UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Visualización 3D de terrenos multirresolución
Autores: Lucas Federico Borrelli
Director: María José Abásolo
Codirector: ---
Asesor profesional: ---
Carrera: Licenciatura en Sistemas

Resumen

Los algoritmos diseñados para representar y visualizar terrenos en mundos virtuales 3D son muy importantes en aplicaciones como sistemas de información geográfica, simuladores de vuelo o de entrenamiento, videojuegos, etc. El principal desafío radica en alcanzar la performance necesaria para una visualización en tiempo real de grandes extensiones de terreno. Las técnicas desarrolladas aplican el concepto de multirresolución para reducir la complejidad del modelo de terrenos. El propósito de este trabajo ha sido desarrollar un algoritmo multirresolución eficiente que utilice los últimos avances del hardware gráfico, la GPU (Graphics Processing Unit). Estos avances son muy relevantes respecto de la multirresolución, ya que permiten generar representaciones simplificadas en tiempo real. Este trabajo presenta un algoritmo de visualización de terrenos basado en el algoritmo CDLOD. En su diseño se aplica un criterio más preciso de selección de nivel de detalle, el cual considera la rugosidad particular de cada zona del terreno en base a la percepción que el usuario tendrá de las mismas. Además, se incorporan las capacidades de teselado por hardware de la GPU correspondientes al Shader Model 5, y se implementa un mecanismo de geomorphing para suavizar las transiciones entre niveles de detalle.

Palabras Claves

Terrenos, Gráficos 3D, Multirresolución, GPU, Shaders, Teselado.

Trabajos Realizados

Las tareas más importantes realizadas en este trabajo son en primer lugar la recopilación y fichado de los algoritmos de visualización de terrenos más importantes encontrados en la bibliografía. Luego, el estudio del funcionamiento de la GPU, profundizando en sus últimas características (Shader Model 5). En base a esto se han desarrollo dos aplicaciones: un preprocesador de datos de altura y un visualizador del modelo multirresolución del terreno. Finalmente se han llevado a cabo una serie de pruebas de performance y la recopilación y análisis de los datos obtenidos.

Conclusiones

La técnica desarrollada realiza una representación multirresolución eficiente y escalable. El criterio de selección de nivel de detalle reduce la cantidad total de geometría para una misma calidad visual, y el mecanismo de geomorphing minimiza los cambios de resolución perceptibles. El diseño basado en Shader Model 5 disminuye el procesamiento realizado en CPU. Los resultados obtenidos posibilitan la utilización del algoritmo en aplicaciones interactivas que requieren de la visualización en tiempo real de modelos de terrenos.

Trabajos Futuros

Se han identificado tres líneas de trabajo futuro, de menor a mayor generalidad. En primer lugar, la unión entre bloques de terreno podría mejorarse realizando una adaptación de los bordes entre bloques vecinos. Por otro lado, podría desarrollarse un mecanismo de streaming de datos para soportar terrenos muy extensos y detallados. Finalmente, pueden explorarse otras técnicas multirresolución para potenciar la riqueza visual e inmersión del usuario, como texture-blending, modelos de iluminación y sombreado, o incluso elementos sobre la superficie como vegetación.

Fecha de la presentación: Diciembre 2014

UNIVERSIDAD NACIONAL
DE LA PLATA

FACULTAD DE INFORMÁTICA

Visualización 3D de terrenos multirresolución

TESINA DE LICENCIATURA EN SISTEMAS

Presentada por: *Lucas Federico Borrelli*
Dirigida por: *Dra. María José Abásolo*

Diciembre 2014

AGRADECIMIENTOS

Quiero agradecer a todas las personas que de una u otra forma, me han ayudado durante los últimos 7 años de mi vida a alcanzar este momento:

- A mi novia Malen, por su cariño y fuerzas durante todos estos años. Sin ella no hubiese sido posible llegar hasta acá.
- A mis padres, por creer, apoyar y confiar en esta gran aventura en mi vida.
- A mis hermanos y especialmente a mis sobrinos, por comprender y sobrellevar siempre mi ausencia debido al estudio.
- A mi directora María José, por su ayuda y orientación brindada para la realización y concreción de este trabajo.
- Finalmente, quiero agradecer a los profesores, compañeros y amigos de la facultad, con los que he vivido imborrables momentos durante estos años. A todos ellos les deseo lo mejor.

ÍNDICE

RESUMEN

Los algoritmos diseñados para representar y visualizar terrenos en mundos virtuales 3D son muy importantes en aplicaciones como sistemas de información geográfica, simuladores de vuelo o de entrenamiento, videojuegos, etc. El principal desafío radica en alcanzar la performance necesaria para una visualización en tiempo real de grandes extensiones de terreno. Las técnicas desarrolladas aplican el concepto de multirresolución para reducir la complejidad del modelo de terrenos.

El propósito de este trabajo ha sido desarrollar un algoritmo multirresolución eficiente que utilice los últimos avances del hardware gráfico, la GPU (*Graphics Processing Unit*). Estos avances son muy relevantes respecto de la multirresolución, ya que permiten generar representaciones simplificadas en tiempo real.

Este trabajo presenta un algoritmo de visualización de terrenos basado en el algoritmo CDLOD. En su diseño se aplica un criterio más preciso de selección de nivel de detalle, el cual considera la rugosidad particular de cada zona del terreno en base a la percepción que el usuario tendrá de las mismas. Además, se incorporan las capacidades de teselado por hardware de la GPU correspondientes al Shader Model 5, y se implementa un mecanismo de *geomorphing* para suavizar las transiciones entre niveles de detalle.

Finalmente, la técnica desarrollada permite realizar una representación multirresolución eficiente y escalable en niveles de detalle. Los buenos resultados obtenidos posibilitan su utilización en aplicaciones interactivas que requieren de la visualización en tiempo real de modelos de terrenos para escenarios virtuales.

***Palabras clave**: Terrenos, Gráficos 3D, Multirresolución, GPU, Shaders, Teselado.*

ABSTRACT

The algorithms designed for representing and visualizing terrains in 3D virtual worlds are very important in application such as geographic information systems, flight or training simulators, videogames, etc. The main challenge lies on reaching the performance for a real-time visualization of large terrains. The developed techniques apply the concept of multiresolution to reduce the complexity of the terrain model.

The purpose of this work was to develop an efficient multiresolution algorithm which employs the last advances in graphics hardware, the GPU (Graphics Processing Unit). These advances are especially relevant in the field of multiresolution because they make it possible to generate simplified representations in real-time.

This work presents a terrain visualization algorithm based on CDLOD algorithm. Its design includes a more precise criterion for level of detail selection which takes into account the roughness particularities of terrain zones based on user perception. It also exploits the hardware tessellation capabilities introduced in GPUs that support Shader Model 5. In addition, it implements a geomorphing mechanism to smooth the transitions between levels of details.

Finally, the developed technique achieves an efficient and scalable multiresolution representation in levels of detail. The obtained results allow interactive applications to utilize the algorithm in real-time visualization of terrain models.

***Keywords**: Terrains, 3D graphics, Multirresolution, GPU, Shaders, Tessellation.*

1 INTRODUCCIÓN

1.1 Motivación

Los gráficos por computadora, en especial los referidos al 3D, han tenido un crecimiento muy importante durante los últimos 15 años. Este avance ha sido potenciado gracias a la expansión y al incremento de las capacidades de procesamiento y memoria de las placas gráficas especializadas para esta tarea, las GPUs (*Graphics Processing Unit*).

Dentro de los gráficos por computadora, son particularmente importantes los algoritmos relacionados con la representación y visualización de terrenos. Estos se encargan de administrar los datos de modelos de elevación -mapas de altura o *heightmaps*- para lograr representar escenarios basados en terrenos, y son importantes en un gran número de aplicaciones, como por ejemplo simuladores de vuelo o de entrenamiento, sistemas de información geográfica, realidad virtual, videojuegos, etc.

La principal dificultad a la que se enfrentan las técnicas de visualización de terrenos es obtener una visualización eficiente en tiempo real para lograr terrenos que luzcan realistas. La visualización en tiempo real impone restricciones estrictas en la eficiencia de los sistemas de visualización, por lo que es necesario a su vez negociar la calidad de la visualización con un uso limitado de recursos. Es por esta razón que los algoritmos que han sido propuestos utilizan el concepto de multirresolución para generar representaciones simplificadas, por ejemplo, realizando una representación de menor nivel de detalle o LOD (*Level Of Detail*) a medida que la superficie del terreno está más lejos del punto de observación.

Existen dos modelos generales de multirresolución, los modelos de triangulación irregular y los modelos regulares. En el primer caso, que es el más general, el modelo de multirresolución se basa en un enfoque de triangulación totalmente irregular de la malla el cual no impone restricciones en la conectividad. Por otro lado, los modelos de conectividad regular, imponen restricciones en la conectividad de la malla y simplifican las operaciones de refinamiento. Estos modelos son menos costosos en términos de almacenamiento, transmisión y visualización, ya que mucha de la información requerida para todas estas tareas se vuelve implícita y quedan particularmente bien adaptado para los terrenos, ya que los datos de alturas topográficas provienen en general en formato de grilla.

Hoy en día, es posible implementar algoritmos que serán ejecutados directamente por las unidades de procesamiento gráficas (GPUs). Dado el crecimiento exponencial de las capacidades de procesamiento de las mismas, el costo de las operaciones de transformación 3D se ha vuelto prácticamente despreciable en la actualidad. Los

modelos de conectividad regular se han visto muy favorecidos ya que pueden explotar de forma más compacta y eficiente el intercambio CPU-GPU en el hardware actual y gracias a esto, los enfoques regulares han logrado producir los métodos más eficientes de la actualidad.

La investigación de métodos que utilizan la multirresolución para lograr una adaptación dinámica de la complejidad del modelo visualizado ha sido, y todavía es, un campo de investigación muy activo en el área de los gráficos por computadora. Los algoritmos que se han desarrollado no implican que el crecimiento del área se haya detenido, sino que es un área de constante trabajo dirigido a mejorar distintos aspectos para obtener más realismo, aprovechando las capacidades de procesamiento y memoria de las GPUs en constante crecimiento.

Los últimos avances en las capacidades de procesamiento gráfico han introducido la posibilidad de realizar teselados de geometría directamente por hardware (GPU), y con una mínima intervención por parte de la CPU. Estos avances son muy relevantes respecto de la multirresolución, ya que proponen el desarrollo de sistemas más flexibles para la generación de representaciones simplificadas en tiempo real. La visualización de terrenos debe proponer nuevos algoritmos o adaptar los existentes, para explotar al máximo la potencia de cómputo que la nueva tecnología pueda presentar.

1.2 Objetivos y contribución esperada

El objetivo general de este trabajo es estudiar algoritmos de visualización de modelos de terrenos en 3D que aplican el concepto de multiresolución, así como explorar las capacidades de la unidad de procesamiento gráfico o GPU (*Graphics Processing Unit*) para la implementación de dichos algoritmos.

El objetivo particular de esta tesina es el diseño, implementación y prueba de un algoritmo de visualización 3D de terrenos multirresolución, el cual sea implementado tanto en la CPU como en la GPU de acuerdo a las ventajas que ésta ofrezca, para lograr que sea eficiente en el uso de los recursos de procesamiento y con buenos resultados visuales.

Para esto se propone hacer un estudio de diferentes algoritmos de visualización de terrenos multiresolución para conocer el estado del arte de las técnicas utilizadas.

Además, se propone estudiar el funcionamiento de la GPU y la forma de programarla, de manera de poder hacer un uso eficiente de las nuevas potencialidades que ésta presenta para la implementación de algoritmos de visualización de terrenos.

El aporte que resultará de la concreción de esta tesina, es la recopilación y análisis de los algoritmos de visualización de terrenos 3D multiresolución existentes en la bibliografía actual, y realizar la propuesta de diseño e implementación de un algoritmo propio que mejore en algún aspecto el funcionamiento de los algoritmos similares investigados.

1.3 Organización del presente informe

El presente informe se divide en los siguientes capítulos:

El capítulo 1 presenta la introducción al tema de esta tesina "Visualización 3D de terrenos multirresolución".

El capítulo 2 presenta los aspectos fundamentales del modelado de terrenos 3D multirresolución. Se describe la construcción de mallados poligonales y la necesidad de disminuir la complejidad de los mismos. Se introducen los métodos de cambio de resolución, y las triangulaciones que esos métodos utilizan. Además se presenta el concepto de nivel de detalle, la necesidad de disponer de triangulaciones que adapten su resolución, y los problemas que pueden surgir en la creación de las mismas.

El capítulo 3 está dedicado a presentar los mecanismos de generación de gráficos 3D a través del uso de la GPU (*Graphics Proccessing Unit*). Para esto se abordan el concepto *pipeline* gráfico, se presentan las interfaces de programación más populares para la GPU, las últimas características que ésta presenta, y la forma de programarla mediante *shaders*.

El capítulo 4 presenta los algoritmos multirresolución más relevantes de la bibliografía actual. Los algoritmos son clasificados en dos grupos: Aquellos denominados "clásicos" o Pre-GPU, y aquellos más modernos o Post-GPU. El capítulo finaliza realizando una reseña de las derivaciones y extensiones de los algoritmos presentados en secciones anteriores.

El capítulo 5 detalla los aspectos de diseño del algoritmo de visualización de terrenos 3D multirresolución desarrollado como parte de la concreción de este trabajo. Se realiza un análisis de los algoritmos presentados en el capítulo 4, se presentan las motivaciones que impulsaron el diseño del nuevo algoritmo, y se describe cada aspecto del diseño en detalle.

El capítulo 6 describe los detalles de implementación y los resultados de las pruebas realizadas sobre el algoritmo de visualización de terrenos descripto en el capítulo anterior.

El capítulo 7 expone las conclusiones arribadas luego del desarrollo de la tesina, y presenta algunas líneas posibles de trabajos futuros.

Finalmente, se incluyen la bibliografía referenciada y dos apéndices complementarios, el primero con la documentación de clases implementadas y el segundo un manual de usuario.

2 MODELOS DE TERRENOS MULTIRRESOLUCIÓN

En este capítulo se presentan los aspectos más importantes del modelado de terrenos multirresolución, y se compone de las siguientes secciones:

En la sección 2.1 se describen los aspectos básicos en la construcción de modelos de elevación de terrenos a partir de conjuntos de muestras de elevación. En la sección 2.2 se caracterizan los métodos de triangulación posibles para formar el mallado de la superficie del terreno, y en la sección 2.3 se presenta brevemente los posibles modos de colorear la superficie del mismo mediante texturas.

La sección 2.4 describe los distintos aspectos relacionados a la visualización multirresolución de terrenos. Se introduce el concepto de multirresolución para la visualización de terrenos extensos (sección 2.4.1) y se describen los métodos básicos de cambio de resolución (sección 2.4.2). Además se mencionan los criterios y métricas utilizadas en la determinación del nivel de detalle (sección 2.4.3), y por último se describen los problemas que un algoritmo debe enfrentar para la creación de mallados multirresolución (sección 2.4.4).

2.1 Modelado digital de elevaciones

El modelado de terrenos se realiza mediante una superficie en el espacio tridimensional. Geométricamente, esa superficie puede entenderse como una función del tipo z(x,y), siendo x e y las coordenadas de un punto en un plano (x, y). La posición de cualquier punto sobre el terreno, estaría dada por su ubicación en el plano horizontal, más la información de la altura asociada z en ese punto.

En general, la superficie del terreno no puede ser caracterizada precisamente en términos matemáticos, por lo que la manera de realizar su definición, es cuantificando su elevación en intervalos discretos. El conjunto de muestras de elevación de una porción de terreno o zona geográfica se lo denomina Modelo Digital de Elevación o DEM (*Digital Elevation Model*).

En un DEM la información de elevación está especificada a intervalos regulares[1], es

[1] El término DEM suele utilizarse en la bibliografía para referirse específicamente a los conjuntos de muestras con distribución regular. Sin embargo, en muchas ocasiones también se lo utiliza para referenciar otro tipo de representación digital de superficies topográficas, como aquellas en que las muestras se encuentran distribuidas irregularmente e interconectadas para formar el mallado de la superficie. Para esta tesina se optó por utilizar el término DEM para hacer referencia a los conjuntos de muestras en forma de grilla regular. Los conjuntos de datos con distribución irregular o *TINs* (Triangular

decir, que forma una grilla o matriz con los valores o muestras de altura. Por esta razón sólo es necesario almacenar el valor de elevación z de cada muestra, ya que la posición (x,y) respecto del plano puede inferirse por su posición en la grilla. En este sentido, el tratamiento de los modelos de elevación tiende a ser consistente o fácil de realizar, ya que se trabaja sobre una gran matriz numérica.

Normalmente, la información contenida en un DEM es obtenida a través del postprocesamiento de los datos de muestreo de alturas de la superficie de terrenos naturales. Las muestras son obtenidas mediante sensado remoto desde satélites utilizando distintas técnicas, como ser LIDAR (*Light Detection And Ranging*), Interferometría mediante SAR (*Synthetic Aperture Radar*), aunque también pueden utilizarse aviones igualmente equipados, equipos GPS o teodolitos cuando se requiere de información puntual o más precisa.

Los DEMs son distribuidos junto a información adicional, como ser el espaciado entre las muestras y la resolución vertical. Existen formatos especializados que incluyen estos metadatos en un mismo archivo. Estos formatos son por lo general desarrollados por agencias espaciales o cartográficas de distintos países, con el objetivo de crear soluciones robustas y muy abarcativas respecto de la cantidad y calidad de información que deben almacenar. Esto produce a la vez que no sea fácil la utilización de estos formatos. Así por ejemplo no sólo incluyen datos de muestreo, sino también es posible especificar caracterizaciones especiales de los datos, como georreferenciación, reporte de calidad y precisión de datos, zonas especiales de interés, diccionarios de datos, entre otros. Algunos formatos que pueden mencionarse son DTED (*Digital Terrain Elevation Data*) de uso militar y USGS DEM (*United States Geological Survey Digital Elevation Model*) del Servicio Geológico de los Estados Unidos. El "estándar de transferencia de datos espaciales" o SDTS (*Spatial Data Transfer Standard*), fue diseñado desde un principio con el objetivo de aliviar esta situación y mejorar el intercambio y transferencia de información espacial entre plataformas disímiles.

Heightmaps

Una forma práctica de almacenar la matriz numérica de elevaciones de un DEM, es utilizando archivos de imágenes. Las imágenes utilizadas como medio de almacenamiento de modelos de elevación son conocidas como "mapas de altura" o *heightmap*. Los *heightmaps* son un medio muy utilizado en computación gráfica para el almacenamiento de DEMs, ya que son fáciles de manipular, procesar y visualizar, y por la gran disponibilidad de herramientas de software para trabajar con imágenes.

Los *heightmaps* se conforman realizando un mapeo 1 a 1 entre la matriz de datos discretos (píxeles) y los valores o muestras de altura. La manera de codificar los valores de altura, depende del tipo de imagen utilizada. En formatos de imágenes tipo RGB, una muestra de altura puede codificarse con sobrada resolución ya que un píxel puede

irregular Networks) son abordados en la sección 2.2.1, ya que por su concepción conforman triangulaciones.

representar más de 16 millones de colores distintos. Por eso es común que se utilicen sólo escalas de grises para codificar los valores de altura. Así por ejemplo, suelen utilizarse los colores más oscuros para representar las zonas de menor altura, y los cercanos al blanco para las más altas (Ver figura 2.1). Entre la gran variedad de formatos de imágenes disponibles, también debe tenerse en cuenta factores como la compresión utilizada. Algunos formatos como ser PNG o TIFF utilizan compresión sin pérdida, mientras que otros formatos como JPEG utilizan un esquema de compresión con pérdida, por lo que su utilización puede ser inapropiada.

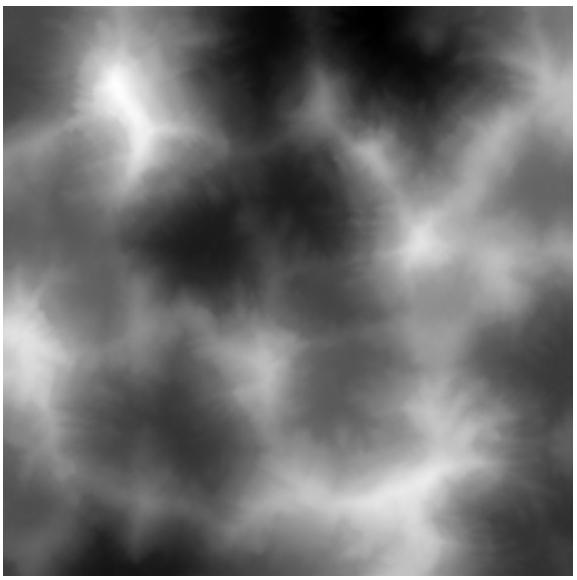

Figura 2.1: Ejemplo de heightmap de zona montañosa. Los colores más claros representan las zonas de mayor relieve (picos y filos de montañas).

Por último, existen herramientas especializadas para la definición, tratamiento y generación procedural de *heightmaps*. Como indica Smelik et al. [32], la filosofía de las herramientas de modelado procedural es la de diseñar funciones o procedimientos matemáticos para crear de forma automática contenido destinado a escenarios virtuales, como ser texturas, modelos geométricos, animaciones, *heightmaps*, etc. Este tipo de herramientas pueden ser muy útiles en el desarrollo de otro software, como ser una herramienta GIS (*Geographic Information System*), ya que no es necesario contar con los datos reales al menos en una primera etapa de prototipado. Además, son muy utilizadas en la generación de *heightmaps* para ambientes artificiales de videojuegos u otros escenarios de entrenamiento o de realidad virtual.

2.2 Triangulaciones

Según explican De Berg et al. [4], triangular es formar una malla de triángulos a partir de una estructura de puntos o de la descomposición de un polígono, mientras que una triangulación es una instancia particular de una misma estructura de puntos o

polígono, triangulado de una determinada manera. El triángulo es la figura plana o polígono más utilizado en computación gráfica. Por su simplicidad, es posible trabajar con ellos sin mayores trastornos, ya que si se trasladada cualquiera de los vértices de un triángulo, no se afecta la coherencia de la figura -no deja de ser triángulo-. En cambio con otras figuras es necesario mantener restricciones especiales de manera de no invalidar la figura. Por otro lado, también existe hardware gráfico especializado que permite trabajar con triángulos y visualizarlos a gran velocidad, como se verá en el capítulo 3.

Disponiendo de los datos de altura, se realiza la representación de la superficie del terreno en el espacio 3D. Para esto, una malla es formada mediante vértices -relativos a un sistema de coordenadas- construidos a partir de las muestras del DEM. Luego, los vértices de la malla son interconectados formando los triángulos los cuales completan los espacios entre los puntos de la superficie. Como todo modelo es una aproximación de la realidad, al completar la superficie con triángulos interconectados entre los vértices, se está realizando una aproximación de la altura de la superficie entre muestras adyacentes. Como resultado, queda formada una malla (triangulación) que representa o aproxima la superficie del terreno.

Existen dos modelos generales según Pajarola et al. [29]. En el caso más general, el modelo se basa en un enfoque de triangulación irregular de la malla. Por otro lado están los modelos de conectividad regular y jerárquicos los cuales imponen restricciones en la conectividad del mallado.

A continuación se describen ambos modelos.

2.2.1 Triangulaciones irregulares

Las triangulaciones irregulares forman mallas de triángulos de tamaño y forma arbitraria. A diferencia de otras triangulaciones, no requieren de una disposición especial del conjunto de vértices. Pueden distinguirse dos clases de triangulaciones irregulares, las redes irregulares de triángulos o TINs (*Tringulated Irregular Networks*) y las triangulaciones *Delaunay*.

Las TINs, como por ejemplo utiliza Hoppe en [18], no imponen restricción alguna en la conectividad de los puntos y son teóricamente capaces de producir la aproximación de geometría mínima del terreno. Las triangulaciones *Delaunay*, como utilizada Cohen et al. [5], permiten una distribución general de los vértices, aunque la conectividad de la malla se restringe respecto de una TIN. El criterio de *Delaunay* establece que para cada triángulo *t* perteneciente a la triangulación, ningún vértice perteneciente a otro triángulo, puede estar dentro del círculo que circunscribe a *t*. Esto restringe el número de triangulaciones posibles, lo que produce una representación menos precisa o que podría no ser óptima. Sin embargo, el propósito de esta restricción es el de maximizar el ángulo mínimo de todos los triángulos para prevenir la aparición de triángulos finos y alargados a lo largo de la malla, ya que en general no contribuyen a la calidad final de la imagen. En la figura 2.2 se muestran dos ejemplos de estas triangulaciones.

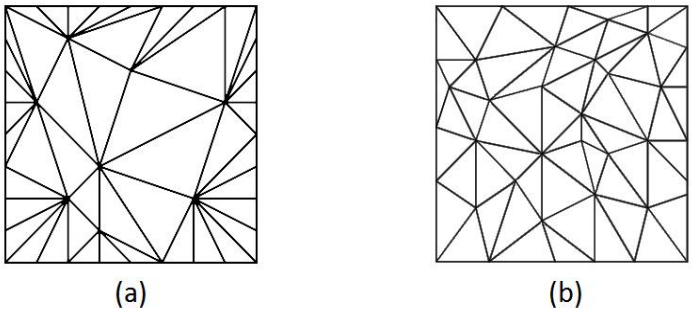

Figura 2.2: Triangulaciones irregulares: en la imagen (a) una TINs (Tringulated Irregular Networks), y en la imagen (b) una triangulación Delaunay.

Estos enfoques son teóricamente capaces de producir las mejores representaciones para un número dado de triángulos. Sin embargo, esta flexibilidad tiene costo, ya que la conectividad de la malla y sus dependencias deben ser representadas explícitamente.

2.2.2 Triangulaciones regulares y jerárquicas

Estos modelos son menos costosos en términos de almacenamiento, transmisión y visualización y modificación, ya que mucha de la información requerida para todas estas tareas se vuelve implícita. Además, quedan particularmente bien adaptado para terrenos, ya que los datos de alturas topográficas provienen en forma de grilla.

Triangulaciones regulares

En una triangulación regular, la malla se conforma mediante una grilla de vértices con una separación regular entre filas o columnas (ver figura 2.3). En general el número de vértices horizontal y vertical de la grilla deben ser iguales y de la forma 2^n+1. Como resultado se obtienen 2^n cuadriláteros, cada uno formado por 4 vértices vecinos. Luego los cuadriláteros son divididos por una diagonal u otra formando 2 triángulos rectángulos isósceles. En los capítulos 4 y 5 se prestará especial atención a estas técnicas ya que son las más utilizadas hoy en día para la visualización de terrenos mediante aceleración por hardware gráfico.

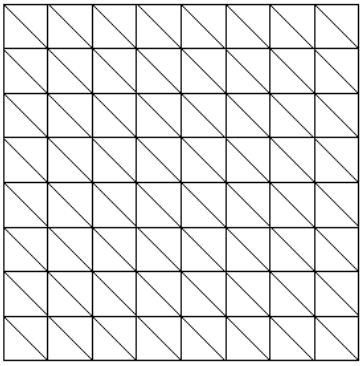

Figura 2.3: Triangulación regular formada a partir de una grilla de 9x9 vértices.

Triangulaciones jerárquicas

Las triangulaciones jerárquicas subdividen recursivamente un polígono -construido con los vértices externos de la grilla- en instancias más pequeñas del mismo polígono. En primer lugar, las triangulaciones *Bintree*, por ejemplo utilizadas por Duchaineau et al. [6], utiliza triángulos rectángulos isósceles y los subdivide recursivamente en dos triángulos más pequeños. La subdivisión es realizada por el medio de la hipotenusa a través del *vértice base*, quedando conformados dos triángulos rectángulos isósceles de la mitad de tamaño y de manera opuesta (Ver figura 2.4). Luego cada uno se esos nuevos triángulos es vuelto a ser dividido y así sucesivamente hasta completar los vértices disponibles. Como resultado del proceso se obtiene un árbol binario de triángulos o *Triangle Bintree*. En general son requeridos dos *Bintree* por grilla de vértices, uno para cada triángulo principal de la grilla.

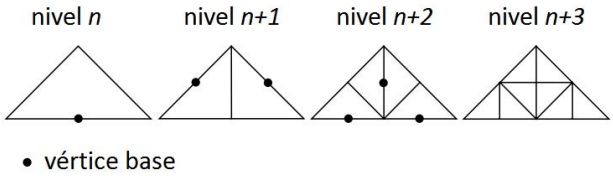

Figura 2.4: Triangulación Bintree.

Existen también triangulaciones construidas a partir de *quadtrees* restringidos, como es el trabajo publicado por Pajarola et al. [27]. Estas triangulaciones se basan en una idea similar a las triangulaciones *Bintree* y, como el mismo autor explica en [29], potencialmente pueden producir la mismas clases de triangulaciones dependiendo de las reglas de subdivisión específicas que se apliquen. En este caso, la unidad de subdivisión es el cuadrado. Cada cuadrado puede ser dividido en cuatro cuadrados más pequeños de un cuarto de tamaño, lo que conforma un árbol cuaternario o *quadtree*. El *quadtree* es restringido de manera de no permitir que dos cuadrados adyacentes difieran en más de un nivel. Finalmente, cada cuadrado es subdividido en cuatro triángulos rectángulos isósceles, un triángulo por lado, a menos que un lado del cuadrado quede adyacente a un cuadrado de menor LOD en cuyo caso dos triángulos son formados en ese lado (ver figura 2.5).

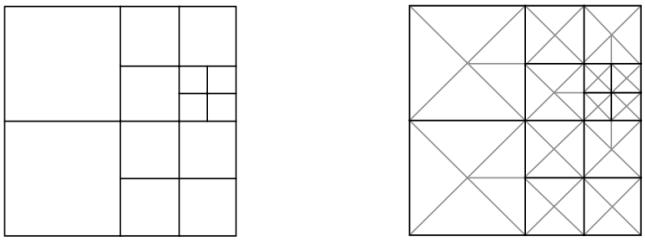

Figura 2.5: Triangulación Quadtree Restringido.

2.3 Texturas

Como se ha venido mencionando, para representar la superficie del terreno, se construyen triángulos entre los puntos o alturas muestreadas, formando una malla tridimensional. Finalmente, es necesario dar color o texturar la superficie representada por la malla, para generar el modelo final del terreno a visualizar. La figura 2.6, resume el proceso de modelado de terrenos.

Existen muchas maneras de colorear la superficie del terreno. En algunos casos, como pueden ser sistemas de información geográfica, se intenta recrear la superficie real lo mejor posible por lo que se utilizan fotografías aéreas e imágenes satelitales georeferenciadas de la misma región representada, obteniéndose el denominado modelo topografico texturado. Otro tipo de aplicaciones requieren del uso de texturas artificiales, como ser textura en cuadrícula, coloreado por elevación, u otro tipo de texturas procesadas. En el caso de los videojuegos u otros ambientes simulados, suelen utilizarse máscaras para combinar distintas texturas con el objetivo de simular las diferentes características de la superficie del terreno.

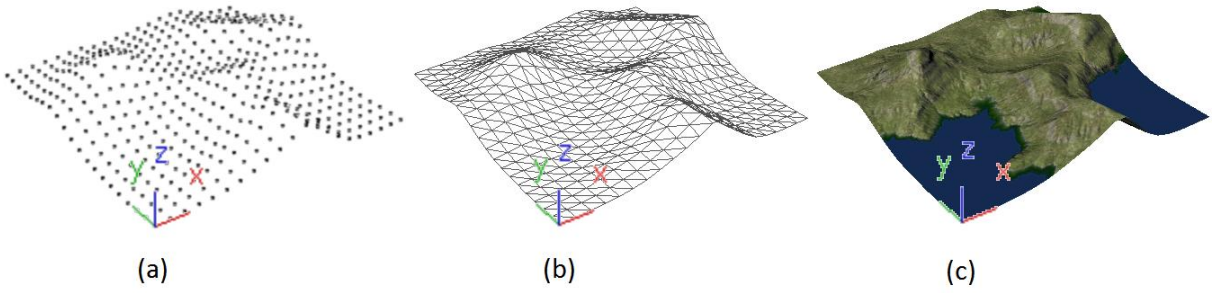

Figura 2.6: Modelado de terreno. En (a) se representan los vértices construidos a partir de las muestras de altura, en (b) se muestra la malla poligonal construida a partir de los vértices de (a), y en (c) el modelo final texturado a visualizar.

2.4 Multirresolución

2.4.1 Qué es la multirresolución

La visualización de terrenos se enmarca dentro de la representación de escenarios virtuales. Estos escenarios deben representar lo más fielmente posible a la realidad que modelan, de manera de lograr la visualización de ambientes más precisos, realistas e inmersivos. Para esto es importante poder dotar al modelo del mayor detalle que sea posible.

Un método simple y efectivo de lograrlo, consistiría en emplear todos los valores de altura del *heightmap* para formar la malla de la superficie del terreno, lo que permitiría formar un modelo completo y preciso. Sin embargo, un enfoque de estas características no es válido en la visualización de terrenos si se pretende representar superficies extensas, necesarias en aplicaciones para modelado digital de terrenos como ser aplicaciones GIS, simuladores de vuelo o videojuegos. El gran tamaño de estos terrenos impide mantener un rendimiento adecuado en este tipo de aplicaciones.

Entonces, el problema radica en poder visualizar un modelo con el mayor detalle que sea posible, a la vez que se permita acelerar las aplicaciones interactivas. Por otro lado, mientras se experimenta un avance constante en las capacidades del hardware, también se pretende cada vez mayor realismo y precisión. Puede decirse que se plantea una carrera que no tiene fin, ya que la complejidad de los modelos siempre va en aumento. Dado esto, la mejor manera de aprovechar las capacidades de cómputo para lograr una visualización de modelos extensos o de mucho detalle, es desarrollando algoritmos que abordan el problema mediante la multirresolución.

El concepto de multirresolución es muy abarcativo, ya que por ejemplo es utilizado en matemática para análisis de funciones, pero de manera general, se puede decir que los modelos multirresolución representan objetos en diferentes niveles de resolución o precisión.

En computación gráfica, particularmente en visualización de gráficos 3D, el concepto de multirresolución se aplica principalmente a objetos geométricos complejos, es decir, representados por mallas genéricas de gran cantidad de polígonos. En este sentido, los modelos multirresolución tienen como objetivo desarrollar algoritmos que permitan visualizar mallas poligonales en diferentes niveles de detalle o LOD (*Level Of Detail*).

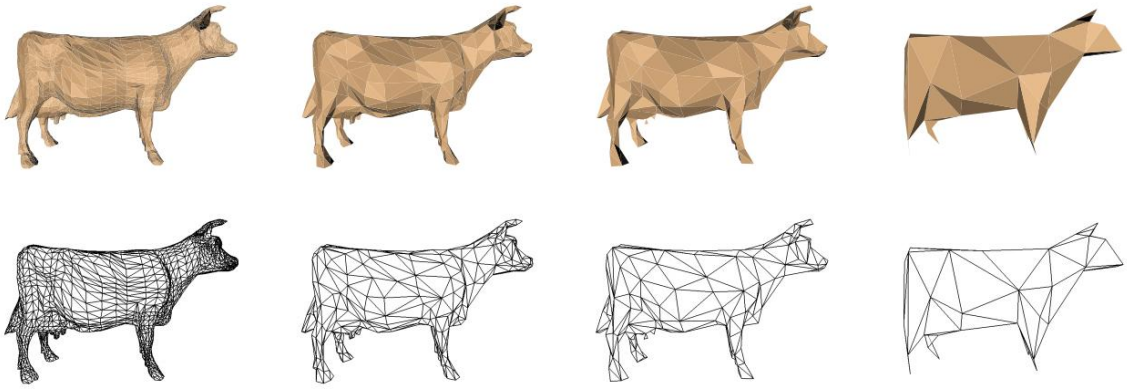

Figura 2.7: Un modelo genérico representado en distintos niveles de detalle, de mayor (izquierda) a menor detalle (derecha). [10].

El nivel de detalle (LOD), es el nivel de resolución o la medida en que una malla o zona de la misma, aproxima su geometría a un objeto original, también denominado "de máxima resolución". Un modelo puede ser representado en diferentes niveles de detalle. Los niveles de menor detalle, aproximan el modelo de manera poco precisa y utilizan menor cantidad de triángulos, mientras que los niveles de mayor detalle utilizan más cantidad de triángulos para lograr una alta aproximación al objeto (ver figura 2.7).

La principal ventaja de los modelos multirresolución es la de permitir un incremento de velocidad en el procesamiento ya que reducen la geometría, es decir el número de polígonos totales a visualizar.

2.4.2 Métodos de refinamiento y simplificación

Los métodos que permiten cambiar la resolución de mallas poligonales, deben procurar que las características que juegan un rol geométrico significativo como ser el volumen, los contornos, y la forma de la malla original no se vean alterados.

Estos métodos pueden ser agrupados en dos categorías según la forma de su proceder. En la primera categoría se encuentran los métodos de simplificación. Estos métodos comienzan con una malla de máxima resolución y realizan un proceso de reducción de complejidad o número de triángulos hasta que una determinada resolución es alcanzada.

La otra clase son los métodos de refinamiento. En este caso, el proceso es el inverso al anterior. El refinamiento comienza desde una malla de muy baja resolución, e iterativamente agregan detalle en las zonas donde sea necesario, como se muestra en la figura 2.8.

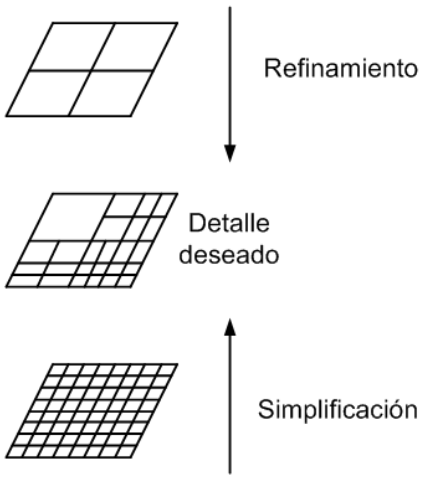

Figura 2.8: Métodos de simplificación y refinamiento

Comparando ambos métodos, es posible obtener mejores triangulaciones con los métodos de simplificación. Esto significa que para un mismo número de triángulos pueden obtenerse mejores aproximaciones del modelo de máxima resolución. La desventaja de la simplificación, es la cantidad de cálculos que se requieren en su proceso, ya que dependen de manera proporcional al tamaño de los datos de entrada. En cambio, aunque los métodos de refinamiento puedan producir triangulaciones menos óptimas ya que en cada iteración de su proceso sólo poseen información parcial del modelo, no son sensibles al tamaño de la estrada. Estos métodos requieren menor cantidad de cálculos ya que en realidad sólo dependen del tamaño de la salida, es decir la malla con el detalle deseado.

Los primeros trabajos publicados en esta área, como por ejemplo [15], se basaron en métodos de simplificación sobre mallas genéricas. Esto se debe principalmente a que los modelos a simplificar no poseían un tamaño demasiado grande, y además porque era muy costosa la generación de gráficos 3D y era necesario poseer un modelo representado con la menor cantidad posible de primitivas. Los algoritmos utilizados hoy en día emplean en general esquemas de refinamiento, ya que los modelos que se requiere visualizar han incrementado su tamaño en algunos órdenes de magnitud, y además porque el hardware gráfico permite visualizar polígonos adicionales a los estrictamente necesarios sin afectar la performance final.

Finalmente, aunque el concepto de multirresolución ha sido extensamente estudiado para mallas 3D genéricas, y sus estructuras de datos y algoritmos también pueden ser aplicables a los modelos de elevación, los sistemas más eficientes de hoy en día se sustentan en un conjunto de métodos especialmente adaptados para la visualización de terrenos. Estos métodos surgen de diferentes modelos especializados en multirresolución de terrenos, de variada eficiencia y generalidad, que han sido propuestos.

2.4.3 Determinación del nivel de detalle

El ajuste de la resolución es posible a partir de algoritmos multirresolución. La particularidad de estos algoritmos es que permiten lograr un nivel de detalle adaptativo con el objetivo de aproximar la malla de distinta manera y en distintas zonas del terreno. Además, como plantea Lindstrom et al. [22], para obtener una alta performance en la visualización, es esencial que la simplificación de mallas de alta resolución se realice en tiempo real. Es decir, que es necesario que el nivel de detalle sea dinámico, y se permita variar la resolución en forma interactiva, por ejemplo mientras el punto de observación se traslada. Como resultado se obtiene una malla que posee mayor detalle en las áreas de interés y menor detalle en el resto del modelo, y que puede ser ensamblada en tiempo de ejecución, dando a la aplicación mucho más control sobre la aproximación de la malla.

Criterios de selección de LOD

Pueden destacarse dos aspectos o criterios utilizados para decidir donde aplicar mayor o menor detalle a lo largo de la superficie del terreno. En primer lugar, pueden considerarse las características especiales de la superficie, como ser la existencia de picos o valles en las distintas regiones del terreno. Este criterio considera estas zonas como base para decidir donde aplicar el detalle. Si se provee de mayor resolución en zonas particularmente rugosas es posible dar mejor calidad visual en las regiones que requieren mayor cantidad de triángulos para realizar una aproximación similar. Un esquema que no tenga en cuenta estas características puede necesitar mucha mayor cantidad de triángulos para alcanzar la misma calidad visual.

Otro aspecto a considerar es el fenómeno de la perspectiva. La perspectiva permite apreciar una varíación de tamaño de un objeto dependiendo de su distancia al punto de observación. Reproduciendo este fenómeno en computadora, el objeto será más pequeño a medida que se aleja, por lo que cubrirá un área menor de píxeles en pantalla. Los triángulos de la malla en esa posición lejana, serán muy pequeños cuando sean convertidos a píxeles, por lo que tendrán una contribución mínima en la imagen final. Por lo tanto, las regiones de la superficie más lejanas al punto de observación pueden ser aproximadas con menor cantidad de triángulos sin comprometer la calidad visual. Así mismo, las partes del terreno que están más cerca al punto de observación pueden ser visualizadas utilizando niveles de mayor detalle. Ya que los niveles de menor detalle pueden ser aproximadas con menos triángulos, grandes áreas del terreno serán visualizadas rápidamente sin influir en la calidad final de la imagen y ganando en performance.

Métricas

En la sección anterior se introdujeron los criterios más utilizados para decidir donde aplicar mayor o menor detalle a lo largo de la superficie del terreno. Para aplicar estos

criterios, los algoritmos multirresolución utilizan métricas con las cuales es posible medir si una aproximación geométrica es adecuada en una determinada zona del terreno. Existe una gran variedad de métricas, de mayor o menor complejidad, y en muchos casos suelen combinarse.

En general, cuanto mayor es la precisión requerida, mayor es la complejidad del cálculo en una métrica, y en muchas ocasiones es preferible rescindir precisión para facilitar el cálculo y ganar en performance. Para detener el proceso de refinamiento o de simplificación, se utilizan umbrales o valores límite que permiten decidir cuando se ha alcanzado el nivel de detalle deseado. Los umbrales al igual que las métricas suelen ser representativos en el contexto de un algoritmo en particular.

Las métricas pueden clasificarse según dos grandes criterios: dependientes, y no dependientes del punto de observación. A continuación se describe brevemente esta clasificación y las subclasificaciones más importantes.

Independientes del punto de observación:

- **Número de primitivas:** Estas métricas son muy sencillas, miden simplemente el número de primitivas insumidas por una aproximación. Aunque podrían utilizarse para controlar que una representación multirresolución no exceda cierta cantidad de triángulos, o que se mantenga dentro de un rango, o para mantener un cierto número de divisiones de triángulos por segundo como parte de un algoritmo adaptativo, este tipo de métricas son poco habituales, ya que no permiten controlar la calidad de aproximación en una zona del terreno, ni tampoco la calidad visual con la que esa zona es percibida por el usuario. De manera *offline*, pueden utilizarse para requerir que el número de primitivas de un modelo muy complejo deba ser reducido por ejemplo a un modelo de la mitad de triángulos, un cuarto etc. Estas métricas son más utilizadas para la comparación de algoritmos desde el punto de vista del número de primitivas utilizadas por cada uno de ellos para representar o aproximar un mismo terreno (o zona del mismo). Puede observarse que para una determinada malla: $A \approx V + T$, con A el número de arista, V el número de vértices y T el número de triángulos. De esta manera es posible también comparar resultados medidos en base a distintas primitivas.
- **Error de aproximación en espacio de objeto:** En este caso la métrica calcula el error que existe entre una aproximación de menor resolución y otra que en general es la aproximación de la malla a máxima resolución, aunque también puede calcularse respecto de otra distinta. El error se mide como la distancia en espacio de objeto y puede calcularse sobre vértices, aristas, la superficie del triángulo, o del promedio de un grupo de triángulos. De esta manera, es posible saber que error en distancia existe entre una posible aproximación a utilizar y la máxima resolución considerada de error

cero. Esta métrica permite aplicar los criterios que consideran las particularidades del terreno, aplicando mayor detalle donde existen cambios abruptos de altura, por ejemplo en pico de montañas, y disminuyendo el detalle en valles o regiones mayormente planas. Variando el umbral de error permitido es posible variar la aproximación total del terreno considerando siempre las rugosidades particulares mencionadas.

Dependientes del punto de observación:

- **Distancia al punto de observación:** Esta métrica tiene en cuanta la distancia que existe entre el punto de observación y una zona particular del terreno. Así por ejemplo pueden considerarse rangos de distancia entre los cuales se aplica un nivel de dalle u otro. A medida que el observador se acerca a una zona, el punto de observación cambia de rango y por tanto corresponde aplicar una aproximación de mayor detalle. De manera inversa, a medida que se está más lejos de una región ésta recibirá menor detalle, por lo tanto esta métrica cumple con el criterio que considera la proyección perspectiva para decidir el nivel de detalle.
- **Error de aproximación proyectado en espacio de imagen:** En este caso se trata de una métrica compuesta. Se considera inicialmente el "Error de aproximación en espacio de objeto". Disponiendo de su valor ya calculado, se lo proyecta mediante perspectiva al espacio de imagen de la pantalla. Este nuevo valor obtenido mide ahora unidades de píxeles en pantalla, representando justamente cuantos píxeles abarca en pantalla el segmento del error de aproximación, desde la distancia a la que se encuentra del punto de observación. Este tipo de métricas son las más utilizadas por los algoritmos multirresolución, ya que proveen los mejores resultados visuales. Permiten controlar el nivel de detalle a aplicar en una determinada región del terreno considerando las particularidades de cada zona (rugosidad) dependiendo directamente de la percepción que el usuario tendrá de las mismas.

2.4.4 Transición entre niveles de detalle

Las técnicas de nivel de detalle deberían idealmente permitir la representación de modelos de terrenos multirresolución sin que se produzcan cambios aparentes o apreciables entre los distintos niveles de detalle de la superficie. El hecho de representar superficies con zonas de distinta resolución introduce nuevos problemas con los que estas técnicas deben lidiar.

Los principales problemas son las discontinuidades espaciales y las discontinuidades temporales. Las discontinuidades espaciales, son aquellas que se manifiestan en forma de grietas (*cracks*) entre zonas adyacentes de distinto LOD, y se producen por no utilizar los mismos vértices a lo largo de los bordes de estas zonas. Los vértices que no

son utilizados por la zona de menor detalle son denominados vértices-T ya que las interconexiones de aristas forman -en general- una T. Luego, las grietas aparecen porque la zona de menor detalle no puede aproximar la altura de los vértices-T (ver figura 2.9). Incluso si la altura de los vértices-T fuese la misma que la altura del borde vecino en ese punto, es posible que puedan producirse defectos en la imagen a causa de errores de aritmética de punto flotante en la interpolación de las aristas.

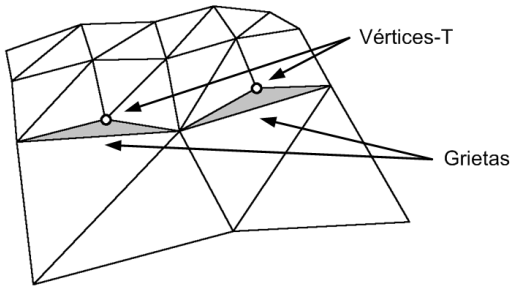

Figura 2.9: Grietas entre zonas de distinto LOD.

Por otro lado, las discontinuidades temporales se manifiestan a lo largo del tiempo, es decir a medida que nuevas imágenes o cuadros son generados como parte de la interactividad de la visualización, por ejemplo, a medida que el punto de observación se traslada. Las discontinuidades temporales hacen referencia al cambio relativamente rápido de la geometría del terreno que pueden producir efectos visualmente apreciables. Estos efectos son conocidos en la bibliografía como *popping* (salto o aparición repentina). Un *pop* es el fenómeno visual que se aprecia como un cambio repentino de la forma o sombreado de la geometría al cambiar la resolución entre dos niveles de detalle distintos. Existe un compromiso entre la resolución utilizada por cuestiones de performance y la notoriedad de este tipo de fenómenos. Cuanto menor es la resolución utilizada para visualizar una zona, mayor es la probabilidad de visualizar un *pop*.

3 GPU (GRAPHICS PROCESSING UNIT)

El presente capítulo está dedicado a presentar los mecanismos de generación de gráficos 3D a través del uso de la GPU (*Graphics Proccessing Unit*). Este capítulo se divide en las siguientes secciones:

La sección 3.1 presenta el concepto *pipeline gráfico* y la funcionalidad esencial que una implementación por software o hardaware (GPU) debe realizar para llevar a cabo la generación gráficos 3D.

En la sección 3.2 se define la GPU, se presentan las interfaces de programación más populares para su utilización, y la forma de programarla mediante *shaders*.

Por último, la sección 3.3 está dedicada al pipeline gráfico expuesto por DirectX versión 11, mediante el cual es posible el acceso a las últimas características que presenta la GPU.

3.1 El pipeline gráfico

3.1.1 Qué es el pipeline gráfico

Un *pipeline* es una secuencia de etapas o tareas que son realizadas en un orden determinado (similar a una línea de ensamblaje industrial), donde cada etapa recibe como entrada el resultado o salida, del proceso de la etapa anterior. Además en un *pipeline*, cada etapa puede trabajar simultáneamente sobre distintos elementos.

En computación gráfica, el término **pipeline gráfico** (*rendering pipeline*) se utiliza para hacer referencia al proceso llevado a cabo para lograr la visualización de gráficos 3D. El pipeline gráfico, es la secuencia de tareas necesarias para generar la imagen a partir de un ambiente o escena virtual.

Una escena virtual esta constituida por objetos 3D (compuestos por vértices) que tienen una posición y orientación en el espacio. Los objetos pueden tener color o textura y pueden considerarse características como ser las condiciones de iluminación o del ambiente.

Una escena también debe incluir una cámara virtual, esto es, el punto de observación desde el cual la escena es observada (ver figura 3.1). Los objetos que deben ser renderizados son aquellos que son “vistos” por la cámara, es decir, los que se encuentran dentro su volumen de visualización o *view frustum* (una pirámide truncada de base rectangular).

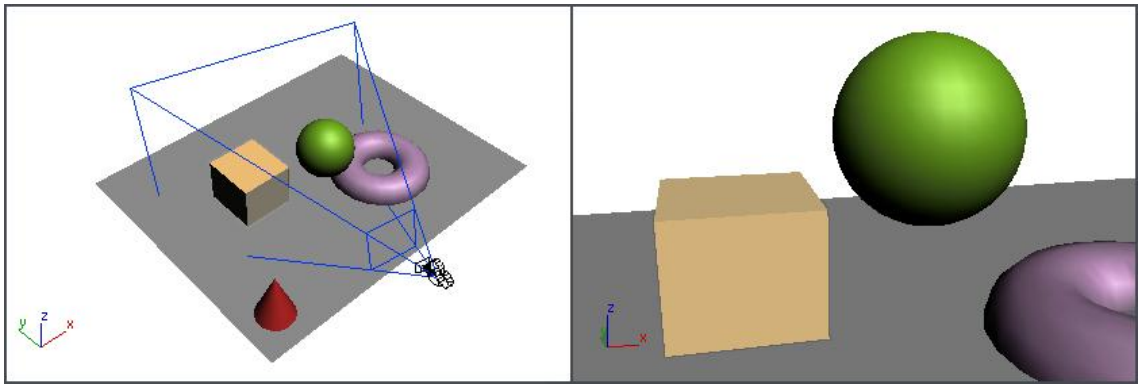

Figura 3.1: A la izquierda la definición de una escena, y a la derecha la escena vista desde la cámara.

Finalmente, toda la información de la escena es pasada a través del pipeline gráfico y utilizada para la generación de una imagen bidimensional o cuadro.

Las tareas que se realizan en el pipeline gráfico, están divididas funcionalmente en etapas: **Transformaciones espaciales**, **Recorte y Mapeo a Pantalla**, y **Rasterización y Mezcla**. La salida de la última etapa es la imagen final renderizada que será mostrado en un dispositivo, como la pantalla de un monitor. La aplicación es la encargada de proveer la definición de la geometría de los objetos de la escena (ver figura 3.2).

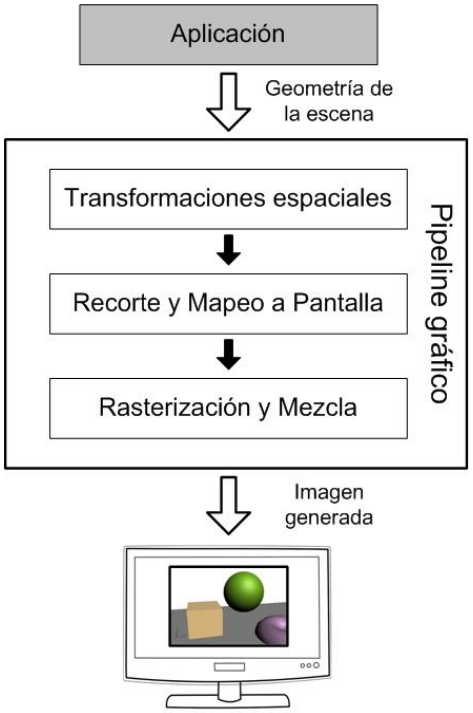

Figura 3.2: El pipeline gráfico

A continuación se describe cada una de las etapas del pipeline gráfico, y el proceso de transformación sobre los datos en cada una de ellas.

3.1.2 Primitivas geométricas

La aplicación es la encargada de organizar los objetos de la escena, mantener la representación geometría de los mismos, y luego suministrar esa representación al pipeline para su procesamiento.

Para representar los objetos de la escena, estos son modelados mediante la definición de mallas o conjuntos de primitivas geométricas, que a su vez están conformadas por vértices. Una primitiva es la unidad mínima de procesamiento y renderización soportada por un determinado pipeline. Las primitivas más comunes (Ver figura 3.3) son los Puntos (1 vértice por primitiva), las Líneas (2 vértices por primitiva), y los Triángulos (3 vértices por primitiva).

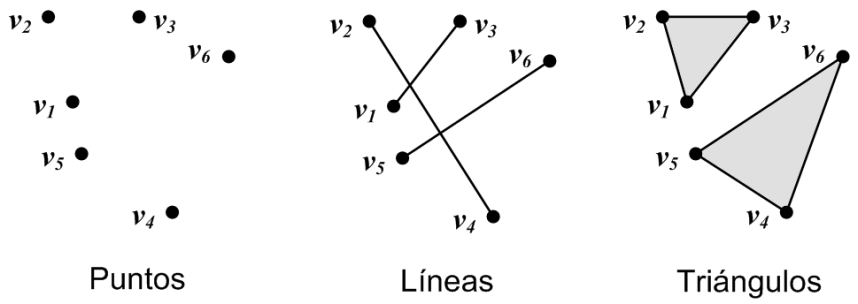

Figura 3.3: Tipos de primitivas

Los vértices de cada primitiva son definidos mediante una estructura de datos que contiene los datos necesarios para el proceso de renderización. La estructura de datos contiene principalmente las coordenadas que definen la *posición* del vértice en el espacio 3D. Opcionalmente, esta estructura puede contener un conjunto de atributos adicionales como ser el *color* utilizado para sombrear el área de pantalla cubierta por la primitiva, un vector *normal* que representa la dirección en que apunta la superficie de la primitiva en la posición del vértice y que puede ser utilizado para cálculos de iluminación, y finalmente las *coordenada de textura* que se utilizan para mapear una imagen sobre la primitiva.

Luego, los datos de la definición de los vértices, junto a la información de interconexión entre ellos para conformar las primitivas, son suministrados o ingresados al pipeline.

3.1.3 Transformaciones espaciales

Los vértices que conforman los objetos de la escena, son definidos inicialmente por la aplicación, respecto de un sistema de coordenadas local a cada objeto. En este estado se dice que un objeto se encuentra “no transformado” o en coordenadas del objeto o modelo.

Para poder generar la imagen final con los objetos situados en una determinada posición y orientacion en la escena, y ser observados desde el punto de vista de la cámara virtual, a cada vértice ingresado al pipeline se le aplica una serie de transformaciones lineales. Cada una de estas transformaciones lineales implica la multiplicación de los vértices de la escena por una matriz de 4x4 cambiando por lo tanto el sistema de coordenadas de los mismos.

La primera transformación se la denomina **transformación del modelo** (o transformación geométrica) y es utilizada para poder trasladar, rotar y dar escala a los objetos, entre otras operaciones. En la figura 3.4 se muestran ejemplos de transformaciones del modelo.

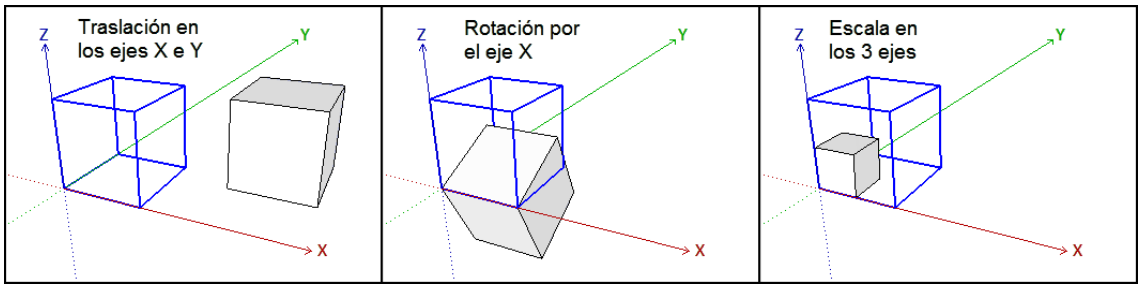

Figura 3.4: Ejemplos de transformaciones de un modelo.

Luego de aplicar la transformación del modelo, el objeto queda en coordenadas o espacio de la escena (Ver figura 3.5). Es común asociar diferentes transformaciones del modelo a un mismo objeto, y utilizar cada transformación para generar una copia diferente del mismo en la escena. Estas copias o "instancias" (como se las denomina), permiten generar el objeto en diferentes posiciones, orientaciones y tamaños sin la necesidad de replicar su geometría. De esta manera, se dice que este espacio es "único" en el sentido en que todos los objetos de la escena residen en el mismo sistema de coordenadas.

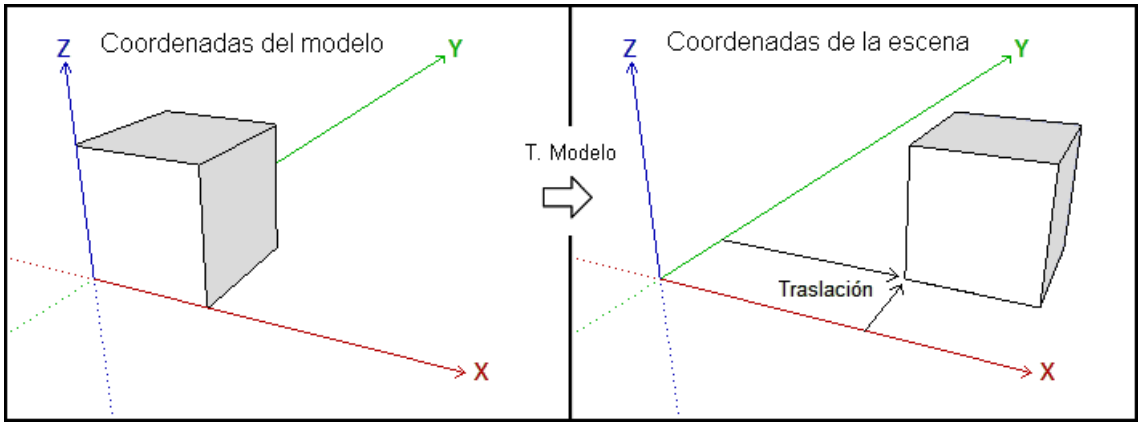

Figura 3.5: Cambio de coordenadas del modelo a coordenadas de la escena por medio de una transformación de traslación.

Luego, se aplica la **transformación de vista** dejando a los objetos en coordenadas o

espacio de la cámara. Los objetos que deben ser renderizados son aquellos que se encuentran dentro del volumen de visualización de la cámara o *view frustum*, por lo que esta operación transforma los vértices de manera tal que el origen de coordenadas de cada objeto, pasa a ser ahora la posición de la cámara (Ver figura 3.6). Además esta transformación deja la dirección de la cámara hacia el eje Z, el eje Y hacia arriba y el eje X hacia la derecha, que facilitará la operación de Rasterización (sección 3.1.5). Esta transformación es en síntesis la implementación de la cámara virtual de la escena. Se pueden tener tantas cámaras o puntos de vista de la escena como se quiera, simplemente se debe reemplazar la matriz de esta transformación por otra.

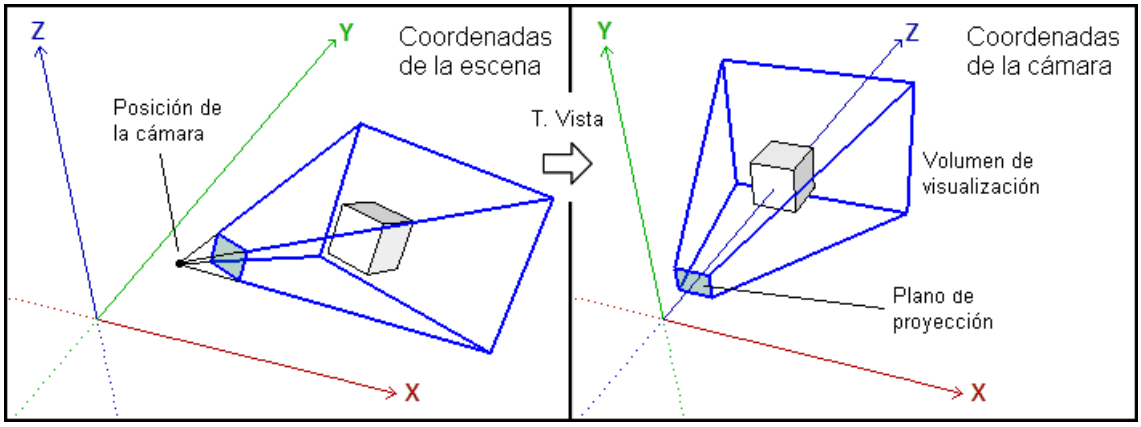

Figura 3.6: Cambio de coordenadas de la escena a coordenadas de la cámara.

Finalmente, se aplica la **transformación de proyección**. Los dos tipos de proyecciones más utilizados son la proyección ortogonal (las líneas paralelas se mantienen paralelas luego de la transformación), y la proyección en perspectiva en la que las líneas paralelas convergen en el horizonte imitando el modo en que percibimos el tamaño de los objetos. Ambos tipos de proyecciones, transforman el *frustum* en un "cubo unitario" con sus vértices extremos en (-1, -1, -1) y (1, 1, 1), o volumen de visualización canónico (Ver figura 3.7). Luego de aplicada esta transformación, los objetos quedan en un sistemas de coordenadas normalizadas para el dispositivo que facilitará la operación de Recorte (sección 3.1.4).

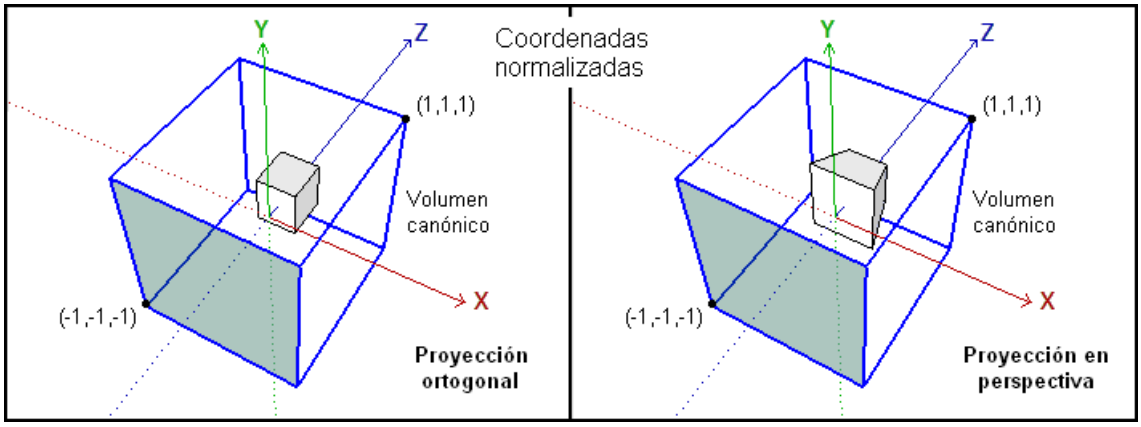

Figura 3.7: La proyección ortogonal y la proyección en perspectiva.

También, vale mencionar que estas transformaciones se podrían aplicar en un mismo paso mediante una matriz única de transformación. Esta matriz sería calculada mediante la multiplicación de las tres matrices (modelo, vista y proyección), ya que el resultado matemático es el mismo que aplicarlas por separado (siempre que la multiplicación se realice en el orden correcto).

3.1.4 Recorte y Mapeo a pantalla

En esta etapa, las primitivas ya fueron transformadas, y son aquellas que se encuentran totalmente fuera del volumen de visualización canónico, las que deben ser descartadas del pipeline ya que no serán renderizadas. Son las primitivas que yacen total o parcialmente dentro del volumen de visualización, las que serán pasadas a la siguiente etapa de Rasterización y Mezcla. Además en esta etapa, las coordenadas de las primitivas son traducidas a la ubicación final del dispositivo donde se visualizará la escena. Estas operaciones son llamadas Recorte y Mapeo a Pantalla, y se muestran en la figura 3.8.

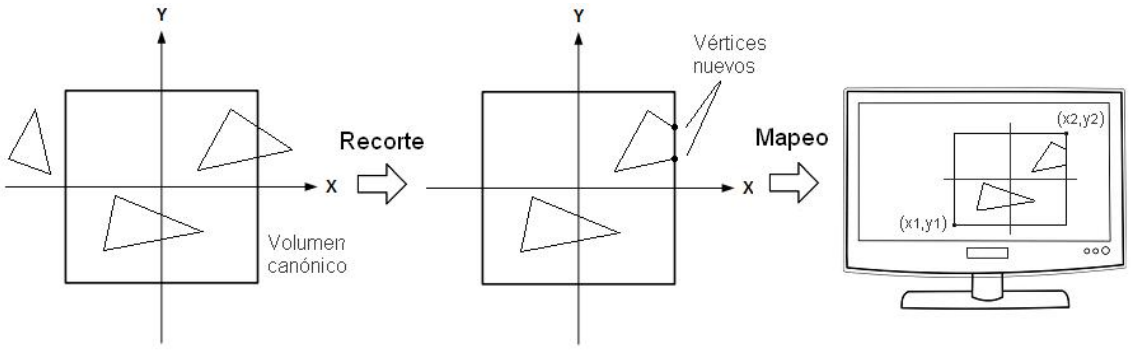

Figura 3.8: Proceso de Recorte y Mapeo a Pantalla

A las primitivas parcialmente dentro del volumen de visualización canónico se les aplica una operación de recorte. Esto es, los vértices que se encuentran fuera del volumen de visualización son reemplazados por nuevos vértices situados en la

intersección entre las aristas de las primitivas y el volumen de visualización.

Luego del recorte, las coordenadas de cada primitiva son mapeadas a coordenadas de pantalla según la posición y tamaño de la ventana o *view port*, donde se visualizará la escena. La operación de Mapeo a Pantalla consiste en una traslación y escalado según las coordenadas (x_1, y_1) y (x_2, y_2) del *view port.* Sólo las coordenadas X e Y de las primitivas son afectadas por esta operación, sin ser afectada la coordenada Z (profundidad). Las coordenadas de pantalla X e Y junto a la coordenada Z canónica son pasadas a la siguiente etapa.

3.1.5 Rasterización y Mezcla

El propósito de esta etapa es dar color a los píxeles de la pantalla que se encuentran cubiertos por las primitivas de los objetos. Para esta tarea se utilizan buffers bidimensionales de tamaño idéntico al de la ventana. Al conjunto de estos buffers se lo denomina *frame-buffer*.

El primero de estos buffers es el *color- buffer*, y es el que almacenará los píxeles finales que luego se mostrarán en pantalla. Para calcular el color de cada píxel se llevan a cabo principalmente dos operaciones: la rasterización y la mezcla.

La rasterización (o escaneo de conversión) es una conversión progresiva entre los datos asociados a los vértices de una primitiva. Esta conversión consiste en realizar una interpolación lineal de las coordenadas 2D (de pantalla) y del valor z asociado (profundidad) a cada vértice. Este proceso genera estructuras temporales denominadas fragmentos. Cada fragmento contiene el color candidato, y la profundidad en el punto (o píxel) que le correspondería ocupar en el *color-buffer* (Ver figura 3.9). El color del fragmento se obtiene por el proceso de interpolación, ya sea de los atributos de color de los vértices o mediante el muestreo de coordenadas de textura.

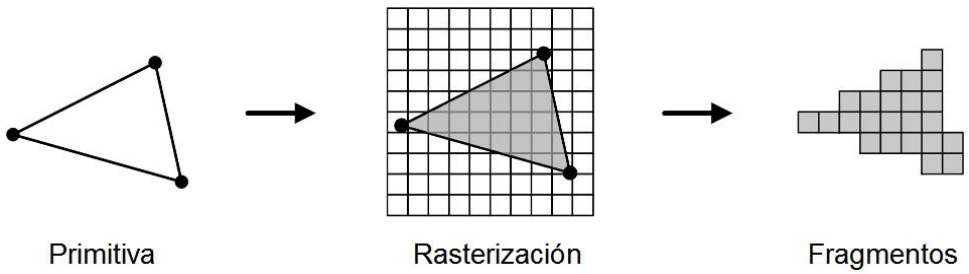

Figura 3.9: Rasterización de un triángulo.

Luego, la operación de mezcla resuelve la visibilidad de cada fragmento, es decir, decide como un fragmento afectará el color final de un píxel. Más de un fragmento de distintas primitivas pueden corresponderse a un mismo píxel de pantalla, por lo que se debe decidir cual de ellos finalmente utilizar. Para esto, un nuevo buffer llamado *z-buffer* es utilizado. Para cada píxel, este buffer almacena la componente z

(profundidad) del último fragmento colocado en el *color-buffer*. Entonces, sólo si el nuevo fragmento posee su valor z menor (más cerca del observador) al almacenado en el *z-buffer*, se coloca el color del fragmento en el *color-buffer* y se actualiza el *z-buffer* con el valor z del fragmento. Si el valor z del fragmento es mayor al correspondiente del *z-buffer* en ese punto, el fragmento es descartado y ninguno de los buffers es afectado. De esta manera, el color del fragmento que terminará en el buffer de color será aquel que está más cerca del observador.

Finalmente, el contenido del *color-buffer* se muestra en pantalla. Para esto suelen utilizarse técnicas que emplean más de un buffer de color. El proceso de generación de imagen es llevado a cabo sobre un buffer en segundo plano. Cuando la imagen ha terminado de generarse, puede ser mostrada en pantalla y un buffer libre puede ser empleado para comenzar a generar un nuevo cuadro. De esta manera es posible evitar efectos indeseados de parpadeo de imagen, permitiendo que el usuario siempre vea el contenido completo de cada cuadro.

3.2 LA GPU

3.2.1 Qué es la GPU

En la década de 1980, compañías como SGI (*Silicon Graphics Incorporated*) y Evans & Sutherland ya producían hardware especializado para procesamiento gráfico, pero debido a su alto costo, no consiguieron ser exitosos en el mercado de consumo masivo.

Los primeros chips de aceleración gráfica en destinarse al mercado masivo de PCs fueron lanzados a mediados de 1990. Estos chips se incluyeron en placas gráficas como la “3dfx Voodoo” o la “RIVA TNT”, y proveían sólo capacidades de rasterización y mezcla.

En 1999 fue lanzada la “NVIDIA GeForce 256”, la primera placa gráfica para PC capaz de realizar el procesamiento completo de vértices directamente por hardware. En ese momento, NVIDIA acuñó el término **GPU** (*Graphics Processing Unit*) para diferenciarse comercialmente de los chips anteriores y, dado que la descripción "Controlador VGA” -como se denominaba el hardware gráfico hasta ese momento- ya no era una descripción adecuada del nuevo hardware para gráficos 3D, el uso del término **GPU** se generalizó y fue adoptado por la industria.

En los últimos 15 años el hardware de la GPU evolucionó constantemente, experimentando una gran variación y mejora de sus capacidades. Nuevas funcionalidades y potencia de cómputo fueron agregadas. La GPU pasó gradualmente de implementar un pipeline de funcionalidad fija (no modificable), a una implementación altamente programable.

Cabe destacar que hoy en día la GPU también puede ser utilizada para un propósito distinto al de renderización de escenas. Debido a su alta capacidad de cómputo y

paralelismo, la GPU también puede ser utilizada para procesamiento de propósito general. Esta rama de estudio se denomina GPGPU (*General-Purpose computation on GPU*), y se encuentra fuera del alcance de este trabajo.

3.2.2 Interfaces de programación

Desde el surgimiento de las primeras GPUs, fueron creadas nuevas librerías con el fin de asistir en el desarrollo de aplicaciones 3D y facilitar el acceso al hardware. Las dos librerías o APIs (*Application Programming Interface*) más populares son OpenGL y DirectX. Ambas librerías permiten modelar los objetos de la escena mediante la definición de mallas de primitivas y brindan una abstracción común en el acceso a las distintas GPUs, permitiendo al programador abstraerse del hardware subyacente.

Tanto OpenGL como DirectX, evolucionaron junto a la GPU en un doble juego, por un lado haciendo accesibles las nuevas funcionalidades disponibles en el hardware, pero muchas veces también impulsando el desarrollo de nuevas técnicas o algoritmos para su implementación en el hardware de la GPU. A continuación de hace una breve descripción de las mismas:

- **OpenGL** (*Open Graphics Library*) es una librería multiplataforma para el desarrollo de aplicaciones gráficas 3D. Fue lanzada como estándar abierto en 1992 por SGI (*Silicon Graphics Incorporated*). OpenGL fue desarrollada a partir de la API propietaria IrisGL, de uso exclusivo en hardware SGI. Para mantener la nueva API, se conformó un consorcio de compañías llamado OpenGL ARB (*OpenGL Architecture Review Board*). Este consorcio propuso y aprobó cambios a las especificaciones, liberó actualizaciones y nuevas versiones hasta el 2006. En ese momento, el control de OpenGL fue transferido a Khronos Group, un consorcio sin fines de lucro dedicado a la creación de estándares abiertos, quien lo mantiene hasta la fecha. Hoy en día, pueden encontrarse implementaciones del estándar de OpenGL en prácticamente todos de los sistemas operativos, como también en la mayoría de las plataformas móviles.
- **DirectX** es una colección de librerías destinadas al desarrollo de aplicaciones multimedia. Fue diseñada por Microsoft para utilizarse en su propio sistema operativo Windows. Dentro de esta colección, se encuentra Direct3D, la interfaz de programación que provee funciones para la renderización de gráficos 3D y su aceleración por hardware. La primera versión de DirectX fue lanzado en 1995 junto al sistema operativo Windows 95. Desde entonces, nuevas versiones de esta colección o paquete de APIs fueron publicadas hasta la actualidad.

Por último cabe mencionar que ambas librerías abstraen al programador de la implementación concreta del pipeline, ya sea por software o por hardware, brindando una visión consistente del mismo. Si bien el principal propósito de estas librerías es permitir el acceso a la GPU con el objetivo de lograr un proceso de renderización acelerado por hardware, también implementan el pipeline. De esta manera no sólo se brinda una visión genérica del mismo, sino que también es posible utilizar

funcionalidades implementadas por software cuando el hardware subyacente no lo soporta.

3.2.3 Shaders

Las capacidades de cómputo de la GPU se incrementaron de manera constante en un proceso evolutivo que continua en la actualidad. En este proceso, se desarrollaron nuevas arquitecturas que hicieron que este hardware fuera cada vez más poderoso. La GPU evolucionó desde un pipeline de funcionalidad fija (no modificables), hacia una implementación cada vez más flexible. Surgieron nuevos modos de controlar el proceso de renderización, y las capacidades del pipeline se extendieron significativamente.

El cambio más notable en esta evolución, es la incorporación de etapas programables al pipeline de la GPU, esto es, etapas que son capaces de ejecutar programas diseñados especialmente para la renderización de objetos geométricos. A estos programas se los denomina *shaders*, un nombre heredado de los primeros pasos en la programabilidad de la GPU, donde los *shaders* eran utilizados para modificar el modo en que la luz afectaba a los objetos. Luego, a medida que más etapas fueron agregadas, el nombre *shader* fue adoptado para referirse a todas las etapas programables del pipeline programable.

En un principio los *shaders* eran escritos en lenguaje ensamblador, pero el largo y complejidad creciente de los mismos, y su incompatibilidad ante GPUs de distintos fabricantes, hicieron que el desarrollo de *shaders* se volviera muy complicado.

Por las razones mencionadas diversos lenguajes de alto nivel fueron creados, definiendo un nuevo esquema de trabajo: un programa escrito en un lenguaje de alto nivel es compilado a un lenguaje ensamblador independiente de la plataforma o IL (*Intermediate Languaje*). Luego los drivers de las distintas GPUs son los encargados de hacer la traducción al conjunto de instrucciones específico soportado en cada unidad de procesamiento gráfico (GPU) particular.

Los lenguajes de alto nivel más populares para la programación de *shaders* son HLSL, GLSL, y Cg:

- **HLSL** (*High Level Shading Language*) [7] fue desarrollado por Microsoft para utilizarse junto a Direct3D. Este lenguaje es muy similar a Cg ya que ambos fueron desarrollados juntos.
- **GLSL** (*OpenGL Shading Language*) [30] fue creado por la *OpenGL Architecture Review Board* para utilizarse junto a las librerías de OpenGL. Su desarrollo estuvo muy influenciado por la sintaxis y filosofía del lenguaje de programación C.
- **Cg** (*C for Graphics*) [8] fue desarrollado por NVIDIA con la colaboración de Microsoft. Hasta 2012, NVIDIA proveía una plataforma completa de programación de *shaders* mediante un conjunto de herramientas denominado “Cg Toolkit”, el cual permitía ser utilizado junto a las librerías de DirectX y OpenGL. Hoy en día ya

no recibe soporte por parte de NVIDIA.

Cada nuevo conjunto de capacidades que fue incorporado a la GPU, se hizo corresponder con un modelo distinto de programación, o *Shader Model*. Los *Shader Models* (SM) fueron estructurados en versiones. Cada versión incorporó nuevas capacidades de programación y la estructura del pipeline programable varió en consecuencia.

Sintéticamente, los primeros modelos de programación fueron lanzados en la primera mitad de la década del 2000. El **SM 1.1** se lanzó en 2001 con la NVIDIA Geforce 3 que incluía *shaders* de vértice y de píxel, aunque con capacidades muy limitadas para los mismos. El **SM 1.0** nunca fue implementado. GPUs soportando **SM 2** fueron lanzadas en 2002 soportando operaciones de punto flotante e instrucciones condicionales como novedad para la programación de *shaders*. El **SM 3** fue introducido en 2004 incrementando notablemente las capacidades de cómputo de la GPU. Además este SM mejoró el acceso a texturas incorporando VTF (*Vertex Texture Fetch*). Esta característica permitió el muestreo de texturas desde los *shaders* de vértice posibilitando un nuevo conjunto de técnicas como ser el desplazamiento de vértices a partir de la información almacenada en una textura [11].

En 2006 se produjo un gran paso en la programabilidad de GPUs al introducirse el **SM 4**. Este modelo agregó a los ya existentes *shaders* de vértice y de píxel, una nueva etapa programable denominada *Geometry shader*. Esta etapa es capaz de procesar primitivas, como también agregar o emitir nuevas primitivas o descartar las existentes. Las primitivas procesadas pueden ser enviadas opcionalmente a un buffer en memoria accesible desde la CPU mediante una nueva característica denominada *Stream Output*, lo que permitió por primera vez varios modos de uso o flujos posibles dentro del pipeline gráfico. Además, en el **SM 4** se extendieron las características de programación y se generalizaron los recursos disponibles para todas las etapas programables. Esto es, los *shaders* de vértices, píxel y geometría comparten un lenguaje común, o *Common Shader Core*. Este modelo unificado de *shaders* presentó un conjunto de instrucciones consistente para todas las etapas programables, y permitió acceder prácticamente a las mismas funcionalidades desde cualquier tipo de *shader* (acceso a texturas, buffers de datos y la ejecución del mismo conjunto de instrucciones aritméticas).

SM 5 fue lanzado en 2010 y es el modelo de programación vigente en la actualidad. La nueva característica que incorpora es la capacidad de teselado de primitivas, la cual tiene especial importancia para el desarrollo de técnicas multirresolución. Por estas razones, se dedicó la siguiente sección a este modelo.

3.2.4 Shader Model 5

El SM 5 es el último paso en la evolución del pipeline de la GPU. Sobre las características de la versión 4, esta versión introdujo un sistema flexible de teselado de primitivas. El teselado tiene como propósito dividir primitivas en pedazos más

pequeños obteniéndose una superficie de mayor detalle. Esto significa que superficies de baja resolución (poco detalle) pueden ser convertidas en superficies de mayor resolución.

SM 5 incorpora un nuevo tipo de primitiva para implementar el teselado. Estas primitivas son denominadas parches (*patch*) y consisten en un conjunto de puntos de control que definen los vértices del polígono a subdividir. Existes tres tipos de parches: las líneas, los triángulos, y los cuadriláteros.

El teselado de parches es llevado a cabo mediante tres nuevas etapas: *Hull Shader*, *Tessellator* y *Domain Shader*. Estas nuevas etapas para el teselado de geometría, trabajan en conjunto y son opcionales, es decir que es posible activar o desactivar su utilización. De estas etapas, el *Hull shader* y el *Domain shader* son programables, mientras el *Tessellator* realiza operaciones de funcionalidad fija.

El número exacto de triángulos generados depende de los factores de teselado utilizados. Un factor de teselado es un valor numérico que puede ser modificado dinámicamente en los *shader*. En el hardware actual, los factores de teselado pueden tener un valor máximo de 64. Teniendo en cuenta que cada quad puede ser subdividido en una grilla de 64x64 quads más pequeños, y como cada uno consiste en dos triángulos, la unidad de teselado puede generar un máximo de 8192 triángulos por quad.

Los factores de teselado pueden no ser valores enteros. Los factores no enteros o fraccionales, proveen un mecanismo de refinamiento transicional entre dos factores enteros. Por ejemplo un valor de 2.5 posiciona los nuevos vértices producidos por el teselado entre las posiciones enteras 2 y 3, lo que permite variar el nivel de refinamiento de manera suave y sin producir cambios abruptos en la topología de la malla. La figura 3.10 muestra distintos patrones obtenidos al teselar cuadriláteros por distintos factores de teselado.

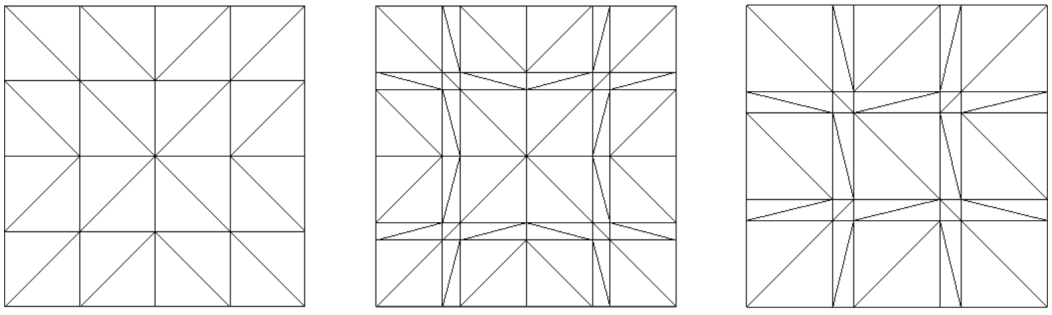

Figura 3.10: Teselados de un cuadrilátero aplicando distintos factores de teselado (enteros y fraccionales).

3.3 El pipeline gráfico de DirectX 11

La mayoría de las aplicaciones gráficas 3D son desarrolladas sobre el sistema operativo Windows utilizando la librería DirectX. Por razones comerciales las nuevas versiones de esta librería se destacan por brindar mejores funcionalidades. Esta librería marca un liderazgo en el desarrollo de software y en muchos casos también ha impulsado la evolución o estandarización del hardware gráfico. Asimismo, la versión 11 de DirectX fue concebida especialmente para soportar el Shader Model 5 [36]. De esta manera y por los motivos mencionados, el software desarrollado en este trabajo fue implementado utilizando la librería Direct3D del paquete DirectX en su versión 11.

En la figura 3.11 se muestra la estructura del pipeline gráfico de DirectX 11. Las cajas redondeadas representan las etapas programables. El resto de las etapas corresponden a etapas de funcionalidad fija, esto significa que no es posible modificar su comportamiento mediante un programa o *shader*. Las nuevas etapas correspondientes a la funcionalidad de teselado se han resaltado en color verde. De las etapas programables, sólo *Vertex shader* y *Pixel shader* son obligatorias, por lo que es necesario proveer el correspondiente *shader*. Las demás etapas programables son optativas.

A continuación se describirán cada una de las etapas del pileline de DirectX 11, describiendo brevemente su modo de utilización y la forma de configuración del teselado.

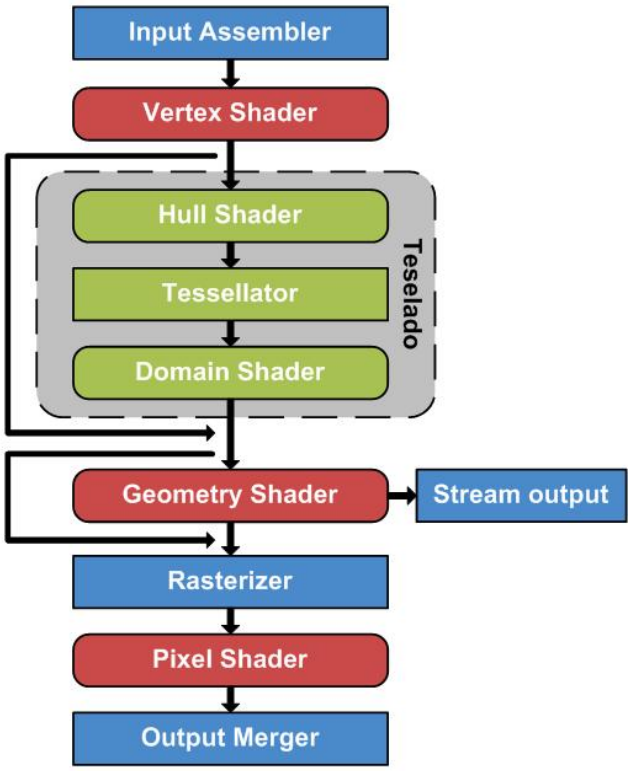

Figura 3.11: El pipeline gráfico de DirectX 11.

3.3.1 Input Assembler

El propósito de esta etapa es preparar la información geométrica en un formato consistente para su procesamiento por el resto del pipeline. Para esto, toma los datos disponibles en los buffer de entrada y conforma los datos de vértices que serán la entrada de la próxima etapa del pipeline (*Vertex shader*).

Existen dos tipos de buffers, buffer de vértices (*Vertex buffer*) y buffer de índices (*Index buffer*). Los datos del buffer de vértices consisten normalmente en la posición, coordenadas de textura, normal, etc.; pudiendose incluir cualquier otro atributo que se desee. El buffer de índices es de uso optativo. Cada índice de un buffer de índices apunta a un vértice del buffer de vértices. La utilización de este buffer permite emplear menor memoria en la representación de la malla ya que es posible que dos o más índices apunten a un mismo vértice, reutilizando un vértice en varias primitivas.

Input Assembler también conforma los datos de primitivas. Las primitivas soportadas son puntos, líneas y triángulos. Para el caso del teselado, las primitivas son denominadas parches. Pueden existir parches de entre 1 y 32 vértices. Los parches de 4 vértices son denominados *quad*.

En el proceso de ensamblado de primitivas también son generados los parámetros identificatorios o IDs de cada elemento geométrico (vértices y primitivas). Estos IDs permiten a los distintos *shaders* del *pipeline*, identificar unívocamente cada vértice y la primitiva a la que pertenecen.

3.3.2 Vertex Shader

Esta etapa realiza operaciones sobre cada vértice de la salida del *Input assembler*. Un *vertex shader* se ejecuta una vez por cada vértice, produciendo un vértice de salida para cada uno de entrada. En el caso de utilizar teselado, la salida se denomina puntos de control en lugar de vértices.

Un *vertex shader* permite manipular o modificar, crear, o ignorar los valores asociados a cada vértice (o punto de control) como ser color, normal coordenadas de textura, etc. De esta manera pueden realizarse efectos como animación o deformación de mallas, muestreo de texturas, cálculos de iluminación, etc.

Las operaciones de transformación espacial (modelo, vista, y proyección) son responsabilidad del programador y estos programas son el lugar natural para realizarlas. Si se utiliza la funcionalidad de teselado, es posible delegar estas operaciones al *Domain shader*, en cuyo caso se debe crear un programa de vértice “pass-through”, es decir dedicado simplemente a trasladar a su salida los datos de entrada que serán necesarios en las etapas siguientes del pipeline. La próxima etapa puede ser *Hull Shader*, *Geometry Shader*, o *Rasterizer*, dependiendo de la técnica o modo de utilización del pipeline.

3.3.3 Teselado

Las etapas de *Hull shader*, *Tessellator*, y *Domain shader*, trabajan juntas para implementar la funcionalidad de teselado de geometría. De estas etapas, *Hull shader* y *Domain shader* son programables, mientras que *Tessellator* realiza operaciones de funcionalidad fija. En conjunto, estas etapas son optativas en el pipeline. La figura 3.12 esquematiza el flujo de datos entre las etapas mencionadas, las cuales serán descriptas a continuación.

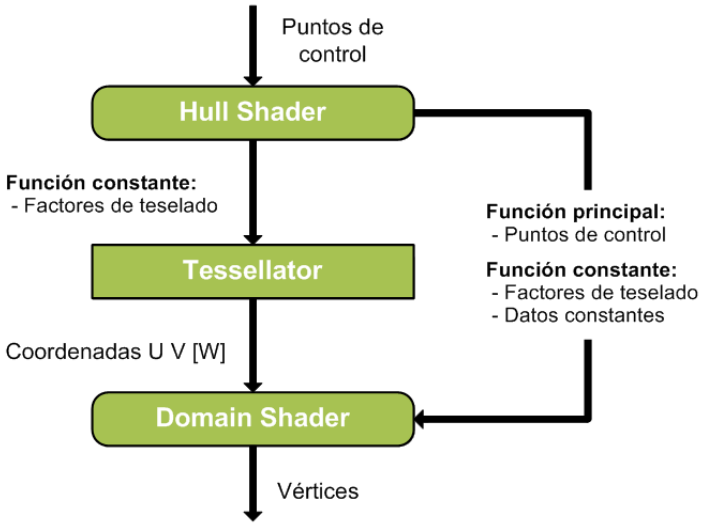

Figura 3.12: Etapas de teselado.

Hull Shader

Esta etapa recibe puntos de control del *Vertex shader*, y se encarga de realizar las operaciones necesarias que preparan las primitivas para el teselado. El *Hull shader* se divide en dos partes, la funcion constante y la función principal, teniendo ambas visibilidad de la salida del *Vertex shader*.

La función constante del *Hull shader* se ejecuta una vez por parche. Su objetivo es el de calcular los factores de teselado que determinarán cuanto subdividir la primitiva. Los factores de teselado pueden poseer cualquier valor entre 0 y 64. Además de los factores normales o interiores, existen los factores de teselado exteriores que definen el número de cortes que se realizará exclusivamente sobre los bordes de la primitiva, permitiendo cambiar la resolución de forma adaptativa. El número de factores de teselado a calcular depende del tipo de dominio utilizado (triángulos, cuadriláteros o líneas). En el caso de los triángulos 4 factores deben ser calculados (3 exteriores, 1 interior). Los cuadriláteros requieren 6 factores (4 exteriores, 2 interiores). Las líneas requieren 2 factores (detalle y densidad). Mediante la función constante del *Hull Shader* también es posible generar información adicional (Datos constantes) referente a cada parche. Esta información puede ser útil para operaciones adicionales en el *Domain shader*.

La función principal del *Hull shader* es ejecutada una vez por punto de control y en cada ejecución tiene visibilidad de todos los puntos del parche. El número de puntos no deben coincidir necesariamente con la topología establecida en el *Input assembler*. Esta función puede variar el número de puntos de control que serán generados. Los puntos de control generados son enviados al *Domain shader*.

Tessellator

Tessellator es una etapa de funcionalidad fija. Su función es subdividir los parches conformados por los puntos de control en muchos polígonos más pequeños. Esta etapa no tiene acceso a los puntos de control. El proceso de teselado depende únicamente de los factores de teselado producidos por el *Hull shader*, y de una serie de atributos atributos definidos en tiempo de compilación (estáticos). Estos atributos son declarados junto a la función principal del *Hull shader*. Los atributos son:

- **domain(x)** especifica el dominio de teselado a utilizar. El parámetro x debe tomar uno de los siguientes valores: `triangle`, `quad`, o `isoline`.
- **partitioning(x)** establece el modo de interpretar los factores de teselado. Los posibles valores de x son: `integer`, `pow2`, `fractional_even`, y `fractiona_odd`. `integer` y `pow2` definen un nivel de refinamiento discreto, interpretando los factores de teselado como números enteros. `pow2` además redondea los valores al número más cercano de la serie 2^n. `fractional_even` y `fractional_odd` interpretan los factores de teselado como números de punto flotante. `fractional_even` y `fractional_odd` difieren en el número de subdivisiones realizadas. `fractional_even` sólo permite un número de subdivisiones par, por lo que redondea el valor del factor al número par más cercano, mientras `fractional_odd` lo hace hacia los números impares.
- **outputtopology(x)** define el tipo de primitiva de salida. Puede elegirse entre `point`, `line`, `triangle_cw` y `triangle_ccw`. Estos dos últimos implican una triangulación con sentido horario y antihorario respectivamente (*clockwise* y *counterclockwise*).
- **outputcontrolpoints(x)** especifica el número de puntos de control de salida y por tanto el número de veces que *Doman shader* será ejecutado.
- **maxtessfactor(x)** establece el valor máximo que pueden poseer los factores de teselado.
- **patchconstantfunc ("function_name")** especifica el nombre de la función constante a utilizar en la presente configuración.

El dominio es teselado en un sistema de coordenadas normalizado (valores entre 0 y 1). La salida de *Tessellator* es un conjunto de pesos que depende del tipo de dominio. Para `triangle` la salida consiste en coordenadas baricéntricas (U, V, W), mientras que para `quad` e `isoline` la salida consiste en coordenadas de textura (U, V). Estas coordenadas describen una posición -donde se llevó a cabo la subdivisión o corte- relativa a la primitiva. Las coordenadas son enviadas al *Domain shader* quien deberá construir vértices a partir de esta información.

Domain Shader

Al igual que *Vertex shader*, esta etapa tiene el propósito de produdir vértices. En este caso, debe producir los vértices que conforman la salida del proceso de teselado. Un *Domain shader* se ejecuta una vez por cada vértice a generar.

Como entrada recibe del *Hull shader* los puntos de control y los factores de teselado, y de Tessellator la coordenada UV[W] del vértice a generar. Con esta información debe calcular la posición del vértice, por ejemplo utilizando la coordenada UV[W] como peso de una interpolación lineal entre los puntos de control. La nueva posición del vértice debe ser transformada a espacio de proyección antes de formar parte de la salida hacia la próxima etapa (*Geometry Shader*, o *Rasterizer*).

3.3.4 Geometry Shader

Geometry shader opera sobre primitivas. Dada una primitiva de entrada, es posible descartarla, modificarla, o incluso emitir primitivas nuevas. Además como se tiene acceso a los vértices de la primitiva de entrada, estos programas permiten agregar, modificar, o descartar los atributos asociados a cada vértice. Al igual que las etapas de teselado, esta etapa es optativa en el pipeline.

Stream Output

Esta etapa esta diseñada para enviar optativamente los datos de las primitivas de salida del *Geometry shader* fuera del pipeline hacia la memoria de la GPU. Los datos pueden ser enviados fuera del *pipeline* y/o ser pasados al rasterizador. Una vez enviados a memoria de GPU, los datos pueden ser recirculados como nuevos datos de entrada del *Input Assembler*, o leídos desde la CPU. Esta etapa es optativa y sólo puede estar activa si también lo está *Geometry shader*.

3.3.5 Rasterizer

Rasterizer es una etapa de funcionalidad fija. Su propósito es el de generar los fragmentos correspondientes a los píxeles cubiertos por cada primitiva. Para esto, transforma los vértices de las primitivas a coordenadas en pantalla, realiza la operación de recorte, y finalmente interpola los atributos de los vértices de cada primitiva para generar los fragmentos que son enviados a la próxima etapa.

3.3.6 Pixel Shader

El propósito de esta etapa es dar color -o sombrear- cada fragmento de entrada. Un *pixel shader* puede asignar un color fijo, acceder a texturas a partir de coordenadas interpoladas por el rasterizado, o incluso utilizar distintos modelos de iluminación. Una

vez sombreado el fragmento es pasado a la próxima etapa.

3.3.7 Output Merger

Esta etapa es responsable de combinar los fragmentos para conformar la imagen o cuadro final. Los fragmentos pueden descartarse o combinarse con otros, dependiendo principalmente de sus atributos de profundidad y color. El resultado queda reflejado en un buffer de color denominado *render-target*, el cual es utilizado para mostrar en pantalla o para accederse desde la CPU.

4 ALGORITMOS DE VISUALIZACIÓN DE TERRENOS

En las últimas dos décadas, distintos algoritmos multirresolución para visualización de terrenos han sido desarrollados. Como indica Ulrich [35], estos algoritmos pueden ser clasificados claramente en dos grupos: Los algoritmos Pre-GPU, que incluyen a los algoritmos que pueden denominarse "clásicos", y los algoritmos Post-GPU, que corresponden a los algoritmos más modernos.

Con el objetivo de brindar una visión más completa de la visualización de terrenos multirresolución, en este capítulo se presentan aquellos algoritmos más relevantes de ambos grupos, describiendo sus principales características y modos de operación. Además, el análisis de estos algoritmos brindará las bases para el desarrollo de un nuevo algoritmo (capítulo 5).

El capítulo comienza describiendo en la sección 4.1, dos de las técnicas comunes más utilizadas por los algoritmos presentados en las secciones subsiguientes, de manera de describirlas previamente a su utilización.

La sección 4.2 se dedica a los algoritmos Pre-GPU, y se rotula cada subsección con el título del *paper* que describe cada técnica. La sección 4.3 es dedicada a los algoritmos Post-GPU. Coincidentemente para este grupo, todos los autores han dado un nombre particular a cada técnica, con los cuales se han rotulado las respectivas subsecciones.

4.1 Técnicas comunes

4.1.1 View frustum culling

La performance de renderización puede ser mejorada utilizando la técnica denominada *View frustum culling*. Esta técnica se basa en organizar la geometría de la escena de manera de descartar de forma temprana -no enviar al *pipeline*- grandes cantidades de geometría, evitando iniciar el procesamiento del *pipeline* con geometría que no es visible.

La técnica es llevada a cabo en la aplicación, y consiste en hacer un test de visibilidad respecto del *frustum* de la cámara. A diferencia de la operación de recorte dentro del *pipeline* que opera respecto de cada primitiva, el chequeo es llevado a cabo respecto de un volumen envolvente que agrupa la malla completa de uno o varios objetos. Luego, si el volumen envolvente “no es visto” por el *frustum* de la cámara, ninguna primitiva de las mallas dentro del volumen lo será.

El test de visibilidad consiste en realizar la intersección espacial entre un volúmen

envolvente y el *frustum* de la cámara. Los tipos más comunes de volúmenes envolventes utilizados son la esfera, las cajas alineadas a los ejes cartesianos (*Axis Aligned Bounding Box, AABB*) y las cajas orientadas (*Oriented Bounding Box, OBB*), como indica Akenine-Möller et al. [1], mientras que la forma más común de representar el *frustum* es mediante 6 planos en el espacio que limitan el espacio de visualización de la cámara. Luego de realizado el test de visibilidad, se determina si los objetos dentro del volumen son visibles o no. De esta manera, la geometría no visible puede ser descartada de forma temprana del proceso de visualización reduciendo significativamente el número de primitivas enviadas hacia el *pipeline* gráfico y por tanto mejorando la performance.

Además, es posible organizar los volúmenes envolventes dentro de una jerarquía, lo que conlleva otro beneficio. Si el volumen envolvente de un nodo padre de la jerarquía no es visible por el *frustum*, entonces sus hijos tampoco son visibles, ya que los mismos se encuentran completamente contenidos dentro del volumen del nodo padre. El hecho de organizar los volúmenes en una jerarquía, lleva a una reducción aún mayor de geometría no visible como así también un incremento en la performance ya que menos volúmenes envolventes deben ser testeados por su intersección con el frustum de la cámara.

4.1.2 Geomorphing

Como se ha mencionado en la sección 2.4.4, el efecto visual llamado *popping* se aprecia como un salto o cambio repentino en la geometría de un objeto, que se produce al intercambiar dos representaciones de distinto nivel de detalle. La técnica *Geomorphing* (*Geometrical morphing*) consiste en realizar una transición suavizada entre dos representaciones distintas, para mitigar o eliminar la percepción de saltos o *pops*.

Geomorphing se basa en trasladar vértices desde su posición original hacia otra que se hace corresponder en otro nivel de detalle que se desea alcanzar. Así por ejemplo cuando se intercambia un modelo complejo por uno más simple, los vértices del modelo complejo son interpolados entre sus posiciones originales y aquellas de la versión simplificada, que en general corresponden a las posiciones más próximas en el mallado de menor resolución (ver figura 4.1). Cuando la transición se completa, ya es posible utilizar el modelo de menor nivel de detalle para visualizar el objeto.

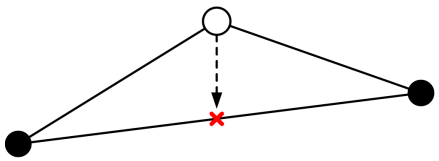

Figura 4.1: Transición hacia una representación de menor resolución. El vértice blanco es interpolado desde su posición hacia la posición marcada en rojo, en cuyo momento es posible representar el modelo solamente utilizando los vértices negros.

Existen dos modos de realizar el *geomorphing* entre dos representaciones de distinta resolución: basando sus transiciones en tiempo, o basándolas en error visual. Las transiciones basadas en tiempo definen una cantidad de tiempo determinada para la duración de la misma. Las transiciones basadas en error visual dependen de la distancia de la cámara al objeto, dando como resultado que el movimiento de vértices es realizado sólo durante el movimiento de la cámara. Las transiciones basadas en error visual son las más utilizadas ya que son menos perceptibles por el usuario.

Como indican Luebke et al. [25], *geomorphing* es una técnica compleja de emplear, pero puede proveer un modo muy efectivo de realizar el intercambio entre las representaciones de distinto nivel de detalle de un objeto. Además según indican estos autores, es Hoppe en [18] quien emplea por primera vez esta técnica en terrenos.

4.2 Algoritmos de visualización de terrenos Pre-GPU

Estos algoritmos fueron desarrollados en una etapa previa al surgimiento de la GPU. La falta de una aceleración por hardware, al menos una aceleración completa del *pipeline* gráfico, impedía la posibilidad de procesar grandes cantidades de triángulos. Por tanto, era necesario aprovechar de la mejor manera posible las capacidades de cómputo limitadas. Así, estos algoritmos fueron diseñados para generar triangulaciones que permiten representar la superficie del terreno de manera óptima, disponiendo de aproximaciones con las que se puede obtener una buena calidad de imagen utilizando la menor cantidad posible de triángulos.

Otra característica de estos algoritmos es la capacidad de generar un refinamiento continuo de la superficie del terreno. Esto significa que el terreno es representado variando la cantidad y tamaño de triángulos a medida que el punto de observación se modifica, y sin producir cambios abruptos en el nivel de detalle. Esto produce que los algoritmos requieran mantener estructuras de datos realtivamente complejas para hacer un seguimiento del estado de las triangulaciones.

4.2.1 "Real-Time, Continuous Level of Detail Rendering of Height Fields"

En 1996, Lindstrom et al. publicaron el trabajo denominado "*Real-Time, Continuous Level of Detail Rendering of Height Fields"* [21]. Este trabajo presenta un algoritmo que utiliza datos de elevación en formato de grillas regulares (*heightmap*), y se basa en un proceso que consta de dos etapas de simplificación para lograr una malla de nivel de detalle continuo. La primera etapa simplifica la malla a nivel de bloques. Luego una segunda etapa aplica mayor simplificación a nivel de vértices. Este trabajo es muy importante ya que sienta las bases de muchos otros algoritmos por venir.

El terreno se representa a partir de una grilla regular de vértices, es decir que posee las muestras a intervalos discretos iguales, cuyas dimensiones (número de vértices por lado) deben ser de la forma 2^n+1 con $n \geq 1$.

En primer lugar se realiza la simplificación a nivel de bloques. Comenzando con la malla a resolución completa, el terreno es particionado en bloques cuadrados de 3 vértices de lado. Los bloques se solapan exactamente por una fila y una columna, de manera que los vértices de los lados de cada bloque son compartidos con los bloques vecinos. Luego, se realiza una fusión de bloques en nuevos bloques más grandes conformados cada uno por 4 de los bloques iniciales vecinos, y removiendo las filas y columnas pares de vértices. De esta manera, los nuevos bloques siguen poseyendo 3 vértices de lado, y cubren un área igual a la suma de las áreas de los 4 bloques originales pero a menor resolución (ver figura 4.2). Partiendo de los nuevos bloques, el proceso continúa recursivamente hasta obtenerse el último bloque de menor resolución que cubre todo el terreno. En este proceso se construye un *quadtree*, de manera que los bloques más pequeños y de mayor resolución forman los nodos hoja, y el bloque más grande y de menor resolución corresponde al nodo raíz.

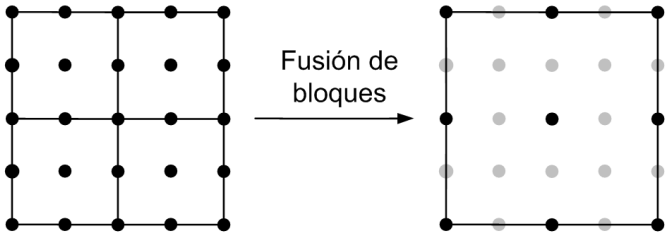

Figura 4.2: Fusión de cuatro bloques vecinos en uno más grande. Al remover los vértices en gris, el bloque resultante posee 3 vértices de lado nuevamente.

Luego, la simplificación a nivel de vértice consiste en un proceso de retriangulación de la geometría de cada bloque por medio de la remoción de vértices. En contraposición al refinamiento que divide triángulos mayores en triángulos más pequeños, al ser éste un proceso de simplificación, pares de triángulos pequeños son fusionados en triángulos mayores. Este proceso comienza con la triangulación de máxima resolución que se ilustra en la figura 4.3-a. Dentro de cada bloque se mantienen las dependencias entre vértices que establecen una estructura tipo *bintree* de triángulos, y en la que se definen qué vértices son utilizados por cada triángulo. Con esta información, pares triángulo/cotriángulo de la estructura son considerados para su fusión. La fusión se realizar mediante la remoción del vértices externo compartido por ambos triángulos del par.

La remoción de un vértice para la fusión de un par de triángulo depende del error que esta acción puede introducir en la imagen final renderizada. Básicamente, una medida cuantitativa es calculada dependiendo de cuan bien el triángulo más grande (resultado de la fusión) puede representar a los dos más pequeños.

En la figura 4.3-b, el triángulo ABC es considerado para fusionarse con su cotriángulo ACD por medio de la remoción del vértice C. El error en espacio de objeto o error de aproximación geométrica δ se muestra en la ecuación 4.1, y corresponde a la distancia vertical entre C y el centro de la hipotenusa del nuevo triángulo de menor resolución

ABD que se formaría si se remueve C.

$$\delta = \left| C - \frac{D+B}{2} \right| \tag{4.1}$$

Luego, el error δ es proyectado en espacio de pantalla para determinar si la fusión se puede realizar. Para esto, se lo compara con un umbral definido por usuario. Si el error proyectado es menor que el umbral, el vértice puede ser removido. Este proceso es recursivo, de manera que los triángulos formados de mayor tamaño son nuevamente evaluados para fusionarse, y continúa hasta que ya no sea posible remover más vértices en el bloque, ya sea porque la remoción de vértices superaría el error tolerado o porque se ha alcanzado la triangulación de menor resolución posible para un bloque.

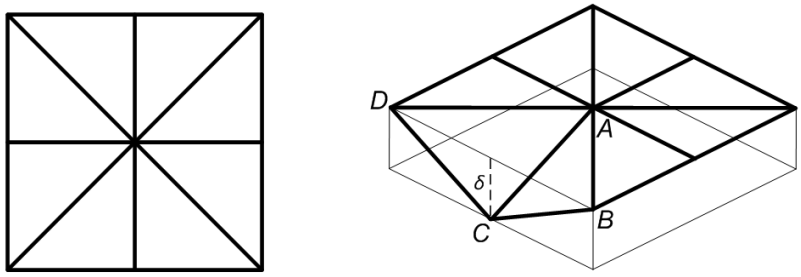

Figura 4.3: En (a) un bloque triangulado a máxima resolución en la que se indica los pares de triángulo/cotriángulo a sufionar. En (b) la remoción del vértice C.

La selección del nivel de detalle se realiza en tiempo de ejecución. La simplificación a nivel de bloques descripta al comienzo, tiene el propósito de mejorar la performance. Como explican Lindstrom et al., realizar sólo la simplificación a nivel de vértices sobre todo el terreno sería un proceso computacionalmente muy costoso. Por eso, una selección de detalle de "grano grueso" es llevado a cabo primero, la cual determina qué bloques del *quadtree* utilizar y evitando analizar cada vértice en particular. Luego, sobre los bloques seleccionados, se realiza la simplificación de "grano fino" a nivel de vértice.

Por lo tanto, es necesaria una métrica que permita evaluar el error en que incurre la triangulación completa de un bloque, de manera que un sólo valor deba ser comparado contra el umbral tolerado. Lindstrom et al. utilizan una métrica conservadora que analiza el máximo error de aproximación geométrica posible para cada bloque. Si ese error máximo, proyectado en espacio de pantalla es mayor que el umbral, se deben considerar bloques de mayor resolución (hijos de un bloque en el *quadtree*). El error máximo δ_{max} de un bloque, es calculado como el máximo δ de todos los vértices que serían removidos en la triangulación de menor detalle del bloque.

Finalmente, una vez que el nivel de detalle ha sido seleccionado, dos bloques vecinos pueden haber sido triangulados a distinta resolución, por lo que es posible que aparezcan gritas entre ellos ya que no comparten los mismos vértices en sus bordes. La solución a este problema consiste en considerar las dependencias de cada vértice en todos los bloques en que son utilizados, de manera que cuando un vértice es removido en un bloque, la acción se propague hacia los bloques vecinos. Lindstrom et al. comentan brevemente dos maneras de implementar una solución. En un caso se debería mantener una sola copia de cada vértice, las cuales se accederían mediante el uso de punteros desde los distintos bloques. La otra alternativa permitiría mantener varias copias de un mismo vértice pero debería asegurar que las mismas se mantengan consistentes, por ejemplo utilizando una lista enlazada circular de copias por vértice.

4.2.2 "ROAMing Terrain: Real-time Optimally Adapting Meshes"

El algoritmo ROAM (*Real-time Optimally Adapting Meshes*) fue publicado por Duchaineau et al. en 1997 [6]. En este trabajo se presenta un algoritmo multirresolución para terrenos, que en lugar de basarse en operaciones de remoción de vértices como el algoritmo de Lindstrom et al. [21], las operaciones se realizan enteramente sobre triángulos. ROAM realiza una representación multirresolución del terreno utilizando árboles binarios de triángulos, con el que genera un mallado localmente adaptativo en el que se asigna mayor detalle en zonas planas y lejanas al punto de observación. Además el algoritmo aprovecha la coherencia entre cuadros, asumiendo que la triangulación producida en un cuadro y la triangulación necesaria en el próximo cuadro, sólo diferirán en algunos triángulos.

ROAM representa la superficie del terreno mediante un árbol binario de triángulos o *bintree*, asignando a los vértices de los triángulos las posiciones correspondientes a un conjunto de datos de elevación de grilla regular. Como se ha descripto en el capítulo 2, en un árbol binario de triángulos cada nodo es un triángulo rectángulo isósceles que cubre un área igual al de sus dos hijos. La raíz (nivel 0) se la define como el triángulo de mayor tamaño y de menor resolución del árbol. En el próximo nivel (nivel 1), se define como los dos triángulos que se forman al dividir la raíz a lo largo del lado más largo (hipotenusa) y el vértice opuesto (ángulo recto), formando así dos nuevos triángulos rectángulos isósceles. El resto del árbol es definido repitiendo recursivamente este proceso de división.

Un subconjunto de triángulos del *bintree* en la que dos triángulos cualesquiera no se solapan forman una triangulación. Sobre una determinada triangulación, ROAM permite realizar dos tipos de operaciones inversas entre sí: La operación de combinación o *merge* que reemplaza en la triangulación dos triángulos hermanos del *bintree* por su triángulo padre, y la operación de división o *split* que reemplaza un triángulo por sus dos triángulos hijos.

Cualquier triangulación puede ser obtenida a partir de otra a través de las operaciones de combinación y división, aunque se deben cumplir las siguientes condiciones para

generar un mallado continuo. En una determinada triangulación los vecinos de un triángulo T pertenecen al mismo nivel n en el *bintree* que T (vecinos izquierdo, derecho o base), o a un nivel de mayor detalle $n+1$ (vecinos izquierdo o derecho), o bien a un nivel de menor resolución $n-1$ (vecinos pegados al lado base). Cuando dos triángulos vecinos pertenecen al mismo nivel del *bintree* y están unidos por la base se los denomina un "diamante", como es el caso de la triangulación de menor resolución que abarca todo el terreno.

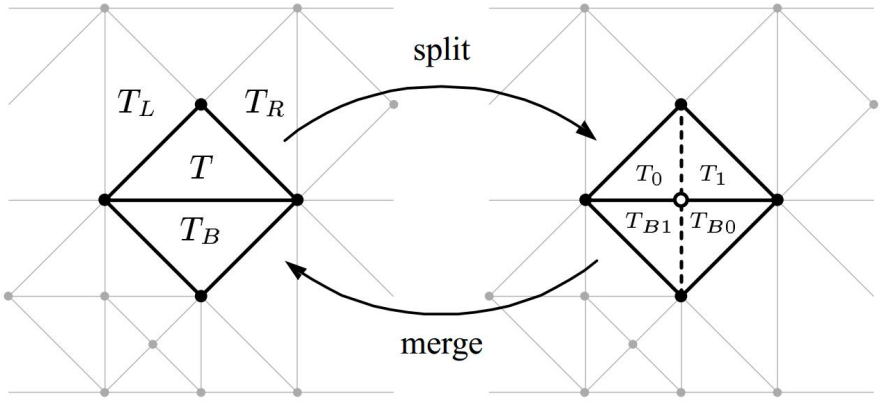

Figura 4.4: Operaciones de combinación (merge) y división (split) sobre el triángulo T. El triángulo T junto al triángulo TB forman un "diamante". [6].

En la figura 4.4, se muestran las operaciones de división (*split*) y combinación (*merge*) sobre un triángulo T. Los triángulos T_I y T_D son los triángulos vecinos izquierdo y derecho respectivamente, y el triángulo T_B es el triángulo vecino por la base. En este caso, al aplicarse la operación de división sobre T, el triángulo T_B también tuvo que ser dividido para cumplir con las condiciones antes expuestas.

Ambas operaciones pueden aplicarse previniendo la aparición de grietas entre triángulos vecinos. Para esto, un triángulo T de la triangulación no puede ser dividido directamente si su triángulo vecino por la base T_B, es del mismo nivel que T en el bintree, o de un nivel de menor detalle. En estos casos, T_B es forzado a dividirse primero. Este mecanismo se aplica de manera recursiva, por lo que al intentar dividir T_B, pueden requerirse nuevamente otros cortes. En la figura 4.5, se muestra como la división forzada de T_B puede desencadenar divisiones en otros triángulos que cumplen la condición mencionada. En el caso de la operación de combinación se aplica el proceso inverso.

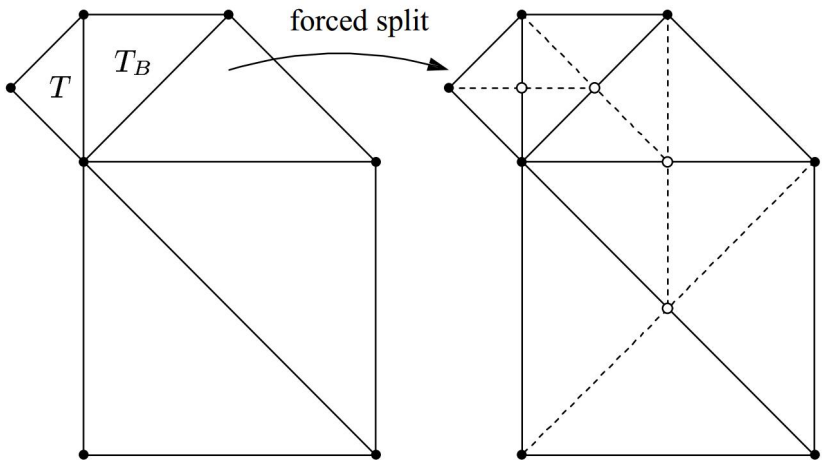

Figura 4.5: La división de un triángulo T puede desencadenar una serie divisiones (forced split), en este caso desde TB hasta los dos triángulos más grandes (líneas discontínuas). [6].

Con el objetivo de optimizar las operaciones de división y combinación, Duchaineau et al. implementan un sistema de dos colas de prioridad, una para cada operación. Estas colas almacenan los triángulos que a ser divididos o combinados respectivamente. Las prioridades utilizadas deben ser monótonas, es decir que la prioridad de un triángulo hijo nunca puede ser mayor que la prioridad de su triángulo padre. De esta manera, es posible hacer el seguimiento de los triángulos que pueden ser divididos o combinados respecto de otra triangulación, lo que le permite aprovechar la coherencia cuadro a cuadro de la escena. En general una triangulación no cambia significativamente desde un cuadro al siguiente, ya que el punto de observación se traslada suavemente y a una velocidad en la que prácticamente las mismas zonas del terreno deben ser visualizadas. Entonces, la misma triangulación puede ser reutilizada aplicando sólo algunas operaciones de división y combinación en cada cuadro, sin necesidad de retriangular el mallado desde cero.

Varias métricas pueden ser utilizadas para calcular las prioridades de los triángulos encolados. La principal métrica de ROAM es la Distorsión Geométrica que se basa en un error de aproximación geométrica proyectado en espacio de pantalla, y con la cual es posible asegurar una representación con error perceptible acotado. En esta métrica, una jerarquía de volúmenes anidados es creada en una etapa de preprocesamiento. Estos volúmenes son llamados *wedgies*. Cada uno se sitúa alrededor de un triángulo, abarcando su área y extendiéndose en altura por un determinado grosor o *tickness*. Para acotar el error, el grosor de los wedgies debe mantenerse anidado, es decir que el espacio que abarca un wedgie siempre debe quedar dentro del espacio del wedgie del triángulo padre. Luego, cada *wedgie* es proyectado en espacio de pantalla y el límite o cota de refinamiento de un triángulo corresponde a la máxima proyección de los segmentos verticales del *wedgie* en todos los puntos del triángulo.

Finalmente, los valores obtenidos por la métrica forman las prioridades de los triángulos en la cola, pero como indican Duchaineau et al. es posible incorporarse

fácilmente otras métricas, como Corrección LOS, Distorsión de Normales, etc. Además los autores señalan que *view frustum culling* puede implementarse como optimización, intersectando el *frustum* de la cámara con los *wedgies* de los triángulos, lo que impediría el posterior refinamiento de los mismos mejorando la performance.

4.2.3 "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering"

Hoppe desarrolla una serie de algoritmos basados en diferentes trabajos de su autoría. En 1996 publica el algoritmo PM (*Progressive Meshes*) [16], el cual permite realizar una representación multirresolución de detalle progresivo de mallas de objetos genéricos utilizando triangulaciones irregulares o TINs (*Triangular Irregular Network*). En 1997, Hoppe publica el trabajo denominado VDPM (*View-Dependent Progressive Meshes*) [17], donde adapta PM para que sea posible una representación basada en parámetros del punto de observación. Por último, Hoppe realiza una síntesis de sus ideas anteriores, presentando en 1998 el trabajo titulado "*Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering"* [18]. En este trabajo muestra como VDPM puede ser utilizado para la visualización de terrenos utilizando un *framework* basado en TINs para la representación multirresolución y progresiva del terreno, logrando obtener aproximaciones muy eficaces del mallado original. Además, en este trabajo Hoppe introduce la técnica *geomorphing* para realizar un cambio suavizado de nivel de detalle.

La superficie del terreno se representa mediante una malla progresiva o PM (*Progressive Mesh*), que permitirá variar en tiempo de ejecución, la resolución de distintas zonas de la superficie del terreno. Para esto, se construye en una etapa de preprocesamiento, una estructura de datos jerárquica de operaciones geométricas. Esta etapa consta de un proceso de simplificación de una malla M^n que se contruye en base a una mallado regular y que representa la superficie del terreno a máxima resolución. A esta malla se le aplica una serie de operaciones de colapso de aristas denominadas *ecol* (*edge collapse*), hasta obtenerse una malla M^0 de menor resolución. Las operaciones *ecol* se almacenan en la estructura de manera inversa, es decir mediante operaciones de división de vértices denominadas *vsplit* (*vertex split*). Así la malla M^n queda determinada como la malla M^0 junto con una secuencia de n operaciones *vsplit* de refinamiento. De esta manera, queda definida una jerarquía de vértices en la que los nodos inferiores contienen los vértices de la malla M^n, y los nodos superiores poseen a los vértices de la malla M^0.

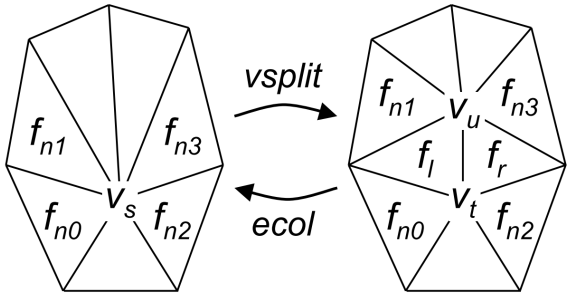

Figure 4.6: Operaciones de división de vértice (vsplit) y de colapso de aristas (ecol). [18.]

En la figura 4.6 se ilustra la relación entre las operaciones *ecol* y *vsplit*. La operación *vsplit* sobre V_s crea dos nuevos vértices V_u y V_t . Los triángulos f_l y f_r rodeadas por los triángulos f_{n0}, f_{n1}, f_{n2} y f_{n3} también son creados. La operación *ecol* trabaja sobre las mismas primitivas pero de manera inversa. En la figura 4.7 se muestra la jerarquía de vértices. El ascenso por esta estructura corresponde a operaciones *ecol* y el descenso a operaciones *vsplit*.

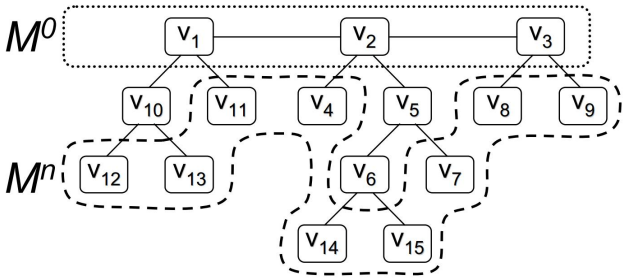

Figure 4.7: La jerarquía de vértices. M^0 corresponde al mallado de menor resolución mientras que M^n es la malla de mayor resolución o mallado original. [18].

En el proceso de construcción de la jerarquía de vértices, la selección de aristas a colapsar depende de un error de aproximación geométrica asociado a la misma. A partir M_n , la arista con menor error es elegida para ser colapsada primero. Con la nueva malla obtenida, la próxima arista de menor error es seleccionada. El proceso continua hasta alcanzar un umbral definido por usuario, obteniéndose así la malla M^0. Respecto del cálculo del error, Hoppe indica que en la métrica utilizada para realizar cada colapso de arista, no sólo se tiene en cuanta el error de aproximación geométrica respecto de los vértices de la malla original M_n (máxima resolución) y los triángulos de la malla en proceso de simplificación, si no que también que agrega al cálculo los vértices formados por las intersecciones de las aristas conectadas a los vértices de la arista siendo colapsada y las aristas de los triángulos de la malla M_n. Así, es posible calcular de forma exacta el error de aproximación geométrica y por tanto la mejor arista a colapsar, y de esta manera minimizar los cambios perceptibles en la geometría

al variar el nivel de detalle.

Una vez creada la jerarquía de vértices, la selección del nivel de detalle en tiempo de ejecución, se realiza actualizando el frente activo de vértices. El frente activo de vértices conforma una malla M irregular refinada selectivamente cuyos vértices constituyen un "frente" a lo largo de la jerarquía (como es el caso de M_n y M_0). La malla selectiva M, se obtiene partiendo de la malla M_0 y aplicando sucesivas operaciones *ecol* y *vsplit*, lo que provoca la variación de la altura del frente en distintas zonas de la jerarquía. La actualización del frente activo se realiza considerando para cada uno de sus vértices si éstos deben ser divididos, combinados o dejados intactos, en base al error de aproximación geométrica asociado a cada uno de estos vértices. La división y combinación de vértices continúa hasta que todos los vértices del frente poseen un error de aproximación geométrica proyectado en espacio de pantalla por debajo del umbral tolerado. De esta manera y como indica Hoppe, la utilización del mecanismo de frente de vértices permite aprovechar la coherencia temporal (coherencia entre cuadros). En cada cuadro un nuevo frente de vértices es formado partiendo del frente utilizado en el cuadro previo, y así no es necesario recorrer toda la jerarquía nuevamente.

Para evitar el efecto visual de saltos al dividir o combinar vértices, Hoppe utiliza la técnica *geomorphing* suavizando la transición visual que resulta de las operaciones *vsplit* y *ecol*. Hoppe utiliza interpolación lineal para trasladar las posiciones de los vértices V_u y V_t hacia o desde la posición de V_s. La interpolación se realiza durante 1 segundo, y así los vértices añadidos o eliminados no surjan o desaparezcan repentinamente. Este proceso sólo se realiza en las regiones visibles por el usuario. Las operaciones geométricas realizadas fuera del *frustum* de la cámara, siguen produciéndose instantáneamente.

Por último, sumado al esquema planteado hasta el momento y con el objetivo de dar soporte a la visualización de grandes extensiones de terreno, la geometría del modelo es dividida en bloques y la jerarquía es construida en un proceso recursivo de simplificación. El proceso comienza con los bloques de mayor nivel de detalle. En cada iteración, una jerarquía de vértices es construida para cada bloque y almacenada en disco, restringiendo las operaciones *ecol* sobre los vértices de los bordes de cada bloque para evitar la formación de grietas entre bloques adyacentes. Luego, una nueva iteración de simplificación comienza uniendo grupos de 2 x 2 bloques mediante la concatenación de sus respectivas jerarquías (una renumeración completa de vértices y parámetros de triángulos es necesaria), y realizando otro serie de operaciones *ecol* sobre la misma. El proceso recursivo continúa hasta obtenerse un sólo bloque final correspondiente a la menor resolución del modelo. Para cada iteración se define un umbral o tolerancia al error de aproximación geométrica de manera de acotar la resolución del nivel y por tanto la cantidad de operaciones *ecol*. Finalmente, en tiempo de ejecución la jerarquía de vértice se construye comenzando por el bloque de menor resolución. Si mayor detalle es necesaria (se excede la tolerancia permitida), nuevos bloques pueden ser cargados y testeados para generar el mallado de triángulos a visualizar.

4.2.4 "Real-Time Generation of Continuous Levels of Detail for Height Fields"

En 1998, Röttger et al. publican el trabajo denominado "*Real-Time Generation of Continuous Levels of Detail for Height Fields"* [31]. Este trabajo presenta un algoritmo basado en un *quadtree* restringido, es decir que los bloques de terreno adyacentes no pueden diferir en más de un nivel de detalle en el árbol. Además, se utiliza una estrategia de refinamiento en lugar de una de simplificación como en el caso de [21]. Así, el recorrido del *quadtree* comienza desde la raíz, lo que permite obtener una mejor performance ya que sólo algunos nodos de la estructura deben ser visitados para alcanzar el nivel de detalle deseado.

El algoritmo requiere que el modelo de datos de elevación inicial posea un tamaño de 2^n+1 vértices de lado. El *quadtree* es representado mediante una matriz booleana, en la que sus elementos corresponden a los nodos del árbol. Cada elemento indica si un nodo es utilizado o no para representar el terreno, y si corresponde a un nodo interno o a una hoja. En la figura 4.8 se muestra un ejemplo de esta matriz y la triangulación que el algoritmo generaría con la misma. Los elementos representados con "1" indican que se trata de un nodo interno, y con "0" que es un nodo hoja. Los elementos marcados con "?" no necesitan ser asignados ya que estos valores no son accedidos para la renderización del terreno. Además en la figura puede verse como cada elemento asignado de la matriz corresponde al centro de la extensión o bloque de vértices que cada nodo del *quadtree* abarca en el terreno.

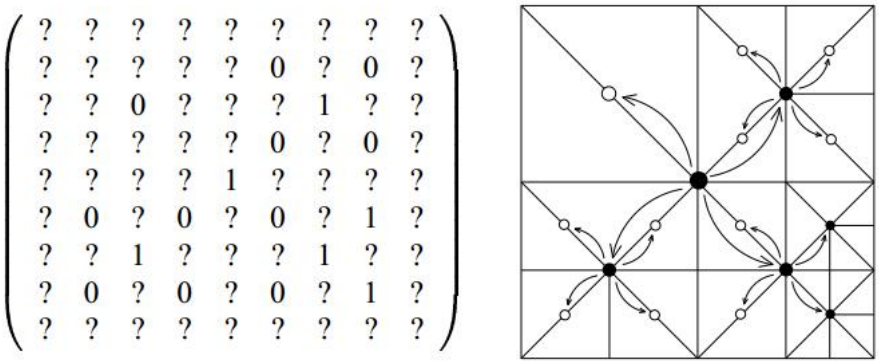

Figura 4.8: Un ejemplo de matriz y la triangulación correspondiente. Las flechas indican el modo de recorrido del quadtree desde el padre hacia los hijos. [31].

La triangulación a visualizar es formada mediante refinamiento, a partir del recorrido recursivo el *quadtree* de acuerdo a los elementos de la matriz. Cuando se llega a una hoja del árbol, los triángulos correspondientes a ese nodo son agregados a la triangulación. Como pueden formarse grietas debido a que bloques vecinos pueden poseer diferente resolución, se evita utilizar el vértice que yace en el centro del borde adyacente al bloque de distinta resolución. Como indican los autores, este método funciona mientras los niveles de detalle de los nodos adyacentes no difieran en más de

un nivel en el *quadtree*. Para cumplir con esta restricción, es necesaria una etapa previa en la que se toma en cuenta la rugosidad del terreno y se completan los valores de la matriz de manera correcta.

Luego, al recorrer el *quadtree* y en cada hoja visitada, se debe detectar si un nodo vecino fue dividido al mismo nivel de detalle o no, para determinar qué triángulos generar en cada borde. Como plantean Röttger et al., es posible detectar esta situación revisando el valor de la matriz correspondiente al nodo vecino. Si el nodo vecino no fue dividido al mismo nivel, se debe evitar el uso del vértice que yace en el centro del borde compartido. El acceso a valores que no han sido asignados no es posible en este esquema, ya que por restricción, las diferencias de nivel entre nodos adyacentes no pueden ser mayores a 1.

Como se ha mencionado en los párrafos anteriores, antes de que la malla pueda ser construida, un proceso de refinamiento es llevado a cabo con el objetivo de completar los valores de la matriz. Comenzando por el centro de la matriz, recursivamente se recorre la misma en el sentido que indican las flechas en la figura 4.8. Si las condiciones son las apropiadas, se debe proceder con el refinamiento, visitando los cuatro hijos. Dos criterios son tenidos en cuenta para realizar el refinamiento, por un lado que el nivel de detalle disminuya a medida que aumenta la distancia al punto de observación, y además que el nivel de detalle aumente en las regiones del terreno que poseen mayor rugosidad.

La rugosidad se calcula a partir de las desviaciones de altura entre el nivel actual y el próximo nivel de cada nodo en el árbol. Los valores se toman en los puntos medios de cada borde y centro del bloque o extensión que abarca el nodo. La rugosidad se calcula como el mayor de estos valores dividido por el largo del bloque. Como indican Röttger et al., el cálculo de la rugosidad mide el error de aproximación en espacio de objeto y es una cota superior para el error que se introduciría si no se continúa con el refinamiento. Luego, la condición de refinamiento se determina mediante la ecuación 4.2.

$$f = \frac{l}{d \cdot C \cdot \max(c \cdot r, 1)} \quad refinar\ si\ f < 1 \tag{4.2}$$

donde l es la distancia del punto de observación al centro del bloque, d es el largo del bloque, C es una constante que controla la resolución mínima global en el terreno, r es la rugosidad en cada nodo, y c es una variable que representa la resolución global deseada la cual puede ser ajustada para obtener un *frame-rate* constante. Finalmente, para garantizar que los bloques adyacentes no difieran en más de un nivel de detalle, los valores de rugosidad son calculados de la siguiente manera. Comenzando por los bloques más pequeños, los valores de rugosidad son propagados hacia arriba por el *quadtree*. El valor de rugosidad de cada bloque es la máxima desviación local más K veces los valores previos calculados de los nodos adyacentes en el próximo nivel de menor resolución. La constante K se calculada utilizando la resolución mínima global

del terreno (ecuación 4.3).

$$K = \frac{C}{2(C-1)} \tag{4.3}$$

Por último, ya que la estructura principal utilizada es un *quadtree*, es posible utilizar *view frumtum culling* para reducir el número de polígonos totales a renderizar. Construyendo una caja envolvente o *bounding-box* en cada nodo del árbol, es posible rechazar la geometría de un nodo fuera del alcance visual de la cámara y por tanto la de todo su subárbol.

4.2.5 Resumen de algoritmos Pre-GPU

En la tabla 4.1 se realiza una síntesis de las principales características de los algoritmos de visualización de terrenos estudiados hasta el momento.

Algoritmo	Datos iniciales	Estructura de datos	Método	Métrica
Real-Time, Continuous Level of Detail Rendering of Height Fields [21].	Datos de elevación en formato de grillas regular 2^n+1.	Arbol cuaternario o *quadtree* donde cada nodo esta constituido por una jerarquía binaria de triángulos.	Doble etapa de simplificación, primero fusionando bloques del *quadtree* y luego fusionando los triangulos dentro de cada bloque.	Máximo error de aproximación entre todos los vértices removidos y proyectado en espacio de pantalla.
ROAMing Terrain: Real-time Optimally Adapting Meshes [6].	Conjunto de datos de elevación de grilla regular 2^n+1.	Arbol binario de triángulos o bintree.	Aprovecha la coherencia entre cuadros aplicando sucesivas operaciones *split* (división de triángulos), y *merge* (combinación de triángulos hermanos).	Distorsión Geométrica basada en la proyección a espacio de pantalla del error de aproximación geométrica calculado a partir de *wedgies*.
Smooth View-	Mallado regular	Estructura	Actualización	Error de

Dependent Level-of-Detail Control and its Application to Terrain Rendering [18].	sin restricciones de tamaño.	arborea de operaciones *vsplit* (división de vértices).	del frente activo aplicando sucesivas operaciones de simplificación *ecol* (edge coolapse) y refinamiento *vsplit* (vertex split).	aproximación geométrica (asociado a la remoción de cada vértice) y proyectado en espacio de pantalla en tiempo de ejecución.
Real-Time Generation of Continuous Levels of Detail for Height Fields [31].	Grilla de 2^n+1 vértices de lado.	*Quadtree* restringido almacenado en formato de matriz.	Triangulación formada por refinamiento mediante el recorrido recursivo del *quadtree*.	Relación que combina dos criterios. Permite disminuir la resolución si aumenta la distancia al observador, y aumentarla en regiones que poseen mayor rugosidad.

Tabla 4.1: Síntesis de algoritmos Pre-GPU.

4.3 Algoritmos Post-GPU

Los algoritmos descriptos hasta el momento fueron desarrollados antes de que el hardware gráfico se volviera un estándar en la industria. Con el advenimiento de la GPU a comienzos de 2000, se experimentó un cambio en la concepción de las soluciones. Los algoritmos debían aprovechar las nuevas capacidades de cómputo cada vez más poderosas del nuevo hardware.

En este sentido, ya no fue necesario preocuparse por generar representaciones que utilicen la menor cantidad de triángulos posibles. Como indica De Bor [3], un nuevo conjunto de algoritmos debieron ser desarrollados, lo cuales deben obtener los mejores resultados junto al hardware 3D. Debido a que el hardware 3D es capaz de procesar y generar grandes cantidades de triángulos por segundo, ~~el~~ nuevo enfoque se basa en enviar tantos triángulos al *pipeline* gráfico como el hardware pueda manejar, a la vez que se libera la CPU de los cálculos necesarios para mantener triangulaciones complejas.

Por otro lado, si bien el nuevo hardware es capaz de renderizar miles de triángulos por

segundo, las velocidades de los buses de comunicación CPU-GPU no han evolucionado al mismo ritmo que la GPU. Esta situación puede provocar un cuello de botella para el envío de la geometría a ser renderizada, limitándose el número de triángulos y desaprovechando la potencia de cómputo de la GPU. La solución a este problema se obtuvo de la capacidad del hardware de mantener la geometría en memoria de la GPU, aspecto que fue aprovechado por los nuevos algoritmos que se desarrollaron. Como resultado, se ha vuelto más efectivo renderizar grandes cantidades de triángulos directamente desde la GPU, con la mínima intervención de la CPU. El hecho de realizar la mayor cantidad de operaciones posibles en la GPU, también ha liberado la CPU para otro tipo de tareas, como ser lógicas más complejas en la aplicación, trabajo en red, simulación de fenómenos físicos, etc.

4.3.1 Geometrical Mipmapping

En 2000, De Boer publica un trabajo denominado "*Fast Terrain Rendering Using Geometrical MipMapping"* [3], introduciendo un nuevo algoritmo multirresolución con el objetivo de aprovechar las características brindadas por el nuevo hardware gráfico (GPU).

El terreno es representado a partir una triangulación regular. Esto es, una grilla de vértices con la misma separación entre filas o columnas y donde los vértices vecinos tomados de a 4 forman cuadriláteros que a su vez son divididos por la diagonal formando 2 triángulos rectángulos isósceles. De Boer impone restricciones a la malla de manera que el número de vértices por fila y por columna sean iguales y de la forma 2^n+1, con $n \geq 1$, lo que conforma 2^n cuadriláteros. Los datos que conforman la elevación de cada vértice son obtenidos desde un *heightmap* de 8 bits en escala de grises que posee las mismas dimensiones que la malla.

Para permitir que regiones del terreno sean visualizadas en distinto detalle, la grilla es dividida en bloques adyacentes más pequeños de la misma forma (2^m+1 con $m \geq 1$ y $m<n$) donde cada uno es simplificado realizando un submuestreo regular formando una secuencia de mallas de LOD decreciente. El autor hace una analogía con la técnica *Mipmapping* de texturas [37]. *Mipmapping* es una técnica de nivel de detalle de texturas en la que una cadena de texturas -o *mipmaps*- de resolución decreciente es creada partiendo de una textura original, y que es utilizada para el muestreo del color de fragmentos dependiendo de la distancia a la cámara. Dada la semejanza entre *mipmaps* y los niveles de detalle de cada bloque geométrico, De Boer bautiza a estos últimos con el nombre de *geomipmaps*. Así un *geomipmap* de nivel 0 corresponde a la mayor resolución, el nivel 1 posee la mitad y así sucesivamente hasta alcanzar el nivel de menor detalle de sólo 2 triángulos (ver figura 4.9-a). La cantidad de niveles o *geomipmaps* depende del tamaño de bloque utilizado y del tamaño total del terreno.

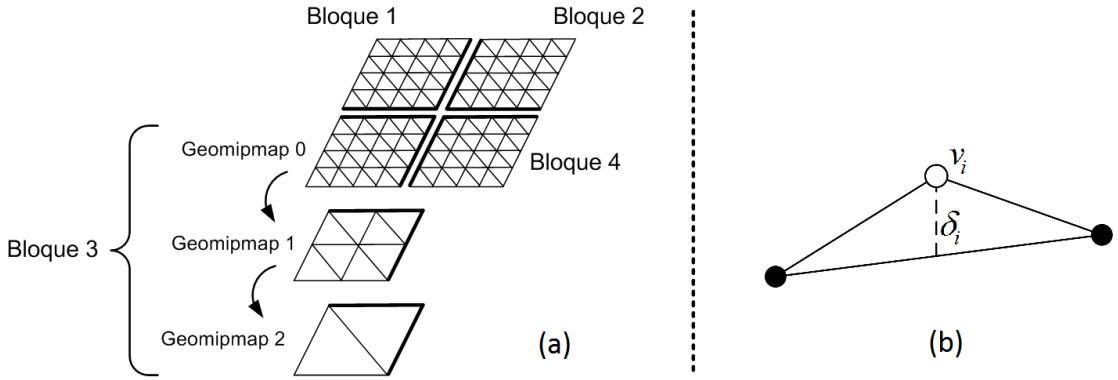

Figura 4.9: (a) Simplificación de bloques para la generación de geomipmaps en un ejemplo con 4 bloques. (b) El error de aproximación geométrica por vértice δ_i es la distancia vertical entre el vértice removido v_i y la malla simplificada.

Además, para evitar que grandes cantidades de geometría sean enviados al *pipeline* gráfico, un *quadtree* de cajas 3D envolventes (*bounding boxes*) es creado. La caja de la raíz abarca todo el terreno y cada uno de sus hijos un cuarto del terreno, y así sucesivamente hasta llegar a las hojas, las cuales coinciden con los bloques resultantes de la subdivisión inicial de la malla. Luego, mediante *view frustum culling* contra cada caja envolvente del *quadtree* se evita o rechaza la renderización de los bloques que no son comprendidos en el espacio visual de la cámara.

Es necesario poder seleccionar que *geomipmap* utilizar en cada región del terreno. Para esto, cada *geomipmap* tiene asociado un error de aproximación geométrica en espacio de objeto, calculado en base a la remoción de vértices respecto del *geomipmap* que lo precede. Al remover cada vértice v_i el error δ_i es calculado como la distancia vertical entre la posición del vértice y la malla simplificada (ver figura 4.9-b). El error asignado a cada *geomipmap* es calculado como el máximo de los errores δ_i de manera de considerar el peor caso. Luego, para la selección del *geomipmap* a renderizar, se utiliza una métrica de error de aproximación en espacio de imagen (sección 2.4.3). Esta métrica proyecta el error del *geomipmap* a espacio de imagen obteniéndose un valor en píxeles que luego es comparado contra un umbral de error máximo tolerado. Si el error proyectado es menor que el umbral, un *geomipmap* de menor resolución puede utilizarse para aproximar la zona del terreno.

Como pueden surgir grietas en el terreno cuando se utilizan dos *geomipmaps* vecinos de distinto nivel de detalle ya que no tendrán el mismo número de vértices en los borden adyacentes. De Boer soluciona el inconveniente modifica las interconexiones entre los vértices del borde del *geomipmap* de mayor detalle. Para esto se realiza una triangulación especial que omite los vértices en el borde de manera de igualar la cantidad de vértices en los bordes de ambos *geomipmaps* como se muestra en la figura 4.10.

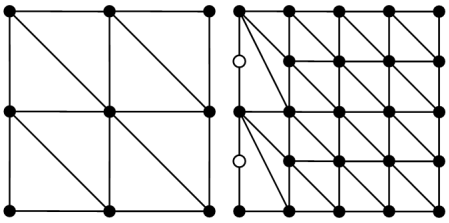

Figura 4.10: Geomipmaps de distinta resolución. En la triangulación del geomipmap de mayor resolución se omiten los vértices no compartidos (en blanco) por el geomipmap vecino.

Por último, De Boer comenta que el algoritmo puede ser extendido en algunos aspectos. En primer lugar se podría implementar *geomorphing* mediante el cambio gradual del nivel de detalle para disminuir la ocurrencia del efecto *popping*. Para esto, el autor propone una técnica que denomina *GeoMipMapping Trilineal* que requeriría contar con las distancias mínimas a la cámara, en la que cada *geomipmap* puede utilizarse antes de pasar al próximo de mayor detalle. Luego como segunda extensión, para el caso en que el terreno no entrara por completo en memoria principal, enumera una secuencia de pasos posibles a llevar a cabo para la carda dinámica de los datos de altura de cada bloque.

Finalmente, en 2003 Larsen et al. [20] basan su técnica en el algoritmo Geometrical Mipmapping, sobre el que proponen un nuevo modo de evitar grietas entre bloques vecinos (rebautizados como mosaicos o *tiles*). La técnica consiste en interconectar los vértices de los bloques de terreno como muestra la figura 4.11. Además los autores también desarrollan un método concreto de *geomorphing* basado en el nuevo modo de triangulación.

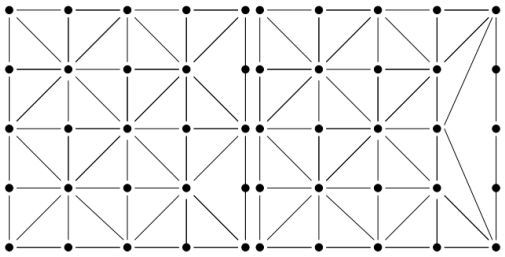

Figura 4.11: Un mismo mosaico que posee un vecino de menor resolución a la derecha. La diferencia en niveles de detalle es de 1 (izquierda) y de 2 (derecha) [20].

4.3.2 Chunked LOD

Ulrich presenta en 2002 un nuevo algoritmo multirresolución de visualización de terrenos extensos utilizando el hardware gráfico moderno (GPU) [35]. El trabajo denominado "*Rendering Massive Terrain using Chunked Level of Detail Contol*"

presenta un algoritmo que tiene aspectos que lo asemejan a Geometrical Mipmapping (sección 4.3.1), aunque difiere de éste en puntos importantes.

El algoritmo se basa en un *quadtree*. Los nodos abarcan una porción menor del terreno a medida que se desciende por el árbol. Así el nodo raíz abarca todo el terreno, y un nodo cualquiera abarca un cuarto de la extensión que abarca su nodo padre. Cada nodo posee su propia malla denominada *chunk*, para representar la zona del terreno que abarca (figura 4.12). El *chunk* de la raíz aproxima todo el terreno con poco nivel de detalle. A medida que se desciende por el árbol, los *chunks* ganan resolución pero también abarcan extensiones menores, por lo que la cantidad de triángulos utilizados por cada *chunk* se mantiene relativamente constante. Los nodos hoja poseen los *chunks* de mayor nivel de detalle y de menor extensión de todo el *quadtree*.

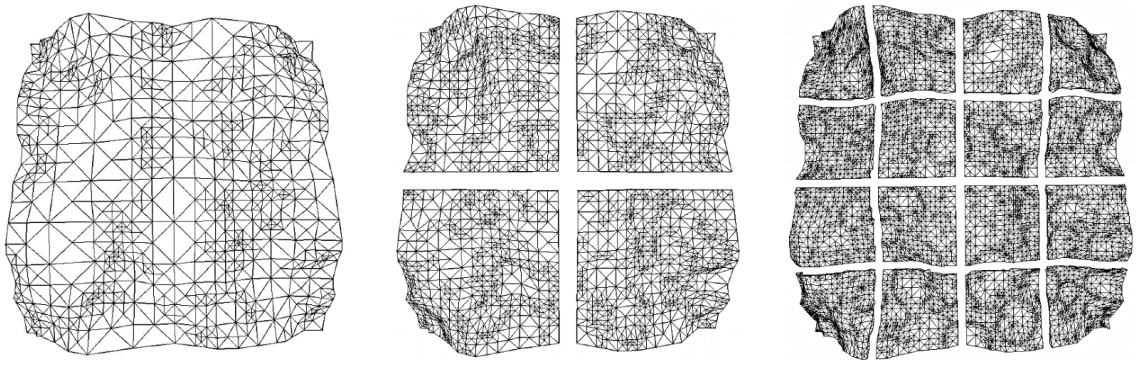

Figura 4.12: Quadtree de chunks. A la izquierda el chunk de la raiz, en el medio los chunks de sus cuatro hijos, y a la derecha los chunks de los nodos hoja [35].

En una etapa de preprocesamiento el *quadtree* es construido y los *chunks* son creados mediante refinamiento. Para formar la malla de cada *chunk* se utiliza un esquema de triangulación jerárquico, aunque el autor no indica exactamente cual. Para la raíz del *quadtree*, se calcula el error de aproximación geométrica, que al igual que en Geometrical Mipmapping es calculado como la máxima desviación en espacio de objeto respecto de los datos de elevación del terreno original, en la que se incurre al omitir vértices al realizar la simplificación. Luego los *chunks* sucesivos son construidos de manera que la cantidad de triángulos se reduzca a la mitad en cada nivel en que se desciende por el árbol, y se asume que sus errores de aproximación en espacio de objeto también se reducen en la misma proporción. Así por ejemplo si un nodo poseyera un error de 16, el error de sus nodos hijo debe ser de 8, y de 4 para los nietos.

En tiempo de ejecución, la selección del nivel de detalle se realiza de forma recursiva comenzando desde la raíz. En cada nodo visitado, el error de aproximación del *chunk* se proyecta en pantalla. Si el error proyectado es menor a un umbral determinado, el *chunk* del nodo es renderizado. En caso contrario, se procede recursivamente con el recorrido de los hijos para determinar si sus *chunks* poseen el nivel de detalle apropiado. Además, este proceso recursivo también es utilizado para realizar *view frustum culling*. Comenzando desde la raíz, si un nodo no es visible por la cámara,

todos los hijos del nodo son descartados y el proceso recursivo se detiene en esa rama del árbol.

La selección de *chunks* se realiza de manera independiente, lo que puede llevar a la aparición de grietas entre los bordes de *chunks* vecinos de distinto LOD. Como solución, Ulrich propone utilizar geometría adicional que se sitúa alrededor de los *chunks* y de manera perpendicular al terreno, y que luce como una "pollera" o *skirt* como las denomina el autor (ver figura 4.13). La parte superior de una pollera debe coincidir con los bordes del *chunk*, y se entiende hacia abajo por una extensión determinada por parámetro.

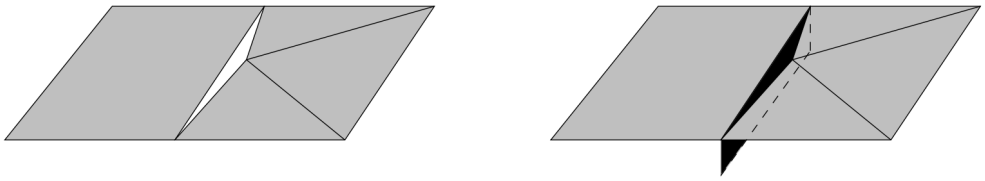

Figura 4.13: Polleras o skirts (color negro) utilizadas para llenar las grietas entre dos chunks (imagen izquierda). [35].

Por último, para minimizar el efecto de *popping* ante un cambio de nivel de detalle, se utiliza *geomorphing*, planteando un cambio gradual del detalle de manera similar a la propuesta de De Boer en Geometrical Mipmapping. Además se menciona una forma simple de empaquetar los datos del *quadtree* en la etapa de preprocesamiento, de manera de ocupar el menor tamaño posible, lo que también permitiría la carga dinámica desde disco. Ya que los *chunks* son independientes, sólo aquellos visibles y en el nivel de detalle utilizado son necesarios. Esto permitiría soportar terrenos extensos mientras el conjunto de *chunks* visibles en un momento dado quepan en memoria principal.

4.3.3 Geometry Clipmaps

En 2004, Losasso et al. publican el trabajo denominado "*Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids*" [24], en el que presentan un algoritmo novedoso de visualización de terrenos por GPU, basado en la idea de *clipmap*, una representación actualizable de *mipmap* en la que cada nivel de resolución de textura es recortado (*clipped*) al un tamaño máximo que puede ser accedido [34]. Este algoritmo emplea la potencia de cómputo presente en las GPU del momento con el objetivo de formar triangulaciones donde cada triángulo sea aproximadamente del tamaño de 1 píxel en pantalla. En este algoritmo, el nivel de detalle se encuentra anidado alrededor del punto de observación y sólo depende de la distancia a éste, y no de la rugosidad del terreno.

Geometry Clipmaps representa el terreno en un conjunto de grillas regulares anidadas que se centran alrededor de la posición 2D del punto de observación sobre el plano del terreno. Cada grilla posee un nivel de dalle diferente, disminuyendo la resolución hacia

las grillas externas (figura 4.14). Los datos de elevación que conforman las grillas son almacenados como buffers de vértices en memoria de la GPU. A medida que el punto de observación se traslada, los vértices son actualizados de manera que las grillas se mantengan siempre centradas respecto del observador.

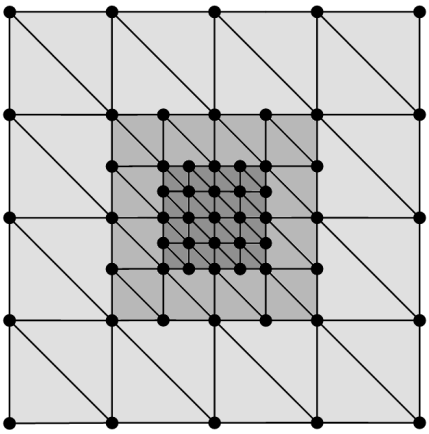

Figura 4.14: Grillas anidadas de resolución decreciente.

Los datos de los vértices se estructuran de manera que puedan ser accedidos de forma circular. A medida que el punto de observación se mueve sobre el plano del terreno, un proceso de actualización incremental es llevado a cabo para mantener los datos en GPU centrados respecto del observador. En lugar de reemplazar la estructura completa, sólo los datos de las áreas obsoletas son reemplazados. Como puede verse en la figura 4.15, sólo es necesario incorporar las franjas correspondientes a las nuevas áreas visibles, mientas que los datos útiles se mantienen intactos.

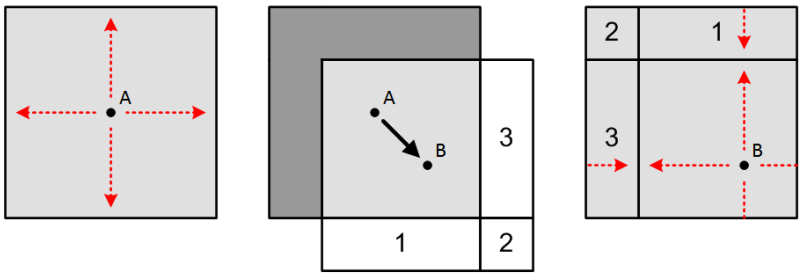

Figura 4.15: Actualización de la geometría mediante acceso circular. El observador se traslada del punto A al punto B por lo que las nuevas zonas visibles (1, 2 y 3) son cargadas en las regiones liberadas (gris oscuro).

La información completa de las alturas del terreno (*heightmap*) es mantenida en memoria principal mediante el esquema de compresión de imágenes PTC (*Pogressive Transform Coder*) de Malvar [26], que permite una reconstrucción progresiva ROI (*Region Of Interest*) eficiente en imágenes de tamaño arbitrario. La descompresión

sólo toma lugar durante la carga a memoria de la GPU de las nuevas regiones reveladas al trasladarse el punto de observación, dando lugar a que la CPU realice la descompresión sin la necesidad de acceder a disco en tiempo de ejecución. Un preprocesamiento *offline* de compresión de los datos es necesario.

Geometry clipmaps, también debe lidiar con las discontinuidades generadas por el uso de distintos niveles de detalle. En primer lugar, el algoritmo define zonas sobre la geometría cercana al límite exterior de cada grilla, en las que se aplica un cambio suave de nivel de detalle o *geomorphing* hacia el nivel de menor resolución, de manera contrarrestar el efecto *popping*. Por otro lado y como indican los autores, podrían aparecer grietas en los bordes de cada grilla ya que existen vértices utilizados por el nivel de mayor resolución pero que no están definidos en el nivel de menor resolución (vértices-T). En este caso, los autores proponen utilizar triángulos adicionales en los bordes de las grillas. Los triángulos se forman interconectando vértices consecutivos a lo largo de los bordes exteriores de las grillas, tomándolos de a tres y reutilizando el último vértice en el próximo trío. Así por ejemplo, si se tuviera una grilla con 5 vértices de lado correspondería formar dos triángulos en cada lado. A estos triángulos se los denomina "triángulos de área cero", ya que en el caso de coincidir los valores de altura de los vértices respecto del plano del terreno, los vértices quedarán coalieados y por tanto el triángulo tendrá área nula. En tal caso, el triángulo será descartado por el pipeline de la GPU.

Por último, para evitar enviar al *pipeline* grandes cantidades de geometría, se aplica la técnica de *view frustum culling* en cada nivel de detalle del *clipmap*. Cada grilla es subdividida en 4 regiones rectangulares, y a cada región se le asigna una caja 3D envolvente. Luego sólo aquellas regiones cuya caja es intersectada con el *frustum* de la cámara son renderizadas.

Finalmente, en 2005 Asirvatham et al. [2] realizan una serie de adecuaciones a Geometry Clipmaps para aprovechar las nuevas características de la GPU del momento, trasladando varios aspectos del algoritmo hacia a la GPU, y liberando notablemente el trabajo realizado en CPU. Entre estos cambios se destaca el hecho de realizar la renderización del terreno directamente desde una textura o *heightmap* almacenado en la GPU. Asímismo y referido a esta misma técnica, en 2011 Guaycochea et al. [13] toman el trabajo de Asirvatham y cambian el criterio de simplificación analizando el error visual incurrido en el uso de un determinado LOD, logrando minimizar el cambio geométrico del terreno perceptible por el usuario.

4.3.4 CDLOD

En 2009, Strugar presenta en su artículo "*Continuous Distance-Dependent Level of Detail for Rendering Heightmaps"* el algoritmo CDLOD [33], el cual se estructura en un *quadtree* de bloques de igual modo que Chunked LOD, pero a diferencia de este se utilizan mallados regulares. Al igual que [2], CDLOD renderiza el terreno desde un *heightmap* almacenado en GPU gracias a las características de acceso a texturas

mejorado presentes a partir del Shader Model 3. Además, CDLOD incorpora una técnica simple para la selección de nivel de detalle que a su vez posibilita realizar una transición suave entre LODs en la que no es necesario evitar grietas o *T-junctions*.

CDLOD hereda de Chuncked LOD la forma de organizar la información de elevación en un *quadtree*, donde cada nodo del mismo representa una zona de distinta resolución. El *quadtree* es utilizado en tiempo de ejecución para seleccionar el nivel de detalle de las distintas partes del terreno. Cada nivel del *quadtree* corresponde a un LOD distinto y posee el doble de resolución que el nivel anterior.

Un nodo de un determinado LOD es seleccionado aplicando exclusivamente el criterio de distancia al punto de observación, con el objetivo de producir aproximadamente la misma cantidad de triángulos en pantalla. Como cada nivel adicional de LOD implica 4 nodos más, la diferencia en distancia cubierta por un nodo de un LOD adicional se ve reducida a la mitad. Strugar muetra que las distancias para cada LOD pueden ser precalculadas. La selección del nivel de detalle es realizada mediante un arreglo de rangos para LODs, donde cada rango es del doble de su anterior, siendo el rango del último el requerido para la distancia máxima de visión (ver figura 4.16).

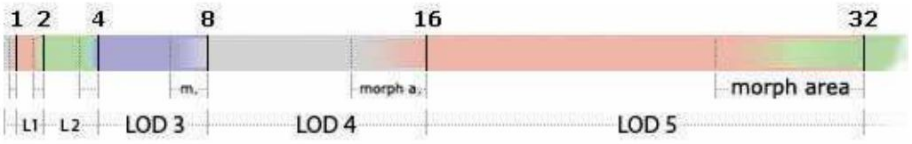

Figura 4.16: Distribución de seis rangos de LOD con sus tamaños relativos. [33].

Para seleccionar los nodos, el *quadtree* es recorrido comenzando desde la raiz y descendiendo por el mismo de manera recursiva. Cada nodo puede ser seleccionado parcialmente de manera que no todos los nodos deban ser renderizados si sólo algunos entran en su rango de LOD. Mientras se recorre el árbol, la técnica de *view frustum culling* también realizada sobre las cajas envolventes que cada nodo posee.

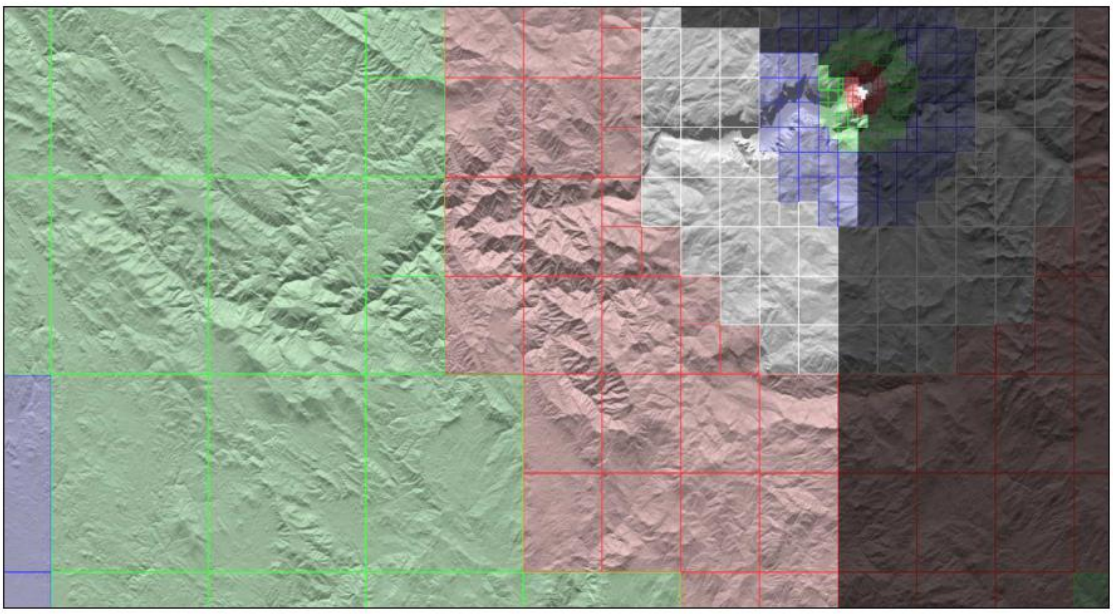

Figura 4.17: Representación del terreno mediante los nodos seleccionados del quadtree. Los nodos descartados por frustum culling se muestran en gris. [33].

La presentación del terreno se realiza renderizando las áreas cubiertas por los nodos seleccionados del *quadtree* (figura 4.17). Para renderizar cada nodo se utiliza una única malla de dimensiones fijas cuyos vértices son transformados para cubrir la zona de la superficie del terreno que corresponda. La transformación de la malla se lleva a cabo directamente en la GPU. El desplazamiento de la misma es conseguido mediante un conjunto de datos (posición, tamaño, LOD) recolectados para cada nodo durante el recorrido del *quadtree*, mientras que la adaptación a la topología particular de la zona se logra mediante el muestreo de un *heightmap* almacenado en memoria de GPU como una textura.

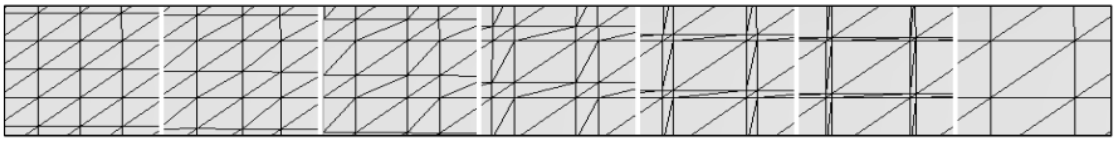

Figura 4.18: Geomorphing de la malla dividido en 7 pasos discretos para su ilustración. [33].

Por último, este algoritmo también aplica la técnica *geomorphing* pero en este caso realizada en GPU. Cada vértice es transformado por separado y soportandose transiciones entre LODs consecutivos. El *morph* se realiza de modo que cada bloque de 8 triángulos de la malla de mayor detalle sea transformado suavemente en el bloque correspondiente de 2 triángulos de la malla de menor detalle, haciendo que gradualmente se agranden los dos triángulos y se reduzcan los 6 triángulos restantes en triángulos degenerados que no son rasterizados (ver figura 4.18). Este proceso

produce transiciones suaves y sin que se formen grietas o vértices-T. Para las transiciones entre LODs, un rango de *morphing* es definido como el último 15% a 30% (configurable según aplicación) de cada rango del arreglo de rangos de LOD (figura 4.16).

4.3.5 Resumen de algoritmos Post-GPU

Geometry Clipmaps [24] se destaca por dos características principales. Por un lado presenta un esquema muy interesante de actualización de datos en GPU. A medida que el observador se mueve, el proceso de actualización de datos es llevado a cabo de forma incremental. Esto quiere decir que no es necesario realizar la carga completa de los datos de altura, lo que permite trabajar con terrenos extensos. La otra característica importante de este algoritmo radica en su forma de representar el terreno. Geometry Clipmaps organiza el terreno mediante un conjunto de grillas regulares (anillos) de distinto nivel de detalle. Estos anillos se anidan respecto del punto de observación los cuales a su vez se dividen en bloques (descartados si no son visibles por la cámara). De esta manera, una implementación de la estructura mencionada debe llevar necesariamente a la iteración sobre una lista de anillos al momento de la renderización. Si bien este esquema puede ser eficiente, no es sencillo realizar cambios sobre el mismo.

Por otro lado se encuentra los algoritmos basados en *quadtrees*, como Geometrical Mipmapping [3] y Chunked LOD [35]. Estos algoritmos son mucho más versátiles y simples de mantener que Geometry Clipmaps, ya que permiten por ejemplo cambiar la manera de seleccionar el nivel de detalle (función LOD) sin la necesidad de modificar la estructura de organización del terreno o el tamaño de los bloques. Además poseen otro beneficio, y es que permiten descartar grandes cantidades de terreno no visible de forma temprana en el recorrido recursivo del árbol.

Geometrical Mipmapping es además un algoritmo sencillo y de gran performance. Su característica principal es que almacena la información de altura sólo en las hojas del *quadtree*. En este esquema sigue siendo posible descartar grandes cantidades de terreno durante el recorrido del árbol ya que el chequeo de visibilidad se sigue realizando sobre los nodos internos. La desventaja que presenta es que todos los bloques de terreno son del mismo tamaño (del tamaño de las hojas). Esto hace que el algoritmo no sea una solución del todo escalable, ya que por ejemplo si el punto de observación se encuentra en altura respecto del plano del terreno y con una dirección visual hacia el horizonte, es necesario renderizar mucha cantidad de bloques “pequeños”, y con una resolución que no puede ser inferior a la mínima establecida en los *mipmaps*.

En el caso de ChunkedLOD, sí aloja la información de altura en todos los nodos del *quadtree*, lo que le permite ser una solución totalmente escalable. Por otro lado, utiliza un esquema de triangulación cuya implementación requiere conocer y mantener la malla de cada bloque de terreno o *chunk*, por lo que es necesario un mecanismo de

actualización de buffers geométricos alojados en GPU. Este esquema de envío y actualización de geometría no es adecuado a la tendencia actual de reducir cada vez más el intercambio CPU-GPU.

Por último, CDLOD [33] se estructura del mismo modo que ChunkedLOD (*quadtree* de bloques), pero le introduce algunas modificaciones. La más significativa respecto de la performance, consiste en alojar la información de altura (*heightmap*) en memoria de GPU. La información de altura puede ser ahora accedida directamente sin intervención de la CPU, lo que minimiza el intercambio de datos hacia la GPU, convirtiéndolo en un algoritmo muy eficiente.

En la tabla 4.2 se sintetizan las principales características de los algoritmos de visualización de terrenos Post-GPU.

Algoritmo	**Datos iniciales**	**Estructura de datos**	**Método**	**Métrica**
Geometrical Mipmapping [3].	Conjunto de datos de elevación de grillas regular 2n+1.	Matriz de bloques multirresolución (geomipmaps) situados en las hojas de un *quadtree*.	Obtención de bloques mediante recorrido del *quadtree*. Simplificación de bloques según métrica.	Máximo error de aproximación entre los vértices removidos en cada nivel de geomipmap, y proyectado en espacio de pantalla.
Chunked LOD [35].	Conjunto de datos de elevación de grillas regular 2n+1.	*Quadtree* de bloques de terreno superpuestos. Cada bloque ocupa el área de sus 4 hijos.	Refinamiento por recorrido descendente del *quadtree*.	Error de aproximación generado por simplificación del mallado original y reducido a la mitad en cada descenso por el *quadtree*. Luego es proyectado en espacio de pantalla.
Geometry Clipmaps [24].	Conjunto de datos de elevación de grillas regular 2n+1.	Conjunto de grillas regulares anidadas de detalle decreciente	Actualización de grillas anidadas de manera de mantenerlas	Debido al esquema de grillas anidadas, el nivel de detalle

		hacias las grillas exteriores.	siempre centradas respecto del observador.	desciende con el cuadrado de la distancia.
CDLOD [33].	Conjunto de datos de elevación de grillas regular 2n+1.	*Quadtree* de bloques de terreno superpuestos. Cada bloque ocupa el área de sus 4 hijos.	Refinamiento por recorrido descendente del *quadtree*.	Esquema basado en vector de rangos de distancia que produce que el nivel de detalle descienda con el cuadrado de la distancia.

Tabla 4.2: Síntesis de algoritmos Post-GPU

.

5 DISEÑO DEL ALGORITMO

En este capítulo realiza la descripción del algoritmo de visualización de terrenos 3D multirresolución desarrollado como parte de la concreción de esta tesina. El capítulo se divide en las siguientes secciones:

En la sección 5.1 se describen las condiciones particulares que han motivado el diseño de un nuevo algoritmo de visualización de terrenos.

El resto del capítulo se dedica a describir cada aspecto del diseño en detalle. Estos aspectos son: representación del modelo de terreno (sección 5.2), selección de nivel de detalle (sección 5.3), renderización (sección 5.4), prevención de grietas entre bloques (sección 5.5), y *geomorphing* (sección 5.6).

5.1 Motivación del diseño

En el capítulo 4 se han presentado los algoritmos multirresolución más relevantes. Estos algoritmos han sido clasificados en dos grupos: aquellos desarrollados en una etapa previa al surgimiento de la GPU, y aquellos basados especialmente en el uso de la misma. Los algoritmos del primer grupo, no son directamente aplicables a la visualización de terrenos hoy en día, al menos en su forma original o básica. Los algoritmos más modernos se ven beneficiados por las características y capacidades de cómputo cada día más poderosas de la GPU. Por esta razón el software desarrollado para este trabajo se basó en estos últimos algoritmos.

Como se ha desarrollado en la sección 4.3.5, el algoritmo CDLOD [33] es uno de los algoritmos más eficiente y escalable de las técnicas del estado del arte. Por esta razón se lo ha considerado como base de partida de un nuevo desarrollo.

El algoritmo CDLOD utiliza la distancia al punto de observación como único criterio de selección de LOD, obteniéndose una distribución pareja de triángulos en pantalla. Sin embargo, se considera que este enfoque sufre de algunas limitaciones en relación al criterio de selección del LOD. Como la selección del LOD sólo depende de la distancia al punto de observación, la percepción de cambios de LOD se ve maximizada, ya que la representación se vuelve independiente de la rugosidad del terreno. Por esto, si se disminuye la cantidad de geometría para representar el terreno en general -por ejemplo para mejorar la performance-, pueden percibirse cambios al visualizar zonas de terreno particularmente rugosas, ya que a una misma distancia se asigna la misma resolución independientemente de la rugosidad. Este esquema provoca el retraso en la aparición de las zonas más rugosas del terreno a medida que el observador se acerca a ellas. Al acercarse se percibe como un fenómeno de surgimiento de accidentes geográficos (picos, cimas, etc.) ya que es entonces cuando se le asigna mayor

resolución. Podría pensarse como solución, mantener el esquema de asignación de detalle según distancia y aumentar la resolución general del terreno, pero esto asignaría geometría a zonas llanas donde no es necesaria para su representación, aumentando el tiempo de renderización.

La motivación de este trabajo surge de considerar las limitaciones expuestas anteriormente originando el diseño de un nuevo algoritmo que posibilite utilizar la geometría de manera más apropiada, obteniendo una mejor utilización de la geometría en cuanto a los niveles de detalle necesarios para aproximar los distintos accidentes del terreno. Para esto es necesario disponer del error de aproximación en espacio de objeto para cada nodo del *quadtree*, lo que permitirá seleccionar los niveles de detalle considerando la rugosidad propia de cada zona del terreno, asignando mayor detalle a zonas más rugosas, y menor detalle a zonas llanas.

Por último, se pretende incorporar las nuevas características de la GPU actual en el diseño de la técnica de visualización. Las últimas características de la GPU corresponden al modelo de programación Shader Model 5. Este modelo de programación incorpora tres nuevas etapas al *pipeline* gráfico que en conjunto permiten implementar un sistema flexible de teselado de primitivas. El teselado por hardware representa un avance considerable para la multirresolución ya que la CPU puede desligarse de realizar la triangulación del mallado según el nivel de detalle requerido. Esto tiene dos beneficios, por un lado el de liberar la CPU para realizar otras tareas, y por otro es posible obtener un algoritmo más versátil, eficiente y fáciles de mantener.

5.2 Representación del terreno multirresolución

5.2.1 Representación Quadtree

La información de altura es codificada en un *heightmap*, es decir una imagen (grilla regular de píxeles) en blanco y negro en la que se toma la intensidad de cada píxel como la altura en ese punto. La representación de la superficie del terreno se realiza conformando una grilla de dimensión 2^n+1, con $n \geq 1$ con muestras de altura separadas a intervalos regulares, para lo cual es necesario suministrar tanto la resolución de altura de los píxeles (o altura máxima), como así también la dimensión o separación entre los mismos.

Un árbol cuaternario o *quadtree* de bloques es utilizado para organizar dicha grilla. Este *quadtree* es del tipo utilizado en CDLOD [33] (originalmente descripto en ChunkedLOD [35]), donde la información de altura es modelada en todos los nodo del árbol. De esta manera, el *quadtree* se utiliza como una estructura de subdivisión 2D de la superficie del terreno como se ilustra en la figura 5.1. Los nodos del *quadtree* representan una porción cuadrada del terreno a distinto nivel de detalle. La raíz abarca toda la superficie del mismo, mientras que el resto de los nodos (nodos internos y hoja) abarcan un cuarto de la extensión que abarca su correspondiente nodo padre.

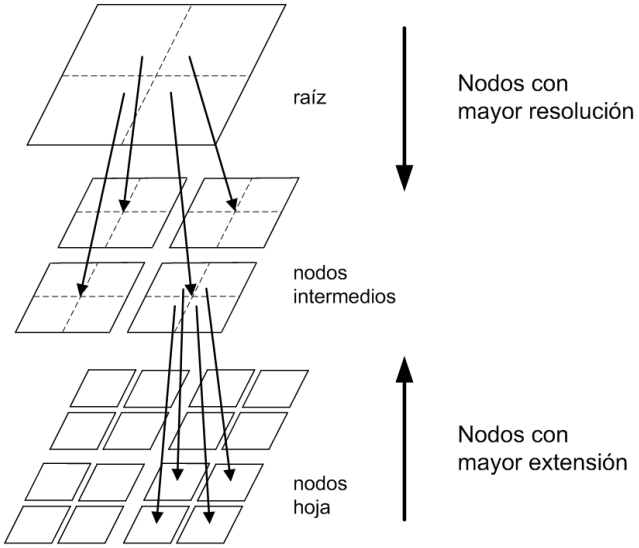

Figura 5.1: Representación del terreno mediante quadtree de bloques.

El modelo multirresolución se logra aplicando distinto nivel de detalle según la altura de los nodos en el *quadtree*. Cada nivel del *quadtree* posee el nivel de detalle correspondiente al cuádruple de resolución que el nivel superior.

La raíz representa todo el terreno a un nivel de detalle muy bajo. Sus cuatro hijos cubren un cuarto de la superficie del terreno pero al doble de resolución, y así sucesivamente. Finalmente las hojas del árbol poseen la menor extensión y el mayor nivel de detalle posible, que corresponde a utilizar todas las muestras disponibles de la grilla para la zona abarcada.

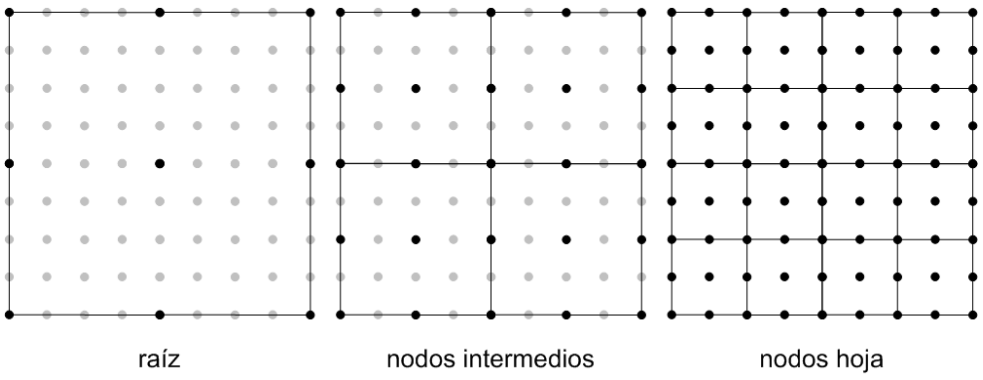

Figura 5.2: Extensión que abarcan los nodos de un quadtree de altura 3 sobre una grilla de 9x9 muestras. Las muestras consideradas para representar la zona abarcada por un nodo se muestran en color negro..

El nivel de detalle de cada nivel del *quadtree* es conseguido realizando un submuestreo regular sobre la grilla de muestras subyacente (figura 5.2), de manera que todos los nodos sin importar su altura en el *quadtree*, emplean la misma cantidad de muestras

para representar la superficie que abarcan. Es necesario restringir la dimensión de la grilla de manera que el número de muestras por fila y por columna sean iguales y de la forma 2^n+1, con $n \geq 1$. De esta manera al subdividir un nodo para obtener sus cuatro hijos, las muestras en los bordes de los nodos hijo se solapan en una fila o columna cubriendo todo el terreno.

5.2.2 Recorrido del quadtree

El algoritmo de renderización consiste en un proceso de refinamiento recursivo. El objetivo de este proceso es el identificar los nodos del *quadtree* que posean el nivel de detalle más apropiado para cada zona del terreno.

El *quadtree* es recorrido de forma descendente comenzando desde la raíz. Cuando se alcanza un nodo con el nivel de detalle apropiado, el proceso recursivo se detiene y el nodo es utilizado para la renderización de un nuevo bloque de terreno. En caso contrario, es decir que no posea el detalle adecuado, se continúa recursivamente con los hijos del nodo como se muestra en el siguiente pseudocódigo:

```
funcion recorrer_quadtree(nodo N)
      Si no intersecta_con_frustum(N)
            retornar
      Si posee_lod_adecuado(N) o es_hoja(N)
            renderizar(N)
      Sino
            Para cada hijo H del nodo N
                  recorrer_quadtree(H)
fin funcion
```

El recorrido del *quadtree* es sumamente sencillo, y permite incorporar la técnica de *view frustum culling* (sección 4.1.1) de forma efectiva. La función `intersecta_con_frustum(nodo)` del pseudocódigo encapsula la implementación de esta técnica, y se basa en descartar los nodos no visiables deteniendo el recorrido recursivo del *quadtree*. Para esto, cada nodo del *quadtree* debe poseer una caja 3D alineada a los ejes cartesianos (*Axis Aligned Bounding Box*, AABB) que envuelve la geometría de la zona que el nodo representa. La caja debe ser obtenida a partir del valor máximo y mínimo de las coordenadas (x, y, z) de los vértices de la zona del terreno -a máxima resolución- abarcada por el nodo. De esta manera, la estructura del *quadtree* es aprovechada para organizar las cajas envolventes de forma jerárquica. Si la caja de un nodo no es visible por el *frustum* de la cámara, entonces todos los nodos hijo tampoco son visibles, ya que los mismos se encuentran contenidos dentro de la caja del nodo padre. Esta organización permite realizar un recorrido óptimo del *quadtree* evitando visitar ramas enteras del mismo. Esto permite descartar grandes cantidades de geometría de las zonas que no son comprendidas en el espacio visual de la cámara.

Por último, la función `posee_lod_adecuado(nodo)` del pseudocódigo permite encapsular la métrica de selección de nivel de detalle, mientras que `renderizar(nodo)` lo hace sobre la técnica de renderización que se utilice, que para este algoritmo consiste en el teselado de la región abarcada por el nodo. En las próximas secciones se describen estas funcionalidades.

5.3 Selección de nivel de detalle

Como se abordó en el capítulo 2, existen principalmente dos criterios dependientes del punto de observación utilizados para determinar el nivel de detalle. Por un lado existe el criterio sujeto a la distancia al punto de observación, el cual asigna mayor o menor detalle dependiendo de cuán cerca la cámara se encuentre de un objeto. A menor distancia mayor resolución, y a mayor distancia menor resolución. Por otro lado se encuentra el criterio dependiente del error de aproximación proyectado en espacio de imagen. En este caso el criterio consiste en analizar el error en que se incurre al utilizar cada representación de distinto LOD y utilizar aquella que satisfaga una tolerancia del error visual percibido. Esto último se logra proyectando en espacio de imagen el error de representación y comparándolo contra un umbral de tolerancia en píxeles.

Durante el recorrido del *quadtree*, se debe decidir si los nodos visibles por la cámara poseen el nivel de detalle adecuado para su utilización en la renderización. Como ya se ha mencionado se utilizará una métrica que emplee el criterio de *error de aproximación en espacio de imagen*.

El cálculo a realizar consiste primero en determinar el error de aproximación geométrica δ del nodo actual. Luego, el error δ es proyectado en espacio de imagen obteniéndose el error visual ε. Luego, ε es comparado contra un umbral de tolerancia τ especificado en píxeles. Si el error visual ε del nodo supera la tolerancia permitida τ, es necesario mayor nivel de detalle en esa zona del terreno por lo que el nodo debe ser refinado. En caso contrario se ha encontrado un nodo con un nivel de detalle aceptable para su visualización, como se muestra en el siguiente pseudocódigo:

```
funcion posee_lod_adecuado(nodo N)
      ε = proyectar_a_pantalla(δ)
      Si ε > τ
            retornar NO
      Sino
            retornar SI
fin funcion
```

A continuación se describe el modo de calcular δ y ε.

5.3.1 Error de aproximación geométrica δ

El error de aproximación geométrica es el error que proviene de representar el terreno mediante una malla de menor resolución a la máxima disponible. Lindstrom et al. [21], definen el error de aproximación geométrica δ como la distancia (en espacio de objeto) que resulta de la diferencia entre la altura de un vértice v correspondiente a la malla de máxima resolución, y la altura que toma el terreno en ese punto cuando el vértice es removido en un nivel de detalle de menor resolución (ver figura 5.3).

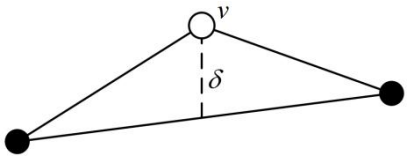

Figura 5.3: Error de aproximación geométrica δ representado como el cambio geométrico en altura δ que ocurre cuando el vértice v es removido para simplificar la malla.

En el esquema basado en *quadtree* adoptado para este trabajo, el cambio geométrico entre dos niveles de detalle consecutivos consiste en la remoción de muchos vértices simultáneamente. La solución adoptada es similar a la utilizada en Geometrical Mipmapping [3] o en el algoritmo de Röttger et al. [31]. El error de aproximación es considerado como el máximo de los errores producidos por la remoción de cada vértice: $\delta = max(\delta_0, \delta_1, \ldots, \delta_{n-1})$ con n siendo el número de vértices removidos. Se asume así el peor caso (máximo error producido) de toda la zona del terreno que cambia de nivel de detalle.

5.3.2 Proyección en espacio de pantalla ε

Disponiendo del error de aproximación geométrica δ, es realiza su proyección a espacio de imagen, para lo cual se aprovecha la proyección perspectiva. El nuevo error ε obtenido es medido en píxeles, lo que permite obtener una magnitud real del error visual percibido por el usuario ante un cambio de nivel de detalle.

Un cálculo preciso de ε es descripto por Lindstrom et al. en [21], sin embargo en algoritmos como Chunked LOD [35] y Geometrical Mipmapping [3] se han utilizado simplificaciones con el objetivo de reducir los cálculos necesarios para su obtención.

Para este trabajo se ha empleado la misma simplificación aplicada en Chunked LOD. La proyección de δ se realiza según la ecuación 5.1, siendo h_{img} la altura en espacio de imagen del plano de proyección, d es la distancia del punto de observación al bloque de terreno, y fov (*field of view*) es el ángulo de campo visual con el cual se realiza la proyección mediante la cámara virtual de la escena.

$$\varepsilon = \delta \frac{h_{img}}{2dtan\,(\frac{fov}{2})} \tag{5.1}$$

El cálculo de ε se realiza asumiendo que el punto de observación se encuentra sobre el plano horizontal y la dirección de visualización siempre es paralela al mismo. Este cálculo simplificado trae un beneficio adicional. Al ser independiente de la dirección de visualización, el nivel de detalle del terreno se mantiene estable cuando el usuario rota la cámara, variando sólo cuando el usuario traslada su posición.

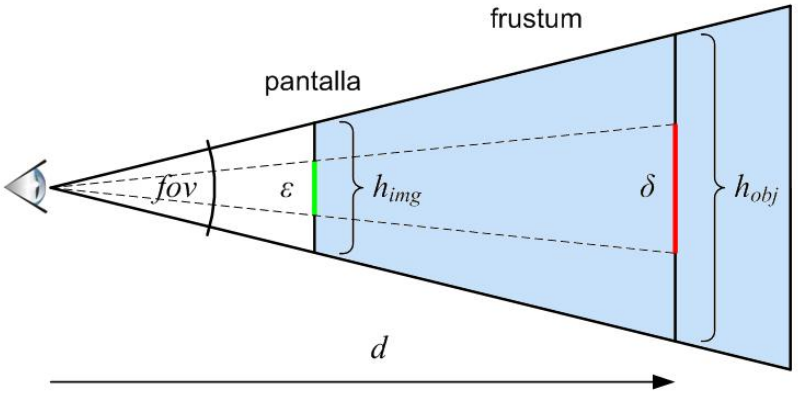

Figura 5.4: Proyección de δ en espacio de objeto, obteniéndose ε en espacio de imagen.

En la figura 5.4 se muestran los elementos que intervienen en el cálculo de ε. El segmento vertical δ a una determinada distancia d del observador es proyectado en pantalla dependiendo del ángulo de campo visual fov. En la ecuación 5.2 se aprecia como ε se obtiene a partir de δ y la relación entre h_{img} (alto de la pantalla en píxeles) y h_{obj} (proyección de h_{img} en espacio de objeto). Luego h_{obj} se obtiene mediante la ecuación 5.3.

$$\frac{\varepsilon}{h_{img}} = \frac{\delta}{h_{obj}} \quad \Rightarrow \quad \varepsilon = \delta \frac{h_{img}}{h_{obj}} \tag{5.2}$$

$$tan\,(\frac{fov}{2}) = \frac{\frac{h_{obj}}{2}}{d} \quad \Rightarrow \quad h_{obj} = 2dtan\,(\frac{fov}{2}) \tag{5.3}$$

5.4 Renderización

Durante el recorrido del *quadtree*, distintos nodos son seleccionados para su renderización. La renderización es llevada a cabo utilizando la nueva característica de teselado incorporada a Shader Model 5 (sección 3.2.4). Mediante esta característica una malla poligonal es generada para cada zona abarcada por los distintos nodos seleccionados. Además, los valores de elevación para la malla generada por el teselado son tomados directamente desde el *heightmap* almacenado en memoria de GPU. De esta manera puede aprovecharse al máximo todos los recursos de hardware, a la vez que se minimiza la intervención de la CPU en la renderización.

El nuevo tipo de primitiva disponible en Shader Model 5 es el parche o *patch*. Existes tres tipos de parches: las líneas, los triángulos, y los cuadriláteros o *quads*. Para esta técnica se han utilizado los *quads* ya que coinciden perfectamente con la estructura de división cuadrada producida por los *quadtrees*. Así un parche de tipo *quad* es ubicado sobre la zona del terreno que abarca el nodo a renderizar. Luego el parche es teselado generando la malla de cada zona particular.

Para el teselado de un parche es necesario especificar los factores de teselado que indican al hardware en cuantos pedazos subdividir la primitiva. En esta técnica pueden utilizarse factores de la forma 2^n, con $n \geq 1$, pero siempre el valor elegido debe ser fijo para todos los nodos del *quadtree* (ver figura 5.5), lo que permite conseguir la representación multirresolución de la seción 5.2.

Por último, los vértices de la malla generada deben ser trasladados mediante *shader*. Un desplazamiento vertical respecto del plano del terreno debe realizarse mediante el acceso a las muestras de altura almacenadas en un recurso de tipo textura. Horizontalmente la posición de los vértices también debe ser modificada según un valor configurable de separación entre muestras, y trasladados según su posición relativa en el *quadtree*.

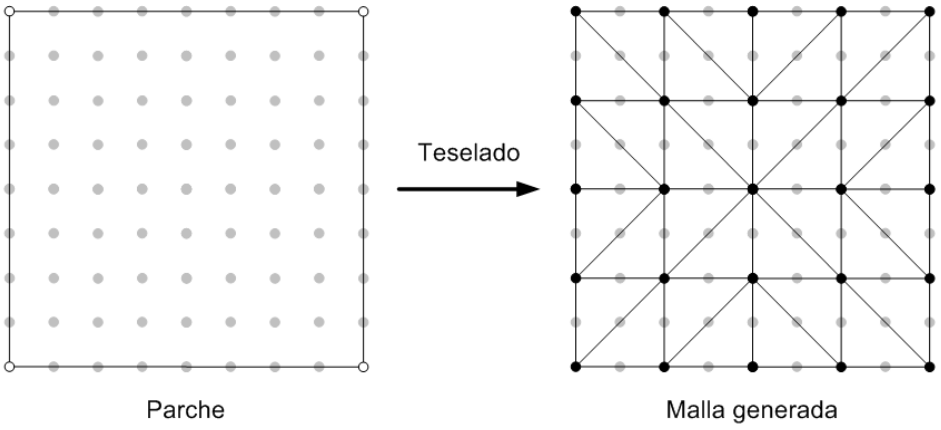

Figura 5.5: Un parche definido por los 4 puntos de control en blanco, y que abarca una zona de 9x9 muéstras, es teselado con un factor igual a 4. En este caso se trata de un nodo perteneciente al anteúltimo nivel del quadtree ya que todavía quedan muestras sin ser consideradas en la malla generada.

5.5 Prevención de grietas entre bloques

Un inconveniente que puede introducir la representación del terreno mediante bloques (o grillas anidadas) de distinto nivel de detalle, es la aparición grietas (sección 2.4.4). Estas discontinuidades pueden manifestarse entre dos bloques de distinto nivel de detalle, al no emplear los mismos vértices a lo largo del borde que comparten entre sí.

Autores como De Boer et al. [3] y Larsen et al. [20], han propuesto resolver el problema realizando modificaciones en los bordes de los bloques mediante la reconexión de vértices de uno de los dos bloques adyacentes entre sí. Por otro lado, con la introducción de la última versión de Shader Model (SM5), hoy también es posible realizar la adaptación de bordes por GPU mediante los factores de teselado externos (sección 3.3.3, *"Hull Shader"*).

Sin embargo, estas técnicas requieren conocer el nivel de detalle de los bloques vecinos de cada bloque. Para esto, se debería por ejemplo realizar la búsqueda de los vecinos de un bloque dentro del *quadtree* o la lista de nodos a renderizar, lo que haría más complejo el algoritmo. Por lo tanto, se decidió mantener la técnica original introducida en Chunked LOD [35], la que consiste en utilizar geometría adicional alrededor de cada bloque de terreno. Esta geometría adicional se denomina "polleras" (*skirts*), ya que consisten en triángulos verticales que comienzan en los bordes de un bloque y se prolongan hacia abajo por una determinada extensión (ver figura en la sección 4.3.2). Las polleras constituyen un método simple y efectivo de resolver el problema de las grietas, y son similares a los "triángulos de área cero" utilizados en Geometry Clipmaps [24]. Además, es posible utilizar primitivas de tipo parche (SM5) para generar el mallado de las mismas.

5.6 Geomorphing

Otro inconveniente relacionado a la representación multirresolución de terrenos, es el efecto denominado *popping* (sección 2.4.4). Este efecto *c*onsiste en la visualización de cambios abruptos de altura en la superficie del terreno, que se producen al modificar el mallado de una zona, o al reemplazar un bloque de terreno por otro de distinto nivel de detalle.

En la sección 5.3 se describió cómo es posible seleccionar el nivel de detalle manteniendo la percepción del error por debajo de un cierto umbral τ. Esto significa que los *pops* no excederán el umbral τ en píxeles seleccionado, ya que el mayor *pop* posible para un bloque es igual a δ. Utilizando un valor pequeño de τ, produciría que los *pops* no sean muy notorios ante un cambio de nivel de detalle. Si se asigna $\tau=1$, es posible hacerlos desaparecer por completo ya que el mayor *pop* no será mayor a un píxel.

Al mismo tiempo, utilizar valores pequeños para τ puede producir la selección de bloques muy refinados. En cambio, si se emplean valores mayores para el umbral, los bloques a renderizar serán de menor resolución, y se alcanzaría una mejor performance. Ulrich [35] considera valores usuales de $\tau<5$, mientras que De Boer [3] utiliza $\tau=8$ para realizar sus pruebas de performance. En todo caso, existe un *tradeoff* entre performance y calidad visual, por lo que el valor específico de τ dependerá de cada aplicación en particular.

De esta manera, y con el objetivo de mejorar la performance a la vez de poder solucionar el problema del *popping* que se produce al utilizar valores de $\tau > 1$, es necesario aplicar la técnica denominada *geomorphing* (sección 4.1.2), es decir, una transición suavizada entre las representaciones de distinto nivel de detalle.

Hasta la versión 4 del Shader Model, el único modo posible de implementar el *geomorphing* en la GPU, era manipulando los vértices de la malla mediante operaciones de traslación realizadas en el *vertex shader* [38]. En el presente trabajo se han aprovechado las nuevas características de Shader Model 5 para poder implementar esta técnica. En Shader Model 5 es posible realizar teselados "transicionales" mediante el uso de factores de teselado fraccionales (sección 3.2.4). Los factores de teselado fraccionales son ideales para realizar las transiciones directamente por hardware (GPU) y de manera sencilla ya que no es necesario el desarrollo de *shaders* específicos para la tarea. Además se adaptan de forma natural al uso de la nueva primitiva de tipo parche, permitiendo realizar transiciones entre dos teselados correspondientes a niveles de detalle consecutivos, como es el caso de este trabajo.

El método o principio de *GeoMipMapping Trilineal* propuesto por De Boer [3], puede ser empleado en este trabajo para conseguir una transición suavizada entre bloques de nivel de detalle consecutivos. Este método requiere conocer dos distancias para cada nodo. Por un lado la distancia s que indica a que distancia el nodo debería ser subdividido en sus cuatro nodos hijo de mayor detalle. A su vez, cada nodo debe conocer la distancia u a la que el mismo debe ser "unificado" junto a sus hermanos y reemplazados por el nodo padre. En la figura 5.6 se esquematizan estas distancias para un nodo determinado, junto a la distancia d entre el punto de observación y un bloque.

Al calcular la fracción entre los segmentos a y b medinate la ecuación 5.4, es posible obtener el peso o fracción que permite realizar una interpolación lineal entre s y u. Esta interpolación lineal es utilizada para efectuar la transición suavizada, que para este trabajo consiste en la transición entre dos factores de teselado enteros.

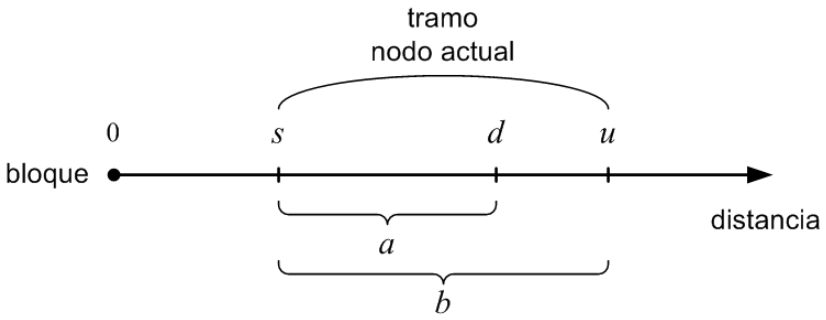

Figura 5.6: Relación entre la distancia de subdivisión s, distancia de unificación u, y la distancia d entre el observador y el bloque de terreno.

La transición entre el teselado del bloque actual y el teselado de los bloques de mayor detalle se realizará de forma incremental y continuará hasta que la distancia d alcance a la distancia s, en cuyo momento el bloque actual será reemplazado por sus 4 bloques hijo de mayor detalle. Esto brindará una transición suavizada entre teselados para cualquier valor de d, eliminando los *pops* gracias a la utilización de los factores de teselado fraccionales de Shader Model 5.

$$f = \frac{a}{b} = \frac{d - s}{u - s} \tag{5.4}$$

$$\tau = \varepsilon \quad \Rightarrow \quad \tau = \delta \frac{h_{img}}{2dtan\left(\frac{fov}{2}\right)} \quad \Rightarrow \quad d = \delta \frac{h_{img}}{2\tau tan\left(\frac{fov}{2}\right)} = s \tag{5.5}$$

Por último, es necesario calcular las distancias s y u. En primer lugar, como se ha mencionado la distancia s de un nodo es la distancia a la que el nodo es subdividido en sus cuatro hijos, y esa es la distancia a la que ε se equipara con τ, como se indica en la ecuación 5.5. Por otro lado, dado que la distancia a la que un nodo se divide es la misma a la que sus hijos se unifican, la distancia u es igual a la distancia s del padre del nodo actual. Así en el recorrido recursivo del *quadtree*, sólo es necesario enviar por parámetro la distancia s hacia los nodos hijo cuando un nodo es subdividido. Por último, la raíz carece de distancia u, la cual es considerada como $2s$.

6 IMPLEMENTACIÓN Y PRUEBA

Este capítulo describe los detalles de implementación y los resultados obtenidos del algoritmo de visualización de terrenos descripto en el capítulo anterior.

El capítulo comienza presentando los aspectos más relevantes sobre la implementación del algoritmo (sección 6.1). En la sección 6.2, se detallan los aspectos relacionados a la creación y almacenamiento del *quadtree*.En la sección 6.3 se presenta la técnica de renderización mediante *shaders*.

El capítulo finaliza exponiendo los resultados obtenidos a partir de un conjunto de pruebas realizadas (sección 6.4).

6.1 Acerca de la implementación

6.1.1 Preprocesamiento y visualización

Como se ha descripto en el capítulo 5 (diseño del algoritmo), el proceso de selección de nivel de detalle requiere contar con errores de aproximación geométrica en cada nodo del *quadtree*. Como es necesario llevar a cabo el análisis de las muestras de altura contenidas en un *heightmap*, los coeficientes no pueden ser calculados en tiempo real. La solución adoptada, al igual que en los algoritmos estudiados para este trabajo, es realizar el cálculo de los coeficientes de error en una etapa previa a la visualización. De esta manera, la implementación del algoritmo ha sido divida en dos aplicaciones: la aplicación principal de visualización o *Visualizador*, y una aplicación auxiliar de preprocesamiento o *Preprocesador*.

Comenzando por el preprocesador, éste se encarga de tomar un *heightmap* como entrada, realizar el análisis de las muestras contenidas en el mismo, generar el *quadtree* junto a los coeficientes de error de aproximación para cada nodo, y finalmente almacenar todos los datos en un archivo en disco para su posterior utilización. Luego, el visualizador funciona como la aplicación principal del sistema ya que se encarga de realizar la visualización propiamente dicha. Para esto, carga desde disco el archivo generado en el preprocesamiento (junto al *heightmap* que se utilizó para la generación del mismo), y finalmente renderiza el terreno en pantalla.

6.1.2 Detalles de implementación

Ambas aplicaciones, Preprocesador y Visualizador, fueron desarrolladas en el lenguaje de programación C++. En el apéndice A pueden consultarse la documentación de clases de la implementación realizada, junto a la descripción de las responsabilidades de cada clase y las operaciones que cada una realiza.

Para el acceso y configuración del pipeline de la GPU en la aplicación de visualización, se ha utilizado la librería DirectX en su versión 11. Esta versión fue desarrollada especialmente para soportar Shader Model 5 [40]. Además se ha incluido la librería utilitaria DXUT, que brinda un conjunto básico de controles gráficos (slider, check box, static text, etc.). El uso de estos controles ha permitido implementar la modificación de los parámetros de visualización de forma interactiva mientras se renderiza el terreno en pantalla, sin la necesidad de recompilar el código o reiniciar la aplicación.

6.2 Preprocesamiento

En la figura 6.1 se muestra un esquema general del preprocesamiento, donde se indican las principales tareas realizadas y su interacción:

- Carga del *heighmap*
- Construcción del *quadtree*
- Cálculo de errores de aproximación geométrica
- Corrección de coeficientes
- Almacenamiento en disco

En las siguientes secciones se describen las distintas tareas.

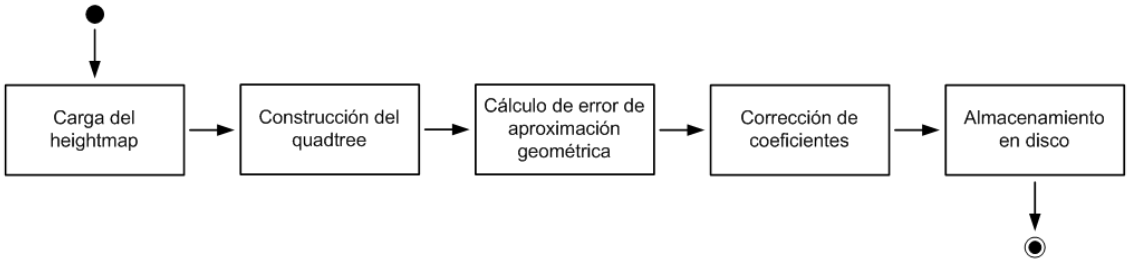

Figura 6.1: Esquema general de trabajo del preprocesador.

6.2.1 Carga del heightmap

Para la carga y manipulación de *heightmaps* en el preprocesamiento, se ha utilizado la librería libre FreeImage [9]. Esta librería soporta trabajar con grandes imágenes y además permite el acceso directo a los buffers de píxeles (muestras de altura) para mayor eficiencia.

6.2.2 Construcción del quadtree

Los nodos del *quadtree* son creados comenzando por la raíz y continuando por los hijos de forma recursiva mientras se desciende por el árbol. Al crear cada nodo, se le asigna un marco definido por dos coordenadas (min_x, min_y), (max_x, max_y). Este marco

indica la zona del mapa de altura (es espacio de muestras) que el nodo abarca (un cuarto de zona de su nodo padre, compartiendo una fila y columna del borde con los nodos hermanos). La profundidad del árbol dependerá del tamaño del *heightmap* utilizado. El proceso recursivo continúa hasta conformar nodos de 65x65 muestras (máximo factor de teselado soportado en Shader Model 5). Sobre el marco de muestras de cada nodo creado, se calcula el error de aproximación geométrica y se lo asocia al mismo.

6.2.3 Cálculo de error de aproximación geométrica

El cálculo del error de aproximación de cada nodo, se realiza sobre las muestras abarcadas (marco de muestras) de cada nodo. A su vez, el marco de muestras es subdividido en bloques contiguos de 65 x 65 muestras (solapados por una fila y columna). Dos triángulos (por diagonal izquierda o derecha) pueden formarse a partir de las muestras de las 4 esquinas de cada bloque. Este cálculo de alturas puede realizarse sobre uno u otro triángulo. La elección de qué triángulo utilizar está supeditada a la topología de la triangulación formada por el teselador de Shader Model 5, por lo que su elección depende del cuadrante al que el bloque pertenezca. Luego, para cada muestra subyacente a cada bloque, se calcula la distancia entre la muestra y el plano formado por el triángulo seleccionado. La máxima distancia encontrada para todos los bloques corresponde al error de aproximación geométrica.

Además, este proceso de iteración entre muestras es aprovechado para obtener los valores de menor y mayor altura entre todas las muestras analizadas. Luego estos valores son utilizados para construir la caja envolvente que permitirá realizar el test de intersección con el *frustum* de la cámara.

6.2.4 Corrección de coeficientes

A medida que se desciende por el *quadtree* los nodos poseen mayor resolución, por lo que es de esperarse que los errores de aproximación geométrica sean cada vez más pequeños. Sin embargo, y debido a utilizar triangulaciones regulares, algunas veces esta condición no se cumple. Como resultado se obtiene que los nodos en estas condiciones son salteados durante la selección del nivel de detalle. Esto afecta la transición suavizada entre niveles de detalle, lo que puede degradar la calidad visual.

Afortunadamente esta condición ha sido poco probable, ya que se ha dado en muy pocos casos respecto del número total de nodos en un *quadtree* (menor al 1%). En la sección 6.4.1 se muestran los datos precisos sobre los tres *heightmaps* utilizados en las pruebas. Dado el bajo número de nodos se decidió implementar como solución un proceso de corrección de coeficientes.

El proceso de corrección de coeficientes consiste en un recorrido adicional del *quadtree* luego de su construcción. En caso de encontrar un nodo en las condiciones mencionadas, es decir la de poseer un coeficiente menor al de su nodo padre, éste es

corregido. La corrección consiste en obtener un valor intermedio entre los dos niveles de detalle superior e inferior del nodo inconsistente. El nuevo coeficiente se calcula como el promedio entre el coeficiente del nodo padre y el mayor coeficiente entre los nodos hijo.

Finalmente, el *quadtree* corregido es almacenado en disco.

6.2.5 Almacenamiento del quadtree

Una vez construido, el *quadtree* es almacenado en disco para su posterior carga y utilización por el visualizador.

Se han utilizado archivos binarios, y el formato empleado consiste en una cabecera seguida de los datos de cada nodo. Los nodos se almacenan en preorden, por lo que es posible reconstruir el *quadtree* sin necesidad de almacenar referencias entre nodos.

La tabla 6.1 especifica el formato de archivo de salida. La extensión adoptada para el nombre de archivo es ".quad".

Cabecera de archivo		
Descripción	***Tipo***	***Cantidad***
Numero de versión de archivo	int (32 bits)	1
Extensión y resolución de pixel	float (32 bits)	2
Numero de nodos del quadtree	int (32 bits)	1
Nombre de archivo de heightmap		
Largo cadena	int (32 bits)	1
Cadena de caracteres	widechar (16 bits)	Largo cadena
Nombre de archivo de colormap		
Largo cadena	int (32 bits)	1
Cadena de caracteres	widechar (16 bits)	Largo cadena
Información por nodo		
Descripción	***Tipo***	***Cantidad***
Indicador de nodo hoja (1 es hoja, 0 es nodo interno)	Int (32 bits)	1
Caja envolvente (en espacio de objeto)		
Coordenadas (x, y, z) esquina mínima	float (32 bits)	3
Coordenadas (x, y, z) esquina máxima	float (32 bits)	3
Zona abarcada (en espacio de muestras)		
Coordenadas (x, y) esquina mínima	int (32 bits)	2
Coordenadas (x, y) esquina máxima	int (32 bits)	2
Error de aproximación geométrica	float (32 bits)	1

Tabla 6.1: Formato de archivos ".quad".

6.3 Visualización

En la figura 6.2 se muestra un esquema general del visualizador, donde se indican las principales tareas realizadas y su interacción:

- Carga del *quadtree* y *heightmap*
- Definición de la geometría
- Captura de entrada
- Actualización del punto de observación
- Recolección de nodos
- Ejecución del *pipeline*

En las siguientes secciones se describen las distintas tareas.

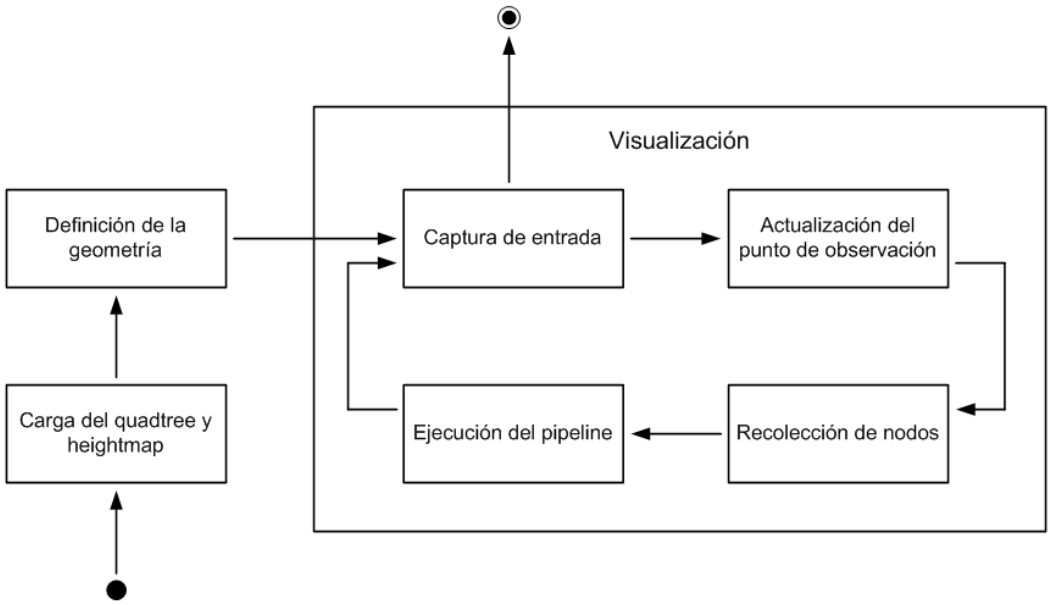

Figura 6.2: Esquema general de trabajo del visualizador.

6.3.1 Carga del quadtree y heightmap

El *quadtree* es cargado en memoria a partir de un archivo ".quad" bajo el formato descripto en la tabla 6.1.

Los mapas de bits asociados (*heightmap* y *colormap*) son cargados y subidos a memoria de GPU mediante la API (*Application User Interface*) de DirectX11.

6.3.2 Definición de la geometría

La renderización del terreno se realiza utilizando una misma geometría, con la cual se define un bloque genérico de terreno. Luego, ese bloque es adaptado para cada zona

del terreno que corresponde renderizar.

La creación del bloque genérico se realiza mediante 8 vértices (o puntos de control) y 5 primitivas tipo quad que luego serán teseladas. En la figura 6.3, se muestra la definición de los vértices y primitivas. De las 5 primitivas, 1 corresponde a la superficie del bloque de terreno y las 4 restantes para formar las polleras a los lados del bloque.

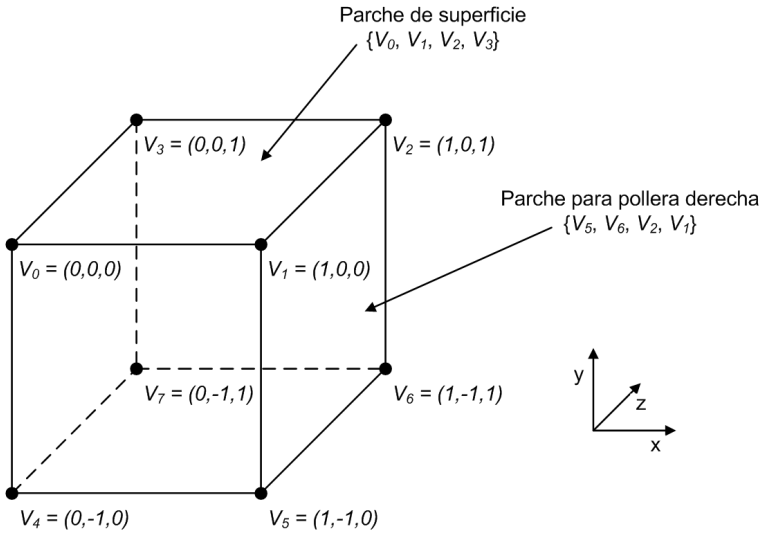

Figura 6.3: Geometría utilizada para representar cada bloque de terreno.

Mediante esta forma de definición de la geometría fue posible una implementación de *shaders* (sección 6.3.6) que permite:

- La identificación en el *Hull shader* de las primitivas para polleras, de modo de poder aplicar el teselado apropiado (distinto del teselado de superficie). Esto significa que no es necesario la utilización de un *shader* específico para esta tarea y su constante reemplazo en el *pipeline* de la GPU (performance).
- Controlar el largo de las polleras de forma transparente en el *Domain Shader* (sin utilizar condicionales), independientemente de si se trata de la primitiva de superficie o de las primitivas para polleras.
- Utilizar las posiciones de los vértices para obtener las coordenadas de textura del mapa de color y del mapa de alturas (*heightmap*), evitando la utilización de parámetros adicionales en la definición de cada vértice.

6.3.3 Captura de entrada

La interactividad de la aplicación depende de la reacción a los eventos de entrada del usuario. Para esto, se captura la entrada de teclado como el movimiento del mouse en pantalla a partir de la API (*Application User Interface*) de Windows.

6.3.4 Actualización del punto de observación

La posición y dirección visual del punto de observación de la escena, son actualizados mediante la entrada de teclado y movimiento del mouse respectivamente.

Las matrices de transformación de vista y proyección se calculan a partir de funciones de ayuda de la API (*Application User Interface*) de DirectX11. Además, el *frustum* de la cámara es modelado mediante 6 planos que envuelven el espacio de visualización del mismo. Estos planos son extraídos de las matriz vista-proyección como indica Gribb et al. [12].

6.3.5 Recolección de nodos

El proceso de renderización ha sido dividido en dos etapas: la recolección de nodos del *quadtree* junto a datos asociados, y la renderización propiamente dicha de cada bloque de terreno mediante esos datos recolectados.

El recorrido del *quadtree* se realiza recursivamente como se ha explicado el diseño (sección 5.2.2). A medida que se recorre el *quadtree*, los nodos seleccionados para renderización son recolectados en una lista junto a 3 datos adicionales. Estos datos son utilizados luego por los *shaders* para la renderización del bloque genérico en cada zona específica del terreno. Los datos recolectados junto a los nodos son:

- ***Offset***: Consiste en una coordenada 2D respecto del plano del terreno. Indica el desplazamiento del bloque a su posición final respecto vértice V_0.
- ***Escala***: Es el factor de escala de LOD para el bloque. Posee rango [0..1] e indica el tamaño a abarcar por el nodo relativo al tamaño del terreno. El nodo raíz se corresponde con un factor igual a 1 ya que abarca todo el terreno. Los hijos con un factor de 0.5, los nietos 0.25 y así sucesivamente.
- ***Teselado***: Es el factor de teselado fraccional que da lugar al *geomorphing*. Se calcula como se ha descripto en la sección 5.6.

6.3.6 Ejecución del pipeline

La técnica de renderización se completa mediante la ejecución de cuatro *shaders* en el *pipeline* de la GPU: *vertex shader, hull shader, domain shader y pixel shader.*

Los *shaders* han sido desarrollados en el lenguaje de programación HLSL por ser el lenguaje con mejor soporte en DirectX.

6.3.6.1 Comunicación CPU-GPU

Los parámetros a comunicar entre CPU y GPU han sido agrupados en estructuras denominadas buffers de constantes (cbuffer). Debido a una cuestión de performance, es conveniente realizar la agrupación de manera que los parámetros coincidan en el

tiempo o situación en que son modificados, lo que permite minimizar el número y tamaño de actualizaciones realizadas. Para esta técnica de renderización se han empleado 3 buffers de constantes. El buffer ***cbEventual*** contiene los parámetros que varían muy rara vez, o una sola al inicio de la ejecución. El buffer ***cbPorCuadro*** contiene la matriz que combina las transformaciones de modelo, vista y proyección, la cual se actualiza una vez al comienzo de cada cuadro. Por último, el buffer ***cbPorBloque*** contiene todos los datos que son necesarios para ubicar y teselar cada bloque de terreno (ver figura 6.4)

```
//--------------------------------------------------------------------------
// Buffers de constantes
//--------------------------------------------------------------------------

// Muy baja frecuencia de actualización.
cbuffer cbEventual : register( b0 )
{
  float cExtHorizontal : packoffset( c0.x ); // Espaciado entre muestras.
  float cResVertical   : packoffset( c0.y ); // Resolución de cada muestra.
  float cLargoPolleras : packoffset( c0.z ); // Largo de las polleras.
}

// Se actualiza al inicio de cada cuadro.
cbuffer cbPorCuadro : register( b1 )
{
  matrix cMatWorldViewProjection : packoffset( c0 ); // Matriz que combina las
                                                     // matrices de modelo,
                                                     // vista y proyección.
}

// Se actualiza una vez por bloque de terreno.
cbuffer cbPorBloque : register( b2 )
{
  float2 cOffset  : packoffset( c0 );   // Desplazamiento final del bloque.
  float cEscala   : packoffset( c0.z ); // Factor de escala del bloque.
  float cTeselado : packoffset( c0.w ); // Factor de teselado fraccional de los
                                        // parches.
};
```

Figura 6.4: Definición de buffers de constantes.

6.3.6.2 Vertex shader

El *vertex shader* envía los datos de entrada (vértices) a las etapas de teselado. En esta técnica de renderización la transformación de los vértices se realiza en el *domain shader*. Como esta etapa del *pipeline* no es opcional, se debe proveer un *vertex shader* por defecto, es decir que realiza simplemente la copia de los véctices de entrada en la salida. La figura 6.5 muestra el *vertex shader* junto a la estructura de datos que utiliza como entrada. La estructura contiene solamente la posición del vértice, como han sido definidos los vértices del bloque genérico de terreno.

```
//--------------------------------------------------------------------------------
// Estructuras de entrada y salida
//--------------------------------------------------------------------------------
// Define los atributos de los puntos de control.
struct CONTROL_POINT
{
  float3 Position : POSITION0; // Posición en espacio de objeto.
};

//--------------------------------------------------------------------------------
// Vertex Shader
//--------------------------------------------------------------------------------
// Vertex shader por defecto: los puntos de control no son transformados.
CONTROL_POINT VSMain( CONTROL_POINT Input )
{
  CONTROL_POINT output = (CONTROL_POINT)0;

  // copia la entrada en la salida
  output.Position = Input.Position;

  return output;
}
```

Figura 6.5: Vertex shader y su estructura de datos de entrada.

6.3.6.3 Hull shader

El *hull shader* se compone de dos funciones: la función constante y la función principal.

La figura 6.6 muestra la función constante del *hull shader* y su estructura de datos de salida. La estructura consiste en 4 factores de teselado exteriores y 2 interiores, necesarios para el teselado de los parches tipo *quad* utilizados en esta técnica. La función constante establece los factores de teselado dependiendo del tipo de parche de entrada. Para el caso del parche principal (superficie), se utiliza el mismo número de cortes para ambas direcciones. En el caso de las polleras, los cortes se realizan mayoritariamente en sentido vertical para acompañar la topografía de los lados del parche de superficie. El factor de teselado ***cTeselado*** es calculado en CPU (sección 6.3.5), y subido a la GPU en el buffer de constantes ***cbPorBloque***.

```
//--------------------------------------------------------------------------------
// Estructuras de entrada y salida
//--------------------------------------------------------------------------------
// Define los atributos de salida de la función constante del hull shader para //
un parche tipo quad. No se utilizan atributos adicionales a los factores de
// teselado.
struct HS_CONSTANT_OUTPUT
{
  float TeseExt[4] : SV_TessFactor;       // Factores de teselado exteriores.
  float TeseInt[2] : SV_InsideTessFactor; // Factores de teselado interiores.
};

//--------------------------------------------------------------------------------
// Hull Shader
```

```
//---------------------------------------------------------------------------
// HS: Función constante. Calcula los factores de teselado para cada parche
HS_CONSTANT_OUTPUT HSPatchConstant( InputPatch<CONTROL_POINT, 4> Input,
                                    uint PrimitiveId : SV_PrimitiveID )
{
  HS_CONSTANT_OUTPUT output = (HS_CONSTANT_OUTPUT)0;

  // parche para superficie
  if(PrimitiveId == 0)
  {
    // todo el parche utiliza el mismo factor de teselado
    output.TeseExt[0] = output.TeseExt[1] = cTeselado;
    output.TeseExt[2] = output.TeseExt[3] = cTeselado;
    output.TeseInt[0] = output.TeseInt[1] = cTeselado;
  }

  // parches para polleras
  else
  {
    // por orden de los puntos de control en la definición de la primitiva:

    // ... el lado 0 y 2 siempre son los bordes laterales de la pollera,
    // se utiliza el factor mínimo posible en teselado fraccional par: 2.
    output.TeseExt[0] = output.TeseExt[2] = output.TeseInt[1] = 2.0f;

    // ... el lado 1 y 3 siempre son los bordes superior e inf. de la pollera,
    // se utiliza un factor igual al de superficie de modo que coincidan.
    output.TeseExt[1] = output.TeseExt[3] = output.TeseInt[0] = cTeselado;
  }

  return output;
}
```

Figura 6.6: Función constante del hull shader.

La figura 6.7 muestra la función principal del *hull shader*. En el encabezado de esta función se realiza la configuración del teselador. El dominio a teselar son primitivas tipo *quad*. Los factores de teselado se interpretan de forma fraccional par para el suavizado del teselado evitando los *pops*. Además, la función principal no se agrega nuevos puntos de control, por lo que su número de salida se mantiene en 4 (*quads*).

En esta técnica de renderización, la transformación de los vértices se realiza en el *domain shader*, por lo que la función principal del *hull shader* no modifica el parche antes de ser teselado, simplemente tiene como salida las cuatro esquinas del *quad*.

```
//---------------------------------------------------------------------------
// Hull Shader
//---------------------------------------------------------------------------
// HS: Función principal. No realiza cambios en los puntos de control.
[domain("quad")]                        // Se debe teselar primitiva tipo quad.
[partitioning("fractional_even")]       // Realizar teselado fraccional par.
[outputtopology("triangle_ccw")] // Primitiva de salida: triángulo antihorario.
[patchconstantfunc("HSPatchConstant")] // Nombre de la función constante.
```

```
[outputcontrolpoints(4)]              // Número de ejecuciones de la función.
CONTROL_POINT HSMain( InputPatch<CONTROL_POINT, 4> Input,
                      uint PointId : SV_OutputControlPointID )
{
  CONTROL_POINT output = (CONTROL_POINT)0;

  // replica cada esquina del quad en la salida
  output.Position = Input[PointId].Position;

  return output;
}
```

Figura 6.7: Función principal del hull shader.

6.3.6.4 Domain shader

El *domain shader* se encarga de calcular la posición de cada vértice creado por el teselador en base a los datos suministrados por el *hull shader*.

La figura 6.8 muestra el *domain shader* y su estructura de datos de salida la cual consiste en la posición final del vértice y la coordenada de texturas del mapa de color utilizada en el *pixel shader*.

Inicialmente, como el teselador devuelve las coordenada de teselado en coordenadas normalizadas [0, 1], la posición del vértice en espacio de objeto se calcula realizando la interpolación lineal entre las cuatro esquinas del parche, y transformándola mediante un escalado ***cEscala*** y la translación ***cOffset***. Estos facores son calculados en CPU (sección 6.3.5), y subidos a la GPU en el buffer de constantes ***cbPorBloque***.

Seguidamente, las coordenadas de textura de ambos mapas de color y altura son calculadas de forma precisa en base a la dimensión de ambos mapas, y la posición 2D del vértice respecto del plano del terreno.

Luego, la posición final del vértice en espacio se calcula de la siguiente manera. La coordenada x y z son escaladas por la extensión horizontal del terreno. La coordenada y se calcula como la altura muestreada del mapa de alturas por la resolución vertical y desplazada hacia abajo respecto del plano del terreno en caso de tratarse de un vértice de pollera.

Por último, la posición del vértice devuelto por el *domain shader* (rasterización) es transformado por la matriz de vista y proyección dejando al vértice en coordenadas normalizadas.

```
//--------------------------------------------------------------------------------
// Estructuras de entrada y salida
//--------------------------------------------------------------------------------
// Define los atributos de salida del domain shader
struct DS_OUTPUT
{
  float2 Texcoord : TEXCOORD0;   // Coordenadas de textura para obtener el
```

```
                                    // color de la superficie desde el colormap.
  float4 Position : SV_POSITION; // Posición del vértice en coordenadas
                                    // homogéneas.
};

//--------------------------------------------------------------------------------
// Domain Shader
//--------------------------------------------------------------------------------
[domain("quad")]
DS_OUTPUT DSMain(HS_CONSTANT_OUTPUT HSConstantData,
                 const OutputPatch<CONTROL_POINT, 4> Input,
                 float2 coordsTese : SV_DomainLocation )
{
  DS_OUTPUT output = (DS_OUTPUT)0;
  float2 heightmapSize, colormapSize;
  float3 posicion;

  // posicion obtenida interpolando de los vertices del parche
  float3 lejano = lerp(Input[0].Position, Input[1].Position, coordsTese.x);
  float3 cercano = lerp(Input[3].Position, Input[2].Position, coordsTese.x);
  float3 posCorte = lerp(lejano, cercano, coordsTese.y);

  // posiciona 2D del parche en espacio de objeto
  float2 pos2D = (posCorte.xz * cEscala) + cOffset;

  // obtiene dimensiones de ambos mapas
  texHeightmap.GetDimensions(heightmapSize.x, heightmapSize.y);
  texColormap.GetDimensions(colormapSize.x, colormapSize.y);

  // calcula las coordenadas corregidas para el mapa de altura y color
  float2 heightmapCoord = pos2D*(heightmapSize-1.0f)/heightmapSize +
                          0.5f/heightmapSize;
  float2 colormapCoord = pos2D*(colormapSize-1.0f)/colormapSize +
                         0.5f/colormapSize;

  // muestrea el mapa de alturas
  float altura = texHeightmap.SampleLevel(samHeightmap, heightmapCoord, 0).x;

  // posición en espacio de escena
  posicion.x = pos2D.x * cExtHorizontal;
  posicion.y = (altura * cResVertical) + (posCorte.y * cLargoPolleras);
  posicion.z = -pos2D.y * cExtHorizontal;

  // posición y coordenadas de textura normalizadas.
  output.Position = mul(float4(posicion, 1), cMatWorldViewProjection);
  output.Texcoord = colormapCoord;

  return output;
}
```

Figura 6.8: Domain shader.

6.3.6.5 Pixel shader

Finalmente, el *pixel shader* obtiene el color candidato de cada pixel desde el mapa de color y lo retorna para su mezcla en el render-target. La figura 6.9 muestra el *pixel*

shader junto a la estructura de datos utilizada como entrada. El único atributo incluido en la estructura es la coordenada de textura producida en el *domain shader*, la posición del fragmento no es necesaria.

```
//--------------------------------------------------------------------------------
// Estructuras de entrada y salida
//--------------------------------------------------------------------------------
// Define los atributos de entrada del pixel shader. Es la rasterización de la
// salida del domain shader.
struct PS_INPUT
{
  float2 Texcoord : TEXCOORD0; // Coordenadas de textura para obtener el
                               // color de la superficie desde el colormap.
};

//--------------------------------------------------------------------------------
// Pixel Shader
//--------------------------------------------------------------------------------
// Genera el color del fragmento a mezclar en el render-target.
float4 PSMain( PS_INPUT Input ) : SV_TARGET
{
  // obtiene el color del fragmento desde el colormap
  return texColormap.Sample(samColormap, Input.Texcoord);
}
```

Figura 6.9: Pixel shader.

6.4 Resultados obtenidos

6.4.1 Datos de prueba

Tres conjuntos de datos han sido utilizados para llevar a cabo las pruebas. El primero de ellos es conocido en la bibliografía como *Puget Sound* [28]. Este terreno ha sido utilizado inicialmente por [22] y luego por muchos otros autores para la evaluación de sus algoritmos, ya que posee regiones con distribuciones de altura muy características: un gran valle central rodeado de zonas montañosas. El segundo terreno de prueba es *Hawaii* [14] el cual consiste en una gran isla-montaña central rodeada de mar a altura cero. Por último, el tercer terreno ha sido generado artificialmente para la prueba del algoritmo. Este terreno fue creado mediante la herramienta de definición y generación procedural de *heightmaps* L3DT [19], y consiste en una gran pradera con picos montañosos y hondonadas con distribución relativamente pareja sobre la superficie.

Cada conjunto de datos está compuesto de un mapa de alturas (*heightmap*) y una textura que se mapea sobre la superficie (*colormap*), ambos de la misma dimensión. Con el objetivo de analizar el comportamiento del algoritmo ante las mismas condiciones, todos los conjuntos de datos poseen las mismas dimensiones. Estas son de 4097x4097 muestras de lado en las que *Puget Sound* se encuentra disponible. En el caso de *Hawaii* fue necesario su remuestreo, y el terreno artificial fue generado

directamente en estas dimensiones. La separación entre muestras utilizada es de 10 metros, lo que constituye una extensión de 40.96 x 40.96 km, mientras que los mapas de altura corresponden a imágenes en formato PNG en escala de grises de 8 bits con una resolución vertical de 12.8 metros. En total, cada uno de estos mapas genera, a partir de las 16.8 millones de muestras que los conforman, un mallado a máxima resolución de 33.6 millones de triángulos.

Estos datos fueron introducidos al preprocesador produciendo tres *quadtrees* respectivamente, uno para cada conjunto de datos. Con las extensiones mencionadas (4097 muestras de lado) se obtuvieron *quadtrees* (árboles cuaternarios) de 7 niveles y 5461 nodos. En la tabla 6.2 se muestran los resultados obtenidos. Como fue anticipado en la sección 6.2.4, el número de nodos con coeficientes corregidos es muy bajo respecto del número total de nodos (menor al 1%). Este número puede variar por múltiples factores (topología propia del terreno, error de punto flotante, etc.). El bajo número obtenido implica una mejor visualización, sin saltos en la selección del nivel de detalle.

Terreno	***Coeficientes corregidos/ Nodos en el quadtree***	***Porcentaje corregidos***
Puget Sound	11/5461	0.20%
Hawaii	24/5461	0.44%
Artificial	2/5461	0.04%

Tabla 6.2: Resultado del preprocesamiento de los conjuntos de datos utilizados en las pruebas.

6.4.2 Evaluación cualitativa

El algoritmo desarrollado posibilita el ajuste de la calidad visual generada mediante la variación de un umbral de tolerancia de error en píxeles. Para poder hacer una evaluación final del algoritmo, es necesario contar con una apreciación cualitativa de la calidad visual lograda por el mismo.

En este sentido y a modo ilustrativo, las figuras 6.10 y 6.11 muestran el mallado de los bloques de terreno utilizando umbrales distintos de tolerancia de error sobre el terreno de prueba artificial. En la figura 6.10 se observa que usando un umbral de error de 2.5 se aplica mayor detalle para aproximar la geometría, que para un umbral de 10.0 como es utilizado en la figura 6.11. A modo representativo, la figura 6.12 muestra el terreno coloreado con textura, sobre el cual se han superpuesto los mallados de las figuras 6.10 y 6.11.

Los resultados obtenidos por medio de la variación del umbral permiten apreciar que un umbral de error visual de 1.0 píxeles produce la resolución máxima posible, siendo que valores más pequeños no tienen sentido. En estas condiciones, activar el *geomorphing* no mejora la calidad. Un umbral de 2.5 permiten obtener muy buena

calidad a excelente si se activa el *geomorphing*, en cuyo caso es inapreciable la diferencia con el anterior. Mediante un umbral de 5.0 se obtienen buenos resultados. Desactivando el *geomorphing* ciertos "artefactos" pueden notarse. Con umbrales de 7.5 los resultados visuales ya no son buenos. En este caso es la misma transición suavizada de los factores de teselado fraccionales la que puede introducir algunos efectos negativos. Estos efectos pueden apreciarse como ondulaciones horizontales al pasar entre dos niveles de detalle. Por último, utilizar umbrales cercanos a 10.0, no produce buenos resultados visuales, pero igualmente fue incluido en las pruebas para comprobar el comportamiento del algoritmo en todas las condiciones.

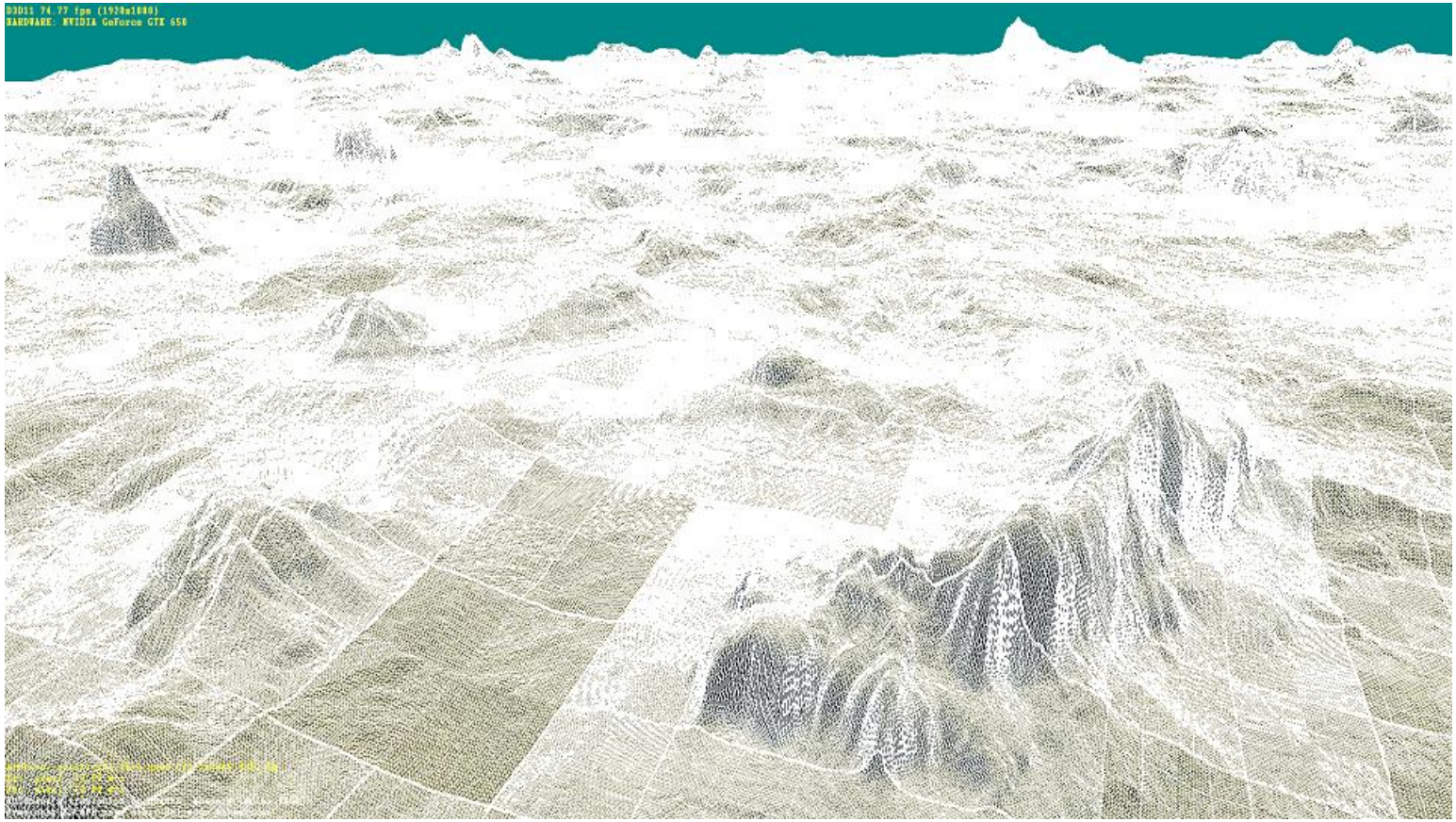

Figura 6.10: Mallado producido utilizando un umbral de error de 2.5 píxeles sobre el terreno de prueba artificial.

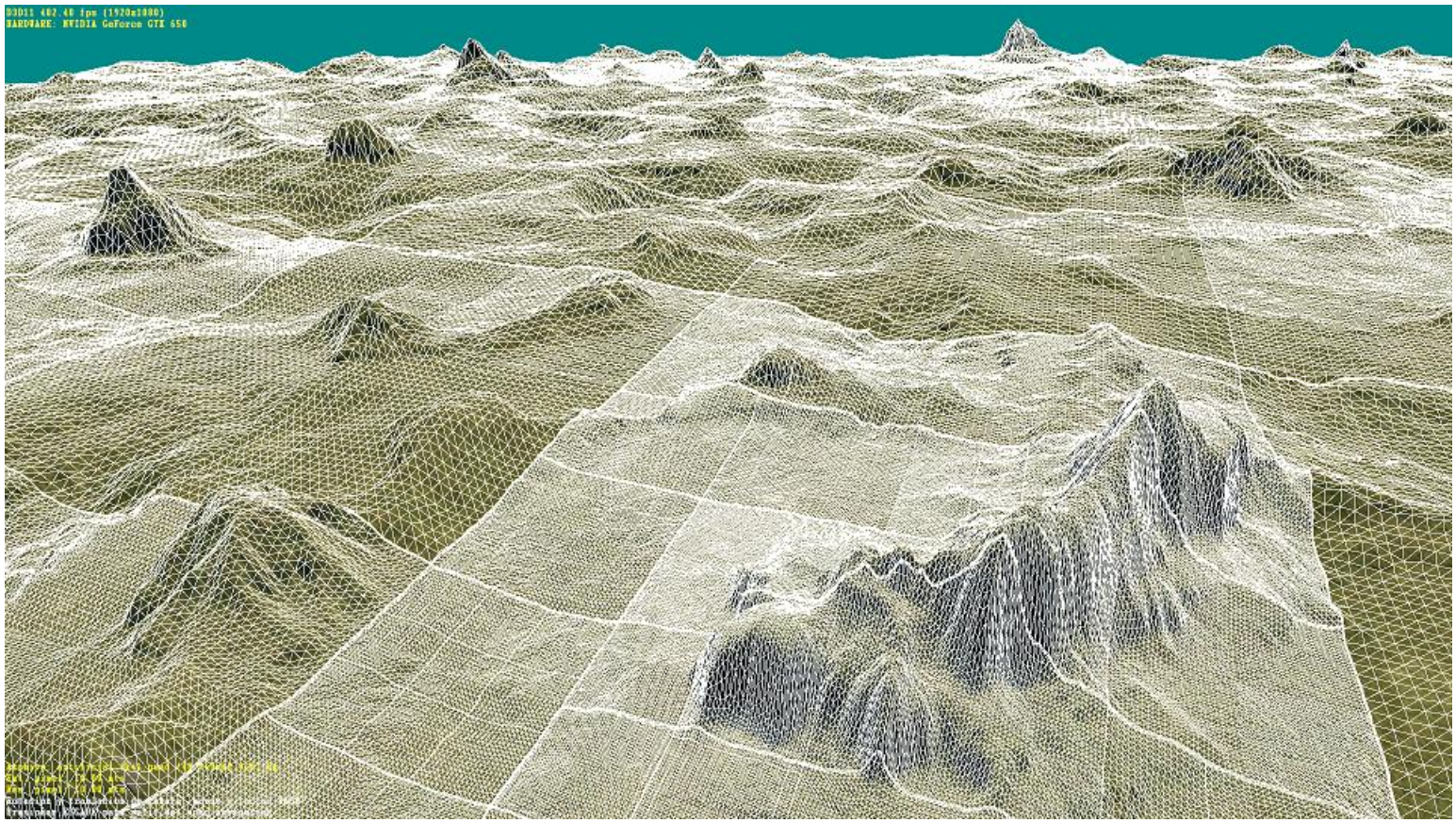

Figura 6.11: Mallado producido utilizando un umbral de error de 10.0 píxeles sobre el terreno de prueba artificial.

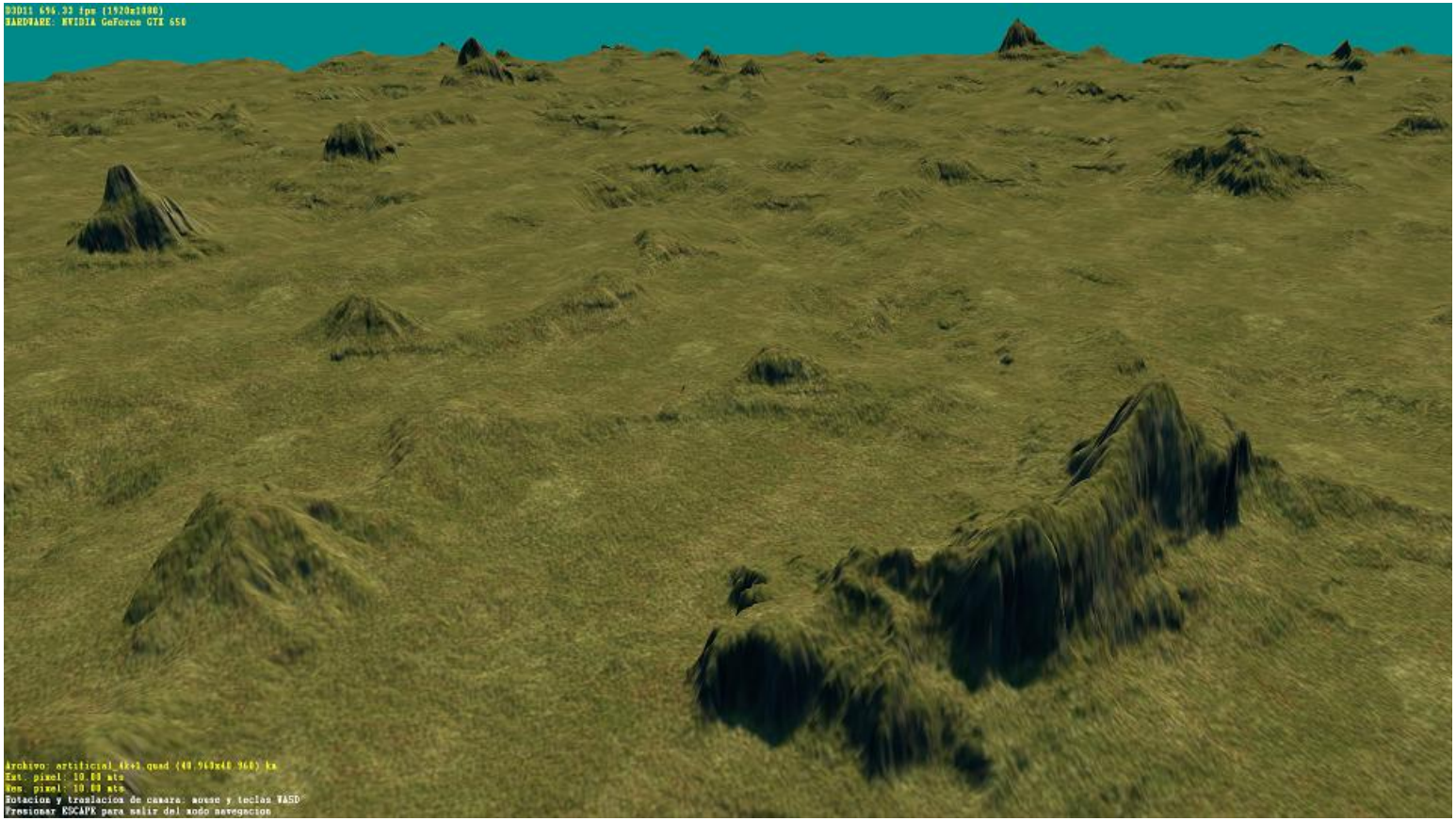

Figura 6.12: Terreno de prueba coloreado con textura sobre en cual se han superpuesto los mallados de las figuras X e Y

En el CD que acompaña este informe, se adjunta un video demostración de un sobrevuelo sobre el terreno de prueba. En el mismo puede observarse, entre otras cuestiones, el mallado generado por el algoritmo utilizando distintas umbrales de error. Además la técnica de *geomorphing* es representada mediante la mezcla dinámica de los colores rojo y verde. El mallado de los bloques de terreno es coloreado hacia el rojo para indicar que el nodo correspondiente esta a punto de subdividirse, y hacia el verde para indicar su colapso con sus nodos hermanos y su reemplazo por el nodo padre. En la figura 6.13 se muestra una captura de pantalla del efecto mencionado, y en la figura 6.14 se muestra la misma zona de terreno mientras se grafican las cajas envolventes de cada nodo renderizado.

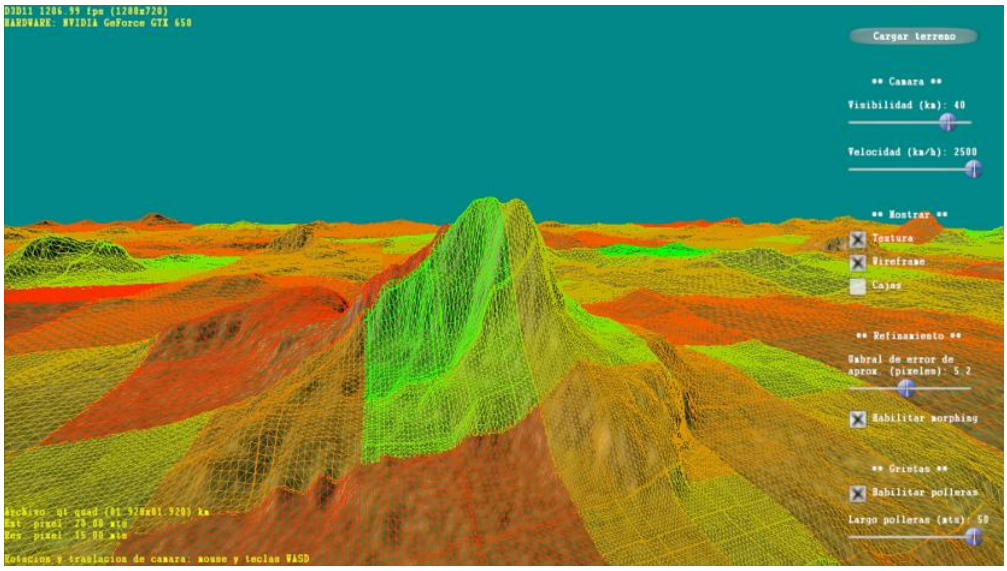

Figura 6.13: Técnica de geomorphing representada mediante la mezcla dinámica de los colores rojo y verde.

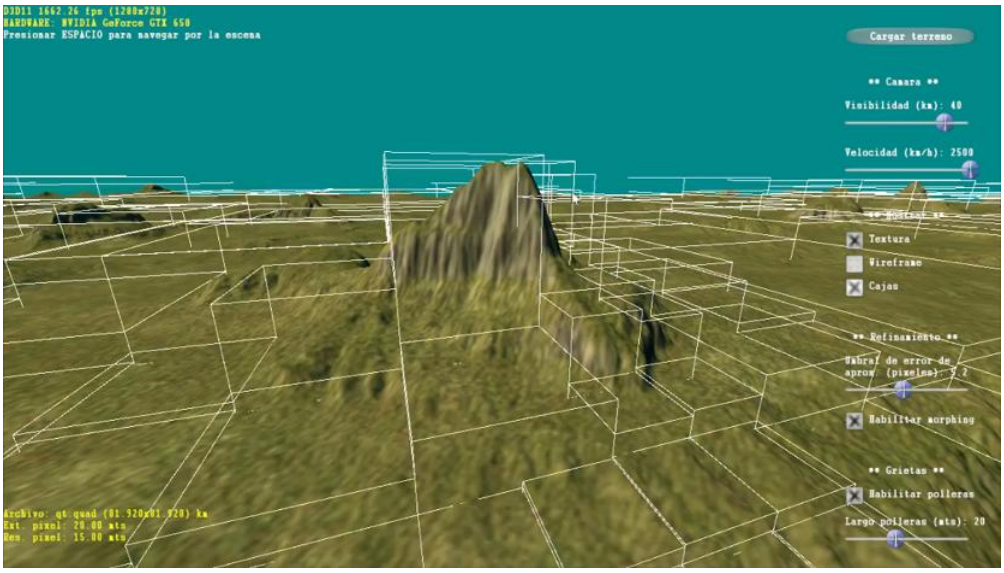

Figura 6.14: Cajas envolventes de nodos renderizados representadas mediante líneas blancas entre sus vértices.

6.4.3 Performance

6.4.3.1 Configuración de los tests

Una serie de tests fueron ejecutados en cada terreno los cuales consisten en un sobrevuelo de la escena que comienza en una esquina del terreno y terminan en el centro del mismo. Con el objetivo de obtener resultados representativos, el sobrevuelo de la escena consiste en distintas pasadas por las zonas más características de cada terreno, como ser picos montañosos (mayor resolución requerida) y valles o mar (menor resolución requerida).

Durante el recorrido de la escena se han registrado una serie de mediciones que permitieron elaborar las estadísticas que se presentan más adelante. Para las pruebas se utilizó un campo visual horizontal de 90 grados y con una distancia visual de 20km que permite muy buena visibilidad. Además estos valores resultan muy adecuados, ya que es la máxima distancia que permite, durante todo el recorrido hasta llegar al centro del terreno, obtener una visualización constante de superficie del terreno (aproximadamente un cuarto del área total del mismo).

El hardware utilizado para la ejecución de los tests, fue una PC con procesador Intel® Core™ i7-3770 de 3.40GHz, memoria RAM de 8GB DDR3, y una placa gráfica NVIDIA GeForce GTX 650 con 2GB DDR5 de memoria de video. La resolución de pantalla fue de 1920x1080 píxeles (HD) y en pantalla completa.

6.4.3.2 Resultados obtenidos

Con el objetivo de comparar performance respecto de la calidad de imagen obtenida, los tests fueron repetidos modificando el umbral de error visual. El resultado, es una variación de performance debido a la cantidad de geometría utilizada para generar la imagen en cada calidad visual. Además, como era de esperarse por el diseño mismo del algoritmo, cuando se sobrevuelan zonas que requieren mayor nivel de detalle, la performance también varía para mantener la calidad seleccionada. A continuación se describe el comportamiento general del sistema, y luego se presentan las tablas de performance obtenida en cada test.

En la figura 6.15, 6.16 y 6.17, se muestran los resultados de tres sobrevuelos -uno en cada terreno de prueba- utilizando un umbral de 2.5. Los resultados obtenidos utilizando otros umbrales, son similares en el sentido de la reacción o comportamiento del sistema de renderización. Este comportamiento se describe a continuación. En la primera figura se muestra la performance obtenida durante el transcurso de los tres tests. Para el caso *Puget Sound*, la duración por cuadro disminuye al transitar por la zona central de valles (menor resolución), y luego vuelve a aumenta al visualizarse las montañas del fondo (mayor rugosidad). En el caso de *Hawaii*, luego de superar dos colinas, el tiempo por cuadro disminuye porque se visualiza mayor superficie marítima. Por último para el terreno artificial, como la distribución de picos montañosos y hondonadas es frecuente y relativamente equidistante, se mantiene una duración de cuadro bien acotada.

Los resultados obtenidos coinciden con lo planteado por Wloka [39]. Wloka indica que la performance en un sistema de renderización por GPU, se encuentra supeditada al número de bloques de geometría (*batches*) enviados a renderizar, y no al número de triángulos en sí por bloque. En este sentido, cada bloque de terreno puede verse como un *batch* que consiste de 5000 triángulos aproximadamente (teselado). La figura 6.16 muestra la cantidad de bloques de terreno utilizados en cada cuadro, para los mismos tests de la figura anterior. Comparando ambas figuras, puede notarse como la performance está directamente relacionada al número de bloques renderizados, ya que se mantienen proporciones similares. Por último, en la figura 6.17 se muestra el número de triángulos promedio por bloque. Los valores se mantienen acotados y no parecen influir en la performance –al menos no tan directamente- como lo hace el número de bloques. Esto se debe a que en cada cuadro, algunos bloques son teselados en mayor grado que otros dependiendo de los factores de teselado utilizados en el *geomorphing*. Lo que se observa en la figura es que esta relación entre los distintos grados de teselado en los bloques, se mantiene relativamente constante para todos los cuadros. De esta manera, el número de triángulos por cuadro es proporcional al número de bloques por cuadro, por lo que la performance final queda dependiente del número total de bloques por cuadro.

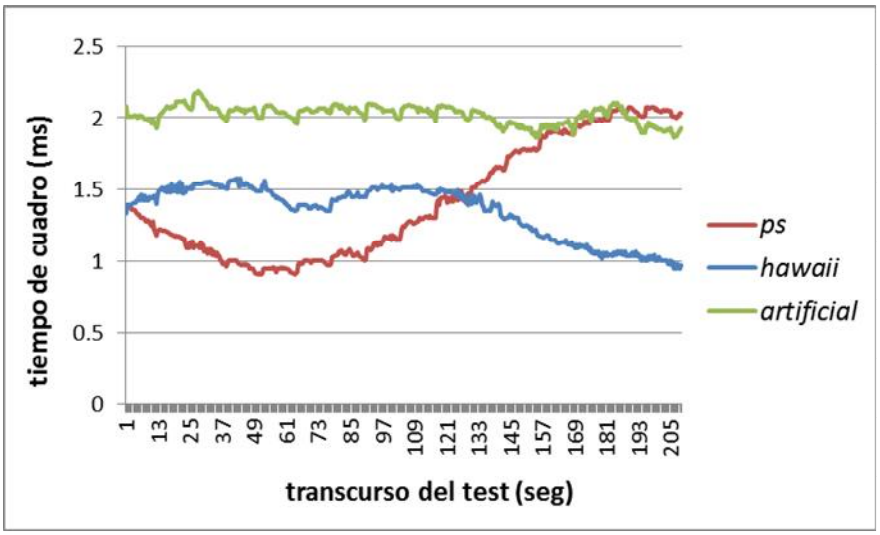

Figura 6.15: Duración de cuadro para los tests con umbral de error igual a 2.5.

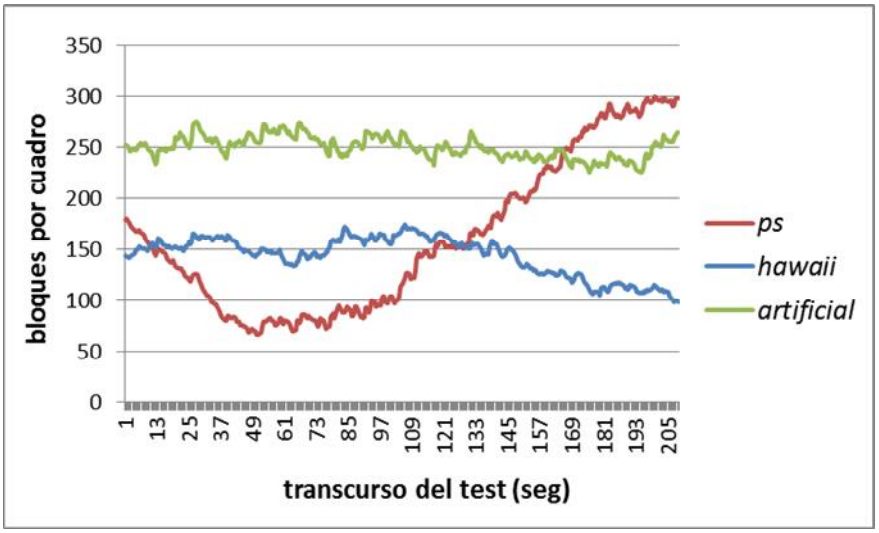

Figura 6.16: Número de bloques empleados por cuadro para los tests con umbral de error igual a 2.5.

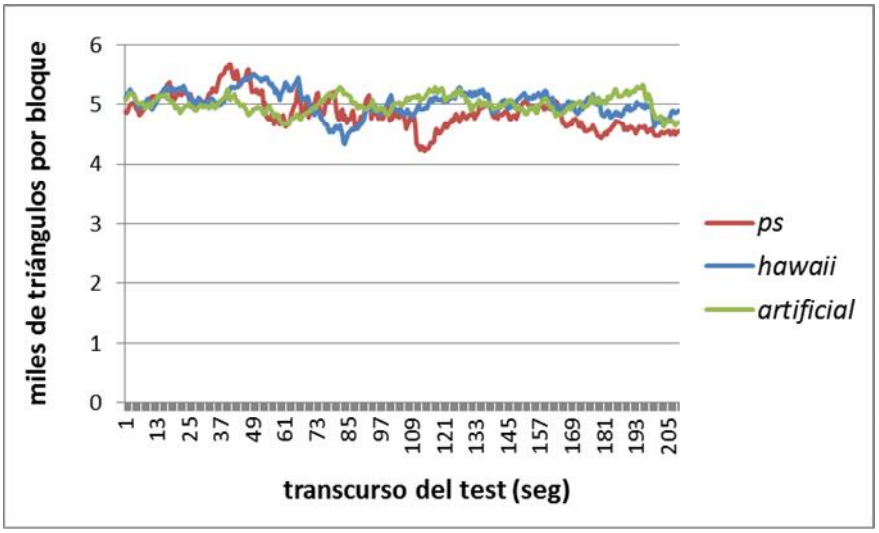

Figura 6.17: Número triángulos promedio por bloque para los tests con umbral de error igual a 2.5.

Como se ha mencionado, los tests fueron repetidos en cada terreno modificando el umbral de error visual. Los valores de umbral utilizados fueron τ = {1.0, 2.5, 5.0, 7.5, 10.0}. Los resultados se resumen en la tabla 6.3 para *Puget Sound*, tabla 6.4 para *Hawaii* y tabla 6.5 para el terreno artificial. Los valores de duración por cuadro se han estandarizado en unidades de cuadros por segundo o FPS (Frames Per Second).

En todos los casos puede observarse que, a medida que se relaja el umbral de error permitido, es necesario un menor número de bloques para aproximar la calidad seleccionada, y por tanto el número de cuadros que es posible generar por segundo aumenta respectivamente. Además, se observa que la media de FPS no registra grandes variaciones respecto de los extremos mínimo y máximo en cada test. Sólo un caso y a máxima calidad supera el 50%, siendo que la mayoría ronda el 20%. Los tests que registran menor variación corresponden al terreno artificial, debido

principalmente a la alta frecuencia de zonas rugosas. Por la misma razón, también registra menores medias respecto de los otros terrenos.

Umbral	***#bloques (#tris) / seg.***	***FPS***		
		Mínimo	***Media***	***Máximo***
1.0	145K (672M)	183 (-29%)	257	363 (+41%)
2.5	108K (523M)	482 (-37%)	762	1106 (+45%)
5.0	61K (315M)	1057 (-21%)	1343	1599 (+19%)
7.5	40K (201M)	1356 (-17%)	1634	1878 (+15%)
10.0	28K (140M)	1598 (-12%)	1810	2000 (+10%)

Tabla 6.3: Resultados obtenidos en la visualización del terreno Puget Sound.

Umbral	***#bloques (#tris) / seg.***	***FPS***		
		Mínimo	***Media***	***Máximo***
1.0	139K (688M)	269 (-20%)	337	544 (+61%)
2.5	107K (535M)	634 (-16%)	756	1058 (+40%)
5.0	72K (346M)	1047 (-19%)	1296	1618 (+25%)
7.5	44K (225M)	1308 (-19%)	1608	2119 (+32%)
10.0	30K (164M)	1472 (-17%)	1779	2339 (+31%)

Tabla 6.4: Resultados obtenidos en la visualización del terreno Hawaii.

Umbral	***#bloques (#tris) / seg.***	***FPS***		
		Mínimo	***Media***	***Máximo***
1.0	140K (694M)	201 (-6%)	214	227 (+6%)
2.5	124K (618M)	457 (-8%)	495	538 (+9%)
5.0	96K (481M)	872 (-7%)	934	1022 (+9%)
7.5	73K (367M)	1130 (-10%)	1249	1308 (+5%)
10.0	56K (284M)	1355 (-6%)	1449	1540 (+6%)

Tabla 6.5: Resultados obtenidos en la visualización del terreno artificial.

6.4.4 Consideraciones finales

Como se observa en las tablas 6.3, 6.4 y 6.5, las velocidades obtenidas dependen en primera instancia del umbral de error utilizado, y luego en un segundo lugar de las características de rugosidad encontradas en cada terreno.

Los mejores resultados han de ser aquellos que logren un balance entre calidad visual y performance. En base a la evaluación cualitativa (sección 6.4.2) y a los resultados de performance obtenidos (sección 6.4.3) se observa lo siguiente. Entre los umbrales de 1.0 y hasta 2.5 se registran mejoras en la performance y sin pérdida de calidad visual aparente. A partir de valores de 5.0 y mayores comienzan a producirse efectos inapropiados, por lo que ya no es tan conveniente su utilización. De esta manera, puede concluirse que un valor de umbral en el rango 2.5 a 5.0 permite lograr el mejor balance entre calidad visual y performance.

En en el rango mencionado, se han obtenido performances mínimas entre 500 FPS (umbral en 2.5) y 1000 FPS (umbral en 5.0). Como indican Akenine-Möller et al. [1], el *pipeline* gráfico debe generar cuadros con una frecuencia de al menos 15 FPS para que la secuencia de imágenes se pueda considerar como animación, aunque indica que los mejores resultados en cuanto a interactividad y fluidez de imagen se obtienen a partir de los 60 FPS. Los resultados obtenidos superan ampliamente estos valores, de manera que la performance lograda es muy satisfactoria.

7 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se exponen las conclusiones arribadas luego del desarrollo de esta tesina, y se plantean algunas posibles líneas de trabajos futuros.

7.1 Conclusiones

Se realizó la investigación del estado del arte de la visualización de modelos terrenos en 3D multiresolución. Se han incluido desde los primeros algoritmos basados en CPU hasta los más modernos basados en GPU. El análisis de las características y principales ventajas de los algoritmos más recientes basados en GPU, han motivado el diseño e implementación de un algoritmo de renderización de terrenos que tomara características ventajosas de los algoritmos estudiados y la vez resolviera alguna de las las limitaciones de los mismos.

Se realizó el diseño e implementación de un nuevo algoritmo para la visualización 3D de terrenos multirresolución, el cual puede ser utilizado en aplicaciones como por ejemplo simuladores de vuelo o de entrenamiento, sistemas de información geográfica, realidad virtual, videojuegos, etc.

Al igual que algoritmos como CDLOD [33], el algoritmo diseñado se basa en *quadtree*, una estructura regular que puede aprovechar las características de la GPU para realizar una representación multirresolución eficiente y escalable.

Se propuso abordar los inconvenientes que presenta el algoritmo CDLOD, el cual al considerar como único criterio de selección de nivel de detalle la distancia al punto de observación, se produce el surgimiento de accidentes geográficos o pops a medida que el observador se acerca a zonas rugosas. En el diseño del algoritmo se incluyó un criterio más apropiado en la selección del nivel de detalle, el cual considera la percepción que el usuario tendrá de las particularidades de cada zona del terreno. Para calcular el error de aproximación geométrica para cada bloque del terreno a renderizar, se desarrolló un preprocesador encargado de generar el *quadtree* requerido para el funcionamiento del algoritmo de forma “offline” a partir de los datos de altura (*heightmap*).

De esta manera es posible tener menos detalle en zonas llanas y tener mayor detalle en las zonas que poseen mayor rugosidad, obteniéndose una mejor distribución de triángulos para aproximar la superficie del terreno. Se logra así mayor calidad visual disminuyendo la percepción de cambios de resolución al visualizar y también evitando la posible aparición de *pops*, ya que las zonas con mayor rugosidad se adelantan en el tiempo a recibir mayor resolución que zonas más llanas. Además, el algoritmo permite controlar la calidad visual/performance dependiendo de la necesidad o aplicación particular donde se implemente el algoritmo, ajustando un parámetro de umbral de

error en píxeles.

Durante el transcurso de este trabajo, se ha estudiado el funcionamiento de la GPU, profundizando en las últimas características que ésta presenta como es el Shader Model 5 (SM5). Esto permitió obtener un algoritmo que controla la resolución para cada zona del terreno directamente por hardware, a la vez que se aprovecha el nuevo poder de cómputo y se libera la CPU para realizar otras tareas. Se desarrollaron los *shaders* necesarios a ser ejecutados en el *pipeline* de GPU, mediante los cuales se lleva a cabo la técnica final de renderización. Como resultado, se logró una implementación considerablemente sencilla del proceso de división de primitivas fundamental para la multirresolución. Esto es, mediante las nuevas funcionalidades de teselado fue posible obtener un algoritmo que efectiviza la resolución por hardware y que a su vez facilita la implementación de otras características como por ejemplo la técnica de *geomorphing*. El *geomorphing*, empleado como una transición suavizada entre niveles de detalle, permite por un lado contrarrestar aún más el efecto *popping*, a la vez que posibilita relajar el umbral de error para disminuir la cantidad total de triángulos en pantalla.

Para evaluar la viabilidad del algoritmo se llevaron a cabo una serie de tests de performance que dieron como resultado un muy buen rendimiento de la técnica desarrollada. De esta manera, el bajo consumo de recursos de procesamiento de CPU y GPU por parte de la misma, posibilita su utilización en aplicaciones interactivas que requieran de la visualización en tiempo real de modelos de terrenos para escenarios virtuales.

7.2 Trabajos futuros

Pueden identificarse las siguientes tres líneas de trabajo futuro, mediante las cuales es posible enriquecer el trabajo realizado hasta el momento. Las mismas han sido dispuestas de menor a mayor generalidad.

Unión entre bloques vecinos

La técnica de polleras o *skirts* utilizada en este trabajo, es una solución simple y efectiva de ocultar las grietas entre bloques vecinos de distinto nivel de detalle. Sin embargo, es posible que la técnica disminuya su efectividad en aplicaciones que requieren de visualizaciones muy cercanas al suelo o de altos niveles de detalle con que se colorea la superficie del terreno. Los buenos resultados de performance obtenidos, posibilitan ahora el desarrollo de técnicas más complejas y de mayor consumo de CPU. Una primera aproximación a la solución, podría consistir en desarrollar un mecanismo de consultas de bloques adyacentes para un determinado bloque, y que se implemente mediante búsquedas por el *quadtree*. Luego, para cada bloque de terreno a visualizar sería posible conocer el nivel de detalle de bloques vecinos y así poder adaptar sus bordes. Además, la adaptación de la geometría de bordes a la resolución de bloques vecinos podría realizarse mediante los factores de

teselado exteriores de Shader Model 5. Como optimización, los nodos recolectados durante el recorrido inicial para renderización (nodos visibles) podrían ser dispuestos en forma ordenada, de manera de poder restringir las búsquedas de nodos vecinos en una sola dirección por el *quadtree* (descendente o ascendente). Además, se podría aprovechar la coherencia entre cuadros desarrollando un mecanismo de "frente activo" de nodos que "recuerde" los nodos utilizados en cuadros anteriores, lo que permitiría restringir aún más las búsquedas a un número muy pequeño de saltos entre nodos promedio por cuadro.

Streaming de datos

Para terrenos muy extensos donde la información completa de altura y textura no quepa en memoria de GPU (o en la porción de memoria destinada a la técnica de visualización de terrenos), un mecanismo de carga dinámica (*streaming*) de datos puede ser necesario. Este mecanismo debería mantener en memoria un subconjunto de los datos necesarios para generar la imagen respecto de la ubicación del observador, y actualizarse a medida que éste se mueve. Un primer enfoque podría consistir en un buffer circular que permitiría mantener los datos en GPU centrados respecto del observador, con el cual sea posible realizar la carga sólo de las nuevas áreas expuestas (no visibles), como utilizan Losasso et al. [24]. Otro enfoque distinto podría consistir en asociar un pequeño *heightmap* individual por cada nodo del *quadtree*. Mediante un proceso de recorte y resampleo del *heightmap* original, que puede implementarse en la etapa de preprocesamiento, los nodos del *quadtree* pueden mantener sólo la información que su nivel de detalle y zona requiere. De esta manera, cuando un nodo deba ser renderizado sería posible cargar en GPU sólo la información necesaria para su visualización si es que ya no se encuentra previamente cargada. Por último, otras posibilidades a explorar son los mecanismos de compresión de datos que permitan disminuyan los requerimientos de memoria y agilicen la carga de datos desde disco o su transmisión por la red. La descompresión podría ocurrir en CPU antes de ser subida a la GPU, o incluso podría llevarse a cabo directamente en GPU como propone Lindstrom [23].

Ambientes más inmersivos

También es posible incorporar una serie de técnicas a la visualización que permitan potenciar no sólo la riqueza visual producida, sino también el realismo y la inmersión del usuario en el ambiente virtual. En primer lugar podrían incluirse técnicas de combinación de textura (*texture blending*) para conseguir colorear la superficie del terreno con mayor detalle. En particular, podrían explorarse aquellas técnicas que emplean un gran número de texturas pequeñas, repitiéndolas y combinándolas entre sí para mejorar la representación de distintos sustratos a lo largo del terreno. Por otro lado, pueden explorarse diferentes modelos de iluminación y sombreado, que permitan variar el color de la superficie dependiendo de su rugosidad y de la posición de una o más luces incorporadas al ambiente representado. Además, la intensidad y

color de la iluminación podría depender de la hora del día o clima simulados, haciendo que la escena luzca mucho más realista. Finalmente, pueden mencionarse otras alternativas como la incorporación de elementos sobre la superficie del terreno. La vegetación es uno de los elementos más comunes en cualquier ambiente de exteriores y que puede hacer que los escenarios virtuales luzcan mucho más convincentes. Sería necesario el estudio de las técnicas multirresolución relacionadas y de los diferentes métodos que existan, más o menos asistidos, para realizar la distribución de estos elementos sobre la superficie del terreno.

BIBLIOGRAFÍA

[1] Akenine-Möller T., Haines E., Hoffman N. *Real-Time Rendering*. 3ra edición, A.K.Peters Ltd., 2008.

[2] Asirvatham A., Hoppe H. *Terrain Rendering Using GPU-based Geometry Clipmaps*. En Pharr M. GPU Gems 2. Addison-Wesley. pp. 27–45, 2005.

[3] De Boer W. H. Fast *Terrain Rendering Using Geometrical MipMapping*. World Wide Web, Octubre de 2000. Disponible en http://www.flipcode.com/archives/ article_geomipmaps.pdf.

[4] De Berg M., Cheong O., Van Kreveld M., Overmars M. *Computational Geometry. Algorithms and Applications*. 3ra edición. Springer, 2008.

[5] Cohen-Or D., Levanoni Y. *Temporal continuity of levels of detail in Delaunay triangulated terrain*. IEEE Visualization ’96 Conference Proceedings, pp. 37–42, 1996.

[6] Duchaineau M., Wolinsky M., Sigeti D. E., Miller M. C., Aldrich C., Mineev-Weinstein M. B. *ROAMing Terrain: Real-Time Optimally Adapting Meshes.* IEEE Visualization '97 Conference Proceedings. pp. 81-88, 1997.

[7] Feinstein D. *Hlsl Development Cookbook*. Packt Publishing, 2013.

[8] Fernando R., Kilgard M. *The Cg Tutorial: The Definitive Guide To Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[9] FreeImage. Versión 3.16.0, marzo 2014. Disponible en http:// freeimage.sourceforge.net/download.html.

[10] Garland M. *Multiresolution Modeling: Survey & Future Opportunities*. Eurographics '99. State of the Art Reports, pp. 111-131, 1999.

[11] Gerasimov P., Fernando R., Green, S. *Shader Model 3.0: Using Vertex Textures.* NVIDIA Corporation Whitepaper, 2004. Disponible en ftp://download.nvidia.com/developer/ Papers/2004/Vertex_Textures/Vertex_Textures.pdf.

[12] Gribb G., Hartmann K. *Fast extraction of viewing frustum planes from the world-view-projection matrix.* World Wide Web, 2001. Disponible en http://gamedevs.org/uploads/fast-extraction-viewing-frustum-planes-from-world-view-projection-matrix.pdf.

[13] Guaycochea L. E., Abbate H. A. *Error-Bound Terrain Rendering Approach based on Geometry Clipmaps*. XVII Congreso Argentino de Ciencias de la Computación, 2011.

[14] Hawaii data set. Disponible en http://www.vertexasylum.com/ downloads/cdlod/dataset_hawaii.exe. Accedido en agosto 2014.

[15] Hoppe H., DeRose T., Duchamp T., McDonald J., Stuetzle W. *Mesh optimization*. ACM SIGGRAPH ’93 Proceedings. pp 19-26, 1993.

[16] Hoppe, H. *Progressive Meshes*. ACM SIGGRAPH ’96 Proceedings, pp. 99-108, 1996.

[17] Hoppe, H. View-Dependent Refinement of Progressive Meshes. ACM SIGGRAPH '97 Proceedings, pp. 189-198, 1997.

[18] Hoppe H. *Smooth view-dependent level-of-detail control and its application to terrain rendering*. IEEE Visualization ’98 Conference Proceedings, pp- 35-42, 1998.

[19] Bundysoft L3DT. Versión 11.11, noviembre 2011. Disponible en http:// www.bundysoft.com/L3DT/downloads/standard.php.

[20] Larsen B. D., Christensen N. J. *Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail*. Journal of WSCG, 11(2), pp. 282–9, 2003.

[21] Lindstrom P., Koller D., Ribarsky W., Hodges L. F., Faust N., Turner G. A. *Real-Time, Continuous Level of Detail Rendering of Height Fields*. Proceedings ACM SIGGRAPH ’96 Conference on Computer Graphics, pp. 109–118, 1996.

[22] Lindstrom P., Pascucci V. *Visualization of Large Terrains Made Easy*. IEEE Visualization ’01 Conference Proceedings, pp. 363-370, 2001.

[23] Lindstrom P., Cohen J. D. *On-the-Fly Decompression and Rendering of Multiresolution Terrain*. IEEE Visualization ‘09, 2009.

[24] Losasso F., Hoppe H. *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids*. ACM Transactions on Graphics (SIGGRAPH), 23(3), pp. 769-776, 2004.

[25] Luebke D., Reddy M., Cohen J. D., Varshney A., Watson B., Huebner R. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, 2003.

[26] Malvar H. *Fast Progressive Image Coding without Wavelets*. Data Compression Conference (DCC '00), pp. 243-252, 2000.

[27] Pajarola R. *Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation*. IEEE Visualization ’98 Conference Proceedings, pp. 19-26, 1998.

[28] Puget Sound data set. Disponible en http://www.cc.gatech.edu/ projects/large_models/ps.html. Accedido en agosto 2014.

[29] Pajarola R., Gobbetti E. *Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering*. The Visual Computer 23(8), 2007.

[30] Rost R. J., Licea-Kane B. M., Ginsburg D., Kessenich J. M., Lichtenbelt B., Malan H., Weiblen M. *OpenGL Shading Language*. 3ra edición. Addison-Wesley, 2010.

[31] Röttger S., Heidrich W., Slusallek P., Seidel H.-P. *Real-Time Generation of Continuous Levels of Detail for Height Fields*. Proceedings of the 6th

International Conference in Central Europe on Computer Graphics and Visualization, pp. 315–322, 1998.

[32] Smelik R. M., De Kraker K. J., Groenewegen S. A., Tutenel T., Bidarra R. *A Survey of Procedural Methods for Terrain Modelling*. Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS), 2009.

[33] Strugar F. *Continuous Distance-Dependent Level of Detail for Rendering Heightmaps*. Journal of Graphics, GPU, and Game Tools, 14(4), pp. 57-74, 2009.

[34] [34] Tanner C. C., Migdal C. J., Jones M. T. *The clipmap: a virtual mipmap*. Proceedings of the 25th annual conference on Computer Graphics and interactive techniques. pp. 151–158, 1998.

[35] Ulrich T. *Rendering Massive Terrains using Chunked Level of Detail Control*. Course Notes of ACM SIGGRAPH ’02, ACM Press, 2002.

[36] Valdetaro A., Nunes G., Raposo A., Feijó B. *Understanding Shader Model 5.0 with DirectX 11*. IX Brazilian symposium on computer games and digital entertainment, 2010.

[37] Williams, L. *Pyramidal Parametrics*. ACM SIGGRAPH '83 Proceedings, pp. 1-11, 1983.

[38] Wagner D. *Terrain Geomorphing in the Vertex Shader*. En Wolfgang E. Shader-X 2: Shader Programming Tips & Tricks With Directx 9. Wordware Publishing, 2004.

[39] Wloka, M. *Batch, Batch, Batch: What Does It Really Mean?*. Game Developers Conference '03. 2003

[40] Zink J., Pettineo M., Hoxley J. *Practical Rendering and Computation with Direct3D 11*. A.K.Peters Ltd./CRC Press, 2011.

Apéndice A - DOCUMENTACIÓN DE CLASES

Este apéndice presenta una descripción de la arquitectura de las aplicaciones desarrolladas en este trabajo. Para esto se utilizan diagramas UML de clases, junto a la documentación describiendo las responsabilidades y métodos que cada clase implementa.

La sección A.1 describe las clases utilizadas en la aplicación denominada Preprocesador. Luego la sección A.2 describe las clases del Visualizador.

A.1 Preprocesador

A.1.1 Diagrama de clases

La figura A.1, muestra las clases utilizadas para la implementación de la aplicación de preprocesamiento. A continuación se realiza una presentación de cada clase, y luego en la próxima sección se documenta cada clase por separado.

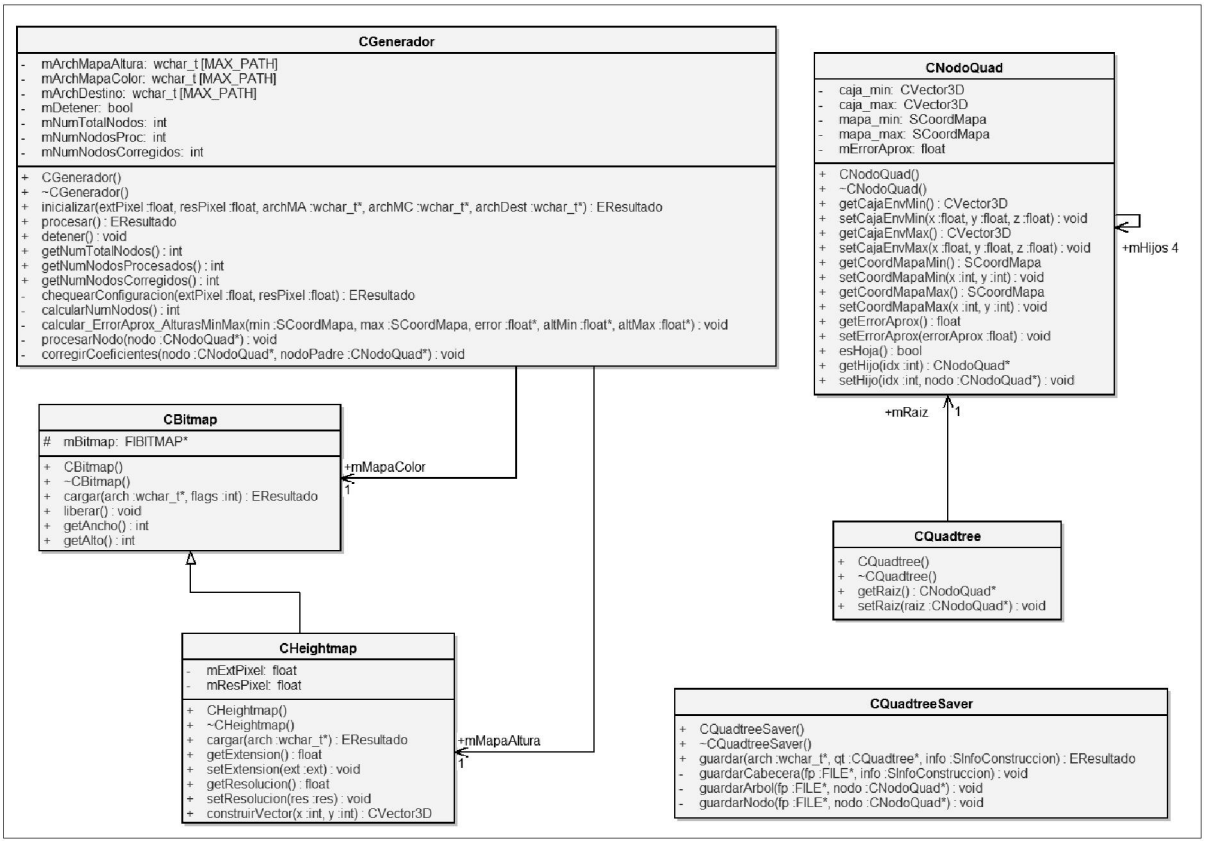

Figura A.1: Diagrama de clases del preprocesador.

La clase más importante es *CGenerador*, ya que se encarga de realizar el proceso principal de esta aplicación, es decir, generar la estructura de datos *quadtree* y calcular para cada uno de sus nodos el error de aproximación geométrica (secciones 6.2.2 a 6.2.4). La responsabilidad del resto de las clases, es la de dar apoyo a *CGenerador* brindando distintos servicios.

Las clases *CQuadtree* y *CNodoQuad* modelan la estructura de datos *quadtree* utilizada por el algoritmo, permitendo mantener y administrar la estructura en memoria. Una vez creado el *quadtree*, la clase *CQuadtreeSaver* se encarga de realizar el almacenamiento del mismo en disco, bajo el formato de archivo definido para este propósito (sección 6.2.5).

Las Clases *CBitmap* y *CHeightmap* forman una jerarquía que se encarga de la carga de las texturas del mapa de color y mapa de alturas respectivamente. El mapa de color es necesario para realizar la corroboración de que su dimensión posee la forma 2n+1. Por otro lado, la clase *CHeightmap* se deriva *CBitmap*, agregando la capacidad de construir los vectores 3d a partir del acceso a las muestras de altura, utilizados en el cálculo de los errores de aproximación.

A.1.2 Descripción de clases

A continuación se documenta la interfaz de las clases implementadas en el preprocesador.

Datos generales	
Clase	*CGenerador*
Responsabilidad	*Generar la estructura de datos quadtree y calcular para cada uno de sus nodos el error de aproximación geométrica.*
Deriva de	
Métodos	
CGenerador (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CGenerador (destructor)	Descripción *Destruye los mapas de altura y color.*
inicializar	Descripción *Inicializa los parámetros de configuración para la generación del quadtree.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *float extPixel: Extensión de píxel en metros.* ● *float resPixel: Resolución de pixel en metros.* ● *wchar_t *archMA: Path hacia el mapa de altura.*

	● *wchar_t *archMC: Path hacia el mapa de color.* ● *wchar_t *archDest: Path hacia archivo de destino ".quad"*
procesar	Descripción *Inicia el procesamiento y creación del quadtree.* Retorno *EResultado: Enumeración de tipos de error.*
detener	Descripción *Detiene el procesamiento y creación del quadtree.*
getNumTotalNodos	Descripción *Retorna el número total de nodos creados en el quadtree.* Retorno *int: número de nodos creados.*
getNumNodosProcesados	Descripción *Retorna el número total de nodos creados en el quadtree.* Retorno *int: número de nodos creados.*
getNumNodosCorregidos	Descripción *Retorna el número de nodos con coeficientes corregidos.* Retorno *int: número de nodos con coeficiente corregidos.*
procesarNodo *(privado)*	Descripción *Construye el quadtree de forma recursiva.* Parámetros ● *CNodoQuad *nodo: Nodo a procesar.*
calcular_ErrorAprox_AlturasMinMax *(privado)*	Descripción *Calcular el error de aproximación y los límites de altura para la zona indicada.* Parámetros ● *SCoordMapa min: Coordenada mínima del mapa de altura.* ● *SCoordMapa max: Coodenada máxima del mapa de altura.* ● *float *error: Error de aproximación calculado.* ● *float *altMin: Altura máxima hallada.* ● *float *altMax: Altura mínima hallada.*
corregirCoeficientes *(privado)*	Descripción *Corrige de forma recursiva los coeficientes que son mayores al de su nodo padre.* Parámetros

	● *CNodoQuad *nodo: Subárbol a corregir.* ● *CNodoQuad *nodoPadre: Nodo padre. NULL si nodo es raíz.*

Datos generales	
Clase	*CQuadtree*
Responsabilidad	*Modela la estructura de datos quadtree.*
Deriva de	
Métodos	
CQuadtree (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CQuadtree (destructor)	Descripción *Destruye el quadtree liberando memoria alojada.*
getRaiz	Descripción *Retorna la raíz del quadtree.* Retorno *CNodoQuad *: Raíz del quadtree.*
setRaiz	Descripción *Asigna la raíz del quadtree.* Parámetros ● *CNodoQuad *raiz: Nueva raíz del quadtree.*

Datos generales	
Clase	*CNodoQuad*
Responsabilidad	*Modela un nodo de la estructura quadtree. Permite mantener y administrar subárboles en memoria.*
Deriva de	
Métodos	
CNodoQuad (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CNodoQuad (destructor)	Descripción *Destruye el nodo y todo su subárbol.*
getCajaEnvMin	Descripción *Retorna la coordenada mínima de la caja envolvente del nodo.* Retorno *CVector3D: Vector coordenada mínima.*
setCajaEnvMin	Descripción *Asigna la coordenada mínima de la caja envolvente del nodo.* Parámetros ● *x: Coordenada x mínima de la caja.* ● *y: Coordenada y mínima de la caja.*

	● *z: Coordenada z mínima de la caja.*
getCajaEnvMax	Descripción *Retorna la coordenada máxima de la caja envolvente del nodo.* Retorno *CVector3D: Vector coordenada máxima.*
setCajaEnvMax	Descripción *Asigna la coordenada máxima de la caja envolvente del nodo.* Parámetros ● *x: Coordenada x máxima de la caja.* ● *y: Coordenada y máxima de la caja.* ● *z: Coordenada z máxima de la caja.*
getCoordMapaMin	Descripción *Retorna la coordenada mínima de la región abarcada por el nodo en el mapa.* Retorno *SCoordMapa: Coordenada mínima del mapa.*
setCoordMapaMin	Descripción *Asigna la coordenada mínima de la región abarcada por el nodo en el mapa.* Parámetros ● *x: Coordenada x mínima del mapa.* ● *y: Coordenada y mínima del mapa.*
getCoordMapaMax	Descripción *Retorna la coordenada máxima de la región abarcada por el nodo en el mapa.* Retorno *SCoordMapa: Coordenada máxima del mapa.*
setCoordMapaMax	Descripción *Asigna la coordenada máxima de la región abarcada por el nodo en el mapa.* Parámetros ● *x: Coordenada x máxima del mapa.* ● *y: Coordenada y máxima del mapa.*
getErrorAprox	Descripción *Retorna el error de aproximación geométrica del nodo.* Retorno *float: Error de aproximación geométrica.*
setErrorAprox	Descripción *Asigna el error de aproximación geométrica del nodo.* Parámetros ● *float errorAprox: Error de aproximación geométrica.*
getHijo	Descripción

	Retorna el hijo indicado del nodo. Retorno *CNodoQuad *: Nodo hijo.* Parámetros ● *int idx: Índice de hijo entre 0 y 3.*
setHijo	Descripción *Establece un hijo del nodo.* Parámetros ● *int idx: Índice de hijo a establecer.* ● *CNodoQuad *nodo: Nodo a establecer como hijo.*
esHoja	Descripción *Consulta si los 4 hijos son NULL.* Retorno *bool: Indica si es o no un nodo hoja del quadtree.*

Datos generales	
Clase	*CQuadtreeSaver*
Responsabilidad	*Realiza el almacenamiento de un quadtree en disco en el formato de archivo definido a tal propósito.*
Deriva de	
Métodos	
CQuadtreeSaver (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CQuadtreeSaver (destructor)	Descripción *Destruye la clase.*
guardar	Descripción *Guarda el quadtree indicado en disco.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *wchar_t *arch: Path al archivo de destino.* ● *CQuadtree *qt: Quadtree a almacenar.* ● *SInfoConstruccion info: Parámetros de construcción del quadtree a incluir en la cabecera del archivo.*
guardarCabecera (privado)	Descripción *Guarda la cabecera de archivo.* Parámetros ● *FILE *fp: Descriptor de archivo.* ● *SInfoConstruccion info: Parámetros de construcción del quadtree a incluir en la cabecera del archivo.*

guardarArbol *(privado)*	Descripción *Recorre recursivamente un subárbol del quadtree guardando cada nodo en el archivo de destino.* Parámetros ● *FILE *fp: Descriptor de archivo.* ● *CNodoQuad *nodo: Nodo padre del subárbol.*
guardarNodo *(privado)*	*Descripción* *Guarda los datos del nodo en el archivo destino.* *Parámetros* ● *FILE *fp: Descriptor de archivo.* ● *CNodoQuad *nodo: Nodo del quadtree.*

Datos generales	
Clase	*CBitmap*
Responsabilidad	*Modela un mapa de bits genérico. Se encarga de realizar la carga texturas desde disco.*
Deriva de	
Métodos	
CBitmap *(constructor)*	Descripción *Inicializa por defecto los atributos de la clase.*
~CBitmap *(destructor)*	Descripción *Destruye la clase liberando los recursos del mapa de bits.*
cargar	Descripción *Carga el mapa de bits desde disco.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *wchar_t *arch: Path del archivo en disco.* ● *int flags: Expone los flags de la librería FreeImage para la carga de bitmaps.*
liberar	Descripción *Libera los recursos del mapa de bits.*
getAncho	Descripción *Retorna el ancho del bitmap.* Retorno *int: Ancho del bitmap en píxeles.*
getAlto	Descripción *Retorna el alto del bitmap.* Retorno *int: Alto del bitmap en píxeles.*

Datos generales	
Clase	*CHeightmap*
Responsabilidad	*Modela el mapa de altura (heightmap). Es responsable de construir vectores 3d utilizados en el cálculo de los errores de aproximación, a partir del acceso a las muestras de altura.*
Deriva de	*CBitmap*
Métodos	
CHeightmap (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CHeightmap (destructor)	Descripción *Destruye la clase liberando los recursos del mapa de altura.*
cargar	Descripción *Carga el mapa de altura desde disco.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *wchar_t *arch: Path del archivo en disco.*
getExtension	Descripción *Retorna la extensión de píxel utilizada en el cálculo de los errores de aproximación.* Retorno *float: Extensión de píxel en metros.*
setExtension	Descripción *Asigna la extensión de píxel a utilizar en el cálculo de los errores de aproximación.* Parámetros ● *float ext: Extensión de píxel en metros.*
getResolucion	Descripción *Retorna la resolución de píxel utilizada en el cálculo de los errores de aproximación.* Retorno *float: Resolución de píxel en metros.*
setResolucion	Descripción *Asigna la Resolución de píxel a utilizar en el cálculo de los errores de aproximación.* Parámetros ● *float ext: Resolución de píxel en metros.*
construirVector	Descripción *Construye el vector correspondiente a la muestra (x,y) según la extensión y resolución configurados.* Retorno

	float: Vector coordenada de la muestra en espacio de objeto. Parámetros ● *int x: Coordenada x en espacio de muestras.* ● *int y: Coordenada y en espacio de muestras.*

A.2 Visualizador

A.2.1 Diagrama de clases

La figura A.2 muestra el diseño de clases que conforman la aplicación de visualización. En forma general puede decirse que existen dos grupos de clases. En la parte (a) de la figura, se agrupan aquellas clases que dan soporte al entorno de visualización y que brindan las facilidades para la interacción del usuario con la aplicación. El resto de las clases se muestran en la parte (b) de la figura, y corresponden a aquellas clases que modelan las funcionalidades específicas del algoritmo de visualización presentado en el capítulo 5.

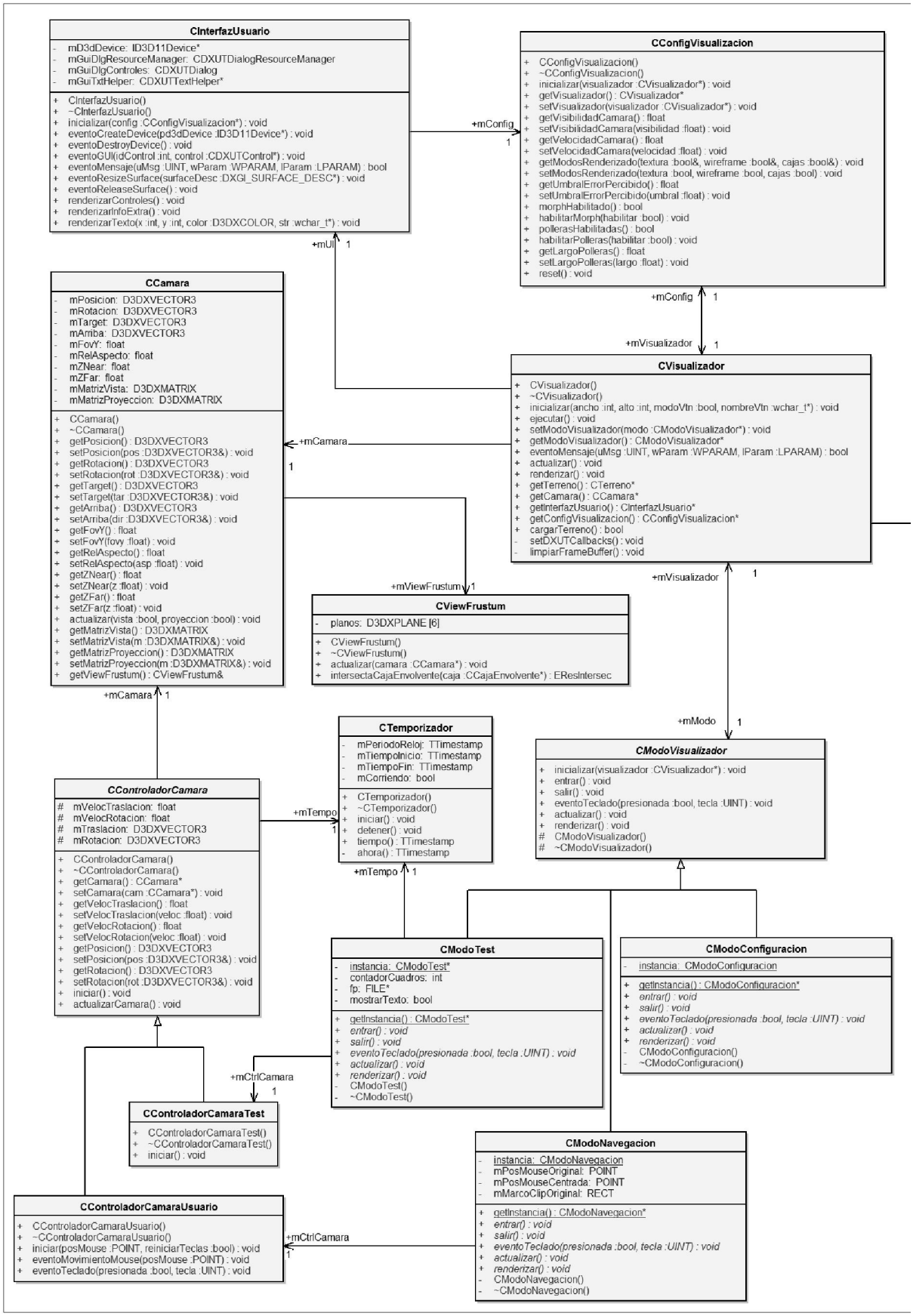

Figura A.2-parte(a): Diagrama de clases del visualizador.

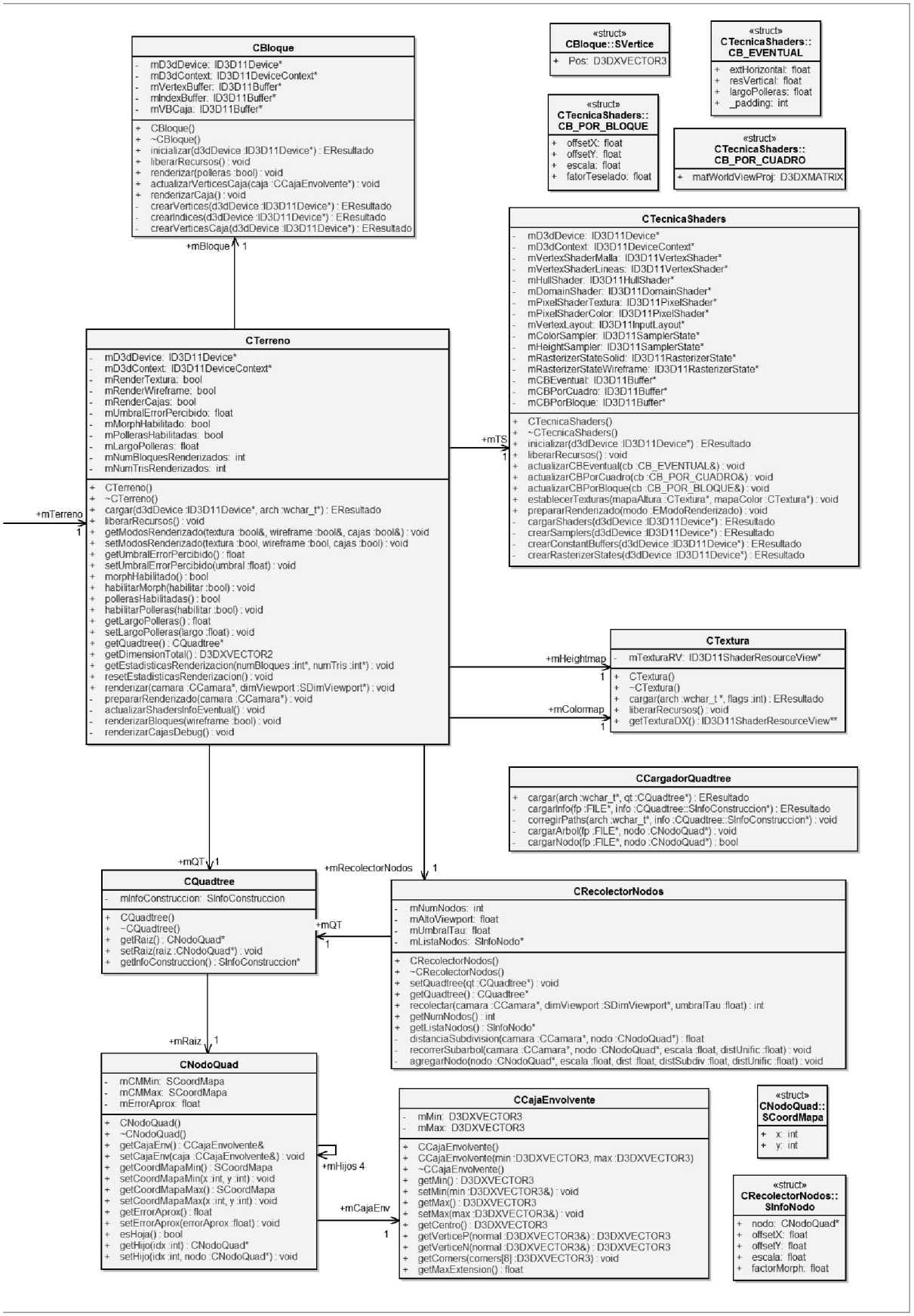

Figura A.2-parte(b): Diagrama de clases del visualizador.

Comenzando por las clases de la figura A.2-parte(a), la clase principal de este grupo es *CVisualizador*. Esta clase representa a la aplicación en sí, permitiendo la inicialización de la ventana de la aplicación y la implementación del bucle de ejecución principal de actualización y renderización de la escena. Además permite el acceso y navegación por las clases más importantes del visualizador, y la captura y redirección de mensajes del sistema operativo hacia las clases que requieren su conocimiento, como los modos de funcionamiento de la aplicación y la interfaz de usuario.

La clase *CInterfazUsuario*, hace uso de la librería utilitaria DXUT para disponer de un conjunto de controles gráficos que permiten al usuario interactuar con la aplicación. Esta clase es responsable de crear, administrar y renderizar estos controles, como también permite la renderización de textos informativos en pantalla por parte de los distintos modos de funcionamiento de la aplicación. Para llevar a cabo esta tarea, captura distintos eventos producidos por la librería y actualiza el contenido de los controles, evitando acceder directamente a los distintos objetos que contienen los datos concretos. Esa tarea es delegada a la clase *CConfigVisualización*, quien a través de *CVisualizador*, sabe cómo acceder a distintos objetos tanto para obtener como para modificar sus atributos. La responsabilidad de *CConfigVisualización* consiste en inicializar y mantener la configuración de la aplicación, abstrayendo a la interfaz de usuario de la forma de llevar a cabo esa tarea.

La clase *CCamara* modela la cámara de la escena, su posición y orientación en el espacio, como así también los parámetros para realizar la proyección perspectiva. Esta clase es responsable de calcular las matrices de vista y proyección utilizadas para transformar los vértices de la escena. La tarea de mantener el *frustum* de la cámara para realizar la técnica de *View frustum culling*, es delegada a la clase *CViewFrustum*, la cual modela el *frustum* mediante 6 planos que contienen al espacio de visualización.

La jerarquía formada por la clase base *CModoVisualizador*, y las clases derivadas *CModoTest*, *CModoNavegación* y *CModoConfiguración*, permiten variar el comportamiento de la aplicación modelando los distintos modos de funcionamiento en que puede encontrarse la misma (patrón State). La clase *CVisualizador*, delega en estas clases las tareas de actualización y renderización del bucle de ejecución principal. La clase *CModoTest* es responsable de llevar a cabo los tests de performance y guardar en disco el resultado de los mismos. *CModoNavegación* permite la libre navegación por parte del usuario en la escena virtual, y *CModoConfiguración* habilita la renderización del menú principal en pantalla, permitiendo la configuración del algoritmo de visualización. El control específico de la cámara en los modos de navegación y test es delegado a las clases *CControladorCamaraUsuario* y *CControladorCamaraTest* respectivamente. Además cada una de estas clases correspondientes a los modos de funcionamiento de la aplicación, modela el patrón Singleton, restringiendo la creación a una sola instancia por clase.

La jerarquía formada por las clases *CControladorCamara*, *CControladorCamaraUsuario* y *CControladorCamaraTest*, se encarga de realizar el control de la cámara en base al tiempo transcurrido entre cuadros (animación). La clase base *CControladorCamara* provee la funcionalidad para controlar la cámara en base a la velocidad a la cual la cámara se traslada y rota. Las clases derivadas *CControladorCamaraUsuario* y

CControladorCamaraTest modelan la funcionalidad específica necesaria en los modos de funcionamiento navegación y test de la aplicación respectivamente. La primera clase, modifica la cámara dependiendo de la entrada de teclado y del movimiento del mouse, mientras la segunda, configura una posición y dirección iniciales en función del test realizado.

La clase *CTemporizador*, modela un contador de tiempo que es utilizado por *CControladorCamara* para animar la cámara en base al tiempo trascurrido y no a los cuadros renderizados, y por CModoTest para controlar el tiempo de duración de cada test.

En la figura A.2-parte (b), se encuentran las clases que modelan las funcionalidades necesarias para la implementación del algoritmo de visualización. La clase principal de este segundo grupo es *CTerreno*, la cual representa el conjunto de datos de un modelo de terreno. Además esta clase conforma el punto de entrada para la configuración y renderización del terreno representado. Esta clase es responsable de ejecutar el esqueleto del algoritmo de visualización diseñado, delegando en el resto de las clases las tareas específicas del mismo.

Las clases *CNodoQuad* y *CQuadtree*, modelan la estructura del *quadtree*, y permiten la creación y administración del mismo. La carga desde disco, es delegada a la clase *CCargadorQuadtree*, la cual implementa el formato de archivo definido a tal propósito (sección 6.3.1).

La clase *CTextura*, es responsable de cargar desde disco de las texturas utilizadas en los mapas de altura y color, y de la administración de estos recursos en memoria.

La clase *CRecolectorNodos* es responsable de recorrer el *quadtree*, conformando la lista de nodos a renderizar. Además esta clase se encarga de calcular para cada nodo recolectado, el factor de *morphing* y otros parámetros utilizados en la renderización de cada zona visible del terreno (sección 6.3.5). La clase *CCajaEnvolvente* modela una caja alineada a los ejes que es utilizada por los nodos del *quadtree* para envolver su geometría en el espacio. Esta clase permite rechazar (no recolectar) aquellos nodos que se encuentran fuera del alcance visual de la cámara.

Finalmente, la clase *CBloque* modela el bloque genérico de terreno. Su función principal es la de crear los buffers que contienen la geometría necesaria para la definición del mismo (sección 6.3.2). Luego, la clase *CTecnicaShaders* modela la técnica de renderización en sí (sección 6.3.6). Esta clase es responsable de cargar y compilar los *shaders* desde disco y de posibilitar la comunicación con los mismos para actualizar los parámetros necesarios en la renderización de cada bloque de terreno.

A.2.2 Descripción de clases

A continuación se documenta la interfaz de las clases implementadas en la aplicación de visualización.

Datos generales	
Clase	*CVisualizador*
Responsabilidad	*Modela a la aplicación en sí. Es responsable de iniciar la ventana principal y de implementar el bucle de ejecución principal de actualización y renderización de la escena. Además captura y redirecciona los mensajes del sistema operativo hacia los modos de funcionamiento de la aplicación y la interfaz de usuario.*
Deriva de	
Métodos	
CVisualizador (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CVisualizador (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
inicializar	Descripción *Inicializa la aplicación construyendo y construye la ventana principal de renderización.* Parámetros ● *int ancho: Ancho de la ventana.* ● *int alto: Alto de la ventana* ● *bool modoVtn: Modo ventana o pantalla completa.* ● *wchar_t *nombreVtn: Título de la ventana.*
ejecutar	Descripción *Iniciar el bucle de ejecución principal de actualización y renderización de la escena.*
eventoMensaje	Descripción *Captura y redirecciona los mensajes del sistema operativo hacia los modos de funcionamiento de la aplicación y la interfaz de usuario.* Retorno *bool: Indica si fue capturado el mensaje.* Parámetros ● *UINT uMsg: Mensaje del sistema operativo.* ● *WPARAM wParam: Parámetro wide del mensaje.* ● *LPARAM lParam: Parámetro Low del mensaje.*
actualizar	Descripción *Realiza la actualización de la cámara y controles en pantalla antes de la renderización.*
renderizar	Descripción *Renderiza el terreno y controles en pantalla.*
cargarTerreno	Descripción *Despliega un cuadro de diálogo y carga el terreno indicado desde disco.* Retorno *bool: Indica si la carga fue satisfactoria.*

Datos generales	
Clase	*CInterfazUsuario*
Responsabilidad	*Esta clase es responsable de crear, administrar y renderizar controles gráficos que permiten al usuario interactuar con la aplicación, como también permite la renderización de textos informativos en pantalla.*
Deriva de	
Métodos	
CInterfazUsuario (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CInterfazUsuario (destructor)	Descripción *Destruye la clase y libera los recursos de controles gráficos.*
eventoCreateDevice	Descripción *Reconstruye los controles gráficos basándose en el nuevo dispositivo pasado por parámetro.* Parámetros ● *ID3D11Device* pd3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.*
eventoDestroyDevice	Descripción *Libera controles gráficos y espera la creación de un nuevo dispositivo.*
eventoGUI	Descripción *Maneja los eventos de la GUI devueltos por la librería DXUT.* Parámetros ● *int idControl: id del control que generó el evento.* ● *CDXUTControl *control: Control gráfico de DXUT.*
eventoMensaje	Descripción Maneja los mensajes del sistema operativo para actualizar los controles gráficos. Retorno *bool: Indica si fue capturado el mensaje.* Parámetros ● *UINT uMsg: Mensaje del sistema operativo.* ● *WPARAM wParam: Parámetro wide del mensaje.* ● *LPARAM lParam: Parámetro Low del mensaje.*
renderizarControles	Descripción *Renderiza controles gráficos.*
renderizarTexto	Descripción *Permite renderizar texto adicional en pantalla.* Parámetros ● *int x: Coordenada x de pantalla.* ● *int y: Coordenada y de pantalla.* ● *D3DXCOLOR color: Color RGB del texto.* ● *wchar_t *str: Texto a renderizar.*
renderizarInfoExtra	Descripción

	Renderiza información de GPU utilizada, FPS y resolución de pantalla.

Datos generales	
Clase	*CConfigVisualización*
Responsabilidad	*Es responsable de inicializar y mantener la configuración de la aplicación, abstrayendo a la interfaz de usuario de la forma de llevar a cabo esa tarea.*
Deriva de	
Métodos	
CConfigVisualizacion (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CConfigVisualizacion (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
getVisibilidadCamara	Descripción *Obtiene la distancia visual de la cámara.* Retorno *float: Distancia en km.*
setVisibilidadCamara	Descripción *Establece la distancia visual de la cámara.* Parámetros ● *float visibilidad: Distancia visual de la cámara en km.*
getVelocidadCamara	Descripción *Obtiene la velocidad de la cámara.* Retorno *float: Velocidad en km/h.*
setVelocidadCamara	Descripción *Establece la velocidad de la cámara.* Parámetros ● *float velocidad: Velocidad en km/h.*
getModosRenderizado	Descripción *Obtiene los modos de renderizado activados.* Parámetros ● *bool &textura: Indica si se renderiza el terreno mediante mapa de color.* ● *bool &wireframe: Indica si se renderiza el "esqueleto" del mallado multirresolución.* ● *bool &cajas: Indica si se renderizan las cajas envolventes de los bloques de terreno.*
setModosRenderizado	Descripción *Activa/Desactiva los modos de renderizado.* Parámetros

	● *bool textura: Activa/Desactiva el renderizado del terreno mediante mapa de color.* ● *bool wireframe: Activa/Desactiva el renderizado del "esqueleto" del mallado multirresolución.* ● *bool cajas: Activa/Desactiva el renderizado de las cajas envolventes de los bloques de terreno.*
getUmbralErrorPercibido	Descripción *Obtiene el umbral de percepción visual.* Retorno *float: Umbral de percepción en píxeles.*
setUmbralErrorPercibido	Descripción *Establece el umbral de percepción visual.* Parámetros ● *float umbral: Umbral de percepción en píxeles.*
morphHabilitado	Descripción *Obtiene si el morphing se encuentra habilitado.* Retorno *bool: Habilitación del morphing.*
habilitarMorph	Descripción *Habilita/Deshabilita el morphing.* Parámetros ● *bool habilitar: Indica si habilitar o deshabilitar el morphing.*
pollerasHabilitadas	Descripción *Obtiene si las polleras se encuentran habilitado.* Retorno *bool: Habilitación de las polleras.*
habilitarPolleras	Descripción *Habilita/Deshabilita las polleras.* Parámetros ● *bool habilitar: Indica si habilitar o deshabilitar las polleras.*
getLargoPolleras	Descripción *Obtiene el largo vertical de las polleras.* Retorno *float: Largo de las polleras en metros.*
setLargoPolleras	Descripción *Establece el largo vertical de las polleras.* Parámetros ● *float largo: Largo de las polleras en metros.*
reset	Descripción *Establece la configuración por defecto.*

Datos generales	
Clase	*CCamara*
Responsabilidad	*Esta clase es responsable de calcular las matrices de vista y proyección posición y orientación en el espacio de la cámara virtual de la escena.*
Deriva de	
Métodos	
CCamara (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
getPosicion	Descripción *Obtiene la posición de la cámara.* Retorno *D3DXVECTOR3: Posición de la cámara en espacio de la escena.*
setPosicion	Descripción *Establece la posición de la cámara.* Parámetros ● *const D3DXVECTOR3 &pos: Posición de la cámara en espacio de la escena.*
getRotacion	Descripción *Obtiene la rotación de la cámara.* Retorno *D3DXVECTOR3: Rotación de la cámara en espacio de la escena.*
setRotacion	Descripción *Establece la rotación de la cámara.* Parámetros ● *const D3DXVECTOR3 &rot: Rotación de la cámara en espacio de la escena.*
getFovY	Descripción *Obtiene el campo visual vertical de la cámara.* Retorno *float: Campo visual vertical de la cámara en radianes.*
setFovY	Descripción *Establece el campo visual vertical de la cámara.* Parámetros ● *float fovy: Campo visual vertical de la cámara en radianes.*
getRelAspecto	Descripción *Obtiene la relación de aspecto ancho-alto del viewport.* Retorno *float: Relación de aspecto ancho-alto del viewport en píxeles.*
setRelAspecto	Descripción *Establece la relación de aspecto ancho-alto del viewport.*

	Parámetros ● *float asp: Relación de aspecto ancho-alto del viewport en píxeles.*
actualizar	Descripción *Actualiza las matrices de vista y proyección.* Parámetros ● *bool vista: Indica si actualizar la matriz vista.* ● *bool proyeccion: Indica si actualizar la matriz proyección.*
getMatrizVista	Descripción *Obtiene la matriz vista de la cámara.* Retorno *D3DXMATRIX: Matriz vista de la cámara.*
getMatrizProyeccion	Descripción *Obtiene la matriz proyección de la cámara.* Retorno *D3DXMATRIX: Matriz proyección de la cámara.*

Datos generales	
Clase	*CViewFrustum*
Responsabilidad	*Es responsable de calcular y mantener el frustum de la cámara para realizar la técnica que permite realizar la técnica de View frustum culling.*
Deriva de	
Métodos	
CViewFrustum (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
actualizar	Descripción *Actualiza el frustum a partir de la cámara pasada como párametro.* Parámetros ● *CCamara *camara: Cámara de la cual se calcula el frustum.*
intersectaCajaEnvolvente	Descripción *Indica si el frustum y una cajaEnvolvente se intersectan en el espacio.* Retorno *EResIntersec: Indica el tipo de intersección realizada en relación a la caja envolvente: {ERI_ADENTRO, ERI_AFUERA, ERI_INTERSECTA}.* Parámetros ● *CCajaEnvolvente *caja: Caja envolvente con la cual realizar el chequeo de intersección.*

Datos generales	
Clase	*CModoVisualizador (Abstracta)*
Responsabilidad	*Esta clase es responsable de variar el comportamiento de la aplicación modelando los distintos modos de funcionamiento en que puede encontrarse la misma.*
Deriva de	
Métodos	
CModoVisualizador *(constructor)* *(Protegido)*	Descripción *Inicializa por defecto los atributos de la clase.*
entrar *(virtual pura)*	Descripción *Evento de entrada al modo.*
salir *(virtual pura)*	Descripción *Evento de salida del modo.*
eventoTeclado *(virtual pura)*	Descripción *Evento del teclado.* Parámetros ● *bool presionada: Indica si la tecla fue presionada o liberada.* ● *UINT tecla: indica que tecla fue presionada o liberada.*
actualizar *(virtual pura)*	Descripción *Realiza el trabajo del modo.*
renderizar *(virtual pura)*	Descripción *Renderiza los elementos de pantalla que correspondan al modo.*

Datos generales	
Clase	*CModoTest*
Responsabilidad	*Esta clase lleva a cabo los tests de performance y guardar en disco el resultado de los mismos.*
Deriva de	*CModoVisualizador*
Métodos	
CModoTest *(constructor)* *(privado)*	Descripción *Inicializa por defecto los atributos de la clase.*
~CModoTest *(destructor)* *(privado)*	Descripción *Destruye la clase liberando los recursos adquiridos.*
getInstancia	*Descripción* *Singleton para obtener la única instancia de la clase.* Retorno *static CModoTest *: Única instancia de la clase.*
entrar	Descripción *Abre archivo ".csv" para almacenaje de estadísticas de performance.*
salir	Descripción

	Cierra archivo de estadísticas.
eventoTeclado	Descripción *Sale del modo si se presiona la tecla ESC.* Parámetros ● *bool presionada: Indica si la tecla fue presionada o liberada.* ● *UINT tecla: indica que tecla fue presionada o liberada.*
actualizar	Descripción *Actualiza el controlador de la cámara y almacena un registro de estadísticas de performance.*
renderizar	Descripción *Renderiza el terreno y un mensaje de salida del modo.*

Datos generales	
Clase	*CModoNavegación*
Responsabilidad	*Esta clase permite la libre navegación por parte del usuario por la escena virtual.*
Deriva de	*CModoVisualizador*
Métodos	
CModoNavegación (constructor) (privado)	Descripción *Inicializa por defecto los atributos de la clase.*
~CModoNavegación (destructor) (privado)	Descripción *Destruye la clase liberando los recursos adquiridos.*
getInstancia	*Descripción* *Singleton para obtener la única instancia de la clase.* Retorno *static CModoNavegacion *: Única instancia de la clase.*
entrar	Descripción *Oculta el puntero del mouse e inicializa el controlador de la cámara.*
salir	Descripción *Habilita la visualización del puntero del mouse.*
eventoTeclado	Descripción *Sale del modo si se presiona la tecla ESC. En caso contrario, redirecciona el evento al controlador de la cámara.* Parámetros ● *bool presionada: Indica si la tecla fue presionada o liberada.* ● *UINT tecla: indica que tecla fue presionada o liberada.*
actualizar	Descripción *Captura el movimiento del puntero del mouse y lo envía al controlador de la cámara.*
renderizar	Descripción *Renderiza el terreno y los mensajes que indican como controlar la*

	cámara.

Datos generales	
Clase	*CModoConfiguración*
Responsabilidad	*Esta clase habilita la interacción del menú principal en pantalla para permitir la configuración del algoritmo de visualización.*
Deriva de	*CModoVisualizador*
Métodos	
CModoVisualizador (constructor) (privado)	Descripción *Inicializa por defecto los atributos de la clase.*
~CModoVisualizador (destructor) (privado)	Descripción *Destruye la clase liberando los recursos adquiridos.*
getInstancia	*Descripción* *Singleton para obtener la única instancia de la clase.* Retorno *static CModoConfiguracion *: Única instancia de la clase.*
entrar	Descripción *Habilita la visualización del menú de controles de configuración del algoritmo.*
salir	Descripción *No realiza operación.*
eventoTeclado	Descripción *Cambia al modo navegación si se presiona la barra espaciadora.* Parámetros ● *bool presionada: Indica si la tecla fue presionada o liberada.* ● *UINT tecla: indica que tecla fue presionada o liberada.*
actualizar	Descripción *Actualiza el menú de controles gráficos.*
renderizar	Descripción *Renderiza el terreno y los controles del menú de configuración.*

Datos generales	
Clase	*CControladorCamara (Abstracta)*
Responsabilidad	*Provee la funcionalidad para controlar la cámara en base a la velocidad configurable a la cual la cámara se debe trasladar y rotar y al tiempo transcurrido entre cuadros (animación).*
Deriva de	
Métodos	
CControladorCamara (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*

~CControladorCamara (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
getCamara	Descripción *Retorna la cámara animada por el controlador.* Retorno *CCamara *: Cámara animada.*
setCamara	Descripción *Establece la cámara a animar.* Parámetros ● *CCamara *cam: Cámara a animar.*
getVelocTraslacion	Descripción *Retorna la velocidad de avance de la cámara.* Retorno *float: Velocidad de avance en km/h.*
setVelocTraslacion	Descripción *Establece la velocidad de avance de la cámara.* Parámetros ● *float veloc: Velocidad de avance en km/h.*
getVelocRotacion	Descripción *Retorna la velocidad de rotación de la cámara.* Retorno *float: Coeficiente de velocidad de rotación en base al movimiento del puntero del mouse.*
setVelocRotacion	Descripción *Establece la velocidad de avance de la cámara.* Parámetros ● *float veloc: Coeficiente de velocidad de rotación en base al movimiento del puntero del mouse.*
iniciar (virtual)	Descripción *Inicia el control temporizado para la animación.*
actualizarCamara (virtual)	Descripción *Actualiza la posición y rotación de la cámara.*

Datos generales	
Clase	*CControladorCamaraUsuario*
Responsabilidad	*Modela la funcionalidad específica necesaria en los modos de funcionamiento navegación de la aplicación, animando la cámara dependiendo de la entrada de teclado y del movimiento del mouse.*
Deriva de	*CControladorCamara*
Métodos	
CControladorCamara (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*

~CControladorCamara (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
iniciar	Descripción *Inicia el control de la cámara estableciendo la posición inicial del mouse y teclas como no presionadas.* Parámetros ● *POINT posMouse: Posición del mouse al comenzar el control de la cámara.* ● *bool reiniciarTeclas: Indica establecer las teclas como no presionadas.*
eventoMovimientoMouse	Descripción *Rota la cámara ante la nueva posición del puntero del mouse.* Parámetros ● *POINT posMouse: Nueva posición del puntero del mouse.*
eventoTeclado	Descripción *Traslada la cámara ante las el cambio de estado del teclado.* Parámetros ● *bool presionada: Indica si la tecla fue presionada o liberada.* ● *UINT tecla: indica que tecla fue presionada o liberada.*

Datos generales	
Clase	*CControladorCamaraTest*
Responsabilidad	*Modela la funcionalidad específica necesaria en los modos de funcionamiento test de la aplicación, configurando la posición y dirección iniciales en función del test realizado.*
Deriva de	*CControladorCamara*
Métodos	
CControladorCamara (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CControladorCamara (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
iniciar (virtual)	Descripción *Inicia el control de la cámara estableciendo la posición y rotación fijas de la cámara para llevar adelante un test similar en todos los casos.*

Datos generales	
Clase	*CTemporizador*
Responsabilidad	*Provee bservicios temporización.*
Deriva de	
Métodos	

CTemporizador (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CTemporizador (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
iniciar	Descripción *Inicia el contador de tiempo.*
detener	Descripción *Detiene el contador de tiempo.*
tiempo	Descripción *Obtiene el tiempo transcurrido desde el inicio del contador.* Retorno *TTimestamp: Tiempos transcurrido en milisegundos.*

Datos generales	
Clase	*CTerreno*
Responsabilidad	*Esta clase representa el conjunto de datos de un modelo de terreno, y es responsable de ejecutar el algoritmo de visualización diseñado.*
Deriva de	
Métodos	
CTerreno (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CTerreno (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
cargar	Descripción *Carga un archivo ".quad" y mapas asociados desde disco.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.* ● *ID3D11DeviceContext* d3dContext: Contexto de renderización del dispositivo DX11.* ● *wchar_t *arch: Path del archivo ".quad" origen.*
liberarRecursos	Descripción *Libera los recursos adquiridos por la clase.*
getModosRenderizado	Descripción *Obtiene los modos de renderizado activados.* Parámetros ● *bool &textura: Indica si se renderiza el terreno mediante mapa de color.* ● *bool &wireframe: Indica si se renderiza el "esqueleto" del mallado multirresolución.* ● *bool &cajas: Indica si se renderizan las cajas envolventes de los bloques de terreno.*

setModosRenderizado	Descripción *Activa/Desactiva los modos de renderizado.* Parámetros ● *bool textura: Activa/Desactiva el renderizado del terreno mediante mapa de color.* ● *bool wireframe: Activa/Desactiva el renderizado del "esqueleto" del mallado multirresolución.* ● *bool cajas: Activa/Desactiva el renderizado de las cajas envolventes de los bloques de terreno.*
getUmbralErrorPercibido	Descripción *Obtiene el umbral de percepción visual.* Retorno *float: Umbral de percepción en píxeles.*
setUmbralErrorPercibido	Descripción *Establece el umbral de percepción visual.* Parámetros ● *float umbral: Umbral de percepción en píxeles.*
morphHabilitado	Descripción *Obtiene si el morphing se encuentra habilitado.* Retorno *bool: Habilitación del morphing.*
habilitarMorph	Descripción *Habilita/Deshabilita el morphing.* Parámetros ● *bool habilitar: Indica si habilitar o deshabilitar el morphing.*
pollerasHabilitadas	Descripción *Obtiene si las polleras se encuentran habilitado.* Retorno *bool: Habilitación de las polleras.*
habilitarPolleras	Descripción *Habilita/Deshabilita las polleras.* Parámetros ● *bool habilitar: Indica si habilitar o deshabilitar las polleras.*
getLargoPolleras	Descripción *Obtiene el largo vertical de las polleras.* Retorno *float: Largo de las polleras en metros.*
setLargoPolleras	Descripción *Establece el largo vertical de las polleras.* Parámetros ● *float largo: Largo de las polleras en metros.*

getDimensionTotal	Descripción *Obtiene la dimensión total del terreno.* Retorno *D3DXVECTOR2: Dimensión 2D del terreno.*
getEstadisticasRenderizacion	Descripción *Obtiene las estadísticas de renderización desde el último reseteo de contadores.* Parámetros ● *int *numBloques: Número de bloques renderizados.* ● *int *numTris: Número de triángulos renderizados.*
resetEstadisticasRenderizacion	Descripción Pone en cero los contadores estadísticos de geometría renderizada.
renderizar	Descripción *Renderiza el terreno en pantalla.* Parámetros ● *CCamara *camara: Cámara desde la cual el terreno es observado.* ● *SDimViewport *dimViewport: Dimensión del viewport necesario para el cálculo y proyección de los errores de aproximación geométrica.*

Datos generales	
Clase	*CQuadtree*
Responsabilidad	*Modela la estructura de datos quadtree.*
Deriva de	
Métodos	
CQuadtree (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CQuadtree (destructor)	Descripción *Destruye el quadtree liberando memoria alojada.*
getRaiz	Descripción *Retorna la raíz del quadtree.* Retorno *CNodoQuad *: Raíz del quadtree.*
setRaiz	Descripción *Asigna la raíz del quadtree.* Parámetros ● *CNodoQuad *raiz: Nueva raíz del quadtree.*
getInfoConstruccion	Descripción *Retorna la información de construcción generada en el preprocesamiento.* Retorno

	*SInfoConstruccion *: Información de construcción del quadtree.*

Datos generales	
Clase	*CNodoQuad*
Responsabilidad	*Modela un nodo de la estructura quadtree. Permite mantener y administrar subárboles en memoria.*
Deriva de	
Métodos	
CNodoQuad (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CNodoQuad (destructor)	Descripción *Destruye el nodo y todo su subárbol.*
getCajaEnv	Descripción *Retorna la caja envolvente del nodo.* Retorno *CCajaEnvolvente &: Caja envolvente del nodo.*
setCajaEnv	Descripción *Asigna caja envolvente del nodo.* Parámetros ● *const CCajaEnvolvente &caja: Nueva caja envolvente del nodo.*
getCoordMapaMin	Descripción *Retorna la coordenada mínima de la región abarcada por el nodo en el mapa.* Retorno *SCoordMapa: Coordenada mínima del mapa.*
setCoordMapaMin	Descripción *Asigna la coordenada mínima de la región abarcada por el nodo en el mapa.* Parámetros ● *x: Coordenada x mínima del mapa.* ● *y: Coordenada y mínima del mapa.*
getCoordMapaMax	Descripción *Retorna la coordenada máxima de la región abarcada por el nodo en el mapa.* Retorno *SCoordMapa: Coordenada máxima del mapa.*
setCoordMapaMax	Descripción *Asigna la coordenada máxima de la región abarcada por el nodo en el mapa.* Parámetros ● *x: Coordenada x máxima del mapa.*

	● *y: Coordenada y máxima del mapa.*
getErrorAprox	Descripción *Retorna el error de aproximación geométrica del nodo.* Retorno *float: Error de aproximación geométrica.*
setErrorAprox	Descripción *Asigna el error de aproximación geométrica del nodo.* Parámetros ● *float errorAprox: Error de aproximación geométrica.*
getHijo	Descripción *Retorna el hijo indicado del nodo.* Retorno *CNodoQuad *: Nodo hijo.* Parámetros ● *int idx: Índice de hijo entre 0 y 3.*
setHijo	Descripción *Establece un hijo del nodo.* Parámetros ● *int idx: Índice de hijo a establecer.* ● *CNodoQuad *nodo: Nodo a establecer como hijo.*
esHoja	Descripción *Consulta si los 4 hijos son NULL.* Retorno *bool: Indica si es o no un nodo hoja del quadtree.*

Datos generales	
Clase	*CCargadorQuadtree*
Responsabilidad	*Realiza la carga de un quadtree desde disco en el formato de archivo definido a tal propósito.*
Deriva de	
Métodos	
CQuadtreeSaver (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CQuadtreeSaver (destructor)	Descripción *Destruye la clase.*
cargar	Descripción *Carga el quadtree desde desde disco.* Retorno *EResultado: Enumeración de tipos de error.*

	Parámetros ● *wchar_t *arch: Path al archivo origen.* ● *CQuadtree *qt: Quadtree cargado desde disco.*
cargarInfo *(privado)*	Descripción *Carga la información de construcción del quadtree.* Parámetros ● *FILE *fp: Descriptor de archivo.* ● *SInfoConstruccion info: Parámetros de construcción del quadtree.*
cargarArbol *(privado)*	Descripción *Carga recursivamente un subárbol del quadtree.* Parámetros ● *FILE *fp: Descriptor de archivo.* ● *CNodoQuad *nodo: Nodo padre del subárbol.*
cargarNodo *(privado)*	*Descripción* *Carga los datos del nodo desde el archivo origen.* *Parámetros* ● *FILE *fp: Descriptor de archivo.* ● *CNodoQuad *nodo: Nodo del quadtree.*

Datos generales	
Clase	*CTextura*
Responsabilidad	*Esta clase es responsable de cargar desde disco texturas utilizadas como mapas de altura y color, y de la administración de sus recursos en memoria.*
Deriva de	
Métodos	
CTextura *(constructor)*	Descripción *Inicializa por defecto los atributos de la clase.*
~CTextura *(destructor)*	Descripción *Destruye la clase liberando los recursos adquiridos.*
cargar	Descripción *Carga la textura desde disco.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico y sus recursos.* ● *wchar_t *arch: Path del archivo de textura en disco.* ● *int flags: Expone los flags de la librería DX11 para la carga de recursos en memoria de GPU.*
liberarRecursos	Descripción

	Libera los recursos adquiridos por la clase.

Datos generales	
Clase	*CRecolectorNodos*
Responsabilidad	*Esta clase es responsable de recorrer el quadtree, conformando la lista de nodos a renderizar.*
Deriva de	
Métodos	
CRecolectorNodos (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CRecolectorNodos (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
cargar	Descripción *Carga el mapa de bits desde disco.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *wchar_t *arch: Path del archivo en disco.* ● *int flags: Expone los flags de la librería FreeImage para la carga de bitmaps.*
getQuadtree	Descripción *Retorna el quadtree sobre el que se realiza la recolección de nodos.* Retorno *CQuadtree *: Quadtree asociado.*
setQuadtree	Descripción *Establece el quadtree sobre el que se realiza la recolección de nodos.* Parámetros ● *CQuadtree *qt: Quadtree a asociar.*
recolectar	Descripción *Recolecta nodos del quadtree asociado dependiendo del punto de observación (cámara) y el umbral (tau) de tolerancia visual.* Retorno *int: Número de nodos recolectados.* Parámetros ● *CCamara *camara: Cámara desde la cual el terreno es observado.* ● *SDimViewport *dimViewport: Dimensión del viewport necesario para el cálculo y proyección de los errores de aproximación geométrica.* ● *float umbralTau: Umbral de percepción visual.*
getListaNodos	Descripción *Retorna la lista de nodos e información asociada recolectados del*

	quadtree. Retorno *SInfoNodo *: Lista de nodos e información recolectados.*
getNumNodos	Descripción *Retorna el número de nodos recolectados.* Retorno *int: Número de nodos recolectados.*

distanciaSubdivision (privado)	Descripción *Calcula la distancia de subdivisión utilizada en la interpolación del geomorphing.* Retorno *float: Distancia en metros.* Parámetros ● *CCamara *camara: Cámara desde la cual el terreno es observado.* ● *CNodoQuad *nodo: Nodo a visualizar.*
recorrerSubarbol (privado)	Descripción *Recorre un subárbol recolectando nodos.* Parámetros ● *CCamara *camara: Cámara desde la cual el terreno es observado.* ● *CNodoQuad *nodo: Nodo padre del suárbol.* ● *float escala: Escala de LOD del nodo padre.* ● *float distUnific: Distancia de unificación utilizada en la interpolación del geomorphing en metros.*
agregarNodo (privado)	Descripción *Agrega un nodo a la lista de nodos recolectados.* Parámetros ● *CNodoQuad *nodo: Nodo a agregar.* ● *float escala: Escala de LOD del nodo.* ● *float dist: Distancia entre la cámara y el nodo.* ● *float distSubdiv: Distancia de subdivisión del geomorphing en metros.* ● *float distUnific: Distancia de unificación del geomorphing en metros.*

Datos generales	
Clase	*CCajaEnvolvente*
Responsabilidad	*Modela una caja alineada a los ejes utilizada para envolver la*

	geometría de los nodos del quadtree en el espacio.
Deriva de	
Métodos	
CCajaEnvolvente (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
CCajaEnvolvente (constructor)	Descripción *Inicializa los la caja envolvente con los vértices indicados por parámetro.* Parámetros ● *D3DXVECTOR3 min: Vértice mínimo de la caja en espacio de objeto.* ● *D3DXVECTOR3 max: Vértice máxima de la caja en espacio de objeto.*
getMin	Descripción *Retorna el vértice mínimo de la caja envolvente.* Retorno *D3DXVECTOR3: Vértice mínimo en espacio de objeto.*
setMin	Descripción *Asigna el vértice mínimo de la caja envolvente.* Parámetros ● *const D3DXVECTOR3 &min: Vértice mínimo en espacio de objeto.*
getMax	Descripción *Retorna el vértice máximo de la caja envolvente.* Retorno *D3DXVECTOR3: Vértice máximo en espacio de objeto.*
setMax	Descripción *Asigna el vértice máximo de la caja envolvente.* Parámetros ● *const D3DXVECTOR3 &max: Vértice máximo en espacio de objeto.*
getCorners	Descripción *Obtiene los 8 vértices de la caja envolvente.* Parámetros ● D3DXVECTOR3 corners[8]*: Vértices de las 8 esquinas en espacio de objeto.*

Datos generales	
Clase	*CBloque*
Responsabilidad	Modela el bloque genérico de terreno. Su función principal es la de crear los buffers que contienen la geometría necesaria para su

	utilización en el pipeline.
Deriva de	
Métodos	
CBloque *(constructor)*	Descripción *Inicializa por defecto los atributos de la clase.*
~CBloque *(destructor)*	Descripción *Destruye la clase liberando los recursos adquiridos.*
inicializar	Descripción Inicializa los buffers que contienen la geometría que define el bloque genérico de terreno. Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.* ● *ID3D11DeviceContext* d3dContext: Contexto de renderización del dispositivo DX11.*
liberarRecursos	Descripción *Libera los recursos adquiridos por la clase*.
renderizar	Descripción *Envía a renderizar el bloque de terreno.* Parámetros ● *bool polleras: Indica si las polleras deben renderizarse o no.*
actualizarVerticesCaja	Descripción *Actualiza el buffer de vértices para la renderización de las cajas envolventes.* Parámetros ● *CCajaEnvolvente *caja: Caja envolvente con la cual actualizar los vértices del buffer.*
renderizarCaja	Descripción *Envía a renderizar el buffer de vértices para cajas envolventes.*
crearVertices *(privado)*	Descripción *Crea el buffer de vértices del bloque genérico de terreno.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.*
crearIndices *(privado)*	Descripción *Crea el buffer de índices del bloque genérico de terreno.* Retorno *EResultado: Enumeración de tipos de error.*

	Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.*
crearVerticesCaja (privado)	Descripción *Crea el buffer de vértices para cajas envolventes.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.*

Datos generales	
Clase	*CTecnicaShaders*
Responsabilidad	*Esta clase es responsable de cargar y compilar los shaders desde disco y de posibilitar la comunicación con los mismos para actualizar los parámetros necesarios en la renderización de cada bloque de terreno.*
Deriva de	
Métodos	
CTecnicaShaders (constructor)	Descripción *Inicializa por defecto los atributos de la clase.*
~CTecnicaShaders (destructor)	Descripción *Destruye la clase liberando los recursos adquiridos.*
inicializar	Descripción *Carga y compila los shaders a ejecutar en el pipeline gráfico de la GPU, y crea los buffers de constantes para la comunicación con la misma.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.* ● *ID3D11DeviceContext* d3dContext: Contexto de renderización del dispositivo DX11.*
liberarRecursos	Descripción *Libera los recursos adquiridos por la clase*.
actualizarCBEventual	Descripción *Actualiza el buffer de constantes "Eventual" alojado en memoria de GPU.* Parámetros ● *const CB_EVENTUAL &cb: Datos del buffer.*
actualizarCBPorCuadro	Descripción

	Actualiza el buffer de constantes "Por Cuadro" alojado en memoria de GPU. Parámetros ● *const CB_POR_CUADRO &cb: Datos del buffer.*
actualizarCBPorBloque	Descripción *Actualiza el buffer de constantes "Por Bloque" alojado en memoria de GPU.* Parámetros ● *const CB_POR_BLOQUE &cb: Datos del buffer.*
establecerTexturas	Descripción *Establece las texturas del mapa altura y color a ser muestreadas por los shaders.* Parámetros ● *CTextura *mapaAltura: Textura del mapa de altura.* ● *CTextura *mapaColor: Textura del mapa de color.*
establecerTexturas	Descripción Configura el contexto de renderización activando los shaders necesarios en el pipeline gráfico. Parámetros ● *EModoRenderizado modo: Enumeración indicando de renderizado {EMR_TEXTURA, EMR_WIREFRAME, EMR_CAJAENVOLVENTE}.*
cargarShaders	Descripción *Carga y compila los shaders a ejecutar en el pipeline gráfico de la GPU.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.*
crearConstantBuffers	Descripción *Crea los buffers de constantes para la comunicación con la GPU.* Retorno *EResultado: Enumeración de tipos de error.* Parámetros ● *ID3D11Device* d3dDevice: Dispositivo DX11 para interacción con el pipeline gráfico.*

Apéndice B - MANUAL DE USUARIO

Este apéndice describe el modo de utilización de las aplicaciones desarrolladas en este trabajo.

La sección B.1 describe como ejecutar el preprocesador y el visualizador. La sección B.2 y B.3 describen el modo de uso de estas aplicaciones, respectivamente.

B.1 Instalación y ejecución

Tanto la aplicación de visualización como la de preprocesamiento no requieren instalación para su funcionamiento, pueden ser copiadas al disco o ejecutadas directamente del el CD que acompaña este informe, aunque se recomienda la primera para un correcto funcionamiento de todas las funcionalidades disponibles.

Para la ejecución de la aplicación de visualización se requiere de una PC con sistema operativo Windows 7 o superior, 1GB de memoria RAM, y una placa gráfica (GPU) con soporte de Shader Model 5 (NVIDIA GeForce serie 400 o AMD Radeon HD serie 5000 en adelante). Un monitor con soporte de resolución HD 1920x1080. Mouse y teclado.

Ubicación de aplicaciones:

- **Preprocesador:** "[Unidad_CD]:\aplicaciones\bin\preprocesador\"
  Ejecutable: "Preprocesador.exe"
- **Visualizador:** "[Unidad_CD]:\aplicaciones\bin\visualizador\"
  Ejecutable: "Visualizador.exe"

B.2 Preprocesador

B.2.1 Parámetros de configuración

El preprocesador inicia con la ventana de la figura B.1. Su función es la de permitir el ingreso de todos los parámetros necesarios en la generación y almacenado en disco del *quadtree*, para su posterior carga y utilización por el visualizador.

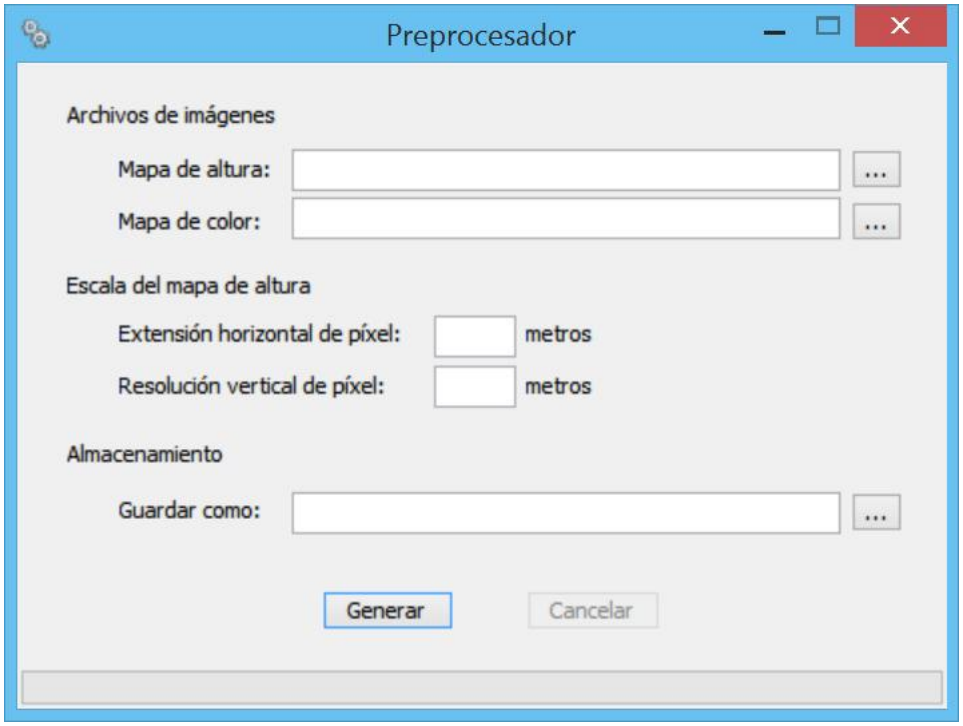

Figura B.1: Ventana principal del preprocesador.

Los controles de la ventana han sido agrupados en 3 secciones:

- **Archivos de imágenes:** Permite el ingreso del path hacia los archivos de imágen de mapa de altura (*heightmap*) y mapa de color (*colormap*). Mediante los botones rotulados con "…" se accede a una ventana de diálogo que permite seleccionar los archivos de imagen. Los archivos de imágenes soportados son BMP, JPEG, PNG y TIFF en 8 y 16 bits según formato. Una vez seleccionado, el path es reflejado en el cuadro de texto correspondiente para su rápida modificación si fuese necesario.
- **Escala del mapa de altura:** Admite el ingreso de la extensión y resolución de píxel. La extensión corresponde a la distancia entre muestras sobre el plano del terreno. La resolución representa cada paso o cambio de valor de la muestra, correspondiendo a 1/255 de la altura máxima posible en imágenes de 8 bits. Los valores se expresan en metros y pueden ingresarse valores fraccionales, debiendo utilizarse "." o "," como separador de decimales según configuración del sistema.
- **Almacenamiento:** Permite el ingreso del path hacia el archivo que contendrá el *quadtree* generado. Mediante el botón rotulado con "…" se accede a una ventana de diálogo que permite seleccionar el archivo ".quad" de destino. Una vez seleccionado, el path es reflejado en el cuadro de texto para su rápida modificación si fuese necesario.

B.2.2 Generación del quadtree

Una vez ingresados todos los parámetros de configuración (descriptos en la sección anterior), la generación del *quadtree* comienza presionando el botón "Generar". Durante el proceso de construcción del *quadtree* una barra de progreso ubicada al pie de la ventana indica el avance del proceso de generación. Si fuese necesario este proceso puede cancelarse mediante el botón "Cancelar".

Al completarse la barra de progreso, el proceso de generación habrá terminado. Una ventana de información se mostrará indicando esta situación y el número de coeficientes de error corregidos como se muestra en la figura B.2.

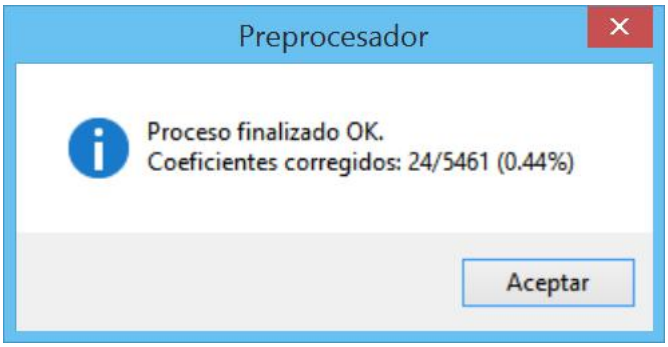

Figura B.2: Proceso finalizado OK.

B.3 Visualizador

Con el objetivo de obtener una aplicación de visualización completa, fácil de usar, e interactiva con el usuario, la aplicación de visualización ha sido extendida soportando tres modos de funcionamiento: configuración, navegación y test.

- **Configuración:** Este modo permite cargar los archivos ".quad" generados por el preprocesador, y configurar los aspectos de visualización más importantes del algoritmo como ser el umbral error de aproximación, deshabilitación del *geomorphing*, entre otros. Desde este modo es posible acceder a los modos Navegación y Test, ya sea por menú o por teclado.
- **Navegación:** Este modo permite navegar por la escena realizando un vuelo libre por el espacio 3D. Para esto se utiliza el mouse y el teclado controlando la rotación y traslación del punto de observación. Este modo permite apreciar el funcionamiento del algoritmo en tiempo real ya que toda la visualización se realiza según la configuración realizada en el modo anterior.
- **Test:** Este modo fue diseñado para llevar a cabo el test del algoritmo. En este modo la cámara es controlada automáticamente mientras se registran estadísticas de performance utilizadas luego para analizar el comportamiento del algoritmo.

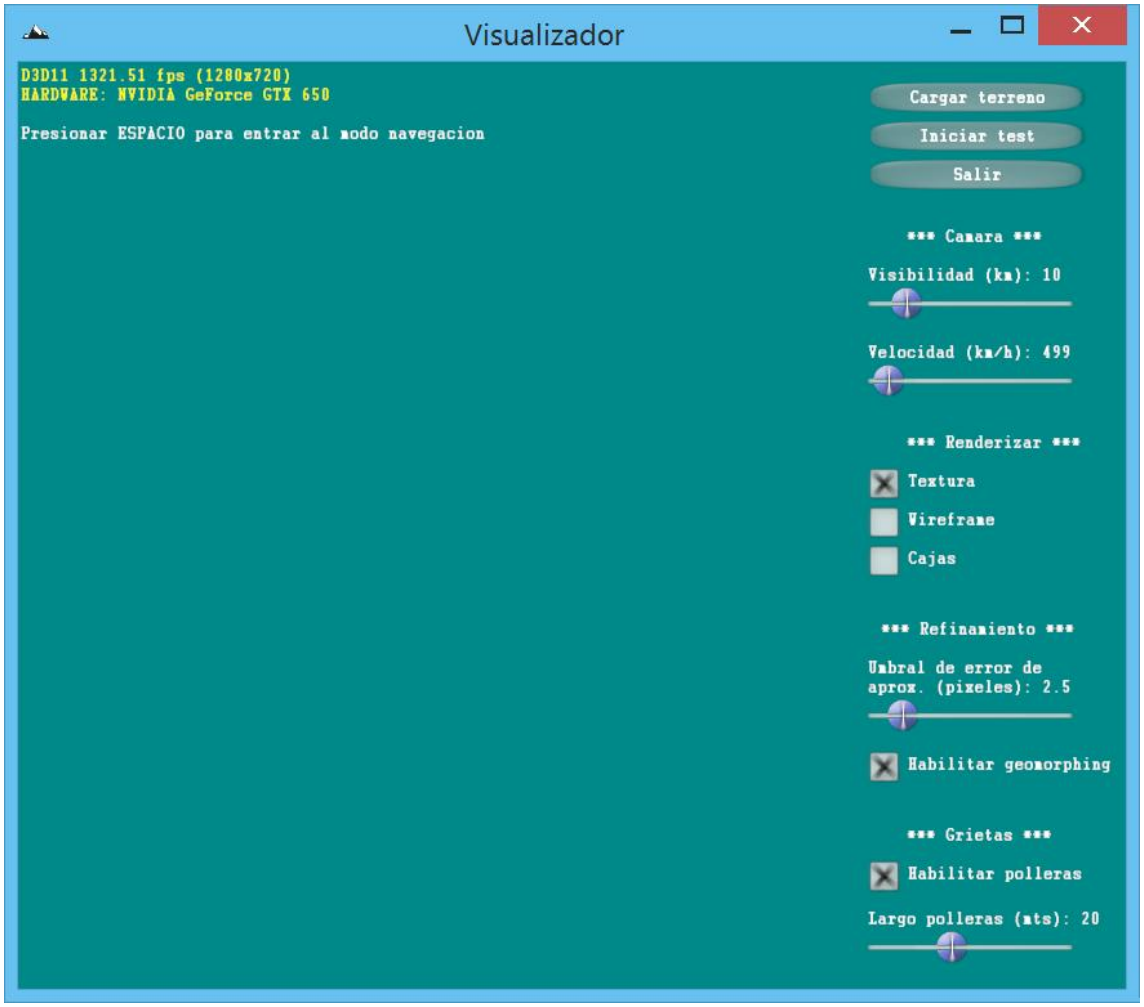

Figura B.3: Ventana principal del visualizador (la imagen de la captura ha sido reducida en ancho para su mejor visualización).

B.3.1 Modo Configuración

La aplicación inicia en este modo desplegando la ventana de la figura B.3. En la misma pueden identificarse tres grupos de elementos mostrados en pantalla:

Rótulos informativos

En la esquina superior izquierda se ubican un grupo rótulos o textos informativos que indican la frecuencia de renderización en FPS (Frames Per Second), la resolución de pantalla, el modelo de GPU, entre otros.

Botones de operación

En la esquina superior derecha se sitúa un grupo con los siguientes tres botones:

- ***Cargar terreno*****:** Despliega una ventana de diálogo para la selección de archivos ".quad" generados por el preprocesador. Una vez seleccionado, el *quadtree* es cargado junto a los mapas de altura y color correspondientes, y el terreno comienza a ser visualizado en pantalla.
- ***Iniciar test*****:** Comienza la ejecución de un test de performance (sección B.3.3).
- ***Salir*****:** Sale del visualizador.

Menú de configuración

En el lateral derecho de la ventana principal se ubican los controles dedicados a la configuración de los principales aspectos de visualización algoritmo, de la cámara y de otros elementos de debug. Toda configuración hecha en este menú es respetada en los modos Navegación y Test.

Estos controles se encuentran agrupados en 4 secciones:

- ***Camara*****:** Contiene 2 *slide bars* que permiten modificar la distancia visual (Km) y la velocidad de traslación (Km/h) de la cámara.
- ***Renderizar*****:** Contiene 3 *check boxes*. ***Textura*** es el primero de ellos e indica si se debe realizar la renderización del terreno coloreando su superficie con el mapa de color. Los otros dos ***Wireframe*** y ***Cajas*** son de debug y permiten renderizar el "esqueleto" del mallado multirresolución y las cajas envolventes de los bloques de terreno, respectivamente.
- ***Refinamiento*****:** Contiene un *slide var* que permite modificar la tolerancia al error de aproximación en unidad de píxeles y un *check box* para la habilitación/deshabilitación de la técnica *geomorphing*.
- ***Grietas*****:** Contiene un *check box* para habilitar/deshabilitar la renderización de las polleras, y un *slide bar* que permite asiganar el largo de las mismas en tiempo real.

B.3.2 Modo Navegación

A este modo se ingresa presionando la barra espaciadora desde el modo configuración. Al ingresar a este modo, el puntero del mouse es ocultado. Este modo permite básicamente la libre navegación por la escena. La visualización del terreno se realiza según la configuración realizada en el modo configuración. El punto de observación puede modificarse mediante los siguientes controles:

- ***Movimiento del mouse:*** Controla la rotación de la cámara.
- ***Teclas W.A.S.D:*** Controla el avance, retroceso y translación horizontal de la cámara.
- ***Teclas Q.E:*** Controla la traslación vertical de la cámara.
- ***Tecla ESC:*** Retorna al modo de configuración.

B.3.3 Modo Test

A este modo se ingresa presionando el botón ***Iniciar test*** desde el modo configuración.

Al entrar en este modo un archivo ".csv" (*Comma Separated Values*) es creado y un test de performance es ejecutado. Además el menú de configuración es ocultado de modo que no influya la performance del test.

Un test consiste en un sobrevuelo comenzando en una esquina del terreno cargado, terminando en el centro del mismo. Durante el transcurso del test se calculan los tiempos de cuadros, bloques y triángulos generados mientras la cámara se traslada automáticamente hacia el centro del terreno. Estos valores son almacenados en el archivo ".csv", en el cual también se incluyen los datos de la configuración de visualización utilizada durante el test y la duración del mismo.

Al finalizar el test se vuelve automáticamente al modo de configuración, donde los parámetros de visualización pueden ser modificados para volver a iniciar otro test distinto.

Los archivos de test son almacenados en la carpeta "\visualizador\tests\[NOMBRE].cvs", donde NOMBRE corresponde al nombre del archivo ".quad" cargado, seguido del umbral de error de aproximación utilizado en el test.