UNIVERSITAT JAUME I
Department of Languages and Computer Systems

Enhancing the Rendering of Natural Scenes: Rain, Sea and Terrain

Ph. D. dissertation
Anna Puig Centelles

Advisors: Dr. Miguel Chover Sellés
and Dr. Mateu Sbert Casasayas

Castelló, February 2010

UNIVERSITAT JAUME·I
Departament de Llenguatges i Sistemes Informàtics

Mejoras en la Visualización de Escenas Naturales: Lluvia, Mar y Terreno

Tesis doctoral
Anna Puig Centelles
Director: Dr. Miguel Chover Sellés y Dr. Mateu Sbert Casasayas

Castellón, Febrero 2010

Los escenarios de exteriores en aplicaciones gráficas incluyen generalmente fenómenos naturales que son complicados de simular en tiempo real. Así, en el campo de la visualización interactiva se han propuesto soluciones que son capaces de ofrecer realismo a un coste elevado. El principal objetivo de esta tesis es presentar un conjunto de técnicas para mejorar la visualización en tiempo real de estos fenómenos naturales. Proponemos una solución para facilitar la creación y el control de escenas de lluvia por medio de un nuevo método de nivel de detalle. Posteriormente, esta técnica se ha extendido para incluir el manejo de colisiones de las gotas con el entorno y la simulación de las salpicaduras. Como una mejora sobre las técnicas de simulación de mar, se ha desarrollado un nuevo algoritmo de teselación dependiente de la vista, explotando el hardware gráfico actual y permitiendo el mantenimiento de la coherencia entre las aproximaciones calculadas. Por último, se ha propuesto una aplicación de *sketching* para diseñar terreno, ofreciendo una alternativa a los generadores de terreno que ofrecen un control al usuario muy limitado. Estas distintas técnicas cubren la simulación de diferentes fenómenos naturales y ofrecen interesantes mejoras.

Objeto y objetivos de la investigación

Esta tesis tiene como meta manejar modelos que son altamente complejos para los que las técnicas tradicionales de modelado no son suficientes. El objetivo es desarrollar nuevos algoritmos de modelado procedural. Así, en lugar de tener que especificar los detalles de un objeto 3D, podremos utilizar unos pocos parámetros con los que un algoritmo será capaz de generar el objeto. El modelado procedural permite la generación de escenas 3D con una interacción mínima por parte del usuario. El usuario de este tipo de herramientas será capaz de generar las escenas requeridas mediante la especificación de un conjunto de parámetros, evitando tener que dedicar muchas horas e incluso días al modelado manual del escenario.

En lo últimos años el área de investigación en modelado procedural ha recibido una gran atención. Los trabajos más recientes intentan aprovechar la tecnología existente en el hardware gráfico, haciendo posible que la geometría de la escena se genere en tiempo real en la propia tarjea gráfica.

Los trabajos desarrollados hasta la fecha en este campo se han enfocado hacia la obtención de objetos de forma automática, más que en la sencillez de uso de las herramientas creadas. El objetivo de esta tesis es acercar el modelado procedural a usuarios sin un gran conocimiento en modelado de entornos, de forma que, además, la generación de la geometría se adapte al hardware gráfico disponible.

Fenómenos Naturales

El modelado de fenómenos naturales ha estado siempre entre los grandes retos de la informática gráfica. Escenas de exteriores que ofrezcan realismo deben incluir este tipo de fenómenos para ofrecer al usuario una experiencia inmersiva. El problema surge de la complejidad de estos efectos, que a menudo necesitan algoritmos muy complejos y costosos para simularlos de manera realista. En este sentido, es posible encontrar una gran cantidad de investigación en este campo, con el objetivo de simular terreno, nubes, fuego, árboles, hierba y muchos otros.

Mientras los físicos se esfuerzan en encontrar descripciones matemáticas válidas para estos fenómenos, la comunidad de informática gráfica intenta simular la belleza de esos efectos de la manera más realista posible. La dificultad de esta tarea no solo depende de la complejidad

y diversidad de objetos y fenómenos naturales interactuando, sino también de la enorme cantidad de pequeños detalles que es necesario modelar para obtener una visualización realista y correcta desde el punto de vista físico. Así, todavía existe una necesidad de herramientas de modelado interactivas y de técnicas de simulación que sean capaces de manejar la complejidad de las escenas sintéticas.

Los efectos atmosféricos varían mucho en sus propiedades físicas y en los efectos visuales que producen. Además, la simulación del clima incluye numerosos efectos visuales actuando al mismo tiempo. Basándonos en sus diferencias, los efectos atmosféricos pueden clasificarse en estáticos (niebla, neblina) o dinámicos (lluvia, nieve, granizo). La visualización realista de efectos atmosféricos mejora las escenas de exteriores y son un aspecto fundamental para crear entornos inmersivos. En concreto, el problema de la visualización de precipitación se ha intentado resolver en numerosas ocasiones, no solo para la propia visualización de la lluvia [1, 2] sino también para su interacción con otras superficies [3, 4] o incluso para la acumulación de agua sobre la superficie [5]

La lluvia es un fenómeno natural extremadamente complejo. Mientras que las partículas que causan otros efectos como la niebla o la bruma, las gotas de lluvia son suficientemente grandes como para ser visibles al ojo humano. Cada gota refracta y refleja la radiancia de la escena y la iluminación del entorno hacia el observador. La mayor parte de las soluciones usan sistemas de partículas para simular lluvia. Estos sistemas se han empleado satisfactoriamente en trabajos anteriores para simular distintos tipos de fenómenos difusos como fuego o humo [6]. Sin embargo, el problema de las soluciones actuales es su alto coste de gestión ya que es necesario procesar una gran cantidad de gotas para ofrecer un resultado realista. Una posible solución para superar las limitaciones de los métodos de lluvia anteriores es la explotación de las capacidades de las tarjetas gráficas actuales, cuya continua evolución puede incrementar el rendimiento final. Además, es interesante considerar la posibilidad de aplicar técnicas de modelado por nivel de detalle a los sistemas de partículas existentes.

Desde un punto de vista diferente, la simulación de agua se ha estudiado con detalle debido a los complejos fenómenos que tienen lugar al mismo tiempo. Así, la interacción del agua con otras superficies y sus propiedades ópticas son muy importantes en aplicaciones 3D y en la industria del cine. Dentro de los sistemas de simulación de agua, las olas del mar representan un complicado reto, ya que el mar está compuesto

por diferentes elementos que conforman un sistema de alta complejidad. Los procesos físicos inherentes a la simulación de mar se han analizado en numerosas ocasiones, de manera que se han realizado diversas propuestas que intentan simplificar estos procesos mientras se mantiene un alto grado de realismo. Así, se han desarrollado distintas técnicas para manejar las mallas que simulan la superficie del mar, aunque presentan problemas para simular grandes superficies de agua. Como posible solución, existen diversos métodos que presentan técnicas de teselación en el hardware gráfico [7, 8, 9], aunque muy pocos investigadores han considerado la utilidad del hardware gráfico para teselar la superficie del mar.

Por último, es importante considerar la simulación de terreno. El terreno en una escena es un elemento clave que puede disminuir la sensación de realismo si no se considera correctamente. Durante muchos años se ha investigado la visualización de terreno y es posible encontrar muchas soluciones para su representación realista e interactiva. Sin embargo, mucho menos esfuerzo ha sido dedicado al desarrollo de técnicas que simplifiquen la creación del terreno. Las diferentes propuestas que se han presentado para la generación de terreno ofrecen, generalmente, un control por parte del usuario muy complejo y muy reducido. Además, este control se suele basar en un conjunto de parámetros que son difíciles de manejar. En este sentido, existe una demanda de interfaces más intuitivas, ya que diseñadores y usuarios más generales encuentran frustrante la complejidad y falta de intuitividad de las herramientas de modelado actuales.

Planteamiento y metodología utilizados

Esta tesis tiene como objetivo fundamental el desarrollo de nuevas técnicas de simulación de fenómenos naturales que mejoren su manejo y visualización en motores de juegos y aplicaciones interactivas.

Para ello, se propone realizar las tareas siguientes:

Estado del arte

Como paso previo al trabajo de la tesis, es necesario desarrollar un estudio en profundidad de las técnicas consideradas en esta tesis. Asimismo, se pretende analizar los fenómenos físicos relacionados para poder ofrecer una visualización acorde a cómo suceden en la Naturaleza.

Modelado de fenómenos naturales

Como ya se ha comentado previamente, el objetivo fundamental de esta tesis es el desarrollo de nuevas técnicas que mejoren la eficiencia en la visualización de estos fenómenos. Para ello se propone el estudio de soluciones que estén totalmente integradas en el hardware gráfico existente y que tengan en cuenta todas las características de las nuevas tarjetas. Asimismo, se desea considerar el uso de librerías como CUDA que ofrecen una abstracción de la propia programación de la tarjeta gráfica, facilitando su programación, ofreciendo mayor funcionalidad y obteniendo un mayor rendimiento.

Integración en motores de juegos

La necesidad de obtener unos resultados aplicables obliga a estudiar las posibilidades de integración de los modelos desarrollados en varios motores de juegos, para poder comprobar su calidad en aplicaciones reales y mejorar los aspectos que sean necesarios.

Evaluación y comparación de los modelos desarrollados

La verificación y prueba de los resultados obtenidos se realizará sobre el software de soporte a los proyectos de investigación en los que se enmarca el trabajo de esta tesis. Por supuesto, se pretende demostrar la calidad de las soluciones propuestas frente a otras soluciones existentes anteriormente. La calidad de los resultados de esta tesis se podrá evaluar en tests de usabilidad con usuarios reales, obteniendo información sobre fallos, limitaciones y aspectos a mejorar.

Aportaciones originales

Como ya se ha comentado anteriormente, el objetivo de esta tesis es mejorar la generación y visualización de distintos fenómenos atmosféricos. De esta manera, se pretende ofrecer al usuario final herramientas para la creación y gestión de estos fenómenos dentro de las aplicaciones 3D. Además, las características del hardware más reciente se han tenido en consideración a la hora de desarrollar las distintas técnicas para poder ofrecer un alto rendimiento.

En el Capítulo 3 se propone una solución para facilitar la creación y el control de las escenas de lluvia, mejorando los métodos anteriores y

manteniendo la apariencia realista. Uno de los principales objetivos de este capítulo es desarrollar una herramienta con la que se puedan generar ambientes lluviosos automáticamente, creando zonas de lluvia donde miles de gotas son simuladas en tiempo real. Dentro de cada zona, simulamos la lluvia mediante un sistema de partículas al que añadimos un algoritmo multirresolución. Los algoritmos multirresolución se han aplicado satisfactoriamente para solucionar problemas en diversas áreas [10], existiendo una gran cantidad de investigación en este campo [11]. El uso de técnicas de nivel de detalle nos permite incrementar el rendimiento mientras reducimos la carga de las diferentes etapas de la *pipeline* gráfica. Con este objetivo, incluimos técnicas multirresolución para adaptar el número de partículas, su ubicación y su tamaño a las condiciones de la escena. Las propiedades físicas también se tienen en cuenta para ofrecer una solución satisfactoria. Este método de simulación está totalmente integrado en el hardware gráfico para ofrecer una visualización de lluvia en tiempo real.

La solución descrita en el Capítulo 4 presenta una mejora evolutiva de la técnica de simulación de lluvia anterior. Esta nueva solución también se basa en el uso de sistemas de partículas para simular lluvia, pero ofrece un sistema de detección de colisiones. Así, el sistema es capaz de detectar colisiones de gotas sobre el escenario para simular las salpicaduras. Además, la solución se ha desarrollado de manera que es capaz de reaccionar correctamente a los cambios de posición de los objetos en la escena. Este nuevo método de simulación de lluvia obtiene un alto rendimiento gracias al uso de CUDA, que permite realizar todos los cálculos de la simulación de lluvia, colisiones y salpicaduras en una sola pasada.

Respecto a la simulación de mar, el Capítulo 5 presenta una nueva técnica que utiliza un algoritmo de teselación en la tarjeta gráfica. Este algoritmo ofrece una teselación dependiente de la vista para incrementar y reducir el nivel de detalle de la malla que simula la superficie del mar. Con este método se evita la aparición de agujeros en la malla y se explota la coherencia entre aproximaciones, reutilizando la geometría extraída al incrementar y decrementar el detalle. Para mejorar la visualización, animamos las olas del mar mediante *ruido de Perlin* y consideramos la reflexión y el *fresnel* para obtener una sensación realista.

El principal objetivo del Capítulo 6 es ofrecer una herramienta para la generación de terreno que sea precisa y fácil de usar, permitiendo a usuarios no profesionales diseñar su propia isla. Nuestra meta

Figure 1: Sensación de lluvia obtenida con la solución propuesta en una escena con un *skybox* texturizado con una imagen de Chicago.

es desarrollar métodos sencillos y prácticos con los que crear modelos de terreno. Intentamos mostrar que algoritmos relativamente simples se pueden combinar para obtener resultados rápidos y satisfactorios. En este capítulo consideramos tanto el algoritmo de generación de terreno como su integración en una aplicación de *sketching*. Esta aplicación ofrece representaciones 2D y 3D del terreno, con el objetivo de simplificar el interfaz y ofrecer mayor información sobre el terreno que se está generando. También presentamos distintas opciones que permiten al usuario modificar y mejorar el terreno generado. Por último, se considera la posibilidad de integrar los terrenos generados en aplicaciones y motores de juegos reales.

Conclusiones

En el Capítulo 2 se ha descrito el trabajo previo relacionado con cada fenómeno natural que se ha considerado. Así, se ha presentado el estado de arte en visualización de lluvia, donde se puede ver que las soluciones existentes todavía presentan limitaciones para escenarios reales donde el usuario se mueve muy rápido. Las físicas relativas a la lluvia también se describieron, ya que la solución propuesta en esta tesis está basada en cómo se comportan las gotas de lluvia. Tras el estudio de la simulación de

lluvia, presentamos el trabajo relacionado con la visualización de escenas de mar y las técnicas de teselación que se aplican. En este caso pudimos ver cómo no existen demasiadas técnicas de teselación basadas en el hardware gráfico. Por último, el resto del capítulo introduce el trabajo existente sobre la generación de terreno, considerando soluciones como las técnicas procedurales de generación sintética de terreno. Este estudio nos mostró que las soluciones para la generación de terreno no ofrecen un gran control sobre el resultado obtenido, resultando en muchos casos complejas desde el punto de vista del usuario.

Tras analizar el estado del arte de los distintos fenómenos, pudimos ver que todavía existía espacio de mejora. En este sentido, el Capítulo 3 presenta un conjunto de técnicas para crear y visualizar eficientemente lluvia realista (ver Figura 1). La solución propuesta ofrece distintas posibilidades dependiendo de la relación entre la localización del usuario y de la zona de lluvia. Además, se han incluido técnicas multirresolución que se ejecutan únicamente en la tarjeta gráfica. Por otra parte, se ha presentado también un estudio de la generación de partículas directamente en el hardware. Los resultados obtenidos mejoran los conseguidos por soluciones anteriores. El estudio realizado entre usuarios ha demostrado que nuestro modelo es capaz de ofrecer sensaciones de una intensidad de lluvia similares pero con muchas menos partículas. Además, la forma que se ha elegido para el contenedor de lluvia evita que tengamos que recolocar todo el contenedor continuamente, lo que suponía unas de las principales desventajas de soluciones anteriores y que las hacía inadecuadas para videojuegos u otras aplicaciones en las que el usuario realiza movimientos de cámara muy rápidos. Ofrecemos una solución que es rápida, simple, eficiente y fácilmente integrable en entornos de realidad virtual ya existentes.

Con el objetivo de extender la técnica de simulación de lluvia presentada y para ofrecer una mayor explotación del hardware gráfico actual, el Capítulo 4 describe una técnica para detectar, procesar y simular colisiones de la lluvia sobre el escenario de una manera eficiente. Esta mejora de la simulación se ha realizado mediante el uso de CUDA, ya que su flexibilidad nos permite ofrecer técnicas avanzadas como la detección de colisiones. Así, la eficiencia obtenida en las diferentes pruebas ha sido posible gracias a la gran cantidad de partículas y salpicaduras que el sistema es capaz de visualizar y procesar. Además, CUDA permite que la aplicación gráfica disminuya el uso de CPU en un 50 %, permitiendo que la aplicación dedique ese tiempo ahorrado en otros cálculos.

Desde una perspectiva diferente, el Capítulo 5 presenta una nueva técnica de teselación que se ha aplicado a la visualización de mar en la tarjeta gráfica. La solución propuesta evita la aparición de agujeros y otros artefactos durante la animación de la superficie. Otro aspecto importante es la posibilidad de reutilizar la última teselación calculada cuando aumentamos o disminuimos el nivel de detalle. Así, minimizamos las operaciones a realizar en ambos casos, reduciendo el coste temporal relacionado con el proceso de teselación. El sistema propuesto considera el uso de *ruido de Perlin* para animar la superficie del mar y la inclusión de algunos efectos ópticos para contribuir al realismo de la escena sin incrementar excesivamente el tiempo de visualización.

Finalmente, el Capítulo 6 describe de una manera detallada una herramienta para la generación de terreno que resulta interesante para aquellos usuarios que desean tener un alto grado de control sobre el proceso de generación del terreno. Esta herramienta se ha enfocado fundamentalmente al diseño de islas, aunque podría extenderse para trabajar con otros tipos de terreno. La aplicación es muy fácil de usar y requiere una interfaz de usuario mínima, donde todas las principales operaciones se controlan a través de un ratón de dos botones. Por medio de esta aplicación el usuario puede añadir, eliminar o dar forma a montañas existentes. Además, es posible modificar la silueta de la isla y añadir pequeñas perturbaciones distribuidas aleatoriamente sobre el terreno para darle más realismo. Mediante un estudio de usabilidad que se ha desarrollado entre personas de distintas habilidades informáticas se ha podido observar que la interfaz desarrollada es cómoda y adecuada en la mayor parte de los casos.

Líneas futuras de investigación

En esta tesis se presentan distintas mejoras en la simulación de fenómenos naturales. La simulación de fenómenos ambientales recibe una gran atención por parte de los investigadores ya que su inclusión en juegos y películas de animación por ordenador mejora el realismo de las escenas obtenidas. En este sentido, estamos interesados en continuar trabajando en estas líneas para mejorar el rendimiento y el realismo de las diferentes soluciones presentadas en esta tesis.

En primer lugar, el modelo presentado para la simulación de lluvia podría mejorarse con métodos para simular la interacción de la luz con las gotas u otros efectos que se han presentado en el trabajo rela-

cionado. Además, nuestro interés se centra en la aplicación del modelado multirresolución a estos efectos, con el objetivo de obtener una simulación más realista mientras se mantiene un bajo coste computacional. También se podría aplicar el método propuesto a la simulación de nieve. Respecto a la solución basada en CUDA, planeamos la utilización de la detección de colisiones para ofrecer mayor realismo, por ejemplo, mediante el cálculo de la acumulación de lluvia sobre el terreno. Desde otro punto de vista, consideramos interesante el estudio de la alteración de la precipitación y las salpicaduras en situaciones de viento, de manera que el sistema sea capaz de resolver correctamente situaciones donde las gotas no caen de forma totalmente vertical.

En segundo lugar, y de una forma similar al caso de la lluvia, creemos que sería adecuada la mejora de la simulación de mar mediante la aplicación de más efectos, mejorando la calidad visual percibida por el usuario. Así, la simulación de agua poco profunda es interesante, ya que el agua del mar se comporta de manera diferente cuando se aproxima a la costa. En este sentido, nos gustaría considerar cómo las olas rompen, produciendo espuma y salpicaduras. Además, el estudio de métodos que ofrezcan interacción de la superficie del mar con otros objetos es también muy atractivo, ya que en un escenario real se suelen incluir objetos que caen o flotan sobre el mar.

Por último, el área de investigación más prometedora de nuestra herramienta de generación de terreno es la posibilidad de añadir más características a la aplicación existente. Por ejemplo, sería interesante cambiar la aplicación de manera que el usuario pudiera marcar una zona de la isla como playa o montaña. De una manera similar se podría permitir al artista incluir fenómenos atmosféricos, vegetación y otros elementos decorativos en el terreno. Una mejora que se podría incluir es la simulación de procesos físicos, de manera que, por ejemplo, la altura de las montañas de la isla no se determinara mediante una fórmula matemática sino por la manera que la lava fluye tras una erupción volcánica. Así, la forma resultante podría verse afectada por el viento y la lluvia.

.... to Jaime and Ana, my lovely parents.

Preface

Abstract

Realistic outdoor scenarios often include different natural phenomena which are difficult to simulate in real time. In the field of real-time applications a number of solutions have been proposed which offer realistic but costly solutions. The main aim of this dissertation is to present a set of techniques which have been developed to improve the real-time visualization of these natural phenomena. We have proposed a solution to facilitate the creation and control of rain scenes by means of a new level-of-detail scheme. Later, this rain framework was extended to include the management of collisions with the environment and the simulation of the splashes. As an improvement on ocean simulation techniques, we have also developed a new adaptive tessellation algorithm which exploits the features of current GPUs to allow coherence among extracted approximations. Lastly, a sketching interface for designing terrain is proposed as an alternative to those synthetic terrain generators which offer a limited user control. These techniques cover the simulation of different natural phenomena offering interesting improvements.

Keywords: Real-Time Rendering, GPU, Rain, Level-of-Detail, Ocean Simulation, Tessellation, Terrain Synthesis, Sketching

Funding

This work has been supported by the Spanish Ministry of Science and Technology (grants TSI-2004-02940, TIN2007-68066-C04-01 and TIN2007-68066-C04-02) and by Fundació Bancaixa (P1 1B2007-56). We would like to thank Pierre Rousseau and Sarah Tariq for their support.

IV

Acknowledgements

It is said that accomplishing a Ph.D. is a significant achievement in one's life. Although it is habitually accredited to a single person, its author, in most cases there is a number of people who have contributed one way or another to this goal. Having completed my thesis, it is time to give credit to some of the people that played a role in my Ph.D.

My directors, Miguel Chover and Mateu Sbert who combined all I needed in an advisor: support, motivation and excellent tutoring in research.

Then, I would like to thank the experts committee: Celso, Charles and specially to Benoît who was a great tutor during my stay at the University of Limoges. I thank all of them for their effort and time in reviewing this dissertation. I have used all their comments to improve the latest versions.

Later come my colleagues at the Universitat Jaume I. My office-mates: the sweet Maria and the wise Olga and also my lab-mate Zoe. I would like to thank Óscar B., Ramón, Jesús, Francis, Nico, Peter, Carlos G., Charli, Laura, Arturo, Guillermo, Magali, Óscar C., Toni, Eli, Inma, Cristina, Ricki, Joaquin, Pacual and Mike for a pleasant and friendly working environment. And for many nice discussions during lunch time to Gema, Tomás, Gaetan, Siddharth, Esther, Mota, Olga, Ernesto, Gus, Jose, Pablo, Raúl, M^{a}José, Rafa, Pedro, Isabel, Pepe, M^{a}Angeles and so on.

I would also like to thank some people that played a role in my decision to pursue a Ph.D. and also helped me to reasearch and enjoy The Netherlands: Spyros, Radu, Annette, Niels, Jochem, Maik, Marjoleijn and all the people from the Vrije Universiteit in Amsterdam.

Also the people from the Universidad Rey Juan Carlos in Madrid: Manolo, Roberto, Juan M., Cris, Isra, Luís L., Antonio Fernandez, Juan

C., Luís R., Mica, Alberto, Alex, Álvaro, Teo, Francisco, Tomás and all of them who teached me and helped me at work and showed me the grandiose city. And Vicente Cholvi, my former tutor during my first steps on this Ph.D.

At this point, I would like to thank my colleagues at Limoges for providing me with moral support and friendship for finishing this Ph.D.: Dimitri, Choo, Nadir, Celine, Cyril, Ahmadou, Julien, Laurent, Pierre, Michelle, Hubert, Vincent and everybody who teached me and explained me 'la politesse française'.

Last but not least, I would like to say a big THANK YOU to my parents who have always unconditionally supported me in all respects, I am grateful to them not only for their support during the years of my Ph.D., but also for their guidance throughout the path of my life.

Of course, I should thank my husband Oscar for being always my better half.

Anna Puig Centelles

Castellon, January 2010

Index

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Virtual environments today require a deep interactive experience with larger worlds to explore and with a higher degree of perceived realism. In this new century virtuality is becoming reality. Life in virtual worlds takes much of the time of many internet users, being their stay in the virtual world in some occasions more real than their stay in the real world. Nowadays, applications such as scientific simulations, virtual reality or computer games are increasing the detail of their environments with the aim of offering more realism. These environments are usually large and highly detailed, being therefore very difficult to be modeled by hand.

Nowadays the state of the art in the creation of virtual worlds develops out of objects created by men, such as buildings or streets and massive objects like grounds. All this kind of models are manually generated from the utilization of a 3D modeling program. The effort required is huge and through the growing power of rendering platforms the hopes of virtual world users is constantly increasing. Even so, more and more costly, expensive and manual effort has to be invested in order to obtain a creation of a more complex content.

1.1. Objectives of the Research

This thesis looks at models that are highly complex and for which traditional modeling techniques are not sufficient. Our objective is to develop new procedural modeling algorithms. Thus, instead of specifying the details of a 3D object, we will use some parameters for a procedure that will create the object. Procedural modeling allows the generation of 3D scenes with the minimum user interaction. The users of these tools will be able to generate the required scenes by specifying some simple parameters, without having to spend many hours or even days modeling objects.

In the recent years the research area studying procedural modeling has received a lot of effort and the latest works try to profit from the new existing technology in graphics hardware, making it possible to the geometry of a scene to be generated in rendering time in the graphics card itself.

The works developed to date on this aspect focus on obtaining the objects in an automatic way, rather than on the easiness of the creation tools. The aim of this thesis is to make procedural modeling possible for users without much knowledge on environment modeling and that the objects generated can be adapted to the graphic hardware available.

1.2. Natural Phenomena

Modeling natural phenomena has always been among the most challenging problems in computer graphics. Realistic outdoor scenarios must include these phenomena in order to offer an immersive experience. The complexity of these effects usually requires very complicated algorithms to simulate them realistically. In this sense, it is possible to find in the literature a great amount of research on this topic, simulating terrain, clouds, fire, trees, grass and many more.

While physicists put their efforts in finding valid mathematical descriptions of the phenomena, the computer graphics community tries to simulate the beauty of such effects as realistic as possible. The difficulty in this aim depends not only on the complexity and diversity of objects and natural phenomena interacting together, but also on the huge amount of small details that should be modeled to obtain realistic models and physically correct simulations. Therefore, there is still a need for interactive modeling and simulation techniques capable for handling

Figure 1.1: Rain simulation inside the commercial game *Heavy Rain* (expected to be released in 2010).

complex synthetic sceneries.

Atmospheric effects vary widely in their physical properties and in the visual effects they produce. Furthermore, weather simulation consists of numerous visual effects interacting together. Based on their differences, weather conditions can be broadly classified as steady (fog, mist and haze) or dynamic (rain, snow and hail). Realistic rendering of weather effects enhance outdoor scenes and they are compulsory features for creating realistic immersive environments. The problem of depicting atmospheric precipitations has been approached in numerous occasions, not only for rain visualization [1, 2], but also for its interaction with other surfaces [3, 4] or even for the accumulation of water on the ground [5]. An example of the importance of these effects is the snapshot of the *Hard Rain* videogame depicted in Figure 1.1, where an extremely detailed rainy environment is shown. This commercial game has been developed by the company *Quantic Dream* and it is expected to be released in 2010. The quality of the image shows how the inclusion of atmospheric effects enhances the final renders, although rendering these effects realistically is a hard problem, especially in real time.

Figure 1.2: Ocean simulation inside the commercial game called *Crysis* (released in 2007).

Rain is an extremely complex atmospheric natural phenomenon. Unlike the particles that cause other weather conditions such as haze or fog, rain drops are large and visible to the naked eye. Each drop refracts and reflects both scene radiance and environmental illumination towards an observer. Realistic-looking rain greatly enhances scenes of outdoor reality, with applications including computer games and motion pictures. Most solutions use a particle-system to simulate rain. These systems have been employed successfully in previous works to simulate several types of diffuse phenomena, such as smoke or fire [6]. Nevertheless, a problem of the existing solutions is their high management cost as it is necessary to process a great amount of raindrops to offer a realistic visualization. A possible solution to overcome the limitations of previous rain simulation algorithms is the exploitation of the current GPU possibilities, whose constant evolution can considerably increase the final performance. Moreover, it is worth considering the possibilities of applying multiresolution techniques to existing particle systems.

From a different point of view, water simulation has been thoroughly investigated due to the complex phenomena that take place at the same time. Thus, the interaction of water with other surfaces as well as its

Figure 1.3: Terrain simulation inside the commercial game *Rage* (projected release in 2010).

optical properties are very important in 3D applications and also for the movie industry. Within the simulation of water, ocean waves represent a very complicated challenge, as ocean is composed of different elements that form a very complex system. The physics underlying ocean simulation have been analyzed and many proposals have appeared that try to simplify the complex processes that take place but offering a high degree of realism. Figure 1.2 presents a screenshot of the game named *Crysis* developed by the company *Crytek*, where natural phenomena are thoroughly simulated, offering a very realistic look. Thus, different techniques have been proposed to manage the meshes representing ocean surfaces although they still present some limitations that prevent them from simulating large ocean scenes. In this sense, it is possible to find in the literature many solutions aimed at tessellating an initial mesh while maximizing the use of GPU features [7, 8, 9]. Nevertheless, not so many tessellation techniques which consider graphics hardware have been applied to ocean surfaces.

Finally, it is important to mention the terrain simulation. Synthetic terrain is a key element in many applications that can lessen the sense of realism if it is not addressed correctly. In Figure 1.3 a terrain of the

commercial game *Rage* presented by *Id Software* is shown, proving how terrain can be a key element in some applications. For many decades this issue has been considerer and it is possible to find in the literature many solutions to its realistic and interactive rendering. Nevertheless, not so many efforts have been dedicated to the development of techniques that can ease the creation of terrain, as usually terrain is seen as the background of the scene and its importance is often ignored. Different proposals have been presented but they do not provide enough user control and, moreover, the parameters are usually difficult to control. In this sense, there is an increasing demand for more intuitive interfaces, as many designers and more general users are often disappointed by the complexity, difficulty and unintuitive nature of current modeling interfaces.

1.3. Applied Metodology

The main aim of the presented Ph.D. work is to improve the rendering of different natural phenomena. Hence, it is our objective to offer the final user with easy tools for the creation and management of these effects inside their applications. Additionally, the latest features of graphics hardware have been considered in order to obtain a high performance while offering a great realism.

We propose in Chapter 3 a solution to facilitate the creation and control of rain scenes and to improve on previously used methods while offering a realistic appearance of rain. One of the main objectives of our model is to be able to generate rainy environments automatically by creating raining areas where thousands of drops are simulated in real time. Within each rain area, we will simulate rain with a particle system in which we will include a new level-of-detail framework in order to improve the performance. Multiresolution modeling has been successfully applied to solve problems in many areas [10] and there is an important body of literature on the subject [11]. The use of level-of-detail (LOD) techniques increases the rendering efficiency by decreasing the workload on different stages of the graphics *pipeline*. With this objective, we include multiresolution techniques in order to adapt the number of particles, their location and their size according to the view conditions. The physical properties of rain are incorporated into the final approach that we propose. The presented method is completely integrated on the GPU in order to render rain streaks in real time.

The solution suggested in Chapter 4 is an improved evolution of the rain simulation technique simulated in Chapter 3. Our approximation is also based on the utilization of particle systems in order to simulate rain, but includes a collision detection system. This system is able to detect collisions of raindrops on the scenario and to simulate the splashes accordingly. Moreover, the solution has been developed so that the system acts properly with objects that dynamically change their position within the scene. This technique obtains a very high performance through the use of CUDA, since it considerably frees the CPU from an enormous load of operations. In this proposal one pass of the graphics hardware is sufficient to make the calculations that update the positions of particles, calculates collisions and creates splashes.

We propose in Chapter 5 a new technique for modeling and rendering realistic ocean scenes. This technique consists in tessellating the ocean surface on GPU. This algorithm introduces a new adaptive tessellation scheme for increasing or decreasing the level of detail of the mesh simulating the ocean. This tessellation algorithm avoids the appearance of T-vertices that can produce artifacts in an animated mesh like the ocean surface. We exploit the geometry shader capabilities in order to reuse the already calculated data. We also simulate reflection on the generated sea and animate ocean waves by means of GPU-based Perlin noise.

The main goal of our work in Chapter 6 is to provide the final user with an easy-to-use accurate terrain generation solution, which allows nonprofessional users to design their own desired island. Our aim consists in developing convenient and simple ways to create computer models of terrain. We try to show that relatively simple algorithms can be combined to provide fast and successful results. In this chapter we introduce a simple terrain algorithm and we also consider its integration into a sketching application. The application will offer both a 2D and a 3D representation of the terrain, in order to simplify the interface and provide the user with more interactive feedback about the terrain that has been designed. Moreover, once these terrains are generated, the tool offers the possibility of adapting and adjusting the surface created to obtain the results desired by the user. We also present the possibility of importing the generated terrain generated into a game engine for its use in a 3D scenario.

1.4. Original Contributions

The Ph.D. thesis that is presented in this dissertation is organized as follows:

- Chapter 2: **Previous Work**
  We present the state-of-the-art on the different proposals presented in this dissertation. Thus, we begin by considering the work previously carried out on rain rendering and presenting a characterization of the rain models. We also analyze the physics of rain. Then, we describe GPU-based tessellation schemes applied to rendering ocean surfaces, as well as the latest tendencies in realistic ocean simulation. Lastly, we propose different sets of techniques that can be used to sketch terrain.

- Chapter 3: **Creation and Control of Rain in Virtual Environments**
  We introduce a new framework to facilitate the creation and control of rain scenes and to improve on previously used methods while offering a realistic appearance of rain. We accomplish this objective in two ways. On the one hand, we create and define the areas in which it is raining. On the other hand, we perform a suitable management of the particle systems inside them. We include multiresolution techniques in order to adapt the number of particles, their location and their size according to the view conditions. Furthermore, in this work the physical properties of rain are analyzed and the presented method is completely integrated on the GPU.

- Chapter 4: **Rain Simulation on Dynamic Scenes**
  In this chapter we extend the previous solution by including, aside from the rainfall simulation, a system for the detection and handling of the collisions of particles against the scenario, which allows for the simulation of splashes at the same time. This system obtains a very high performance thanks to the use of CUDA.

- Chapter 5: **Ocean Simulation**
  We propose a new technique for tessellating ocean surfaces on GPU. This technique is capable of offering view-dependent approximations of the mesh while maintaining coherence among the extracted approximations. This feature is very important as most

solutions previously presented must re-tessellate from the initial approximation while we can use the latest extracted one both when refining and coarsening the mesh. This implementation also considers fractal noise for wave animation and includes reflection and fresnel term to enhance the final rendering.

- Chapter 6: **Automatic Terrain Generation**
  The main goal of this chapter is to provide the final user with an easy-to-use accurate terrain generation solution, which allows nonprofessional users to design their own desired island. The application will offer both 2D and 3D representations of the terrain, in order to simplify the interface and provide the user with more interactive feedback about the terrain that has been designed.

- Chapter 7: **Conclusions and Future Work**
  Finally, this chapter summarizes the contributions and concludes the work presented in this Ph.D. dissertation. In addition this chapter offers a list of the publications related to this thesis, as well as a list of research projects that have funded the work.

CHAPTER 2

State of the Art

In recent years the development of graphics hardware and efficient rendering algorithms enabled game developers to create and render large landscapes with interactive rates. However, those scenes are still rough approximations that do not reach the complexity of real nature.

Creating a good scene requires powerful modelling algorithms on different aspects. These modeling techniques must be capable of offering a realistic and fast visualization while respecting the underlying physics which are necessary to obtain a simulation that fits the way these natural phenomena behave in Nature.

In this chapter we will review the latest work on the natural phenomena that we have considered in this Ph.D. dissertation:

- Rain simulation, focusing on those approaches oriented towards the use of GPUs capabilities. Moreover, we also present a study of rain physics that is important to be considered if we want to offer a realistic simulation.
- Ocean simulation, concentrating in those approaches which use tessellation on GPU to manage the detail of the ocean surfaces.
- Terrain generation, offering a review of the techniques that users can use to create terrain and stressing the possibilities offered by sketching.

2.1. Related Work for Rain Rendering

During the last few years several approaches to render rainfall have been developed in the field of computer graphics. Some of these methods have been performed to add rain to a single image of a scene or to a captured video with moving objects and sources, like [20] or [21].

From a different perspective, Garg et al. [22] carried out different means to capture the whole environment in a drop. Thus, due to the complex appearance of each raindrop these authors proposed the development of rain databases, so that each raindrop reflects certain values for lighting, viewpoint effects, reflection or refraction. In [23] the authors have developed an image-based rendering algorithm that uses a streak database to add rain to a single image or a captured video with moving objects. This database includes raindrops under a wide range of different viewing and lighting conditions.

Nevertheless, real-time rain has been traditionally rendered in two ways, either as a camera-centered geometry with *scrolling textures* or as a *particle system*.

Scrolling textures

The basic idea of this approach is to use a texture that covers the whole scene which is continuously scrolled following the direction of the falling rain. Dynamic textures are sequences of images of moving scenes that have also been applied for simulating sea-waves, smoke, foliage, or whirlwinds. However, they exhibit certain properties that are stationary in time [24]. To overcome this limitation, some authors [1] have developed more complex solutions which include several layers of rain for simulating rainfall at different distances from the observer.

In [25], the authors present a novel technique for realistically and efficiently rendering precipitation in scenes with moving camera positions. They map textures onto a double cone, and translate and elongate them using hardware texture transforms. This technique is fast but it does not allow interaction between rain and the environment. In fact, this method fails to create a truly convincing and interesting rain impression because of unrealistic rendering with the rain not reacting accurately to scene illumination, such as lightnings or spotlights. Furthermore, the results can appear to lack depth.

Particle systems

This method has been successfully used to simulate certain fuzzy phenomena which are difficult to visualize with conventional rendering techniques. Traditionally, this has been the approach chosen for real-time rendering of rain, even though particle systems tend to be expensive, especially if we want to render heavy rain. As a consequence, authors of previous methods have oriented their efforts toward making the use of these systems cheaper.

Some of the first authors to simulate rain with particle systems were K. Kusamoto et al. [26], who combined a particle system with a physical motion model of the rain with no GPU acceleration or light interaction. Later, Z. Feng et al. [27] introduced an efficient method to solve the problem of real-time rain simulation for 3D scenes with complex geometries by taking advantage of the parallelism and programmability of GPUs. They presented a collision detection method for raindrops in the particle system and introduced a particle subsystem of raindrops splashing after collision. When rendering scenes, the distance from the viewer was also considered by applying depth of field (DOF) effects.

The work presented in [28] proposes a technique that consists of two parts: off-line image analysis of rain videos and real-time particle-based synthesis of rain. The proposed rendering technique is purely GPU based. It consists of a hybrid approach transfers details from video of real rain to a particle-based system. It is cost-effective, yet capable of real-time realistic rain rendering. They consider rain fog, rain splatters and other effects and either simulate them by hand or by extracting the information from a video. Nevertheless, the authors acknowledge that their system is not directly applicable for games as they do not interact with the environment.

From a different perspective, the work developed by N. Tatarchuk at ATI [1] presents a very detailed rainy environment, including rainfall rendering, dripping raindrops, streaky reflections and accurate details, such as the tire marks in puddles in the street. They provide a high degree of artistic control for achieving the desired final look. They developed a hybrid system combining particle systems and scrolling textures. The particle-based subsystem is used to obtain effects for dripping raindrops and splashes. The image-space method is used to actually simulate rain, by using multiple scrolling textures of falling raindrops in a single compositing pass over the rendered scene. Therefore, to simu-

late strong rainfall, they simultaneously use the concepts of individual raindrop rendering and the principles of stochastic distribution for simulation of dynamic textures. All their techniques utilize a unified HDR illumination model to allow the rain to respond dynamically and correctly to the environment lighting and viewpoint changes as well as the atmospheric effects. The illumination system also provides integrated support for dynamic soft shadows. This hybrid method is very powerful but it requires 300 unique *Shaders* dedicated to rain alone and the powerful Radeon X1800 graphics card. Another problem is that the camera cannot change its angle, as the subsystem in charge of simulating rain is based on scrolling textures.

Pierre Rousseau et al. [12] propose a realistic real-time rain rendering method that simulates the refraction of the scene inside a raindrop. The technique they present attempts to generate accurate results without a high computational cost. In their real-time rendering method, their intention is to have the physical properties of raindrops for high quality results like in ray-tracing methods, but without increasing the computational cost. The scene is captured to a texture which is distorted according to the optical properties of raindrops. This texture is mapped onto each raindrop by means of a vertex shader. Their method also takes into account retinal persistence where almost spherical raindrops appear as streaks and interaction with light sources. Later, the authors extended their work [29] to offer collision detection and wind interaction.

More recently, the method presented by nVidia proposes a realistic rain application [30]. This sample presents a particle system approach for animating and rendering rain streaks that works entirely on the GPU, using new features of the Shader Model 4.0. Rain particles are animated over time and in each frame they are expanded into billboards to be rendered using the *Geometry Shader*. Finally, the rendering of the rain particles uses a library of textures stored in a *Texture Array*, which encodes the appearance of different raindrops under different viewpoint and lighting directions. These textures are obtained from the work presented in [23], which offers a data base of raindrop textures obtained under different light conditions and camera angles that has been presented at the beginning of this section.

Additionally, the work presented in [2] introduces a new framework which proposes a set of methods to model a raining scene following physical mechanisms. More precisely, they thoroughly address physical properties of rain, visual appearance, foggy effects, light interactions and

scattering. The main drawback of this approach is that, despite offering very realistic simulations, their method cannot render a scene with a sufficient framerate to offer interactive walkthroughs.

The main constraint of all the methods described above is that they are not suitable for scenarios where the user moves quickly, since adjusting the particle system to the new camera condition is too expensive.

2.1.1. Characterization of Rain Models

Table 2.1 shows a summary of a comparison of the methods explained in this review. The description takes into account several aspects:

- Authors: this indicates the author, the company or University and the year.
- Techniques: this says whether the technique for rendering the rain uses a particle system or scrolling textures.
- Collision: this indicates whether they consider that particles collide with one another or with some other surface.
- GPU: this indicates whether the model analyzed uses any of the shader capacities of current GPUs.
- Moving camera: whether the camera position or the camera angle can be changed by the user.
- Reflection: whether the rain considers the reflection of light.
- Refraction: whether the rain takes into account the light refraction.
- Fog / Motion blur: this indicates whether they include fog or the motion blur that occurs when a moving object streaks rapidly.
- Light: whether light interaction is considered in the scene.
- Wind: whether the wind is considered when rendering the direction of the rain.
- Others: this includes different features that can be considered in the different models, such as snow, splashes, which are like collisions with a surface but with the proper water effect; the depth of field (DOF) or the distance in front of and beyond the object; ripples or surface waves on the water; water dripping from different

Authors	Techniques	Collision	GPU	Moving camera	Reflection	Refraction	Fog / Motion blur	Light	Wind	Others
K.Kusamoto et al. Yamaguchi University 2001 [26]	Particle system	Yes	-	-	-	-	-	-	-	-
N.Wang et B.Wade Microsoft Corporation 2004 [25]	Scrolling textures	No	Vertex	Yes	No	No	Blurriness	No	No	Snow
Z.Feng et al. Zhejiang University 2006 [27]	Particle system	Yes	Vertex / Pixel	Yes	No	No	Motion Blur	Yes	Yes	Splashes, DOF
L.Wang et al. Microsoft Corporation 2006 [28]	Particle system	No	Vertex / Pixel	Yes	Yes	Yes	Fog	Yes	Yes	Splashes, Ripples
N.Tatarchuk ATI Corporation 2006 [1]	Scrolling textures / Particle system	Yes	Vertex / Pixel	No	Yes	Yes	Fog	Yes	Yes	Splashes, Ripples, Dripping, Puddles
P.Rousseau et al. Limoges University 2006 [12]	Particle system	No	Vertex / Pixel	Yes	No	Yes	No	Yes	No	Snow
S.Tariq NVIDIA Corporation 2007 [30]	Particle system	No	Vertex / Pixel Geometry	Yes	Yes	Yes	Fog	Yes	Yes	Splashes, Texture Arrays
P.Rousseau et al. Limoges University 2008 [29]	Particle system	Yes	Vertex / Pixel	Yes	No	Yes	No	Yes	Yes	Snow
C.Wang Normal University 2008 [2]	Particle system	No	Vertex / Pixel	Yes	Yes	Yes	Fog	Yes	No	Scattering, Wet ground, Rainbow

Table 2.1: Characterization of rain models.

objects; puddles or small accumulations of water on surfaces; and texture arrays used by the geometry shaders.

2.1.2. Physics of rain

Atmospheric sciences have studied the phenomenon of rain in depth for many decades [31, 32]. Many of these studies analyze the appearance of raindrops, describing their physical properties such as their shape, size and velocity distributions. It is also possible to find meteorological research that examines the visual properties of rainfall density, size, speed of the raindrops and the direction of motion [33, 34].

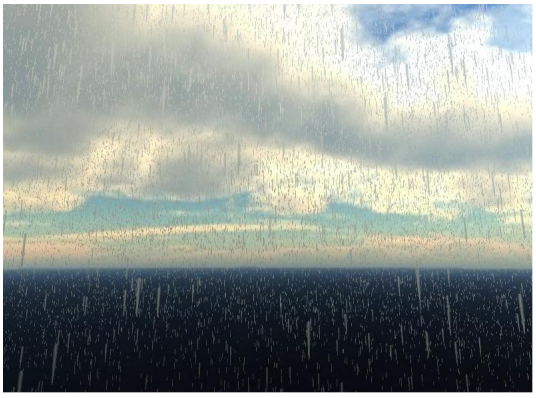
Before addressing real-time rain simulation in following chapters, it is important to analyze the features of real rain and the way rain behaves in Nature.

Rain types

Rain occurs when moist air rises and cools. The cooling causes the moisture to condense and fall as rain. There are three different mechanisms that cause air to rise and each one gives a different type of rain with its own distinct type of cloud and properties [35]. These three types of rain are:

- *Convection Rain*: this happens when the ground is warm and there is a low pressure. It tends to occur in the later part of the day or early evening. The Sun warms the ground during the day. By afternoon the moist is being forced to rise, producing very heavy rain that arrives rapidly and does not last long.
- *Relief or Orographic Rain*: this occurs when the air moves along the ground or over the sea, passes hills or mountains. These force the air to rise in order to pass the obstruction. The rising air cools and, as a consequence, clouds are formed and rain appears. Relief rain is characterized by thick clouds with drizzly conditions.
- *Frontal or Cyclonic Rain*: this appears if a warm air mass meets a cold air mass, the two air masses will generally not mix. The warm air, being less dense, will gently slide over the cold air. As it rises, it will cool and condense into clouds and rain. Rain begins slowly and remains steady for several hours if not days.

(a) Tear-shaped rain.



(b) Streak-shaped rain.

Figure 2.1: Rain appearance from the method of Rousseau et al. [12]. Comparison between the original spherical raindrops and the streaks perceived by a human eye or a camera.

Are raindrops shaped as tears?

It is worth mentioning that it is commonly believed that a raindrop is shaped as a tear. This idea is not right due to our real perception, the one that we obtain when we stare at the rain. Physically, raindrops are either spherical or elliptical, depending on their size. Nevertheless, our eye, as it happens with a camera, perceives the raindrops as streaks. In both cases, when observing the rain a drop keeps falling during a small time lapse that is not perceived neither by the retina nor the shutter.

That effect is called *retinal persistence* when talking about the human eye or *motion blur* regarding a camera. This effect has been simulated in Figure 2.1. In Figure 2.1(a) we represent the original spherical drops shape that should be perceived in a snapshot taken with a camera using an extremely fast shutter. On Figure 2.1(b) we applied the *motion blur* concept when rendering the rain. These elongated raindrops reflect more accurately what we actually perceive when looking at a rainy scene.

Size and shape of raindrops

Above all, we can say that these different kinds of rainfall will contain raindrops of different sizes. Small drops generally outnumber large drops, but as the intensity of the rainfall increases, the number of larger drops grows.

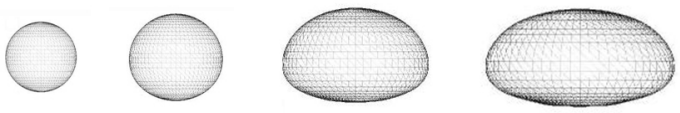


Figure 2.2: Shape of a drop [13]. From left to right droplets with radius 0.5 mm, 1.0 mm, 3.0 mm and 4.5 mm.

It is important to note that small raindrops are almost spherical while the bigger ones get flattened at the bottom, thus taking on an ellipsoidal shape [13]. This shape is due to the equilibrium between the surface tension and the aerodynamic pressure. The former attempts to minimize the contact surface between air and raindrop and the latter stretches the drop horizontally (see Figure 2.2). Small drops up to 0.5 mm are spherical and the bigger ones become ellipsoidal. It was found that raindrop size follows an exponential two-parametric distribution [36].

Speed of raindrops

It is also important to analyze the specific velocity associated with drops of every shape. The speed at which a raindrop falls depends on its radius and the terminal velocity depends on gravity and friction [13]. As it is depicted in Table 2.2, at ground level measurements have been shown that a spherical drop with a radius from 0.1 mm to 0.45 mm

Spherical Drops		Ellipsoidal Drops			
radius (mm)	speed (m/s)	radius (mm)	speed (m/s)	radius (mm)	speed (m/s)
0.1	0.72	0.5	4.0	2.5	9.2
0.15	1.17	0.75	5.43	2.75	9.23
0.2	1.62	1.0	6.59	3.0	9.23
0.25	2.06	1.25	7.46	3.25	9.23
0.3	2.47	1.5	8.1	3.5	9.23
0.35	2.87	1.75	8.58	3.75	9.23
0.4	3.27	2.0	8.91	4.0	9.23
0.45	3.67	2.25	9.11		

Table 2.2: Speed of raindrops depending on their radii [13].

has a terminal velocity of between 0.72 m/s and 3.67 m/s. Otherwise, an ellipsoidal drop with a radius from 0.5 mm to 4.0 mm, on the other hand, has a terminal velocity between 4.0 m/s and 9.23 m/s.

Visual properties of raindrops

Rain drops that are large enough, with a radius from 1mm to 10mm, can be individually detected by the camera and their motions produce randomly varying spatial and temporal intensities in an image. A single rain drop can be viewed as an optical lens that refracts and reflects light from the environment towards an observer.

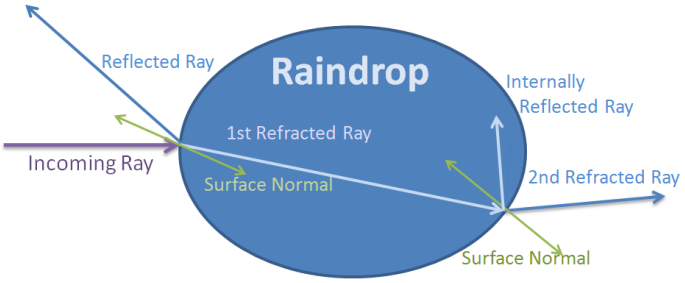


Figure 2.3: Reflection and refraction of an incoming ray in a raindrop.

Light could be defined as a set of monochromatic rays that can refract and reflect at interfaces among different propagation media. Figure 2.3 depicts the optical properties that affect a raindrop. In order to know the direction of the reflected ray we apply *Reflection's law* and to know the

Figure 2.4: Refraction of the environment and reflection of light in a drop [14].

direction of the refracted ray we can get it from *Snell's law*. Fresnel factor can give us the ratio between reflection and refraction (see Section 2.2.4). In [37] there is a model for the equilibrium shape of raindrops. It has been determined from Laplace's equation using an internal hydrostatic pressure with an external aerodynamic pressure based on measurements for a sphere but adjusted for the effect of distortion.

Finally, the appearance of a drop refracts the inverted view of the environment. The image perceived through a water-drop is a rotated and distorted wide angle image of the background scene, as it is depicted in Figure 2.4. It is possible to observe the refraction of the environment over the surface of the whole raindrop and even how the reflection is sited on the edge of the raindrop.

2.2. Related Work for Ocean Rendering

In this state of the art we will firstly present the techniques that have been developed to offer a realistic visualization of the mesh simulating the ocean, including wave animation and effects like foam, reflection, objects interactions and so on. We do not review here papers dedicated

to running water or rivers such as [38, 39], or the interaction of objects with ocean surfaces [40]. Nevertheless, a more general state of the art report can be found in [41].

Later, we will describe the tessellation techniques that have been developed for rendering ocean scenes. Lastly, we will briefly describe different physical effects that researchers consider when simulating realistic ocean surfaces.

2.2.1. Ocean Simulation

Common ocean waves, regardless of being big or small, are formed due to:

- the interaction between wind and water.
- the interaction between water and land.
- the tide, mostly due to the effect of the Moon's gravity on the Earth.

Moreover, waves can interact and propagate far from their original locations, resulting in a very complex system. In this section we present a taxonomy of ocean simulation frameworks by following the type of animation of the ocean, as this is a key aspect for offering a realistic visualization. Following this classification we can distinguish between five sets of models for modeling ocean surfaces:

Based on parametrical models

Parametric approaches [42, 43, 44, 45] represent the ocean surface as a sum of periodic functions which describe waves as a motion of particles.

The physicist Gerstner presented a first theory in 1802 to approximate the solution to fluid dynamics by describing the surface in terms of the motion of individual points on the surface [46]. Gerstner showed that the motion of each water particle is a circle of radius r around a fixed point, giving a wave profile that can be described by a mathematical function called *trochoid*. The mathematical concept trochoid was a word created by Gilles de Roberval for the curve described by a fixed point as a circle rolls along a straight line (see Figure 2.5). The use of trochoids with different radius allows us to obtain more or less sharpened and crested waves.

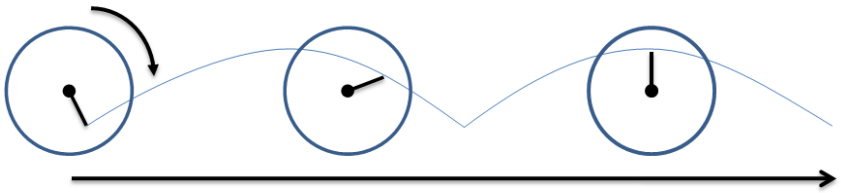


Figure 2.5: Waves generated with a trochoid.

One of the firsts descriptions of water waves in computer graphics was presented by Fournier and Reeves [42], who modeled waves by using a water surface model based on Gerstner waves. In the same year, Peachey proposed the generation of the height field by computing the superposition of several long-crested waveforms [43]. This author used particle systems to model the foam produced by wave breaking or colliding with obstacles. Later, [47] improved the wave simulation offered by the work of Fournier and Reeves. Tso et al. [44] proposed a more precise way to solve the propagation (wave tracing) by approximating the resulting ocean surface with a Beta-spline surface, which the authors claimed to offer advantages over a polygonal representation.

To sum up, we could say that all these approaches are very efficient although the scene is not very realistic. We must note that noise is generally used in all the previous models in order to avoid the visual regularity due to the fact that only one or two *wave trains* are simulated. A wave train is a succession of waves arising from the same source, having the same characteristics and propagating along the same path.

Based on physical models

Physical simulation is costly and that is the reason why many authors resort to other techniques which are faster and they concentrate on giving a final realistic look to the simulated ocean. Nevertheless, it is our interest to address some classical physical solutions.

The Navier-Stokes equations offer a set of partial derivative equations which describe fluid movements. Kass [38] used simplified numerical methods to solve the Navier-Stokes equation for animation of water waves. Stam [48] adopted Fast Fourier Transform (FFT) to simulate

the waves. Foster [39] proposed a modified semi-Lagrangian equation to simulate viscous liquids around objects. This later approach was applied to simulate the "mud shower" in the animation film *Shrek*. More recently, Thurey et al. [49] proposed a simiplification of the Navier-Stokes equations to offer real-time simulation of shallow water under some restrictions.

We could conclude that, all these approaches have a good quality of waves, but the implementation of these theories is usually difficult and simulating a large scene entails a long computation time. In the specific case of the aforementioned film *Shrek*, the method presented in [39] required 3 minutes to render each frame of the animation. Nevertheless, it is important to mention that these methods are mostly aimed at simulating breaking waves and also interactions with objects floating on the ocean surface.

Based on spectral models

This family of approaches, also known as statistical methods, is based on oceanographic measures, synthesized by spectral analysis. Spectral analysis assumes that the sea state can be considered as a combination or superposition of a large number of regular sinusoidal wave components with different frequencies, heights, and directions. As an example of these oceanographic measures, in 1964, Pierson and Moskowitz [50] developed a model for the spectrum of fully developed wind seas. This model was based on 460 ship-recorded wave records selected from a data-base of over 1000 records collected over five years.

Mathematically, spectral analysis is based on the Fourier Transform of the sea surface. Hence, these methods represent the ocean surface as a height field computed from a sum of sinusoids of various amplitudes and phases; smallscale waves and ripples are modeled directly by adding noise perturbation [51, 52].

This set of techniques has been widely used by researchers. Spectral solutions were firstly introduced by Mastin et al. [53], who proposed a realistic spectral solution by using wave synthesis based on empirical observations of oceans with the Pierson-Moskowitz filter [50]. The basic idea is to produce a height field having the same spectrum as the ocean surface. This approach computes the wave distribution by a FFT. The main benefits of this approach are that many different waves are simultaneously simulated, with visually pleasing results.

Premoze [54] combined the physical models and oceanography models, but the obtained solution is only adequate for calm sea. They include the apparition of foam on wave crests, which locally modifies the optical properties of the ocean. The authors take also into account the turbidity of the ocean, which can be understood as the opacity of the surface depending on particles suspended in the water. This parameter allows us to determine the amount of light scattered towards the surface, and provides realistic simulations of several types of water.

Tessendorf [52] shows that dispersive propagation can be managed and that the resulting field can be modified to yield trochoid waves. A positive property of FFTs is its cyclicity, as it can be applied as a tile which allows us to enlarge the simulation surface as long as the repetitiveness is not obvious. The problem of FFTs is homogeneity: no local property can exist, so no refraction can be handled. The realistic images offered in this paper were possible with the help of Renderman. This method was the one included to simulate the ocean in the *Titanic* movie.

More recently, Mitchell from ATI [55] introduced a Fourier-based GPU-synthesized height and normal maps. From a different perspective, Gonzato et al. [40] proposed a semiautomatic method to reconstruct the surface of the ocean from a video containing a real ocean scene.

Summarizing, these approaches ensure high realism, but they are not easily controllable. Moreover, since the mathematic model and computation are very complex, these methods are more adequate for animation than for real-time rendering.

Based on time-varying fractals

Fractals can be an adequate solution for simulating open sea, although they would not be capable of simulating how waves break on the seashore. A very general procedural technique for the simulation of water surfaces by means of stochastic fractals was proposed in [56].

The versatility of Ken Perlin's noise algorithm has been commonly used in real-time and offline graphics applications, starting from its first use in the film *Tron*. Perlin [57] used a noise synthesis approach to simulate the appearance of the ocean surface seen from distance. It could be considered as a particular kind of stochastic fractal that is generated as an addition of several appropriately scaled and dilated copies of a continuous noise function.

Johanson [58] adopted this approach to simulate ocean waves, but only a small area of the sea surface. In paper [59] the authors show that vertex shaders can be exploited to interactively generate non-stationary stochastic fractals, which are used to simulate the dynamics of water. Later, in Yang et al. [60] the authors use Perlin noise to generate the height field map of an unbounded ocean surface.

Although it has been shown that this particular kind of simulation process is only well suited for a very limited kind of wave phenomena, its ease and efficiency in implementation and the possibility to use this process to simulate other phenomena make it a very appealing alternative.

Hybrid approaches

Most models that attempt to accurately represent the ocean surface are usually based on parametric or spectral approaches. To overcome the problems of each of these families of solutions, hybrid procedural models were proposed.

Thon et al. [61] use a hybrid approach where the spectrum synthesized using a spectral approach is used to control the trochoids. This is only applicable in the calm sea case, where trochoids of small amplitude are very similar to sines. Smaller scale waves are obtained by directly tuning some extra Perlin noise.

Frechot [62] presents a new hybrid approach where the effort was focused on wave animation and not in other effects like fresnel reflectivity or foam. The authors use classical oceanographic parametric wave spectra to fit real world measurements, applying Gerstner parametric equations and Fourier transform.

Improving on the work of Thon et al. [61] and inside the same research group, Darles et al. [63] integrate a wave model defined as an amount of trochoids waves into a unique data structure. This data structure allows them to consider spatial and temporal coherence as well as reducing aliasing effects. To increase the realism of the generated scenes, they also propose new formulations to integrate physically-based phenomena such as second order scattering, foam and sprays, without significantly reducing performance. Their method was developed as a plug-in for Autodesk Maya 6.5.

2.2.2. Tessellation techniques

It is possible to find in the literature many solutions aimed at tessellating an initial mesh, but in this section we will concentrate in those techniques which exploit GPU features. Following this idea, it is possible to find solutions which propose sending to the GPU a mesh at minimum level of detail and applying later a refining pattern to every face of the model [7, 64, 8, 65, 9]. Programmable graphics hardware has allowed many surface tessellation approaches to migrate to the GPU, including isosurface extraction [66], subdivison surfaces [67], NURBS patches [68], and procedural detail [69, 70].

Tessellation techniques for ocean rendering

There have been several recent attempts to generate real-time water surfaces on graphics hardware.

Schneider and Westermann [59] entirely perform visual simulation on the GPU at interactive rates. They use OpenGL evaluators and NURB surfaces to tessellate the geometry on GPU. Moreover, they also use vertex shaders to generate the noise function that animates water simulation. They also provide techniques for simulating refraction, reflection and other optical characteristics.

Presenting a simple LOD management, the work described in [71] offers a solution where the wave geometry is represented as a dynamic displacement map for close areas (near patch) and a dynamic bump map for farther areas (far patch). The nearest patch can change its resolution according to the height of the viewpoint while the far patch is pre-calculated and re-located during simulation. They use the spectral method of Tessendorf [52] to animate the ocean surface and their work is based on the use of vertex and pixel shaders.

Recently, adaptive schemes have successfully been used for efficient modeling, rendering or animation of complex objects [72, 73, 74]. The idea is to minimize the sampling of the geometry according to criteria such as the distance from the viewpoint. Since the adaptive sampling is done on the fly for each frame, this fits well with procedural surface displacement, which can easily be animated.

Hinsinger et al. [75] rely on an adaptive sampling of the ocean surface, dictated by the camera position. Moreover, their animation model is also adaptive, since they filter the waves that cannot be observed from the current viewpoint. The tessellation and waveform superposition is

performed on the CPU and uploaded to the GPU each frame, which is the bottleneck of their approach.

Johanson [58] presented the projected grid concept, which offers an alternative way to render displaced surfaces which can be very efficient. The idea is to create a grid whose vertices are even-spaced in post-perspective camera space. This representation provides spatial scalability and the possibility of developing a fully-GPU implementation is described.

In paper [60], the authors offer adaptive GPU-based ocean surface tessellation by using a previous adaptive scheme for terrain rendering. Their tessellation scheme avoids the loading of vertex attributes from CPU to GPU at each frame. Their main limitation is the fact that their tessellation scheme uses a restricted quad-tree where two neighbouring areas with different resolutions can only vary to a limited extent.

Also in [76], authors presented an ocean simulation which is adaptively tessellated and driven by both per-vertex waves and per-pixel waves, using the Gerstner wave model for animating the ocean due to its simplicity and non-periodicity. The tessellation occurs in eye space, mapping a regular grid to the intersection of the ocean plane and the camera viewport. This allows them to only simulate and render geometry that is seen and tessellates more finely in the foreground than in the background.

Chiu et al. [77] offered an adaptive GPU-based ocean surface tessellation, where the refinement took place in screenspace. Moreover, they also provided optical effects for shallow water, and spray dynamics by means of particle systems.

2.2.3. Characterization of Ocean Models

Table 2.3 presents a summary of a comparison of the most recent methods from those that have been presented in this section. The columns included in this table refer to:

- Authors: this indicates the author, the company or university and the year.
- GPU usage: this indicates whether the model analyzed uses any of the shader capacities of current GPUs.
- Tessellation: whether the technique for ocean rendering tessellates the ocean surface in real time.

Authors	GPU Usage	Tessellation	Animation Technique	Reflection	Refraction	Others
J. Tessendorf. 1999 [52]	Vertex, Pixel	No	Spectral	Yes	Yes	Fresnel, Caustics, Godrays
J. Schneider et al. Aachen University of Technology 2001 [59]	Vertex	Yes	Fractals (Perlin)	Yes	Yes	Fresnel
D. Hinsinger et al. iMAGIS-GRAVIR Research Project 2002 [75]	Vertex buffers	Yes	Parametrical (Gerstner)	Yes	No	Avoid unnecessary animation, fresnel
C. Johanson Lund University 2004 [58]	Vertex Pixel	Yes	Fractals (Perlin)	Yes	Yes	Fresnel, sunlight
J. L. Mitchell ATI Research 2005 [55]	Vertex, Pixel	No	Spectral (FFT)	Yes	No	Interactions with objects, wedge
J. Demers NVIDIA Corporation 2005 [76]	Vertex, Pixel	Yes	Parametrical (Gerstner)	Yes	No	Fresnel foam, physics Spray
X. Yang et al. University of Defense Technology 2005 [60]	Vertex, Pixel	Yes	Fractal (Perlin)	No	No	Shallow water
Y.-F. Chiu et al. National Tsing Hua University 2006 [77]	Vertex, Pixel		Spectral (Tessendorf)	Yes	Yes	Fresnel, spray dynamics, depth-dependent water color
J. Fréchot LaBRI France 2006 [62]		No	Hybrid	No	No	
Y. Hu et al. Microsoft Research Asia 2006 [71]	Vertex, pixel	Yes	Spectral (Tessendorf)	Yes	Yes	Fresnel
E. Darles et al. Limoges University 2007 [63]	-	Yes	Hybrid (Tessendorf)	Yes	Yes	Fresnel glare foam, spray
N. Thurey et al. ETH Zurich 2007 [49]	-	No	Physical (Navier-Stokes)	Yes	No	Shallow water, interactions with objects, foam, spray

Table 2.3: Characterization of ocean models.

- Animation technique: this indicates which type of animation technique is selected.
- Reflection: whether the ocean considers the reflection of light.
- Refraction: whether the ocean takes into account the refraction.
- Others: this includes different features that can be considered in the different models, such as the fresnel factor or spray dynamics (see Section 2.2.4).

2.2.4. Physics of Ocean

In many of the ocean simulation techniques proposed above different sets of physical aspects were considered. Adding these effects to the final simulations increases the realism of the obtained scenes and enhances the simulation. In this section we will briefly address some of the main physical aspects that have been considered in the aforementioned oceans simulation frameworks.

Optical effects

In the oceanographic literature, ocean optics became an intensive topic of research since the 1940s. Unfortunately for oceanographers, the opacity of sea water makes the job of collecting optical images in the ocean a difficult task. In this sense, Jaffe et al. [78] provide a brief history of underwater optical imaging and the techniques and systems applied to retrieve information.

Among the optical properties that can be considered when simulating an ocean surface, reflection and refraction are key elements (see Section 2.1.2). In addition, the *Fresnel* reflectance is a physically based factor that is also very important to consider, above all when simulating materials such as plastics, glass or water.

The Fresnel Term offers a reflection-refraction ratio. Augustin-Jean Fresnel worked out the laws of optics in the early 19th century. His equations give the degree of reflectance and transmittance at the border of two media with different density. In this sense, the Fresnel term refers to the increase of the reflective property of a surface when we look at it with a grazing angle [79]. As a consequence, the reflectance we can notice in this kind of surfaces depends on the viewing angle.

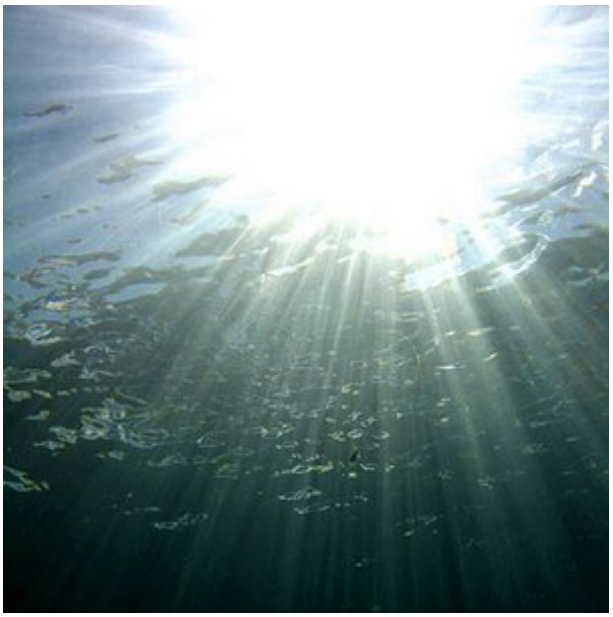


Figure 2.6: Godrays in an underwater scenario (image courtesy of John Langford [15]).

To finish with the main optical effects involved, it is worth mentioning the *caustics*. Caustics are due to the transport of light from a specular to a diffuse surface. In the specific case of ocean simulation, some authors consider the patterns created when moving water focuses refracted light, especially in underwater scenes.

Other effects

It is possible to find many other enhancements that researchers add to their approaches so that the simulation is more realistic.

Godrays are simulated in underwater scenarios and offer very impressive results. These rays of light are visible when looking from underwater towards a light source. This effect is due to the existence of small particles floating in the water, which can get between the observer and the light source producing volumes of shadow (see Figure 2.6). As a consequence, the patterns created are mostly due to the existence of rays of *darkness* instead of rays of *light*.

Figure 2.7: Whitecaps in an open ocean (image courtesy of Jeff Johnson [16]).

From a different perspective, *whitecap* is an effect considered in many simulation techniques. Whitecaps appear in open sea when the wind blows the tops off the smaller waves, producing some noticeable bubbles on sufficiently windy days (see Figure 2.7). *Foam*, although visually similar, appears when the waves break. Lastly, it is important to mention the *spray*, which usually is applied to refer to the small particles of water that can be blown from the sea.

Lastly, some authors consider the interaction of objects within the ocean surface. In the specific case of objects moving on the ocean surface, we must consider the *Kelvin wedge*. Moving ships on open water create specific patterns of waves that were first studied by Lord Kelvin. Two kinds of waves can be noted inside the wedge: divergent waves, with V-shaped fronts that move away from the shipś path, and transverse waves that tend to follow the ship [80]. Figure 2.8 depicts the V-shaped waves created by a boat.

2.3. Related Work for Terrain Sketching

In this section we analyze and characterize the different approaches that currently exist for terrain generation. After that we will consider the different software tools which are available for creating artificial terrain. Finally, we will give some basic ideas on sketching and its application to

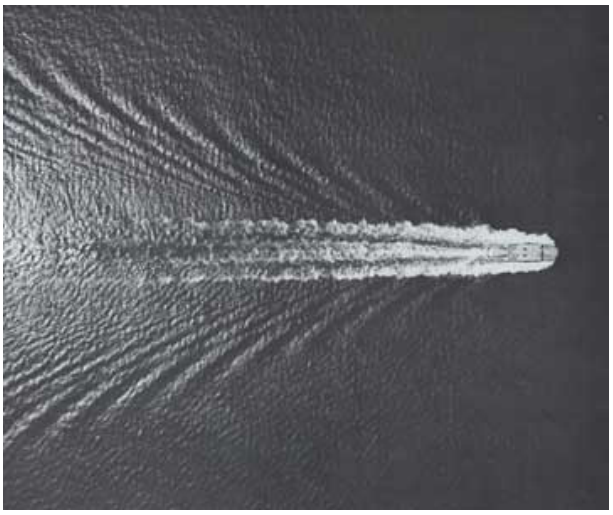


Figure 2.8: *Kelvin wedge*: V-shaped waves follwing a ship in movement (image captured from Google Earth).

our purposes.

2.3.1. Terrain Generation

The literature offers a wealth of research on synthetic terrain generation. When generating artificial terrain, the techniques can be grouped into three different categories:

Procedural Approaches

This category gathers those methods in which the terrain is generated automatically. These methods can be further separated into fractal techniques and physically-based techniques.

- **Fractal Landscape Terrain Generation**. The most popular procedural approach is fractal-based terrain generation [81], which is efficient but difficult for users to control. It is possible to find a review of recent fractal approaches in book [82] or in paper [83] .
- **Physical Erosion Simulation**. Physically-based techniques simulate the effects of physical processes such as erosion by streams

[84], water [85] or wind [86]. A recent technique that combines a non-expensive fluid simulation with an erosion algorithm is presented in [87]. It also supports effects like dissolving, transportation and sedimentation of material in the process of erosion.

Fractal landscape terrain generation and *physical erosion simulation* are both approaches that add terrain details through procedural refinement. Nevertheless, modifying their parameters to obtain a desired terrain may be a painstaking task.

Another proposal appears in [88], where the authors provide an alternative method for terrain generation that employs a two-pass *genetic algorithm approach* to produce a variety of terrain types using only intuitive user inputs. The process is efficient but very difficult for a user to control. In order to improve on genetic solutions, Frade et al. [89] developed a Genetic Terrain Programming approach which allowed users to *evolve* their terrains under some aesthetic concept or desired feature. Their presented the possibility of generating a family of terrains that meets the users criteria, although the resulting terrain was difficult to adjust.

Real Terrain Information

This approach groups the techniques from the Geographic Information Systems (GIS), where elevation data come from real-world measurements [90]. GIS data can be acquired from a number of sources [91], [92] and in different formats, such as the U.S. Geological Survey's Digital Elevation Model (DEM) format [93]. Some authors use contour lines of terrain to reconstruct the surfaces by interpolating the values between the different lines [94]. Similarly, another possible source of information could be the study of the extraction of terrain from photographs [95].

All these approaches have the advantage of offering highly realistic terrains in very little time, but with little user control.

User Defined Approaches

This is the most flexible type of techniques, in which a human artist creates the terrain manually, using an image editing program, 3D modeling software, specialized terrain editor programs or the editors that are included in game engines.

For simplifying the modeling task, a solution that has been proposed by several authors is the use of an image to define the features of the desired terrain. The authors of [96] allow the user to control the terrain generation process by using easily understood and predictable parameters. As input they receive an image generated by an image editing program in order to perform their own terrain generation process.

2.3.2. Terrain Software

Among the user-defined approaches, specific terrain generation software has received a great attention from the modeling community. In this section we introduce some terrain tools for simulating artificial environments. In these applications the user sets parameters and the program creates a pseudo-random landscape which meets those parameters. In all of these programs, terrain is modeled and imported/exported as a heightmap.

Among the existing software for terrain synthesis we highlight:

- CityScape [97] which is more oriented towards the development of complete cities but includes interesting algorithms to offer smooth deformations of the landscape in real time.

- Grome [98] which, like Terragen, uses a procedural creation of geometry. Like other applications, they offer the possibility of adding objects to the scene, to offer a complete scene creation. This software also includes natural erosion, procedural textures and the possibility of working with layers to simplify the user experience.

- L3DT [99] is another software application that generates artificial heightfields and exports their data to multiple formats including the terrain format used by TGEA [100, 101].

- Mojoworld [102] offers a very detailed application where the user is given the possibility of generating a complete planet, and presents tools to apply transformations for the whole planet or just a small portion of it. This company has thoroughly considered physics to offer realistic processes.

- Terragen [103], which has evolved from a terrain generator to an application with complex atmospheric effects, HDR lightning and

Figure 2.9: Scene created with Terragen by Hannes Janetzko simulating the Amazon River.

includes the possibility of importing your own objects. These features enable the user to obtain complex environments which can also be animated in time.

- Terraineer [104] offers the possibility of experimenting with different height generation algorithms.

- Worldbuilder [105], which includes the possibility of adding vegetation, different water effects and complex materials, offering very realistic final renders. Moreover, the authors provide the user with an easy-to-use elevation editor.

- World Machine [106] is based on fractals and a complex graph system to organize the different elements that give form to the terrain. In this sense, this software includes the modeling of physical weathering processes like wind, water and other natural processes

All of these systems have evolved in the latest years and are capable of offering very realistic terrains. This quality can be noticeable in environments like the one depicted in Figure 2.9, obtained with the commercial software Terragen. Nonetheless, this increase in visual quality also involves an increase in the complexity of the application.

2.3.3. Sketching

Finally, we would like to present *sketching* as a very promising tool for terrain generation. The sketching applications can offer a great amount of user control for the terrain synthesis process.

Sketching is commonly understood as the process of rapidly executing freehand drawing where the obtained sketches are not considered to be finished work. In the specific case of computer-aided modeling, sketching on a piece of paper is often used in the prototyping stage, before a experienced 3D modeler converts this ideas into a 3D model by means of specific software like 3D Studio Max [107] or Maya [108]. This kind of software follows usually the WIMP paradigm (Window, Icon, Menu, Pointer), which is the approach followed by most software.

Sketch-Based Interfaces for Modelling (SBIM) appeared as an evolution of the traditional WIMP paradigm. These interfaces combine the quick and intuitive feel of paper with the advantages of working directly in a computer. Obviously, it is necessary to develop adequate user interfaces and processing algorithms to capture the input data and obtain the 3D data. It is possible to find recent surveys on sketching [109, 110], which cover many concepts and techniques related to this research area.

We could distinguish between two approaches in sketch-based modeling. On the one hand we can find those solutions where the users draw the sketches by means of a mouse, a pen or any other input device. Afterwards, the system will interpret these drawings and output the corresponding 3D objects. In this sense, research has produced prototype tools for interpreting sketches of abstract polyhedra [111, 112, 113]. On the other hand, it is possible to find solutions which propose sketching several drawings on a paper and scan them so that the software can compute the 3D models [114]. Similarly, it is possible to fins proposals for extracting models of buildings from photographs [115]. Note that the converse problem (converting a photograph or even a CAD model into a freehand sketch) has also been studied, as a way of teaching the skill of field sketching to students of geography, biology and geology [116].

Mathematically, terrain is a freeform surface, so methods for creating freeform surfaces are clearly of interest here. There have been some recent developments on the automated interpretation of freeform surfaces from sketches, but they either interpret the drawing as being that of a single solid object [117, 118, 119, 120, 121] or leave the freeform surface floating in mid-air as a patch, without continuing to the horizon in the

manner of a landscape, this is the case of [122, 123].

However, interpreting a terrain sketch as a floating free-form patch presents some problems. Qin et al [122] require the user to draw a grid of quadrilaterals to represent the surface, and a trained neural network is also needed to interpret it. In Kaplan and Cohen's approach [123] the boundary of the patch must be clear in the original input, either by virtue of its obvious contrast with the background or by being specified by the user. Their approach also requires user intervention to resolve ambiguities.

More recently, a system for designing freeform surfaces with a collection of 3D curves has been proposed in [124]. They are able to create objects within a fairly easy-to-use interface, which is based on drawing simple lines. By using a similar simple sketching interface, the work presented in [125] introduces an over-sketching application for feature-preserving surface mesh editing. This application allows simple yet realistic mesh deformations to be obtained.

2.3.4. Sketching terrain

Sketching has been commonly applied to model general 3D objects, although there have been proposals to model natural elements like plants [126], clouds [127] or terrain [128, 129].

The Harold system [130] was an initial approach for sketching-based modeling. The basic idea was to create a 3D world with the sketched information but populating it with the sketches, so that the final image is similar to the sketched one but allows the user to change the viewpoint and interact with the scenario. With this objective the main primitive of the Harold system was the billboard in which the strokes were stored. Regarding terrain, the sketched lines of the user define bumps and hills that are considered to lift the affected objects. This system was very promising as it could capture a child's drawing into a 3D world, although the scenes observed from a different viewpoint usually included strange artifacts. Nevertheless, their objective was not realistic rendering and their results could not be outputted to be used in other applications.

Later, Watanabe et al. [128] proposed a Java-based application in which the user could use simple line strokes to create the mountains. Moreover, the user could add noise, textures or rivers to offer a more realistic appearance. This initial work offered some promising results although the terrain totally lacked realism and the operations allowed

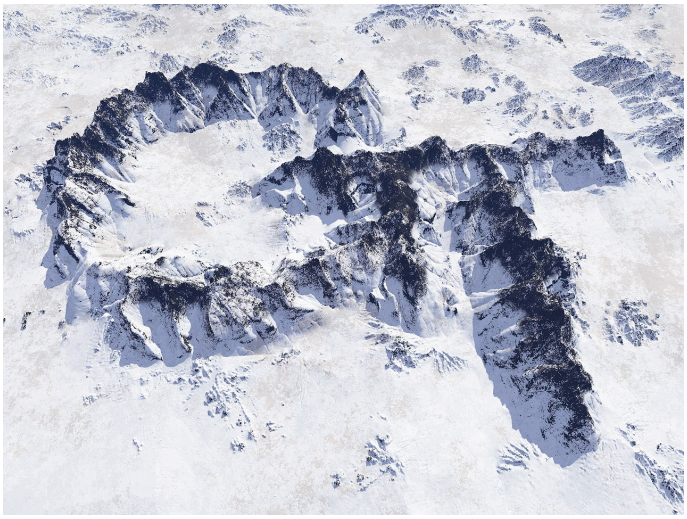


Figure 2.10: Image from the work of Zhou et al [17] were the shape of the letters GT are used to define the terrain generation.

by the software were very limited.

The work developed by Zhou et al. proposed a solution from a totally different perspective [17]. In their approach patches from sample terrain (obtained from a DEM) were used to generate new terrain and the synthesis was guided by a user-sketched feature map that specified where terrain features occurred in the resulting synthetic terrain. Although the results were very realistic, the user implication in the finally obtained terrain was limited and complicated. The beautiful images (see Figure 2.10) offered by the authors were obtained with Terragen [103], although later the proposal was included into the World Machine commercial application [106] as a plug-in.

Following a similar technique, Belhadj [131] presented in the same year a method for reconstructing terrain from Digital Elevation Models. His approach uses fractal-based algorithms for performing the reconstruction and enables the users to sketch details on the terrain or create a new model from scratch.

Recently, Gain et al. proposed a new sketching application for terrain modeling [129]. In this solution the authors offer the users the possibility to sketch the silhouette of the heights of the mountains and also the area of influence of this silhouette, widening or stretching the mountains.

They also developed a fast multiresolution surface deformation so that the mesh representing the terrain can adapt to those areas where the surface is more detailed. Nevertheless, the proposed solution still requires a complex interaction from the user where multiple views of the terrain are needed in order to obtain the desired terrain. Moreover, intersecting mountains can become difficult to work with as the user must decide if a new mountain must be in front or behind an existing one while drawing the silhouette.

Based on the graph theory, the method proposed by Rusnell et al. [132] uses user-drawn strokes to define the main features of the terrain and applies *path planning* to generate the terrain. Although the control over the obtained terrain is slightly limited, they present nice visual images by using the Terragen software [103].

2.4. Conclusions

This chapter has presented the state of the art on different aspects of modeling natural phenomena. We have analyzed the latest solutions developed for real-time rendering of rain and ocean and also for modeling terrain.

Most techniques for rain rendering are oriented towards particle systems, as they can offer enough flexibility to model effects like light interactions or collisions. A limitation of most solutions is the fact that they are not suitable for scenarios where the camera moves very fast, as re-locating the particle system to follow the movements of the user is costly and poses a limitation for maintaining realism.

Regarding ocean rendering and animation, the latest approaches propose the use of animated heightmaps to model the ocean surface. Within these surfaces, a large number of effects like reflection, refraction, fresnel factor or collisions with other objects have been studied. Nevertheless, the underlying geometry has received less attention and some tessellation techniques have been proposed. Therefore it would be interesting to develop new techniques which can adapt in real time the detail of the surface mesh according to different parameters.

Lastly, for terrain generation we have described different sets of techniques that have been proposed to offer highly realistic synthetic terrain. The software community has commercialized many software tools that can offer incredibly realistic environments. However, these software applications are usually very difficult to use and the artist is presented with

a large amount of tools to model the terrain. Therefore, usability is a key element and sketching is a very promising solution that can simplify the interface while still producing realistic terrain.

CHAPTER 3

Creation and Control of Rain in Virtual Environments

Realistic outdoor scenarios often include rain and other atmospheric phenomena, which are difficult to simulate in real time. In the field of real-time applications, a number of solutions have been proposed which offer realistic but costly rain systems. Our proposal consists in developing a solution to facilitate the creation and control of rain scenes and to improve on previously used methods while offering a realistic appearance of rain. Firstly, we create and define the areas in which it is raining. Secondly, we perform a suitable management of the particle systems inside them. We include multiresolution techniques in order to adapt the number of particles, their location and their size according to the view conditions. Furthermore, in this work the physical properties of rainare incorporated into the final approach that we propose. The presented method is completely integrated on the GPU.

3.1. Introduction

Rain is an extremely complex natural atmospheric phenomenon. Research has been carried out in the area of computer graphics in order to describe methods which define not only the rainfall [1, 2], but also their

interaction with other surfaces [3, 133, 4] or even the water accumulation [5].

Our interests focus on rendering rain streaks in real time. Most of the previously proposed solutions can be classified into two groups: image-based and particle-systems-based. Traditionally, authors have developed rain simulation frameworks which are based on particle systems. A particle system is a collection of particles that have a set of properties that are updated at discrete times, which creates a dynamic system that evolves over time by modifying its properties when necessary [134]. These systems have been successfully used to simulate in real time different kinds of fuzzy phenomena, like smoke or fire. Nevertheless, they present some limitations due to the cost of translating and rendering the high amount of raindrops that must be represented in order to offer a realistic rain appearance.

Although these works describe methods for real-time rendering of realistic rain, they are not easily applied in virtual environments. This happens due to the fact that these techniques use nearly all the computer resources available, leaving the rest of elements of the scene without almost any resource. Due to this fact, recent games which incorporate rain rendering still use simplistic approaches.

Some of the previous solutions attempt to take advantage of the capabilities of current GPU, which use an architecture with a programmable *pipeline* that allows it to be supplemented with *vertex*, *geometry* and *pixel shaders* running in the hardware [135]. Despite the constant evolution of graphics hardware, there is still room for the development of a complete system for real-time rain rendering in complex environments due to the high computational cost of existing solutions and to the fact that these solutions fail in interactive scenarios where the user moves continuously, since adjusting the particle system to the new camera condition is too expensive.

Level-of-detail modeling techniques have been proposed in many cases as a solution to diminsh the geometry complexity of the scene to visualize. As for the problem of rain rendering, we could apply level-of-detail techniques to those methods based on particle systems. Nevertheless, little research has been performed on multiresolution models for particle systems. We could highlight the approach presented in [136] where the authors propose a hierarchy of particles and consider the use of physical properties as the criterion for switching among LODs in order to obtain a higher resolution when necessary. More recently, Gundersen and

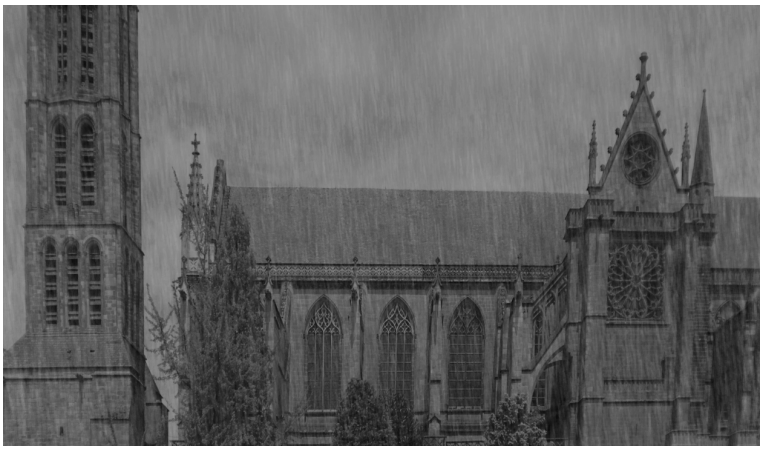


Figure 3.1: Rain simulated by our algorithms in a scene with a textured skybox of the Limoges cathedral.

Tangvald [137] developed a non-GPU LOD strategy for fire simulation where they altered the size and amount of particles when rendering the scene.

The work presented in this chapter is proposed as a solution to efficiently design and visualize rainy scenes (see Figure 3.1). One of the main objectives is to offer a rain scheme that is suitable for scenes where the user keeps moving all the time, as happens in virtual reality environments or computer games. To achieve this objective we propose a method for automatically generating rain environments with the definition of rain areas and with an adequate management of the particles created inside these areas. Furthermore, we include LOD techniques in order to adjust the size and the number of particles to the conditions of the scene, lessening the cost of managing and rendering the particle systems.

The solution we are presenting includes the following features:

- Physically-based simulation of realistic rain, by considering some of the features presented in the previous chapter, Section 2.1.2 to develop a realistic rain simulation.
- Definition and management of rain zones, as well as methods for obtaining smooth transitions between zones with different rain conditions.

- Multiresolution particle systems for rain.
- GPU implementation, exploiting *vertex* and *pixel shaders* and also the more recent *geometry shader*.

This chapter is organized as follows. In Section 3.2 we introduce the concept of raining area as well as its interactions with the observer. Then, Section 3.3 describes how the physics of rain analyzed in the previous chapter have been considered in this simulation scheme. Section 3.4 presents the level-of-detail particle system for simulating rain. Section 3.5 provides a proposal for improving the raindrops creation on GPU. Section 3.6 offers the results obtained with our model and a comparison with other solutions. Lastly, Section 3.7 contains some remarks on the solution presented and the results obtained.

3.2. Simulating Rain Areas

Real-world raining scenarios include both rain areas and also non-rainy areas. Even when it is raining in a very wide area, we must assume that we could always find non-rainy areas nearby. Traditionally, rain simulation has not considered this issue and their rain systems do not provide smooth transitions between areas with different rain conditions.

The rain framework we are introducing tries to establish with reasonable precision where it is raining and where it is not raining inside a given scenario. Furthermore, we also want to handle transitions in order to create a visually realistic raining effect. An important aspect of this realism is the possibility of looking at a *far rain* area from a non-rainy place.

We have considered that a *raining area* is simulated as an ellipse with two initially given radii,

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1 \tag{3.1}$$

centered on x_0, y_0 with radius a, b, being $a, b \in \Re$ and $a > b$. Inside this ellipse, we have defined an attenuation zone on the borders of the ellipse. This zone will be used for decreasing progressively the number of rendered raindrops when the user is leaving the rain area. The diagram in Figure 3.2 depict the attenuation area that we have just mentioned.

In addition to the definition of the rain area, it will also be necessary to create and manage a *rain container*. This rain container encloses the whole set of raindrops included in our particle system. Under our circumstances, the rain container will be shaped as an elliptical cylinder with its radii adjusted to the observer's field-of-view (FOV). In order to achieve better performance and to obtain a more cost-effective distribution of particles for the level of detail, we decided to skip the back part of the cylinder. Our semi-cylindrical rain container is better adapted to the field-of-view of the user. It is more ergonomically adjusted to the field-of-view than a box [12], it is more efficient than a whole cilynder [30] and, in addition, it allows the user to perform small turns to the right or to the left without requiring to reposition the whole rain container. More precisely, if the user makes changes in its orientation of less than 45°in any direction then it will not be necessary to reorient the container as the rain perception is maintained. In Figure 3.2 we have displayed a bird's eye view of our semi-cylindrical rain container, where it can be observed how the observer, the *frustum* and the rain container are located in the scenario.

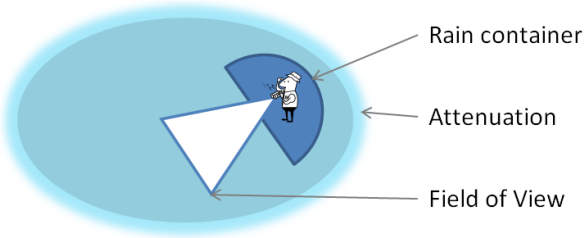


Figure 3.2: Rain container.

As we already commented, the shape of the container that we have selected offers some improvements on the rain containers from previous solutions [12, 30]. The management cost of their particle systems makes it impossible to manage bigger containers, as this would oblige the systems to create and update a huge number of particles. In order to overcome this limitation and use a bigger container, our solution incorporates level-of-detail techniques to modify the particle size and location while preserving the realistic appearance of rain. Thus, by combining these techniques with the selected shape of the container, our

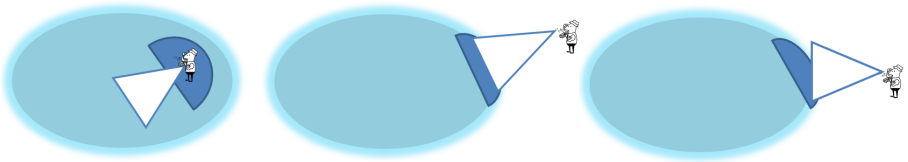


Figure 3.3: User point of view of the rain area, showing the two possibilities. In the first picture the *close rain* and in the other two, the *far rain*.

method offers an appropriate solution for reacting efficiently to changes in the position and direction of the *frustum* of the user.

3.2.1. User Interaction

Once we have defined the *rain area* and the *rain container*, it is necessary to consider how the user interacts with the rain area. These interactions will affect the rain container behavior and the features of the particle system enclosed within the container.

The user could interact with a rain area in two different ways:

1. The user is inside the rain area, completely surrounded by raindrops. We have named this situation *close rain*.

2. The user is outside the rain area, while looking at it from a distant place or moving away from the area. In these cases we use the *far rain* approach.

These two possibilities are shown in Figure 3.3. It is important to note that the rain container has different behaviors depending on whether we are using the *close rain* or the *far rain* approach. So, the image on the left refers to *close rain* while the other two images show two possible circumstances in which we must use *far rain*.

Close Rain

This case occurs when the observer is in the rain area. In order to check whether the observer is inside this area, we use Equation 3.1. We can test the location of the observer by checking whether the left member

of the equation is less than 1. In such cases, we assume that the observer is inside the ellipse.

The rain container is initially centered on the observer, as can be seen in Figure 3.2. Nevertheless, the location of this container is continuously updated in order to follow the user's movements. So, if the user moves then the container moves with him; if the user changes the view direction then the container will be reoriented too. We have given the user a room for movement to allow him to make small movements and to change the viewing direction without altering the position of the whole rain container. Thus, if the user exceeds the boundaries of this room the rain container will be relocated to follow the user's movement and the new viewing conditions.

Far Rain

Far rain refers to a scenario where the observer is located far away from the rain area. This situation happens when the camera is situated in an area where it is not raining and the camera can look at the area where it is actually raining.

In order to check whether the user can see the rain area, we will do a simple test of the rain area against the *frustum* plane. If the visibility test is passed, we should detect whether the observer can see the whole rain area or just a part of it. If the observer can only see a part of it, it will be necessary to calculate the size of this part.

Regarding the behaviour of the rain container, the first step consists in locating the rain container within the perimeter of the ellipsoidal rain area. Inside this perimeter, the container is centered in the part that is visible to the user. Secondly, we should modify the size of the rain container in order to cover the whole part of the ellipse that the observer can see. Figure 3.3 shows two possible adaptations of the *far rain* container with regard to the features of the observer view. In a similar way, the height of the container should be expanded in order to simulate the raindrops falling from the sky.

Figure 3.4 shows an image of the *far rain* that we have just defined. We have rendered a part of the rain streaks in red in order to make it easier for the user to perceive how the rain zone is far away from the observer. We can notice how the particles are covering the suitable area and it is not raining in the area close to the user. This is just an example to show what a user could see from outside the rain area. We

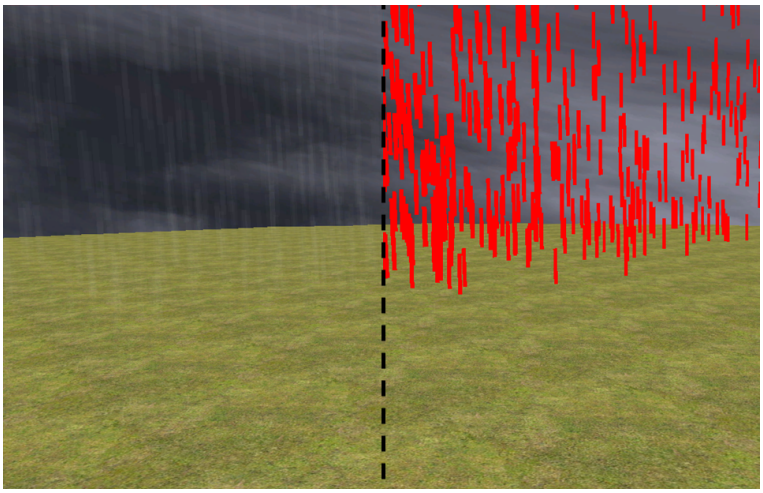


Figure 3.4: *Far rain* snapshot. On the right streaks are rendered in red to facilitate its perception.

could simulate hard rain by increasing the number of particles and their size.

Transitions between rain cases

Within the proposed solution for simulating rain particles, it is important to control the transitions that an observer can experiment when changing from *close rain* to *far rain* and vice versa.

The transition from *close rain* to *far rain* occurs when the user leaves the rain area. As the user approaches the limits of the raining area, it will cross the attenuation zone, perceiving a progressive decrease of the number of particles. When the user is positioned on the perimeter of the ellipse that defines the rain area the system must react properly. In that case, the rain container remains static at that point. The user will perceive a final decrease of the amount of raindrops while it moves away from the rain container which is no longer following the user's movements.

Regarding the transition from *far rain* to *close rain*, we again use the perimeter of the rain area. As we have mentioned earlier, the size of the container is adapted to the size of the observer's frustum. When the observer gets closer to the rain area, the size will decrease until the

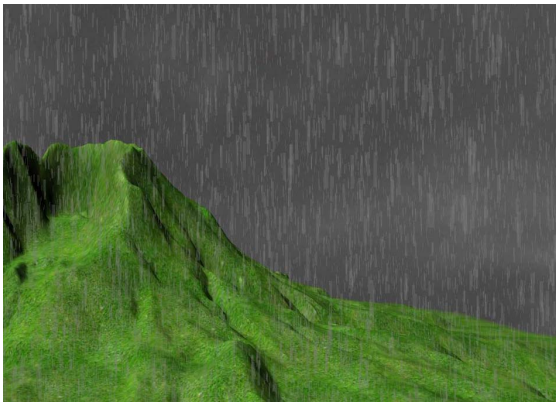
Table 3.1: Rain types characteristics.

Rain Type	Type of Drops	Radius (mm.)	Speed (m./s.)	Particle Number
Convection Rain	Raindrop	[1.25,3.75]	[7.46, 9.23]	50,000
Frontal Rain	Droplet	[0.4,1.25]	[3.67,7.46]	25,000
Relief Rain	Drizzle drop	[0.2,0.35]	[1.62, 2.87]	15,000

user crosses the perimeter of the initially defined rain area. That is the moment when we make the change between the two ways our solution can work. When swapping from *far* to *close rain*, the system will initially render a small amount of particles that will increase progressively as the user crosses the attenuation area.

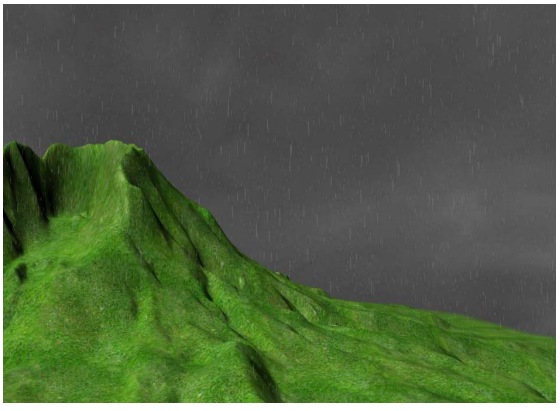
3.3. Rain Physics

In Section 2.1.2 we introduced several aspects of the physics of rain that are important for rain simulation. As one of the objectives of the proposed solution is to consider the physics of rainfall, when initializing the particle system we will adjust the attributes of the raindrops depending on the selected rain type. Table 3.1 presents the characteristics that we have chosen for rendering the different rain types. Every type has a range for the raindrops size; moreover, every raindrop size has associated a range for its own speed (see Table 2.2). All these adjustements are performed when initializing the particle systems, randomly selecting each value inside the suitable range. It is important to note that the size and the speed of the particles must be different from one particle to another, in order to prevent all the particles from having the same appearance and from following exactly the same trajectory. In order to represent the *retinal persistence* effect realistically, the rain particles will be shaped as vertical streaks, whose pixels will get contributions of successive positions of the raindrops. In our rain simulation system we will represent the raindrops by using *quads* as the rendering primitive, as they are more accurate to the streak-shaped perceived raindrop. Figure 3.5 depicts three images of our rain simulation approach by rendering the three rain types introduced in Table 3.1.



(a) Convection rain.

(b) Frontal rain.



(c) Relief rain.

Figure 3.5: Rain appearance throuhgout the three different rain types.

3.4. Level-of-Detail for Rain Particles

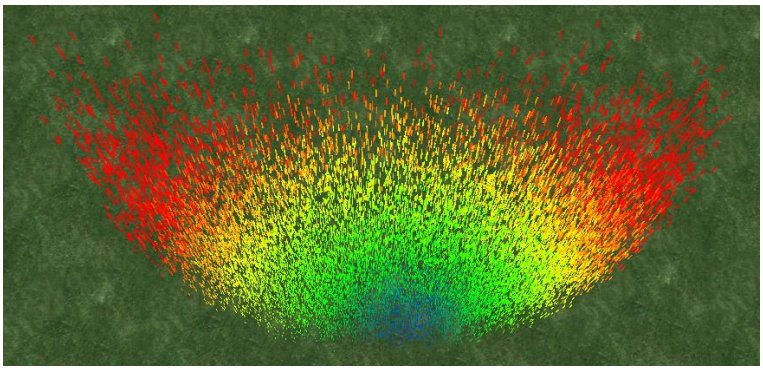
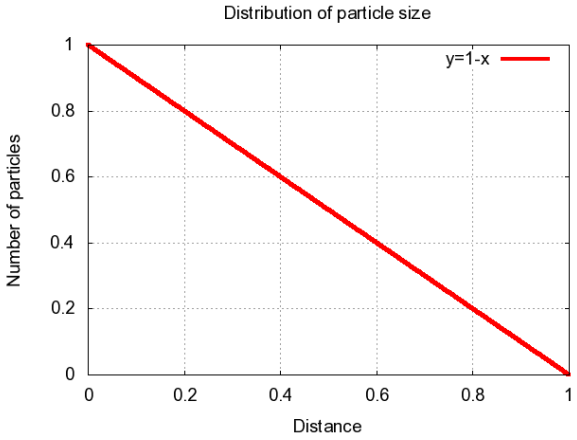


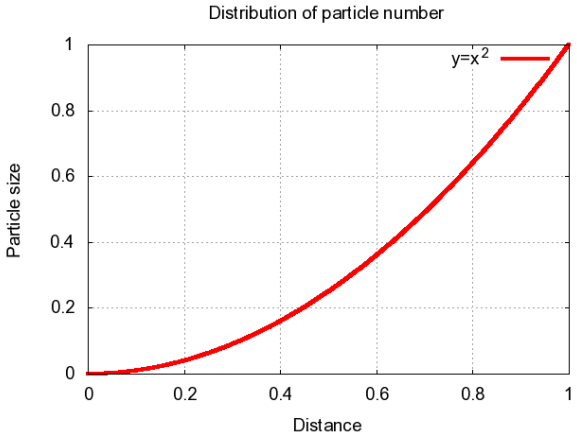
Figure 3.6: Size and number of particles depicted from a bird's eye view.

The particles inside the rain container are modified in order to follow a level-of-detail pattern. The main idea of our algorithm is to adapt the number of particles and their size according to the distance from the observer. Thus, we render more particles in those areas which are closer to the observer, and fewer in those which are farther away. The reader is referred to Figure 3.6, which shows a bird's eye view of a scene where the observer is located in the bottom center of the figure. The color of the particles follows a gradation according to the distance to the observer's eye, starting with blue for the closer particles and ending in red for the farther ones. Furthermore, we can also see how the closer the particles are to the observer, the more particles are rendered. Regarding the size, to render this figure we also considered the distance criterion to adapt the size of each particle according to the distance. Therefore, particles that are located closer to the viewer are smaller than those which are farther away.

It is important to comment on the functions used to modify the particles characteristics. We must note that these functions are one of the basic parts of our multiresolution model. Initially we decide on a number of particles that we will maintain for the entire simulation process. The distribution of particles should not be the same throughout the whole container. As we have already mentioned, we should render more particles in the zone that is closer to the viewer.



(a) Linear distribution for particle number.



(b) Quadratic distribution for particle size.

Figure 3.7: Different distribution functions for size and number of particles.

In the moment we initialize the particles, we must maintain a proportion between the distance and the number of particles. This relation follows a linear distribution (see Figure 3.7), as we need them to both decrease and increase at the same time, as it is defined in Equation 3.2. As we can see in the equation, we need the number of particles to decrease while the distance keeps growing. In order to obtain a realistic

simulation, it is necessary that the number of particles decreases progressively depending on the distance. A non-linear distribution would not be suitable because it would imply a faster increase of the number of particles at the beginning or at the end of the container. As a consequence, it would not give a realistic rain impression.

The size of the particles is continuously updated. In this case, the relation is not linear but quadratic (see Figure 3.7). This distribution is defined in Equation 3.3. The function indicates that the size will increase little by little at the beginning and faster at the end. We only want to visualize big particles in those zones which include a small number of particles (those areas located further away from the observer).

It is important to mention that, following the physics, the speed of the particles is the same for all of them, although the size of the particles is increased according to LOD. Perceptually to the eye, farther raindrops fall slower than closer ones. This perception is correct since it gives the observer a proper perspective projection. Moreover, if we modified the speed of the back particles, then the sense of depth would be lost and all the particles would look like being close to the observer.

$$num\_particles \cong 1 - distance \tag{3.2}$$

$$size \cong distance^2 \tag{3.3}$$

3.4.1. Implementation

One of the main aims in the implementation of our framework is to make optimum use of the graphics hardware. Similar to the way that a CPU works, the GPU in the graphics card also has a pipeline filled with small stages allowing it to perform different tasks very quickly. The idea is to upload the input data to the graphics memory in order to improve the rendering by using the GPU programmable units, which are also called *Shaders*.

The basic framework of our rain model is quite similar to what is presented in the work by Pierre Rousseau et al. [12] and Sarah Tariq [30]. The framework has the following steps:

- Initializing a particle system for rendering the rain inside a defined zone.

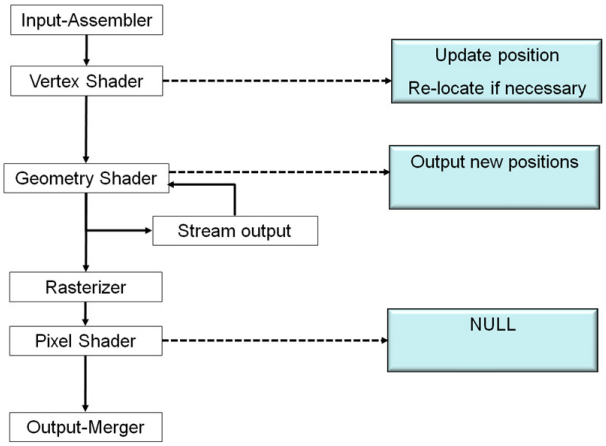
- Defining an initial position for the particles that is reused every time the particles fall outside of the bounds and need to appear again.
- Updating the location of the rain container to follow the user's movement.
- Using the graphics pipeline twice for each scene rendering. In the first pass we update the position of the particles when falling and in the second pass we render the particles by using the GPU to expand dots into oriented-to-the-viewer quads.

In order to adapt this framework to the method we are presenting, we must modify several of the mentioned steps. Firstly, the initialization of the rain particles must follow the level-of-detail criteria presented in the previous section. Secondly, the shaders on the GPU should be also modified. The vertex shader of the first pass must consider the size and location of the rain container to relocate the particles. On the second pass, the pixel shader that creates the quads must adapt their sizes following the aforementioned criteria. Figure 3.8, shows the graphics pipeline with the two passes that we have just mentioned. Nevertheless, in this section we will address all these processes in more detail. Moreover, Algorithms 1 and 2 present a pseudo-code for the Vertex Shader of the first passs and the Geometry Shader of the second pass respectively, as these are the more important shaders of our framework.

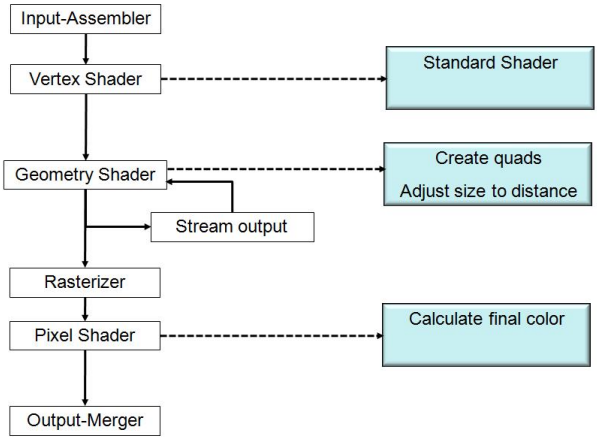
During the development of this framework we tried to perform all these tasks in only one single pass. The difficulties appeared when working in the pipeline at the same time with points and quads. The first pipeline is in charge of translating vertically each raindrop to simulate the rainfall. In this pass we use points as the rendering primitive. Following that, the second pipeline pass will initially receive points but will output quads for the rain rendering. By updating directly the quads we would be obliged to translate, re-locate and output four vertices instead of one for each raindrop, which our tests have proven to be much more costly.

Particle initialization

We initially populate the rain container. The raindrops are distributed following the criteria introduced in the previous section in order to



(a) First pipeline pass.



(b) Second pipeline pass.

Figure 3.8: The stages in the graphics pipeline .

achieve a linear distribution of the particles depending on the distance. The reader is again referred to Figure 3.6. These particles are created in the CPU and later into the GPU. All the information about the raindrops is kept in a data structure: original size, original position, current position and speed. These are registers which are directly uploaded from the CPU to the GPU, thus avoiding the use of the textures that were used by other authors in previous models [12].

Here we present a piece of code in DirectX10 containing the register structure used to keep the particle data just mentioned above:

```
struct RainVertex{
D3DXVECTOR3 originalPosition;
D3DXVECTOR3 originalSize;
D3DXVECTOR3 currentPosition;
D3DXVECTOR3 speed;
}
```

It is important to note that the size and the speed of the particles must be different from one particle to another, in order to prevent all the particles from having the same appearance and from following exactly the same trajectory. As we commented in the introduction, one of the objectives of our solution is to consider the physics of rain. As a consequence, when initializing the particle system we will select their features, such as their size and speed, depending on the selected rain type. Following the information presented in Section 2.1.2, Table 3.1 presents the characteristics that we will use for rendering the different rain types.

Before visualizing the raindrops we must indicate how many drops the GPU should render. The amount of particles will vary depending on the rain type we are rendering, depending on whether the user is in an attenuation zone and depending on whether we are visualizing *close* or *far rain*.

As we mentioned at the beginning of this section, we need two pipeline passes for each rendering of the final scene.

In the first pipeline pass, see Figure 3.8(a) and Algorithm 1, the *Vertex Shader* updates positions and relocates particles that have gone out of bounds.

The *Geometry Shader* uses the *StreamOutput* to store the updated positions in vertex buffers. These vertex buffers for the rain will be used to *pingpong*, which consists in having two *Vertex Buffers*, one for reading the current positions and the other for writing and updating them. Later on, once the changes have been performed, we swap them.

In the second pipeline pass, see Figure 3.8(b) and Algorithm 2, the rain particles are actually rendered. The *Geometry Shader* transforms each vertex into a *quad* to represent the rain streak. Using the original vertex and the position of the viewer, the other three vertices are

calculated so that the obtained quad is oriented toward the camera. Moreover, in this step we assure that the streak has an adequate size depending on the rain type and the distance to the viewer. The *Pixel Shader* is in charge of calculating the final color. In our framework, all the rain streaks have the same transparency value, although we could easily modify it to obtain further effects.

Particle animation

We have just introduced the initial creation of our particle system and the basic ideas for rendering the particles on the GPU. Nevertheless, it is important to analyze the way the particles evolve as time passes. This evolution affects the position and the size of the particles.

Firstly, regarding the position of the drops, the *Vertex Shader* of the first pass is responsible for displacing the raindrops and relocating them at new positions if they fall out of bounds. In this latter situation, the new position is calculated considering the original position stored in the data structure of the particle and the current position of the observer. Secondly, we must adjust the particle size depending on the distance to the viewer. The purpose of the *Geometry Shader* in the second pass is thus to calculate the most suitable size before converting the particle into a *quad*.

The *far rain* method requires a slightly different implementation of these *Shaders*. We must remember that the *far rain* approach involves working with a bigger rain container. As a consequence, when the particles fall out of bounds, the drops are relocated considering their original position but also the current size of the container, in order to cover all the raining area that the observer can see. Moreover, it needs greater particle sizes so that they can cover the whole rain area that is visible to the observer.

3.5. Improving the Level-of-Detail for Rain Particles

The solution that we have presented forced the system to initially define the position of each particle and also ther quantity of raindrops. As a consequence, the spatial distribution of the raindrops was fixed and initially defined on the CPU. This limitation can be overcome if we develop a technique that decides to render more or less particles directly on the GPU.

Algorithm 1 Vertex Shader pseudo-code of the first pass.

```
// Input variables
float * speed;
float * originalPosition;
float * viewerPosition;
float height;

// Input/Output variables
float * currentPosition;

currentPosition = currentPosition + speed;

if (particlePosition.y ≤ (viewerPosition.y - height)) then
  currentPosition=originalPosition;
end if
```

Algorithm 2 Geometry Shader pseudo-code of the second pass.

```
// Input variables
float * currentPosition;
float * viewerPositionOrientation;
float * originalSize;

float distance=calculateDistance(currentPosition,
                                 viewerPositionOrientation);
float size=calculateSize(distance,originalSize);

float* vertices=calculateVerticesQuad(size,currentPosition,
                                      viewerPositionOrienta-
tion,distance);

Output(vertices);
```

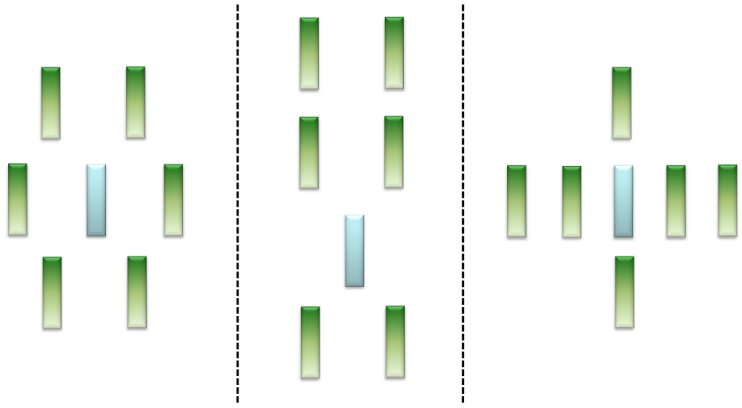


Figure 3.9: Sample patterns for rain generation.

The improvement we propose in this section maintains the need of two rendering passes presented before:

- The first pass is in charge of updating the position of each particle.
- The second pass applies the multiresolution techniques to modify the size of the particles and also renders the final particle system.

The main idea of the improvement is to use the second GPU pass to dynamically create more particles in those areas that are closer to the user.

A possible way to create in real-time more particles in those areas that need it is to use patterns. Thus, the idea is that the *Geometry Shader* of the second rendering pass uses the particle that receives as input as a seed to generate more particles. In Figure 3.9 we present three sample patterns that are used to generate more particles from an initial one. The seed particle is depicted in blue and the new generated ones in green.

As we have mentioned before, the distance is the criterion used to decide the most suitable size for a particle. This criterion is also used to decide how many particles we should render from each seed particle. It is relevant to note that we have decided to create a maximum number of 6 raindrops for each seed particles. Our tests have proven that with that number it is enough to get a proper rain impression without increasing too much the rendering cost. Moreover, we have applied those three

patterns in our test as they are sufficient to avoid the user to perceive repeated patterns. Nevertheless, both the number of generated particles and the patterns followed could be easily modified in our framework.

3.5.1. Implementation

When we introduced the implementation of the previous rain scheme, we commented that, in order to avoid all the raindrops following similar falling paths, we initially give each particle a slightly different falling speed and direction. Similarly, creating the particles strictly following the presented patterns would entail noticeable repeating images. To avoid these visual artifacts, we have included in the information of each particle a different random value. This value will be used to modify the final positions of the particles created using the patterns.

Consequently, the final data structure that our framework will use to update and render the particle system will be composed of six elements (four of them used in the solution presented before):

- currentPosition, updated in the first pass of every render.
- originalPosition, used to re-locate the particle when it leaves the end of the container.
- originalSize, different among particles to avoid visual repetitions.
- speed, intially selected acording to the original size of the particle.
- pattern, randomly selected among the three samples presented above.
- randomValue, used to stochastically translate the particles created following the pattern.

The first pass of the rendering pipeline will remain unaltered, as we will still have to translate the vertices and relocate them when necessary. For each particle, the *Geometry Shader* of the second pass will be in charge of applying the level-of-detail. In this *shader* will be when we will apply the patterns to generate more particles when required.

Thus, for each particle the *Geometry Shader* of the second pass will execute a code similar to that presented in Algorithm 3. This *shader* will start by calculating the appropriate size according to the distance and

the original size of the raindrop. Then it will generate the 4 vertices of the quad using the new size adjusted to the distance and also the position and orientation of the user to orient the quad towards the camera. Later, depending on the distance from the raindrop to the camera, we will create as many replicas as desired. For each of these new quads we will calculate the position of the vertices using the original ones and the random value to avoid visual repetitions.

Algorithm 3 New Geometry Shader pseudo-code of the second pass.

```
// Input variables
float * currentPosition;
float * viewerPositionOrientation;
float * originalSize;
float pattern,randomValue;

float distance=calculateDistance(currentPosition,
                                 viewerPositionOrientation);
float size=calculateSize(distance,originalSize);

float* vertices=calculateVerticesQuad(size,
                                      currentPosition,
                                      viewerPositionOrientation,distance);

Output(vertices);

if size ≤ maximimumDistance) then
  float* newVertices;
  int replicasNumber = calculateReplicas(distance);

  for i = 0 to replicasNumber do
    newVertices=calculateVerticesReplica(vertices,
                                         pattern,randomValue,
                                         viewerPositionOrientation);
    Output(newVertices);
  end for
end if
```

3.6. Results

In this section we will study the performance of our proposed method by analyzing the visual quality obtained as well as the number of particles that can be rendered in real-time. Our solution was programmed with HLSL and C++ on a Windows Vista Operating System. We have used DirectX10, which incorporates many improvements and makes it possible to use *geometry shaders* and the *StreamOutput* flow to memory resources. The tests were carried out on a Pentium D 2.8 GHz. with 2 GB. RAM and an nVidia GeForce 8800 GT graphics card.

3.6.1. Level-of-detail

Figure 3.10 compares the rain perception that the user obtains when enabling or disabling our level-of-detail scheme. Both of the images of this Figure offers on the left side the results obtained without enabling LOD and on the right side the rain obtained when applying our LOD solution.

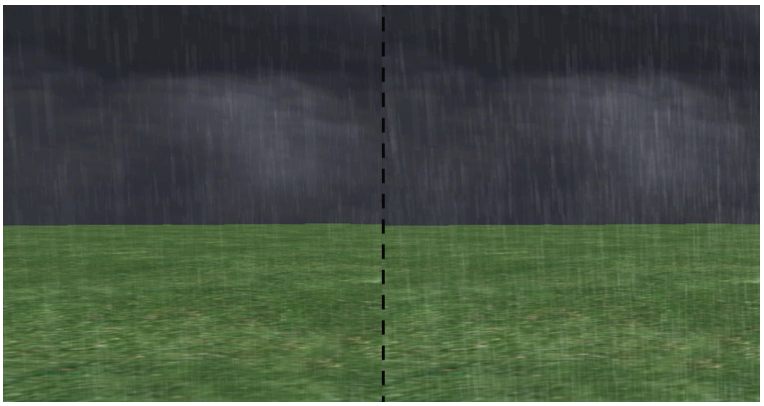
In Figure 3.10a it can be seen how the use of LOD can offer a harder rain impression with the same amount of particles. As a consequence, if we use our multiresolution techniques we simulate much more rain particles with the same computational cost.

Figure 3.10b shows the same rain screenshot but coloring the particles depending on the distance, as we did in Figure 3.6. On the one hand, the image on the left uses the same size for all particles and, as a consequence, those particles located in the back part of the rain container are rendered as small quads. On the other, the image on the right shows how the particles in the back part present a bigger size, giving more rain impression.

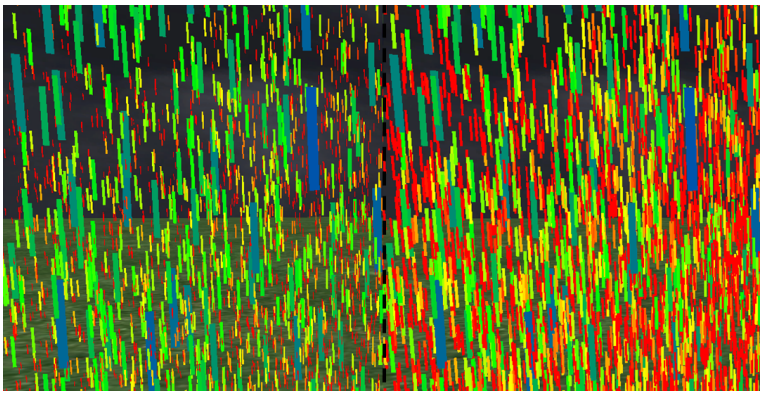
3.6.2. Visual Comparison

In the previous section, Figure 3.5 presented three images of our rain simulation system depicting the three rain types described in Table 3.1. These images are helpful to demonstrate that our approach is capable of rendering different rain scenarios properly.

In this section we offer a visual comparison between our framework and two previously developed rain methods based on particle systems: Rousseau et al. [12] and Sarah Tariq [30]. Figure 3.11 offers three screenshots of these approaches rendering a hard rain environment with similar



(a) Rain appearance with 20,000 particles.

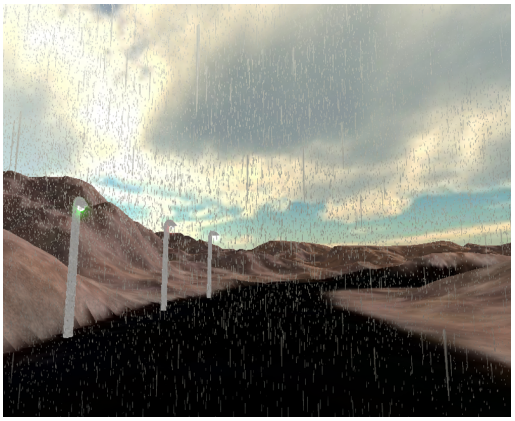


(b) Rain appearance with 20,000 particles. Particles are colored according to the distance.

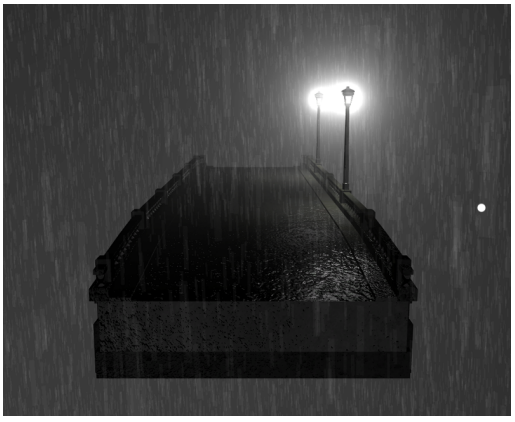
Figure 3.10: Comparison of rain appearance in our model without (left) and with LOD (right) enabled.

rain intensity. It can be seen how our approach can offer a similar rain appearance with considerably fewer particles, the number being reduced to about 30 % of those required by Sarah Tariq's method and to around 55 % of the number used by Rousseau et al.

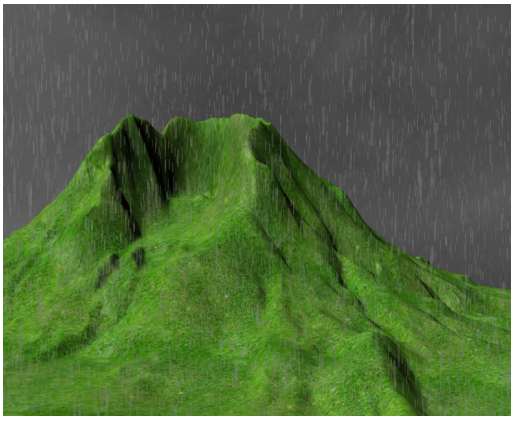
From a different point of view, Figure 3.12 presents the three rain frameworks rendering a similar scene with the same amount of particles. It can be seen how the rain intensity sensation given by our framework is much higher than the ones offered by the other rain solutions. We have also included for each framework an image where the particles



(a) Rousseau et al. rain model with 90,000 particles.

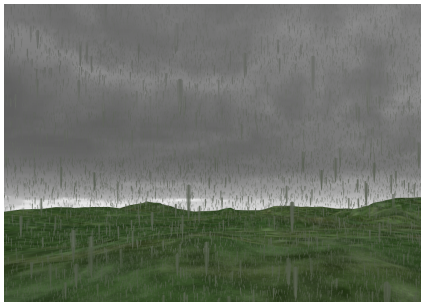


(b) S. Tariq rain model with 150,000 particles.

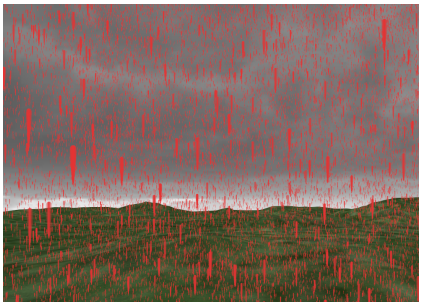


(c) Our rain model with 50,000 particles.

Figure 3.11: Comparison of rain appearance in our model and recent previous methods.



(a) Rousseau et al. rain model.



(b) Rousseau et al. rain model with red-colored particles.

(c) S. Tariq rain model.

(d) S. Tariq rain model with red-colored particles.

(e) Our rain model.

(f) Our rain model with red-colored particles.

Figure 3.12: Comparison of rain appearance in a similar scene when rendering 100,000 particles.

are colored in order to clarify the rain sensation that can be obtained. It is important to mention that the particles in movement are much more realistic than the simple static snapshots that we are depicting throughout the paper.

3.6.3. User Study

We have considered that it is compulsory to perform a user study in order to evaluate the quality of our rendered rain. The objective of this study is to obtain the number of particles that would be necessary for obtaining a similar rain impression by using different rain rendering frameworks.

Our test consisted on comparing our solution against other 3D rain frameworks: [12, 30]. We have decided to analyze two rain intensities, one rendered in our application with 25,000 particles and named *Slight Rain* and the other with 50,000 particles and named *Intense Rain*.

The user is presented with two screens at each time. One of them renders all the time our rain application, while the other will present in turns the other frameworks. Each time, the user is asked to indicate how many particles would be necessary in the other solution in order to obtain the same visualization intensity given by ours. To facilitate this task, the user can use two keys of the keyboard to increase or decrease the amount of rendered particles of the compared system.

The test has been conducted among 25 people. The order in which the tests have been passed is random every time. As many volunteers were not completely familiar with computer graphics, we carefully explained them some basic concepts in order to allow them to understand the test more deeply so that the results could be as realistic as possible.

Model	Rousseau et al. [12]	S. Tariq [30]	Our Model
Slight Rain	37,000	62,000	25,000
Hard Rain	95,000	167,000	60,000

Table 3.2: Number of particles needed for obtaining the same rain intensity perception.

In Table 3.2 we present the average number of particles indicated by our volunteers. It can be seen how our approach can offer a similar rain appearance with considerably fewer particles, the number being reduced

Table 3.3: Comparison of the results obtained from the hard rain scenarios presented in Figure 3.11.

Model	Rousseau et al. [12]	S. Tariq [30]	Our Model
FPS	212	266	443

Table 3.4: Comparison of the results obtained in a similar scene when rendering 100,000 particles.

Model	Rousseau et al. [12]	S. Tariq [30]	Our Model
FPS	202	333	359

to about a 30 % of those required by previous solutions.

3.6.4. Performance Comparison

Table 3.3 cointains the results obtained when rendering the scenes depicted in Figure 3.11. It is important to note that the frame-rate test performed to obtain the *fps* was carried out visualizing only rain particles, without including any other elements from the scenario such as backgrounds, light interactions or other environment features. Table 3.3 shows how the number of frames per second (fps) obtained in our method is much higher than in the other solutions. So, the model that we are presenting is capable of obtaining a visual aspect that is quite similar to the other methods with a smaller number of particles and a performance increased in 60 %.

Similarly, Table 3.4 provides the frames per second obtained for each scene simulated in Figure 3.12. These results show that, when displaying the same amount of particles, our approach still presents the fastest render. Thus, as it can be seen in Figure 3.12, when rendering 100,000 particles our rain simulation system is capable of offering a harder rain sensation with a higher performance.

Performance of the pattern-based approach

In a previous section we introduced an improvement over the rain simulation framework were we proposed the creation of particles on the GPU in real time, being capable of generating more particles in those areas that require them. The objective of this improvement was to in-

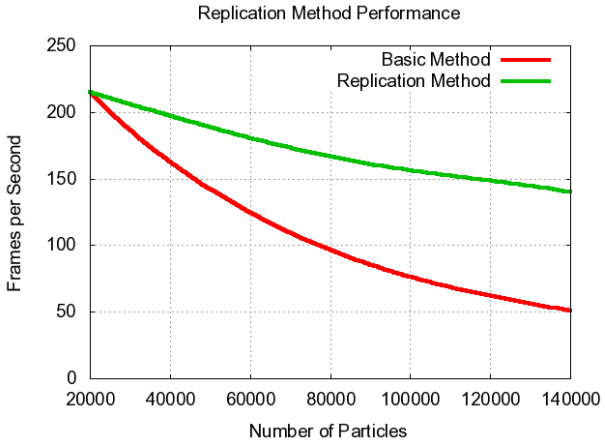


Figure 3.13: Comparison of the performance obtained with and without replicating particles (patterns) on the GPU.

crease the performance of the rain simulation. We present in Table 3.5 the frame rate obtained for the different rain intensities proposed above. In order to compare the performance of the different rain frameworks we render the amount of particles indicated by the users in the study (see Table 3.2). It can be seen how our approach can render the rain scenes with a frame-rate increased in nearly 35 %.

Model	Our Model	Our Pattern-Based Model
FPS	443	579

Table 3.5: Comparison of the performance obtained without and with the pattern-based approach.

We have also considered interesting to test the difference in performance that can be obtained when creating the raindrops on the GPU. In Figure 3.13 we depict in green the frame rate obtained when rendering different amounts of raindrops without creating multiple quads on the GPU. In the same Figure, we show in red the *fps* obtained when applying our new method. We translate 20,000 particles but in the GPU we replicate them by 2, 3 and so on in order to obtain as many raindrops as desired. We can see how the replication method outperforms drastically the previous method. In this study we have considered that

the seed raindrop only replicates into 6 more raindrops, rendering at most 140,000 particles. The reason for selecting this number of replicas is because it fits more properly with the objectives of our multiresolution proposal.

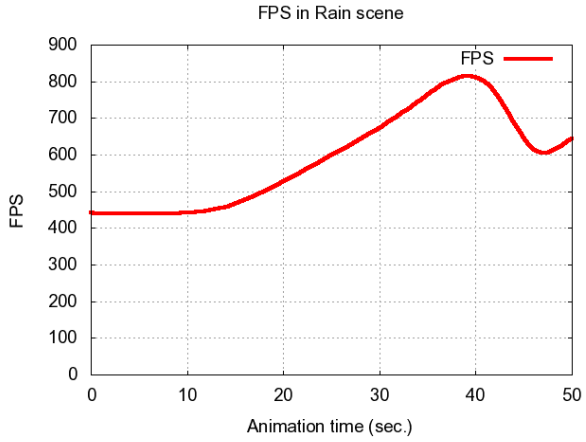
3.6.5. Scene Test

For testing the performance of our model inside a final application, we have analyzed the frame-rate and the vertices rendered along a route which starts at a point in the middle of the rain area and finishes outside this area. Figure 3.14 provides the results obtained during 50 seconds. Along this route, the number of vertices decreases once the user enters the attenuation zone (second 10) until it leaves completely the rain area (second 40). As a consequence, the frame rate fluctuates in accordance to the final amount of geometry that is visualized. After nearly 40 seconds, the user is outside the rain area. Then, the user turns back on itself to look at the rain area, activating the *far rain* approach and increasing the number of rendered particles (second 45). It is important to comment that, when rendering a similar number of particles, the *close rain* is faster than the *far rain*. This is due to the fact that the particles rendered with the *far rain* approach are considerably bigger, which entails generating more pixels.

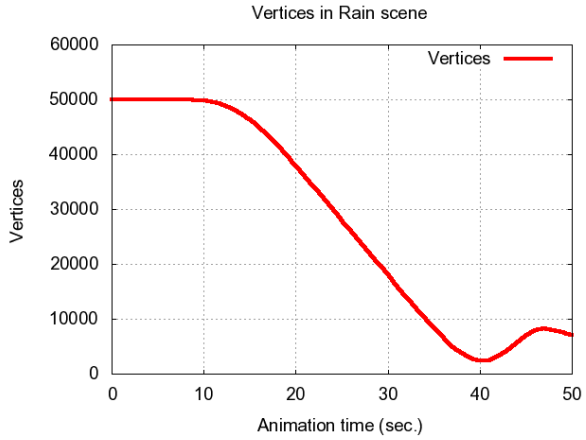
3.7. Conclusions

In this chapter we have introduced a set of techniques to create and efficiently visualize scenarios with realistic rain. The technique presented here provides different approaches, depending on the relation between the user location and the rain area location. In addition, we have included multiresolution techniques which are applied directly and uniquely on the GPU.

The results we obtained improve on those achieved by previous solutions. The conducted user study proved that our presented solution is capable of offering similar rain intensity sensations with much less particles. This reduction in particles directly involves an increase of the obtained performance. Moreover, the use of level-of-detail techniques that are fully integrated into the *Geometry Shader* strongly decreases the temporal cost of the rain system. The selected shape of the rain container prevents us from relocating the whole container continuously,



(a) FPS obtained along the time of the test.



(b) Vertices rendered along the time of the test.

Figure 3.14: Testing our model inside a rain scene during 50 seconds.

which was one of the main drawbacks of previous solutions [12, 30]. As a consequence, we improve on an important limitation of these systems, the fast camera movements. Previous particle systems were not suitable for game environments where the user makes fast continuous movements.

Finally, we have shown that our approach is capable of handling large rain areas consisting of hundreds of thousands of particles in real time. This work has proved that it is possible to incorporate a multiresolution

scheme into particle systems in order to simulate rain areas. We must note that the high performance interactivity of our framework would only be achievable by using GPU *shaders*. Nevertheless, our system is not only focused on increasing the performance. We have introduced a new framework which can deal with different kinds of rain and also with different interactions between the user and the rain simulation system.

The framework we have proposed could be easily improved with features like raindrop splashes, light interaction, collisions and further effects that have been presented in previous works. Furthermore, we are interested in applying level-of-detail techniques to these effects in order to obtain a more realistic simulation while maintaining a low computational cost.

CHAPTER 4

Rain Simulation on Dynamic Scenes

Rain is a complex phenomenon and its simulation is usually very costly. In the previous chapter we introduced a new framework for rendering rain which enhanced the performance by means of new level-of-detail techniques. In this chapter we propose an improved simulation system which, aside from the rainfall simulation, offers a subsystem for the detection and handling of the collisions of particles against the scenario, which allows for the simulation of splashes at the same time. This proposal works completely on the graphics card. The solution suggested is based on the utilization of particle systems to simulate rain. This system obtains a very high performance thanks to the hardware programming capabilities of CUDA.

4.1. Introduction

The use of an efficient and realistic rain simulation system considerably increases the realism of outdoor scenes. Most of the systems proposed for real-time rain simulation are based on the use of particle systems [26, 28, 30, 2]. Nevertheless, the use of particle systems to represent rain may have significant limitations due to the cost involved in handling

Figure 4.1: Intense rain environment.

the great quantity of particles which is necessary to offer environments of intense rain with realistic appearance. Virtual environments where this kind of solutions are integrated have thereby serious performance problems, as rain simulation takes up a very important part of the computer resources.

The concept of GPGPU (General Purpose Computing on Graphics Processing Units) was presented as a low-cost alternative to parallel processing systems [138]. Hence, instead of employing a large number of computers, several authors resorted to graphics processors to make mathematical calculations. CUDA (Compute Unified Device Architecture) is a recent technology created by NVIDIA with the objective of making the most of the great processing capacity of the current graphics cards to solve problems with a high computational load [139].

In this work we propose a rain simulation system which, as it happened in the previous chapter, is managed and updated only in the graphics hardware. The solution suggested is based on the utilization of particle systems to simulate rain and includes the management of rain under variable wind conditions. This system obtains a very high

performance through the use of CUDA, since it considerably frees the CPU from an enormous load of operations. CUDA offers a very flexible framework which allows us to include the implementation of an approach to detect and handle the collisions of rain particles against the scenario and also to generate splashes. Developing a system of this kind by means of the traditional graphics pipeline would be very complex. Figure 4.1 shows an image of the suggested rain simulation system in an intense rain environment.

The framework we propose in this chapter offers three main advantages with regard to previous methods based on the GPU:

- It is not necessary to make two passes of the graphics hardware to make the calculations that update the positions of particles. All the calculations can be made in one single pass, remaining the second one only in charge of visualizing the geometry obtained.
- It does not introduce the overcost of using a graphics library for a task independent of graphics, as happens with the calculations made for the particle simulation. Moreover, CUDA provides methods for the direct visualization of the results, given the interoperability of the memory in the graphics card between OpenGL and CUDA.
- It is possible to add more natural effects related to the precipitation, such as the dynamic generation of geometry to simulate splashes.

This chapter is structured as follows. Section 4.2 presents the main features of the framework of the solution proposed. Section 4.3 describes in detail the rain model presented, including the subsystem in charge of the calculation of collisions. Later, section 4.4 presents the results obtained, both in performance and in visual quality. Finally section 4.5 summarizes the contributions of our work and it outlines future lines of work.

4.2. General Features

The use of graphics hardware can allow us to accelerate rain visualization considerably. Thus, several solutions have appeared which use *shaders*, not only to offer final effects, but also to process efficiently the

particle systems. It is important to mention that in our rain simulation system we will represent the raindrops by using *quads* as the rendering primitive, as they are more accurate to the streak-shaped perceived raindrop.

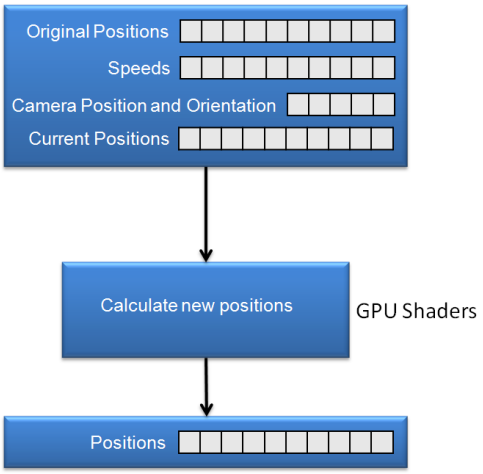
Traditionally, rain simulation models in graphics hardware needed two passes of the graphics pipeline to obtain the geometry. In the solution proposed in the previous chapter we also used this approach based on two rendering passes. Figure 4.2 shows a diagram that summarizes these two traditional passes:

- The first pass moves the particles following the falling movement of the drop with a certain velocity, which can be defined as constant for all the particles or be stored in a vector for each element of the particle system. Furthermore, in this pass the system repositions those particles that have come out of the scenario to continue with the simulation without losing the sensation of a continuously falling rain.

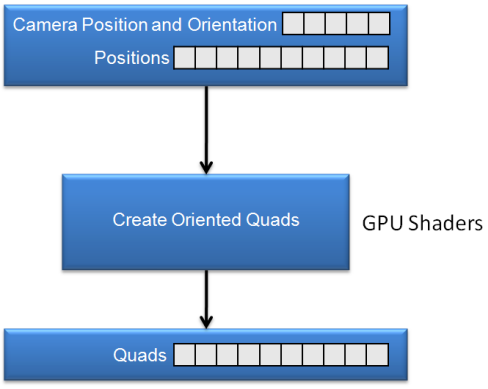
- In the second pass, the system creates the four points forming the quad used to simulate the drops. The calculation of these new positions takes into account the current position of the camera to orient the *quads* towards the user as impostors. At this point, some models choose to texturize the *quad* and others assign a final color to it.

Another aspect of the rain systems that is important to comment is the fact that our framework follows the movements of the user. Thus, the simulation system only represents raindrops in those areas which are closer to the viewer. Then, if the user modifies its position, the whole rain simulation system must react properly and locate the raindrops according to the new camera position. This is the reason why the *camera position and orientation* are necessary in the first rendering pass.

By means of the framework that we propose it is possible to optimize the use of the graphics card. Obviously, it will still be necessary to make two passes. The first one will be made with CUDA and the second one with OpenGL, as it is not possible to make the update and the visualization of the particle system in only one GPU pass, since CUDA is not a graphics library that allows the visualization of the updated geometry.



(a) First GPU pass.



(b) Second GPU pass.

Figure 4.2: Rendering passes necessary to obtain the geometry to visualize.

4.2.1. CUDA Usage

The problems presented by graphic applications do not generally require a very high processing capacity, contrary to the problems that have been traditionally solved through GPGPU techniques. Even though, the use of CUDA in a graphics application is very interesting because of the way in which information can be accessed and shared between the different processes running in parallel. In CUDA, a thread has its own processor, variables (registers), processor states, etc. A block of threads is represented as a virtual multiprocessor. The blocks can be run in any order in a concurrent or sequential way. The memory is shared between the threads; in a similar way, there is shared memory between the blocks of a kernel. This means that it is possible to work on the same data in different threads and in different blocks, besides being able to do it in an asynchronous way with the CPU. This supposes a great advantage with respect to the previous architecture. Moreover, it also offers promising possibilities for the development of new solutions or the improvement of previous graphics methods.

4.3. Our Rain Model

In our work we propose the use of CUDA to optimize the update of the particles in the rain simulation system. Thus, we will be able to write at the same time the vector that stores the updated positions and the vector that stores the generated *quads*. The use of this framework allows us, on the one hand, to reduce the time consumed in the calculation of the new positions and, on the other, to incorporate new functionalities that, by using traditional rendering techniques, would be very complex to obtain.

The main advantage offered by our solution is that all the calculations are made in only one CUDA *kernel*. The way CUDA works with memory positions allows us to save information in several buffers at the same time, to employ read/write buffers and to decide in which buffer position the generated information should be saved.

As we have commented in previous chapters, the retinal persistence of the human eye (or the motion blur of the shutter of a camera) makes the raindrops to be perceived as strikes. It is possible to find some works that have tried to capture in a texture the appearance of these strikes. The complexity resulting from incorporating these textures into the real-

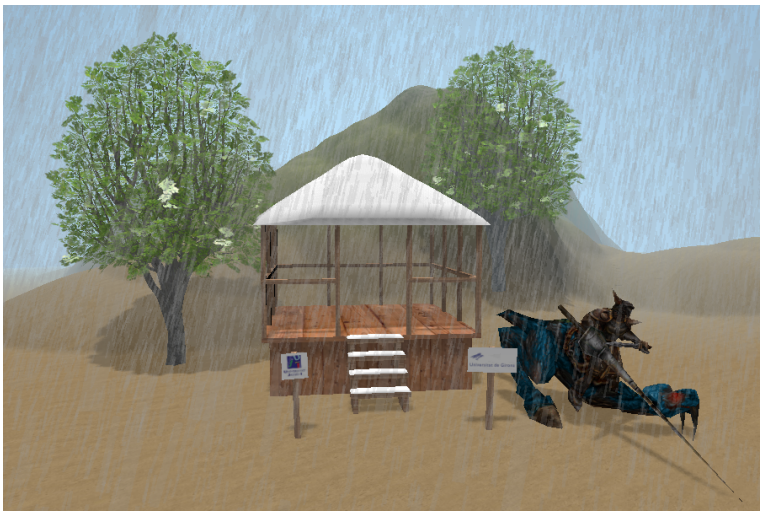


Figure 4.3: Intense rain environment with wind.

time rain simulation increases the cost of the system without offering a considerably better visual aspect in contrast to the systems that do not use textures. Therefore, in our system we have chosen to assign the *quads* a color with a certain level of transparency to give them their final appearance. In the results section we will see how, despite not using textures to simulate drops, our system is capable of offering a rain simulation which offers a sensation of realism to the final user.

4.3.1. Wind

Our proposed solution offers the possibility of adding new features to the rain simulation. In addition to the simulation of splashes, we have also considered interesting to include the effect of the wind on the falling raindrops. The wind simulation implies altering the falling vector of the raindrops, so that they do not fall completely vertical. Moreover, it is also necessary to perform a small modification of the algorithm in charge of generating the oriented *quads*. This way, it will be necessary to consider the falling direction of the particles so that the *quad* is also oriented following this direction.

Figure 4.3 presents a snapshot of our rain simulation system where wind has been added. It can be seen how the shape of the *quads* has

been adapted to the new falling direction of the raindrops.

4.3.2. Collisions

One of the aims of our suggested system is to offer better capabilities. In this sense, we consider that the calculation of collisions is a very important requirement in a complete solution to the problem of precipitation simulation. The calculation of collisions offers advantages in two senses. On the one hand, it allows us to optimize the update of the positions of the particles as it avoids moving those drops that have already collided, as happens with mountains or buildings, and that are not visible anymore. On the other hand, the main contribution refers to realism, since the generation of splashes is a fundamental part in rain simulation.

The splashing of a raindrop is a complex phenomenon where, in general, the distribution of droplets of a splash depends on many different features of the elements involved. In this sense, Garg et al. [140] measured the water drops on different surfaces to offer a physically realistic simulation. This high degree of physical simulation is usually beyond real-time applications.

Some of the models that we reviewed in the previous work chapter include the detection of collisions. The solution proposed by Feng et al. [27] presents a method for detecting collisions and it includes a subsystem to simulate the splashes of drops after collision. Later, the method introduced in [30] suggests the simulation of the splashes by repeating the application of displacement textures on the scenario. This method is not very precise, although the results obtained are visually satisfactory. More recently, Rousseau et al. [29] described a method for collision detection based on capturing the normal and height of each point in the scenario with a camera above the scene. Thus, by using this texture in rendering time they will be able to detect collisions and calculate the rebound direction with the normals.

In our framework we propose a simple system, although it would be possible to use more complex methods with no technical difficulty. Similarly to Rousseau et al. [29], the method we have applied consists in making a capture of the heightmap of the scene we are visualizing, although we have not considered necessary to control the rebound directions. This capture is made for every image we create, so that the system is capable of adapting the calculation of the collisions to the state of the

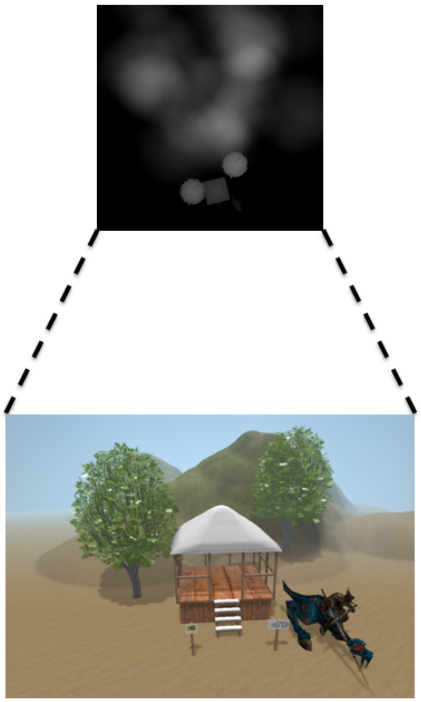


Figure 4.4: Diagram of the capture of the heightmap.

scene in each moment, considering the movements of the characters and other objects in the environment. This heightmap is stored as a texture inside the graphics card memory. Figure 4.4 shows the process of obtaining the heightmap from an image of the scene obtained from a camera positioned at such height and distance that allows us to observe the whole scene.

The solution proposed will detect if a raindrop collides with the scenario by consulting this texture. In the solution suggested, when detecting a collision we decided to create a *quad* in that position and texturize it with an image that simulates water dispersion after the collision. In a similar way, it would be possible to create several particles that simulate the drops created from the collided drop, which could follow a new trajectory. But in the system presented we have chosen to apply a more simple method, as it is capable of offering a realistic sensation with a lower cost.

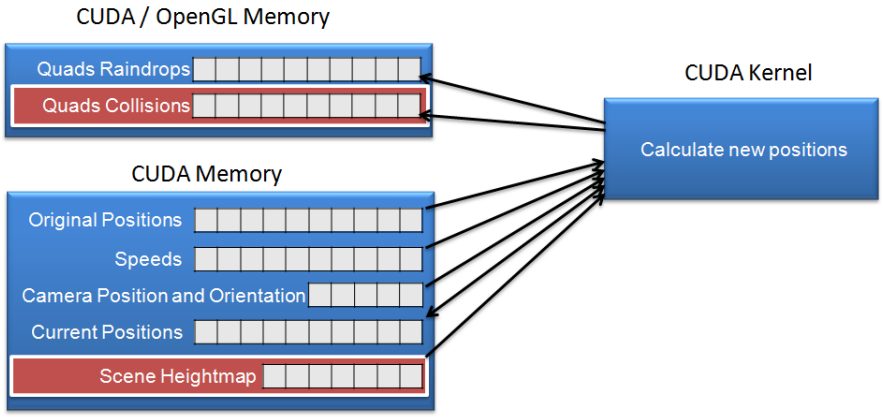


Figure 4.5: Diagram of memory usage.

4.3.3. Data structures and algorithms

Figure 4.5 summarizes the data distribution in the graphics card memory that we propose for the use of CUDA. This figure includes, shaded in red, the structures necessary to make the calculation of collisions and the simulation of splashes. In this Figure we can see how it will be necessary to store in memory the original position of the drops and their speed. Moreover, we need a buffer to store the current position of the particles, which will be read and written at the same time to update the raindrops location. From a slightly different perspective, we will also need two buffers to store the generated *quads* for the raindrops and the collisions, which are generated considering the orientation and position of the camera. These buffers will be shared with OpenGL to visualize the obtained geometry. It is important to comment that if the particle suffers a collision it is necessary to reposition it above the field of vision of the user in order to continue with the rain simulation.

Algorithm 4 offers the whole rain simulation process in a detailed way through its description using pseudocode. The CUDA *kernel* proposed includes the calculation of collisions, so that every time we displace a particle, we compare its resulting position with the height stored in the previously commented texture. Thus, if we detect a collision, we will reposition the particles in the upper area of the scene considering the current camera position, so that the rain system follows the movements

of the user. Moreover, we will add a particle to our particle subsystem in charge of collisions. This subsystem creates a *quad* in the locations where collisions happen.

Algorithm 4 Pseudo-code of the position update process.

```
// Scene capture
float* heightMap;
heightMap=renderTexture();

// Process of raindrops update
float *particlePositions, *particleQuads, *particleSpeeds;
float *particleOriginalPositions;
int numCollisions=0;
float *quadsCollisions, *cameraPosition, *cameraOrientation;

for particle = 0 to numParticles do
  // Scroll the particle
  particlePositions[particle] -=particleSpeeds[particle];
  if collision(particlePositions[particle],heightMap) then
    // Create collision
    numCollisions++;
    quadsCollisions[numCollisions] = calculateQuad(
                          particlePositions[particle],cameraPosition);

    // Reposition particle
    particlePositions[particle]= particleOriginalPositions[particle];
    // Follow camera movement
    translate(particlePositions[particula],cameraPosition);
    rotate(particlePositions[particle],cameraOrientation);
  end if
  particleQuads[particle] = calculateQuad(particlePositions[particle],
                                          cameraPosition);
end for
```

The collisions subsystem needs some specific considerations. The amount of collisions that are generated at each frame is variable and we therefore need some techniques to manage this variable amount of geometry. One of the features of CUDA that allows the efficient calculation of collisions are atomic operations, specifically the instruction

atomicInc() [139]. This instruction reads the contents of a register in memory, it adds one integer and it writes the result in the same memory direction. Atomic operations guarantee that there are no interferences between execution threads, so that a specific memory position is not accessed by other thread until the operation is finished. In our case, we can maintain a single collision counter which is also used to store the position where the collision happens in a correct and subsequent way inside the collision vector. Thus, as output of the *kernel* we will obtain a vector of *quads* located in the positions where collisions have happened and oriented again towards the observer. This vector will have a fixed size in memory, but its content will have a variable size, depending on the characteristics of the scene.

4.4. Results

In this section we will analyze the solution proposed from several perspectives, since, apart from the performance, we are interested in analyzing the visual quality of the obtained simulation. The scene used in the various tests is the one presented in Figure 4.1. All the tests have been conducted on Windows XP Professional in a computer with an Intel QuadCore Q6600 a 2.4GHz processor, 4 GB RAM and a GeForce GTX280 graphics card with 1 GB RAM.

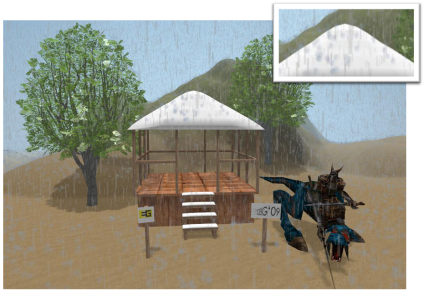
4.4.1. Visual Quality

One of the main objectives of the work proposed is to offer a rain simulation method which is realistic to the human eye. In this sense, Figure 4.1 presented an intense rain environment where we simulated 2 million particles in real time. To this simulation we have to add the calculation of collisions for each raindrop and the splash generation when necessary, which is simulated with a texturized *quad* with a color similar to the raindrop.

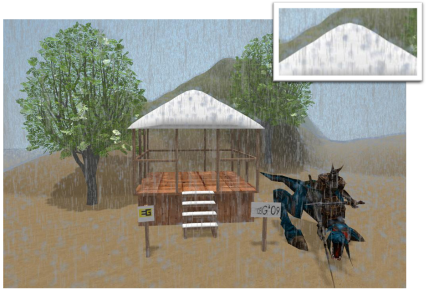
Figure 4.6 shows several screen captures of our rain simulation method. Each of them represents a different number of particles, which will give way to a different perception of rain intensity.

(a) Rain sensations with 512,000 particles.

(b) Rain sensations with 1 million particles.



(c) Rain sensations with 2 million particles.



(d) Rain sensations with 4 million particles.

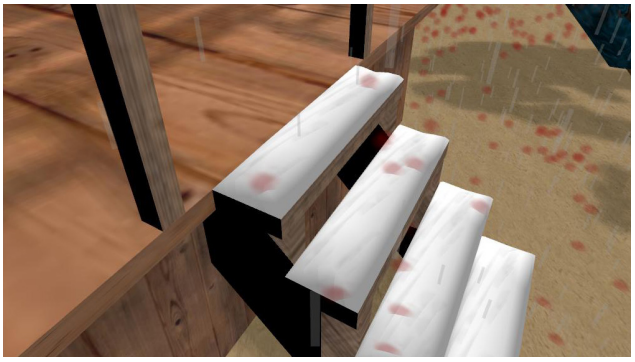
Figure 4.6: Rain environments with different intensities.

4.4.2. Collisions and Splashes

As we have already mentioned, the inclusion of splashes adds realism to the final scene. The images introduced until now allowed us to observe how the splashes are distributed in the scene. In this section we want to especially remark that the selected method for the calculation of collisions allows us to easily detect the surfaces on which the drops collide.

Figure 4.7 shows two images detailing the same scene. In this case, splashes have been coloured in red so that it is easier to distinguish them from the environment. The first one allows us to see how the system adapts to the geometry of the scene, since it can be observed how the splashes take place on the steps of the hut or on the warrior riding the dragon. In addition, Figure 4.7(b) shows a perspective of the steps, seeing how splashes are never generated inside the hut, since the

(a) Scene perspective.



(b) Detail of the steps of the hut.

Figure 4.7: Rain simulation examples with collision detection.

drops collide on its roof. Finally, it is important to note that our method captures the heightmap in every frame rendered, so that the calculation of collisions is continuously adapted to the movements of characters and objects in the scenario.

4.4.3. Performance

The utilization of our approach for the simulation of the raindrops offers significant advantages regarding performance. Table 4.1 presents a study of the performance (fps) of our system when simulating various amounts of rain. In this case, the intensities chosen are the ones presented in Figure 4.6. Furthermore, we also include separately the cost of the calculation of collisions, in order to be able to analyze how much the simulation of splashes affects the performance of our proposal. It is

Number of Raindrops	CUDA Version	CUDA Version without collisions	Shaders Version without collisions
500,000	76.43	83.14	127.22
1,000,000	50.34	56.12	76.92
2,000,000	30.12	35.71	43.29
4,000,000	15.62	19.02	21.78

Table 4.1: Performance comparison for rain environments with different intensities (fps).

important to mention that in this table we have also included the results of the *shaders version* presented in the previous chapter.

From the results shown in Table 4.1 we can conclude that the simulation system proposed is capable of simulating a high quantity of drops in real time. Regarding the calculation and the visualization of the splashes, it increases the processing time by 15 % on average. It is important to remark that, the more particles the rain system handles, the more percentage of time is necessary to dedicate to collisions. For this reason, for the simulation which works with 500,000 particles the system slows down by 8 %, whereas if we want to visualize 4 million particles we will have to consider that the performance is affected in a 21 %.

In this section we also want to compare our proposal with the methods based on the use of the traditional graphics pipeline [30]. In this case, the method with which we will compare ourselves makes no calculation of collisions. We can say that the performance obtained with shaders increases by 30 % on average. In this case, the difference between both methods decreases as we visualize more particles, so if we want to simulate 4 million particles the performance difference is of 14 %.

Otherwise, it is important to comment that the use that the rain system makes of the CPU in one case and in the other is very different. Figure 4.8 shows the percentage of use of the CPU for 30 seconds of the simulation. We can see that the rain simulation based on *shaders* takes 57 % of the CPU time, whereas with CUDA this occupation is reduced to 26 %. Function calls by CUDA are asynchronous, allowing the simulation to be more fluent.

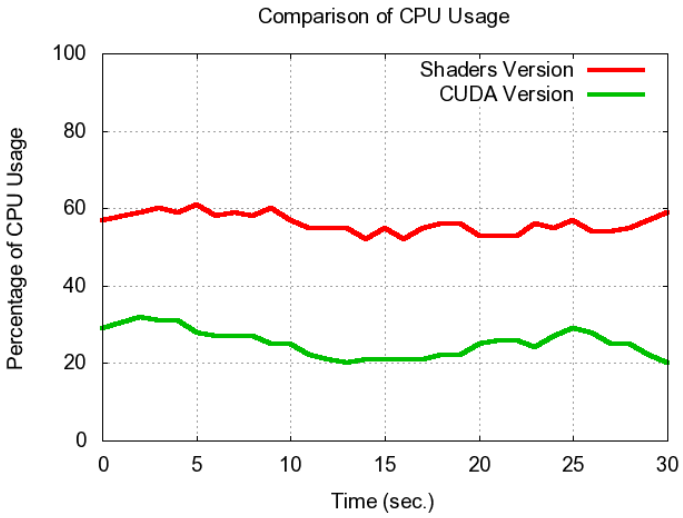


Figure 4.8: Percentage of CPU use during 30 seconds of animation.

4.5. Conclusions

In this chapter we have presented a set of techniques to efficiently visualize scenarios with rain of different intensities and falling directions with a realistic appearance. The improvement of the simulation has been possible thanks to the use of new techniques for GPU programming, as their flexibility allows us to offer advanced techniques such as the collisions detection. Thus, the quality obtained in the different tests is due to the possibility of including a high quantity of particles and splashes. Moreover, CUDA enables the graphics application to reduce by half the use of CPU time if compared with the shaders version, which allows the application to dedicate that time to make other calculations.

The solution we have described offers great advantages and encourages us to continue improving the simulation. In this sense, we are currently interested in maximizing the use of collisions to offer better effects, as calculating the accumulation of rain on the ground due to the collided particles. Moreover, studying how the rain affects the properties of the different materials would be very interesting to increase the perceived realism. Furthermore, we consider interesting to study the alteration of the precipitation and the splashes in situations of wind, so that the

system solves correctly situations where drops do not fall in a totally vertical way.

CHAPTER 5

Ocean Simulation

Modeling and rendering realistic ocean scenes has been thoroughly investigated for many years. Its appearance has been studied and it is possible to find very detailed simulations where a high degree of realism is achieved. Nevertheless, among the solutions for ocean rendering, real-time management of the huge heightmaps that are necessary for rendering an ocean scene is still not solved. We propose a new technique for tessellating the ocean surface on GPU. This technique is capable of offering view-dependent approximations of the mesh while maintaining *coherence* among the extracted approximations, exploiting the graphics hardware capabilities in order to reuse the already calculated data. We also include in our simulation reflection on the generated sea and animation of ocean waves by means of GPU-based Perlin noise.

5.1. Introduction

Describing ocean waves is a very complicated challenge, as ocean is composed of different elements that form a very complex system. It is possible to find very complex mathematical models that simulate the behaviour of ocean waves, some of them based on the direct observation of the sea [53, 40]. Nevertheless, the game industry usually prefers to lose physical realism due to the high demand for real-time simulation.

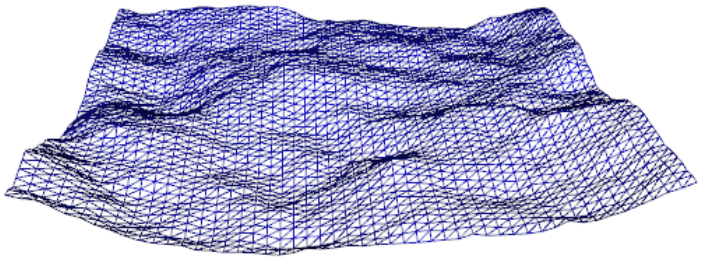


Figure 5.1: An ocean can be seen as an animated heightmap.

Thus, real-time applications usually used simplified models that still offer physical realism but guarantee high frame rates.

Many decades ago, Turner Whitted was among the firsts to attempt the simulation of water. In his simulation the ripples were created by bump mapping the surface, perturbing the surface normal according to a single sinusoidal function [141] and ray tracing was used to obtain reflections.

The approaches to simulate ocean that were based on bump mapping techniques [141, 57] cannot interact realistically with other surfaces or cast shadows on them. To avoid these shortcomings, Nelson Max [142] used a heightfield to render wave surfaces for his film *Carla's Island*.

This approach is still followed and, therefore, an ocean is usually simulated as an unbounded water surface that is represented in the gaming environment as a heightmap. Other complex phenomena, such as foam, spray or splashes are usually modeled and rendered using particle systems [143, 144, 145]. In these simulations, the height of each vertex is modified in real time to offer the sensation of wave movement. It can be seen as the use of a displacement map to alter the position of each vertex [146]. Figure 5.1 depicts a snapshot of a mesh simulating ocean movement in a given instant of the animation.

Some authors have criticised the use of heightfields to model waves [42], as those data structures store only one height value for any given (x,y) pair. In this sense, it can be possible to have situations where there is more than one height value for each position, as it happens

Figure 5.2: Picture of a breaking wave where a heightmap would not be adequate.

with breaking waves (see Figure 5.2). In our case we will consider a sea where these choppy waves are not expected. For this reason, the use of a squared heightmap is still adequate in our proposed ocean simulation. Kryachko [147] proposed the use of a static radial grid instead of a squared one. On that account, by centring this radial grid at the camera position we can have more points in those areas that are closer to the viewer. Although this solution is capable of offering more detail in the areas closer to the viewer, it poses severe restrictions and does not assure a high performance.

Managing the geometry of the mesh representing the ocean still poses a limitation in simulating ocean. A technique that several authors propose is the tessellation of the surface. In the field of computer graphics, tessellation techniques are often used to divide a surface in a set of polygons. Thus, we can tessellate a polygon and convert it into a set of triangles or we can tessellate a curved surface. These approaches are typically used to amplify coarse geometry.

This chapter introduces a new adaptive tessellation scheme which works completely on GPU. The main feature of the framework that we are presenting is the possibility of refining or coarsening the mesh while maintaining *coherence*. By coherence we refer to the re-use of information

between changes in the level-of-detail. In such way, the latest extracted approximation is used in the next step, optimizing the tessellation process and improving the performance.

The main characteristics of the proposed ocean model are:

- The ocean surface is refined on the GPU by means of a new view-dependent tessellation algorithm.
- Geometry shader capabilities are exploited to reuse extracted approximations.
- Wave movements are simulated with Perlin noise [57] on GPU.

This chapter has the following structure. As our objective is mainly to present a new tessellation scheme, Section 5.2 thoroughly describes the tessellation technique that we present. Section 5.3 describes the ocean simulation process, which describes how the tessellation technique is combined with other processes to offer a realistic impression on GPU. Section 5.4 offers some results on the presented technique. Lastly, Section 5.5 includes some conclusions on the developed techniques and outlines the future work.

5.2. Our GPU-based Tessellation Scheme

As we have mentioned in the previous section, tessellation is a widely used technique in ocean simulation. Adaptive approaches are much more interesting, as they can refine those areas that need more detail, while those areas which are less interesting can be coarsened. Nevertheless, there has not been developed any ocean tessellation technique which considers the use of the latest features of graphics hardware. It is our objective to exploit these features in order to improve the performance of previous adaptive tessellation techniques.

Many of the tessellation algorithms presented previously in the state of the art chapter modify the detail of the triangles following some criterion applied to the triangle. The calculations involved could consider the distance of the triangle to the camera or its position in the screen. Nevertheless, applying the level-of-detail criterion in a triangle basis implies a limitation for adaptive solutions. As an example, Figure 5.3 presents a tessellation step where, applying some criterion, the bottom-left triangle has to be refined while its neighbour does not have to. Later, if we

Figure 5.3: Example of crack after a tessellation step.

apply some modifications to the position of the vertices we can obtain a noticeable *crack*, a hole in the mesh. These cracks are due to the introduction of *T-vertices* in the input mesh. T-vertices appear commonly in tessellation algorithms when a vertex is positioned on the edge of another triangle [134], resulting in two edge junctions making a *T-shape*. An example of this problem can be seen in Figure 5.3, where the vertex added in the tessellation step represents a T-vertex.

In order to avoid crack problems, some authors have proposed to apply the refinement criterion only to the edges of the triangle. Therefore, if an edge needed refinement, then both triangles sharing the edge would act accordingly. In this case, following the example presented before in Figure 5.3, both adjacent triangles would perform the appropriate tessellation tasks to create new triangles with the same new vertices, assuring that no crack is generated.

5.2.1. Tessellation patterns

Guided by the idea of developing an edge-based tessellation algorithm that avoids cracks, Ulrich described some edge-based patterns for tessellating triangles [18]. Figure 5.4 presents, on the left side, an initial rectangular triangle where its hypotenuse and catheti (more commonly known as *legs*) are depicted anti-clockwise as H, C_1 and C_2 respectively. Next, the seven tessellation patterns introduced by Ulrich are presented (labeled from 1 to 7), where the edges of the original triangle that need refinement are depicted in red. As we stated before, the work that we are proposing is based on using a refinement criterion based on the edges and not on the complete triangle. As a result, each pattern shows the tessellation that would be necessary depending on the combination of edges that need refinement. For example, in the bottom-left case the hy-

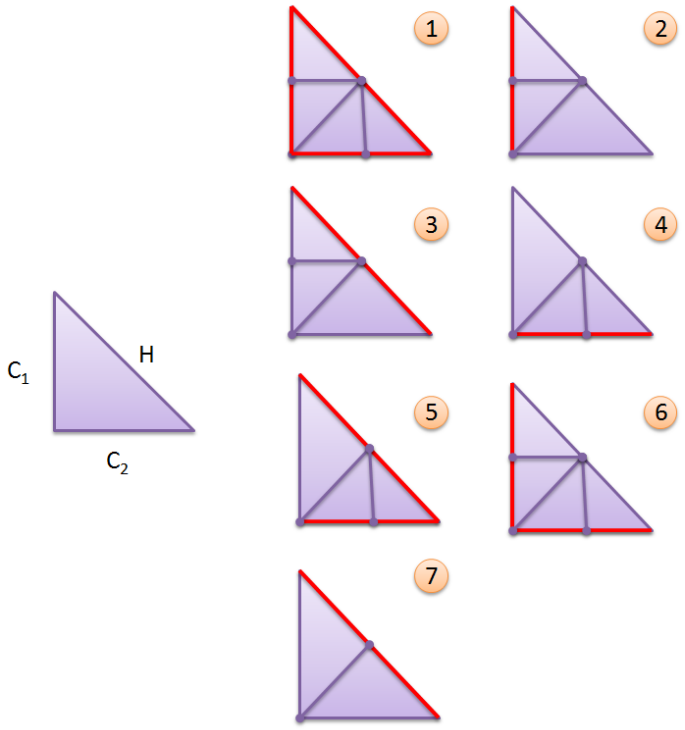


Figure 5.4: Tessellation patterns from Ulrich [18]. The red colour indicates the edges that need refinement.

potenuse needed refinement and a new vertex has been added to create two new triangles. The main problem with Ulrich's proposal was that it still presented cracks, as his patterns were based on the use of T-vertices. Thus, in some of the patterns presented it can be seen how there are vertices created on an edge that does not need refinement, resulting in a geometry similar to that presented in Figure 5.3.

In order to avoid this limitation, the work presented in [19] modified the previous patterns that included T-vertices. In Figure 5.5, three of Ulrich's patterns (number 2, 4 and 6) have been surrounded by a red dotted line. These patterns were the ones that had T-vertices. On the right side, surrounded by a green dotted line, the three modifications introduced in [19] are shown, where it can be seen how no T-vertex is added.

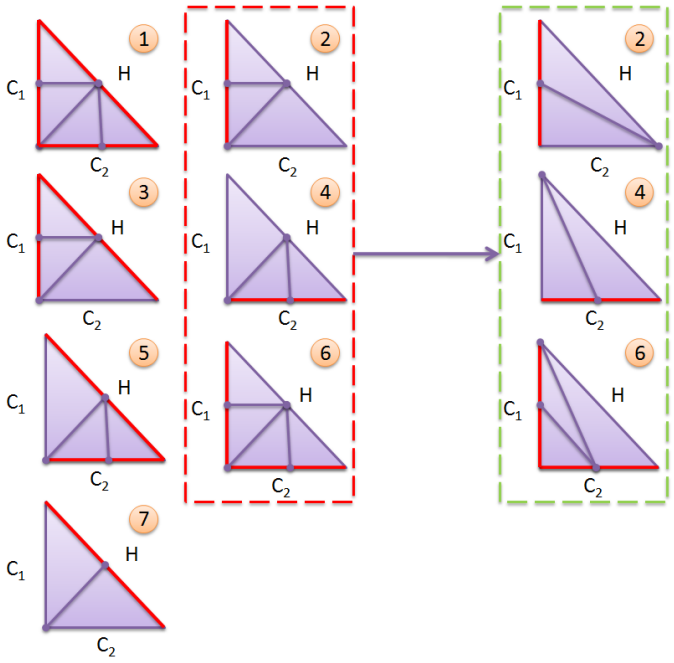


Figure 5.5: Tessellation patterns with T-vertices (in red) [18] and without T-vertices (in green) [19].

In our case, we will use the patterns presented in [19], as they can assure that no T-vertices are introduced and that the continuity of the mesh is maintained. These patterns produce more elongated triangles if compared with Ulrich's patterns, which could result in more complex lighting or texturizing. Nevertheless, our algorithm will calculate these values from the vertices of its parent triangle. In this sense, it is also worth mentioning that we will not store precomputed patterns on GPU memory as other solutions do [69]. We just code in the *Geometry Shader* the seven cases that we follow so that the coordinates of the new vertices can be calculated from the coordinates of the two vertices that define the edge.

5.2.2. Our proposed algorithm

As we are processing the mesh in a *Geometry Shader*, each triangle is processed separately. For this reason, we have developed a technique

which is capable of altering the geometry of two triangles that share an edge without any communication among them. With this approach we will be able to exploit the parallelism of graphics hardware.

Refining the mesh

When *refining* the mesh, the algorithm checks each edge to see whether they need refinement. Depending on the combination of edges that need more detail, the algorithm selects a pattern for tessellating the input triangle (see Figure 5.5). Each of these generated triangles stores the spatial coordinates, the texture information and any other information needed for rendering. Moreover, it is necessary to output for each new triangle two pieces of information that enable our solution:

- A number indicating the id of the triangle.
- A number coding the tessellation patterns that have been applied, keeping this information in $patternInfo$.

The need of storing these two values is due to the fact that we must know how a particular triangle was created in order to know how we should modify it when swapping to a lower level of detail. On the one hand, the id value will uniquely identify each of the generated triangles. Thus, this value will be used as a means to create the hierarchy of triangles, as it enables us to calculate easily the parent triangle of any given triangle. This id number is given following the formula:

$$id = id \cdot maxOutput + originalTris + childType \tag{5.1}$$

Although this formula is quite simple, the different constants involved need further explanation. The $maxOutput$ value is understood as the maximum number of triangles that can be output from a parent triangle using the available patterns. The patterns presented in Figure 5 involve outputting a maximum number of 4 new triangles and, as a consequence, in our case $maxOutput$ is a value equal to 4. The $originalTris$ constant refers to the number of initially existing triangles on the source mesh, which depends on the input mesh the application uses. Finally, $childType$ is a value used to differentiate between the triangles output from a parent triangle. As the patterns used output a maximum number of 4 new triangles, in our case the $childType$ is a value ranging from 0

to 3. Let's say we are applying pattern 1, which outputs 4 new triangles. When calculating the *id* of each of these new triangles the value *childType* will be one of the numbers in the range [0, 3] to assure that each *id* is different. Thus, the *childType* value is necessary to differentiate between the triangles that belong to the same parent, which is compulsory for the correct coarsement of the mesh as we will see in the following section.

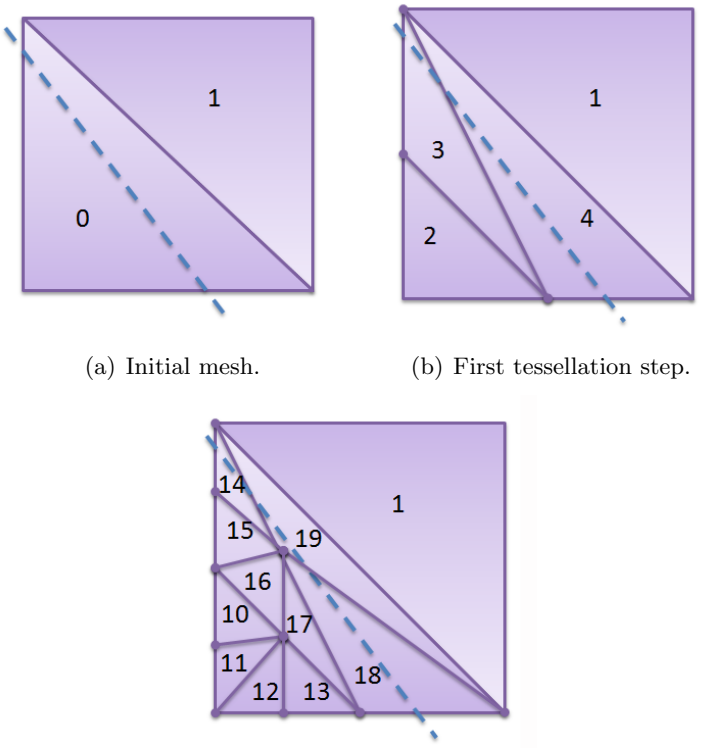
On the other hand, and following with the two elements that have to be output for each triangle, the $patternInfo$ number is used to store all the patterns that have been applied to refine a triangle. This value is obtained following Equation 5.2, which has been specifically prepared to code the different patterns applied to a triangle in one single value. In this equation $latestPattern$ refers to the type of the latest pattern, the one that we have used to create this triangle. The value $numberOfPatterns$ refers to the number of available patterns that we can apply in our tessellation algorithm. In our case we use the 7 patterns presented in Figure 5.5 and, as a consequence, $numberOfPatterns$ should be equal to 7.

$$patternInfo = patternInfo \cdot numberOfPatterns + latestPattern \tag{5.2}$$

This $patternInfo$ value will be the same for all the triangles belonging to the same parent. This piece of information is important due to the fact that we must know how a particular triangle was created in order to know how we should modify it when swapping to a lower level of detail.

The *id* and $patternInfo$ values presented above are the elements that enable our algorithm to recover less detailed approximations without having to start again from the coarsest approximation. It is important to underline that this is one of the main features of the method we are proposing. Finally, it is worth mentioning that, initially, the *id* values of the original triangles are given sequentially (starting from *id* 0) and all the $patternInfo$ values are equal to 0.

To clarify the process and the equations that we have introduced, Figure 5.6 presents an example of how the tessellation process works. On Figure 5.6(a) we present the initial mesh composed of two triangles, intially labeled with *id*s 0 and 1. The dotted line in blue represents the plane that we will use to define which area of the mesh needs refine-



(a) Initial mesh.

(b) First tessellation step.

(c) Second tessellation step.

Figure 5.6: Tessellation example with the *id* value of each triangle.

ments, where the whole area below this line will be considered to require more detail. Each of the two initial triangles go through the extraction process of the algorithm that we are presenting. In the specific case of the triangle number 1, the algorithm detects that none of its edges needs refinement and, as a consequence, no change will be made. Nevertheless, the algorithm detects that triangle with *id* 0 needs refinement because the center points of the two legs of the triangle are below the dotted line. Then, we choose from the patterns the one that reflects this combination and we apply it, so that we obtain the three new triangles shown in Figure 5.6(b). It can be seen how the *id* values of the new triangles are calculated following the formula 5.1, assuring that no repeated *id* is given. Following with the refinement process, the next tessellation step shows that different patterns have been applied to triangles 2, 3 and 4,

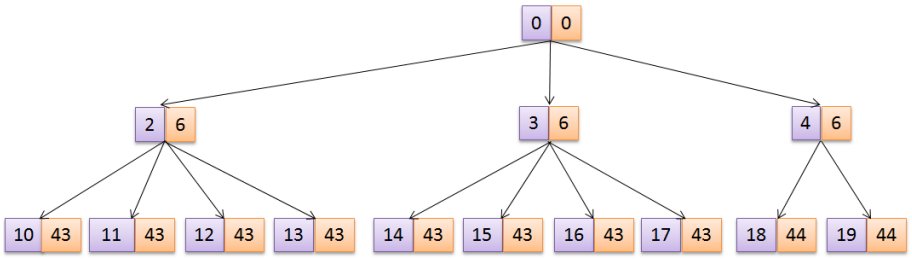


Figure 5.7: Tessellation tree. Each node presents the *id* and the *patternInfo* values of each triangle.

as they represent different types of tessellation.

Figure 5.7 presents the tree of triangles that can be obtained in the example that we are presenting. For each node we present, on the left and in blue color, the *id* of the triangles and on the right and in red color the *patternInfo* value of each triangle. Both sets of values are calculated following the formulas presented in Equations 5.1 and 5.2. It is important to mention that the number of children of each node will depend on the pattern applied, as they output a different number of triangles. By using the previously proposed tessellation patterns in Figure 5.5, we can refine one triangle and obtain 2, 3 or 4 new triangles.

Coarsening the mesh

A different process should be applied when diminishing the detail of the mesh. The *id* and the *patternInfo* values of the triangles have been precisely given in order to simplify the coarsening process. Thus, following with the example given in the previous subsection, if we wanted to reduce the detail and return to the state shown in Figure 5.6(b), each of the triangles located under the dotted line would execute the same coarsening process.

When tessellating a triangle, for example with the pattern that is used when the hypotenuse and both legs are refined (see pattern 1 in Figure 5.5), four triangles are output. Nevertheless, only one of these triangles will be needed when diminishing the detail. Three of them will be discarded and the other one will be modified to recreate the geometry of the parent triangle. Therefore, the *childId* value of each triangle enables differentiating child triangles to if the triangle can be

discarded or if it is the triangle in charge of retrieving the geometry of the parent triangle. If it is the latter case, the triangle will calculate the spatial coordinates of its parent triangle.

The first step would be to find out whether the triangle that we are processing can be discarded or if it is the triangle in charge of retrieving the geometry of the parent triangle. The *childType* used when calculating the *id* of each triangle is necessary for this particular differentiation, as in those cases where this value is equal to 0, the algorithm assumes this triangle is in charge of recovering the geometry of the parent triangle. The *childType* can be retrieved by using Equation 5.3.

$$childType = mod((id - originalTris), maxOutput) \tag{5.3}$$

$$id = (id - originalTris)/maxOutput \tag{5.4}$$

In order to retrieve the spatial coordinates of the parent triangle we must know which pattern was applied to create the existing triangle. This is due to the fact that for each pattern we will perform different calculations for retrieving the three vertices of the parent triangle. In this situation the $patternInfo$ value helps us to know which pattern was applied, as the latest pattern can be obtained with the next equation:

$$ilatestPattern = mod(patternInfo, numberOfPatterns) \tag{5.5}$$

Once we know which pattern was applied, we calculate the position of the vertices and we output the new geometry with the new *id* value obtained in Equation 5.4 and the new $patternInfo$ value obtained with Equation 5.6. The way we calculated this value assures that we will be able to continue coarsening the mesh or refining it without any problem.

$$patternInfo = patternInfo/numberOfPatterns \tag{5.6}$$

Following with the example presented in Figure 5.6, if we wanted to coarsen the geometry each triangle would go through a coarsening process. Let us suppose that we are processing the triangle with *id* 10. If we calculate its *childType* we obtain a 0 value, indicating that triangle 10 is the one that must become the parent triangle, whose *id* can be retrieved with Equation 5.4. In this case, the *latestPattern* would indicate that pattern 1 was applied and we would calculate the spatial coordinates

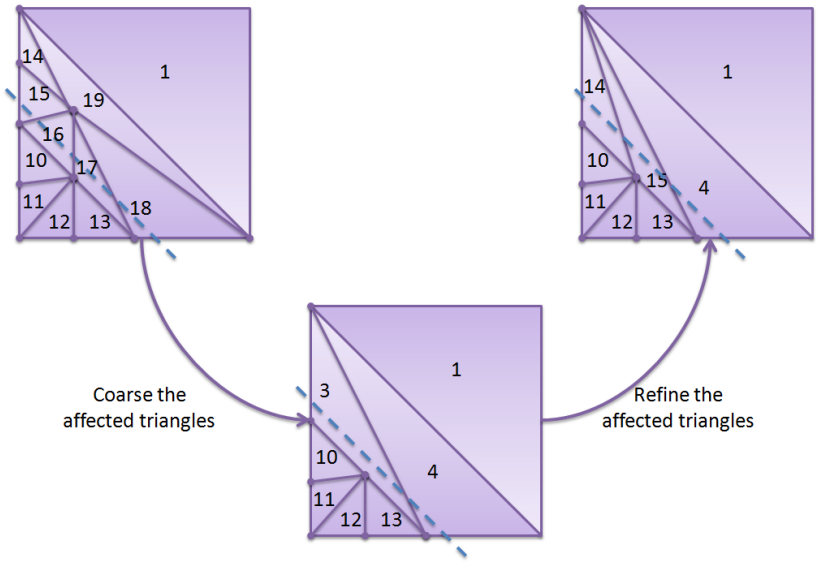


Figure 5.8: Example of re-tessellation when the refinement criterion is changed.

of the parent triangle accordingly. Nevertheless, triangles 11, 12 and 13 have a *childType* different from 0 and would be discarded. Once again we would like to remember that all these operations have been coded in the shaders, so that we are able to know which operations to perform depending on the type of pattern we applied.

5.2.3. Camera movement

In the example presented above (see Figure 5.6), we have described how the tessellation process works but we have considered that the location of the plane splitting the area where the mesh is and is not tessellated remains unaltered. Nevertheless, in a real case this line keeps moving all the time as it is usually related to the camera position.

Figure 5.8 is based on the second tessellation step shown in Figure 5.6(c). It presents a case where the position of the dotted line is modified, altering the criterion used to decide which triangles we have to refine. In these cases, a slightly different process is applied to correct the appropriate triangles. This algorithm checks each triangle to

see whether, with the new criterion, their parent triangle would need a different tessellation. For example, triangles with *id* number 1 or number 10 would not require any change as their parent would experience the same tessellation (or refinement) with both positions of the dotted line. Nevertheless, the parent of the triangle with *id* value equal to 18 had two legs below the dotted line and now both of them are above this line. In this case, the algorithm would coarse the triangle and refine it again. Similarly, triangle 19 (sibling of triangle 18) would also detect that its parent would have been affected by the critetion change.

Following with triangles 18 and 19, we would coarse them eliminating one of them while the other becomes triangle 4 again. Then, we apply the adequate pattern to refine again the triangle. Similarly, triangles 14, 15, 16 and 17 are affected and three of them are eliminated while the remaining one becomes triangle 3 and is refined again, creating new triangles with *id*s 14 and 15. These coarsening and refining processes are performed following the methods presented above.

It is important to underline that both processes (coarsening and refining again) are executed at the same time, so that we can coarse the triangle in more than one level of detail and refine it again. The *id* values have been calculated so that we can know at any point if the triangle we are processing was useful in any of the previous levels of detail. Moreover, although this process seems tedious, only a small portion of the triangles in the mesh will go through this process.

5.3. Ocean Simulation

The previous technique is capable of modifying the detail of the mesh in real time to offer a fast rendering of the ocean. It is important to mention that the geometry obtained in this pass will be output and stored in GPU memory, so that it can be used in the following frames for further tessellations or for maintaining the current tessellation if necessary.

Nevertheless, in addition to the geometry management of the mesh simulating the ocean, we must perform other tasks in order to obtain a visually-satisfying ocean simulation. In this section we will briefly describe the different techniques used to enhance the realism of the simulation.

Perlin noise

In this sense, one of the first features that we must consider is the algorithm applied to simulate the ocean waves. In the state of the art we have presented many techniques that have been developed to model ocean surfaces. Among them, we have selected the Perlin noise [57] as it is capable of producing smooth waves. Thus, in our specific case, Perlin noise can be used to make some very impressive looking sea effects, recreating sea variations by simply adding up noisy functions at a range of different scales.

Perlin noise has been a commonly used technique of computer graphics since 1985. This algorithm for noise generation has been used in many applications and its implementation in current hardware is available [148, 149]. Moreover, Perlin noise is faster than other methods and it is easily ported to GPU shaders, in contrast to other algorithms like the FFT ones (explained in Chapter 2) which are slower and difficult to code.

In our implementation, the shader in charge of updating the animation of the ocean will calculate the appropriate height according to the position of the vertex within the mesh and to the time of the animation. For enabling the Perlin noise calculation on GPU, we initially upload a texture containing some noise information that is necessary for the real-time noise update.

Rendering enhancements

In addition to animating the waves, we must also consider other interactions of the ocean, such as refraction, reflection, foam, etc. Reflection can be obtained by applying environmental mapping on GPU. This technique consists in using 3D texture coordinates to access a cubemap storing the precalculated reflex. The Fresnel term is commonly computed by calculating, for each pixel, the dot product between the normal and the eye vector. This value is used to access a one-dimensional texture which stores different reflections for different fresnel values [150]. In the simulation that we prepared, we only considered reflection and fresnel factor, although we could apply any of the techniques that are available in the literature. These two effects are simple to code and sufficient to offer a realistic impression.

5.4. Results

In this section we will study the performance of our tessellation method by analyzing the visual quality obtained as well as the calculation time of the extracted approximation. Our scheme was programmed with GLSL and C++ on a Windows Vista Operating System. The tests were carried out on a Pentium D 2.8 GHz. with 2 GB. RAM and an nVidia GeForce 8800 GT graphics card.

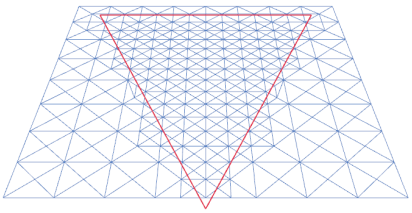
We also proposed the integration of the whole ocean simulation in an application which controls the tessellation and the final rendering quality.

5.4.1. Visual Results

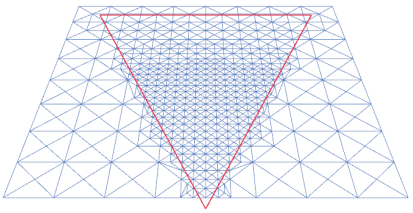
First, we offer some visual results of the tessellation algorithm that we have described. Figures 5.9 and 5.13 present a mesh in wireframe where different tessellations have been applied.

Figure 5.13 presents a tessellation case where an initial mesh composed of four triangles (on top) is refined according to the distance to the camera. In the more refined meshes that this figure presents it is possible to see how the tessellation is not uniform, as those areas of the mesh which are closer to the observer are more tessellated than those that are farther. Moreover, Figure 5.13(h) presents the most detailed tessellation with Perlin noise applied to simulate ocean waves.

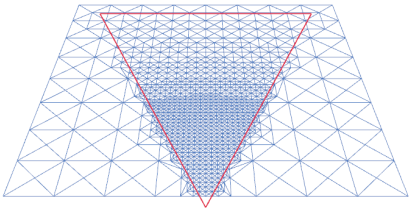
We can find another tessellation example in Figure 5.9. In this case, we have considered that a fictitious frustum (depicted in red) has been located on the mesh to guide the tessellation process which, again, also considers the distance to the camera. In this case, the initial mesh is composed of 256 triangles. In this example four tessellation steps are presented. It is important to mention that some areas of the mesh that are outside the frustum are also tessellated in order to avoid T-vertices, as we explained when describing the features of our proposal. These figures show how the tessellation process is capable of increasing the detail of an input mesh without introducing cracks or other artifacts. Moreover, the last image of this Figure presents the tessellated surface animated with Perlin noise, where only those triangles within the frustum are animated.



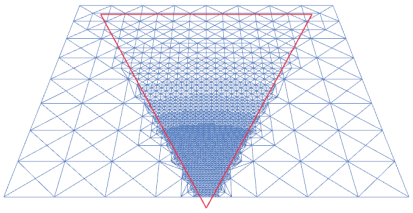
(a) First tessellation step (598 triangles).



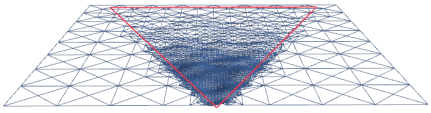
(b) Second tessellation step (1,264 triangles).



(c) Third tessellation step (2,546 triangles).



(d) Fourth tessellation step (5,002 triangles).



(e) Fourth tessellation step with surface animation.

Figure 5.9: Sample tessellation guided by a simulated frustum (in red).

5.4.2. Performance

In order to test the performance of our proposed tessellation technique, we have conducted some tests where an initial mesh composed of 4 triangles is tessellated. The detail of the input mesh is initially increased and later coarsened following a smooth transition obtained from a smooth trajectory of the camera to get closer to the mesh.

Figure 5.10 presents the time needed for tessellating, animating and rendering the different tessellation steps depicted in Figure 5.13, where

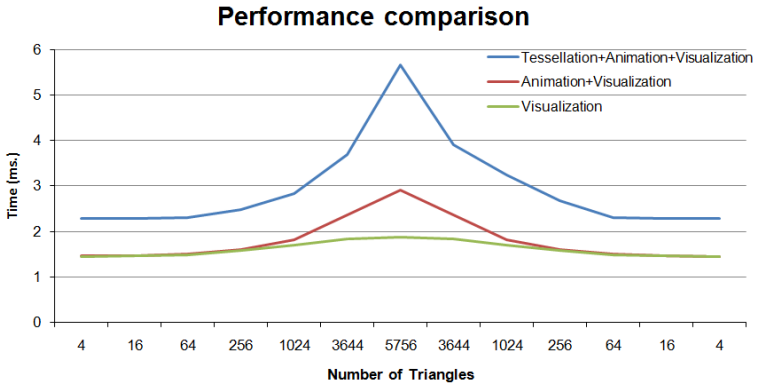


Figure 5.10: Performance obtained using a distance criterion (Figure 5.13).

the tessellation depends on the distance to the camera. Table 5.1 presents the results obtained in this test, helping us to show how the noise calculations for animation involve increasing the rendering time in 10 % while the tessellation supposes an increase of 60 %.

For offering further tessellation experiments, Figure 5.11 presents the results of a similar test where all the geometry is tessellated at the same time, without any specific criterion. In this case, the obtained geometry will be composed of 2^n triangles, where n is the tessellation step. In this case, we can observe how the cost of the tessellation is exponential, offering very high temporal costs when outputting 65,536 triangles. Again, the results are depicted in Table 5.2 to help us analyze the way this tessellation algorithm works. In this test, the noise calculation doubles the rendering time while the tessellation step only involves a 60 % increase. It is worth mentioning that, in our simulation, we will never include so many triangles as only those areas that need detail will be tessellated. Nevertheless, we considered interesting to show how the temporal cost of the algorithm can be affected by the quantity of output triangles.

5.4.3. Coherence exploitation

An important improvement of the proposed approach is the possibility of exploiting coherence among the extracted tessellations. Table 5.3 presents the temporal results of a scenario similar to that presented

Number of Triangles	Visualization	Visualization + Animation	Visualization + Animation + Tessellation
4	1.45	1.46	2.29
16	1.46	1.46	2.29
64	1.48	1.50	2.30
256	1.58	1.61	2.48
1,024	1.71	1.82	2.83
3,644	1.83	2.36	3.70
5,756	1.88	2.91	5.67
3,644	1.83	2.36	3.90
1,024	1.71	1.82	3.24
256	1.58	1.61	2.67
64	1.48	1.50	2.30
16	1.46	1.46	2.29
4	1.45	1.45	2.29

Table 5.1: Comparison of time (in milliseconds) required for visualizing, animating and tessellating the input mesh using a distance criterion (see Figure 5.13).

in Figure 5.13, where the distance to the camera is used to guide the tessellation. These temporal costs include visualization, animation and tessellation of the input mesh. The column on the right offers the results without coherence maintenance, which nearly double the cost of our coherence-based algorithm. These results show that we can offer better performance us our tessellation scheme can exploit coherence among extracted tessellation, in contrast to previous solution which had to start again from the input mesh.

5.4.4. Ocean simulation

Finally, Figure 5.12 presents a snapshot of the ocean simulation we have proposed. As we mentioned before, the system includes reflections and the fresnel factor to give realism to the scene.

Number of Triangles	Visualization	Visualization + Animation	Visualization + Animation + Tessellation
4	1.45	1.46	2.29
16	1.46	1.46	2.29
64	1.48	1.50	2.44
256	1.58	1.61	2.44
1,024	1.71	1.82	2.83
4,096	1.80	2.49	3.85
16,384	2.08	5.50	8.68
65,536	2.71	20.20	30.84
16,384	2.08	5.50	10.68
4,096	1.80	2.49	4.15
1,024	1.71	1.82	2.83
256	1.58	1.61	2.44
64	1.48	1.50	2.44
16	1.46	1.46	2.29
4	1.45	1.45	2.29

Table 5.2: Comparison of time (in milliseconds) required for visualizing, animating and tessellating if completely tessellating the mesh.

Number of Triangles	Coherence Exploitation	No Coherence Exploitation
16	2.49	2.49
64	2.49	3.63
256	2.52	4.89
1,024	2.84	6.46
3,644	4.16	10.05
5,756	6.47	14.04
3,644	5.16	10.05
1,024	3.73	6.46
256	2.69	4.89
64	2.31	3.63
16	2.27	249

Table 5.3: Performance comparison (visualization, animation and tessellation) with and without exploiting coherence (in milliseconds).

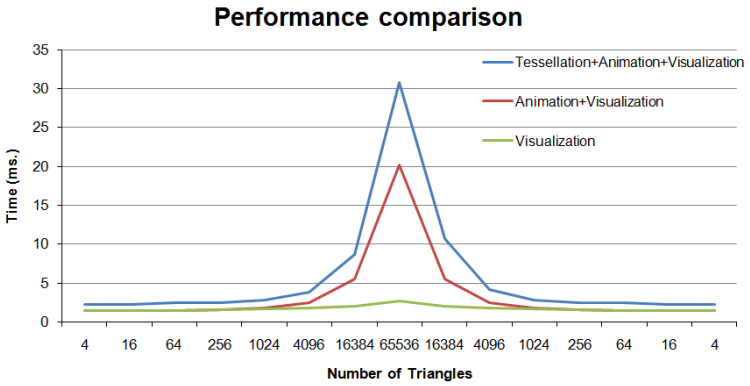


Figure 5.11: Performance obtained when completely tessellating the mesh.

5.5. Conclusions

Ocean simulation has been addressed by many researchers to offer realistic visualization, although some of them were not aimed at real time animation. In this sense, we have reviewed many related papers in order to choose the main features that affect the realism of the surface of the sea, although only some of them have proposed improvements on the management of the underlying geometry.

We have presented a method for simulating ocean in real-time. The presented approach is based on the use of a new adaptive tessellation scheme which exploits coherence among extracted approximations. Accordingly, by storing some information, we are capable of reusing the latest extracted mesh when refining and coarsening the surface. In this framework, the final simulation includes reflection and considers the fresnel term to offer realistic approximations, although our main objective was the development of a new tessellation scheme.

For future work we are focused on the inclusion of more effects like refraction or the interaction of objects with the surface. In this sense, we must perform further research to combine the use of fractal noise with the interactions of objects with the ocean.

From a different perspective, it is worth mentioning that this tessellation algorithm could be also applied to terrain rendering. As a consequence, it is our interest to analyze the possibilities offered by a GPU-

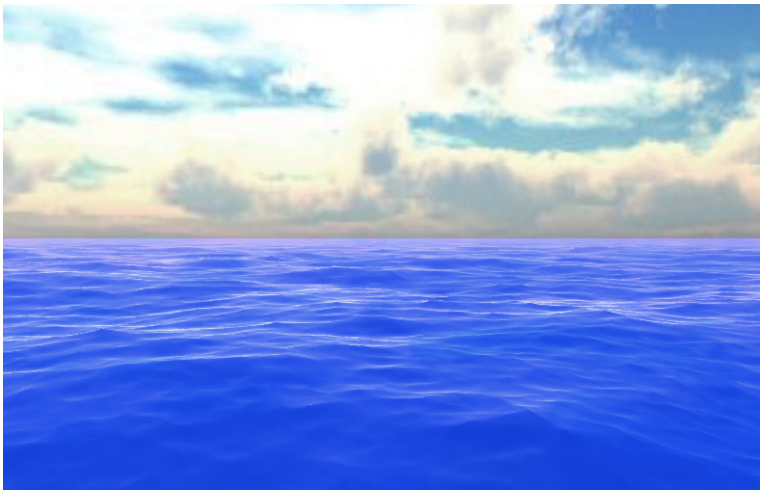
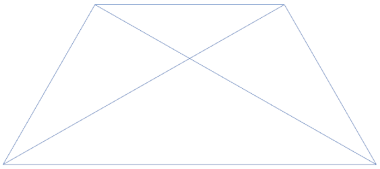
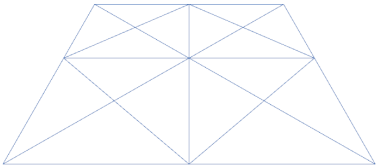


Figure 5.12: Simulation integrated into the final application.

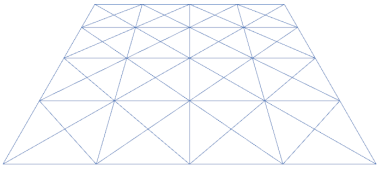
based tessellation technique in terrain visualization.



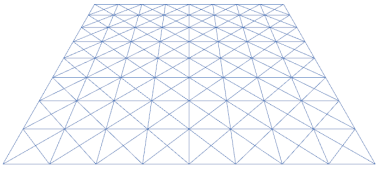
(a) Initial mesh (4 triangles).



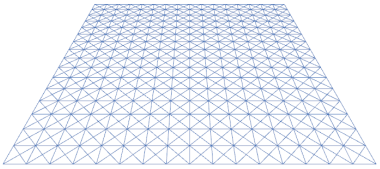
(b) First tessellation step (16 triangles).



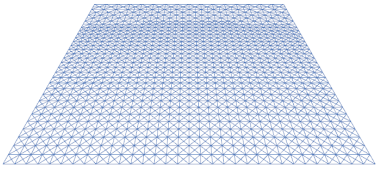
(c) Second tessellation step (64 triangles).



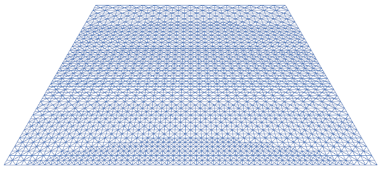
(d) Third tessellation step (256 triangles).



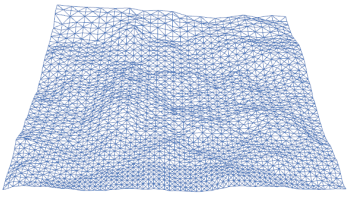
(e) Fourth tessellation step (1,024 triangles).



(f) Fifth tessellation step (3,644 triangles).



(g) Sixth tessellation step (5,756 triangles).



(h) Sixth tessellation step with animated surface.

Figure 5.13: Sample tessellation following a distance criterion.

CHAPTER 6

Automatic Terrain Generation

Terrain is a very important element in these outdoor scenarios. Users from the scientific community or even the gaming environment are currently requiring a higher level of customization. In this sense, the objective of the work presented in this chapter is to provide the final user with an easy-to-use terrain generation application. We propose a sketching solution which, combined with a simple terrain algorithm, is capable of offering a realistic but synthetic terrain. The application is composed of two windows, which offer 2D and 3D representations of the terrain respectively. These windows are sufficient for providing the user with an interactive feedback about the terrain that is being designed. Moreover, our approach offers the possibility of using an image as a guide for sketching the desired shape. Our framework offers algorithms for both creating and modifying terrain features, thus improving the final results with more realism and greater customization for the user.

6.1. Introduction

In recent years, computer graphics has undergone an intense evolution as new graphics hardware offers a final image quality that was unimaginable just a few years ago. As a result, interactive graphics applications, such as computer games or virtual reality environments, now

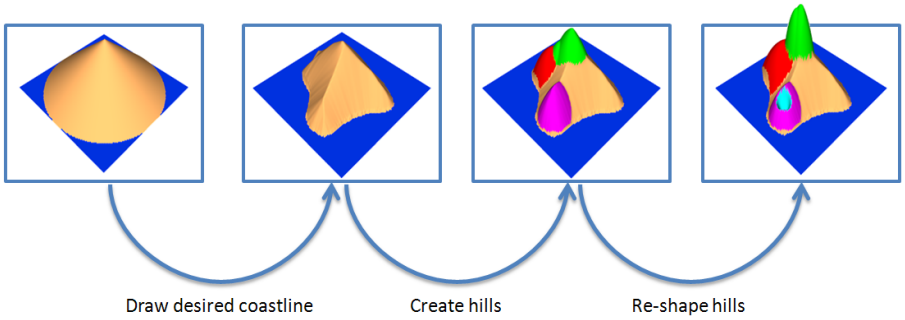


Figure 6.1: Designing a sample island terrain in three steps.

include more complex scenes offering very detailed environments. Terrain is therefore a key element in many applications that can lessen the sense of realism if it is not addressed correctly.

Terrain generation is a research area which has been active for many decades and growing power of modern computers has made them capable of producing increasingly more realistic scenarios. Synthetic terrain generation is a process which creates elevation values throughout a two dimensional grid. The need for highly realistic scenarios often involves developing algorithms that can generate more detailed terrains with more user control over the final terrain that is created. Different terrain generation techniques that are capable of offering very realistic artificial terrains have been reported in the literature. Nevertheless, not many applications provide enough user control and those that do are often too difficult to control.

Sketching is a tool that is well suited to the design of architectural elements and it provides the user with a considerable amount of control over the created elements. It is possible to find recent surveys on sketching [109, 110], where the reader can account for the great amount of work that has been developed in this area. However, less work has focused on sketching the underlying terrain or extracting it from a photograph. Thus, buildings are often considered as the *foreground* and are taken into account properly, whereas terrain is seen as *background* which is often ignored.

Some sketching terrain approaches have been developed in recent years [130, 128, 129]. Only the work developed by Gain et al. [129] proposed a complete sketching solution with a sufficient amount of user

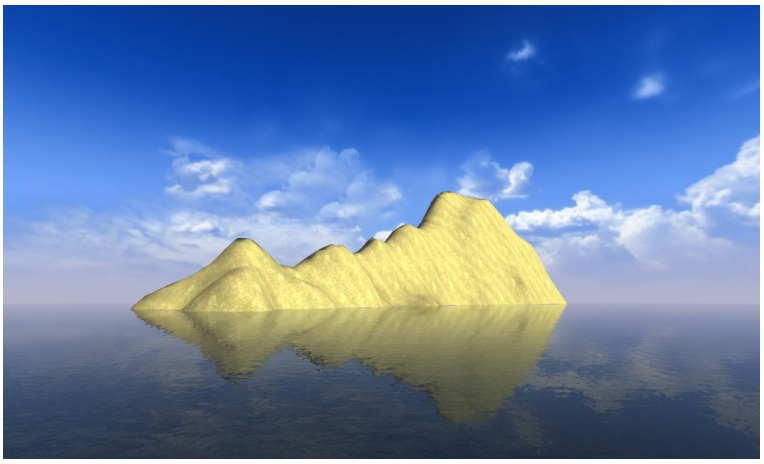


Figure 6.2: Example of a terrain obtained with our framework and imported into the Torque Game Engine.

control. Nevertheless, their solution still presented a complex interaction from the user where multiple views of the terrain are needed in order to obtain the desired terrain. In this sense, managing intersecting mountains can become difficult as the user must decide if a new mountain must be in front or behind an existing one while drawing the silhouette. The problem of boundary constraints can be overcome if we restrict ourselves to sketching islands. The boundary constraints for islands are simple: an island has a coastline, that is, a continuous bounding curve. Everything within the coastline can be assumed to be above sea level, and everything outside the coastline can be assumed to be below sea level and invisible.

Our aim consists in developing convenient and simple ways to create computer models of terrain. Our goal is similar to the idea given in [151], where the authors show that relatively simple algorithms can provide non-professional users with fast, successful results. In this sense, Figure 6.1 shows how we can create a terrain with three simple steps using the method we are presenting. Thus, we can sketch the island, create some hills and also re-locate and re-shape them as desired.

It is worth mentioning that, more precisely, in this chapter we address the problem of creating computer models of islands for use in computer games. Figure 6.2 presents a terrain generated with our solution and

imported into a game engine for its use in a 3D scenario. In the following sections we will describe how this terrain is generated automatically from the user's input.

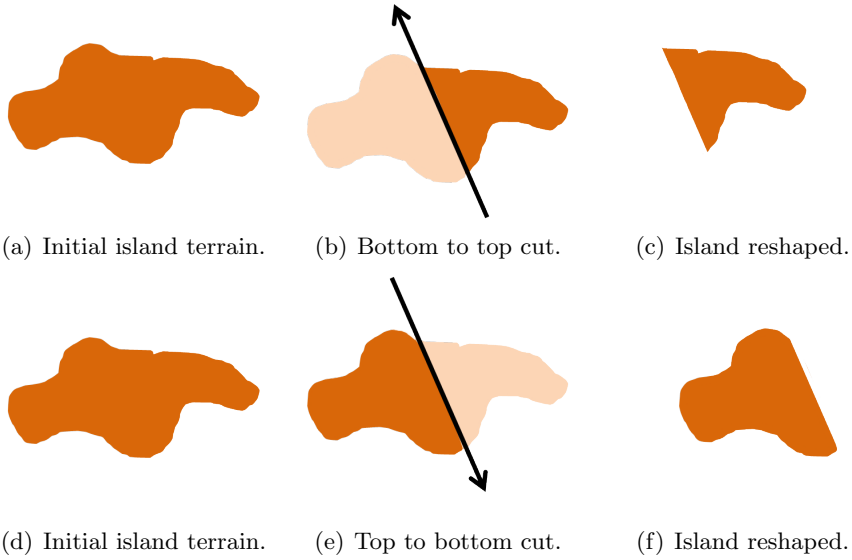
In this work we describe a terrain creation algorithm for island terrains based on *heightmaps*, which are regularly-spaced two-dimensional grids of height coordinates. These grids can be later processed by a modeling software or game engine to obtain the 3D surface of the desired terrain. The elevation of the terrain is automatically calculated from the coastline sketched by the user, who can also create hills and apply filters in order to achieve more realistic and irregular terrain. The perturbations created by the hills and the filters are both created by means of a simple yet efficient terrain generation algorithm. Once again, it is important to mention that our aim is to provide the user with more control over the terrain that is generated.

Furthermore, we also consider the integration of our algorithm into a sketching application. This application combines a 2D representation window and a 3D displaying window in order to simplify the drawing process. First, the user creates a silhouette of the island in the 2D window and then, the user will be able to modify the terrain appearance from the 2D and the 3D windows. The terrain thus obtained will be output as a heightmap that may then be imported into a game engine.

This work is organized as follows. Section 6.2 describes our terrain generation algorithm. After that, Section 6.3 analyzes our sketching application and presents the user interface and the possible operations that can be used to create the terrain. In Section 6.4 we describe a detailed implementation of the data structures and processes of our framework. Later, Section 6.5 depicts our results and it also discusses a usability test performed among different potential users. Finally, Section 6.6 presents our conclusions and gives some ideas for future work.

6.2. Our Terrain Generation Algorithm

The method that we present for generating islands is based on the use of heightmaps and it allows users to define and modify the coastline of the island, using an image as guide if desired. Furthermore, they can also create and reshape any number of hills, which will interact with each other and with the existing terrain. Finally, we offer the possibility of applying filters to give the final terrain a more realistic appearance.



(a) Initial island terrain. (b) Bottom to top cut. (c) Island reshaped.

(d) Initial island terrain. (e) Top to bottom cut. (f) Island reshaped.

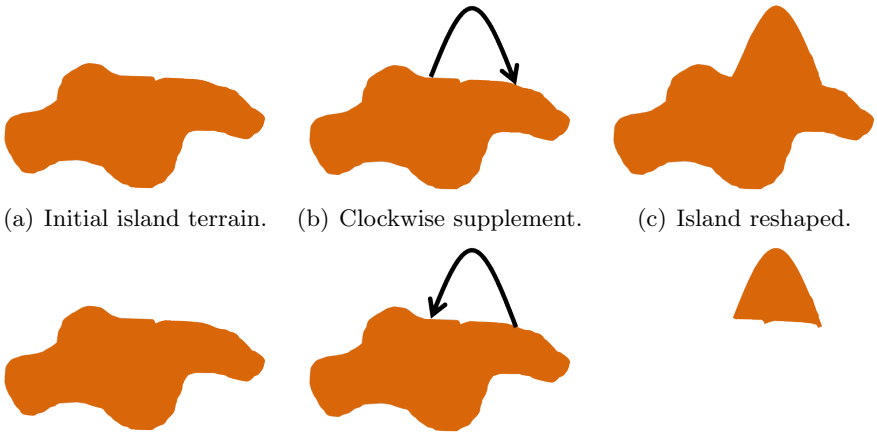
Figure 6.3: Cutting and reshaping the island.

6.2.1. Reshaping the Coastline

The user can draw the silhouette of the island freely, but it will be necessary to delineate a continuous curve by sketching a closed shape, as shown in the left image of Figure 6.3. The shape of the coastline can be changed by redrawing, starting and ending at points near the existing coastline and drawing a continuous curve in any direction between those two points. Thus, it is possible to apply different operations in order to modify the existing coastline.

The user may decide to cut a piece of the island off. As a consequence, the terrain will be split into two areas. Depending on the direction of the cut, the algorithm will decide which one of these areas is to be rejected. The direction of the cut is being understood as the direction running from the start to the finish points. The rejected area will be the one on the left-hand side of the direction of the cut. Figure 6.3 presents an initial coastline and the silhouettes obtained after performing cuts with the same start and finish points but with different directions.

Furthermore, the user can also add new pieces to the existing area. Again, the algorithm will behave differently depending on the direction



(a) Initial island terrain. (b) Clockwise supplement. (c) Island reshaped.

(d) Initial island terrain. (e) Anticlockwise supplement. (f) Island reshaped.

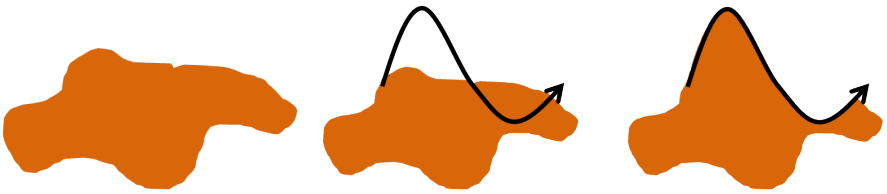
Figure 6.4: Supplementing and reshaping the island.

of the sketched draw. If the line has been sketched clockwise, the *new* area will be added to the existing one. In contrast, if the line has been performed in an anticlockwise direction, then this *new* area will be maintained as the *new* area and the *old* one will be rejected. In Figure 6.4 we can see an example of these possible ways of modifying the area by adding or subtracting a piece of terrain.

We must note that the algorithm will differentiate between cut and supplement operations by testing whether the line goes through the terrain area or not. In those cases in which a line is used to perform more than one operation, each point where the line intersects the silhouette will be interpreted as the finish and start points of the consecutive operations. In Figure 6.5 we depict an area that is being modified by two consecutive operations: the first one consists in an external clockwise supplement and the second one is a curved internal cut that is rejecting the piece of its left-hand side.

6.2.2. Updating the Terrain Height

Every time the coastline silhouette is modified, the terrain algorithm has to react adequately to those changes and recalculate the height of the terrain in order to offer a smooth continuous surface.



(a) Initial island terrain. (b) Combined operations. (c) Island reshaped.

Figure 6.5: Two consecutive operations.

Since we are simulating the terrain of an island, we must take into account the level of the sea. We have to ensure that every single point within the sketched coastline is above sea level. As a consequence, when the coastline changes, it may be necessary to modify the elevations of some onshore points. Ideally, points close to *old* parts of the coastline which remain unchanged should also remain unchanged, but points close to *new* parts of the coastline should be elevated above sea level regardless of their previous height. Therefore, if we reshape the coastline then we have to check whether all the points contained inside the island have the appropriate height.

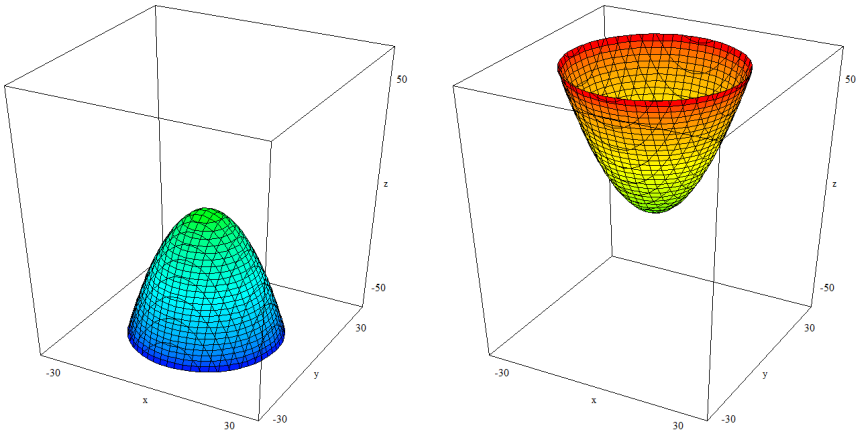
In order to obtain the new height values, we take into account the distance from each point to the nearest piece of *new* coastline D_n and to the nearest piece of *old* coastline D_o, both scaled to the range 0 to 1. The height of each onshore point H_i is calculated as a weighted value between the *old* height H_o and the *new* one H_n, the latter being proportional to D_n. We implement the calculation of the heights with Equation 6.1,where the weight W is calculated by Equation 6.2.

$$H_i = H_o(1{,}0 - W) + H_n W \tag{6.1}$$

$$W = 0{,}5 + 0{,}5 tanh((D_o)^2 - D_n) \tag{6.2}$$

The hyperbolic tangent ($tanh$) function is chosen because it has the appropriate shape, which is close to -1 for points near the *old* coastline and close to 1 for points near the *new* coastline. Moreover, it never goes outside this range. D_0 is squared so that points close to neither coastline are treated as being closer to *old* rather than to *new* coastline.

Whether a pixel is onshore or not is assessed by referring to a silhouette of the island which is recalculated after each change to the coastline.



(a) Elliptic paraboloid downwards. (b) Elliptic paraboloid upwards.

Figure 6.6: Example view of elliptic paraboloids.

The process entails drawing the coastline on a blank array of pixels and using a flood-fill routine (starting from a point clearly outside the coastline) to distinguish sea from land.

6.2.3. Generating Hills

In our work we want to ease the modification of the orography, which refers to the relief of mountains, hills and any other elevated region of a terrain. The idea is to allow the user to create multiple hills having the desired radii, height and location over the terrain.

In our algorithm we define hills as *elliptic paraboloids*. An elliptic paraboloid is shaped like an oval cup. In a suitable coordinate system, it can be represented by the equation:

$$\frac{z}{c} = \frac{x^2}{a^2} + \frac{y^2}{b^2} \tag{6.3}$$

considering that the *elliptic paraboloid* is centered on $0, 0, 0$ with radius a, b, c (along the x, y and z axes), being $a, b, c \in \Re$ and $a > b$. The variable raised to the first power indicates the axis of the paraboloid, in our case z.

This function represents an elliptical paraboloid which opens upwards and can be seen in Figure 6.6b. A negative value of c defines an elliptical paraboloid which open downwards. This latter quadratic surface will be used in our algorithm to define the hills. It is important to note that we allow the user to define valleys by means of elliptical paraboloids which open upwards as can be seen in Figure 6.6a.

With this equation the user can introduce the central point, the radius and the height of each hill. Once we have this information, our algorithm will be able to calculate the height of each point affected by the hill. All those points are obtained with the central point and the radius that have been defined. The height of each single point will be modified by following Equation 6.3. More precisely, we will add the new height to the previous one. By so doing, we allow for the creation of valleys and volcanic mountains.

6.2.4. Filtering the Terrain

In order to obtain a better appearance for the terrain being designed, we can introduce some fuzzy bumps to deform the regular surface. We have implemented a filter to introduce *noise* into the previously defined rounded terrain. This filter can be applied as many times as the user desires and it will give us a number of perturbations proportional to the surface area of the island. The perturbations will be randomly distributed throughout the island surface. These perturbations will also have an elliptical paraboloid shape, but they will be wider than taller and they will be produced upwards or downwards in a stochastic manner. Figure 6.7 shows how the perturbations are located all around the island. The green bumps are the small hills and the red ones are the small valleys that will deform the regular surface of the terrain. It is worth mentioning that more complex procedural noise may have been applied, although the proposed approach is sufficient for our objectives.

Due to the hand-made production of hill and the filters that are applied, there is no way two identical terrain surfaces can be otained.

6.3. User Interface

This section describes our sketching application for terrain generation by using the ideas presented earlier. The proposed application tries

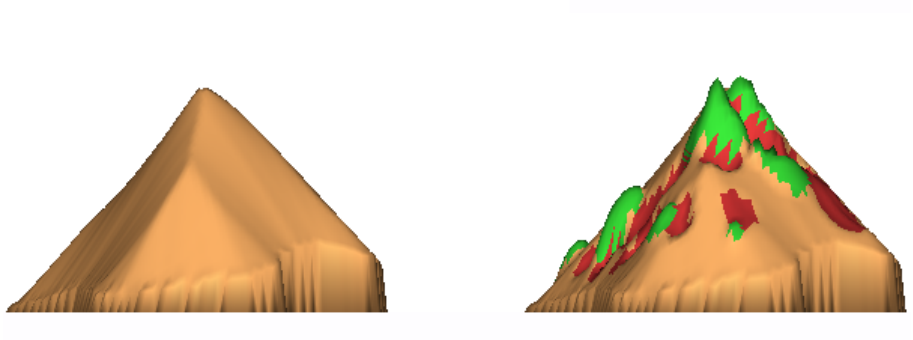


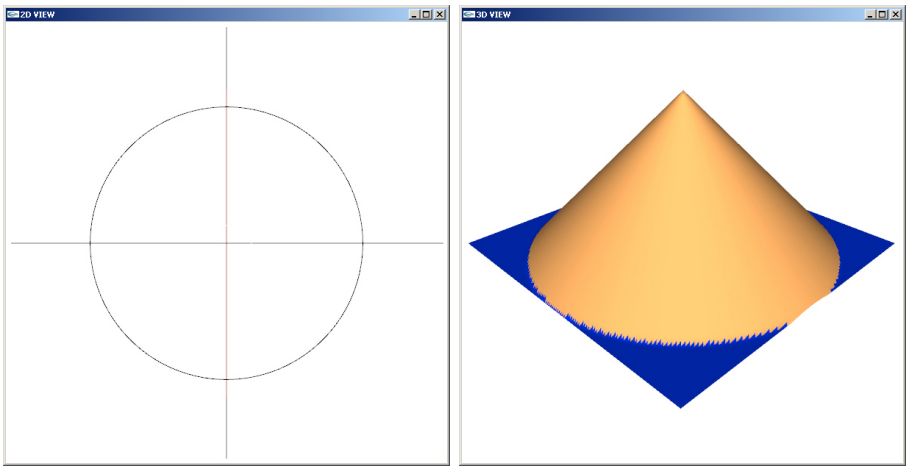
Figure 6.7: Filter applied to the terrain. The green bumps represent the small hills and the red ones the small valleys.

to maintain the three basic elements that form the traditional sketching on a piece of paper:

- feedback, enabling the artist to visually compare the improvements on the sketch.
- re-sketching, allowing the user to modify the previous appearance of the object.
- incremental refinement, adding detail to the object until the artist is satisfied.

These three elements are fundamental for the proper use of a sketching tool [109] and have been addressed in our proposal in order to assure that the final user experiences correctly the reasoning process.

Our proposed framework offers the user an interactive sketching application. This solution consists of two windows. The 2D window depicts the silhouette of the coastline of the island, as seen in Figure 6.8a. The 3D window represents the volumetric view of the whole island, as seen in Figure 6.8b. This 3D view presents a smooth surface which is automatically constructed with the information stored in the heightmap. When the implementation starts, the 2D window contains a circular coastline, as shown in Figure 6.8a. The 3D window, shown in Figure 6.8b, depicts a conical island, which is the initial terrain that the user will be able to modify.



(a) 2D window. (b) 3D window.

Figure 6.8: 2D and 3D Window on Startup.

The presented interface is close to the ideal of a modeless single-tool interface, with all of its major operations being controlled by a single device (pen or single-button mouse). A problem that appears in some of the most advanced sketching applications, like [152], is that they require a multi-modal push-button interface. Our intention is to maintain the original sketching objectives in order to keep our application as simple and natural to use as possible. Nevertheless, it has been necessary to develop a two-button mouse software to integrate all the functionalities presented in the previous section. In the following subsections we will provide a detailed description of the interaction with the aforementioned windows in our application.

6.3.1. 2D Modeling Operations

The 2D window allows the user to perform two basic sketching operations: defining the silhouette of the island and adding and modifying hills.

When interacting with the left mouse button in this window, the user is allowed to design the coastline of the island. It is possible to draw a free-form silhouette interactively and the system will simultaneously update the terrain. As we have mentioned in previous sections, it is

possible to cut and extend the existing terrain by defining lines that start and end on the coastline. Figure 6.9 shows how we could draw any irregular shape to delimit the terrain of our island. Our system includes an additional feature, which has proved popular with users: when a change is made to the coastline, the *old* coastline gradually fades away, taking about two seconds to do so. Pressing the right mouse button during this period removes the amended coastline and reverts back to the *old* coastline.

In our application, when the user clicks with the right button either inside or outside the coastline, the application understands that the user is defining the central point of a hill. Then a colored circular line will appear on the terrain surface surrounding the central point that has just been created. This line represents the area influenced by that particular hill. The user may add as many hills as desired and each one will be depicted in a different color (see Figure 6.9). It is important to comment that when defining hills either inside or outside the coastline, both will affect the terrain that has been generated since both radii affect the coastline. After defining the hills, the user will be able to modify the radii and the location of the hills inside the island:

- Right-clicking on the center point of the hill and dragging, allows the user to change the position of the hill. The user can eliminate a hill by dragging it outside the island until its radius is completely outside the coastline.
- Right-clicking and dragging on the circular line allows the user to modify the hill radius.

The application includes the possibility of *zooming* in on the sketched island by clicking the right button of the mouse. Using the zoom can help the user to get a better overview of the terrain. We have to click outside the island in order to zoom, but always away from the coastline. This is because if the user clicks too close to the coastline, the application will interpret that the user wants to create or modify a hill.

6.3.2. 3D Modeling Operations

In our application, the 3D window shows a volumetric view of the terrain. The user will be able to click with the right button on any of the previously defined hills and can decide on the height of each hill by

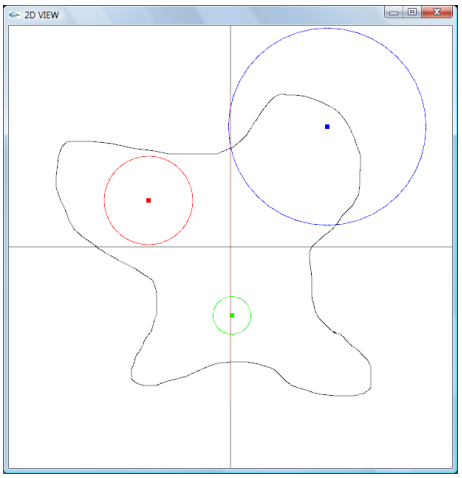


Figure 6.9: 2D window representing the radii of different hills.

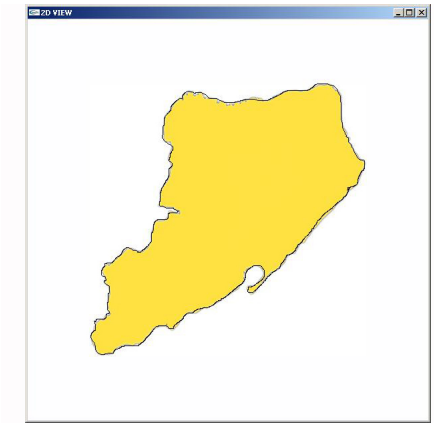
dragging the mouse up and down. If we drag upwards, then the height will be positive and we will create a hill, and if we drag downwards the terrain will be a valley. Furthermore, by dragging the mouse left and right, the user will be able to decrease and increase the size of the radius of the selected mountain.

The 3D window also allows for the use of filters, which the user can decide to apply to the whole terrain in order to introduce some fuzzy bumps. Clicking with the left mouse button on any point on the island and dragging upwards will add filters to the terrain. The more we drag upwards, the more bumps are created. On the contrary, if we drag downwards then the application will understand that we want to decrease the number of perturbations.

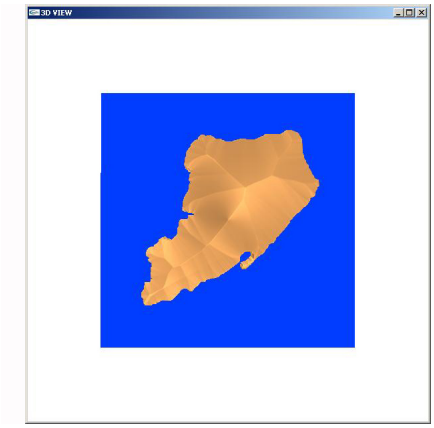
In addition, the 3D window offers two more functions:

- Clicking with the left button away from any hill and dragging, acts as a rotating function. It allows the island to be rotated inside the window.

- Clicking with the right button away from the terrain and dragging acts as a zoom function, that is to say, this increases or decreases the apparent size of the island.



(a) 2D window with the map as background.



(b) 3D window.

Figure 6.10: Staten Island (New York) simulated using an image as a guide.

6.3.3. Contour map

As a guide for the design of the island, it could be possible to use any image as background of the 2D window. Figure 6.10 presentes a recreation of the shoreline of Staten Island in New York (USA). Similarly, it could be possible to add hills to those locations indicated by the image in a contour map.

6.3.4. Putting all together

The set of operations described above enables the users to design an island with the shape an features that they desire. All the aforementioned functions can be applied in any order, as the terrain update process is capable of reacting properly to any action performed.

In order to exemplify the usage of our application, in Figure 6.11 we describe a step-by-step design of an island using our framework. Thus, we start from the initial terrain showed in Figure 6.8 and and we draw the coastline we desire. Then, we locate some hills around the island and we modify their radii, height and location. Then, we apply some filters in order to alter the regular surface of the terrain. With these simple steps we are able to obtain a terrain which is adequate to import it into any 3D application that supports heightmaps.

6.3.5. Output

Our implementation allows the user to save the heightmap in order to be able to use it in different applications, such as game engines and virtual reality applications. Figure 6.12 presents the 2D window displaying an island and its visualization in a selected game engine.

The application that we are presenting has also been prepared to output Terragen [103] files, which consist of *chunks* of information in a specific format indicating the height of the different points of our terrain.

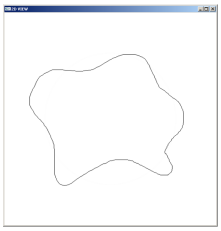
6.4. Detailed Implementation

In this section we will describe in detail the different data structures and processes that make up our framework.

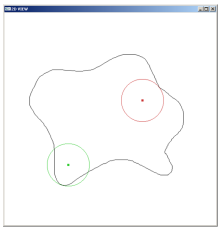
6.4.1. Data Structures

The final terrain that will be visualized or output is composed of three heightmaps that will be added one after the other. As a consequence, with these three data structures we obtain the *Final Grid*. These heightmaps are stored as grids (2D matrix) of floats:

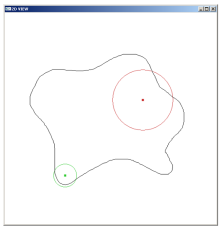
- *Heights Grid*, which contains the heights of the terrain obtained after modifying the coastline.



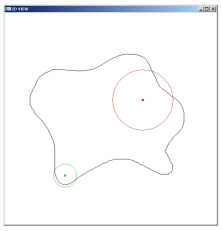
(a) Sketching the coastline.



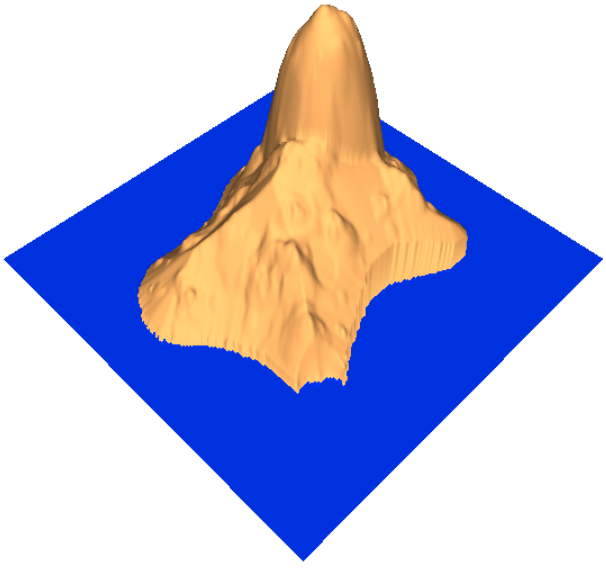
(b) Defining some hills.



(c) Reshaping and relocating the hills.

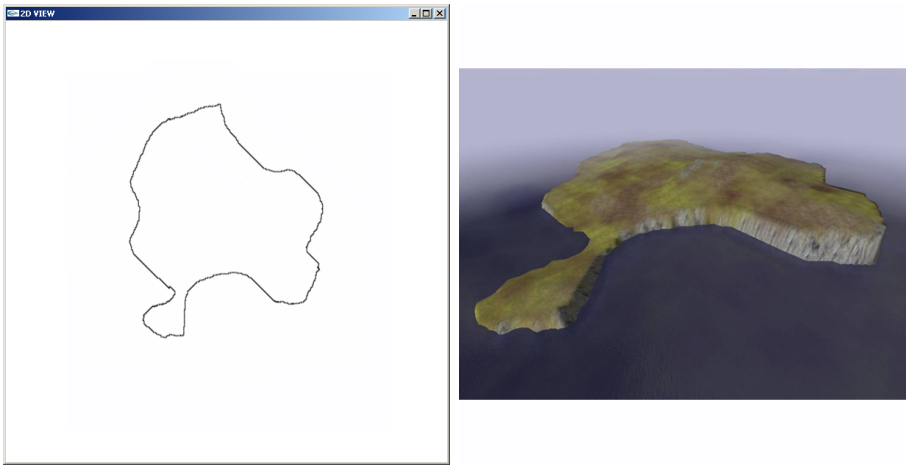


(d) Adding some perturbations.



(e) Visualization of the terrain obtained.

Figure 6.11: Designing a sample terrain in four steps.



(a) 2D Window displaying a sample island.

(b) Game engine rendering a sample island.

Figure 6.12: Island output and rendered in a game engine.

- *Hills Grid*, which stores the increments or decrements in height, due to the hill volumes.
- *Filters Grid*, which holds the variations introduced by the filters that have been inserted.
- *Final Grid*, which stores the sum of the three previous grids.

We assume that the *Heights Grid* defines the basic features of the terrain. Then, the other data structures will add more details to the terrain. It is important to note that the division of the terrain information into those three data structures simplifies the process of updating any of them. Thus, for example, adding a hill only involves modifying the *Hills* grid.

In Figure 6.13 we can see an example of the different grids that make up the *Final Grid*. The *Heights Grid* is obtained after defining the coastline of the island. The *Hills Grid* contains three different hills. Lastly, the *Filters Grid* stores the perturbations introduced by the user. Consequently, these three grids combined together give form to the final terrain.

Moreover, we also need other auxiliary data structures, such as:

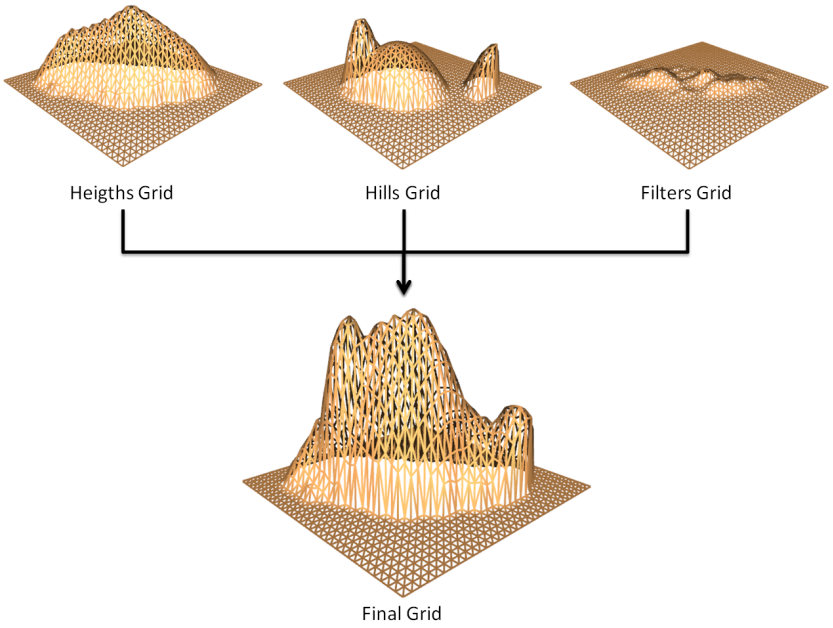


Figure 6.13: Sample composition of the *Final* grid of heights obtained by adding the previously updated grids.

- *Coastline Vector*, which contains the set of points that form the silhouette of the island.
- *HillPoints Vector*, which stores the position, the height and the radius of each hill.
- *FilterPoints Vector*, which holds the position, the height and the radius of the different perturbations.
- *Vertices Vector*, which includes the vertices information for rendering the final terrain.
- *Indices Vector*, which stores the indices information for rendering the final terrain.

This representation uses an internal triangle mesh to represent the 3D island. The 2D heightmap is linked with the 3D representation in order to allow fast and efficient updates, offering a real-time visualization

of the 2D and 3D windows. The implementation has been optimized to ensure that each modeling operation entails updating the minimum amount of information, including both the heightmap and the vertices information, that is to say: spatial coordinates, normals, colors, etc.

6.4.2. Processes

In Section 6.3 we presented the different operations that can be applied by the user. Now we will describe the most important ones more thoroughly.

When the user interacts with the silhouette of the island, the *Coastline Vector* and the *Heights Grid* have to be updated accordingly. If the user creates a *new* isolated coastline then the *Coastline Vector* is completely updated and the *old* coastline is maintained to allow for the *undo* operation. But, if the user reshapes an existing coastline then the application has to calculate the intersection points between them. Depending on the type of operation (defined in Section 6.3) that the user has performed, the algorithm will select the proper way to combine the *old* and the *new* coastline.

Once the new *Coastline Vector* has been modified, the algorithm has to update the *Heights Grid* properly. All the points on this grid have to be updated, so that all the ones that are out of the coastline have a zero height while those points which are *onshore* will have a height value calculated following the algorithm presented in Section 6.2.2.

When the user creates or modifies a hill, the *HillPoints Vector* is updated with the new values. Again, the *Hills Grid* has to be properly modified to follow these changes. For each hill, the algorithm calculates its area of influence. Within that area, all those points which are onshore will have a height calculated by following Equation 6.3. All those points influenced by more than one hill will add the height values that belong to each hill.

The filtering process is similar to the addition of hills. The only difference is that instead of creating hills by following the user input, the algorithms will give each perturbation stochastic positions, heights and radii.

The user interface was programmed using the GLUT (OpenGL Utility Toolkit) library, offering a simple windowing application programming interface for OpenGL. The interaction with the user was accomplished by applying the classical methods that this library offers. Nevertheless,

at this point mention should be made of the method used to select the hills in the 3D Window. If the user interacts with the 3D Window, the application renders the contents of this window using the GL_SELECT rasterization mode. This mode, combined with the use of a small frustum located around the clicking point, offers a simple yet efficient method to find out which of the hills has been selected. More precisely, the name of the selected primitive is obtained by reading the contents of the *Selection Buffer* used in this technique, as this buffer stores the information of the objects *rendered* in the picking area.

6.5. Results

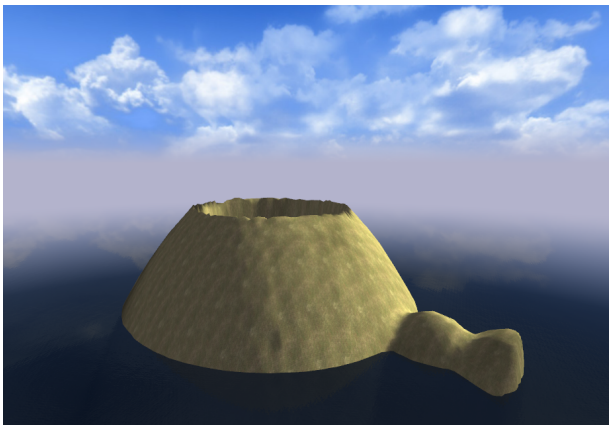
This section presents our results using a Pentium D 2.8 GHz processor with 2 GB. RAM and an nVidia GeForce 8800 GT graphics card. The framework was implemented in C++ with OpenGL. The island was always rendered in real-time and the modifications were always appreciated in a smooth continuous way. It is important to note that the sample images that we have shown throughout the chapter were performed with shiny colors that lack realism but clarify the process that we are explaining.

In order to show the possibilities of our framework, we exported different heightmaps obtained from our terrain generation algorithm. Then we introduced those heightmaps as input into a game engine and we obtained several examples of islands. The game engine that we selected to render our islands in our tests is the Torque Game Engine Advanced (TGEA) [100, 101] released by GarageGames.

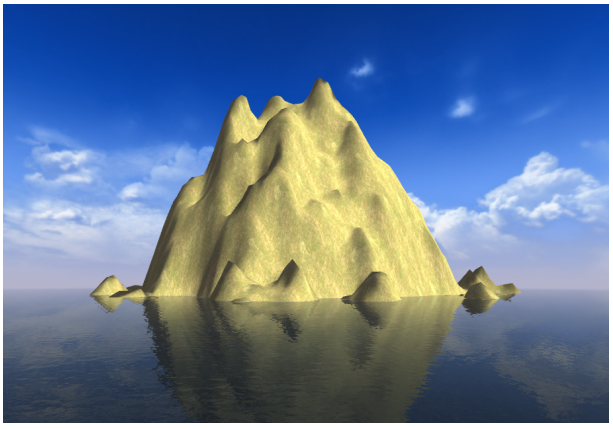
In Figure 6.14a we have rendered a volcano with two tiny hills on one side. The volcano consists of a big hill but with a hole in the middle. This hole is performed by applying a slightly smaller negative-height hill, located in the middle of the first hill we created.

Figure 6.2 depicts an elongated island. The island is quite abrupt after having applied some filters in order to give the final terrain a more realistic appearance. Initially we set up just one big hill on one side of the island and a small one on the other side of the island. Finally, the filters added some perturbations to the terrain.

Finally, in Figure 6.14b we created a more circular island. This island is also very craggy after having applied many filters. At first, we created a single huge round hill. After that, we included a great number of perturbations until we obtained the desired appearance for the island.



(a) Volcano-shaped island.



(b) Craggy island obtained after applying many filters to an initially round island.

Figure 6.14: Examples of Islands created with our algorithm and included in the Torque Game Engine.

Although our sketching interface only creates single islands, a game engine could load several islands simultaneously in order to create an archipelago, as seen in Figure 6.15.

Our implementation has been prepared to create also Terragen-compatible output. Terragen is a commonly used commercial software for creating and visualizing terrain [103]. Figure 6.16 offers two images that were obtained by importing two generated islands into Terragen.

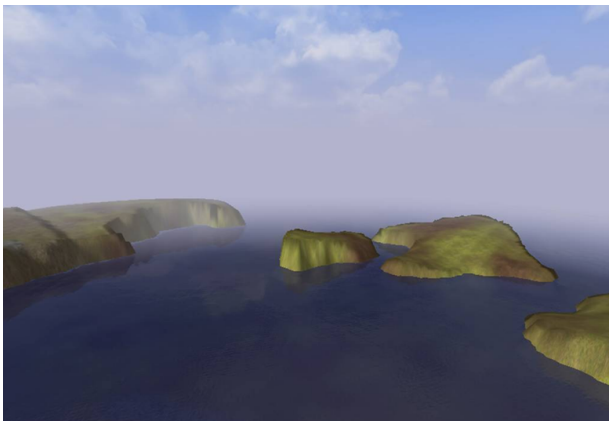


Figure 6.15: Archipelago created with our solution.

6.5.1. User Study

We have considered that it is compulsory to perform a user study in order to evaluate the quality of our sketching application.

One of the first tasks to perform when conducting a usability study is to decide on the persons that we are going to recruit. Different studies have proven that 5 users are enough to assess the quality of software applications [153], although evaluating visualization results need a different amount of volunteers in order to obtain valuable results [154]. As a consequence, we will need more users as we also test their *cognitive* capabilities, which means in our case, the faculty of a user to process information after having perceived the visual inputs.

In our case we have conducted the test to 30 people, grouped in three catergories depending on their expertise both in computer use in general and in computer design in particular. Our tests were carried out in our university lab and all of the participants were staff or students at the university.

The objective of our informal user study consisted in analyzing our sketching application from different perspectives. First, we asked our volunteers to grade from 1 to 10 the overall quality perception of the application, being 1 the worst and 10 the best. Secondly, we monitorized their activity to record the time passed until they obtained a satisfactory terrain. Finally, we asked them to report any difficulty or mistake they could find when using our application.

Figure 6.16: Example of Islands created with our algorithm and included in TERRAGEN.

Study Group	Number of volunteers	Overall Satisfaction	Avg. Required Time (min.)	Observed Problems
Computer Scientists	12	8	3	12
Designers	7	6	5	7
Other Disciplines	11	9	4	2

Table 6.1: Results obtained with 30 university volunteers.

The test sessions consisted in giving the first-time users some basic indications in order to make them know how our application works. After that, we made them try to draw an island silhouette with the desired hills. They played with different coastline, hills and filters until they outlined the desired appearance of island and terrain.

In Table 6.1 we present the results obtained with university volunteers. Most of them were satisfied with their results after less than five minutes. It is important to mention that both computer scientist and designers found several problems that helped us to improve the application. Most of them found the application difficult to use at the beginning, although after some practice they started modeling their terrains. These

initial usability problems helped us to modify some aspects of the application and also encouraged us to create a brief guide to explain how the application works. Moreover, most users found the application *funny* and played with it after acquiring a little of expertise. Finally, we also encouraged some of them to include the terrain inside the Torque Game Engine [100, 101] with our help, in order to give them the possibility to experience gaming over their modeled island and terrain.

6.6. Conclusions

This chapter presents an algorithm for terrain generation which is suitable for users who wish to have full control over the whole creation process. We have also presented a simple tool for creating solid models of imaginary islands. The tool is easy to use and requires only a minimal user interface, with all of its major operations being controlled by a two-button mouse. From this application, the user can add, remove and reshape existing hills interactively and the terrain will be updated accordingly. Moreover, the user is able to modify the silhouette of the island and add fuzzy bumps as desired.

Accordingly, the images of islands that we have shown in the previous section show how our approach is capable of offering realistic terrains adapted to the needs of the users. Thus, the final user can decide on the final appearance of the island, as it is possible to apply any number of filters. Nevertheless, the user could choose not to apply filters in order to obtain a fairly rounded terrain which could be useful for a cartoon-like environment. The usability study, which was performed among people with different levels of computer skills showed that the user interface we finally selected was comfortable and adequate in most cases.

The most promising area for future work consists in adding new features to our existing application. We note that increasing the features is, in principle, easy. For example, it would be straightforward to change the application so that the designer could mark particular areas of the island as beach or forest. In this sense, we are interested in allowing the user to include weather phenomena, vegetation and other decorative elements on their island. The problem with any of these is that each additional option would make the user interface more complicated, thereby losing a major advantage of the existing user interface, its simplicity. Nevertheless, a more complicated user interface could be justified by analogy with sketching on paper, as paper maps often use different colours of ink

for forests and lakes.

A group of new features which could be added without compromising the user interface is simulation of physical processes. For example, the height contours of an island could be determined, not by a convenient trigonometrical formula, but by the way tha lava could flow after a real volcanic eruption and the way that the resulting shape could be sculpted by wind and rainfall.

Chapter 7

Conclusions and Future Work

The work presented in this Ph.D. dissertation has been proposed as an improvement on natural phenomena simulation. With this objective in mind, we have presented solutions to the creation and efficient rendering of rain, sea and terrain.

To sum up, this dissertation has started by presenting a framework for the creation, management and rendering of rain, introducing the concept of level-of-detail in rain simulation and exploiting the capabilities of the *Geometry Shader*. Then, we have described an improvement to the previous solution which uses CUDA to offer rain interaction with the scene, detecting the collisions of the raindrops with the scenario and rendering the related splashes. After the rain rendering study, we have presented a solution for the real-time rendering of sea, proposing a new tessellation technique for the heightmap simulating the sea surface. Lastly, an approach to design terrain has been proposed, which simplifies the user interaction and offers a high degree of control over the generated terrain.

Throughout the different contributions, our aim has been to enhance the visualization of each of these natural phenomena by offering an easy management and a high performance by means of the graphics hardware in different ways.

This chapter is organized as follows. First, we conclude on the contributions offered by the different proposals. Then, we outline ideas for

future work. Finally, we list the publications related to this Ph.D. dissertation and the research projects that have enabled the development of this thesis work.

7.1. Conclusions

Chapter 1 introduced the theme of the research work presented in this dissertation, describing natural phenomena and related procedural modeling techniques. Procedural modeling offers an ideal framework to adapt the complexity of the visualized geometry to the limitations or the specific needs of the final applications.

In Chapter 2 we described the previous work related to each natural phenomenon that we have considered. Thus, we presented the state of the art on rain rendering, where we could see that the existing solutions still presented limitations for real scenarios where the user moved very fast. Although interesting approaches had been presented, most of them were too complex for an implementation in a real-time application. Physics of rain were also described, as our solution was based on how real raindrops behave. After the study on rain rendering, we presented the related work for the sea simulation and the tessellation techniques that were applied. In this case we could see how it is possible to find a wealth of research on ocean simulation, applying many different techniques to obtain a realistic and interactive simulation. Nevertheless, there are not many GPU-based tessellation techniques oriented towards sea simulation. Lastly, the remaining of the chapter presented the related work on terrain generation, by covering solutions like the procedural approaches for real terrain simulation. This study showed us that terrain generation frameworks were far from offering a high degree of control to the final user. In this sense, although some solutions offered very realistic environments, they usually resort to complex interfaces were most of the parameters that define the terrain are fixed or difficult to adjust.

After we analyzed the state of the art on the different topics, we could see that there was still room for improvement. Consequently, Chapter 3 proposed a set of techniques to create and efficiently visualize scenarios with realistic rain. The proposed framework offered different approaches depending on the relation between the user location and the rain area location. In addition, we included multiresolution techniques which were applied directly and only on the GPU. Moreover, a study of the generation of particles on the GPU was also proposed, preventing us

from having to initially distribute the particles throughout the container and enabling us to distribute them in real time. The results that we obtained improved on those achieved by previous solutions. The conducted user study proved that our presented solution was capable of offering similar rain intensity sensations with much less particles. The use of patterns improved the performance as it reduced the amount of particles to translate while maintaining a similar appearance. Moreover, the selected shape of the rain container avoiding relocating the whole container continuously, which was one of the main drawbacks of previous solutions as they were not suitable for game environments where the user made fast and continuous camera movements. We offered a solution which was fast, simple, efficient and easily integrated into existing virtual-reality environments.

In order to improve on the previous rain technique and to offer further exploitation of graphics hardware, Chapter 4 described a technique to efficiently detect, process and simulate collisions of raindrops on the scenario. The improvement of the simulation was possible thanks to the use of CUDA, as its flexibility allowed us to simpify the algorithms to manage collisions and the related splashes. Thus, traditional hardware programming would involve many rendering passes and a complex memory management to obtain these effets. The quality obtained in the different tests was due to the possibility of including a high quantity of particles and splashes. Moreover, CUDA enabled the graphics application to decrease the use of CPU time by 50 %, which allowed the 3D software to dedicate that time to make other calculations.

From a different perspective, Chapter 5 presented a new fully-GPU tessellation technique which was applied to ocean rendering. The scheme proposed avoided the appearance of T-vertices and other artifacts that can produce holes in the animated surface of the ocean. Another important aspect of this tessellation algorithm was the coherence exploitation, as it is capable of reusing the latest approximations when refining and coarsening the mesh. In this sense, we minimize the operations to perform in both cases, reducing the temporal cost involved in the tessellation process. This coherence maintenance is possible to the fact that we store some small pieces of information in each triangle that are sufficient for altering the level of detail. Although many simulation techniques were studied in the state-of-the-art Chapter, the proposed framework considered the use of Perlin noise to animate the water surface, as its implementation on the GPU is simple, although more complex tech-

niques could be equally applied. Some optical effects were considered, as they contribute to the realism of the ocean scene without increasing too much the rendering time.

Finally, Chapter 6 thoroughly described a tool for terrain generation which was suitable for users who wish to have full control over the whole creation process. The tool was easy to use and required only a minimal user interface, with all of its major operations being controlled by a two-button mouse. From this application, the user could add, remove and reshape existing hills interactively and the terrain would be updated accordingly. Moreover, the user was able to modify the silhouette of the island and add fuzzy bumps as desired. With these objectives in mind, the final interface was designed to offer an intuitive environment and was refined with the input of the users that tested the sketching application. The usability study, which was performed among persons with different levels of computer skills, showed that the user interface we finally selected is comfortable and adequate in most cases. The possibility of exporting the obtained terraine to game engines is a key aspect for allowing the users to create their own gaming scenarios.

7.2. Future work

In this dissertation we have presented different improvements on the simulation of natural phenomena. Simulation of enviromental phenomena still receives a great amount of attention from researchers as its incorporation in games and feature films increases the realism of the scenes. In this sense, it is our interest to continue working on these issues to improve the realism and performance of the solutions.

Firstly, the rain simulation framework that we have proposed could be improved with methods for simulating light interaction and other effects that have been presented in previous works by other authors. Furthermore, we would also be interested in studying the possibilities of applying level-of-detail techniques to these effects in order to obtain a more realistic simulation while maintaining a low computational cost. Moreover, it could be possible to apply the proposed framework to the simulation of snow, as this phenomenon is also visually perceptible and its simulation could be improved with our multiresolution proposal. Regarding the CUDA-based framework, we plan to use the collision detection for further effects, like the accumulation of rain on the ground and the alteration of the properties of the surfaces depending on the amount

of water received. This latter line of work is specially interesting as it can considerably contribute to the development of a physically and visually realistic rain simulation. Finally, we consider interesting to study the alteration of the precipitation and the splashes in situations of wind, so that the system solves correctly situations where drops do not fall in a totally vertical way.

Secondly, we consider that it would be interesting to improve on the ocean simulation by adding more effects to enhance the final visual quality and by studying other methods to animate the ocean surface. Shallow water is also of interest, as ocean behaves differently when approaching the shoreline. In this sense, we would like to consider how waves break, producing foam and spray, in order to adapt GPU methods and level-of-detail algorithms to the simulation of this phenomena. Moreover, the study of methods to offer the interaction of the ocean surface with other objects is very attractive, as a real scenario usually considers objects falling and floating on the sea. From a different perspective, the proposed tessellation algorithm could be also applied to terrain rendering. As a consequence, it is our interest to analyze the possibilities offered by a GPU-based tessellation technique in terrain visualization. Nevertheless, it is worth mentioning that the appearance of Directx 11 in the near future will involve further advances in computer graphics. Among the new stages that the rendering pipeline will include, we could highlight the tessellation unit, which will be able to produce semi-regular tessellations [155] by itself. This feature can be directly used for the ocean surfaces and, thus, we believe that this unit will be key in the near future.

Then, the sketching tool was mainly focused on designing islands, although we consider that it would be interesting to extended the algorithms in order to manage other types of scenarios. The most promising area for our terrain generation tool is to add new features to our existing application. For example, it could be interesting to change the application so that one could mark particular areas of the island as beach or forest. In this sense, the authors are interested in allowing the user to include weather phenomena, vegetation and other decorative elements on their island. Moreover, all this information could be gathered by the application and output to a render engine or a final application to render the defined scenario. One feature that could be added is the simulation of physical processes. For example, the height contours of an island could be determined, not by a trigonometrical formula but by the way lava flows after a real volcanic eruption and the way the resulting shape is

sculpted by wind and rainfall.

Lastly, we would also like to research on the possibilities offered by combining the different approaches presented. In this sense, for example, we would like to explore the calculations of rain collisions on an animated ocean surface. Similarly, we consider interesting to investigate on the use of the CUDA programming framework to ocean tessellation.

7.3. Publications

To endorse the different works presented in this dissertation, this section lists the publications that have been obtained with the presented contributions. Moreover, a list of publications which are not directly related to this Ph.D. dissertation is presented, as well as a list of research projects that have funded this research.

- One journal publication and two more under review:

  - **Creation and Control of Rain in Virtual Environments**
    A. Puig-Centelles, O. Ripollés, M. Chover
    The Visual Computer, vol. 25(11), pp. 1037-1052, 2009.

  - **Adaptive Tessellation in Ocean Surfaces**
    A. Puig-Centelles, F. Ramos, O. Ripollés, M. Chover, M. Sbert
    The Visual Computer, Under review.

  - **Automatic Terrain Generation**
    A. Puig-Centelles, P. A. C. Varely, O. Ripollés, M. Chover
    Interacting with Computers, Under review.

- Six conferences, two of them under review:

  - **Rain Simulation on Dynamic Scenes**
    A. Puig-Centelles, N. Sunyer, O. Ripollés, M. Chover, M. Sbert
    Computer Graphics International (CGI), 2010, Under review.

- **Tessellating Ocean Surfaces**
  A. Puig-Centelles, F. Ramos, O. Ripollés, M. Chover, M. Sbert
  Int. Conf. on Computer Graphics Theory and Applications (GRAPP), 2010, Under Review.

- **Simulación de Lluvia sobre Escenas Dinámicas**
  N. Sunyer, A. Puig-Centelles, O. Ripollés, M. Chover, M. Sbert
  Congreso Español de Informática Gráfica (CEIG), pp. 247-250, 2009.

- **Optimizing the Management and Rendering of Rain**
  A. Puig-Centelles, O. Ripolles, M. Chover
  Int. Conf. on Computer Graphics Theory and Applications (GRAPP), pp. 373-378, 2009.

- **Automatic Terrain Generation with a Sketching Interface**
  A. Puig-Centelles, P. A. C. Varely, O. Ripolles, M. Chover
  15^{th} Winter School of Computer Graphics (WSCG), pp. 39-46, 2009.

- **Multiresolution Techniques for Rain Rendering in virtual Environments**
  A. Puig-Centelles, O. Ripolles, M. Chover
  Int. Symp. on Computer and Information Sciences (ISCIS), pp. 1-4, 2008.

- **Técnicas para visualización de lluvia en entornos virtuales**
  A. Puig-Centelles, O. Ripolles, M. Chover
  Congreso Español de Informática Gráfica (CEIG), pp. 159-167, 2008.

- **Sketching Islands for a Game Environment**
  P. A. C. Varely, A. Puig-Centelles, M. Chover
  5th European Conference on Visual Media Production (CVMP),

pp. 1-10, 2008.

Other publications which are not directly related to this dissertation:

- One journal Publications:
  - **Rendering continuous level-of-detail meshes by Masking Strips**
    O. Ripollés, M. Chover, J. Gumbau, F. Ramos, A. Puig-Centelles
    Graphical Models, vol. 71 (5), pp. 169-196, 2009.

- Four conferences:
  - **Multimedia Autonomous Learning Based on Video Tutorials**
    A. Puig-Centelles, O. Ripollés, O. Belmonte, M. Arregui, O. Coltell, M. Chover
    International Technology, Education and Development Conference (INTED), 2010.
  - **Virtual Classroom for Practical Internet Autonomous Learning: New Materials and Course Assesment**
    M. Arregui, J. Huerta, A. Puig-Centelles, J.M. Pérez, O. Ripollés, O. Coltell
    International Technology, Education and Development Conference (INTED), 2010.
  - **View-Dependent Multiresolution Modeling on the GPU**
    O. Ripollés, J. Gumbau, M. Chover, F. Ramos, A. Puig-Centelles
    17^{th} Winter School of Computer Graphics (WSCG), pp. 121-126, 2009.
  - **Educational instant messaging in a 3D environment**
    A. Puig-Centelles, O. Ripolles, M. Chover, P. Prades
    Int. Conf. on Cognition and Exploratory Learning in Digital

Age (CELDA), pp. 49-56, 2007.

Finally, it is worth mentioning that the work presented in this dissertation is embedded within two research projects:

- **Contenido Inteligente para Aplicaciones de Realidad Virtual: una Aproximación Basada en Geometría**
  Ministerio de Educación y Ciencia, (TIN2007-68066-C04-02 and TIN2007-68066-C04-01), 2007 - 2010.
- **Geometría Inteligente**
  Fundació Caixa Castelló-Bancaixa (P1 1B2007-56), 2007 - 2010.

Bibliography

[1] Natalya Tatarchuk. Artist-directable real-time rain rendering in city environments. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 23–64, New York, NY, USA, 2006. ACM.

[2] W. Changbo, Z. Wang, X. Zhang, L. Huang, Z. Yang, and Q. Peng. Real-time modeling and rendering of raining scenes. *Visual Computer*, 24:605–616, 2008.

[3] Kazufumi Kaneda, Shinya Ikeda, and Hideo Yamashita. Animation of water droplets moving down a surface. *The Journal of Visualization and Computer Animation*, 10(1):15–26, 1999.

[4] Ines Stuppacher and Peter Supan. Rendering of water drops in real-time. In *Proceedings of Central European Seminar on Computer Graphics for students (CESCG)*, 2007.

[5] Paul Fearing. Computer modelling of fallen snow. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 37–46, 2000.

[6] W. T. Reeves. Particle systems: a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[7] Tamy Boubekeur and Christophe Schlick. A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum*, 27(1):102–114, 2008.

[8] N. Tatarchuk. Advanced topics in GPU tessellation: Algorithms and lessons learned. Gamefest Presentation, http://ati.amd.com/developer/Gamefest08/Tatarchuk-Tessellation-Gamefest2008.pdf, 2008.

[9] M. Schwarz and M. Stamminger. Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum*, 28(2):365–374, 2009.

[10] E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications. In *Eurographics '97 Tutorial Notes*, 1997.

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

[12] Pierre Rousseau, Vincent Jolivet, and Djamchid Ghazanfarpour. Realistic real-time rain rendering. *Computers & Graphics*, 30(4):507–518, 2006.

[13] O. N. Ross. Optical remote sensing of rainfall micro-structures. Technical report, Fachbereich Physik der Freien Universität Berlin. Diplomarbeit thesis, 2000.

[14] Harold Davis. http://www.digitalfieldguide.com/blog/576, 2006.

[15] John Langford. http://www.cosmiccandidcamera.com/2008/09/my-idea-of-paradise.html, 2008.

[16] Jeff Johnson. http://www.woodskunk.addr.com/jbj/Mountains/mountain_page.html, 2007.

[17] H. Zhou, J. Sun, G. Turk, and J. M. Rehg. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848, 2007.

[18] Thatcher Ulrich. Continuous lod terrain meshing using adaptive quadtrees. `http://www.gamasutra.com/features/20000228/ulrich_01.htm`, 2008.

[19] Timo Schmiade. Adaptive GPU-based terrain rendering. Master's thesis, Computer Graphics Group, University of Siegen, 2008.

[20] Sonia Stanik and Michael Werman. Simulation of rain in videos. In *International Journal of Computer Vision Texture 2003 (The 3rd international workshop on texture analysis and synthesis)*, pages 95–100, 2003.

[21] Kshitiz Garg and Shree K. Nayar. Detection and removal of rain from videos. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 01:528–535, 2004.

[22] K. Garg and S.K. Nayar. Photometric model of a rain drop. Technical report, Technical Report, Department of Computer Science, Columbia University, Sep 2004.

[23] Kshitiz Garg and Shree K. Nayar. Photorealistic rendering of rain streaks. *ACM Transactions on Graphics*, 25(3):996–1002, July 2006.

[24] Ying Nian Wu Stefano Soatto, Gianfranco Doretto. Dynamic textures. In *International Conference on Computer Vision*, pages 439–446, July 2001.

[25] Niniane Wang and Bretton Wade. Rendering falling rain and snow. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, page 14, New York, NY, USA, 2004. ACM.

[26] Kensuke Kusamoto, Katsumi Tadamura, and Yoshihiko Tabuchi. A method for rendering realistic rain-fall animation with motion of view. *Information Processing Society of Japan, SIG Notes*, 1109(106):21–26, 2001.

[27] Zhong-Xin Feng, Min Tang, Jinxiang Dong, and Shang-Ching Chou. Real-time rain simulation. In *Computer Supported Cooperative Work in Design, LNCS 3865*, pages 626–635. Springer-Verlag Berlin Heidelberg, 2006.

[28] L. Wang, Z. Lin, T. Fang, X. Yang, X. Yu, and S. B. Kang. Real-time rendering of realistic rain. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 156, New York, NY, USA, 2006. ACM.

[29] P. Rousseau, V. Jolivet, and D. Ghazanfarpour. Gpu rainfall. *Journal of graphics, gpu, and game tools*, 13(4):17–33, 2008.

[30] Sarah Tariq. Rain. nvidia whitepaper. http://developer.download.nvidia.com/SDK/10/direct3d/Source/rain/doc/RainSDKWhitePaper.pdf, 2007.

[31] J. S. Marshall and W. McK. Palmer. *The distribution of raindrops with size*. Journal of Meteorology. McGill University, Montreal, 1948.

[32] Roy M. Rasmussen. A review of theoretical and observational studies in cloud and precipitation physics. *Rev. Geophys. Vol. 33 Suppl. American Geophysical Union, US*, 1995.

[33] Pierre McComber. Sensitivity of selected freezing rain models to meteorological data. In *In Proceedings of the 57th Annual Eastern snow Conference*, Syracuse, NY, USA, 2000.

[34] J.F Straube and E.F.P.Burnett. Simplified prediction of driving rain on buildings. In *In Proceedings of International Physics Conference*, pages 375–382, Eindhoven, The Netherlands, 2000.

[35] Cleve Hallenbeck. Summer types of rainfall in upper pecos valley. *Monthly Weather Review*, pages 209–216, 1917.

[36] A.C. Best. The size distribution of raindrops. *Quarterly Journal of the Royal Meteorological Society*, 76(327):16–36, 1950.

[37] Kenneth V. Beard and Catherine Chuang. A new model for the equilibrium shape of raindrops. *Journal of the atmospheric sciences*, 44(11):1509–1524, 1986.

[38] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 49–57, 1990.

[39] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, 2001.

[40] Jean-Christophe Gonzato, Thomas Arcila, and Benoit Crespin. Virtual objects on real oceans. In *Proceedings of GRAPHICON'2008*, pages 49–56, 2008.

[41] A. Iglesias. Computer graphics for water modeling and rendering: a survey. *Future Generation Computer Systtems*, 20(8):1355–1374, 2004.

[42] Alain Fournier and William T. Reeves. A simple model of ocean waves. *SIGGRAPH Comput. Graph.*, 20(4):75–84, 1986.

[43] D. R. Peachey. Modeling wave and surf. *Computer. Graphics*, 20(4):75–83, 1986.

[44] Pauline Y. Tsó and Brian A. Barsky. Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Trans. Graph.*, 6(3):191–214, 1987.

[45] J. M. Cieutat, J. C. Gonzato, and P. Guitton. A new efficient wave model for maritime training simulator. In *SCCG '01: Proceedings of the 17th Spring conference on Computer graphics*, page 202, 2001.

[46] F. J. Gerstner. Theorie der wellen (reprint from the original of 1802). *Ann des Physik*, 32:412–440, 1809.

[47] J. Gonzato and B. L. Saëc. On modelling and rendering ocean scenes. *The Journal of Visualization and Computer Animation*, 11(1):27–37, 2000.

[48] Jos Stam and Eugene Fiume. Turbulent wind fields for gaseous phenomena. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 369–376, 1993.

[49] Nils Thurey, Matthias Muller-Fischer, Simon Schirm, and Markus Gross. Real-time breakingwaves for shallow water simulations. In *PG '07: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, pages 39–46, 2007.

[50] Willard J. Pierson and Lionel Moskowitz. A proposed spectral form for fully developed wind seas based on the similarity theory of s. a. kitaigorodskii. *JOURNAL OF GEOPHYSICAL RESEARCH*, 69(24):5181 – 5190, 1964.

[51] E. Monahan and G. Mac Niocaill. *Oceanic Whitecaps: Their Role in Air Sea Exchange Processes*. D. Reidel, 1986.

[52] J. Tessendorf. Simulating ocean water. In *Siggraph Course Notes*, 1999.

[53] Gary A. Mastin, Peter A. Watterberg, and John F. Mareda. Fourier synthesis of ocean scenes. *IEEE Comput. Graph. Appl.*, 7(3):16–23, 1987.

[54] Simon Premoze and Michael Ashikhmin. Rendering natural waters. In *Proceedings of the 8th Pacific Conference on Computer Graphics and Applications*, page 23, 2000.

[55] J. L. Mitchell. Real-time synthesis and rendering of ocean water. Technical report, ATI, 2005.

[56] Forest Kenton Musgrave. *Methods for realistic landscape imaging.* PhD thesis, 1993.

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

[58] Claes Johanson. Real time water rendering-introducing the projected grid concept. Technical report, Master of Science Thesis, Lund University, 2004.

[59] Jens Schneider and Rüdiger Westermann. Towards real-time visual simulation of water surfaces. In *Vision, Modelling and Visualization (VMV01)*, pages 211–218, 2001.

[60] Xudong Yang, Xuexian Pi, Liang Zeng, and Sikun Li. Gpu-based real-time simulation and rendering of unbounded ocean surface. In *CAD-CG '05: Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics*, pages 428–433, 2005.

[61] Sébastien Thon and Djamchid Ghazanfarpour. Ocean waves synthesis and animation using real world information. *Computers & Graphics*, 26(1):99–108, 2002.

[62] Jocelyn Fréchot. Realistic simulation of ocean surface using wave spectra. In *Proceedings of the First International Conference on Computer Graphics Theory and Applications (GRAPP 2006)*, pages 76–83, 2006.

[63] Emmanuelle Darles, Benoît Crespin, and Djamchid Ghazanfarpour. Accelerating and enhancing rendering of realistic ocean

scenes. In *Proc. of the 15-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2007)*, pages 287–294, 2007.

[64] Haik Lorenz and Jürgen Döllner. Dynamic mesh refinement on gpu using geometry shaders. In *Proceedings of the 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2008*, pages 97–104, 2008.

[65] Christopher Dyken, Martin Reimers, and Johan Seland. Semi-uniform adaptive patch tessellation. *Computer Graphics Forum*, page In press, 2009.

[66] L. Buatois, G. Caumon, and B. Lévy. GPU accelerated isosurface extraction on tetrahedral grids. In *International Symposium on Visual Computing*, pages 383–392, 2006.

[67] L. Shiue, I. Jones, and J. Peters. A real-time GPU subdivision kernel. *ACM Transactions on Graphics*, 24(3):1010–1015, 2005.

[68] M. Guthe, A. Balázs, and R. Klein. GPU-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, 2005.

[69] Tamy Boubekeur and Christophe Schlick. Generic mesh refinement on GPU. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 99–104, 2005.

[70] M. Bokeloh and M. Wand. Hardware accelerated multi-resolution geometry synthesis. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 191–198, 2006.

[71] Yaohua Hu, Luiz Velho, Xin Tong, Baining Guo, and Harry Shum. Realistic, real-time rendering of ocean waves: Research articles. *Computer Animation and Virtual Worlds*, 17(1):59–67, 2006.

[72] E. J. Stollnitz, A. D. Derose, and D. H. Salesin. Wavelets for computer graphics. *Computer Graphics and Applications, IEEE*, 15(3):76–84, 1995.

[73] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *SIGGRAPH '00*, pages 335–342, 2000.

[74] Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr. Dynamic real-time deformations using space & time adaptive sampling. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 31–36, 2001.

[75] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. Interactive animation of ocean waves. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 161–166, 2002.

[76] J. Demers. The making of clear sailing, secrets of the NVIDIA demo team. http://www.nzone.com/object/nzone_clearsailing_makingof1.html, 2005.

[77] Yung-Feng Chiu and Chun-Fa Chang. GPU-based ocean rendering. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 2125–2128, 2006.

[78] J.S. Jaffe, K. D. Moore, J. McLean, and M.P. Strand. Underwater optical imaging: Status and prospects. *Oceanography*, 14(3):66 – 76, 2001.

[79] Tomas Akenine-Moller and Eric Haines. *Real-Time Rendering (2nd Edition)*. AK Peters, 2002.

[80] C. E. Aguiar and A. Ribeiro de Souza. Google Earth Physics. *Physics Education*, 44(6):624 – 626, 2009.

[81] Jens Schneider, Tobias Boldte, and Ruediger Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, Modeling and Visualization 2006*, pages 145–152, 2006.

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

[83] Carsten Dachsbacher. Interactive terrain rendering: Towards realism with procedural models and graphics hardware. Technical report, Fiedrich Alexander Universität Erlangen-Nürnberg, Germany. Thesis, 2006.

[84] Alex D. Kelley, Michael C. Malin, and Gregory M. Nielson. Terrain simulation using a model of stream erosion. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 263–268, New York, NY, USA, 1988. ACM.

[85] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 41–50, New York, NY, USA, 1989. ACM.

[86] Kansas State University Wind Erosion Research Unit. Weru. wind erosion simulation models. http://www.weru.ksu.edu/weps.html, 2003.

[87] M. Wacker B. Neidhold and O. Deussen. Interactive physically based fluid and erosion simulation. In *Eurographics Workshop on Natural Phenomena*, pages 25–32, 2006.

[88] Teong Joo Ong, Ryan Saunders, John Keyser, and John J. Leggett. Terrain generation using genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1463–1470, New York, NY, USA, 2005. ACM.

[89] Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. Breeding terrains with genetic terrain programming: The evolution of terrain generators. *International Journal of Computer Games Technology*, 2009, 2009.

[90] M. F. Worboys. *GIS: A Computing Perspective*. Taylor and Francis, 1995.

[91] Spanish Government. National geographical institut. http://www.ign.es, 2008.

[92] USA. The federal geographic data committee. http://www.fgdc.gov/, 2008.

[93] United States Geological Survey. Usgs. national mapping program standards. http://rockyweb.cr.usgs.gov/nmpstds/demstds.html, 2003.

[94] Kai Hormann, Salvatore Spinello, and Peter Schröder. c^1-continuous terrain reconstruction from sparse contours. In *Proceedings of 8th Int. Workshop Vision, Modeling, and Visualization*, pages 289–297, 2003.

[95] Sergiy Fefilatyev, Volha Smarodzinava, Lawrence O. Hall, and Dmitry B. Goldgof. Horizon detection using machine learning techniques. In *ICMLA '06: Proceedings of the 5th International Conference on Machine Learning and Applications*, pages 17–21, Washington, DC, USA, 2006. IEEE Computer Society.

[96] Qicheng Li, Guoping Wang, Feng Zhou, Xiaohui Tang, and Kun Yang. Example-based realistic terrain generation. In *16th International conference on artificial reality and telexistence (ICAT), LNCS 4282*, pages 811–818, 2006.

[97] PixelActive Inc. Cityscape 1.7. http://pixelactive3d.com/Products/CityScape, 2009.

[98] Quad Software. Grome. http://www.quadsoftware.com/, 2008.

[99] A. Torpy. L3dt. http://www.bundysoft.com/L3DT/, 2008.

[100] Garage Games. Torque game engine advanced. http://www.garagegames.com/, 2008.

[101] E.F. Maurina III. The game programmer's guide to torque. AK Peters, Ltd, 2006.

[102] Pandromeda. Mojoworld. http://www.pandromeda.com/, 2006.

[103] Planetside Software. Terragen. http://www.planetside.co.uk/terragen/.

[104] Kamil Stachowski. Terraineer. http://terraineer.sourceforge.net/, 2006.

[105] Digital Element. Worldbuilder. http://www.digi-element.com/wb/, 2006.

[106] S. Schmitt. World machine. http://www.world-machine.com/, 2006.

[107] Autodesk Inc. Autodesk 3ds max. http://usa.autodesk.com/adsk/.

[108] Autodesk Inc. Autodesk maya. http://usa.autodesk.com/adsk/.

[109] M. T. Cook and A. Agah. A survey of sketch-based 3-d modeling techniques. *Interacting with Computers*, 21(3):201–211, 2009.

[110] Luke Olsen, Faramarz F. Samavati, Mario Costa Sousaa, and Joaquim A. Jorge. Sketch-based modeling: A survey. *Computers & Graphics*, 33(1):85 – 103, 2009.

[111] P.A.C. Varley. Automatic creation of boundary-representation models from single line drawings. Technical report, PhD Thesis, University of Wales, 2003.

[112] M. Masry and H. Lipson. A sketch-based interface for iterative design and analysis of 3d objects. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 31, New York, NY, USA, 2007. ACM.

[113] Liangliang Cao, Jianzhuang Liu, and Xiaoou Tang. What the back of the object looks like: 3d reconstruction from line drawings without hidden lines. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(3):507–517, 2008.

[114] S. Lee, D. Feng, and B. Gooch. Automatic construction of 3d models from architectural line drawings. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 123–130, 2008.

[115] D. Hoiem, A. Efros, and M. Hebert. Automatic photo pop-up. *ACM Trans. Graph.*, 24(3):577–584, 2005.

[116] B. Moore. A digital revival of the art of field sketching. in M. Healey and J. Roberts, Engaging students in active learning http://www2.glos.ac.uk/gdn/active/student.htm, 2004.

[117] Koel Das, Pablo Diaz-Gutierrez, and M Gopi. Sketching free-form surfaces using network of curves. In *EUROGRAPHICS Workshop on Sketch Based Interfaces and Modeling*, 2005.

[118] J. H. Gong, H. Zhang, G.F. Zhang, and J.G. Sun. Solid reconstruction using recog-nition of quadric surfaces from orthographic views. *Computer-Aided Design*, 38(8):821–835, 2006.

[119] L.B. Kara and K. Shimada. Sketch-based design of 3d geometry. In *Eurographics Workshop on Sketch-Based Modelling*, pages 59–66, 2006.

[120] M. Prasad and A. Fitzgibbon. Single view reconstruction of curved surfaces. *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2:1345–1354, 2006.

[121] H. Wang and L. Markosian. Sketching free-form shapes. In *Invited Workshop on Pen Computing, Brown University*, 2007.

[122] S.F. Qin, G. Sun, D.K. Wright, S. Lim, U. Khan, and C. Mao. 2D sketch based recognition of 3D freeform shape by using the rbf neural network. In *Eurographics Workshop on Sketch Based Interfaces and Modeling*, pages 119–126, 2005.

[123] M. Kaplan and E. Cohen. Producing models from line drawings of curved surfaces. In *Eurographics Workshop on Sketch Based Interfaces and Modeling*, pages 51–58, 2006.

[124] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Fibermesh: designing freeform surfaces with 3d curves. *ACM Trans. Graph.*, 26(3), 2007.

[125] Johannes Zimmermann, Andrew Nealen, and Marc Alexa. Sketching contours. *Computers & Graphics*, 32(5):486 – 499, 2008.

[126] Fabricio Anastacio, Mario C. Sousa, Faramarz Samavati, and Joaquim A. Jorge. Modeling plant structures using concept sketches. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 105–113, New York, NY, USA, 2006. ACM.

[127] Jamie Wither, Antoine Bouthors, and Marie-Paule Cani. Rapid sketch modeling of clouds. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM)*, 2008.

[128] Nayuko Watanabe and Takeo Igarashi. A sketching interface for terrain modeling. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Posters*, page 73, 2004.

[129] James Gain, Patrick Marais, and Wolfgang Strasser. Terrain sketching. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 31–38, 2009.

[130] Jonathan M. Cohen, John F. Hughes, and Robert C.Zeleznik. Harold: a world made of drawings. In *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 83–90, 2000.

[131] Farès Belhadj. Terrain modeling: a constrained fractal model. In *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 197–204, 2007.

[132] Brennan Rusnell, David Mould, and Mark Eramian. Feature-rich distance-based terrain synthesis. *Visual Computer*, 25:573–579, 2009.

[133] Huamin Wang, Peter J. Mucha, and Greg Turk. Water drops on surfaces. *ACM Trans. Graph.*, 24(3):921–929, 2005.

[134] Jeffrey McConnell. *Computer Graphics: Theory Into Practice.* Jones and Bartlett Publishers, 2006.

[135] Kei Iwasaki, Yoshinori Dobashi, Fujiichi Yoshimoto, and Tomoyuki Nishita. Gpu-based rendering of point-sampled water surfaces. *Vis. Comput.*, 24(2):77–84, 2008.

[136] David O'Brien, Susan Fisher, and Ming C. Lin. Automatic simplification of particle system dynamics. *Computer Animation, 2001. The Fourteenth Conference on Computer Animation. Proceedings*, pages 210–257, 2001.

[137] Odd Erik Gundersen and Lars Tangvald. Level of Detail for Physically Based Fire . In *Theory and Practice of Computer Graphics*, pages 213–220, 2007.

[138] Tor Dokken, Trond R. Hagen, and Jon M. Hjelmervik. The GPU as a high performance computational resource. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, pages 21–26, New York, NY, USA, 2005. ACM.

[139] Nvidia CUDA compute unified device architecture - programming guide, 2007.

[140] Kshitiz Garg, Gurunandan Krishnan, and Shree K. Nayar. Material based splashing of water drops. In *Proceedings of Eurographics Symposium on Rendering*, Jun 2007.

[141] Turner Whitted. The hacker's guide to making pretty pictures. In *SIGGRAPH '85 Image Rendering Tricks course notes*. 1985.

[142] Nelson L. Max. Vectorized procedural models for natural terrain: Waves and islands in the sunset. *SIGGRAPH Comput. Graph.*, 15(3):317–324, 1981.

[143] Lasse Jensen and Robert Goliàs. Deepwater animation and rendering. In *http://www.gamasutra.com*, 2001.

[144] Tsunemi Takahashi, Hiroko Fujii, Atsushi Kunimatsu, Kazuhiro Hiwada, Takahiro Saito, Ken Tanaka, and Heihachi Ueki. Realistic animation of fluid with splash and foam. *Comput. Graph. Forum*, 22(3):391–400, 2003.

[145] Nathan Holmberg and Burkhard C. Wünsche. Efficient modeling and rendering of turbulent water over natural terrain. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 15–22, 2004.

[146] Laszlo Szirmay-Kalos and Tamas Umenhoffer. Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum*, 27(1), 2008.

[147] Yuri Kryachko. Using vertex texture displacement for realistic water rendering. In *GPU Gems 2*, pages 283–294, 2005.

[148] Andrei Tatarinov. Perlin fire, nvidia whitepaper. http://developer.download.nvidia.com/whitepapers/2007/SDK10/PerlinFire.pdf, 2007.

[149] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98, 2009.

[150] Alex Vlachos John R. Isidoro and Chris Brennan. Rendering ocean water. In Wolfgang Engel, editor, *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware, Plano, Texas, 2002.

[151] Yuki Mori and Takeo Igarashi. Plushie: an interactive design system for plush toys. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 45, New York, NY, USA, 2007. ACM.

[152] Shigeru Owada, Frank Nielsen, Kazuo Nakazawa, and Takeo Igarashi. A sketching interface for modeling the internal structures of 3d shapes. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 38, New York, NY, USA, 2007. ACM.

[153] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *CHI '93: Proceedings of the INTERACT '93*, pages 206–213, 1993.

[154] Robert Kosara, Christopher G. Healey, Victoria Interrante, David H. Laidlaw, and Colin Ware. User studies: Why, how, and when? *IEEE Comput. Graph. Appl.*, 23(4):20–25, 2003.

[155] Sarah Tariq. D3D11 tesselation. Game Developers Conference. Session: Advanced Visual Effects with Direct3D for PC, `http://developer.download.nvidia.com/presentations/2009/GDC/GDC09_D3D11Tessellation.pdf`, 2009.