Implementación de Ray Tracing sobre hardware gráfico programable

Abstract

El objetivo de este trabajo es implementar el algoritmo de ray tracing en el hardware gráfico programable.

El método de ray tracing tiene la ventaja de que posibilita obtener imágenes con alto grado de realismo, permitiendo implementar reflexiones mutuas, sombras y distintos métodos de iluminación. En el caso de este trabajo, se adaptó este método, tradicionalmente recursivo, al modelo computacional de procesamiento de flujos.

Los procesadores gráficos están diseñados para renderizar utilizando el método scan-line. Pese a esta especificidad computacional, es posible utilizar dichos procesadores para realizar otro tipo de cálculos, como en el caso de este trabajo en el que se utiliza para procesar flujos. Además este hardware tiene características que se pueden aprovechar para reducir los tiempos de renderizado, como el procesamiento en paralelo y las operaciones vectoriales. Adicionalmente, se está desarrollando muy rápidamente, mejorando su performance y ampliando sus funcionalidades.

The aim of this work is to implement the ray tracing algorithm on programmable graphics hardware.

The ray tracing algorithm is one of the preferred strategies for obtaining highly realistic images, because the computational model is well suited to implement mutual reflections, shadows and many other sophisticated illumination methods.

Graphics processors are an emerging technologic facility, which is revolutionizing the world of computer graphics and game development. This hardware has features that can be taken advantage of to dramatically reduce the rendering times, given its parallel processing and specialized vector arithmetic capabilities. However, graphics hardware was primarily designed for scan-line rendering. Besides this specificity, it is possible to use these processors in order to do other types of computation. To achieve the implementation of ray tracing in this context, in our work the intrinsically recursive ray tracing algorithm was adapted to a stream processing model that can be implemented in graphics hardware.

Director:
Claudio Delrieux

Tesistas:
Fal, Carlos Leandro LU 137/99
Vasconcelos, Guillermo LU 309/99

Fecha
mayo de 2006

1 Indice

2 Introducción

Hasta hace algunos años, era impensable obtener aplicaciones interactivas con imágenes fotorrealistas generadas en tiempo real, utilizando computadoras convencionales. Por lo tanto, a la hora de diseñar aplicaciones gráficas, se debía elegir entre esas dos características.

En los últimos años, con el advenimiento del hardware gráfico programable, esta brecha entre el fotorrealismo y la interactividad se fue acortando. Este hardware permite implementar distintos algoritmos de renderizado y modelos de iluminación directamente en la placa gráfica. Al ser un hardware especializado en la generación de imágenes y tener un modelo de procesamiento en paralelo (SIMD), realiza las operaciones gráficas con mayor velocidad que la CPU.

Para generar imágenes fotorrealistas, uno de los métodos más difundidos y utilizados es el ray tracing. Mediante este método se pueden generar imágenes con iluminación global y sombreado (shading) de alto realismo. El sombreado determina las tonalidades con las que se representa un objeto, teniendo en cuenta la interacción del mismo con las fuentes de iluminación y otros objetos de la escena. El cálculo exacto de esta interacción no es computacionalmente posible por su complejidad, por lo que se utilizan aproximaciones llamadas modelos de iluminación. El método de ray tracing permite utilizar muchos de estos modelos de iluminación para obtener la calidad de imagen deseada. Hasta hace algún tiempo, sólo era posible hacer ray tracing en tiempo real utilizando hardware específico como sistemas multiprocesadores.

Nuestro objetivo a través de este trabajo es implementar versiones del algoritmo de ray tracing sobre el hardware gráfico programable. Si bien este hardware está diseñado específicamente para renderizar mediante el método tradicional de scan-line, provee suficiente poder de programación para implementar el ray tracing. Analizaremos también si con el estado actual de la tecnología es posible ejecutar el algoritmo a suficiente velocidad como para desarrollar aplicaciones interactivas que muestren escenas interesantes en un tiempo aceptable.

Nuestra presentación consiste en una implementación del método básico de ray tracing aprovechando las capacidades del hardware gráfico, así como variantes del mismo. También realizamos una comparación en cuanto a la performance con respecto al ray tracing implementado en la CPU. Luego analizamos los resultados obtenidos y estudiamos posibles mejoras en la implementación. También mostramos las conclusiones obtenidas sobre el estado actual de esta tecnología y la distancia a la que nos encontramos de alcanzar la meta del fotorrealismo en tiempo real en aplicaciones prácticas.

3 Conceptos preliminares

Este trabajo se basa en la técnica del ray tracing, y utiliza además la capacidad de programación de las nuevas GPUs, a través del lenguaje CG.

A continuación damos una breve explicación sobre cada uno de estos elementos, mostrando las características principales de ellos.

3.1 Ray Tracing

Como mencionamos anteriormente, el método de ray tracing es uno de los métodos de renderizado más utilizados. Esto se debe a dos esenciales características:

- Realismo: Las imágenes generadas mediante este método tienen una carga fotorrealista muy importante debido a las capacidades para representar reflexiones especulares mutuas, sombras, adaptación a distintos modelos de iluminación y posibilidades para expresar otras reflexiones e inclusive refracciones con gran fidelidad.
- Simpleza: El concepto que abarca el ray tracing es muy fácil de comprender. Su implementación es relativamente sencilla. Provee la capacidad de utilizar el modelo de iluminación más conveniente, y facilidades para modificar los parámetros que son utilizados en los cálculos de las distintas características de la imagen.

En contraposición con estas ventajas, la principal desventaja que posee este método (y la cual estamos abordando en nuestro trabajo) es la complejidad de los cálculos que se deben realizar para renderizar cada pixel y, por ende, el tiempo total en el que se genera una imagen completa. En este trabajo no nos enfocamos en reducir la complejidad del cálculo para cada punto de la imagen, sino en aprovechar las características que brindan los nuevos procesadores de las placas de video para poder realizar estos cálculos con mayor velocidad.

3.1.1 Descripción del método

El concepto básico de ray tracing consiste en posicionarse en el lugar del observador, enviar rayos que atraviesen la pantalla y calcular las intersecciones de cada rayo con los distintos objetos que se encuentran en la escena. Si un determinado rayo interseca algún objeto, el pixel por el que el rayo entró a la imagen debe pintarse del color de dicho objeto en el contexto de iluminación global que corresponde al punto de intersección del rayo con el objeto.

A este concepto básico debemos agregarle las características de iluminación de la escena, y las características reflexivas y refractivas del material del objeto. También debemos tener en cuenta para la iluminación, sombra, reflexión y refracción los demás objetos que se encuentran en la escena y que interactúan con el objeto en cuestión.

En la implementación más simple, cada pixel es atravesado por un solo rayo que pasa por su centro. Esto puede generar bordes dentados en las siluetas de los objetos dibujados, y pixels incorrectamente coloreados en aristas y rincones, debido a que el

proceso de renderizado es esencialmente un muestreo, y por lo tanto está sujeto a problemas de aliasing cuando la frecuencia espacial de la escena está por encima de las condiciones establecidas por el teorema del muestreo. Para mitigar este problema se pueden utilizar diversas técnicas de antialiasing que en general consisten en lanzar más de un rayo por pixel y calcular el color final a partir de una combinación de los resultados de estos rayos.

De acuerdo con el modo en el que se dibuja la escena, vemos que este es un método que realiza el renderizado pixel por pixel. Las placas gráficas están diseñadas para el rendering scan-line, que se basa en renderizar cara por cara. Para lograr esto, se toman las primitivas que forman la escena y se las multiplica por una matriz de transformación (compuesta por una secuencia de transformaciones apiladas), que lleva sus coordenadas en el mundo a coordenadas en la pantalla. Este es el punto en el que radica la mayoría de las ventajas y desventajas que tiene el ray tracing con respecto a los demás métodos, y tiene un peso muy importante en el tiempo de renderizado de la imagen.

3.2 Placas de video programables

Desde hace algún tiempo, los procesadores de las placas de video (GPU) están adquiriendo mayor velocidad y poder de procesamiento. Con la inclusión de los programas de vértices y fragmentos en las GPU y el desarrollo constante de los mismos, el poder de procesamiento de las GPU dio un salto cualitativo muy importante.

Estos dos procesadores, junto con el scan converter de las placas, están diseñados para ejecutar estos programas con gran eficiencia, por ser un procesado paralelo con fines específicos (por oposición a la CPU, que es un procesador serial de propósito general). Por lo tanto, los tiempos de renderizado son incomparablemente más rápidos.

Las instrucciones que poseen las GPU están orientadas a la generación de imágenes utilizando el método scan-line, es decir, el vínculo entre el procesado de vértices y el procesado de fragmentos (pixels) es generado mediante el scan converter, el cual ensambla los vértices en caras poligonales, y realiza la interpolación bilineal de las coordenadas resultantes de los pixels a partir de las coordenadas entrantes de los vértices. Este proceso, el cual es muy costoso en arquitecturas seriales, se realiza en un solo paso de cómputo, y además como subproducto permite realizar otras interpolaciones bilineales de atributos de los vértices (profundidad, color, texcoords, etc.).

Pese a esta especificidad computacional, es posible utilizar el procesador para realizar otro tipo de cálculos, como en el caso de este trabajo en el que se utiliza para procesar streams (flujos de datos). Para ello es necesario revisitar el modelo computacional de las GPU, abstrayéndose de su origen y propósito en la génesis de su arquitectura, y viendo el procesamiento como una tarea genérica. En particular, trataremos de considerar a la GPU como un procesador de flujos de datos (stream processing). Los programas de vértices se ejecutan para todos los vértices del modelo, mientras que los de fragmento se ejecutan para todos los pixels de la imagen resultante. En estos últimos es donde se debe implementar el método de ray tracing, debido a que su cómputo se realiza pixel a pixel, y se utiliza el procesador de vértices como

implementación de una estructura que permite preparar el cómputo de una manera satisfactoria.

3.3 Stream Processing

El procesamiento de streams (o flujos de información) es una manera de estructurar un programa en forma de pipeline. Un procesador de streams se compone de distintas etapas de procesamiento llamadas kernels y "caminos" por los que pasan los flujos de información. La información ingresa al sistema, y va siendo procesada y transformada sucesivamente por los distintos kernels. Cada kernel realiza una tarea específica, y siempre la misma sobre todos los flujos de información.

Este modelo se puede implementar en las GPU en forma eficiente debido a que tanto los kernels como los procesadores de fragmentos pueden realizar una operación sobre muchos datos en paralelo. Los flujos de información no tienen un mapeo directo con la GPU, pero se pueden almacenar los valores intermedios en texturas y pasar estas texturas a los diferentes programas de fragmento para implementar el camino de los datos entre los diferentes kernels.

3.4 CG

Cg (C for Graphics) es un lenguaje creado por Nvidia para facilitar la programación de las unidades de vértices y fragmentos de las GPU. Provee funciones de alto nivel y un grado de abstracción que permite que el mismo programa sea compilado y pueda ejecutarse en distintas GPUs, siempre que cuenten con los requerimientos mínimos.

Cg provee funciones para trabajar con distintos tipos de datos, como escalares, vectores y matrices, con cierta facilidad. Brinda además funciones para acceder a las texturas que se encuentran en la memoria de la GPU.

Junto con el compilador de Cg, Nvidia desarrolló también una API llamada CG Runtime que se utiliza de interfaz entre los programas de la GPU y las APIs 3d que son ejecutadas en la CPU (OpenGl o DirectX).

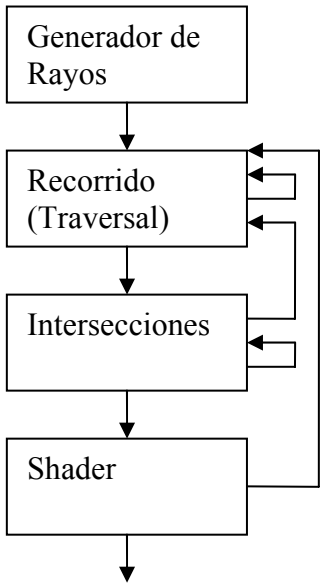
4 Descripción de la solución

Debido a que el hardware gráfico programable está diseñado para optimizar la ejecución del método scan-line, para implementar el método de ray tracing hay que mapearlo a un modelo que sea ejecutable por el hardware. En este trabajo se propone pensar al ray tracing como un pipeline, con distintas etapas por las que va pasando cada pixel hasta que queda definido su color final.

Para poder implementar el método, son necesarias algunas suposiciones iniciales para simplificar el problema y hacerlo más manejable por el hardware. El método implementado utiliza como única primitiva el triángulo, debido a que es la primitiva más utilizada para describir objetos y la que se suele usar en el método de scan-line. Además, el hardware gráfico está organizado para polígonos convexos, cuanto más simples mejor.

Además, para que los métodos de ray tracing tengan un rendimiento aceptable, se deben organizar los objetos en una estructura aceleradora, de manera que no sea necesario comprobar las intersecciones de los rayos con todos los objetos de la escena, sino con un subconjunto relevante. La implementación utiliza como estructura aceleradora una grilla uniforme, debido a que el recorrido de esta grilla es implementable por hardware con relativa sencillez y da en general buenos resultados de performance (como toda estructura aceleradora, su performance depende de la escena en particular). Cada uno de los cubos definidos por esta grilla es llamado voxel.

En el siguiente cuadro se muestran las etapas del pipeline y los posibles flujos de datos entre las mismas. Es importante considerar que la comunicación entre kernels se realiza enviando todos los pixels en paralelo a través de una textura. Esto es lo definitivamente valioso del desarrollo de programas en GPU, dado que el modelo conceptual es un modelo pixel a pixel, mientras que el modelo computacional de ejecución realiza todas las operaciones en paralelo.



Se genera un rayo (Generador de Rayos), se busca su entrada a la estructura aceleradora y se atraviesa la estructura hasta encontrar un voxel que contenga

triángulos (recorrido o traversal). Se interseca el rayo con todos los triángulos del voxel (Intersecciones), y si se halló una intersección se pasa el rayo al Shader. Si no interseca, se pasa de nuevo a Traversal para que siga recorriendo la estructura. Una vez en Shader, se calcula el color del rayo y, si corresponde, se generan los rayos de reflexión y sombras.

4.1 Generador de rayos

El generador de rayos es el kernel más simple de los cuatro, tanto en sus funciones como en su implementación. Genera un rayo por pixel de la imagen final en base a la posición y orientación de la cámara y guarda la información de los rayos generados en texturas. Se ejecuta una vez al comienzo del ciclo de renderizado, para todos los fragmentos de la escena. Eventualmente, si por alguna razón se dispusiera un esquema diferente de muestreo, entonces habría que modificar la programación de este kernel. Situaciones donde puede ocurrir un muestreo diferente son, por ejemplo, realizar varias muestras ponderadas (oversampling) por pixel para luego aplicar anti-aliasing, realizar subdivisiones adaptativas para optimizar el cómputo (undersampling adaptativo), etc.

4.2 Traversal

La etapa de traversal cumple dos funciones diferenciadas. La primera de estas funciones consiste en encontrar los voxels de entrada de los rayos en la estructura aceleradora. Esto implica extender los rayos hasta que llegan a los bordes de la grilla, y encontrar el voxel que corresponde a esta intersección. Esta tarea se realiza justo después del generador de rayos, y una sola vez por ciclo de renderizado.

La segunda función que tiene este kernel es la de recorrer la grilla, atravesando los sucesivos voxels en la dirección de los rayos, hasta encontrar voxels que contengan triángulos o llegar al final de la grilla. El recorrido de la grilla se realiza utilizando un algoritmo del tipo DDA 3D, donde cada iteración del algoritmo se corresponde con una ejecución del kernel. Esto significa que por cada ejecución del kernel, el rayo avanza un voxel solamente. Durante la iteración se decide si el próximo kernel que se debe ejecutar para el rayo en cuestión es el detector de intersecciones, de nuevo el traversal o ninguno, si el rayo sale fuera de la grilla.

4.3 Detector de intersecciones

Esta etapa recorre los triángulos de cada voxel buscando intersecciones entre los mismos y el rayo que se está analizando. En cada ejecución de este kernel se evalúa la intersección del rayo con un triángulo del voxel en particular. Cuando ya se evaluó la intersección con todos los triángulos del voxel, se decide si el próximo kernel a ejecutar es el traversal (si no hubo ninguna intersección) o el shader.

En el caso en que dos o más triángulos son intersecados por el rayo, se toma como triángulo de intersección al que sea alcanzado por el rayo a menor distancia del origen del mismo. También puede suceder que el punto de intersección del rayo con un triángulo caiga fuera del voxel que se está analizando. En este caso esta intersección no se tomará como válida en este momento y sólo será válida si el rayo llega hasta el voxel al que pertenece dicho punto de intersección.

4.4 Shader

Esta etapa tiene como función generar el color de los pixels de la imagen final. Para esto toma el triángulo intersecado por el rayo, y calcula la iluminación utilizando las propiedades de ese triángulo y su normal interpolada en el punto de intersección teniendo en cuenta también la posición de las luces. El kernel puede utilizar diferentes modelos de iluminación para obtener el color del pixel en la imagen final.

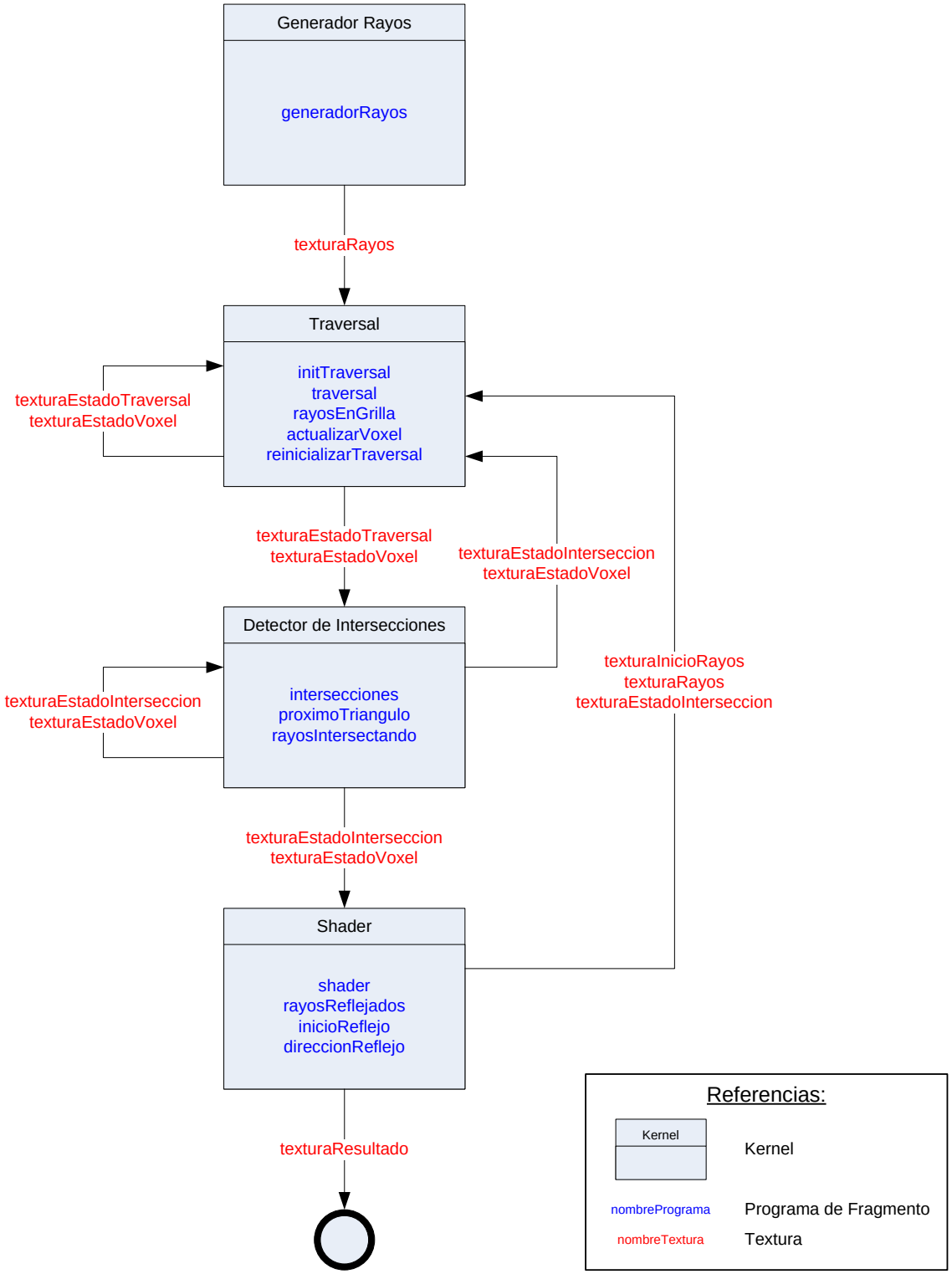
Adicionalmente este kernel puede generar nuevos rayos de sombras o reflexiones que son pasados al kernel traversal para continuar el cálculo. El resultante de estos nuevos rayos es combinado con el original para obtener el color final para ese pixel.

4.4.1 Interreflexión

Para implementar interreflexión en el ray tracing tradicional se utiliza recursión. Como el modelo de ejecución de las placas gráficas no permite recursión, esto se implementó volviendo atrás en el pipeline, redefiniendo el origen y la dirección de cada rayo que interseque a una cara reflectiva, en base al punto y al ángulo de intersección. El resultado final de cada pixel, se obtiene mediante una combinación de los resultados parciales ponderados por los coeficientes de reflexión de las distintas superficies intersecadas durante toda la ejecución del pipeline.

5 Implementación

5.1 Componentes del Pipeline



5.2 Estructura aceleradora

Como mencionamos anteriormente, para mejorar la performance del programa trabajamos con una estructura aceleradora que se carga off-line (se carga una vez al comenzar el programa, y permanece invariante durante toda la ejecución). La finalidad de esta estructura es analizar solamente combinaciones rayo-triángulo que tienen chances de intersecarse, y no calcular intersecciones de todos los rayos con todos los triángulos. Existen distintos tipos de estructuras aceleradoras. En este caso se emplea una grilla tridimensional uniforme que secciona la escena en **n** cubos, llamados voxels, y ubica dentro de cada cubo los triángulos que tienen alguna parte en su interior. De esta manera, mientras los rayos van atravesando la escena, sólo se realiza el cálculo de intersección con los triángulos que pertenecen al cubo en el que cada rayo se encuentra.

Como el costo del armado de esta estructura es muy alto, es necesario armarla antes de comenzar con el renderizado. Por esta razón la aplicación sólo muestra escenas estáticas. De otro modo se debería rearmar la estructura ante cada refresco de la pantalla.

5.2.1 Armado de la estructura

Si bien esta estructura acelera el proceso de renderizado de manera notable, el tiempo que se consume durante el armado de la misma es muy alto, sobre todo cuando hablamos de escenas con gran complejidad. Por esta razón, optamos por almacenar los datos que se obtienen en el armado de esta estructura en un archivo que llamamos archivo caché. De esta manera, la próxima vez que se ejecute el programa con el mismo modelo y tamaño de grilla, se leerá el archivo caché en lugar de volver a generar la estructura a partir del modelo y efectuando todos los cálculos.

Puntualmente, el archivo caché es un archivo binario que contiene los datos para las texturas que contienen los vértices, las normales, los colores y las listas de los triángulos que tienen incidencia en cada voxel. Estos datos se calculan a partir del modelo, teniendo en cuenta la cantidad de voxels de la estructura aceleradora.

Cuando se realiza la carga del archivo, se les asignan los valores a las variables y las texturas directamente. Esto agiliza en gran medida la carga inicial de los datos necesarios para renderizar la escena.

5.3 Texturas empleadas

Para almacenar datos durante la ejecución del programa se utilizan texturas que son modificadas y accedidas desde OpenGL y CG. De acuerdo a la persistencia que tienen los datos que se utilizan a lo largo de la ejecución del programa, podemos clasificarlos en dos tipos:

5.3.1 Texturas estáticas

Se calculan al comienzo de la ejecución del programa y perduran invariantes hasta el final de la misma. Dentro de estas se encuentran las texturas que representan a la escena. Son las siguientes:

Nombre de la textura	Contenido
texturaVertices	Arreglo de 3 texturas con los 3 vértices de cada triangulo. Se las accede con un ID de triángulo **(id)**, y cada una de estas texturas tiene la posición (X,Y,Z) de un vértice del triángulo **id.**
texturaColores	Se accede con el ID de triángulo **(id)** y se obtiene el color (R, G, B) y el coeficiente de reflexión del triángulo **id.**
texturaNormales	Arreglo de 3 texturas con las normales de los 3 vértices de cada triangulo. Se las accede con un ID de triángulo **(id)**, y cada una de estas texturas tiene la normal (X,Y,Z) de un vértice del triángulo **id**
texturaTriangulos	Contiene los números de triángulos que se encuentran en cada voxel. Los voxels están uno atrás del otro y separados por un **-1**. Se indexa con la **texturaTriangulosEnVoxel**
texturaTriangulosEnVoxel	Contiene el puntero al comienzo de cada voxel en la **texturaTriangulos**. Para hacer la transformación de 3D a 2D, para el voxel X,Y,Z, tomamos X*tamGrillaY+Y, Z

5.3.1.1 Interpolación de normales

Debido a que el formato *ply* que utiliza la aplicación no contiene las normales, éstas deben ser calculadas. Para esto primero se calculan las normales para cada cara, y luego se le asigna a cada vértice el promedio de las normales de todas las caras que lo comparten. Para realizar este cálculo se emplea una estructura de datos que se compone de una lista de vértices, y asociado a cada uno la lista de caras de las cuales el vértice forma parte. De esta manera se pueden obtener rápidamente las normales que hay que promediar para obtener la normal correspondiente al vértice. Este es el dato que se almacena en *texturaNormales*.

5.3.2 Texturas dinámicas

Varían durante el renderizado de la imagen, y llevan los estados en los que se encuentran los distintos rayos. Dentro de esta categoría se encuentran todos los datos que son modificados por la ejecución de los distintos kernels.

Nombre de la textura	Contenido
texturaRayos	Guarda en X,Y,Z la dirección de cada rayo.
texturaEstadoTraversal	En X,Y,Z guarda el voxel al cual apunta el rayo, en W va el coeficiente por el cual se multiplica la dirección del rayo para entrar al voxel actual que se está analizando.
texturaEstadoVoxel	En X está la posición a la que apunta en texturaTriangulos. En W está el número de triángulo activo para el rayo.
texturaEstadoInterseccion	En X está el coeficiente por el cual se multiplica la dirección del rayo para alcanzar el punto de intersección y en W está el triángulo con el cual se produce la intersección.
texturaResultado	Contiene el color actual de cada pixel. En α tiene el coeficiente que aporta el color actual al color final de la imagen para cada pixel.

5.4 Interreflexión

Se produce una reflexión cuando un rayo interseca un objeto cuyo coeficiente especular es mayor a 0. Esta reflexión genera un rayo con origen en el punto de intersección y reflejado con respecto a la normal del objeto en dicho punto.

Para calcular el color final del pixel, se combinan los resultados parciales de los rayos correspondientes a dicho pixel (el original y sus reflejos), según la siguiente fórmula:

$$\text{Color Final} = C_1 (1 - \alpha_1) + \alpha_1 (C_2 (1 - \alpha_2)) + \alpha_1 \cdot \alpha_2 (C_3(1 - \alpha_3)) + ...$$

donde:

$\mathbf{C_i}$ es el color del i-ésimo objeto intersecado por el rayo
$\boldsymbol{\alpha_i}$ es el coeficiente especular del i-ésimo objeto intersecado por el rayo.

El cálculo de los rayos reflejados se realiza una vez que todos los rayos terminaron su recorrido por la grilla, ya sea intersecando con algún objeto o saliendo de la misma. Luego se realiza una nueva iteración hasta que estos nuevos rayos terminen su recorrido. Esta operación se repite hasta que ningún rayo sea reflejado, o se alcance el número máximo de iteraciones establecido.

5.5 Programas de fragmento

5.5.1 Generador de Rayos

5.5.1.1 generadorRayos

Modifica: *texturaRayos*
Se ejecuta solamente una vez por ciclo de renderizado.
La función de este programa es generar la dirección inicial de los rayos. Esta dirección es obtenida como la normalización del vector comprendido entre las coordenadas de cada pixel con z = 0 y el punto (0, 0, -1).

5.5.2 Traversal

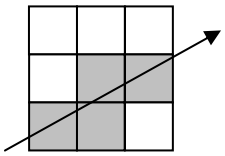
5.5.2.1 initTraversal

Modifica: *texturaEstadoTraversal*
Se ejecuta solamente una vez por ciclo de renderizado.
Calcula el voxel de entrada de cada rayo en la estructura aceleradora. Para esto calcula el mínimo factor por el que se multiplica a la dirección del rayo para alcanzar un borde de la grilla. Con esto calcula el punto por el que ingresa a la grilla, y a que voxel pertenece.

5.5.2.2 traversal

Modifica: *texturaEstadoTraversal*
El objetivo de este programa es hallar el próximo voxel por el que tiene que pasar el rayo.
Para esto se utiliza un algoritmo DDA modificado para que considere todos los voxels

por los cuales pasa el rayo. A continuación, se muestra una figura que ejemplifica el funcionamiento de este programa de fragmento.



5.5.2.3 rayosEnGrilla

Modifica: *cantActivos*
Calcula la cantidad de rayos que están activos, es decir, que no encontraron todavía un triángulo y están en algún voxel de la grilla.

5.5.2.4 actualizarVoxel

Modifica: *texturaEstadoVoxel*
Asigna el primer triángulo con el cual se tiene que hacer la intersección para cada rayo, de acuerdo al voxel en que se encuentra. Esta información es obtenida de *texturaTriangulosEnVoxel* y *texturaTriangulos*.

5.5.2.5 reinicializarTraversal

Modifica: *texturaEstadoTraversal*
Luego de que se modifica el inicio y la dirección de algún rayo, este programa actualiza el factor por el cual se debería multiplicar la dirección de dicho rayo para entrar al voxel actual.

5.5.3 Detector de Intersecciones

5.5.3.1 intersecciones

Modifica: *texturaEstadoInterseccion*
Analiza si existe intersección entre cada rayo y el triángulo que le corresponde en la iteración actual (este triángulo es obtenido de *texturaEstadoVoxel*). Si se produce una intersección analiza también si ésta se produce más cerca del inicio del rayo que la intersección más cercana calculada hasta el momento, y si se produce dentro del voxel que se está analizando.

5.5.3.2 proximoTriangulo

Modifica: *texturaEstadoVoxel*
Obtiene el próximo triángulo a analizar para encontrar intersecciones con cada rayo. Si no hay más triángulo en el voxel para un rayo, éste queda marcado como listo para traversal.

5.5.3.3 rayosIntersectando

Modifica: *cantIntersectando*

Calcula la cantidad de rayos que tienen al menos un triángulo pendiente para intersectar en el voxel actual.

5.5.4 Shader

5.5.4.1 shader

Modifica: *texturaResultado*
Calcula el color que se debe mostrar en cada pixel de la imagen final. Dependiendo del coeficiente de reflexión del objeto con el que interseca el rayo, el valor obtenido puede ser sólo un porcentaje del color final.
Para calcular el color obtiene la normal en el punto de intersección interpolando las normales de los vértices y calcula el ángulo de incidencia de la luz en base a esta normal.

5.5.4.2 rayosReflejados

Modifica: *cantReflejados*
Calcula la cantidad de rayos que han sido reflejados en la ejecución del shader. Es decir, los que intersecaron con un objeto reflectante.

5.5.4.3 inicioReflejo

Modifica: *texturaInicioRayos*
Establece el punto de inicio de los rayos reflejados, utilizando el punto de intersección anterior.

5.5.4.4 direccionReflejo

Modifica: *texturaRayos*
Establece la dirección de los rayos reflejados, utilizando la dirección anterior y la normal en el punto de intersección.

5.5.5 Auxiliares

5.5.5.1 mostrarTextura

Muestra por pantalla el contenido de una textura. Se utiliza para mostrar la *texturaResultado*, una vez que está completamente calculada.

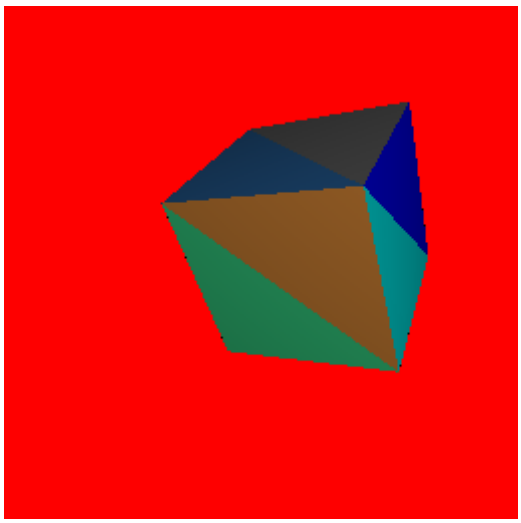
6 Resultados

En las siguientes imágenes se ven los resultados obtenidos al ejecutar la aplicación para diferentes escenas. Junto a las imágenes se incluye información relevante a la misma, como el tiempo de renderizado y la cantidad de divisiones de la grilla, por ejemplo. También se muestra el resultado de la comparación de algunos aspectos que se modifican al variar diferentes parámetros o condiciones de ejecución. Todas las imágenes fueron renderizadas en 256 x 256 pixels.

6.1 Imágenes renderizadas

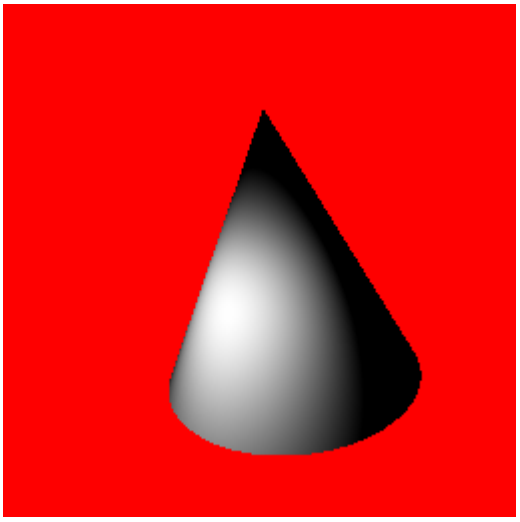
6.1.1 Sin Reflexión

En las siguientes tres escenas se ve el resultado del renderizado sin reflexión y con iluminación difusa. Las escenas están ordenadas por orden de complejidad creciente.



Escena	cubo3
Tiempo (ms)	370
Grilla	3x3x3
Caras	12

Escena	Cone
Tiempo (ms)	3050
Grilla	5x5x5
Caras	100

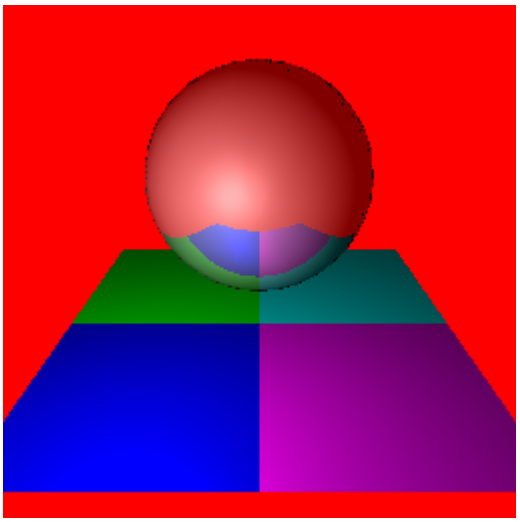


Escena	vaca
Tiempo (ms)	10095
Grilla	20x20x20
Caras	5804

6.1.2 Con reflexión

La siguiente escena tiene una superficie con un coeficiente de reflexión definido. Se puede ver el reflejo del cuadrado en la esfera, deformado por la curvatura de la superficie de la esfera.

Escena	sphereRefl
Tiempo (ms)	7852
Grilla	15x15x15
Caras	848
Iteraciones	3



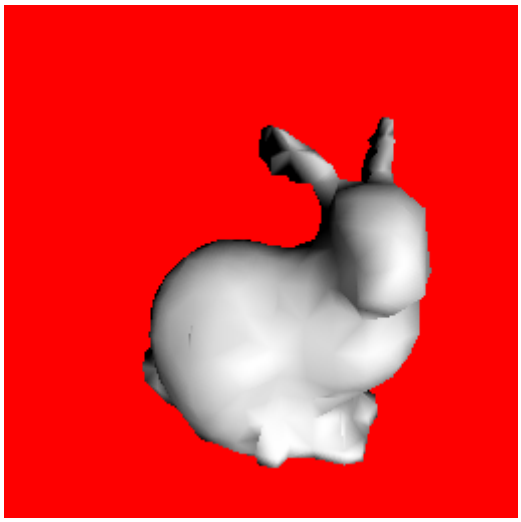
6.2 Comparaciones

Las series de imágenes que se encuentran a continuación están relacionadas y varían entre sí en algún parámetro. Lo que se muestra es como varía el resultado de acuerdo a ese parámetro. En algunos casos lo que varía es el tiempo mientras que en otros casos cambia la imagen final o ambos.

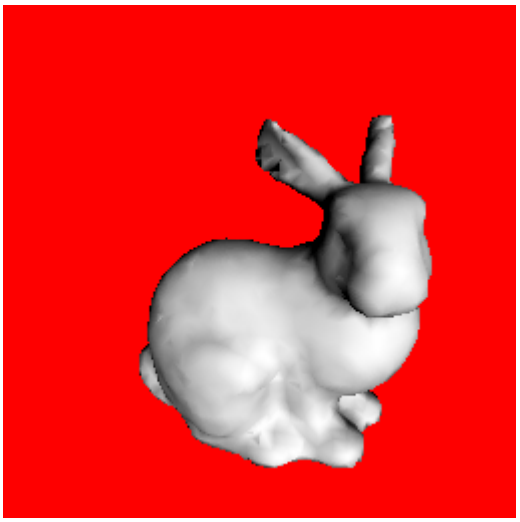
6.2.1 Complejidad de modelos

En esta comparación se utilizan dos objetos con la misma forma básica, pero con distinto nivel de detalle en el modelo. Se ve como la variación en la cantidad de caras no es proporcional con el tiempo que insume el renderizado, pudiendo aumentarse la

complejidad del modelo sin que se torne impracticable. En este caso la cantidad de caras aumenta más de 4 veces, mientras que el tiempo insumido no llega al doble.



Escena	bunny
Tiempo (ms)	3485
Grilla	15x15x15
Caras	948



Escena	bunny3
Tiempo (ms)	6068
Grilla	15x15x15
Caras	3851

6.2.2 Tipos de shaders

En las dos series de imágenes siguientes se ve el resultado de renderizar las escenas con distintos tipos de shaders. Estos shaders son fácilmente intercambiables, y no generan una diferencia apreciable en el tiempo de renderizado. Cuando se agrega el componente especular, se ven zonas de brillo especular que le dan apariencia metálica a la superficie. Con el agregado de la atenuación se le da más realismo a la fuente de luz constante, suavizando además los brillos especulares.

Difuso

Difuso + especular

Difuso + especular + atenuación

Escena	sphere
Tiempo (ms)	2800
Grilla	15x15x15
Caras	840

Difuso

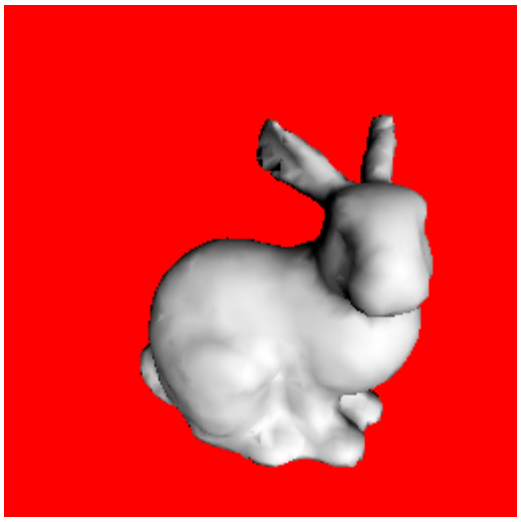
Difuso + especular

Difuso + especular + atenuación

Escena	beethoven
Tiempo (ms)	7990
Grilla	15x15x15
Caras	5030

6.2.3 Factores de intersección

En el siguiente cuadro se muestran los resultados de variar el factor de intersecciones. Este parámetro se compara con el factor entre la cantidad de rayos que están activos en el paso de intersección contra la cantidad total de rayos activos. Con esto se decide si el siguiente paso es de intersección o de traversal. La finalidad de este cociente es evitar la generación de un retraso en toda la ejecución a causa de algunos voxels que contengan una mayor cantidad de triánguos que los demás. El valor ideal para este parámetro depende de cada escena.



Escena	bunny3		
Grilla	15x15x15		
Caras	3851		
Factor Intersección	0.1	0.3	0.5
Tiempo (ms)	6068	5075	5729
Intersecciones	262	180	165
Traversals	35	50	77

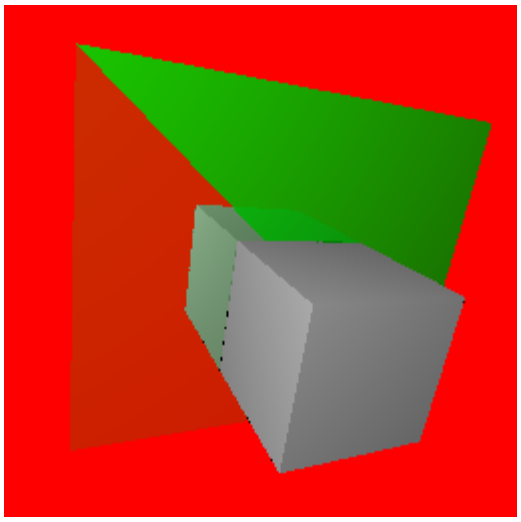
6.2.4 Tamaño de grilla

En el siguiente cuadro se ve como cambia el tiempo en una escena compleja con muchas caras cuando se utiliza una grilla con mayor o menor granularidad. Ante una menor cantidad de celdas en la grilla se necesitan más pasos de intersección para renderizar la imagen. Cuando aumenta la cantidad de celdas se reduce el número de pasos de intersección pero se aumenta el de pasos de traversal. El valor ideal de tamaño de la grilla depende de cada escena. En este caso la grilla de 30x30x30 dio los mejores resultados.

Escena	vaca				
Caras	5804				
Grilla	15x15x15	20x20x20	25x25x25	30x30x30	35x35x35
Tiempo (ms)	14401	10095	9093	8131	9394
Intersecciones	657	411	336	255	220
Traversals	55	76	98	112	164

6.2.5 Coeficiente de reflexión

En la siguiente imagen se puede apreciar la variación del coeficiente de reflexión en la cara del cuadrado. En el triángulo inferior el coeficiente es de 0.9, con lo cual refleja casi totalmente el cubo. El triángulo superior en cambio tiene un coeficiente de 0.2 por lo que refleja menos de la imagen y se ve más su color original.

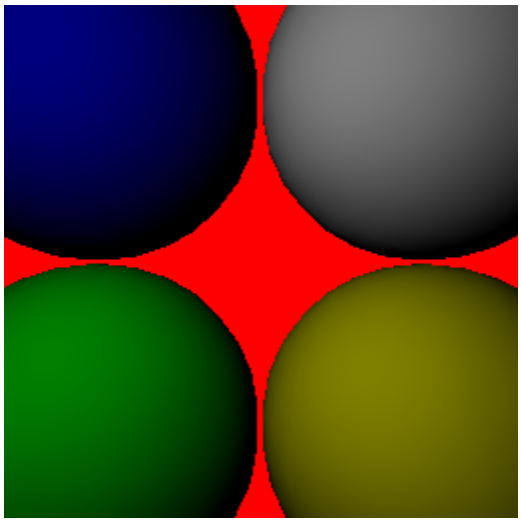


Escena	cuboRefl
Tiempo (ms)	2504
Grilla	5x5x5
Caras	14

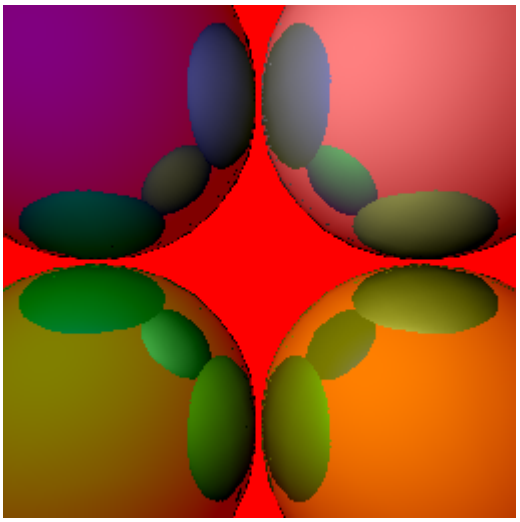
Coeficientes de Reflexión: 0.9 y 0.2

6.2.6 Iteraciones de reflexión

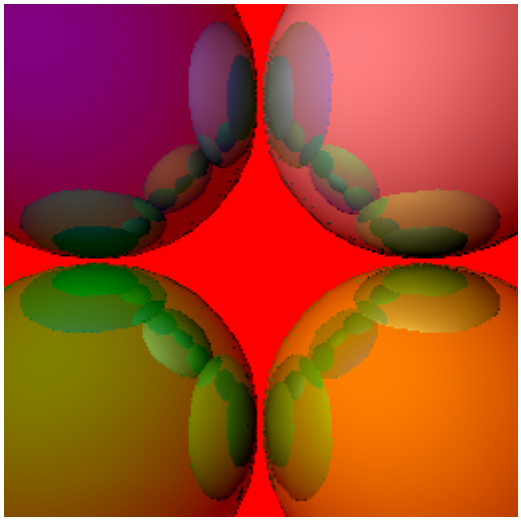
En la siguiente serie de imágenes se ve como influye la cantidad de iteraciones de reflexión cuando se sitúan objetos que se reflejan entre sí. La primera imagen no tiene reflejo. En la segunda se muestran los reflejos luego de una iteración de los rayos reflejados y en la tercera se agrega la siguiente iteración. Esto es el reflejo del reflejo.



Una iteración



Dos iteraciones



Tres iteraciones

Escena	4spheres		
Grilla	15x15x15		
Caras	3360		
Iteraciones	1	2	3
Tiempo (ms)	5167	10656	15563

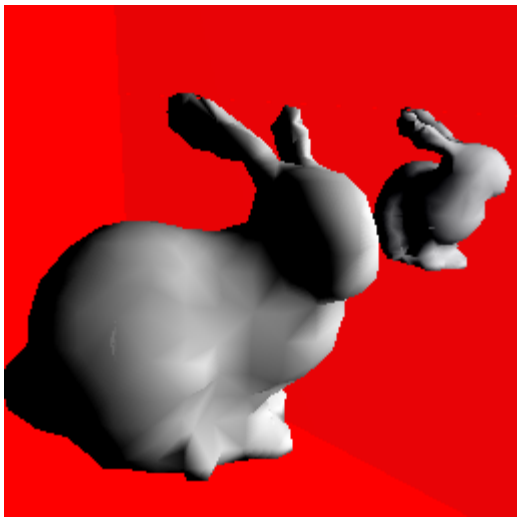
6.2.7 GPU 5700 – CPU – GPU 6600

En el siguiente cuadro se ve una comparación de los tiempos de ejecución de algunas escenas en tres ambientes diferentes. En la primera columna se ven los tiempos ejecutando en una GPU Nvidia Geforce FX 5700, que es la GPU de base de la implementación y en donde están ejecutadas todas las pruebas anteriores. En la segunda columna está el tiempo para la implementación que se ejecuta en la CPU (Pentium 3 600 Mhz). En la tercera columna está el tiempo de la misma escena en una GPU Nvidia Geforce FX 6600.

Escena	**GPU 5700**	**CPU**	**GPU 6600**
Sphere	2800	2864	500
Bunny3	5075	2013	1079
BunnyRefl	10355	2504	1788

En la escena *Bunny3*, la CPU tuvo mejores tiempos que la GPU debido a que muchos rayos caen fuera del objeto y son rápidamente descartados. En la GPU en cambio, los pasos se efectúan siempre para todos los rayos. En la GPU 6600, que cuenta

con ejecución condicional en los programas de fragmento, estos rayos tienen menor peso dado que si bien se procesan, se requieren pocas instrucciones para ello. En la escena *Sphere*, en cambio, el tiempo de la GPU fue menor al de la CPU debido a que la mayoría de los rayos intersecan al objeto. En el caso de la escena *BunnyRefl*, al poseer reflexión, la CPU tiene ventaja debido a que los rayos que no se reflejan no se siguen calculando, mientras que en la GPU dichos rayos siguen siendo pasados como parte del stream. Esto tiene mucho impacto en la GPU 5700 e impacto moderado en la 6600 debido a que estos rayos son calculados rápidamente.



Escena	bunnyRefl
Tiempo (ms)	10355
Grilla	30x30x30
Caras	950
Iteraciones	3

7 Conclusiones

Al comenzar este trabajo, el objetivo era lograr implementar el algoritmo de ray tracing íntegramente en el hardware gráfico programable, y dar un paso más hacia el fotorrealismo en tiempo real. Este objetivo está cumplido, el ray tracing en la GPU es posible, aunque con el estado actual de la tecnología de consumo no se llega a tiempos interactivos para escenas no triviales. De todas maneras, con el rápido avance de los procesadores gráficos, esta meta no es muy distante. Además de incrementarse rápidamente la velocidad de procesamiento de los mismos, se le agregan funcionalidades que permiten implementar con una mayor eficiencia el ray tracing, como por ejemplo los saltos condicionales en los programas de fragmento y la escritura a múltiples texturas en un mismo ciclo de renderizado.

Este avance es, al menos por el momento, más rápido que el de las CPUs, lo que le da un buen potencial a esta solución. También hay que tener en cuenta que los procesadores gráficos trabajan en paralelo y esta característica es altamente aprovechable por el método de ray tracing. Además, al ejecutarse en la GPU libera a la CPU para realizar otros cálculos simultáneamente.

Estas son las razones principales por las que es factible que en unos años se puedan utilizar implementaciones similares a esta para aplicaciones interactivas.

A lo largo del desarrollo del presente trabajo, nos enfrentamos a algunos problemas o limitaciones entre los que destacamos los siguientes:

- Las herramientas de desarrollo para los programas de fragmentos y vértices no estás muy avanzadas. Por esta razón, en algunos momentos se dificulta la programación y el depuramiento del código.
- El rápido avance de la tecnología hace que una implementación para una determinada generación de procesadores gráficos quede desactualizada y no sea la óptima para la siguiente generación.
- El ray tracing requiere mucha precisión en los cálculos y para ciertas operaciones las placas actuales provocan errores de precisión que se traducen en pixels renderizados erróneamente.
- La implementación del ray tracing en la CPU renderiza más rápido que en la GPU que utilizamos en este trabajo, para muchas de las escenas (aunque en las pruebas realizadas con una GPU más potente, esto se revierte).

La importancia fundamental de este trabajo es haber demostrado que es factible implementar este método en el hardware gráfico programable. Tomando esto como base, se puede continuar el desarrollo, agregándole nuevas funcionalidades y adaptándolo para las nuevas tecnologías y de esta forma acercarse cada vez más al fotorrealismo en tiempo real.

7.1 Trabajos futuros

Al ser el campo de la programación del hardware gráfico tan novedoso, y al tener el ray tracing tantas variantes, son muchas las posibilidades de expansión. A continuación nombramos algunas de ellas:

- Utilizar multiple rendering targets (MRT), una función de los procesadores gráficos más modernos que permite en un mismo programa de fragmento

escribir más de una textura. Esto permitiría reducir el número de programas necesarios y por lo tanto reducir el tiempo de renderizado.
- Utilizar las máscaras de la GPU para desactivar la ejecución de los programas de fragmento sobre pixels que ya llegaron a un resultado. De esta manera se evitarían ejecuciones y cálculos innecesarios.
- Implementar métodos de anti-aliasing para mejorar la calidad de las imágenes obtenidas, evitando los bordes dentados.
- Permitir el renderizado de objetos semi-transparentes que refracten la luz. Esta es una extensión típica del ray tracing.
- Evaluar diferentes estructuras aceleradoras que permitan modificar la escena dinámicamente.

8 Bibliografía

- [PUR/02] Purcell Timothy John, Buck Ian, Mark William R., Hanrahan Pat.
"Ray tracing on programmable graphics hardware", ACM Transactions on Graphics. 21 (3), pp. 703-712, 2002. (Proceedings of ACM SIGGRAPH 2002).

- [RAN/03] Randima Fernando and Mark J. Kilgard.
"The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics"
ISBN 0-321-19496-9 Addison-Wesley, 2003

- [WHI/1980] Whitted Turner.
"An Improved Illumination Model for Shaded Display"
Communications of the ACM 23, 6, 343–349.

- [PUR/04] Purcell Timothy John
"Ray Tracing on a Stream Processor"
Ph.D. dissertation, Stanford University, 2004

- [GEN/1993] Genetti Jon and Gordon Dan.
"Ray Tracing with Adaptive Supersampling in Object Space"
N. Jaffe, editor, Graphics Interface '93, 70-77

- [AMA/1987] Amanatides, J., and Woo, A.
"A fast voxel traversal algorithm for ray tracing."
In Eurographics '87, 3–10.

Apéndice A – Manual de usuario

A.1 Funcionalidad Básica

El programa renderiza escenas estáticas y permite realizar las siguientes acciones interactivamente:

- Trasladar y rotar la cámara.
- Trasladar la luz.
- Seleccionar los distintos modelos de iluminación y sus parámetros.
- Seleccionar la cantidad de iteraciones que se deben efectuar en caso de que haya reflexiones.
- Modificar el ángulo de visión.

A.2 Modelos compatibles

Actualmente, el programa sólo es compatible con archivos en formato **PLY** (con algunos agregados para permitir los diferentes colores y coeficientes especulares de las caras), con caras triangulares. Este archivo consta de un encabezado, una lista de vértices y una lista de caras. A continuación, se detalla la estructura de un archivo de este tipo:

```
ply
format ascii 1.0
element vertex num_vert
property float x
property float y
property float z
element face num_caras
property list uchar int vertex_index
end_header
V0X V0Y V0Z
V1X V1Y V1Z
V2X V2Y V2Z
     .
     .
     .
Vnum_vert-1X V num_vert-1Y V num_vert-1Z
VC C0V1 C0V2 C0V3 C0R C0G C0B C0CE
VC C1V1 C1V2 C1V3 C1R C1G C1B C1CE
VC C2V1 C2V2 C2V3 C2R C2G C2B C2CE
     .
     .
     .
VC Cnumcaras-1 V1 Cnumcaras-1V2 Cnumcaras-1V3 Cnumcaras-1R....
```

Donde:

`num_vert` = Cantidad de vértices del modelo.
`num_caras` = Cantidad de caras del modelo.

V_nX, V_nY, V_nZ = Coordenadas X, Y y Z, respectivamente, del vértice n (el nùmero de vértice va de 0 a num_vert-1).
VC = Cantidad de vértices de esa cara (para la versión actual del programa, siempre vale 3).
C_nV_m = Número de vértice del modelo, que es el m-ésimo vértice de la cara n (los números de caras van entre 0 y num_caras-1, mientras que m tiene que valer 1, 2 y 3 para todas las caras).
C_nR, C_nG, C_nB = Componentes R, G y B, respectivamente, de la cara n (van de 0 a 1). Si estos parámetros se omiten, la cara se dibujará blanca (1 1 1).
C_nCE = Inverso del coeficiente especular para la cara n (va de 0 a 1, siendo 0 totalmente reflexivo y 1 sin reflexión). Si este parámetro se omite, se tomará como 1.

Nota: Como las normales de las caras adyacentes se interpolan para producir una sensación de suavidad en las uniones de las caras, si se quiere armar un modelo en el que 2 ó más caras adyacentes tengan las aristas pronunciadas, se debe hacer que estas caras no compartan los vértices, y de este modo, las normales no sean interpoladas.

A.3 Modo de ejecución

Para ejecutar el programa, se debe generar un archivo de configuración de inicio (archivo **INI**) que contendrá los parámetros para el renderizado. También debe existir una carpeta con el nombre **Cache** en el mismo directorio en el que se ejecuta el programa. Dentro de esta carpeta, se crearán archivos que contendrán la información de la grilla (estructura aceleradora) que se armó a partir del modelo. Además, debe estar instalado en el sistema el Nvidia CG Toolkit 1.4 o superior.

En el archivo **INI** se deben detallar ciertos parámetros con el formato

parámetro: valor

Los parámetros aceptados son los siguientes:

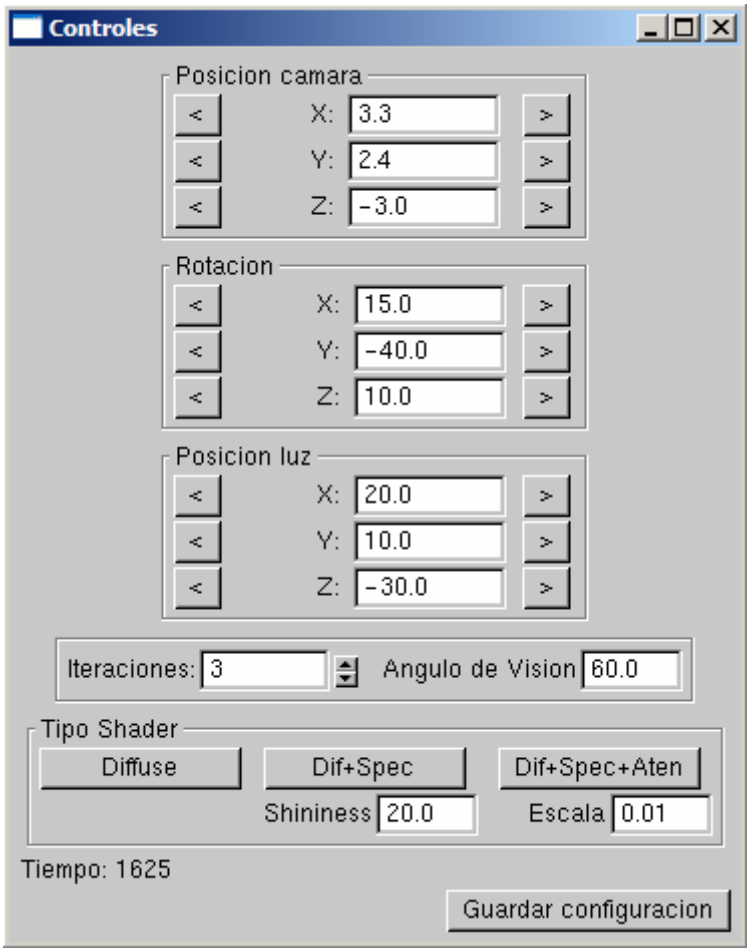
- **Tamano Texturas** (int) es la cantidad de pixels de ancho y largo que tendrá la imagen. Por convención se utilizan potencias de 2.
- **Tamano Grilla** (int, int, int) es la cantidad de subdivisiones que tiene la grilla en las coordenadas X, Y y Z, respectivamente.
- **Archivo PLY** (string(50)) Es el nombre del archivo **PLY** que contiene el modelo
- **Posicion Luz** (float, float, float) Son las coordenadas X, Y y Z en donde se encontrará la luz al comienzo de la ejecución (estos valores pueden cambiarse interactivamente).
- **Posicion Camara** (float, float, float) Son las coordenadas X, Y y Z en donde se encontrará la cámara al comienzo de la ejecución (estos valores pueden cambiarse interactivamente)
- **Direccion Camara** (float, float, float) Son los grados en los que aparecerá rotada la cámara en X, Y y Z, al comienzo de la ejecución (estos valores pueden cambiarse interactivamente).
- **Angulo de Vision** (float) Es la apertura (en grados) que tendrá el espectro de visión de la cámara. Este valor influye en la perspectiva y se puede modificar interactivamente.
- **Factor Intersecciones** (float entre 0 y 1) Es la relación máxima que debe haber entre los pixels que se encuentran realizando prueba de intersección y los pixels totales de la imagen para que comiencen a efectuarse los pasos de Traversal.

- **Maximo Reflexiones** (int) Es el número máximo de iteraciones que se debe realizar, en caso de que haya rayos reflejados (este valor puede ser modificado interactivamente).
- **Tipo Shader** (string(50) = **shader.cg | shader dif+spec.cg | shader dif+spec+aten.cg**) Este parámetro indica qué características se tendrán en cuenta para el efecto de iluminación. En el primer caso, la iluminación será difusa, en el segundo se le agregará un efecto de brillo a las superficies, y en el tercero se le agregará una atenuación a este efecto de brillo (este valor puede ser modificado interactivamente).
- **Shininess** (int) Es el valor del parámetro de shininess del cálculo del coeficiente especular (este valor puede ser modificado interactivamente).
- **Escala** (float) Este valor indica cuánto influye la distancia entre la fuente de luz y una cara en la atenuación de la iluminación (este valor puede ser modificado interactivamente).

A.4 Interfaz Gráfica

A partir de la interfaz gráfica, se puede visualizar el tiempo de renderizado de cada imagen, modificar algunos valores interactivamente y grabar la configuración de manera que la próxima vez que se ejecute el mismo modelo, la imagen inicial sea la que se está visualizando en el momento de la grabación. Estas 3 opciones las brinda una segunda ventana que se abre cuando se ejecuta el programa

A continuación, se muestra una captura de esta ventana de configuración y se detallan sus funcionalidades:



En los paneles **Posicion camara**, **Rotacion**, **Posicion luz**, **Iteraciones** y **Tipo Shader** se pueden modificar los valores que fueron tomados del archivo **INI**. La relación entre cada valor y el parámetro en el archivo **INI** es clara.

En el sector inferior izquierdo de la ventana, se puede visualizar el tiempo (en milisegundos) que tardó en hacer el renderizado de la imagen.

El botón **Guardar configuracion** permite establecer la configuración actual como la configuración predeterminada en la próxima ejecución del programa. Lo que realiza este botón puntualmente es modificar el archivo **INI** con los valores que se encuentran en la ventana. A su vez, genera un archivo de back-up con los datos existentes originalmente en el archivo **INI**. A este nuevo archivo le agrega el sufijo **.bkp** en su nombre.