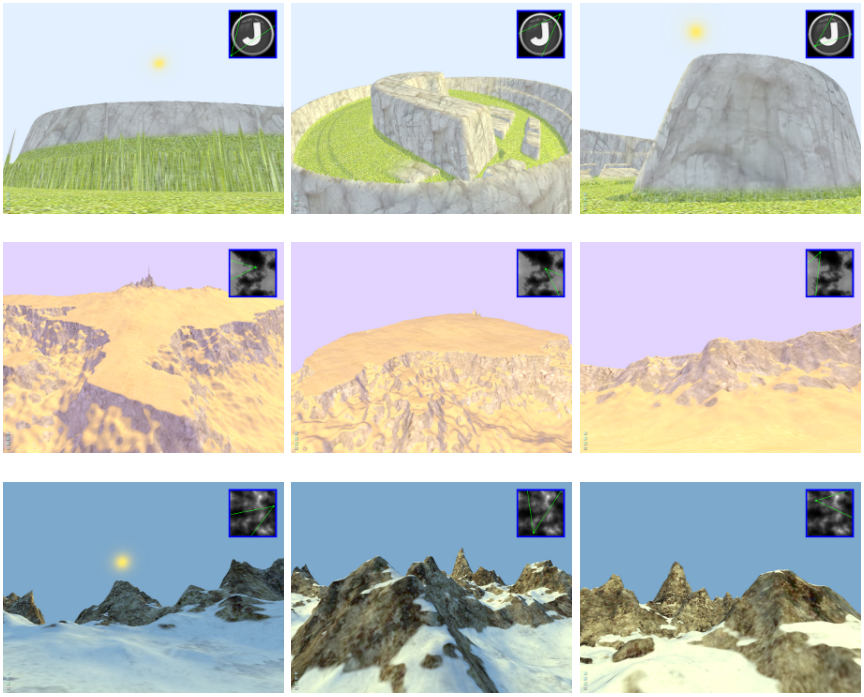
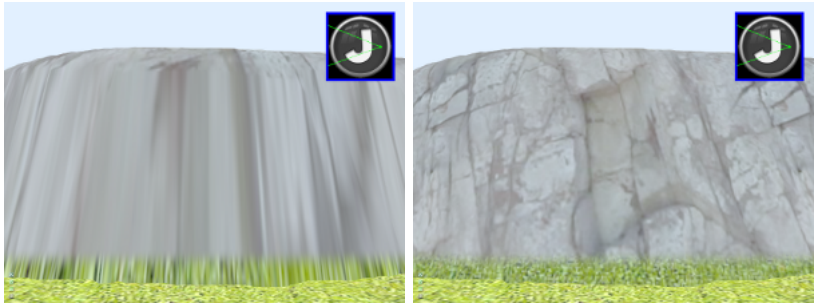


Figura 7.17: Distintas capturas de pantalla de cada escenario de prueba del Visualizador.



(a) (b)

Figura 7.18: Opción de proyección triplanar del Visualizador en el primer escenario: (a) Proyección planar y (b) Proyección triplanar.

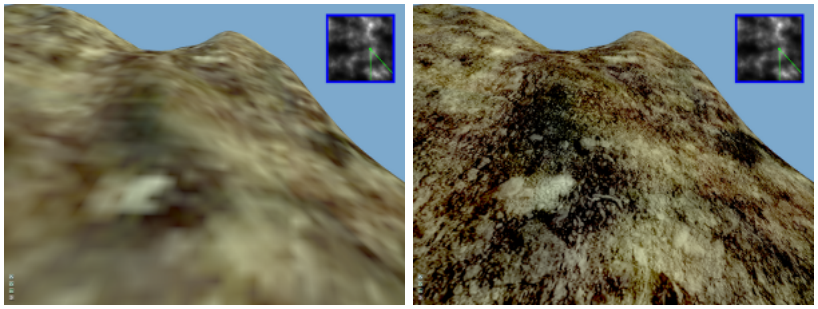
tivamente, además se definió el número de repeticiones de la textura a 5,0. La escala del terreno, se definió con una altura máxima de 50 y un tamaño de celda de 1. El Sol, se ubicó a una altura de 65° en dirección 30°. El color del cielo, se definió a 201, 226 y 255 que corresponde al valor del canal rojo, verde y azul respectivamente.

Este escenario fue especialmente útil para verificar el funcionamiento correcto del mapa de navegación y la proyección triplanar. En la segunda captura de pantalla, se puede apreciar como el mapa de navegación además de facilitar la ubicación del usuario con respecto al mapa de altura, permite verificar la correspondencia correcta entre el terreno renderizado y el mapa de altura. La figura 7.18 permite ratificar que la proyección triplanar soluciona (al menos visualmente) el problema del estiramiento de texturas.

El mapa de altura de 1024×1024 píxeles utilizado en el segundo escenario, fue generado por la aplicación utilizando el algoritmo básico fBm con un tamaño de las características del 100 % y el resto de los parámetros con los valores por defecto, además se aplicó el efecto meseta con 75 % de intesidad. En cuanto a las texturas, la primera se descargó de la biblioteca de texturas publicada en [4] mientras la segunda es una de las texturas utilizadas por Scape [9].

En el segundo escenario, se aplicó un texturizado basado en pendiente con $[0{,}0; 0{,}3]$ y $(0{,}3; 1{,}0]$ como el rango de valores de los materiales arena y tierra oscura respectivamente, además se definió 5,0 como el número de repeticiones de la textura. La escala del terreno, se definió con una altura máxima de 200 y un tamaño de celda de 1. El Sol, se ubicó a una altura de 75° en dirección 0°. El color del cielo se definió a 201, 176 y 255 que corresponde al valor del canal rojo, verde y azul respectivamente.

El mapa de altura de 1024×1024 píxeles utilizado en el tercer escenario, fue



(a) (b)

Figura 7.19: Opción de mapas de detalle del Visualizador en el tercer escenario: (a) Sin mapa de detalle y (b) Con mapa de detalle.

generado por la aplicación mediante el multifractal híbrido con un tamaño de las características del 95 %, $H = 0{,}25$, $o = 0{,}7$ y el resto de los parámetros con los valores por defecto, además se aplicó el efecto glaciar con 25 % de intensidad. En cuanto a las texturas, corresponden a texturas aplicadas por el programa Charack [12]. El mapa de detalle fue descargado del foro Arrangement [15].

En el tercer escenario, se aplicó un texturizado basado en pendiente con $[0{,}0; 0{,}2]$ y $(0{,}2; 1{,}0]$ como el rango de valores de los materiales nieve y roca respectivamente, además se definió el número de repeticiones de la textura a 3,0. La escala del terreno, se definió con una altura máxima de 300 y un tamaño de celda de 1. El Sol, se ubicó a una altura de 50° en dirección 210°, y el color del cielo, se definió a 61, 112 y 163 que corresponde al valor del canal rojo, verde y azul respectivamente.

La figura 7.19 demuestra como el mapa de detalle efectivamente agrega detalles a las regiones cercanas a la cámara, donde se definió 50,0 como el número de repeticiones del mapa detalle y 100,0 como la distancia del efecto.

Capítulo 8

Conclusiones y trabajos futuros

8.1 Conclusiones

En el presente Trabajo Especial de Grado, se desarrolló un Generador de diversos tipos de terrenos basado en la extensión de síntesis de ruido propuesta por De Carpentier y Bidarra [5]. Además, se implementó un Visualizador tridimensional de mapas de altura adecuado para estudiar y explorar los terrenos generados.

De Carpentier y Bidarra extienden la síntesis de ruido, al incluir transformaciones al procedimiento con la finalidad de generar terrenos con formas más complejas. No obstante, los algoritmos implementados por los autores se basan en transformaciones específicas. En cambio, el Generador planteado en el presente trabajo, permite utilizar para todo algoritmo implementado, cualquiera de las transformaciones disponibles. De esta manera, fue posible ampliar de forma significativa las capacidades y posibilidades del Generador, permitiendo crear una gama de terrenos, que van desde montañas heterogéneas a través de glaciares, hasta mesetas.

A través de las pruebas realizadas, fue posible estudiar detallamente los algoritmos, las funciones base y las transformaciones incluidas en el Generador, analizando e ilustrando como cada parámetro influye sobre la forma del terreno generado. Tomando en cuenta que precisamente uno de los inconvenientes más importantes de la generación fractal, es la comprensión de los parámetros involucrados y como influyen sobre el resultado, se puede afirmar que las pruebas realizadas en combinación con el cálculo de los gradientes locales del ruido de valor y el ruido de Perlin, representan un aporte importante al campo.

Los mapas de altura fueron generados en un tiempo de ejecución aceptable de acuerdo a las expectativas del trabajo. Sin embargo, considerando que la altura de cada vértice fue calculada de manera independiente, se podría paralelizar el cálculo, lo que reduciría drásticamente el tiempo de generación.

Con las pruebas realizadas sobre el Visualizador, se puede concluir que cumple el propósito para el cual fue desarrollado. El Visualizador permite navegar terrenos representados por mapas de altura de forma interactiva, incluyendo un mapa de navegación primordial para el estudio de los mismos. Además, proporciona texturizado basado en altura o pendiente de múltiples materiales, que es necesario para modelar diversos tipos de terrenos. Por último, soporta características que aportan al realismo de la visualización: iluminación dinámica, visualización del cielo, mapas de detalle, entre otras.

Debido a que las pruebas fueron realizadas en ambientes diferentes, se pudo apreciar que un ambiente más moderno y potente, tiene la capacidad de acelerar de forma sustancial tanto el Generador como el Visualizador, inclusive permitiendo la navegación interactiva en los escenarios planteados más exigentes.

No obstante, la propuesta implementada presenta algunas limitaciones. En principio, no es posible almacenar mapas de altura de 16 bits de precisión, debido a que no es una característica soportada por FreeImage [17]. En cuanto al Visualizador, no posee soporte para nivel de detalle, por lo que produce *aliasing* espacial, especialmente notable cuando se aumenta el número de repeticiones de la textura manifestándose el fenómeno *strobing* [24]. Además, la visualización tridimensional de mapas de altura de grandes dimensiones, es computacionalmente prohibitiva y por ende inviable. Finalmente, la generación no es interactiva.

8.2 Trabajos futuros y recomendaciones

Una propuesta que resultaría interesante para trabajos futuros, sería extender el Generador mediante la inclusión de nuevos algoritmos, funciones base y/o transformaciones. En el caso de los algoritmos, se propone incluir la turbulencia Suiza *(Swiss turbulence)*, un algoritmo basado en el ruido erosivo implementado por De Carpenter en su editor de terrenos Scape [9]. Para las funciones base, se recomienda incluir el *simplex noise* propuesto por Perlin [41] y el ruido de Worley [58]. Por último, en las transformaciones se propone incorporar el ruido de cresta y el ruido ondulante, como transformaciones aplicadas a la funcion base.

Es importante resaltar, que las funciones base además de la implementación, presentan el reto adicional de calcular el gradiente local para los algoritmos que lo requieran. De igual forma, es necesario el cálculo de las derivadas de las transformaciones aplicadas a la función base.

Para superar las limitaciones relacionadas a la generación interactiva y el Visualizador, se sugiere implementar el enfoque propuesto por Schneider et al. [52], que permite la generación y la renderización en tiempo real con soporte a nivel de detalle adaptitivo. Además, el enfoque puede ser extendido a dominios infinitos y esféricos.

Apéndice A

Cálculo analítico del gradiente local de las funciones base

A.1 Ruido de valor

Para calcular el gradiente local de forma analítica, se establece el ruido de valor como una función matemática $N(x, y)$, que a su vez, calcula una interpolación bilineal basada en el punto de entrada $p = (x, y)$ y la cuadrícula de valores pseudoaleatorios:

$$\begin{aligned} &N(x,y) = lerp\Big(w\Big(frac(y)\Big), lerp\Big(w\Big(frac(x)\Big), a, b\Big), lerp\Big(w\Big(frac(x)\Big), c, d\Big)\Big) \\ &lerp(t,a,b) = a + t(b-a) \\ &w(t) = 6t^5 - 15t^4 + 10t^3 \\ &frac(t) = t - \lfloor t \rfloor \end{aligned} \tag{A.1}$$

donde (a, b, c y d) son los valores pseudoaleatorios asociados a los vértices más cercanos a p en la cuadrícula, que son constantes en la definición de la función. La permutación aplicada para obtener los valores de los vértices, se asume implícita.

Luego, la ecuación A.1 se expande mediante aplicar las interpolaciones lineales ($lerp$) y se organiza mediante factorización:

$$\begin{aligned} &N(x,y) = a + (b-a)w\Big(fract(x)\Big) + (c-a)w\Big(fract(y)\Big) + \\ &(a-b-c+d)w\Big(fract(x)\Big)w\Big(fract(y)\Big) \end{aligned} \tag{A.2}$$

Aplicando la regla de la cadena, se obtiene:

$$\begin{array}{l}\frac{\partial w\Big(fract(t)\Big)}{\partial t} = \frac{\partial w\Big(fract(t)\Big)}{\partial t}\frac{\partial fract(t)}{\partial t} \\ \frac{\partial fract(t)}{\partial t} = \frac{\partial t}{\partial t} - \frac{\partial \lfloor t \rfloor}{\partial t} = 1 - 0 = 1\end{array} \tag{A.3}$$

La derivada de la función piso es 0 en los puntos no enteros, pero es indefinida en los puntos enteros producto de la discontinuidad. Sin embargo, por la manera en la que es utilizada la función piso en el ruido de valor, la discontinuidad no ocurre. Por lo tanto, tomando en cuenta los resultados de A.3, es posible reescribir la ecuación sin afectar el cálculo de las derivadas parciales:

$$\begin{array}{l}N(u,v) = a + (b-a)w(u) + (c-a)w(v) + (a-b-c+d)w(u)w(v) \\ u = fract(x) \\ v = fract(y) \\ \frac{\partial N(x,y)}{\partial x} = \frac{\partial N(u,v)}{\partial u} \\ \frac{\partial N(x,y)}{\partial y} = \frac{\partial N(u,v)}{\partial v}\end{array} \tag{A.4}$$

Las derivadas parciales de la ecuación A.4 son:

$$\begin{array}{l}\frac{\partial N(u,v)}{\partial u} = (b-a)\frac{\partial w(u)}{\partial u} + (a-b-c+d)w(v)\frac{\partial w(u)}{\partial u} \\ \frac{\partial N(u,v)}{\partial v} = (c-a)\frac{\partial w(v)}{\partial v} + (a-b-c+d)w(u)\frac{\partial w(v)}{\partial v} \\ \frac{\partial w(t)}{\partial t} = 30t^4 - 60t^3 + 30t^2\end{array} \tag{A.5}$$

Finalmente, se factoriza el resultado y el gradiente local del ruido de valor es:

$$\begin{array}{l}\frac{\partial N(u,v)}{\partial u} = \Big((b-a) + (a-b-c+d)w(v)\Big)\frac{\partial w(u)}{\partial u} \\ \frac{\partial N(u,v)}{\partial v} = \Big((c-a) + (a-b-c+d)w(u)\Big)\frac{\partial w(v)}{\partial v} \\ \frac{\partial w(t)}{\partial t} = 30t^4 - 60t^3 + 30t^2\end{array} \tag{A.6}$$

A.2 Ruido de Perlin

Para calcular el gradiente local del ruido de Perlin analíticamente, se utilizará un procedimiento similar al caso del ruido de valor. Primero, se define el ruido de Perlin como una función $N(x,y)$:

$$\begin{array}{l}N(u,v) = lerp\Big(w(v), lerp\Big(w(u), g_a \cdot (u,v), g_b \cdot (u-1,v)\Big), \\ lerp\Big(w(u), g_c \cdot (u,v-1), g_d \cdot (u-1,v-1)\Big)\Big) \\ lerp(t,a,b) = a + t(b-a) \\ (x_1,y_1)\cdot(x_2,y_2) = x_1x_2 + y_1y_2 \\ w(t) = 6t^5 - 15t^4 + 10t^3 \\ u = x - \lfloor x \rfloor \\ v = y - \lfloor y \rfloor\end{array} \tag{A.7}$$

donde (g_a, g_b, g_c y g_d) son los gradientes unitarios pseudoaleatorios asociados a los vértices más cercanos a p en la cuadrícula, que son constantes en la definición de la función. Es posible definir la función con respecto a u y v, ya que no afecta el cálculo del gradiente, por el mismo motivo que se indicó en el cálculo del gradiente del ruido de valor.

Luego, la ecuación A.7 se expande mediante aplicar las interpolaciones lineales ($lerp$) y se organiza mediante factorización:

$$\begin{aligned}
&N(u,v) = a + (b-a)w(u) + (c-a)w(v) + (a-b-c+d)w(u)w(v)\\
&a = g_a \cdot (u, v)\\
&b = g_b \cdot (u-1, v)\\
&c = g_c \cdot (u, v-1)\\
&d = g_d \cdot (u-1, v-1)
\end{aligned} \tag{A.8}$$

Para calcular las derivadas parciales de la ecuación A.8, es necesario calcular las derivadas parciales de los productos punto involucrados:

$$\begin{aligned}
\frac{\partial\big((x,y)\cdot(u+c_1,v+c_2)\big)}{\partial u} &= \frac{\partial\big(x(u+c_1)+y(v+c_2)\big)}{\partial u} = x\\
\frac{\partial\big((x,y)\cdot(u+c_1,v+c_2)\big)}{\partial v} &= \frac{\partial\big(x(u+c_1)+y(v+c_2)\big)}{\partial v} = y
\end{aligned} \tag{A.9}$$

donde (x, y) son las coordenadas del gradiente involucrado, c_1 y c_2 son constantes. Por lo tanto, las derivadas parciales de la ecuación A.8 son:

$$\begin{aligned}
&\tfrac{\partial N(u,v)}{\partial u} = g_{ax} + \Big[(g_{bx}-g_{ax})w(u) + (b-a)\tfrac{\partial w(u)}{\partial u}\Big] + \Big[w(v)(g_{cx}-g_{ax})\Big] +\\
&w(v)\Big[(g_{ax}-g_{bx}-g_{cx}+g_{dx})w(u) + (a-b-c+d)\tfrac{\partial w(u)}{\partial u}\Big]\\
&\tfrac{\partial N(u,v)}{\partial v} = g_{ay} + \Big[(g_{by}-g_{ay})w(u)\Big] + \Big[(g_{cy}-g_{ay})w(v) + (c-a)\tfrac{\partial w(v)}{\partial v}\Big] +\\
&w(u)\Big[(g_{ay}-g_{by}-g_{cy}+g_{dy})w(v) + (a-b-c+d)\tfrac{\partial w(v)}{\partial v}\Big]\\
&\tfrac{\partial w(t)}{\partial t} = 30t^4 - 60t^3 + 30t^2
\end{aligned} \tag{A.10}$$

Finalmente, se factoriza el resultado y el gradiente local del ruido de Perlin es:

$$\begin{aligned}
&\tfrac{\partial N(u,v)}{\partial u} = g_{ax} + (g_{bx}-g_{ax})w(u) + (g_{cx}-g_{ax})w(v) +\\
&(g_{ax}-g_{bx}-g_{cx}+g_{dx})w(u)w(v) + \Big[(b-a) + (a-b-c+d)w(v)\Big]\tfrac{\partial w(u)}{\partial u}\\
&\tfrac{\partial N(u,v)}{\partial v} = g_{ay} + (g_{by}-g_{ay})w(u) + (g_{cy}-g_{ay})w(v) +\\
&(g_{ay}-g_{by}-g_{cy}+g_{dy})w(u)w(v) + \Big[(c-a) + (a-b-c+d)w(u)\Big]\tfrac{\partial w(v)}{\partial v}\\
&\tfrac{\partial w(t)}{\partial t} = 30t^4 - 60t^3 + 30t^2
\end{aligned} \tag{A.11}$$

Bibliografía

[1] Song Ho Ahn. High resolution timer. `http://www.songho.ca/misc/timer/timer.html`, 2005. Consultado en Noviembre del 2014.

[2] AntTweakBar. AntTweakBar. `http://anttweakbar.sourceforge.net/doc/`. Consultado en Octubre del 2014.

[3] Charles Bloom. Texture splatting. `http://www.cbloom.com/3d/techdocs/splatting.txt`, 2000. Consultado en Junio del 2014.

[4] Paul Bourke. Texture library. `http://paulbourke.net/texture_colour/`. Consultado en Diciembre del 2014.

[5] Giliam J.P. de Carpentier and Rafael Bidarra. Interactive gpu-based procedural heightfield brushes. In *Proceedings of Fourth International Conference on the Foundations of Digital Games*, pages 55–62, Port Canaveral, FL, apr 2009.

[6] G-Truc Creation. OpenGL Mathematics. `http://glm.g-truc.net/0.9.5/index.html`. Consultado en Octubre del 2014.

[7] W. H. De Boer. Fast Terrain Rendering Using Geometrical Mipmapping, 2000.

[8] Giliam de Carpentier. Effective gpu-based synthesis and editing of realistic heightfields. Master's thesis, Delft University of Technology, 2008.

[9] Giliam de Carpentier. Scape: 5. overview and downloads. `http://www.decarpentier.nl/scape-brush-pipeline`, 2012. Consultado en Junio del 2014.

[10] Dia. Dia. `https://wiki.gnome.org/Apps/Dia/`. Consultado en Octubre del 2014.

[11] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[12] Bevilacqua et al. Charack: Pseudo-infinite 3d virtual world generation. https://code.google.com/p/charack/, 2009. Consultado en Diciembre del 2014.

[13] Jonathan Ferraris and Christos Gatzidis. A rule-based approach to 3d terrain generation via texture splatting. In *Proceedings of the International Conference on Advances in Computer Enterntainment Technology*, ACE '09, pages 407–408, New York, NY, USA, 2009. ACM.

[14] James D. Foley and Andries Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[15] ARRANGEMENT forums. detail textures rock. http://am.half-lifecreations.com/forums/index.php?topic=348.0, 2009. Consultado en Diciembre del 2014.

[16] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, June 1982.

[17] FreeImage. FreeImage: The productivity booster. http://freeimage.sourceforge.net/. Consultado en Octubre del 2014.

[18] Ryan Geiss. Generating complex procedural terrains using the gpu. In Hubert Nguyen, editor, *GPU Gems 3*, pages 7–37. Addison-Wesley, 2008.

[19] Nate Glasser. Texture splatting in direct3d. http://www.gamedev.net/page/resources/_/technical/game-programming/texture-splatting-in-direct3d-r2238, 2005. Consultado en Junio del 2014.

[20] Simon Green. Implementing improved perlin noise. In Matt Pharr, editor, *GPU Gems 2*, pages 409–416. Addison-Wesley, 2005.

[21] Larry Gritz and Eugene dEon. The importance of being linear. In Hubert Nguyen, editor, *GPU Gems 3*, pages 529–542. Addison-Wesley, 2008.

[22] Qlinks Media Group. Geo Community Website. http://www.geocomm.com/. Consultado en Junio del 2014.

[23] S. Gustavson. Simplex noise demystified. Technical report, Linköping University, 2005.

[24] S. James. *3D Graphics with XNA Game Studio 4.0*. Packt, Birmingham, B27 6PA, UK., 2010.

[25] R. Krten. Generating realistic terrain. http://www.drdobbs.com/parallel/generating-realistic-terrain/184409269, 1994. Consultado en Junio del 2014.

[26] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, D.S. Ebert, J.P. Lewis, Ken Perlin, and Matthias Zwicker. State of the art in procedural noise functions. In Helwig Hauser and Erik Reinhard, editors, *EG 2010 - State of the Art Reports*. Eurographics Association, May 2010.

[27] Lighthouse3D. Frames per second. `http://www.lighthouse3d.com/tutorials/glut-tutorial/frames-per-second/`. Consultado en Junio del 2014.

[28] Lighthouse3D. View frustum culling tutorial. `http://www.lighthouse3d.com/tutorials/view-frustum-culling/`. Consultado en Octubre del 2014.

[29] B.B. Mandelbrot. *The Fractal Geometry of Nature*. Henry Holt and Company, 1983.

[30] Ian McEwan, David Sheets, Stefan Gustavson, and Mark Richardson. Efficient computational noise in glsl. *CoRR*, abs/1204.1461, 2012.

[31] Morgan McGuire and Kyle Whitson. Indirection mapping for quasi-conformal relief mapping. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '08)*, February 2008.

[32] Rodrigo Medina. Tutorial: Creación de una skydome usando opengl. `http://sodvi.com/Skydome%20Tutorial.pdf`. Consultado en Octubre del 2014.

[33] Gavin S P Miller. The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph.*, 20(4):39–48, August 1986.

[34] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, 23(3):41–50, July 1989.

[35] Kris Nicholson. Gpu based algorithms for terrain texturing. Technical report, University of Canterbury,, 2008.

[36] Jacob Olsen. Realtime procedural terrain generation. Technical report, University of Southern Denmark, 2004.

[37] Heinz-Otto Peitgen and Dietmar Saupe, editors. *The Science of Fractal Images*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[38] K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3):253–262, July 1989.

[39] Ken Perlin. Improved noise reference implementation. `http://mrl.nyu.edu/~perlin/noise/`. Consultado en Octubre del 2014.

[40] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[41] Ken Perlin. Noise hardware. In *Real-Time Shading*. ACM SIGGRAPH 2001 Course Notes, ACM Press, 2001.

[42] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002.

[43] Ken Perlin. Implementing improved perlin noise. In Randima Fernando, editor, *GPU Gems*, pages 73–85. Addison-Wesley, 2004.

[44] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[45] Trent Polack. *Focus on 3D Terrain Programming*. Premier Press, 2002.

[46] Virtual Terrain Project. Virtual Terrain Project. `http://vterrain.org/`. Consultado en Junio del 2014.

[47] Iñigo Quilez. advanced perlin noise. `http://www.iquilezles.org/www/articles/morenoise/morenoise.htm`, 2008. Consultado en Octubre del 2014.

[48] Iñigo Quílez. Elevated. `https://www.shadertoy.com/view/MdX3Rr`, 2010. Consultado en Octubre del 2014.

[49] R. L Saunders. Terrainosaurus: Realistic terrain synthesis using genetic algorithms. Master's thesis, Texas A&M University, 2006.

[50] Dietmar Saupe. Point evaluation of multi-variable random fractals. In Hartmut Jurgens and Dietmar Saupe, editors, *Visualisierung in Mathematik und Naturwissenschaften*, pages 114–126. Springer Berlin Heidelberg, 1989.

[51] S Schmitt. World Machine software. `http://www.world-machine.com/`. Consultado en Junio del 2014.

[52] Jens Schneider, Tobias Boldte, and Ruediger Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, Modeling and Visualization 2006*, 2006.

[53] Jason Shankel. Fractal terrain generation - fault formation. In Mark DeLoura, editor, *Game Programming Gems*, pages 499–502. Charles River Media, 2000.

[54] Jason Shankel. Fractal terrain generation - midpoint displacement. In Mark DeLoura, editor, *Game Programming Gems*, pages 503–507. Charles River Media, 2000.

[55] Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, pages n/a–n/a, 2014.

[56] Planetside software. Terragen software. http://planetside.co.uk/. Consultado en Junio del 2014.

[57] United States Geological Survey. USGS Website. http://www.usgs.gov/. Consultado en Junio del 2014.

[58] Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 291–294, New York, NY, USA, 1996. ACM.