Grupo Centro de Cálculo
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo – Uruguay

Proyecto de Grado

Ingeniería en Computación

Elementos de Iluminación Global

Informe Final

Autores

Emiliano Rodríguez

Juan Montesano

Diego Braga

Tutor

Mag. Ing. Eduardo Fernández

Resumen

La generación de imágenes realistas es una rama en el campo de la computación gráfica que genera nuevas imágenes a partir de otros datos. Típicamente, se comienza de una descripción completa de una escena tridimensional, especificando el tamaño y ubicación de los objetos, y la posición y características de emisión de las fuentes de luz. A partir de esta información, se genera una nueva imagen vista desde una cámara virtual ubicada en la escena. El objetivo es lograr que, en lo posible, las diferencias entre estas imágenes generadas y una fotografía real no sean apreciables. Esto requiere que los procesos físicos subyacentes relacionados con los materiales y el comportamiento de la luz sean modelados y simulados de forma precisa. Sólo conociendo exactamente qué se está tratando de simular es posible conocer dónde se pueden introducir simplificaciones en la simulación y cómo éstas afectarán las imágenes resultantes.

La generación de imágenes fotorrealistas es un objetivo muy ambicioso, y ha sido una de las principales fuerzas motrices en generación de gráficos por computadora en las últimas décadas.

La presente documentación contiene la información sobre el desarrollo y los resultados de un proyecto de grado de la carrera de Ingeniería en Computación de la Facultad de Ingeniería de la Universidad de la República Oriental del Uruguay. Dicho trabajo consistió en la mejora y desarrollo de nuevas funcionalidades de RadTR, una aplicación para uso académico dentro del Instituto de Computación que facilita la investigación y desarrollo de nuevas técnicas de iluminación Global, más específicamente sobre el algoritmo de Radiosidad.

Se pueden distinguir tres etapas en el proyecto, las cuales se describen a continuación:

1) Realización de un estudio de RadTR 1.0, y estudios del estado del arte en temas relacionados al proyecto.

2) Implementación de reformas y nuevas funcionalidades con la finalidad de mejorar la capacidad del programa para la generación de escenas (carga de modelos y texturas), aumento de la performance tanto del renderizado como de los algoritmos de Radiosidad (mejoras en el código y utilización de tecnologías avanzadas en programación de tarjetas gráficas), mejora de la interacción entre el usuario y el aplicativo, y conexión con la plataforma MATLAB.

3) Documentación del proyecto y un manual de usuario del aplicativo.

Como resultado se obtuvo una plataforma de experimentación mejorada y con nuevas funcionalidades para el área de computación gráfica, en especial en los temas relacionados con la iluminación global y radiosidad.

Palabras clave: Iluminación global, Radiosidad, MATLAB, CUDA, Computación gráfica, GPU, Tiempo Real.

Contenido

Resumen 3

1 Introducción 11

2 Estado del arte 13

2.1 Renderizado en Tiempo Real 13

2.2 Pipeline Gráfico 13

2.3 Tarjetas gráficas 14

2.3.1 GPU 14

2.3.2 Programación de la GPU 16

2.4 CUDA 17

2.5 OpenGL 18

2.6 Iluminación 18

2.7 MATLAB 19

2.7.1 El compilador MATLAB 20

2.7.2 MATLAB *Coder* 20

2.7.3 Usos de MATLAB *Compiler* y MATLAB *Coder* 21

2.8 Radiosidad de rango bajo 21

2.9 Software base: RadTR 1.0 23

2.9.1 Módulo escena 23

2.9.2 Importador/Exportador AC3D y Hemicubo 24

2.9.3 Cálculo de radiosidad 24

2.9.4 Módulo Main 25

2.9.5 Uso habitual del sistema 25

3 Alcance 27

3.1 Objetivos iniciales 27

3.2 Alcance del proyecto 28

3.3 Estimación del esfuerzo 29
4 Desarrollo 33
4.1 Arquitectura de software 33
4.2 Importador/Exportador de Modelos 3D 34
4.3 Módulo Main 34
4.4 Aumento de la iluminación 35
4.5 Factores de forma 35
4.5.1 Paraboloide 35
4.6 Radiosidad en Tiempo Real 37
4.7 Interfaz gráfica 39
4.8 MATLAB y RadTR 2.0 40
4.8.1 Interfaz mediante archivos 41
4.8.2 Interfaz mediante librerías compartidas 42
4.8.3 MATLAB *Coder* 43
4.8.4 Conclusiones sobre MATLAB *Coder* y MATLAB *Compiler* 44
5 Implementación 45
5.1 Escena: estructuras principales 45
5.2 Importación y exportación de Modelos 3D 45
5.3 Módulo Main: seleccionar e iluminar superficies 46
5.4 Cálculo de los factores de forma 46
5.4.1 Hemicubo 47
5.4.2 Paraboloide 47
5.4.3 Shaders 48
5.5 Módulo matemático 49
5.6 Radiosidad en tiempo real 49
5.7 Iluminación del cielo 52
5.8 Interfaz gráfica 53
5.9 Fichero de configuración 54

5.10 MATLAB y RadTR 2.0 54
5.10.1 Interfaz mediante archivos 54
5.10.2 Utilización del compilador MATLAB 55
6 Pruebas 57
6.1 Paraboloide vs Hemicubo 57
6.2 Paraboloide y subdivisión en *geometry shader* 59
6.3 Radiosidad en tiempo real 61
6.3.1 Escena Cornell Box 61
6.3.2 Escena LivingLiteColor 63
6.3.3 Observaciones 64
7 Conclusiones y trabajos futuros 67
7.1 Introducción 67
7.2 Conclusiones 67
7.3 Trabajos futuros 68
Glosario 71
Referencias 75
ANEXOS 81
ANEXO A – Preparación del ambiente de desarrollo 83
ANEXO B – Compilador MATLAB 84
Configuración del compilador 84
Para compilar para 32 bits 86
MCR 86
mxArray y mwArray 87
Generando librerías en C 87
Generando librerías en C++ 88
Archivos creados al generar las librerías 89
Estructura de un programa que utiliza librerías compartidas 90
Archivos a distribuir a un desarrollador 91

Interfaz de datos de usuario del MCR 91
ANEXO C – MATLAB *Coder* – Interfaz para arrays dinámicos 93
ANEXO D - Estructuras de datos principales 94
ANEXO E - Funciones de escena 96
ANEXO F - Radiosidad en tiempo real, código en CUDA y C 97
ANEXO G - Configuración de RadTR 2.0 100
ANEXO H - Jerarquía de superficies 102
ANEXO I - Hemicubo 103
ANEXO J - *Shaders* de división de triángulos en paraboloide 104
División de triángulos 104
División desde el centro del triángulo 105
División homogénea 107
División diagonal 109
División con *Tessellation shader* 111
Sin división 111
Anexo K – Manual de usuario 113
Menú 113
Importar - Cargar Emisión 115
Importar – Cargar Escena 115
Exportar – Generar Ficheros 115
Exportar – Guardar Escena 116
Herramientas – Screenshot 116
Herramientas - Calcular radiosidad 116
Herramientas - Radiosidad Tiempo Real 116
Herramientas - Calcular Factor Forma Cielo 118
Herramientas – Animación (F11) 119
Herramientas – Subdividir 119
Herramientas – Shaders 119

Herramientas - Carga Secuencial 120
Herramientas - Aumentar Iluminación 121
Ver - Cambiar Wireframe 121
Ver - Cambiar Caché 121
Ver - Ver Normal 121
Ver - Ver Hemicubo 122
Ver - Ver Paraboloide 122
Ver - Ver Proyección Paralela 122
Ver - Mostrar/Ocultar Información 122
Ver - Cámara - Guardar Estado Actual 123
Ver - Cámara - Cargar Estado 123
Ver - Cámara - Setear Estado 123
Switch Shading 124
Seleccionar Parche 124
Iluminar Parche 124
Activar/Desactivar Modo Precisión 124
Activar/Desactivar Carga Periódica de Emisión 125
Anexo L – Ejemplos de utilización de MATLAB con RadTR 2.0 126
Generando librerías compartidas e inclusión en Visual Studio 126
Ejemplo de aplicación de MATLAB *Coder* 126

1 Introducción

La iluminación en computación gráfica puede ser vista como el problema de simular la propagación de luz en una escena. Se puede abordar de diversas maneras, teniendo en cuenta qué tan físicamente realista se quiere la iluminación, qué tipo de aplicación se quiere desarrollar (ej., un videojuego), cuáles son los recursos de hardware disponibles, etc.

En Computación Gráfica, el concepto de iluminación global refiere a un grupo de algoritmos utilizados para agregar una iluminación físicamente realista a las escenas, que tienen en cuenta no solo la luz generada por las fuentes de luz (iluminación directa), sino que también el posible reflejo de esa luz en los diferentes objetos de la escena (iluminación indirecta). Algunos de los algoritmos dentro de esta temática son Radiosidad, *Ray Tracing* [1] y *Photon Mapping* [2]. La mejora del realismo debido al uso de la iluminación indirecta, tiene por contrapartida el aumento de complejidad en los cálculos. Esta consecuencia puede ir a contramano de la necesidad cada vez más vital de renderizar escenas en tiempo real, especialmente en las áreas de video-juegos, animaciones con fines didácticos y para el diseño arquitectónico.

El algoritmo de Radiosidad consiste en determinar la iluminación final de una escena constituida inicialmente por superficies donde sólo algunas emiten luz. El principal objetivo de los diferentes algoritmos de radiosidad es emitir hacia toda la escena la energía lumínica contenida en las superficies [3].

El proyecto consistió en la mejora de RadTR 1.0, un programa pensado para proporcionar herramientas en el ámbito de investigación universitaria sobre algoritmos de iluminación y de renderizado en tiempo real, centrado en el algoritmo de Radiosidad. El software final fue denominado RadTR 2.0.

RadTR 1.0 se programó en lenguaje C++, siendo utilizado en ambientes con sistema operativo Windows y arquitectura de 32 bits. Utiliza OpenGL como librería gráfica para el renderizado.

Para la mejora de performance en la generación de escenas y cálculos en los algoritmos, se utilizaron algunas estructuras especiales de OpenGL y se realizó la integración en el aplicativo de programación en tarjetas gráficas, específicamente en tarjetas NVIDIA [4] utilizando como lenguaje CUDA.

También se agregó la conexión e integración del programa con MATLAB, debido a que parte de los cálculos se realizan en dicho entorno. Para la conexión se agrega la capacidad de interacción entre RadTR 2.0 y una aplicación MATLAB por archivos en el *file system*, mientras que la integración consiste en compilar código MATLAB e invocarlo desde código contenedor en la aplicación.

El presente documento se organiza en siete capítulos. En el capítulo 2 se presenta el estado del arte correspondiente a algunas tecnologías utilizadas, así como también se describen algunos conceptos necesarios para el desarrollo y comprensión de este proyecto. El capítulo 3 detalla el alcance del proyecto, mencionando si tuvo alteraciones en el transcurso del mismo. En el capítulo 4 se describen las decisiones generales que se tomaron para resolver los puntos establecidos en el alcance. Luego, en el capítulo 5, se puntualizan las decisiones

tomadas para la implementación, sin pasar a detalles de código, los cuales se explican en los correspondientes anexos. En el capítulo 6, se muestran algunas pruebas realizadas sobre las funcionalidades principales de RadTR 2.0. Finalmente, en el capítulo 7, se expresan las conclusiones acerca del trabajo, indicando posibles mejoras a futuro.

Además, como información adicional, se presentan las referencias, glosario y anexos al final del documento.

2 Estado del arte

2.1 Renderizado en Tiempo Real

El renderizado en tiempo real (RTR por sus siglas en inglés, *Real Time Rendering*) refiere a generar rápidamente imágenes en una computadora. Es el área más interactiva de la computación gráfica. Una imagen aparece en la pantalla, el usuario acciona o reacciona, y esto repercute en lo que se genere posteriormente. Este ciclo de reacción y renderizado sucede tan velozmente que el usuario no visualiza imágenes individuales [5].

La tasa que se utiliza para medir la frecuencia de renderizado es imágenes por segundo (FPS, por sus siglas en inglés), que es una medida que indica la velocidad en que una aplicación gráfica despliega imágenes consecutivas. Se considera que 15 FPS o más indica que se está frente a una aplicación con renderizado en tiempo real.

El realismo y la efectividad de las aplicaciones con RTR se basan en dos elementos importantes: el Pipeline Gráfico y los avances en hardware gráfico.

2.2 Pipeline Gráfico

El *pipeline* gráfico es considerado como el componente central en el renderizado en tiempo real. La función principal es la de generar imágenes en dos dimensiones dada una escena tridimensional representada en un modelo que se denomina modelo de escena 3D [5].

Un modelo de escena 3D consiste en:

- Conjunto de elementos geométricos (vértices, líneas, puntos y triángulos) que representan objetos en la realidad a modelar. Los vértices son vectores de tres dimensiones y pueden incluir normales.
- Fuentes de luces.
- Propiedades de material de los objetos.
- Texturas (imágenes que serán “pegadas” a los objetos).

Este *pipeline* consta de tres elementos secuenciales denominados: Aplicación, Etapa Geométrica y Etapa de Rasterizado.

La primera etapa denominada Aplicación se desarrolla en la CPU. Ésta es desarrollada en gran parte por el programador. Aquí se desarrollan algoritmos de detección de colisiones, técnicas de aceleración y otros algoritmos avanzados. Como salida de esta etapa se deben enviar las primitivas (triángulos, vértices, información de texturas, etc.) al dispositivo de hardware gráfico.

En la Etapa Geométrica se realizan operaciones geométricas a los datos de entrada enviados por la etapa de aplicación. Estas operaciones podrían consistir en: mover objetos, mover la cámara, computar la luz en los vértices de los triángulos, proyección en la pantalla (3D a 2D), *clipping*, mapeo de ventanas, entre otras. En esta etapa del *pipeline* se realizan operaciones por vértices. La salida son triángulos 2D con atributos, o sea, pertenecientes a un mismo plano (la pantalla).

La Etapa de Rasterizado tiene como objetivo obtener la información de la Etapa Geométrica y convertirla en píxeles visibles en la pantalla. En esta etapa del *pipeline* se realizan operaciones por píxeles, como por ejemplo la interpolación de texturas y colores, test de profundidad (z-*buffer*) y *dithering*.

2.3 Tarjetas gráficas

Una tarjeta gráfica [6] es un dispositivo de hardware encargado de transformar las señales eléctricas que llegan desde el microprocesador en información comprensible y representable para un dispositivo de salida gráfico (monitor, televisor, pantalla de dispositivo, etc.).

Los elementos que se pueden distinguir en una tarjeta gráfica son: una unidad de procesamiento optimizada para el procesamiento de gráficos (denominada GPU), memoria para almacenar y transportar información, convertidor digital-analógico de RAM (denominado RAMDAC), BIOS de la tarjeta, interfaz con la placa y conexiones de salidas.

Históricamente el hardware de aceleración gráfica comenzó como un elemento del final del *pipeline* encargado inicialmente de la rasterización de triángulos. Las generaciones sucesivas de este tipo de hardware han hecho que este componente avanzara a etapas anteriores del *pipeline*. Actualmente, algunos algoritmos de etapas anteriores se implementan y ejecutan en estos dispositivos de hardware. Es que estos tipos de hardware dedicados tienen como ventaja principal la mejora de velocidad, y esto es crítico en la computación gráfica, especialmente en el renderizado en tiempo real [5].

2.3.1 GPU

La GPU (de sus siglas en inglés, *Graphics Processing Unit*) [7] [8] es un procesador especializado con funciones relativamente avanzadas de generación de gráficos, en especial para gráficos 3D. Se pueden distinguir dos elementos característicos que muestran esta especialización:

- Se encuentra optimizada para el cálculo con operaciones de coma flotante. Esto permite concentrar los cálculos de este tipo en esta unidad de cálculo, liberando así de trabajo a la CPU.

- Presenta elementos arquitectónicos que ayudan a paralelizar los cálculos. Esta característica se da como respuesta a que muchas aplicaciones gráficas conllevan

un alto grado de paralelismo inherente, ya que sus unidades fundamentales de cálculo (vértices y pixeles) son usualmente independientes.

Una de las razones por las que la CPU no se adapta bien a las aplicaciones gráficas de alto rendimiento es su modelo de programación en serie, que no soporta el paralelismo y los patrones de comunicación necesarios para este tipo de aplicaciones. La principal causa de la diferencia de eficiencia entre la CPU y la GPU radica en la arquitectura. Mientras la CPU posee una arquitectura basada en el modelo de Von Neumann, la GPU se basa en el tipo de arquitectura SPMD [8], que facilita el procesamiento en paralelo y la segmentación de tareas.

La frecuencia de reloj en las GPU puede ser muy baja en comparación con lo ofrecido por las CPU, sin embargo, gracias a su arquitectura en paralelo se llega a tener una potencia de cálculo mucho mayor, siempre y cuando la aplicación a ejecutar pueda paralelizarse.

El procesamiento vectorial o basado en flujo (*stream programing model* [8] [9]) se adapta a las estructuras de los programas gráficos de forma que ofrece una gran eficiencia en la computación y la comunicación. Este modelo de computación es la base para la programación de la GPU moderna.

En el modelo de programación vectorial todos los datos son representados como un flujo de datos (*streaming*), el cual se define como un conjunto ordenado de datos del mismo tipo. Este tipo puede ser desde un entero hasta una matriz de transformación compuesta por números en coma flotante. Las operaciones sobre flujos de datos muy grandes serán más eficientes que en el modelo en serie, ya que el coste por operación se reduce.

Por esto se realiza una "adaptación" del *pipeline* gráfico al modelo de procesamiento vectorial (*streaming*). La formulación vectorial trata todos los datos como flujos y los núcleos de computación como *kernels*.

La mecánica es la siguiente: por cada dato del flujo de entrada opera un *kernel*. Puede considerarse el *kernel* como un operador que actúa sobre cada dato en forma de función, aunque realmente puede contener un algoritmo genérico. Mediante la utilización del *kernel* sobre un flujo de datos se está ejecutando su código con todos los datos de dicho flujo. La característica principal de esta forma de procesamiento es que los datos son procesados de forma individual entre ellos, con lo que puede alcanzarse un alto grado de paralelismo mediante la dedicación de múltiples unidades de computación a la ejecución de un *kernel* sobre un flujo de datos.

El *pipeline* gráfico se adapta muy bien al procesamiento vectorial. Esto se debe a que se divide en varias etapas de procesamiento conectadas por flujos de datos. Esta estructura es análoga a un flujo de entrada sobre el que opera un *kernel*. Los flujos de datos entre etapas del *pipeline* gráfico están muy localizados; siempre como salida de una etapa y como entrada de la siguiente. Las operaciones que se realizan en cada etapa suelen ser uniformes para un conjunto de primitivas, por lo que pueden codificarse como *kernels*. Las GPU brindan la tecnología necesaria para llevar a cabo lo explicado anteriormente.

La evolución que se está produciendo en la industria del hardware gráfico está permitiendo en los últimos años un salto cuantitativo y cualitativo en el rendimiento de las aplicaciones gráficas. El uso de procesadores gráficos cada vez más potentes y capaces de realizar operaciones específicas utilizadas en Computación Gráfica permite la optimización de

muchos de los métodos empleados. Además de la posibilidad del desarrollador de añadir programas en el proceso de *rendering* (*graphics pipeline*) añade la posibilidad de realizar de una manera mucho más rápida algoritmos que han de procesarse para los modelos gráficos.

Últimamente se ha percibido la presencia de las GPUs en los equipos actuales como una posibilidad para resolver problemas de tipo genérico, o simplemente para paralelizar algunos procesos utilizando de manera conjunta la CPU y la GPU [10].

2.3.2 Programación de la GPU

Al inicio, la programación de la GPU se realizaba con llamadas a servicios de interrupción de la BIOS. Tras esto, la programación de la GPU se empezó a hacer en el lenguaje ensamblador específico a cada modelo. Posteriormente, se agregó una capa intermedia entre el hardware y el software, diseñando las API (del inglés, *Application Program Interface*), que proporcionaban un lenguaje más homogéneo para los modelos existentes en el mercado. Ejemplos de este tipo de APIs son OpenGL [11] (*Open Graphics Language*) y DirectX [12] (desarrollado por Microsoft [13]). Tras el desarrollo de APIs, se decidió crear un lenguaje más natural y cercano al programador, es decir, desarrollar un lenguaje de alto nivel para gráficos. Por ello de OpenGL y DirectX surgieron las siguientes propuestas:

- GLSL [11] (del inglés, *OpenGL Shading Language*). Lenguaje estándar de alto nivel asociado a la biblioteca OpenGL implementado en principio por todos los fabricantes.
- Cg (del inglés, *C for graphics*). Lenguaje propietario creado por NVIDIA.
- HLSL [14] (del inglés, *High Level Shading Language*). Desarrollado por Microsoft en colaboración con NVIDIA, prácticamente idéntico a Cg.

En la Figura 1 se presenta un esquema de programación utilizando la GPU como unidad de procesamiento, mostrando los posibles componentes a participar. Se tiene el componente Aplicación que generalmente es el programa principal a ejecutarse en una CPU. Uno de los posibles componentes de nexo entre la aplicación y la GPU es una librería gráfica (por ejemplo, OpenGL y DirectX) que brinda una API básica. Otra herramienta para la programación son los lenguajes de *shaders* que permiten personalizar el *pipeline* gráfico. Un tercer componente son los lenguajes de programación de propósito general en GPU (por ejemplo, CUDA y OpenCL) con los cuales la aplicación compila un programa en un lenguaje de alto nivel y lo transfiere a la GPU para que lo ejecute.

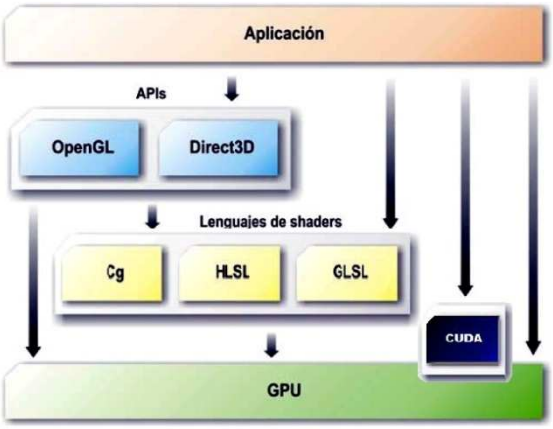


Figura 1: Esquema de Programación con GPU

2.4 CUDA

En el año 2007, NVIDIA [4] presenta CUDA [15] (*Compute Unified Device Architecture*). CUDA es una arquitectura de computación paralela para el cómputo de problemas de propósito general enfocada al cálculo masivamente paralelo. Desde su aparición se produjo un cambio radical en la arquitectura de las GPUs de NVIDIA. Su arquitectura, ahora unificada, no hace distinción entre procesadores de píxeles y vértices.

En la Figura 2 se muestra una pila de capas de software que incluye bibliotecas, CUDA *Runtime* y CUDA *Driver*. La capa de bibliotecas de CUDA está compuesta por dos bibliotecas matemáticas de alto nivel, CUFFT y CUBLAS. CUFFT se utiliza para el cálculo de Transformada de Fourier y CUBLAS es una implementación de BLAS (*Basic Linear Algebra Subprograms*) en GPU. CUDA *Runtime* provee una API que incluye un conjunto de instrucciones accesibles a través de lenguajes de alto nivel como son Fortran y C. CUDA Driver es un controlador de hardware dedicado a la transferencia de datos entre la CPU a la GPU.

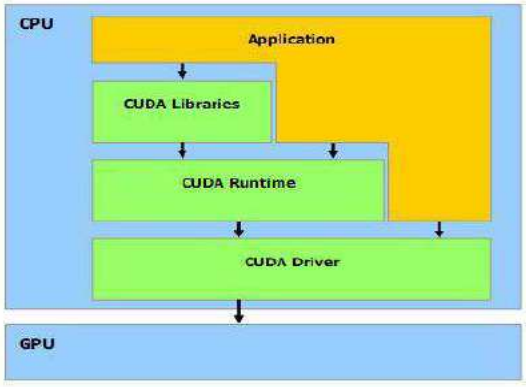


Figura 2: Capas de software de CUDA [16]

Al programar en CUDA, la GPU se ve como un dispositivo de cómputo capaz de ejecutar un número muy elevado de hilos en paralelo. La GPU funciona como un coprocesador de la CPU. Si una parte de una aplicación se ejecuta muchas veces pero en forma independiente sobre diferentes datos, puede ser aislada en una función que se ejecutará en el dispositivo mediante muchos hilos de ejecución en forma concurrente. La función se compila usando el conjunto de instrucciones del dispositivo. El programa resultante, llamado núcleo (*kernel*), se descarga en el dispositivo para su ejecución.

2.5 OpenGL

OpenGL [11] es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. Fue desarrollada por Silicon Graphics Inc. (SGI) en 1992. Su nombre viene del inglés *Open Graphics Library*, cuya traducción es biblioteca de gráficos abierta (o mejor, libre, teniendo en cuenta su política de licencias). Es una interfaz muy parecida estructuralmente al lenguaje C, de hecho su programación y compilación se realiza en dicho ambiente. Es claro que existen funciones y declaraciones exclusivas de OpenGL, pero a un programador en C, se le hace muy familiar este lenguaje.

A grandes rasgos, OpenGL es una especificación, es decir, un documento que describe un conjunto de funciones y su comportamiento exacto. A partir de ella, los fabricantes de hardware crean implementaciones (bibliotecas de funciones creadas para enlazar con las funciones de la especificación OpenGL, utilizando aceleración hardware cuando sea posible). Dichos fabricantes tienen que superar pruebas específicas que les permitan calificar su implementación como una implementación de OpenGL. Existen implementaciones eficientes de OpenGL suministradas por fabricantes para Mac OS, Microsoft Windows, Linux, varias plataformas Unix y PlayStation 3. Existen también varias implementaciones software que permiten que OpenGL esté disponible para diversas plataformas sin soporte de fabricante. La especificación OpenGL era revisada por el *OpenGL Architecture Review Board* (ARB) [17], fundado en 1992. El 31 de julio de 2006 se anunció que el control de OpenGL pasaría del ARB al Grupo Khronos. Con ello se intentaba mejorar el marketing de OpenGL y eliminar las barreras de desarrollo. El gran número de empresas con variados intereses que han pasado tanto por el antiguo ARB como por el grupo actual han hecho de OpenGL una API de propósito general con un amplio rango de posibilidades.

2.6 Iluminación

Cuando se renderiza imágenes de modelos tridimensionales, no sólo la representación debe tener la forma geométrica correcta, sino que deben presentar también una apariencia visual deseada. En muchos casos semejantes a fotografías de la vida real y en otros no, como por ejemplo en áreas del Rendering No Fotorrealista [5]. Cuando se intenta realizar una

representación realista de una escena se deben tener en cuenta los siguientes fenómenos físicos:

- La luz es emitida por una o varias fuentes (natural o artificial).
- La luz interactúa con los objetos: parte es absorbida, parte es dispersada y propagada en nuevas direcciones.
- Finalmente la luz es absorbida por un sensor (cámara u ojo humano).

La luz es modelada como rayos geométricos, ondas electromagnéticas o fotones. Sin importar como es tratada, la luz es energía electromagnética radiante que viaja en el espacio [5].

Las fuentes de luz emiten más luz de la que absorben o dispersan. Estas se pueden representar de diversas maneras. Aquellas fuentes extremadamente lejanas como el sol se simulan de una manera simple: su luz viaja en una única dirección que es la misma que en toda la escena. Por esto se denominan luces direccionales. La emisión de una luz direccional puede cuantificarse como la medida de intensidad en una unidad de área perpendicular a la dirección. Este valor denominado irradiancia es equivalente a la suma de fotones que atraviesan la superficie en un segundo. La luz puede ser coloreada, por lo que se representa su irradiancia como un vector RGB [5].

2.7 MATLAB

MATLAB [18] es un lenguaje para computación técnica que integra computación, visualización y programación en un entorno fácil de usar donde los problemas y las soluciones son expresados en la más familiar notación matemática. Los usos más comunes de MATLAB son Matemática y Computación, desarrollo de algoritmos, modelado, simulación, prototipado, análisis de datos, exploración, visualización, gráficas científicas e ingenieriles y desarrollo de aplicaciones.

MATLAB es un sistema interactivo cuyo elemento básico de almacenamiento de información es la matriz, que tiene una característica fundamental y es que no necesita dimensionamiento. Esto le permite resolver varios problemas de computación técnica (especialmente aquellos que tienen formulaciones matriciales y vectoriales) en una fracción de tiempo similar al que se consumiría cuando se escribe un programa en un lenguaje no interactivo como C o FORTRAN.

El nombre MATLAB es un acrónimo de del inglés *Matrix Laboratory* o Laboratorio de Matrices. MATLAB fue originalmente escrito para proveer fácil acceso el software de matrices desarrollado por los proyectos LINPACK y EISPACK. Hoy, los motores de MATLAB incorporan las librerías LINPACK y BLAS.

MATLAB presenta un conjunto de soluciones a aplicaciones específicas de acoplamiento rápido llamadas *toolboxes*. Los *toolboxes* son colecciones muy comprensibles de funciones MATLAB, o archivos de MATLAB (M-files) que extienden el entorno de MATLAB para resolver

clases particulares de problemas, Algunas áreas en las cuales existen *toolboxes* disponibles son:

- Procesamiento de señales.
- Sistemas de control.
- Redes neuronales.
- Lógica difusa.
- Wavelets.
- Simulación.

2.7.1 El compilador MATLAB

MATLAB *Compiler* [19] es un componente de la aplicación MATLAB que permite compilar las aplicaciones creadas en programas independientes o componentes de *software*. Los desarrolladores de MATLAB se refieren a menudo a la compilación como la construcción (*building)*.

Este compilador permite ejecutar los archivos generados en MATLAB fuera de dicha plataforma. Si se desea generar una aplicación independiente, el compilador crea un ejecutable. Si se desea integrarlo a un proyecto en C o C++, el compilador provee una API para utilizar el código creado en MATLAB como una librería compartida.

Algunos detalles sobre la sintaxis y las posibilidades para compilar se detallan en el Anexo B.

2.7.2 MATLAB *Coder*

Existe una buena alternativa a MATLAB *Compiler* si lo que en realidad se quiere es traducir código MATLAB a C. Este producto es MATLAB *Coder* [20].

Esta herramienta es particularmente útil cuando la performance es una prioridad. Por ejemplo, para un producto terminado tal vez no resulte muy adecuado tener que instalar el MCR en la máquina de cada usuario final. Por lo tanto, como último paso del desarrollo se podría convertir todo el código MATLAB a C++ y embeberlo en el proyecto haciendo todas las consideraciones necesarias en cuanto a las nuevas interfaces.

Es por esta razón que en este proyecto se hizo un análisis más profundo en el compilador y no tanto en esta utilidad, ya que lo que se pretende es acondicionar un ambiente de desarrollo cómodo en el que no se necesiten realizar muchos pasos entre la modificación de un archivo MATLAB y la ejecución de la aplicación en C++. Por lo tanto, la idea es trabajar con el compilador siendo consciente que la eficiencia probablemente mejorará cuando dicho código MATLAB se convierta a lenguaje C++ directamente.

2.7.3 Usos de MATLAB *Compiler* y MATLAB *Coder*

En caso que se desee generar código en C legible, eficiente y embebible a partir de código MATLAB se utilizará MATLAB *Coder*.

Por otro lado, se optará por utilizar MATLAB *Compiler* [19] si se pretende:

- Desarrollar un código en C o C++ que interactúa con MATLAB.
- Empaquetar aplicaciones MATLAB como ejecutables o librerías compartidas.
- Incorporar algoritmos basados en MATLAB dentro de aplicaciones desarrolladas utilizando otros lenguajes y tecnologías.
- Encriptar el código MATLAB, de manera que no pueda ser visto ni modificado.

2.8 Radiosidad de rango bajo

Esta sección se basa en un artículo elaborado por Fernández et al. [21].

El problema de radiosidad está modelado por la siguiente ecuación [3]:

$$B(x) = E(x) + \rho(x) \int_S B(x')G(x,x')dA'$$

Siendo *B(x)* la radiosidad en el punto *x* y *G(x,x')* el factor de forma indicando la cantidad de radiosidad que contribuye el punto *x'* al punto *x*.

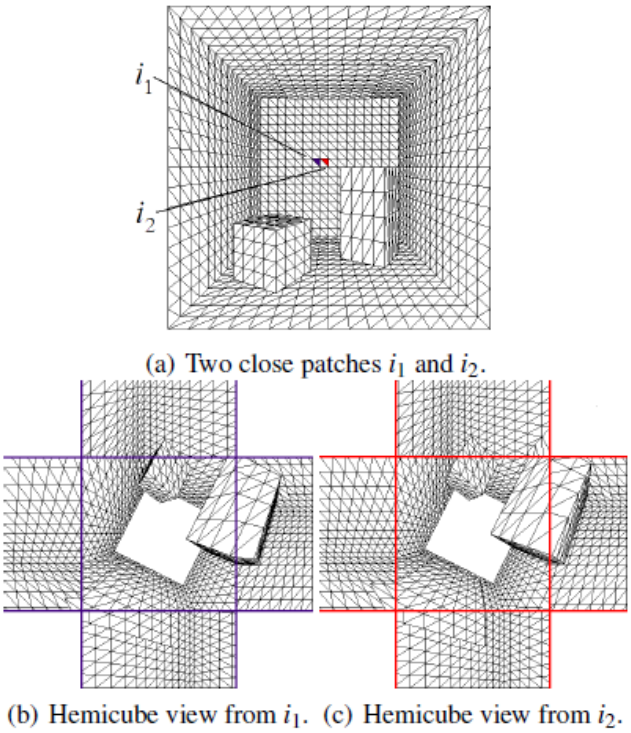
La ecuación anterior se puede transformar a una ecuación lineal [3]:

$$(\mathbf{I} - \mathbf{RF})B = E$$

Donde **I** es la matriz identidad, **R** es una matriz diagonal que almacena el color de cada polígono, **F** es una matriz donde se encuentran los factores de forma ($\mathbf{F}_{ij}$ indica la fracción de luz que refleja el polígono *i* de la luz que le llega del polígono *j*), *B* es la radiosidad final de la escena y *E* indica el conjunto de parches emisores.

La matriz **RF** ocupa una cantidad de memoria $O(n^2)$. Se sabe que **RF** es una matriz de bajo rango numérico, esto es porque cada fila $(\mathbf{RF})_i$ está basada en el punto de vista de la escena del polígono *i*. Existen polígonos que poseen un punto de vista de la escena parecido al polígono *i* (ver Figura 3) lo cual causa que existan filas de **RF** parecidas y que la matriz **RF** posea un bajo rango numérico.

A causa del bajo rango numérico de **RF**, es posible reducir la matriz a un producto de dos matrices $\mathbf{U}_k\mathbf{V}_k^T$ de dimensión *nxk* (*n>>k*), no perdiéndose información relevante acerca de la escena. La memoria necesaria para almacenar $\mathbf{U}_k$ y $\mathbf{V}_k^T$ es $O(kn)$, mucho menor que $O(n^2)$ que ocupa **RF**.



(a) Two close patches i_1 and i_2.

(b) Hemicube view from i_1. (c) Hemicube view from i_2.

Figura 3: Se observa cómo dos polígonos cercanos poseen una vista de la escena parecida.

Sustituyendo **RF** por $\mathbf{U_k V_k}^T$ en la ecuación anterior:

$$\left(\mathbf{I} - \mathbf{U}_k \mathbf{V}_k^T\right) \widetilde{B} = E$$

Despejando, utilizando inversión:

$$\widetilde{B} = E + \mathbf{U}_k \left(\left(\mathbf{I}_k - \mathbf{V}_k^T \mathbf{U}_k\right)^{-1} \left(\mathbf{V}_k^T E\right) \right)$$

Luego se reduce la ecuación anterior teniendo en cuenta que la escena es estática y los factores de forma no cambiarán, obteniéndose un complejidad de *O(nk)*:

$$\widetilde{B} = E - \mathbf{Y}_k \left(\mathbf{V}_k^T E\right),$$
$$\mathbf{Y}_k = -\mathbf{U}_k \left(\mathbf{I}_k - \mathbf{V}_k^T \mathbf{U}_k\right)^{-1}$$

2.9 Software base: RadTR 1.0

El proyecto comenzó con un software base (RadTR 1.0) al cual se le añadieron las funcionalidades objetivo del proyecto. Esta aplicación fue creada en el marco de la electiva técnica Computación Gráfica Avanzada, perteneciente a la carrera de Ingeniería en Computación, por los estudiantes Máximo Martínez, Vosky Clavijo y Rodrigo Porteiro, siendo continuado luego su desarrollo por Eduardo Fernández, docente del Instituto de Computación.

La aplicación fue desarrollada en el lenguaje C++. Consta de una interfaz por línea de comandos para ejecutar las acciones y un *display* cuya única funcionalidad era desplegar la escena en pantalla. Como veremos al final de esta sección, el procedimiento para calcular radiosidad no es práctico, siendo una de las motivaciones de este proyecto el mejorarlo.

En la Figura 4 se realiza un diagrama de la arquitectura de software para organizar mejor la función de cada módulo y cómo se relacionan.

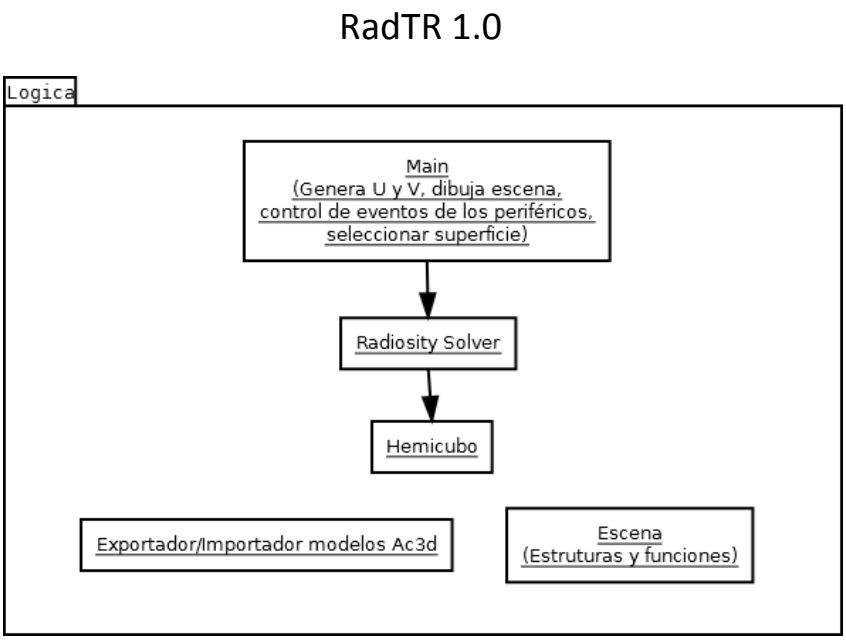


Figura 4: Arquitectura de RadTR 1.0.

2.9.1 Módulo escena

El módulo de escena contiene las estructuras de datos que definen una escena (superficies, vértices, materiales, colores, etc.) y funciones que trabajan sobre la misma. Entre estas funciones se encuentran la interpolación de color de los vértices dada una emisión de parches, cálculo del área de una superficie, cálculo de la normal de la superficie, asignación de

una emisión de luz a una escena (útil luego de calcular radiosidad a una escena) y subdivisión de los triángulos de una escena. La subdivisión es una función importante ya que además de aumentar la resolución de la escena aumentando la cantidad de triángulos también es la constructora de la jerarquía que es utilizada luego en radiosidad en tiempo real. Por más detalles ver anexos “Estructuras de datos principales” y “Funciones de escena”.

2.9.2 Importador/Exportador AC3D y Hemicubo

El importador y exportador de modelos 3D se limitaba solo al formato AC3D. Este formato es bueno para el proyecto porque soporta superficies con emisión de luz. La implementación tenía la limitante de no trabajar con texturas y estaba sujeto solo a este formato.

El software base disponía de un módulo que brinda una implementación de la técnica del hemicubo para calcular los factores de forma, el cual era utilizado por Radiosity Solver para calcular radiosidad de una escena y también para generar las matrices U y V. En el anexo “Hemicubo” se explica los detalles de la implementación.

2.9.3 Cálculo de radiosidad

El módulo Radiosity Solver se encarga de realizar el cálculo de radiosidad en la escena. Posee tres métodos de cálculo de radiosidad: *Shooting Solver*, *Shooting Ambient Solver* y *Gathering Solver*.

Shooting Solver en cada paso del algoritmo busca el parche con más energía a ser disparada, luego calcula los factores de forma para el mismo y transfiere su energía a los demás. Al finalizar el paso inicializa en 0 la energía del parche. La idea de este algoritmo es que en cada paso, el parche con más energía transfiere su energía lumínica a toda la escena.

Shooting Ambient Solver primero calcula la luz ambiente de la escena realizando un promedio de los colores de cada parche y su emisión de luz, luego aplica esa luz ambiental a cada parche de la escena.

Gathering Solver calcula la iluminación de un parche por cada paso del algoritmo, para esto obtiene los factores de forma para dicho parche y suma la radiancia de los parches de la escena según el factor de forma. La idea del algoritmo es calcular la iluminación de cada parche recogiendo la energía lumínica emitida por la escena. Para que este algoritmo se aplique correctamente se tienen que ejecutar como mínimo tantos pasos como cantidad de superficies haya en la escena.

2.9.4 Módulo Main

En este módulo se encontraban muchas de las funcionalidades del software. Se encargaba de dibujar la escena, de la función de selección de un parche a través del mouse, del control de comandos a través de la consola y la generación de las matrices U y V.

2.9.5 Uso habitual del sistema

Habitualmente, los pasos que se realizaban para calcular la radiosidad en una escena en RadTR 1.0 eran los siguientes:

1) Levantar la geometría con formato AC3D en RadTR 1.0.
2) Visualización y comprobación de la geometría de la escena.
3) Cálculo de las matrices U, V, R, F y A desde RadTR 1.0. A es el vector que contiene las áreas de los parches que conforman la escena.
4) Salvar a disco U, V, R, F y A (desde RadTR 1.0).
5) Levantar los archivos desde MATLAB.
6) Realizar cálculos para generar $\tilde{B}$ en MATLAB y almacenarlo en un archivo.
7) Importar desde RadTR 1.0 el archivo $\tilde{B}$.
8) Visualizar el resultado en RadTR 1.0.

Este proceso es generalmente iterativo, encontrando dos ciclos posibles dependiendo de los cambios que se deseen realizar:

- Ciclo 1 a 8: Si se realizan cambios en la geometría de la escena que impliquen volver a calcular las matrices U, V, R, F y A.
- Ciclo 6 a 8: Si se realizan cambios únicamente en la emisión de los parches y cuando se modifica el algoritmo de cálculo de $\tilde{B}$.

Entre las desventajas de este proceso podemos destacar:

- AC3D como única posibilidad de carga de modelos.
- La técnica del hemicubo es la única posibilidad de calcular los factores de forma.
- Transferencia de datos entre MATLAB y la aplicación mediante archivos.
- No hay posibilidad de modificar datos en tiempo real (iluminación inicial).

Buscar soluciones y alternativas para mejorar el proceso descrito anteriormente es una de las motivaciones principales de este proyecto.

3 Alcance

En este capítulo se describe el alcance. Primero se hace referencia a lo detallado en documento de requisitos entregado por el tutor. Posteriormente se describe el alcance real del proyecto.

3.1 Objetivos iniciales

Los objetivos iniciales del proyecto fueron definidos en el documento que contenía la propuesta del proyecto. En el mismo se indicaba que como producto final se debía tener una plataforma experimental en el área de computación gráfica que contemplara:

- La incorporación de un módulo para el ingreso y salida de información.
- Una mejor interacción gráfica con los potenciales usuarios (en primera instancia del ámbito académico).
- Conexión con otras bibliotecas y paquetes.
- Mejorar algunos módulos existentes.
- Facilitar la posibilidad de incorporación de nuevos módulos.

Para cumplir con estos objetivos se propusieron los siguientes resultados esperados:

- Levantar modelos gráficos en formatos variados.
- Realizar conexión con MATLAB (o a código de MATLAB previamente compilado). Invocar a código y realizar pasaje de información (de entrada y de salida), ya sea directamente o a través de memoria secundaria (disco).
- Nueva implementación de algunos módulos del código existente, basándose en técnicas gráficas avanzadas.
- Implementar en tarjetas de video algunas partes del código existente, relacionadas con la visualización de las técnicas de iluminación global en tiempo real.
- En radiosidad, aplicar métodos alternativos al hemicubo para el cálculo eficiente de los factores de forma, para la visualización de la escena desde cada parche.
- Desarrollar un código eficiente que permita visualizar cientos de miles de triángulos a una velocidad de al menos 20 FPS, usando z-buffer.
- Desarrollar una interfaz que permita cambiar las fuentes de luz en tiempo real, con cientos de miles de parches.
- Con el fin de realizar el cálculo de la iluminación global en tiempo real: aprovechar la potencia de las tarjetas gráficas para realizar operaciones matriciales en la tarjeta y así evitar el pasaje de información entre la tarjeta gráfica y la memoria RAM.

3.2 Alcance del proyecto

En la propuesta del proyecto, se definió un cronograma tentativo con un orden y una aproximación de tiempos de desarrollo. A medida que se avanzaba en el cronograma a un nuevo punto a desarrollar, se especificaba por parte del tutor la nueva funcionalidad o los cambios sobre una ya existente.

- Importar y exportar modelos gráficos en formatos variados.
    - Principalmente los formatos 3DS, AC3D y OBJ.
    - También el soporte de otros formatos.
    - Soporte de texturas.
- Una mejor interacción gráfica con los potenciales usuarios (en primera instancia del ámbito académico).
    - Incorporar una interfaz gráfica amigable con el usuario.
    - Utilizar la consola solamente para desplegar información.
- Realizar conexión con MATLAB (o a código de MATLAB previamente compilado). Invocar a código y realizar pasaje de información (de entrada y de salida), ya sea directamente o a través de memoria secundaria (disco). Esta conexión es necesaria para acelerar el proceso de cálculo de Radiosidad descripto anteriormente, evitando la transferencia de información manualmente.
    - Generar código compilado por MATLAB que luego es incorporado a la aplicación. Todos los datos están en memoria evitando transferencias entre ésta y el disco.
    - Definir una comunicación mediante *file system*.
        - Capacidad de levantar una serie de archivos de emisión con una nomenclatura determinada y luego almacenar la imagen generada por los mismos.
        - Posibilidad de levantar periódica y automáticamente archivos de emisión que son generados por código MATLAB interpretado.
- Implementar en tarjetas de video algunas partes del código existente, relacionadas con la visualización de las técnicas de iluminación global en tiempo real.
    - La posibilidad de ejecutar parte del algoritmo de radiosidad en tiempo real en la tarjeta gráfica utilizando CUDA.
- En radiosidad, aplicar métodos alternativos al hemicubo para el cálculo eficiente de los factores de forma, para la visualización de la escena desde cada parche.

  - Desarrollar el método del paraboloide o hemiesfera para el cálculo de factores de forma, teóricamente más eficiente que el método del hemicubo.

  - Desarrollar varias versiones del método del paraboloide donde cada una subdivide de diferentes maneras los triángulos con el fin de alcanzar un mayor grado de precisión de la geometría, para esto utilizar la tecnología *shaders* de las tarjetas gráficas.

- Desarrollar un código eficiente que permita visualizar cientos de miles de triángulos a una velocidad de al menos 20 FPS, usando z-buffer.

  - Utilizar técnicas proporcionadas por OpenGL para aumentar la velocidad de *rendering*.

- Desarrollar una interfaz que permita cambiar las fuentes de luz en tiempo real, con cientos de miles de parches.

  - Esta funcionalidad será utilizada en radiosidad en tiempo real para especificar la iluminación inicial que el usuario desee en todo momento durante la ejecución del algoritmo.

- Con el fin de realizar el cálculo de la iluminación global en tiempo real: aprovechar la potencia de las tarjetas gráficas para realizar operaciones matriciales en la tarjeta y así evitar el pasaje de información entre la tarjeta gráfica y la memoria RAM.

  - El algoritmo de radiosidad en tiempo real implementado en CUDA debe ser capaz de almacenar en la memoria de la tarjeta de video las matrices fijas y solo transferir la información que cambia de un *frame* a otro como son la iluminación inicial y la iluminación final.

- Calcular factores de forma desde el cielo.

  - Calcular los factores de forma de la escena desde el punto de vista del cielo, será útil cuando se considera al cielo como una fuente de luz.

  - Realizar el cálculo de radiosidad aplicando los factores de forma del cielo generados anteriormente para observar el impacto de esta técnica en la escena.

3.3 Estimación del esfuerzo

A continuación se presenta un resumen del esfuerzo realizado sobre las tareas ejecutadas para llevar a cabo el alcance definido. En la Tabla 1 se presentan las actividades generales del proyecto mostrando para cada una las siguientes columnas:

- **Descripción de la actividad**: Breve descripción de la actividad.
- **Estimación Inicial**: Duración estimada inicialmente en el documento elaborado por el tutor que contenía la propuesta del proyecto.

- **Estimación Final**: Duración aproximada de la actividad.
- **Período**: Período de tiempo en el cual se desarrolló la actividad.

Descripción de actividad	Estimación Inicial	Estimación Final	Período
Realizar estudio del software desarrollado en el CeCal, que se tomará como base	2 semanas	3 semanas	semanas 1 a 3
Relevamiento de cargadores de modelos geométricos	2 semanas	1 semana	semana 3
Estudio de posibilidades de tarjetas gráficas para la programación de técnicas gráficas	2 semanas	2 semanas	semanas 3 a 4
Realizar informes de relevamiento y estado del arte en los aspectos anteriores	3 semanas	5 semanas	semana 3 a la 7
Desarrollo de un plan de trabajo de mayor detalle y alcance	3 semanas	1 semana	semana 4
Implementación basada en el plan de trabajo	20 semanas	32 semanas	semanas 5 a 34 y 39* a 43
Documentación**	8 semanas	25 semanas	semanas 44 a 68

Tabla 1: Tiempo y período de actividades realizadas en el proyecto. (*) De la semana 35 a 38 corresponde aproximadamente al mes de enero, donde los integrantes realizaron actividades fuera del proyecto. (**) Se considera el tiempo de elaboración y espera de correcciones.

En la Tabla 2 se realiza una estimación detallada de las tareas que incluye la actividad "Implementación basada en el plan de trabajo".

Descripción de tarea	Estimación Final	Período
Importar y exportar modelos gráficos en formatos variados	2 semanas	semana 5 a 6
Incorporar una interfaz gráfica	2 semanas	semana 5 a 6
Desarrollar el método del paraboloide o hemiesfera	4 semanas	semana 7 a 10
Desarrollar un código eficiente	4 semanas	semana 13 a 16
Desarrollar varias versiones del método del paraboloide	9 semanas	semana 20 a 28
Calcular factores de forma desde el cielo	1 semana	semana 29
Realizar conexión con MATLAB (*file system*)	5 semanas	semana 16 a 20
Realizar conexión con MATLAB (código compilado por MATLAB que luego es incorporado a la aplicación)	8 semanas	semana 23 a 30
Algoritmo de radiosidad en tiempo real implementado con MATLAB	4 semanas	semanas 31 a 34 y semana 39
Algoritmo de radiosidad en tiempo real implementado con MATLAB Coder	1 semanas	semana 40
Desarrollar una interfaz que permita cambiar las fuentes de luz	1 semana	semana 32
Algoritmo de radiosidad en tiempo real implementado en CUDA	4 semanas	semanas 33 a 34 y 39 a 40
Pruebas	5 semanas	semanas 39* a 43

Tabla 2: Tiempo y período de tareas realizadas correspondientes a la actividad de implementación.

4 Desarrollo

En esta sección se describe el desarrollo del proyecto, comienza mostrando un diagrama de la arquitectura del software para guiar al lector de cada funcionalidad desarrollada y por último se explica cada una de ellas.

4.1 Arquitectura de software

Para organizar este capítulo primero se describen los cambios realizados a la arquitectura de RadTR 1.0 para generar RadTR 2.0 (Figura 5), luego se realiza una descripción de los nuevos módulos agregados al programa y los cambios que se tuvieron que realizar a los ya existentes.

Los módulos que se mejoraron fueron la importación y exportación de modelos 3D y el módulo principal Main. Los nuevos módulos son los que desarrollan la técnica del paraboloide, radiosidad en tiempo real, la interfaz gráfica, el módulo de funciones matemáticas, la carga periódica de iluminación de la escena y la iluminación del cielo.

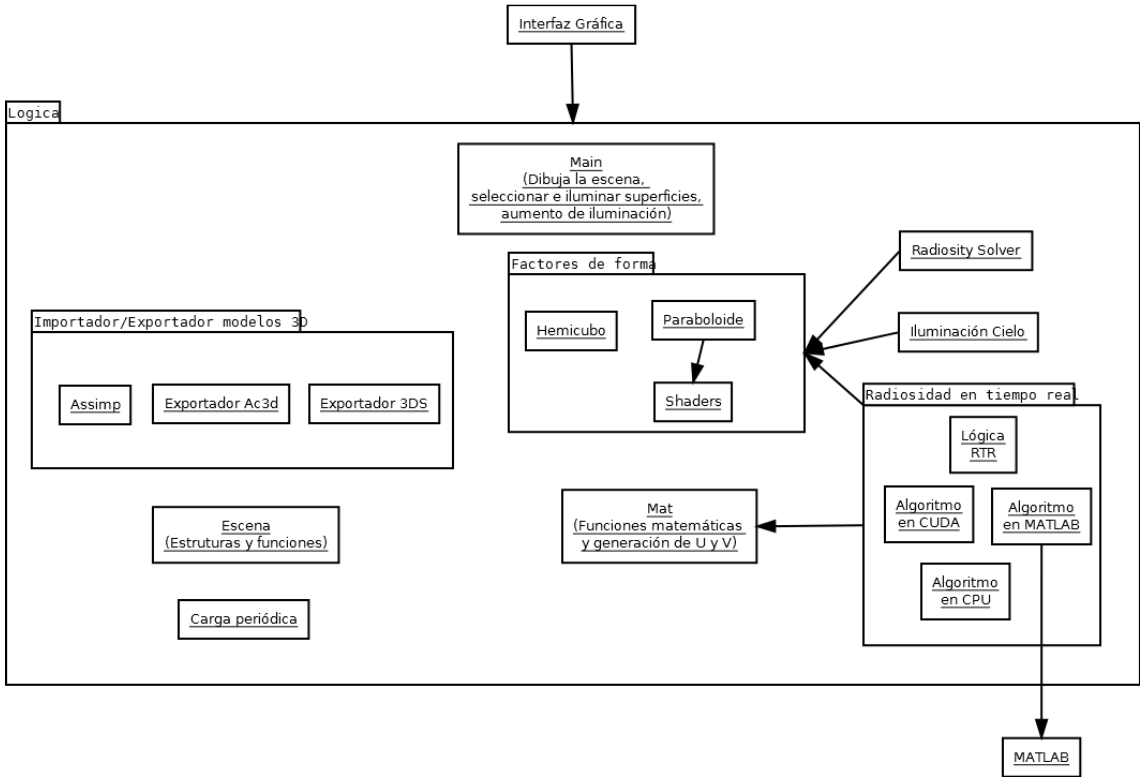


Figura 5: Arquitectura de RadTR 2.0

4.2 Importador/Exportador de Modelos 3D

Un requisito del proyecto es agregar al software el soporte a varios formatos para importar y exportar modelos 3D, ya que hasta entonces solo se contaba con el formato AC3D. Con este requerimiento cumplido le brinda al software una mayor flexibilidad a la hora de utilizar modelos 3D e independizarse de la utilización de otro software para convertir los modelos a formato AC3D.

Para la exportación e importación de modelos 3D se utilizó la librería Assimp por varias razones:

- Se encuentra escrito en C, el lenguaje utilizado en el software.
- Soporta la carga de varios formatos (dae, blend, 3ds, ase, obj, ply, dxf, lwo, lxo, stl, ac, ms3d, cob, scn , smd, vta, mdl, md2,.md3, pk3, mdc, md5, bvh, csm, x, b3d, q3d, q3s, mesh.xml, irrmesh, irr, nff , off , raw, ter, mdl, hmp, ndo), en especial el formato AC que mejor se adapta para trabajar con radiosidad ya que admite superficies emisoras de luz.
- Permite exportar modelos a los formatos dae y 3ds.
- Es una librería mantenida continuamente.
- Es fácil su uso.

Como Assimp no posee la exportación a formato AC se utilizo la funcionalidad de exportación de este formato utilizado en la versión anterior del software.

4.3 Módulo Main

Este módulo originalmente estaba muy cargado ya que poseía la implementación de la mayoría de las funciones del software base, durante el desarrollo se aisló muchas de sus funcionalidades en nuevos módulos como por ejemplo el módulo matemático, que ahora se encarga de la generación de las matrices U y V. También se quitó parte de la comunicación con el usuario y se la agrego al módulo de interfaz gráfica.

En la nueva versión se aceleró el dibujado de la escena utilizando *display list*. A su vez, se agregó una nueva funcionalidad que es el aumento de la iluminación de la escena que se detalla a continuación.

4.4 Aumento de la iluminación

En varios casos, cuando se calcula una iluminación a la escena ésta es demasiado baja como para ser observada. Para poder amplificar la iluminación de una escena se agregó la característica de "aumento de la iluminación". Dado un número *I* asignado por el usuario, se aplica el producto de *I* a la emisión de cada superficie y al color. De esta manera, con valores de *I* mayores a 1 se incrementa la iluminación de la escena y con valores de *I* menores a 1 se la decrementa. En la Figura 6 se muestra gráficamente esta funcionalidad.

Figura 6: Izquierda, escena luego de calcular radiosidad, se observa que está demasiado oscura. Centro, funcionalidad de aumento de iluminación. Derecha, escena luego de aplicar aumento de iluminación.

4.5 Factores de forma

El módulo de factores de forma implementa dos métodos en la nueva versión. El primero es el método del hemicubo, ya presente en el software base. Al mismo se le realizaron algunas mejoras para optimizar su rendimiento. El segundo es el método denominado Paraboloide, incorporado en esta nueva versión.

4.5.1 Paraboloide

La utilización de paraboloide [22] [23] [24] en lugar de hemicubo es una nueva característica que se agregó al software. El objetivo del paraboloide al igual que el hemicubo es acelerar el cálculo de los factores de forma utilizando la tarjeta gráfica. La ventaja teórica de esta técnica es una disminución de cuatro veces la cantidad de triángulos dibujados reduciendo cuatro veces el tiempo de cálculo.

La técnica de paraboloide dibuja toda la escena una sola vez y realiza una proyección de los triángulos sobre un paraboloide (ver Figura 7). Para realizar esto se tienen dos opciones. La primera es aplicar la proyección a cada vértice en CPU y enviar a dibujar la geometría ya

proyectada. Esta opción no es eficiente porque se presentan demasiados vértices para que la CPU los procese rápidamente. La segunda opción, que aprovecha la tecnología de *shaders*, es realizar esta transformación de los vértices en la tarjeta gráfica. De esta manera, se aprovecha la naturaleza paralela de estas tarjetas para procesar rápidamente todos los vértices.

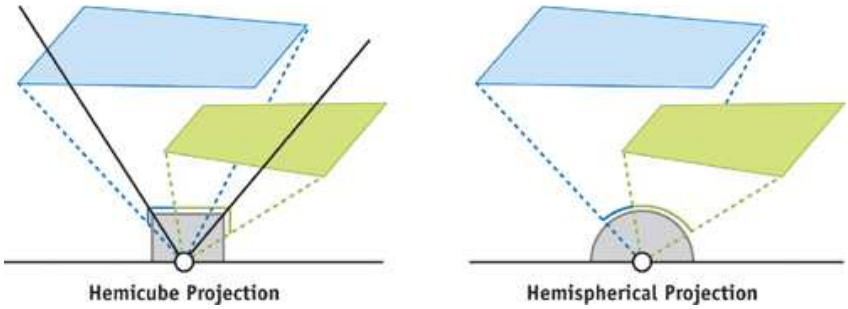


Figura 7: Proyección con hemicubo y con paraboloide (o hemiesfera) [22].

Para la técnica del paraboloide se desarrolló un *shader* que realiza la proyección de cada vértice en un paraboloide. De esta forma el observador tiene visibilidad de la escena de 180º (una hemiesfera). Sin embargo, este método tiene un problema: las proyecciones se aplican solo a los vértices y no a todo el triángulo y, sumado a que la tarjeta gráfica sólo dibuja triángulos con aristas rectas, se puede ver que la proyección de la escena no se aplica correctamente. Para lograr una proyección correcta se debe proyectar cada píxel del triángulo. En la Figura 8 se visualiza el error de aristas rectas cuando se está proyectando.

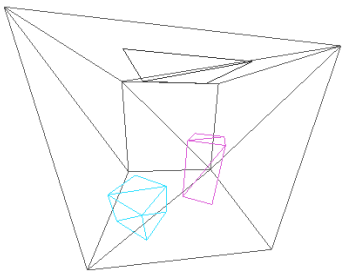


Figura 8: Vista de paraboloide en una escena con 38 triángulos.

Existen dos formas de solucionar este problema. Una es aplicar la proyección a cada pixel del triángulo pero es muy costoso. Otra forma es aumentar la resolución de la escena, aumentando la cantidad de triángulos y así reducir los errores visuales (Figura 9). Este último método es el elegido porque brinda una buena resolución de imagen en tiempos aceptables.

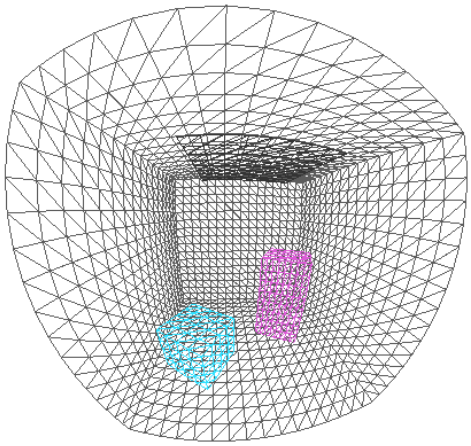


Figura 9: Vista de paraboloide en una escena con 3.584 triángulos.

Para aumentar la cantidad de triángulos se puede utilizar la funcionalidad subdividir, que subdivide la escena. También se desarrolló un conjunto de *shaders* que aprovechan el *geometry* y *tessellation shader* de algunas tarjetas gráficas para subdividir cada triángulo en hardware. Si el software detecta que la tarjeta gráfica posee *geometry* y/o *tessellation shader* el usuario tiene la posibilidad de elegir si realizar la subdivisión de triángulos en la tarjeta y también qué tipo de división. Las divisiones posibles son de "triángulos", "centro del triángulo", "homogénea", "diagonal", "ninguna división" y "división con teselado". En el anexo "*Shaders* de división de triángulos en paraboloide" se describen estas formas de división.

4.6 Radiosidad en Tiempo Real

Uno de los objetivos del proyecto es la implementación en RadTR 2.0 de una funcionalidad que sea capaz de calcular radiosidad a una escena en tiempo real utilizando cierto algoritmo. También debe brindar la posibilidad de reemplazar fácilmente dicho algoritmo por otros en un futuro. En la Figura 10 se muestra la arquitectura de software del módulo Radiosidad en tiempo real y los módulos con los cuales se relaciona.

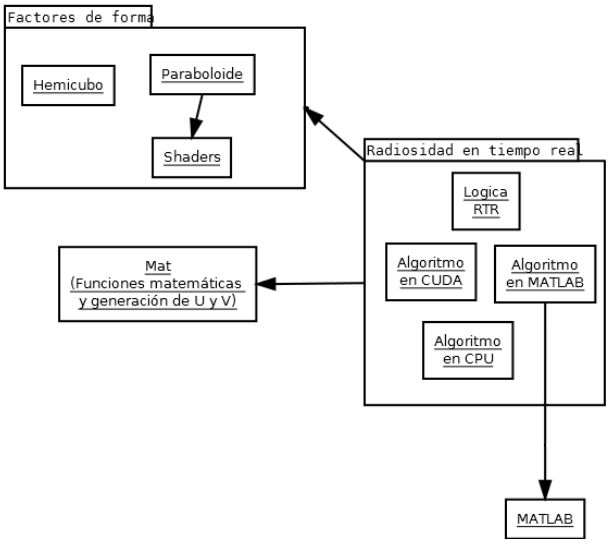


Figura 10: Arquitectura de software relacionada con Radiosidad en tiempo real.

Radiosidad en tiempo real tiene que ser capaz de calcular para cada frame la iluminación de la escena utilizando radiosidad para una iluminación inicial cambiante. Es necesario alcanzar una cantidad suficiente de FPS para que exista la sensación de tiempo real (más de 15 FPS).

Antes de comenzar la radiosidad en tiempo real (RTR) es necesario hallar ciertas matrices necesarias para el cálculo. Estas matrices pueden ser precalculadas porque son fijas (no cambian durante el algoritmo de RTR) y no es viable recalcularlas en cada frame ya que es necesario mucho tiempo en hallarlas (puede llegar a tardar varios minutos). La razón de la cantidad de tiempo para hallar estas matrices es la necesidad de calcular todos los factores de forma de las superficies de la escena y realizar multiplicaciones matriciales pesadas (matrices con dimensiones grandes). Para acelerar este paso se utiliza la técnica de paraboloide para el cálculo de factores de forma y si está presente CUBLAS se realizan las operaciones matriciales utilizando dicha librería. Estas matrices son U, V comprimida e Y.

Para que el programa tenga la capacidad de generar una iluminación inicial que se modifique en cada frame y que el usuario sea capaz interactivamente de definirla, se implementó algo similar a una linterna capaz de iluminar ciertas zonas de la escena (las cuales el usuario sea capaz de determinar la dirección y definir el área iluminada). Para esto, al inicio de cada frame se dibuja la escena con el punto de vista del observador y con la dirección de vista igual a la apuntada por el mouse. Para la generación de esa escena se utiliza un frustum tal que el área de iluminación sea el área dibujada en la pantalla. De esta forma se conoce qué triángulos son iluminados por la linterna virtual, analizando los triángulos que fueron dibujados en pantalla.

Luego de tener la iluminación inicial se realizan operaciones matriciales con las matrices precalculadas y la iluminación inicial para hallar la iluminación final. Para este paso se desarrollaron cuatro posibles implementaciones:

- En CPU, se utilizó lenguaje C++.
- GPU con la librería CUBLAS, ejecuta en tarjetas gráficas.

- MATLAB *Compiler*, la ejecución se realiza en el ambiente MATLAB.
- MATLAB *Coder*, se generó código C a partir del código MATLAB y se compiló ese código junto a RadTR 2.0.

Todas ellas realizan exactamente el mismo cálculo matricial, simplemente se cambia la forma de hacerlo. La razón para tener estos cuatro métodos es principalmente tener cierta flexibilidad a la hora de desarrollar nuevas versiones del algoritmo. En la etapa de investigación de nuevos algoritmos para hallar la iluminación final es deseable realizarlas en MATLAB y probar con el método de ejecución en dicho ambiente. Luego, cuando se tenga un algoritmo final tal vez se quiera probar su rendimiento en CUBLAS y en CPU para comparar tiempos y posiblemente realizar mejoras. Otra ventaja de tener varios métodos es la portabilidad, o sea, poder prescindir de CUBLAS y/o MATLAB para ejecutar el algoritmo de RTR. También es útil para realizar comparaciones de rendimiento de cierto algoritmo ejecutando este en CPU, GPU o MATLAB. Al final del proyecto se incorporó la opción de utilizar la librería BLAS en la implementación en CPU para la multiplicación de matrices.

En la teoría se hallaron las fórmulas del cálculo de radiosidad en tiempo real para valores de iluminación monocromática. Esto quiere decir que el cálculo de la iluminación se hace con luz blanca no importando si ésta es reflejada por un objeto de color. En la práctica se desarrolló un segundo método del algoritmo donde sí importa el color de la luz y cómo ésta interacciona con el color de los objetos de la escena. Este segundo algoritmo obtiene una iluminación más realista de la escena ya que la luz posee el color correcto en cada momento, una desventaja de tener tres valores (RGB) en vez de uno como color de la luz es el aumento del tiempo de ejecución del algoritmo.

4.7 Interfaz gráfica

Una característica deseable y no menos importante es la interfaz gráfica de usuario. El software base carecía de interfaz gráfica, limitándose la interacción con el usuario mediante comandos en consola.

Al tratarse de un software en el que es importante la buena visualización de la escena tridimensional, la interfaz gráfica tiene que ocupar el mínimo espacio posible en pantalla y al mismo tiempo ser fácil e intuitivo acceder a las funcionalidades. Por esta razón, se escogió utilizar un menú donde se encuentra ordenado por categorías las funcionalidades. Dicho menú no se encuentra visible y tiene que ser desplegado utilizando el botón derecho del mouse. Para algunas funciones que se necesite de algunos parámetros se desarrollaron ventanas donde el usuario es capaz de realizar la configuración deseada.

Ahora la consola se utiliza para desplegar información como pueden ser errores, avance de alguna operación o simplemente datos que le pueda ser útil al usuario.

En ocasiones al usuario le interesa tener una posición de la vista para visualizar algún detalle gráfico, luego moverse a través de la escena para visualizar otro lugar y por último volver exactamente a la posición original. Posicionarse exactamente en un lugar es complicado y sumarle que el usuario tiene que recordar posiciones que le resulten interesantes se hace un

trabajo pesado. Para esto se realizó una funcionalidad que es capaz de almacenar posiciones de la vista donde es posible guardar la posición actual o definir manualmente cierta posición. Así, en cualquier momento se puede cargar cualquier posición de vista y automáticamente la cámara se posicionará en dicha posición. Para simplificar la interacción con el usuario las posiciones de vista son almacenadas por nombres.

Para facilitar aún más el modo en que el usuario se mueve a través de la escena se desarrollo un modo en el cual se puede realizar movimientos finos, al cual se le denomina "modo precisión". Este modo cuando está activo tiene dos submodos:

- Al mantener presionada la tecla Ctrl realiza movimientos discretos de una unidad (por ejemplo cuando se quiere avanzar se avanza una unidad hacia adelante).
- Al mantener presionada la tecla Shift realiza movimientos discretos más finos de 0.01 y con movimientos del mouse cambia la dirección de vista pequeños grados.

Una funcionalidad que existía en el software base era la posibilidad de situar al observador en el centro de una superficie y como vector vista la normal de la superficie. Esta característica se mantiene en esta versión y para usarla simplemente se realiza click derecho sobre la superficie que se desea situar y seleccionando la opción "Seleccionar parche".

Una nueva característica que se agregó es la posibilidad de iluminar/oscurecer superficies. Tiene el mismo funcionamiento que "Seleccionar parche" pero en lugar de situar al observador en la superficie elegida lo que hace es iluminar la misma en caso que esté opaca u oscurecerla en caso que esté iluminada. Durante el desarrollo y pruebas de la aplicación se decidió agregar esta funcionalidad para brindar más flexibilidad a la hora de definir una iluminación. Hasta entonces se tenía que modificar la iluminación en un software de modelado 3D, exportar la escena y luego importarla desde la aplicación RadTR 2.0.

4.8 MATLAB y RadTR 2.0

El rol de MATLAB en este proyecto es el de facilitar la realización de cálculos matriciales para el cálculo de radiosidad. El proceso previo a este trabajo se resume en que RadTR 1.0 generaba matrices que luego se levantaban en MATLAB para realizar los cálculos de radiosidad. Los vectores resultantes de dichos cálculos eran luego levantados por RadTR 1.0 para lograr una visualización de la iluminación.

El método anterior tiene la gran desventaja de realizar mucha transferencia de información entre la memoria y el disco. Por lo tanto, se investigaron soluciones alternativas a este problema que en lo posible mantuvieran la información siempre en memoria durante los cálculos, evitando así cuellos de botella debido a los costos de escritura y lectura en disco.

La búsqueda se centró en las herramientas que permitieran ejecutar código creado en MATLAB desde C. Como se detalla en capítulos anteriores, las posibilidades que se encontraron fueron dos: MATLAB *Compiler* y MATLAB *Coder*. De todas formas se

implementaron funcionalidades basadas en la transferencia de archivos ya que podrían ser de utilidad en algunos casos.

En resumen, la comunicación entre RadTR 2.0 y MATLAB debe considerar dos interfaces:

1) **Interfaz mediante archivos:** Carga periódica de archivos de emisión que es generada por código MATLAB interpretado.
2) **Interfaz mediante librerías:** Generación de código compilado por MATLAB para ser incorporado luego a la aplicación.

Claramente, parece ser más prolija la interfaz mediante librerías, ya que se evita el pasaje de la información a disco, utilizando en su lugar la memoria. Para una funcionalidad en tiempo real sería más razonable elegir la segunda opción.

De todas maneras, se implementaron ambas interfaces, pero para funcionalidades diferentes. Mientras que la primera se utilizó para realizar la carga periódica de archivos de emisión, la segunda fue empleada en la resolución de un algoritmo de radiosidad en tiempo real.

Si bien estas son las dos formas que se estudiaron en profundidad, se hizo referencia a un tercer modo de interacción con MATLAB que implica utilizar la herramienta MATLAB *Coder.*

4.8.1 Interfaz mediante archivos

Carga Secuencial

La Carga Secuencial es una funcionalidad nueva de RadTR 2.0 que permite básicamente generar una serie de imágenes de una escena al cargar secuencialmente una serie de archivos que definen su iluminación-composición.

Su utilidad es mostrar cómo puede ir convergiendo un algoritmo recursivo de luminosidad, siendo los archivos de entrada los resultados intermedios de esa recursión, para sacar conclusiones de que tan rápido puede ser un algoritmo, de cuántos pasos son necesarios para comenzar la convergencia de la solución.

Carga periódica

La Carga Periódica es una nueva funcionalidad de la aplicación RadTR 2.0, que se encarga de leer de un archivo de entrada cada cierto tiempo y generar una imagen con los datos obtenidos desde ese archivo. Ese archivo de entrada se presume que es cambiado por otra aplicación concurrentemente que RadTR 2.0 lo está leyendo. En las pruebas el archivo era modificado mediante un programa MATLAB. Como resultado final de la carga periódica se tendrá una secuencia de archivos de imágenes en formato bmp.

Su utilidad es mostrar cómo puede ir convergiendo un algoritmo recursivo de luminosidad, mostrando los resultados intermedios de esa recursión que se van guardando en el archivo a cargar, para sacar conclusiones de que tan rápido puede ser un algoritmo, de cuántos pasos son necesarios para comenzar la convergencia de la solución.

4.8.2 Interfaz mediante librerías compartidas

El objetivo en este caso es encontrar una forma prolija de ejecutar aplicaciones elaboradas en MATLAB desde RadTR 2.0. Es deseable que el tiempo necesario desde la modificación del código escrito en MATLAB y la ejecución final de la aplicación contenedora (RadTR 2.0) sea lo más pequeño posible, acelerando de este modo el proceso de desarrollo.

Antes de poder pensar en un posible diseño la interfaz, se realizó una investigación acerca de las posibilidades tecnológicas de las versiones utilizadas de Visual Studio y MATLAB. Las aplicaciones que se utilizaron para desarrollar RadTR 2.0 son: Microsoft Visual Studio 2010 y MATLAB R2011a. Por lo tanto, se limitará el estudio a estas dos versiones. Existirán cambios en versiones próximas anteriores y posteriores pero, en líneas generales, la idea será la misma.

Si en algún momento se decide cambiar la versión de Visual Studio o MATLAB se deberán tener en cuenta aspectos de compatibilidad tal como se detalla en el Anexo A. Particularmente, al cambiar la versión de MATLAB se deberán chequear los cambios con respecto a versiones anteriores. Estos cambios pueden ir desde modificación de parámetros de una función hasta la eliminación completa de una funcionalidad.

Utilización del compilador MATLAB

El compilador MATLAB es una poderosa herramienta que ofrece numerosas posibilidades. Dentro de ellas se encuentra la posibilidad de generar librerías compartidas en diferentes lenguajes. En particular, interesa la generación en los lenguajes C y C++. También, se realizaron estudios acerca de la forma de generación de estas librerías y las posibilidades de comunicación entre las mismas y el código contenedor que las invoca. Un resumen de dicha investigación ya se pudo ver en el estado del arte y los anexos correspondientes, por lo que la intención aquí es describir las decisiones tomadas y no volver a mencionar dichos aspectos técnicos.

Luego de realizar la investigación sobre el compilador MATLAB se desarrolló un ejemplo para demostrar que es útil y, principalmente, que funciona correctamente con la aplicación. Es debido a estas razones que se decidió aplicarlo en RadTR 2.0 en el cálculo de radiosidad en tiempo real.

Se parte de un archivo de entrada escrito en MATLAB que será invocado en cada paso de la iteración. En el mismo se encontrará la rutina necesaria para realizar los cálculos de radiosidad en tiempo real. Para esta rutina se tienen como entrada datos que se inicializan una vez al comienzo del proceso y datos que pueden ir variando en cada *frame*.

Se encontraron dos alternativas para realizar el pasaje de dichos datos:

1) Que el código MATLAB fuera una función y que se pasaran los datos como argumentos a la misma.
2) Que el código MATLAB fuera un simple *script* y que los datos se transfieran mediante el intercambio de variables de usuario del MCR.

Finalmente, se optó por la opción 2. Si bien la alternativa 1 podría ser buena en el caso en que se tenga claro qué parámetros tendría la función MATLAB, la segunda resultó más general. En ésta, las llamadas desde RadTR 2.0 no dependen de la cantidad de parámetros de la función, por lo que no se debería reconstruir el proyecto cada vez que se agregue un argumento a la misma.

Como el código que se encuentra en el archivo de entrada no es una función, los parámetros para invocar el mismo desde RadTR 2.0 serán siempre los mismos, o sea, ninguno. Por lo tanto, si se modifica el archivo de entrada, no será necesario compilar de nuevo la aplicación, sino que sólo se debería actualizar la librería dinámica generada por el compilador MATLAB.

Algo muy importante que se debe tener en cuenta es que no hay un espacio de variables (como el *workspace* de la aplicación MATLAB) definido entre ejecuciones de funciones en librerías compartidas generadas por el compilador. Ejemplificando, si se tiene una función que asigna una variable A y luego se invoca otra distinta que imprime su valor, esta última indicará que dicho valor no está definido. Por lo tanto, para suplir dicha carencia, se utiliza el intercambio de variables de usuario de MCR.

Luego, en la descripción de la implementación se discute en detalle las decisiones de generación de las librerías para que esta funcionalidad funcione correctamente.

4.8.3 MATLAB *Coder*

MATLAB *Coder* es una alternativa a MATLAB *Compiler*. Se muestra el uso de este generador para el mismo caso que el del compilador, o sea, para el cálculo de la radiosidad en tiempo real.

Como MATLAB *Coder* sólo genera código a partir de funciones MATLAB se debió transformar el script que se utilizaba para MATLAB *Compiler* a una función. Notar que aquí ya se presenta una importante diferencia con el código que se compiló, que no recibía parámetros ya que era un script MATLAB. Esto podría afectar las pruebas de performance, ya que esta llamada se hará para cada *frame* y la cantidad de parámetros podría ser un factor no menor, por más que los mismos sean sólo punteros.

En el Anexo L se muestra el ejemplo de utilización de MATLAB *Coder* aplicado a RadTR 2.0.

Ventajas y desventajas

Con MATLAB *Coder* se encontraron las siguientes ventajas y desventajas:

Ventajas:

- No es necesario tener instalado MCR en la máquina del usuario final para ejecutar.
- Posiblemente una mejor performance que si se utilizara el compilador.
- El código generado puede ser fácilmente modificado (en C++). Esto se considera ventaja cuando la privacidad no es un requisito.

Desventajas:

- Cada vez que se modifica un archivo MATLAB se debe regenerar el mismo y volver a compilarlo junto con el código que lo invoca.
- Hay más limitaciones con respecto a lo que se puede generar, no se dispone de todo el potencial de MATLAB.
- El código no tendría que tener *warnings* y tendría que ser analizado antes de la generación.
- Para cambiar el tipo de un dato de entrada o salida es necesario regenerar el código.

4.8.4 Conclusiones sobre MATLAB *Coder* y MATLAB *Compiler*

Para tareas de desarrollo es más adecuado utilizar el compilador MATLAB, debido a la menor cantidad de objetos que hay que actualizar. Además, proporciona más privacidad con respecto a cómo está implementado el código MATLAB.

Por otro lado si lo que se busca es construir una versión destinada a usuarios finales, tal vez lo más adecuado sea generar el código en C++ con MATLAB *coder*, compilarlo y linkeditarlo junto con la aplicación para luego distribuir el ejecutable. Notar que este caso sería posiblemente más eficiente y, si no se distribuye el código, se mantendría la privacidad.

Como RadTR 2.0 es una aplicación en continuo desarrollo sería más apropiado utilizar MATLAB *Compiler* en lugar de MATLAB *Coder*.

5 Implementación

En este capítulo se describe la implementación de las nuevas características que se añadieron a RadTR 1.0.

5.1 Escena: estructuras principales

En esta sección se describen las mejoras agregadas a las estructuras de datos utilizadas en RadTR 2.0. En el anexo “Estructuras de datos principales” se muestra el código que define dichas estructuras.

Se agregó soporte para el uso de texturas. Se modificó la estructura *scene* para almacenar las coordenadas de textura de cada vértice y la estructura *material* para asignar y almacenar la información de las texturas de cada material. Es posible almacenar texturas de diferentes tipos (color ambiente, color difuso, color especular, mapa de normales, mapa de transparencia, etc.). Actualmente se utilizan sólo texturas para color difuso dejando abierta la posibilidad en un futuro de implementar el uso de otros tipos de texturas.

En la estructura *material* también se agregó el color ambiente, difuso y especular (campos *colorAmb, colorDiff y colorSpec* respectivamente) que actualmente no se utilizan pero se encuentra implementado para futuros usos.

5.2 Importación y exportación de Modelos 3D

A continuación se describen las decisiones de implementación realizadas en el módulo de importación y exportación de modelos 3D. Se le añadió la capacidad de importar otros formatos incluidos el AC3D y la exportación a formato 3ds y dae (manteniendo la exportación a AC3D). También se incorporó el soporte de texturas.

Como Assimp no posee la exportación a formato AC se mantuvo la funcionalidad de exportación en este formato utilizado en RadTR 1.0.

Para unificar Assimp y la funcionalidad de exportación AC con el resto del software RadTR 2.0 se implementó una interfaz para cargar y guardar modelos 3D. Dicha interfaz posee dos métodos: uno para cargar modelos y otro para guardarlos. Los modelos son cargados en la estructura *scene* y *hierarchy*. De esta forma ya quedan listos para ser utilizados por RadTR 2.0.

Para la importación de modelos 3D se implementó el soporte de texturas. Assimp retorna en cada material un conjunto de texturas donde cada una es usada en diferentes tipos de iluminación (ambiente, difuso, especular, etc.). Nuestra implementación carga todas las texturas en la tarjeta gráfica utilizando OpenGL. La carga de todas las texturas puede ser ineficiente ya que actualmente sólo se utilizan las texturas difusas. Esto no es un problema

porque rara vez un modelo 3D posee texturas para otra clase de iluminación y además, en un futuro, será posible utilizar otros tipos de iluminación a base de texturas.

5.3 Módulo Main: seleccionar e iluminar superficies

En esta sección se describe paso a paso el mecanismo que utilizan las funcionalidades "Seleccionar superficie" e "Iluminar superficie" para seleccionar una superficie en pantalla.

1) Cuando el usuario realiza un click en pantalla se obtienen las coordenadas *x* e *y*.

2) Se dibuja la escena en modo código de color para poder distinguir cada superficie.

3) Se copia la pantalla a memoria RAM para poder obtener el color en el pixel con coordenadas *x* e *y* obtenidos en el paso 1.

4) El color RGB obtenido en el paso anterior se traduce en un número positivo que indica el índice de superficie, este número cuando es 0 (color negro) significa que no se seleccionó superficie.

5.4 Cálculo de los factores de forma

Se describen a continuación la implementación y algunas decisiones que se tomaron para el método del paraboloide y la mejora del rendimiento del hemicubo.

Cuando se quiere calcular el factor de forma de una superficie para cualquiera de los dos métodos (paraboloide y hemicubo). Los pasos a seguir son:

1) Colocar al observador sobre la superficie.
2) Dibujar toda la escena para calcular lo que se observa desde dicha superficie y así obtener fácilmente los factores de forma analizando la imagen resultado.

Realizando esto existe un problema cuando se coloca al observador sobre la superficie y al momento de dibujar la escena se dibuje dicha superficie frente al punto de vista y oculte a toda la escena. Para solucionar este inconveniente es posible dibujar toda la escena menos la superficie que se quiere calcular factor de forma pero no será posible usar *display list* (técnica usada para acelerar el dibujado) ya que si está construido para que dibuje toda la escena no hay forma de anular el dibujado de una sola superficie.

La solución utilizada es colocar al observador a una pequeña distancia de la superficie en dirección de la normal. De esta manera se tendrá seguridad de que la superficie quedará por detrás del observador y no aparecerá en el dibujado al utilizar *display list*.

5.4.1 Hemicubo

Se realizaron dos optimizaciones a la implementación original del método. Una mejora se llevó a cabo al momento de analizar la imagen final. Como es posible ver en la Figura 11 se tienen cuatro zonas (en las esquinas) donde el color siempre será negro. En el código original se analizaba el color de cada pixel de la imagen para encontrar superficies, de esta forma se analizaba innecesariamente las zonas en negro. Se realizó una mejora al código restringiendo la búsqueda de superficies sólo en los pixeles donde pueden estar las mismas y descartando las zonas en negro. Con esto se mejoró entre 2% y 5% el tiempo de ejecución dependiendo de la máquina y la escena.

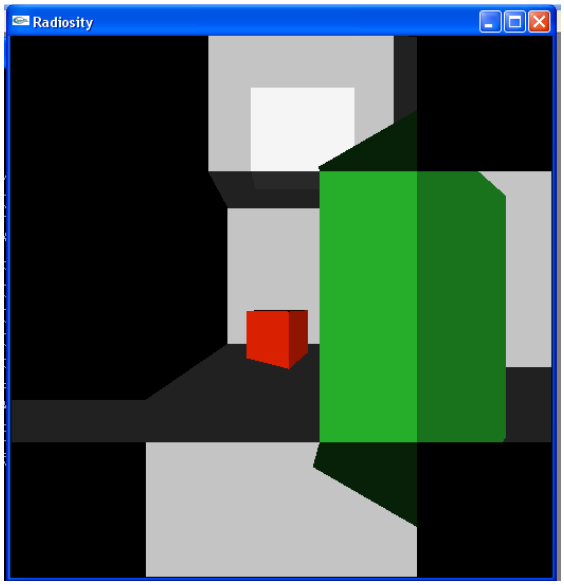


Figura 11: Las zonas negras en las esquinas no se analizan.

La otra mejora es utilizar *display list* de OpenGL en el dibujado de la escena. Esta técnica evita enviar a la tarjeta gráfica toda la geometría de la escena cinco veces por cada factor de forma que se desea calcular. Utilizando *display list* simplemente se realizan cinco llamados a la tarjeta gráfica ya que en un *display list* se encuentra toda la geometría de la escena.

5.4.2 Paraboloide

La clase Paraboloide brinda los mismos servicios que hemicubo. La ventaja que tiene frente a hemicubo es que solo es necesario dibujar una vez la escena para calcular los factores de forma contra las cinco veces del hemicubo. Para que esto sea posible es necesario que la tarjeta de video soporte *shaders*, específicamente *vertex* y *fragment shader* y opcionalmente *geometry shader*.

Paraboloide tiene el mismo funcionamiento que el hemicubo. Dibuja la escena de tal manera que el observador tenga visibilidad de la misma en 180º (ver Figura 12) y luego analiza

cada pixel para calcular los factores de forma de las superficies. La diferencia está en que cuando realiza el dibujado de la escena utiliza un *shader* para distorsionar la escena proyectándola en un paraboloide y así, en un solo dibujado, el observador tiene visible 180º de la escena.

Cuando se recorren todos los pixeles de la imagen final para calcular los factores de forma, a diferencia del hemicubo, el área de la imagen es un círculo. Por esta razón se recorre toda la imagen ya que es más difícil de excluir las zonas en negro eficientemente.

La distorsión de los factores, a diferencia del hemicubo (ver anexo Hemicubo), es homogénea en todo el círculo del paraboloide y en la zona exterior tiene un peso de 0. Por esta razón no es necesario calcular la distorsión de los factores de forma.

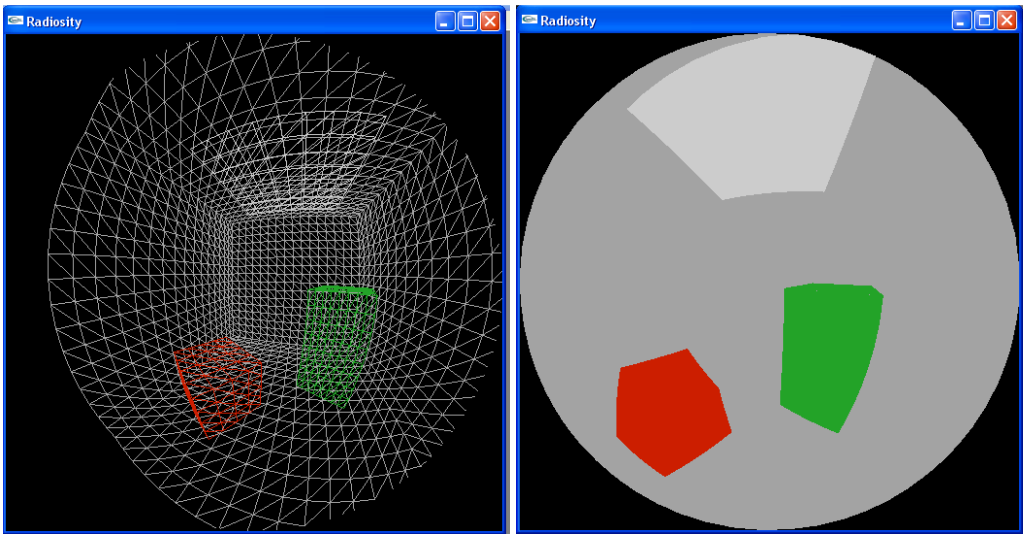


Figura 12: Izquierda, vista en paraboloide en modo wireframe. Derecha, vista normal en paraboloide.

5.4.3 Shaders

La clase Shaders es la encargada de inicializar y cargar los *shaders*. Estos *shaders* son utilizados para generar el paraboloide.

Los shaders se clasifican jerárquicamente de la siguiente forma:

1) *Vertex Fragment Shaders*, se encuentran solo los *shaders* de vértice y de fragmento (o píxel). Tiene el fin de proyectar la escena en un paraboloide.
2) *Geometry Shaders*, agrega un *shader* geométrico a la clasificación anterior. Además de proyectar la escena en un parabolide también realiza división de los triángulos.
3) *Tessellation Shader*, agrega el *shader* de teselado a la clasificación de geometría. Proyecta la escena en un paraboloide y realiza división de triángulos eficientemente.

5.5 Módulo matemático

A causa de la necesidad de realizar operaciones matriciales se desarrolló un módulo matemático al comienzo del proyecto. Dicho módulo implementa varias operaciones matriciales como la multiplicación, suma, transpuesta e inversa. Para acelerar la operación de multiplicación (es la operación que más tiempo consume) se implementó una versión en CUBLAS y se paralelizó la implementación en CPU. Se realizaron pruebas para analizar el rendimiento de la operación multiplicación de las cuales se desprende que sin paralelización en CPU tardó 400 segundos mientras que con paralelización con 2 *cores* disminuyo el tiempo a 218 segundos (un *speedup* de 1.83). Utilizando CUBLAS tardó 35 segundos, 11 veces más rápido comparado con la implementación serial en CPU.

Al final del proyecto se incorporó la librería BLAS para realizar la multiplicación eficiente de matrices. Se utilizó la implementación Armadillo de BLAS.

El módulo Matemático también posee un método para generar las matrices U y V llamado *calcularUV*. Este método recibe como argumentos la escena, la jerarquía y varios parámetros de configuración para las matrices U y V (como son la cantidad de niveles, si utilizar hemicubo o paraboloide, etc.).

5.6 Radiosidad en tiempo real

Uno de los objetivos del proyecto es la implementación de un algoritmo que sea capaz de calcular la iluminación de una escena estática (la geometría no varía) utilizando radiosidad en tiempo real. Para lograr esto se utilizó el poder de cálculo de las tarjetas gráficas, más específicamente la tecnología CUDA. También se implementó el mismo algoritmo para ejecutarlo en CPU y MATLAB.

El módulo que lo implementa es *RadiosidadTiempoReal*. Para su funcionamiento es necesario asignarle la escena. El módulo consta de un método para este fin que se llama *setEscena* que recibe la escena y la jerarquía. También existe el método *limpiarEscena* que quita la escena actualmente asignada. Luego de especificar una escena es posible iniciar RTR (Radiosidad en Tiempo Real). Para esto se utiliza el método *iniciarRTR* que recibe como parámetros: plataforma donde realizar los cálculos (CPU, GPU, MATLAB *Compiler* o MATLAB *Coder*), parámetros para el cálculo de las matrices U y V, cómo realizar el cálculo de los factores de forma (hemicubo o paraboloide) y por último si se debe calcular la iluminación con color RGB o monocromática.

Como se comentó en la sección de desarrollo, es posible aplicar iluminación monocromática o con color. La monocromática considera el color de la luz como blanca en todo momento y es útil para ver cómo la luz interacciona con la escena sin considerar su color. Sin embargo, en la realidad la luz posee un color que es cambiante cuando interacciona con superficies con color. Por este motivo también se implementó un segundo método donde se considera a la luz con color. Este último método es más lento que con luz monocromática ya

que tiene que procesar tres valores (RGB) en lugar de uno pero es de gran utilidad para observar a la escena con una iluminación más fiel a la realidad.

El método *getIluminacion* calcula la iluminación de la escena asignada utilizando radiosidad. Los parámetros que recibe son la posición, dirección, intensidad y tamaño de la fuente de luz. Retorna la iluminación final de cada superficie de la escena.

También se encuentra el método *getUltimaIluminacion* que retorna la última iluminación calculada. Este método no realiza el cálculo de la iluminación.

Por último está el método *finalizarRTR* que finaliza RTR liberando recursos.

Es importante describir la forma de almacenamiento de las matrices. En nuestra aplicación se guardan de forma lineal (un array) porque es una forma eficiente de almacenarlas ya que el acceso a un elemento de la matriz se realiza directamente. Si se desea acceder al elemento *(i,j)* la posición en el array corresponde al índice *i+j*largo*. Observar que se almacenan en el array las filas de la matriz. Los elementos de una fila se encuentran adyacentes. Esta forma de almacenamiento es diferente al que utiliza CUBLAS y MATLAB ya que ellos guardan por columnas, por esta razón cuando se transfiere una matriz *M* desde la aplicación hacia CUDA o MATLAB hay que tratarla como M^T.

A continuación se describe con detalle los pasos de la implementación de la función *iniciarRTR:*

1) Verifica que se tiene la suficiente memoria RAM para realizar los cálculos, se calcula la cantidad de MB necesarios para alojar una matriz de alto cantidad de triángulos de la escena y de ancho cantidad de nodos de la jerarquía. Si esa cantidad supera un límite establecido entonces retorna con error. Tener en cuenta que este límite de memoria se puede cambiar si se trabaja con una computadora con otra cantidad de memoria RAM ya que este límite fue hallado en la práctica utilizando 2 GB de RAM.

2) En caso de que se quiera utilizar CUDA o MATLAB verifica que la máquina los soporte, si no lo soporta se configura para usar la CPU.

3) Reserva memoria para el vector E de tamaño la cantidad de superficies, este vector contendrá la iluminación inicial de la escena.

4) Calcula la matriz *Y*. Esta matriz se calcula como $Y = -U * inversa(I-V^TU)$ para lo cual primero obtiene las matrices *U* y V^T utilizando el método *calcularUV* del módulo Matemático. Este método recibe la escena, la jerarquía, si se desea utilizar paraboloide o hemicubo en el cálculo de factores de forma y si se desea utilizar dos niveles o multinivel en la generación de las matrices. En caso de utilizar más de dos niveles se utilizan otros parámetros auxiliares. Cuando se utilizan dos niveles genera directamente *U* y V^T. En el caso multinivel estas dos matrices hay que generarlas ya que *calcularUV* retorna los valores de *V* y de *U* separadamente. Posteriormente, se calcula la matriz Y.

5) Se comprime la matriz *V*. Luego de la compresión se obtienen dos vectores, uno llamado *Iarray* con los índices de cambio de columna y otro *Ivalores* con los valores de la matriz *V*. Cuando se utilizan dos niveles los valores de *V* serán 1's pero no sucede lo mismo cuando se utilizan más de dos niveles.

6) Si se usa GPU (CUDA) inicializa memoria en la tarjeta de video, transfiere *Y*, *Iarray* e *Ivalores* (*V* comprimida) y reserva memoria para el vector de iluminación inicial *E* y el vector de iluminación final *B*. Notar que la matriz *Y* transferida en CUBLAS se tratará como si fuera Y^T.

   En caso de usar MATLAB también se transfiere Y pero a diferencia de CUBLAS, antes de transferir *Y* se le realiza la transpuesta. Por esta razón los cálculos en MATLAB se realizan con *Y* y no con Y^T. También se transfiere V comprimida (los índices *Iarray* y los *Ivalores*) y se reserva memoria para la emisión inicial *E*.

7) Por último, el método *iniciarRTR* crea un vector *B* que es usado para guardar la iluminación final de la escena.

Si se invoca a la función *iniciarRTR* cuando ya se encuentra iniciado no realiza ninguna acción y retorna satisfactoriamente.

A causa de largos tiempos de cálculos de las matrices *Y* y *V* comprimida (los índices *Iarray* y los *Ivalores* de *V*) se realizó una mejora al método *iniciarRTR* que evita el recálculo de dichas matrices. Cuando finaliza RTR (se invoca a *finalizarRTR*) no elimina dichas matrices. Cuando se vuelve a invocar *iniciarRTR* verifica si se quiere iniciar con la misma configuración, o sea, la misma cantidad de niveles en la generación de las matrices *U* y *V*, en caso de usar multinivel los parámetros sean los mismos y se quiere usar el mismo método del cálculo de los factores de forma (paraboloide o hemicubo). Si se verifica que sí, no se realizan los cálculos de las matrices y se utilizan las matrices calculadas anteriormente. Si la configuración cambia entonces vuelve a calcular las matrices.

El método *getIluminacion* primero realiza verificaciones en busca de problemas (RTR no iniciado, matrices necesarias no calculadas). Luego calcula la emisión inicial con la función *calcularEmisionInicial*. Dicha función recibe la fuente de luz (posición, dirección, intensidad y tamaño) y calcula la iluminación inicial en la escena. Para realizar este cálculo configura al observador como la fuente de luz y dibuja toda la escena, de esta manera las superficies que aparecen en pantalla son las que estarán iluminadas inicialmente. Luego, según el tipo de cálculo (CPU, GPU, MATLAB *Compiler* o MATLAB *Coder*), se invoca al método correspondiente que genera la iluminación de la escena a partir de la iluminación inicial. Por último, retorna la iluminación final en el vector *B*.

Existen cuatro funciones de cálculo de la radiosidad en tiempo real utilizando CPU, GPU, MATLAB Compiler y MATLAB Coder. Dichas funciones son *aplicarIluminacionCPU, aplicarIluminacion, aplicarIluminacionMatlab* y *aplicarIluminacionMatlabCoder* respectivamente. Reciben como parámetros el largo y alto de las matrices *Y* y V. El largo es igual a la cantidad de nodos de la jerarquía mientras que el alto es la cantidad de superficies de la escena. También reciben la iluminación inicial *E*, la matriz *Y* y la matriz *V* comprimida (los índices *I* y los valores). Al finalizar retornan la iluminación final *B*. Las cuatro implementaciones realizan el mismo cálculo matricial, $B = V^T * E$ y luego $B=E-Y*B$.

Para calcular $B=V^T*E$ no es necesario realizar la multiplicación de matrices ya que *V* está comprimida. Cada elemento del vector *B* se calcula como la suma de los productos de los elementos de *E* y de los valores de *V*. Estos elementos son hallados desde el vector *Iarray* en el que se encuentran los índices. La implementación en C++ se encuentra en el anexo "Radiosidad en tiempo real código en CUDA y C".

La implementación en CUDA del cálculo $B=V^T*E$ es similar a la implementación en CPU. Cada hilo CUDA calcula un elemento del vector *B* (ver anexo "Radiosidad en tiempo real, código en CUDA y C").

El usuario tiene la posibilidad de asignar la cantidad de hilos por bloque. Si el tamaño de *B* no es múltiplo de la cantidad de hilos por bloques, se ejecutarán más hilos que el tamaño de *B*, por esta razón algunos hilos no realizarán nada.

El cálculo de *B=E-Y*B* es sencillo ya que es simplemente una multiplicación y una resta de matrices.

Al final del proyecto se incorporó la opción de utilizar la librería BLAS para la multiplicación eficiente de matrices en CPU.

5.7 Iluminación del cielo

El módulo *IluminacionCielo* se encarga de calcular la iluminación de la escena donde la fuente de luz es el cielo, dicho módulo se crea pasando como parámetro la escena y la resolución de imagen utilizada para calcular los factores de forma.

Posee un método *setParalelo* que es posible definir las dimensiones de la vista en paralelo (izquierda, derecha, arriba, abajo, lejano y cercano).

Tiene el método *factorFormaParaleloDirecciones* que dado un conjunto de puntos en el cielo retorna cuantos pixeles de cada superficie de la escena se observa en cada punto. Los puntos de vista están definidos por una posición y una dirección hacia el centro de la escena. La implementación es sencilla, posiciona al observador en cada punto y dibuja la escena con la vista en paralelo, luego suma la cantidad de pixeles de cada superficie. Cuando se dibuja la escena hay que dibujar también las caras traseras de las superficies. En todos los casos cuando se dibujaba la escena solo se dibujaba la cara frontal para disminuir el tiempo de dibujado. Con esto no había problemas porque todas las caras "miraban" hacia adentro de la escena, ahora cuando se calcula los factores de forma observando desde el cielo algunas superficies muestran su parte trasera haciendo que no se dibuje. Esto es un problema porque se podría ver a través de las paredes de la escena. Para evitarlo se calcula el ángulo para cada superficie entre la normal y la vista del observador, si es menor a 90º (se encuentra enfrentada) se dibuja con el color normal (código de color), en caso que sea mayor a 90º se encuentra detrás y se dibuja con un color negro. De esta forma las paredes no mostrarán el contenido de la escena.

Para crear los puntos en el cielo se definen dos ángulos, uno *alfa* que aplica al azimuth y *beta* a la altitud. Comienza con un ángulo inicial de *PI/2* en altitud y 0 de azimuth, el algoritmo aumenta alfa al azimuth y cuando llega a *2*PI* lo reinicia en 0 y disminuye beta a la altitud. En cada paso crea un punto en el cielo donde la dirección apunta al centro *(0,0,0).*

Si se quiere realizar zoom a la escena utilizando las teclas '+' y '-', en la proyección paralela se aumenta o disminuye respectivamente las dimensiones de la vista (izquierda, derecha, arriba y abajo).

En la Figura 13 se muestra la escena Cornell Box con vista en proyección paralela.

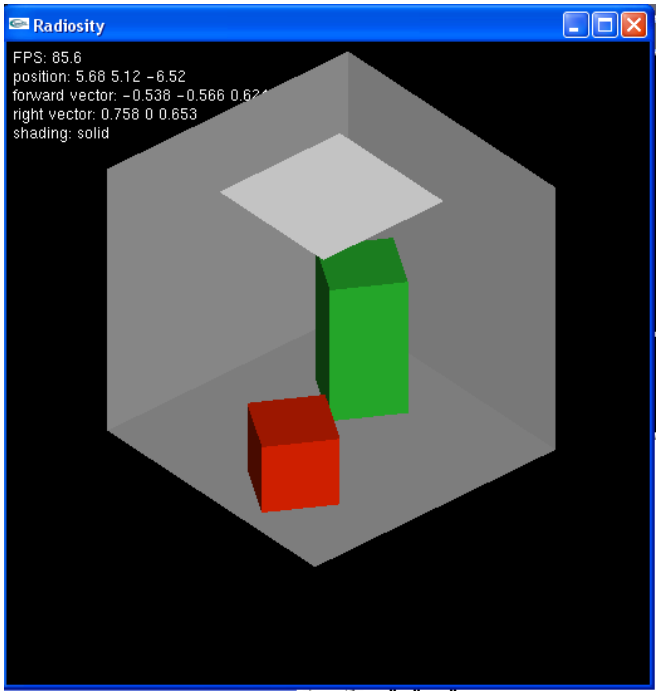


Figura 13: Vista en proyección paralela.

5.8 Interfaz gráfica

Al ejecutar el programa se inicia el módulo encargado de la interfaz gráfica. Este módulo carga la interfaz gráfica (ventanas gráficas, menús, etc.), OpenGL y los *callbacks* (teclado o mouse). El módulo, al inicializar, recibe punteros a funciones donde cada función es la encargada de realizar una funcionalidad del software como por ejemplo subdividir escena, calcular radiosidad, etc. Estas funciones son invocadas desde el módulo cuando el usuario a través de la interfaz gráfica desea aplicarlas.

El módulo UI ejecutará durante toda la ejecución del programa. Es el encargado de orientar la lógica del software. Esto quiere decir que se encarga de desplegar la ventana correcta según la funcionalidad que se quiera ejecutar, de controlar la entrada de información a través de mouse y el teclado, de lanzar correctamente la funcionalidad que el usuario indique, etc.

Existen algunas funcionalidades que utilizan OpenGL (por ejemplo, el cálculo de los factores de forma) que deben ser ejecutadas en ausencia de elementos de glut para su correcto funcionamiento. Si existe algún elemento de glut activo OpenGL dibuja en dicho elemento en vez de la ventana principal. Esto provoca errores visuales ya que los elementos de glut son más pequeños. Para solucionar el problema en un principio se utilizó la función *idle* de glut que es ejecutada varias veces por frame. Cuando se quiere, por ejemplo, calcular la radiosidad de la escena, primero se programa que se desea ejecutar esa función, luego se desactivan las ventanas glut y la función *idle* se encarga de ejecutar radiosidad cuando ya no

existan ventanas de glut activas, evitando así el error descripto anteriormente. Con esta solución se encontró un problema de rendimiento ya que la función *idle* consumía mucho tiempo de CPU llegando a utilizar la mitad del procesador en algunos casos. La solución elegida es programar las tareas que se deben ejecutar en ausencia de elementos de glut y que en cada frame el módulo de interfaz gráfica, luego de desactivar glut, ejecute dichas tareas.

5.9 Fichero de configuración

Existe el fichero *config.h* donde se encuentran ciertas configuraciones del software. Es posible configurar algunos parámetros relacionados con el dibujado de la escena, los factores de forma, radiosidad en tiempo real, etc. Para más detalle ver el anexo "Configuración de RadTR 2.0".

5.10 MATLAB y RadTR 2.0

A continuación se describe cómo fueron implementadas las interfaces para la interacción entre MATLAB y RadTR 2.0.

5.10.1 Interfaz mediante archivos

Carga Secuencial

Se debe tener en un directorio en el *file system* un conjunto de archivos que representan la matriz emisión del Algoritmo Radiosity. Además, se debe tener una escena cargada en la aplicación previamente. Los archivos de luz deben ser compatibles con la escena.

Al iniciar la funcionalidad se crea un *thread* que ejecutará paralelo al programa, junto a variables globales para la comunicación de información entre el *thread* y el hilo principal de RadTR 2.0. Este *thread* se encarga de ir cargando los archivos, generar la estructura interna de luminosidad para la escena, y comunicársela al hilo principal que tomará el cambio y renderizará de acuerdo vaya recibiendo los cambios en dicha estructura. El hilo principal se encargará del renderizado y guardado de imágenes de salida.

Carga periódica

Como precondición, debe existir en un directorio del *file system* un archivo que representan la matriz de emisión del algoritmo Radiosity. Además, se debe tener una escena cargada en la aplicación previamente. El archivo de emisión y sus versiones en el correr del tiempo deben ser compatibles con dicha escena.

Al iniciar la funcionalidad se crea un *thread* que ejecutará paralelo al programa, junto a variables globales para la comunicación de información entre el *thread* y el hilo principal de RadTR 2.0. Este *thread* se encarga de ir cargando el archivo definido por los datos de entrada, generar la estructura interna de luminosidad para la escena, y comunicársela al hilo principal que tomará el cambio y renderizará de acuerdo vaya obteniendo los cambios en dicha estructura.

El hilo principal se encargará del renderizado y presentación en el display del programa.

5.10.2 Utilización del compilador MATLAB

En este apartado se indican las decisiones con respecto a la generación de librerías mediante el compilador MATLAB, así como también de las invocaciones de las mismas.

Generación de librerías compartidas

Existen dos alternativas al momento de elegir el lenguaje en que se generarán las librerías compartidas: C o C++.

Como RadTR 2.0 está escrito en C++, se tiene la posibilidad de generar la librería en dicho lenguaje. Pero esto implicaría que se deba utilizar el tipo de datos *mwArray* y no *mxArray* para la representación de las matrices MATLAB. Por otro lado, las funciones de intercambio de datos del MCR utilizan el tipo de datos *mxArray*, que es la forma de representación de una matriz MATLAB en el lenguaje C. Esto llevaría a que antes de realizar el pasaje de datos se hiciera algún tipo de conversión de los mismos. Si se suma todo esto a que dicha conversión se realizaría en cada paso de una iteración que se ejecuta en la generación cada *frame* se estaría agregando un costo adicional que, por mínimo que sea, es indeseado.

Para evitar esto, se generan librerías en C y se mantiene un solo tipo de datos para representar las matrices MATLAB: el *mxArray*.

En el Anexo L se muestra un ejemplo de generación de librerías compartidas.

6 Pruebas

En esta sección se presentan las pruebas de rendimiento y calidad al software RadTR 2.0.

La computadora que se usó para las pruebas tiene las siguientes características:

- Tarjeta de video NVIDIA GeForce GTS 450.
- Procesador Pentium Dual Core 3GHz.
- Memoria RAM de 4 GB.
- Sistema operativo Windows 7 de 64 bits.

Observación:

El sincronizador vertical (vsync), es una utilidad que viene en los drivers de las tarjetas gráficas para ajustar la cantidad de FPS dibujados en pantalla con la frecuencia del monitor. Dicha herramienta es utilizada para evitar que aplicaciones gráficas consuman más recursos de lo necesario. En estos casos, al realizar pruebas de rendimiento, se decidió desactivarlo para poder leer correctamente la cantidad de FPS de la aplicación.

6.1 Paraboloide vs Hemicubo

En esta prueba se desea analizar el rendimiento del cálculo de los factores de forma al utilizar paraboloide y hemicubo.

A continuación se muestran tiempos de ejecución al calcular radiosidad utilizando paraboloide y hemicubo con diferentes configuraciones de escenas, iteraciones de radiosidad y *display list*.

Escena Cornell Box

Cantidad de triángulos / cantidad de iteraciones	**Segundos con Paraboloide (Con *display list* / Sin *display list*)**	**Segundos con Hemicubo (Con *display list* / Sin *display list*)**
3.584 / 1.000	7.5 / 7.7	7.5 / 9.5
3.584 / 20.000	146/156	146 / 190
51.200 / 1.000	9.8 / 16.6	11.7 / 46.4
51.200 / 20.000	190 / 327	228 / 925

Tabla 3: Tiempos de ejecución en la escena Cornell Box.

Escena Sponza Atrium

Cantidad de triángulos / cantidad de iteraciones	Segundos con Paraboloide (Con display list / Sin display list)	Segundos con Hemicubo (Con display list / Sin display list)
70.335 / 1.000	9.4 / 18.6	10.5 / 59
70.335 / 20.000	193.5 / 374	205 / 1192

Tabla 4: Tiempos de ejecución en la escena Sponza Atrium.

Cuando se utiliza *display list* en escenas pequeñas el paraboloide y el hemicubo consumen el mismo tiempo de ejecución. La explicación a esto es que la cantidad de triángulos en ambos casos es muy pequeña como para observar diferencias en el rendimiento. En escenas más grandes paraboloide es sensiblemente más rápido que hemicubo porque la tarjeta gráfica tiene que dibujar 4 veces menos triángulos y con escenas grandes se ahorra el renderizado de muchos triángulos. Cuando no se utiliza *display list* en escenas grandes paraboloide es alrededor de 3 veces más rápido porque utiliza 4 veces menos ancho de banda de la tarjeta y dibuja 4 veces menos de triángulos.

En la Figura 14 se muestran tres imágenes donde se calculó la iluminación de una escena utilizando hemicubo, paraboloide y paraboloide con teselado.

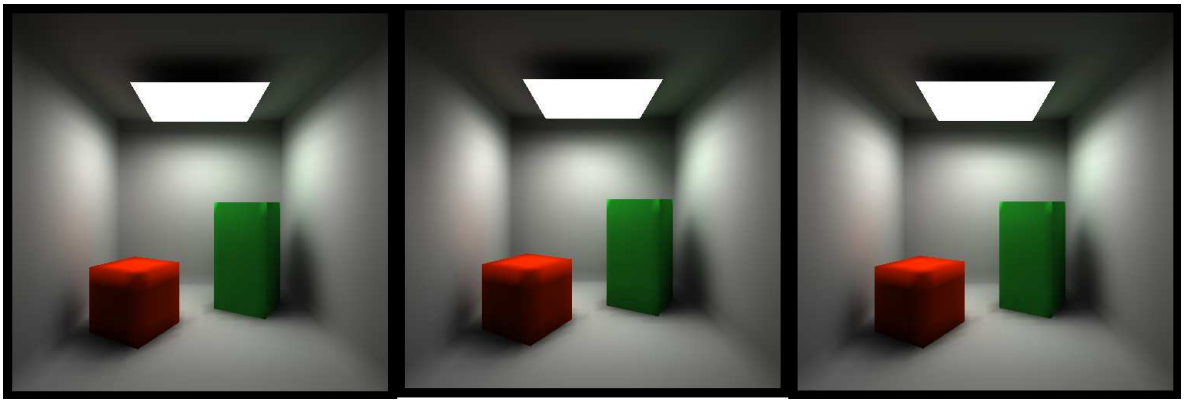


Figura 14: Utilización de diferentes métodos para el cálculo de los factores de forma al calcular radiosidad. Izquierda: Hemicubo. Centro: Paraboloide. Derecha: Paraboloide con división de teselado.

A simple vista no se distinguen diferencias en la iluminación generada por hemicubo, paraboloide y paraboloide con teselado. Pero detalladamente se puede observar que en paraboloide las zonas de la unión de las paredes son un poco más oscuras que hemicubo. Sin embargo, este problema se corrige cuando se utiliza paraboloide con teselado.

Con estos datos se puede observar que paraboloide es más rápido que hemicubo en escenas grandes, también es más rápido en cualquier circunstancia si no se utilizan los *display list* de OpenGL. Sin embargo, tiene como desventaja la necesidad de hardware especial para

ejecutarse. Otra desventaja al observar las imágenes es la generación de factores de forma con un sensible error. Sin embargo, con *tessellation shaders* es posible corregirlo.

6.2 Paraboloide y subdivisión en *geometry shader*

Se analiza el rendimiento de utilizar diferentes tipos de subdivisiones a nivel de *geometry shader* y *tessellation shader* en el cálculo de factores de forma y como mejora la apariencia visual, para esto se aplica radiosidad a dos escenas utilizando paraboloide.

A continuación se muestran los tiempos de ejecución del algoritmo de radiosidad con diferentes tipos de subdivisión para las escenas Sponza Atrium y Cornell box.

Sponza Atrium, 68430 triángulos.

Iteraciones	Sin división	Triángulo	Centro	Homogénea	Diagonal	Teselado con 4 de división (16 subtriángulos)
10.000	286 s	1622 s	324 s	544 s	1400 s	417 s

Tabla 5: Tiempos de ejecución en diferentes tipos de subdivisión para la escena Cornell Box.

Cornel Box, 3200 triángulos.

Iteraciones	Sin división	Triángulo	Centro	Homogénea	Diagonal	Teselado con 4 de división (16 subtriángulos)
1.000	8.7 s	16.6 s	9.4 s	11.3 s	13.6 s	8.9 s

Tabla 6: Tiempos de ejecución en diferentes tipos de subdivisión para la escena Cornell Box.

Con estos datos se observa que el *shader* de división de triángulos por el centro y de teselado son los más rápidos mientras que división diagonal y por triángulo son los más lentos, siendo este último el más lento de todos llegando a consumir hasta 5.6 veces más de tiempo que el *shader* sin división.

En la Figura 15 se muestra las diferencias visuales entre los diferentes tipos de subdivisión en la escena Cornell Box.

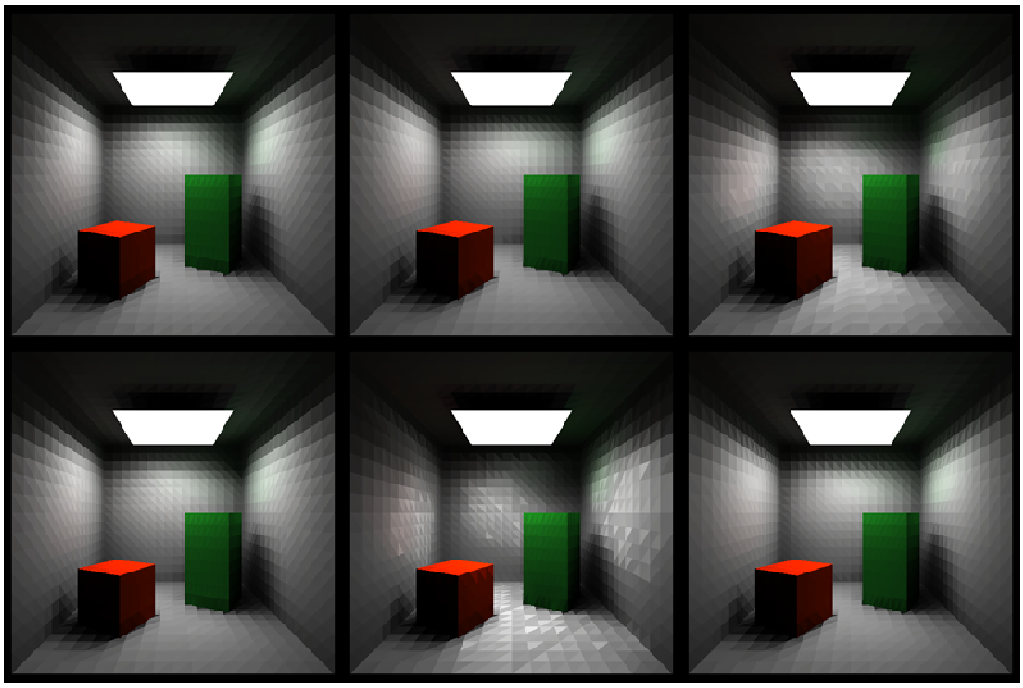


Figura 15: Radiosidad a una escena con 3200 triángulos y 1.000 iteraciones con división a nivel de shader. De izquierda a derecha y de arriba abajo: Sin división, división de triángulos, centro del triángulo, homogénea, diagonal y con teselado.

Todas las imágenes parecen ser correctas visualmente, sin embargo, existen pequeñas diferencias y errores en algunas de ellas. La imagen con división diagonal es la que más fallas tiene ya que existen sombras irregulares en las paredes y en las esquinas. La otra que tiene fallas es la del centro del triángulo, ya que se ven de forma irregular las sombras en la pared de atrás. En la que no se utiliza división también se aprecia la misma falla pero de forma más leve. Las divisiones por triángulos y con teselado presentan una iluminación más homogénea en toda la escena, dando mejores resultados. Si se observa bien la división con teselado, visualmente está mejor iluminada que la de triángulo, y sumado que el teselado es entre dos y cuatro veces más rápido se puede afirmar que es la mejor opción para subdividir triángulos en la tarjeta gráfica. Sin embargo, en escenas grandes, el teselado consume el doble de tiempo comparado con el *shader* que no subdivide y es necesario de hardware que soporte *tessellation shader*.

6.3 Radiosidad en tiempo real

Se desea analizar cómo se comporta el algoritmo de radiosidad en tiempo real con diferentes escenas en tamaño y ubicación geométrica, también probar diferentes métodos de cálculos (CPU, GPU, MATLAB *Compiler*, MATLAB *Coder*). Como resultado final se quiere saber cómo impacta los tamaños y forma de las escenas en el tiempo de cálculo y como se comporta cada método, también es importante observar las iluminaciones finales calculadas. En estas pruebas se utilizó la librería BLAS para la multiplicación de matrices en el método CPU.

6.3.1 Escena Cornell Box

La escena utilizada en esta prueba está compuesta por 3.584 triángulos.

La siguiente tabla muestra el tiempo que tarda en iniciar radiosidad en tiempo real.

Etapas	CPU	GPU*
Generación U y V.	49.497 ms	48.801 ms
V'*U	509 ms	99 ms
I-V'*U	2 ms	2 ms
Inversa(I-V'*U)	13 ms	13 ms
-U	6 ms	5 ms
Y=-U*inversa(I-V'*U)	487 ms	105 ms
Comprimir V	13 ms	13 ms
Total de iniciación	**50.527 ms**	**49.038 ms**

Tabla 7: Tiempo que tarda en iniciar radiosidad en tiempo real con iluminación color en la escena Cornell Box. Observación (*): En el caso GPU solamente se calculan en CUDA las etapas V'*U y Y=-U*inversa(I-V'*U), el resto se ejecuta en CPU.

Iluminación con color

En la siguiente tabla se muestra el porcentaje del tiempo consumido por el algoritmo de radiosidad en tiempo real y la cantidad de FPS. En MATLAB *Coder* no se implementó el algoritmo para iluminación con color.

	CPU	GPU	MATLAB Compiler	MATLAB Coder
Porcentaje del tiempo consumido	53 %	23 %	87 %	-
FPS	75	115	19	-

Tabla 8: Porcentaje de tiempo consumido por el algoritmo de radiosidad en tiempo real utilizando iluminación con color en la escena Cornell Box.

Iluminación sin color

Ejecución del algoritmo de radiosidad en tiempo real. Se muestra el porcentaje del tiempo consumido por el algoritmo y la cantidad de FPS.

	CPU	GPU	MATLAB Compiler	MATLAB Coder
Porcentaje del tiempo consumido	24 %	12 %	23 %	46 %
FPS	115	140	115	79

Tabla 9: Porcentaje de tiempo consumido por el algoritmo de radiosidad en tiempo real utilizando iluminación sin color en la escena Cornell Box.

En la Figura 16 se muestra la diferencia del algoritmo aplicando iluminación con color y sin color.

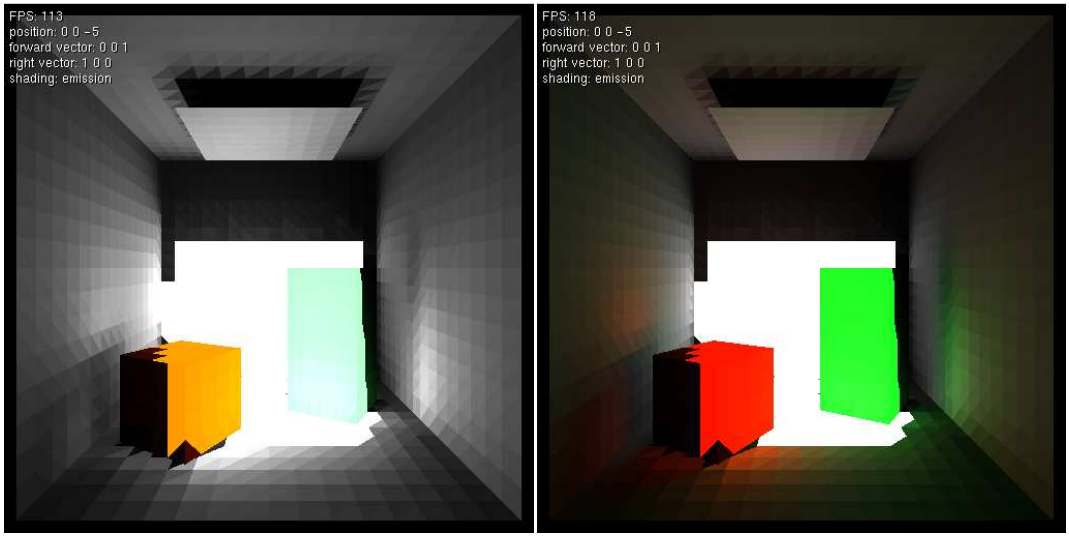


Figura 16: Resultado de radiosidad en tiempo real, se aplico un aumento de la iluminación de 5 en ambas imágenes (por esa razón parece quemada la imagen izquierda), izquierda iluminación sin color, derecha iluminación con color

6.3.2 Escena LivingLiteColor

La escena utilizada en esta prueba está compuesta por 9.696 triángulos. La siguiente tabla muestra el tiempo que tarda en iniciar radiosidad en tiempo real.

Etapas	CPU	GPU*
Generación U y V.	241.137 ms	243.686 ms
V'*U	292.407 ms	29.920 ms
I-V'*U	1.139 ms	705 ms
Inversa(I-V'*U)	111.598 ms	117.336 ms
-U	212 ms	204 ms
Y=-U*inversa(I-V'*U)	293.257 ms	29.872 ms
Comprimir V	854 ms	783 ms
Total de iniciación	**940.604 ms**	**422.302 ms**

Tabla 10: Tiempo que tarda en iniciar radiosidad en tiempo real con iluminación color en la escena LivingLiteColor. Observación (*): En el caso GPU solamente se calculan en CUDA las etapas V'*U y Y=-U*inversa(I-V'*U), el resto se ejecuta en CPU.

Iluminación con color

La siguiente tabla pertenece a la ejecución del algoritmo de radiosidad en tiempo real. Muestra el porcentaje del tiempo consumido por el algoritmo y la cantidad de FPS. En MATLAB *Coder* no se implementó el algoritmo para iluminación con color y en MATLAB *Compiler* no fue posible realizar la prueba ya que la escena consumía mucha memoria.

	CPU	GPU	MATLAB Compiler	MATLAB Coder
Porcentaje del tiempo consumido	97 %	74 %	-	-
FPS	4	29	-	-

Tabla 11: Porcentaje de tiempo consumido por el algoritmo de radiosidad en tiempo real utilizando iluminación con color en la escena LivingLiteColor.

Iluminación sin color

Ejecución del algoritmo de radiosidad en tiempo real, muestra el porcentaje del tiempo consumido por el algoritmo y la cantidad de FPS.

	CPU	GPU	MATLAB Compiler	MATLAB Coder
Porcentaje del tiempo consumido	91 %	51 %	-	-
FPS	10	58	-	-

Tabla 12: Porcentaje de tiempo consumido por el algoritmo de radiosidad en tiempo real utilizando iluminación sin color en la escena LivingLiteColor.

En la Figura 17 se muestran los resultados de radiosidad en tiempo real aplicando iluminación con color y sin color.

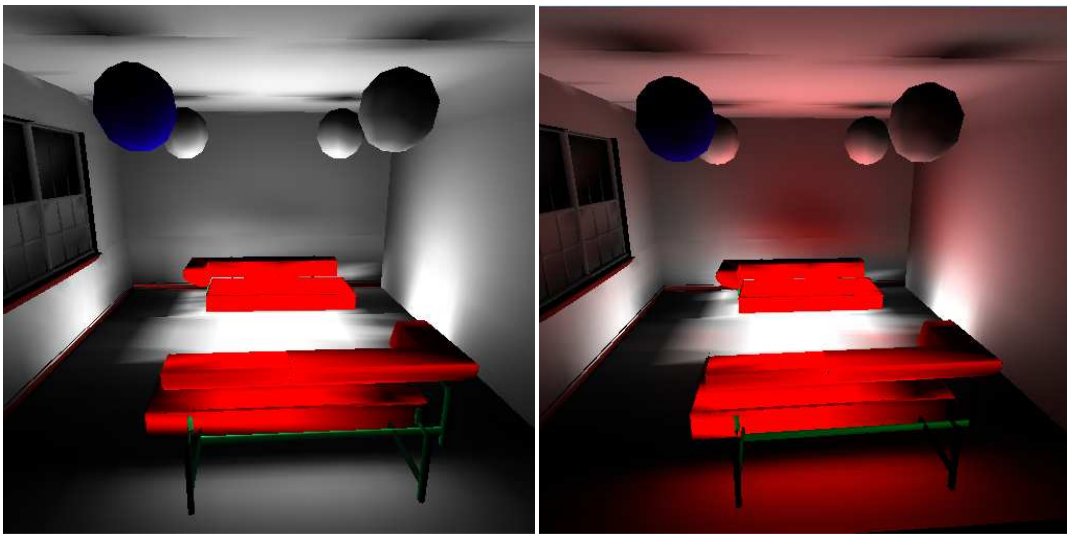


Figura 17: Resultado de radiosidad en tiempo real, izquierda iluminación sin color, derecha iluminación con color.

Sobre el tiempo de inicialización del algoritmo cuando ejecuta en GPU los productos son cinco veces más rápidos con pocos triángulos y con muchos triángulos llega a ser hasta diez veces más rápidos que en CPU. La razón de esto es porque se utiliza la librería CUBLAS en las multiplicaciones de matrices en GPU.

Es importante notar que la inicialización es un proceso que requiere mucho tiempo de cálculo y mucha memoria. En la escena LivingLiteColor en CPU demoró 15 minutos (en GPU 7 minutos) y no fue posible ejecutar el algoritmo en MATLAB por la gran cantidad de memoria que ocupan las matrices. Las etapas que duraron más en la inicialización son la generación de las matrices U y V, el producto *V'*U* y el producto *-U*inversa(I-V'*U)*.

6.3.3 Observaciones

Comparando la ejecución del algoritmo con iluminación sin color en la escena Cornell Box, como dato curioso, se puede observar que MATLAB *Coder* es menos eficiente que MATLAB *Compiler*. Este dato puede ir en contra de las suposiciones realizadas, ya que MATLAB *Coder* está en código C y compilado para CPU mientras que MATLAB *Compiler* utiliza librerías MATLAB para ejecutar. Analizando la posible causa de esto se puede inferir que las operaciones matriciales en MATLAB *Compiler* pueden llegar a estar más optimizadas que el código C generado. Otra posibilidad, que se menciona en el desarrollo, puede ser que las invocaciones en cada *frame* son distintas para MATLAB *Compiler* y MATLAB *Coder*. Mientras en el primero no se pasan parámetros en la llamada, ya que se intercambian datos de usuarios, en el segundo sí, aunque sean sólo punteros.

El algoritmo en GPU es el más rápido en todos los casos alcanzando 1.2 veces más FPS que CPU en el peor caso en la escena Cornell Box y en el mejor caso 7.3 veces más FPS en la escena LivingLiteColor con color. El algoritmo de MATLAB *Compiler* en la escena Cornell Box alcanza el mismo rendimiento que CPU con iluminación sin color pero cuando se utiliza iluminación con color es más lento.

Otro dato a analizar es el porcentaje del tiempo de ejecución del algoritmo comparado en base a los FPS. Para la escena LivingLiteColor, aplicando iluminación con color, el algoritmo en GPU consume 74,0 % mientras que en CPU 97 %. El algoritmo en GPU deja 26,0 % para el renderizado de la escena mientras que CPU deja 3 %. Haciendo cuentas con la cantidad de FPS da como resultado que el renderizado de cada FPS tarda cerca de 10 ms. Con esto se quiere explicar que el tiempo que tarda el algoritmo de radiosidad en tiempo real no solo depende del algoritmo en sí, sino que también depende del tiempo de renderizado de la escena, pudiendo llegar a ser este último un cuello de botella. Con un algoritmo ideal que tarde 0 ms en ejecutarse la máxima cantidad de FPS que se alcanzará será de 100 (en la escena LivingLiteColor).

Con las pruebas realizadas es posible ejecutar el algoritmo de radiosidad en tiempo real alcanzando 29 FPS en una escena con 9.696 triángulos y aplicando iluminación con color, se observan buenos resultados visuales con respecto a cómo la luz interacciona con la escena.

7 Conclusiones y trabajos futuros

7.1 Introducción

En este capítulo se describen las conclusiones generales del proyecto. Inicialmente, se evalúan los resultados alcanzados y las dificultades encontradas para llegar a ellos. En segunda instancia, se describe el contraste entre lo que se planteó hacer y lo que efectivamente se hizo indicando los aportes que se realizaron a la aplicación. Posteriormente, se realiza una autocrítica del trabajo, de lo que se hizo y lo que no. Finalmente, se listan recomendaciones para posibles trabajos futuros, ya sean mejoras o extensiones.

7.2 Conclusiones

A lo largo del proyecto se realizaron transformaciones a una aplicación existente, RadTR 1.0, que llevaron a que la misma presente mejoras en sus prestaciones anteriores, una interfaz gráfica más amigable, capacidad de conectarse con MATLAB de manera más eficiente y nuevas funcionalidades. Estas funcionalidades brindan un nuevo conjunto de herramientas a utilizar por el usuario permitiendo realizar nuevas tareas en un ambiente especializado. Esta especialización se da gracias al uso de tecnologías modernas y algoritmos más eficientes.

El plan de trabajo se realizó siguiendo el orden definido en la propuesta inicial del proyecto. El mismo permitió llegar a los resultados obtenidos ya que definía una serie de actividades necesarias para el correcto desarrollo sobre una aplicación ya existente y el uso de nuevas tecnologías. Se logró implementar los cambios acordados en el producto en el período comprendido entre Abril de 2011 y Febrero de 2012. En las primeras semanas, el esfuerzo se centró en comprender el producto existente a esa fecha y cómo estaba desarrollado. Luego de este tiempo de autoestudio, comenzó el desarrollo de los diversos ítems definidos en los requerimientos del proyecto bajo la supervisión docente. Estos objetivos eran en su mayoría independientes, y por este motivo se presentaron mayores posibilidades de dividir el trabajo entre los integrantes. La no especificación de un plan de trabajo y alcance más preciso previo al comienzo de la etapa de implementación, sumado a la definición detallada de cada requisito en el momento de iniciar su desarrollo provocaron un aumento de tiempo en dicha etapa. La falta de esta especificación se debió a que las funcionalidades a realizar estaban sujetas a cambios en la prioridad según las necesidades del usuario en el transcurso del proyecto.

Entre las dificultades encontradas está la necesidad de comprender el programa existente, y tener que adaptar el desarrollo de las funcionalidades nuevas al código heredado. Otra dificultad, fue la búsqueda de versiones compatibles entre las tecnologías utilizadas en los diferentes componentes de la aplicación, teniendo que invertir parte del tiempo en esto.

El software final es una aplicación capaz de desplegar escenas generadas por la importación de modelos (en diversos formatos), y realizar diversas operaciones y/o

transformaciones sobre las mismas. Estas transformaciones van desde la subdivisión de los triángulos que forman la escena hasta el cálculo de iluminación global mediante el algoritmo de Radiosidad (utilizando diversas tecnologías y técnicas).

El usuario puede interactuar con la aplicación mediante una interfaz gráfica, siendo ésta una opción más amigable que la interfaz anterior de la aplicación que era mediante línea de comandos.

Se agrega a RadTR 2.0 el uso de tecnologías especializadas en el área de programación para computación gráfica (lenguajes de propósito general como CUDA y utilización de *shaders*), mejorando de performance de las funcionalidades y permitiendo una interacción en tiempo real con el usuario. El uso de estas tecnologías limita el ambiente de ejecución de RadTR 2.0. Por ejemplo, para el uso de CUDA se necesita que se ejecute en una computadora con tarjeta de video NVIDIA compatible. Para utilizar las interfaces con MATLAB se debe tener instalado la versión de MCR correcta.

Si se compara lo que se definió a nivel de requerimientos con lo que se desarrolló finalmente, se puede decir que se cumplió con todos los puntos solicitados. Adicionalmente se desarrollaron varias formas de comunicación entre RadTR 2.0 y MATLAB.

La interfaz gráfica desarrollada tiene una implementación sencilla debido a que este requisito no era prioritario. Además, fue dificultoso y costoso realizar una investigación sobre las capacidades de MATLAB para integrarse a un proyecto desarrollado en lenguaje C++. Esto pasó debido al desconocimiento de los integrantes del grupo de MATLAB, y a que MATLAB es una suite que brinda muchas funcionalidades, por lo que se hizo engorroso descubrir la existencia de las funcionalidades específicas que resultaran útiles. La comunicación resultante entre MATLAB y RadTR 2.0 no es la que se planeó en el momento de empezar a desarrollar esta funcionalidad, ya que se tenía como objetivo lograr que RadTR 2.0 fuera capaz de interpretar un archivo .m sin necesidad de que se compile.

Con respecto a la ejecución de radiosidad en tiempo real se llegaron a buenos resultados. Se logró ejecutar el algoritmo lo suficientemente rápido dando la sensación de tiempo real con escenas de tamaño mediano (alrededor de 10.000 triángulos) y visualmente correcto.

7.3 Trabajos futuros

Se presenta la siguiente lista de mejoras o extensiones al producto:

- Mejora de la interfaz gráfica: agregar imágenes para una mejor comprensión visual de las funcionalidades; gráficas de progreso cuando se ejecuta una tarea; mejor disposición de los elementos gráficos.
- Extender las plataformas en las que se puede ejecutar el programa, utilizando tarjetas de video de otras compañías y utilizando otras librerías (OpenCL).
- Capacidad de tener dos *displays* concurrentes para comparar imágenes de algoritmos en tiempo real.

- Posibilidad de interpretar código MATLAB directamente desde la aplicación mediante archivos con formato .m.
- Posibilidad de leer y guardar archivos de comandos para la ejecución ordenada de distintas funcionalidades de RadTR.
- Agregar la posibilidad de calcular radiosidad en tiempo real con escenas dinámicas (objetos de la escena cambian de posición y orientación en cada *frame*).
- Crear una API para que otros programas tengan la posibilidad de utilizar las funcionalidades del software.
- Incluir otros algoritmos de Iluminación global como pueden ser *Ray Tracing* y *Photon Mapping*.

Glosario

A continuación se listan las definiciones de la terminología utilizada en este trabajo.

API: Del inglés *Application Program Interface*, es el conjunto de funciones y procedimiento que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

Armadillo: Librería de algebra lineal escrita en C++.

Assimp: *Open **Ass**et **Imp**ort Library*, es una librería portable de código abierto programada en C++ que sirve para importar varios formatos de modelos 3D de manera uniforme.

Beam Tracing: Algoritmo de iluminación global utilizado para simular la propagación de ondas. Es un derivado de *Ray Tracing* remplaza rayos por haces.

BLAS: Del inglés, *Basic Linear Algebra Subprograms*, es la API de facto para publicar librerías que realizan operaciones básicas del álgebra lineal tales como la suma y multiplicación de matrices.

Cg: Del inglés, *C for graphics*, es un lenguaje propietario de programación de *shaders* creado por NVIDIA.

Cornell Box: Es una prueba destinada a determinar la calidad del renderizado de software comparando la escena renderizada con una fotografía actual de la misma.

CUBLAS: Librería matemática distribuida por NVIDIA para realizar cálculos matemáticos en la GPU utilizando CUDA. El fin de esta librería es brindar un sustituto más rápido que la librería BLAS.

CUDA: Es un lenguaje de programación de NVIDIA, utilizable en los modelos más nuevos de tarjetas gráficas de esta empresa, para propósito general. A través de ella, se puede utilizar la capacidad de cómputo de las tarjetas gráficas para el desarrollo de algoritmos de propósito general, no necesariamente ligados a la renderización (actividad básica de las tarjetas gráficas).

Factor de forma: Fracción de energía que abandona a una superficie y que alcanza a una segunda superficie.

FORTRAN: Lenguaje de programación de propósito general, procedimental e imperativo que está especialmente adaptado al cálculo numérico y la computación científica.

FPS: Imágenes por segundo. Tasa que se utiliza para medir la frecuencia de renderizado. Indica la velocidad en que una aplicación gráfica despliega imágenes consecutivas.

Fragment Shader: También conocido como *Pixel Shader* computa el color y otros atributos de cada píxel.

Geometry Shader: *Shader* que puede generar nuevas primitivas gráficas como puntos líneas y triángulos.

GLSL: Acrónimo del inglés, *OpenGL Shading Language*, es un lenguaje de programación estándar de *shaders* de alto nivel asociado a la biblioteca OpenGL.

GPU: Unidad de Procesamiento, por sus siglas en inglés, *Graphics Processing Unit*. Es un procesador especializado con funciones relativamente avanzadas de procesamiento de imágenes, en especial para gráficos 3D.

Hemicubo: Algoritmo que, centrado en un parche cualquiera, calcula los factores de forma con todos los parches de la escena. Se basa en realizar cinco proyecciones de los parches de la escena en las cinco superficies de un medio cubo. Al final del proceso, cada píxel del hemicubo contiene la información de los parches que son visibles desde el parche inicial.

HLSL: Del inglés, *High Level Shading Language*, es un lenguaje de programación de shaders desarrollado por Microsoft en colaboración con NVIDIA prácticamente idéntico a Cg.

LivingLiteColor: Es una escena utilizada en este trabajo para realizar pruebas de calidad del renderizado del software.

MATLAB: (acrónimo de *MATrix LABoratory*) es un software matemático que ofrece un IDE (entorno de desarrollo integrado) con un lenguaje de programación propio cuyo nombre es M.

MCR: (del inglés, *MATLAB Compiler Runtime)*, un conjunto independiente de librerías que habilita la ejecución de archives MATLAB en computadoras sin una versión instalada de MATLAB.

OpenCL: Acrónimo de ***Open C****omputing* ***L****anguage*. Es un *framework* que posibilita desarrollar programas paralelos que ejecuten en plataformas heterogéneas consistentes en CPUs, GPUs y otros procesadores.

OpenGL: es un *framework* para la programación en tarjetas gráficas, con propósito de generación de escenas. Es una librería utilizada en diversos modelos de tarjetas gráficas.

Paraboloide: Algoritmo con objetivo similar al Hemicubo, salvo que se realiza sobre una hemiesfera.

Photon Mapping: Es un método de iluminación global de dos pasadas. La primera pasada construye el mapa de fotones, emitiendo fotones desde la fuente de luz hacia la escena y almacenándolos en el mapa de fotones cuando estos no golpean en objetos especulares. La segunda pasada utiliza técnicas estadísticas sobre el mapa de fotones para extraer información acerca del flujo incidente y la radiancia reflejada en cualquier punto de la escena.

Pipeline: Es una técnica para mejorar la performance de un proceso divisible en un conjunto de etapas que se repite a lo largo del tiempo. La entrada de cada etapa es la salida de la anterior. No se puede ejecutar una etapa para una nueva tarea hasta que se haya completado esa etapa para la tarea anterior.

Radiosidad: Es una técnica de iluminación global para escenas con superficies difusas. El observador es independiente de los cálculos de la iluminación. También se define como la potencia radiada por unidad de área (W/m^2).

RadTR: Nombre de la aplicación desarrollada.

RadTR 1.0: Versión previa de la aplicación antes de comenzar el desarrollo del proyecto.

RadTR 2.0: Versión final de la aplicación desarrollada en el proyecto.

RTR: Del inglés *Real-Time Rendering*, Renderizado en Tiempo Real.

Vertex Shader: Es un *shader* que opera para cada vértice y su función es modificar la posición, color y coordenadas de textura.

Ray Tracing: Es una técnica de iluminación global en la cual se trazan rayos de luz desde un punto de vista virtual y existe un plano entre dicho punto de vista y la escena que es donde se reconstruye la imagen renderizada.

Shader: Es un programa que se ejecuta en la GPU utilizado principalmente para calcular efectos de renderizado con un alto grado de flexibilidad (mayor al que aportan las APIs como OpenGL y DirectX).

Tessellation Shader: Es una nueva característica de OpenGL 4.x que permite la generación de geometrías con mayor nivel de detalle al subdividir las primitivas (triángulos).

Referencias

[1] A. S. Glassner, An Introduction to Ray Tracing, Morgan Kaufmann. ISBN 0122861604, 1989.

[2] H. W. Jensen, F. Suykens, P. H. Christensen y T. Kato, A Practical Guide to Global Illumination using Photon Mapping, Siggraph, 2000.

[3] M. Cohen, J. Wallace y P. Hanrahan, Radiosity and realistic image synthesis, San Diego, CA, USA: Academic Press Professional, Inc., 1993.

[4] «NVIDIA,» [En línea]. Available: http://www.nvidia.es/. [Último acceso: 15 agosto 2012].

[5] T. Akenine-Möller, E. Haines y N. Hoffman, Real-Time Rendering (3rd Eddition), Natick: A K Peters, Ltd., 2008.

[6] Foley, V. Dam, Feiner, Hughes y Phillips, Introducción a la Graficación por Computador, Adisson-Wesley Iberoamericana, S.A., 1996.

[7] M. U. Martínez, Procesadores gráficos para PC, Editorial ciencia-3, S.L., 2005.

[8] D. Owens, M. Houston, D. Luevke, S. Green y J. E. Stone, GPU Computing, Proceedings of the IEEE, 2008.

[9] T. J. Purcell, Ray tracing on a stream processor, PhD Thesis, Stanford University, 2004.

[10] «General-Purpose Computation on Graphics Hardware,» [En línea]. Available: http://gpgpu.org/. [Último acceso: 15 agosto 2012].

[11] «OpenGL - The Industry's Foundation for High Performance Graphic,» [En línea]. Available: www.opengl.org. [Último acceso: 23 07 2012].

[12] «DirectX Developer Center,» [En línea]. Available: http://msdn.microsoft.com/en-us/directx/aa937781.aspx. [Último acceso: 15 agosto 2012].

[13] «Microsoft Corporation,» [En línea]. Available: http://www.microsoft.com/. [Último acceso: 15 agosto 2012].

[14] «HLSL,» [En línea]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561%28v=vs.85%29.aspx. [Último acceso: 15 agosto 2012].

[15] «Parallel Programming and Computing Platform | CUDA | NVIDIA,» [En línea]. Available: http://www.nvidia.com/object/cuda_home_new.html. [Último acceso: 20 agosto 2012].

[16] «BeHardware - Nvidia CUDA: Preview,» [En línea]. Available: http://www.behardware.com/articles/659-4/nvidia-cuda-preview.html. [Último acceso:

20 agosto 2012].

[17] «About the OpenGL Architecture Review Board Working Group,» [En línea]. Available: http://www.opengl.org/archives/about/arb/. [Último acceso: 15 agosto 2012].

[18] «MATLAB - The Language of Technical Computing,» [En línea]. Available: www.mathworks.com/products/matlab. [Último acceso: 23 julio 2012].

[19] «MATLAB Compiler - Product Overview,» [En línea]. Available: http://www.mathworks.com/help/toolbox/compiler/br5w5e9-2.html. [Último acceso: 23 julio 2012].

[20] «MATLAB Coder,» [En línea]. Available: http://www.mathworks.com/products/matlab-coder/. [Último acceso: 23 julio 2012].

[21] E. Fernández, P. Ezzatti, S. Nesmachnow y G. Besuievsky, Low-rank radiosity using sparse matrices, In Proceding of GRAPP, pages 260-267, 2012.

[22] «Global Illumination Using Progressive Refinement Radiosity, Paraboloide,» [En línea]. Available: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter39.html. [Último acceso: 23 julio 2012].

[23] « Dual Paraboloid Mapping In the Vertex Shader, Jason Zink,» [En línea]. Available: http://members.gamedev.net/JasonZ/Paraboloid/DualParaboloidMappingInTheVertexShader.pdf. [Último acceso: 23 julio 2012].

[24] «Dynamische Environment Maps, Christoph Dietze,» [En línea]. Available: http://www.joachim-dietze.de/pdf/dynenvmaps.pdf. [Último acceso: 23 julio 2012].

[25] «MATLAB - System Requirements and Supported Compilers,» [En línea]. Available: http://www.mathworks.com/support/sysreq/previous_releases.html. [Último acceso: 23 julio 2012].

[26] «MATLAB Compiler - Working with the MCR,» [En línea]. Available: http://www.mathworks.com/help/toolbox/compiler/f12-999353.html. [Último acceso: 23 julio 2012].

[27] «The MATLAB array,» [En línea]. Available: http://www.mathworks.com/help/techdoc/matlab_external/f21585.html#f17318. [Último acceso: 23 julio 2012].

[28] «MATLAB Compiler - mcc,» [En línea]. Available: http://www.mathworks.com/help/toolbox/compiler/mcc.html - (Último acceso: 23/07/12). [Último acceso: 23 julio 2012].

[29] «Call MATLAB Compiler API Functions (mcl*) from C/C++ Code,» [En línea]. Available: http://www.mathworks.com/help/toolbox/compiler/f2-998954.html. [Último acceso: 23 julio 2012].

[30] Mathworks, *Ayuda de MATLAB (2011). Versión 7.12.0.635 (R2011a),* 2011.

[31] «MATLAB Compiler - C Code Interface for Arrays,» [En línea]. Available: http://www.mathworks.com/help/toolbox/eml/ug/br_jxhp.html. [Último acceso: 23 julio 2012].

[32] «MATLAB Compiler - Call MATLAB Compiler API Functions (mcl*) from C/C++ Code,» [En línea]. Available: http://www.mathworks.com/help/toolbox/compiler/f2-998954.html. [Último acceso: 23 julio 12].

ANEXOS

ANEXOS

ANEXO A – Preparación del ambiente de desarrollo.

ANEXO B – Compilador MATLAB.

ANEXO C – MATLAB *Coder* – Interfaz para *arrays* dinámicos.

ANEXO D - Estructuras de datos principales.

ANEXO E - Funciones de escena.

ANEXO F - Radiosidad en tiempo real, código en CUDA y C.

ANEXO G - Configuración de RadTR 2.0.

ANEXO H - Jerarquía de superficies.

ANEXO I – Hemicubo.

ANEXO J - *Shaders* de división de triángulos en paraboloide.

Anexo K – Manual de usuario.

Anexo L – Ejemplos de utilización de MATLAB con RadTR 2.0.

ANEXO A – Preparación del ambiente de desarrollo

Para instalar CUDA es necesario seguir los pasos siguientes:

- Tener Microsoft Visual Studio 2010
- Instalar CUDA Toolkit en "C:\CUDA\" ya que el proyecto está configurado en esa ruta.
- Luego aplicar parche "BuildCustomizationFix" para que visual studio 2010 reconozca código CUDA, se reemplazan los ficheros "CUDA 4.0" y "CUDA 4.0.targets" en "C:\Program Files\MSBuild\Microsoft.Cpp\v4.0\BuildCustomizations".
- Instalar el driver de CUDA.

Aspectos a considerar cuando se instale MATLAB y Visual Studio:

- Antes de instalar alguna de las dos aplicaciones conviene conocer la compatibilidad de la versión a instalar de MATLAB con la versión del compilador MS Visual Studio que se dispone o se instalará. Para esto se deberá ingresar al sitio en línea de MATLAB [25] y chequear especialmente si la columna "MATLAB *Compiler*" está disponible para la versión del compilador que se dispone.
- Cuando se instala MATLAB hay que asegurarse de instalar los componentes que se utilizarán, principalmente el compilador MATLAB. Por lo tanto, es recomendable realizar una instalación personalizada para tener mayor seguridad.
- Si se trabaja en un sistema operativo es de 64 bits, se debe forzar la instalación de MATLAB en 32 bits para que éste pueda generar las librerías en 32 bits. De esta forma se podrá incluirlas en RadTR 2.0 que es de 32 bits.

ANEXO B – Compilador MATLAB

En este anexo se describen algunas características del compilador MATLAB. Se comenzará con la configuración del compilador, para luego sí entrar en los aspectos propios del mismo.

Dado que explicar en profundidad el compilador MATLAB podría llevar un extenso informe, se decidió brindar esta breve guía práctica que contiene lo que interesa del mismo para el presente proyecto. En caso de necesitar más información se podrá recurrir a las referencias citadas.

Configuración del compilador

A continuación se detallan los pasos para configurar por primera vez o cambiar el compilador C/C++ por defecto que se utilizará con el compilador MATLAB. Se realiza lo siguiente:

- Abrir la aplicación MATLAB.
- En la ventana de comandos ejecutar "*mbuild –setup*". Allí se desplegará un mensaje de bienvenida y un *link* en el que se muestran los compiladores compatibles con la versión de MATLAB. Esto es muy importante, y debería ser tenido en cuenta antes de instalar Visual Studio y/o MATLAB. Por ejemplo, en este caso se utilizó la R2011a de MATLAB y Microsoft Visual Studio 2010.

```
>> mbuild -setup


Welcome to mbuild -setup.  This utility will help you set up
a default compiler.  For a list of supported compilers, see
http://www.mathworks.com/support/compilers/R2011a/win64.html
```

- Luego se debe seleccionar un compilador de C/C++ para asociarlo al compilador MATLAB. Por ello se plantea la pregunta de si se quiere que mbuild localice automáticamente los compiladores instalados en el sistema. Se selecciona que sí:

```
Please choose your compiler for building standalone MATLAB
applications:


Would you like mbuild to locate installed compilers [y]/n? y
```

- A este punto se indica que sí se desea que *mbuild* busque los compiladores instalados en el sistema y se despliega la siguiente lista:

```
Select a compiler:
```

```
      [1] Microsoft Visual C++ 2010 in c:\Program Files (x86)\Microsoft
Visual Studio 10.0


      [0] None
```

- Como tan solo se tiene instalado el compilador de Visual Studio 2010 se selecciona el mismo indicando la opción 1.

```
Compiler: 1
```

- Luego, se pide confirmación de la configuración y, si toda la información es correcta, se seleccinoa que sí:

```
      Please verify your choices:


      Compiler: Microsoft Visual C++ 2010

      Location: c:\Program Files (x86)\Microsoft Visual Studio 10.0


      Are these correct [y]/n? y


      ************************************************************************
*********

        Warning: Applications/components generated using Microsoft Visual
C++

                 2010 require that the Microsoft Visual Studio 2010 run-
time

                 libraries be available on the computer used for
deployment.

                 To redistribute your applications/components, be sure
that the

                 deployment machine has these run-time libraries.

      ************************************************************************
*********


      Trying to update options file:
C:\Users\Diego\AppData\Roaming\MathWorks\MATLAB\R2011a\compopts.bat

      From template:
C:\PROGRA~1\MATLAB\R2011a\bin\win64\mbuildopts\msvc100compp.bat


      Done . . .
```

- Luego de esta configuración, el compilador MATLAB podrá generar archivos compatibles con el compilador C/C++ externo.

Para compilar para 32 bits

Si se tiene instalado un sistema operativo de 64 bits y se quiere generar librerías para 32 bits se deberá ejecutar MATLAB en su versión de 32 bits (previamente instalada). En el momento de generar se deberá añadir la flag *–win32* sólo en los siguientes casos:

- Se está utilizando la misma *installation root* (*install_root*) para la versión de 32 bits como la de 64 bits.
- Se está trabajando en la línea de comandos de Windows y no en la ventana de comandos de MATLAB.

MCR

MCR (MATLAB *Compiler Runtime*) es un conjunto independiente de bibliotecas compartidas que permite la ejecución de archivos de MATLAB en equipos sin una versión instalada de MATLAB.

Si al distribuir archivos compilados a los usuarios finales que no tienen MATLAB en sus sistemas, estos deberán instalar MCR. Sólo se tiene que instalar el MCR una vez en sus equipos.

Diferencias entre el MCR y MATLAB

MCR difiere de MATLAB principalmente en los siguientes aspectos:

- Con MCR, los archivos MATLAB son encriptados para proporcionar portabilidad e integridad.
- MATLAB tiene una interfaz gráfica. MCR tiene todas las funcionalidades de MATLAB sin dicha interfaz gráfica.
- MCR depende de la versión. Las aplicaciones creadas se deben correr en la versión de MCR asociada a la versión del compilador de MATLAB con la cual se creó.

Hay que tener en cuenta que la inicialización de las librerías de MCR puede tomar un tiempo no menor, que se puede considerar similar al tiempo en que demora en iniciarse la aplicación MATLAB. Esto se debe a la cantidad de recursos consumidos por MCR, que son necesarios para disponer del poder y la funcionalidad de una versión completa de MATLAB.

En el manual de MATLAB [26] se puede encontrar una guía para instalar MCR. Más adelante en este anexo se muestra una forma sencilla de obtener el instalador del MCR a partir de una versión instalada de MATLAB.

mxArray y mwArray

La matriz MATLAB (o su definición original en inglés, *MATLAB array*) es el único objeto con el cual trabaja el lenguaje MATLAB. Al crear cualquier variable en MATLAB, la misma es almacenada como una matriz MATLAB. Estas matrices son declaradas en C/C++ como del tipo *mxArray*, que es una estructura que contiene la siguiente información acerca de la matriz original [27]:

- El tipo
- Las dimensiones
- Los datos asociados a dicha matriz
- Si es numérica, indica si es real o compleja
- Si es dispersa, sus índices y máxima cantidad de elementos mayores que cero
- Si es un objeto o estructura, el número de campos y sus nombres

El *mxArray* es la representación en código C de un arreglo MATLAB. Para acceder a dicha estructura se utiliza las funciones de la API de *MX Matrix Library.* Para obtener más información acerca de *mxArray* se recomienda visitar el manual de MATLAB [27].

La clase *mwArray* se utiliza para pasar argumentos a través de las funciones de la interfaces generadas por el compilador MATLAB para C++. *mwArray* provee los constructures, métodos y operadores necesarios para la creación e inicialización de matrices. Además, es la clase a utilizar cuando se generen librerías en C++.

Generando librerías en C

A continuación se describe someramente cómo se genera una librería para C. Se supone que se dispone de un archivo MATLAB llamado '*funcion.m'*, del cual se quiere obtener una librería compartida.

En la línea de comandos de MATLAB se deberá ingresar la siguiente línea:

```
mcc -W lib:nombreLibreria -T link:lib funcion.m
```

El significado exacto de cada una de las flags utilizadas se puede obtener de la ayuda del commando *mcc* ingresando en la línea de comandos "mcc -?" o en el manual en línea [28].

En particular, existe una macro para la línea anterior, y quedaría de forma más corta:

```
mcc –l funcion.m
```

Por cada archivo MATLAB compilado, el compilador genera dos funciones de entrada, cuyos nombres comienzan con los prefijos *mlx* y *mlf*. Cada una de estas funciones generadas realiza la misma acción, que no es otra que invocar la función creada en MATLAB. La diferencia entre estas dos funciones es la interfaz de las mismas. Por ejemplo, si la función MATLAB se

llamaba 'funcion.m' las funciones generadas se llamarán *mlfFuncion* y *mlxFuncion* respectivamente [29].

Ambas funciones difieren en la forma en que se le pasan los argumentos. Para la función *mlx* se tiene la siguiente forma:

void mlxFuncion(int nlhs, mxArray *plhs[], int nrhs, mxArray *prhs[])

nlhs: Cantidad de argumentos de salida.

plhs: Vector con los argumentos de salida.

nrhs: Cantidad de argumentos de entrada.

nrhs: Vector con los argumentos de entrada.

Para la función *mlf* se puede tener diferentes interfaces según la cantidad de valores de retorno:

Si la función no tiene valores de retorno:

void mlfFuncion(<lista_de_variables_de_entrada>)

Si tiene al menos un valor de retorno:

void mflFuncion(int cantidad_var_retorno,
<lista_de_punteros_a_var_de_retorno>,
<lista_de_variables_de_entrada>)

Los variables que se pasan como argumentos a las funciones son de tipo mxArray.

Generando librerías en C++

Al igual que en el caso de la generación para C, aquí se considera el archivo 'funcion.m" a partir del cual se generará la librería compartida.

```
mcc -W cpplib:funcion -T link:lib funcion.m
```

Al generar para C++ se crea la interfaz *mlx* al igual que en C, pero se agrega una más que corresponde sólo a C++. La misma, al igual que la función *mlf* del apartado anterior, presenta dos formas según la cantidad de argumentos de salida.

Si la función no tiene valores de retorno:

```
void funcion(<list_de_variables_de_entrada>)
```

Si la función tiene variables de retorno:

```
void funcion(    int <numero_de_variables_de_salida>,
                 <lista_de_variables_de_salida>,
                 <lista_de_variables_de_entrada>);
```

Esta función varía con respecto a la generada en C en que el tipo de los datos de entrada el mwArray y no mxArray. Más adelante se dará una descripción general de estos tipos de datos.

Archivos creados al generar las librerías

Aquí se brinda una breve descripción de la función de cada uno de los archivos que se crean al generar librerías compartidas con el compilador MATLAB. Se utiliza como base la generación en C, siendo análogas las descripciones para la generación en C++.

Se supondrá que se generan librerías para el código que se encuentra en el archivo *funcion.m*, y que la librería tendrá el nombre *'funcion'*. También se asume que se está trabajando en un ambiente Windows. En este contexto, se crearán los siguientes archivos:

funcion.lib: Librería estática necesaria para crear la aplicación que invocará la librería dinámica.

funcion.dll: Archivo binario correspondiente a la librería compartida. Como se está trabajando en un ambiente Windows su extensión es *.dll*.

funcion.h: Es el archivo de cabecera de la librería. Debe ser incluido en las aplicaciones que invoquen a *funcion*.

funcion.c: Contiene las funciones exportadas de la librería y la representación en C de la interfaz de la correspondiente funcion.m. También contiene código de inicialización de la librería.

funcion.exports: Función de exportación que utiliza *mbuild* para linkeditar la librería.

readme.txt: Muestra información de la generación, así como también una breve guía de distribución.

Antes de comenzar a desarrollar se debe verificar que se encuentre instalado el MCR y que su versión es la misma en la cual se generó la librería. Si se está trabajando en la computadora en la cual se generó, el instalador para dicha versión se puede encontrar generalmente en:

<matlabroot>\toolbox\compiler\deploy\win<[32|64]>\MCRInstaller.exe

En dicha dirección se elige 32 o 64 dependiendo de si la versión instalada es de 32 o 64 bits respectivamente. En *<matlabroot>* se debe especificar el directorio de instalación de la correspondiente versión de MATLAB.

Estructura de un programa que utiliza librerías compartidas

Cualquier programa que utilice librerías compartidas generadas por el compilador MATLAB debe tener la siguiente estructura [29]:

1) Declarar variables y procesar los argumentos de entrada.
2) Invocar *mclInitializeApplication* y chequear si hubo error. Esta función configura el estado global del MCR y habilita la construcción de instancias MCR.
3) Para cada librería compilada, invocar a <NombreLibrería>Initialize, para crear la instancia MCR requerida por la librería.
4) Invocar las funciones en la librería y procesar los resultados.
5) Para cada libería, llamar a <NombreLibrería>Terminate, para destruir la instancia asociada al MCR.
6) Invocar la función mclTerminateApplication para liberar los recursos asociados con el estado global del MCR.
7) Continuar con la ejecución del programa. Si no se realizan más acciones se debe liberar la memoria para las variables inicializadas en (1).

Es importante destacar la necesidad de la inicialización y terminación de las librerías que se realizan en los puntos 3 y 6 respectivamente. Toda llamada a las funciones de la librería (las que invocan a la función MATLAB creada) se debe realizar entre una inicialización y una terminación. La invocación en el punto 5 también es muy importante porque libera la memoria utilizada en la ejecución de la librería.

Para más información acerca de las funciones de inicialización y terminación se recomienda visitar el manual en línea de MATLAB [29].

Archivos a distribuir a un desarrollador

En el archivo *readme.*txt se indica que en caso que exista la intención de distribuir el código fuente de una aplicación que utilice una librería generada por MATLAB, se debe empaquetar conjuntamente con esta los siguientes archivos:

- funcion.dll
- funcion.h
- funcion.lib
- MCRInstaller.exe (la versión correspondiente según lo expresado en el apartado anterior)
- El archivo readme.txt

Interfaz de datos de usuario del MCR

MCR proporciona una interfaz de datos de usuario desde la cual se puede acceder fácilmente a los datos en una instancia de MCR. Ésta permite definir claves y valores asociados a las mismas que pueden ser pasadas entre la instancia MCR, el código MATLAB que corre en el MCR y el código contenedor que creó el MCR.

La interfaz de datos de usuario del MCR consiste de las siguientes funciones:

- Dos funciones MATLAB que pueden ser invocadas desde un código escrito en MATLAB
- Dos funciones C que pueden ser invocadas desde la aplicación contenedora.

Las dos funciones MATLAB [30]

SETMCRUSERDATA(KEY, VALUE): Asocia el dato VALUE de MATLAB con la clave KEY en la instancia de MCR actual. Si ya existe un valor asociado con KEY, lo sobrescribe. La clave debe ser una cadena de caracteres, mientras que el valor a almacenar puede tener cualquier tipo soportado por MATLAB. Esta función realiza una copia del valor antes de guardarlo, por lo que si se modifica dicho valor luego en el programa, no se modificará el valor almacenado.

Ejemplo:

```
SETMCRUSERDATA('pares', [2,4,6,8])
```

GETMCRUSERDATA(KEY): Devuelve el dato de MATLAB asociado con la clave KEY en la instancia MCR actual. Si no hay datos asociados devuelve una matriz vacía. Al igual que en el caso anterior la clave debe ser una cadena de caracteres.

Ejemplo:

```
result = GETMCRUSERDATA('pares')
```

```
result =

  2     4     6     8
```

Las dos funciones en C

- bool **mclSetMCRUserData**(long mcrID, const char *key, mxArray *value)

Su funcionalidad es similar a la que se describió anteriormente para el caso de MATLAB. La única diferencia es que recibe además el identificador de la instancia de MCR para la librería compartida. Dicho valor se puede obtener a través de una función que proporciona la interfaz generada por el compilador MATLAB. Devuelve 0 en caso que no haya error.

Ejemplo:

```
mxArray *value = mxCreateString("Uruguay");
if (!mclSetMCRUserData(funcionGetMcrID(),"pais", value)) {
  fprintf(   stderr,
                        "Could not set MCR user data: /n %s",
                        mclGetLastErrorMessage());
  return -3;
}
```

- mxArray* **mclGetMCRUserData**(long mcrID, const char *key)

Tiene una funcionalidad análoga a la función GETMCRUSERDATA de MATLAB. Al igual que a la función mclSetMCRUserData se le debe pasar el identificador de la instancia MCR.

Ejemplo:

```
mxArray *valueget = mclGetMCRUserData(funcionGetMcrID(), "pais");
printf("%s", mxArrayToString(valueget));
```

En ejecución devolvería: "Uruguay"

ANEXO C – MATLAB *Coder* – Interfaz para arrays dinámicos

Se considera muy importante conocer la estructura en que se representan los *arrays* dinámicos generados por MATLAB *Coder*. Este material está basado en la especificación que se encuentra en el manual de MATLAB [31].

En el código generado, MATLAB representa los *arrays* dinámicos como un tipo estructurado llamado *emxArray*. emxArray es una familia de tipos de datos, especializados para cada tipo base. Puede considerarse con una versión embebida del *mxArray* utilizado por MATLAB.

Genéricamente, se puede definir este tipo estructurado como:

```
typedef struct emxArray_<baseTypeName>
{
   <baseTypeName> *data;
   int32_T *size;
   int32_T allocated;
   int32_T numDimensions;
   boolean_T canFreeData;
} emxArray_<baseTypeName>;
```

Por ejemplo, si se quiere aplicar este tipo a los reales quedaría:

```
typedef struct emxArray_real_T
{
   real_T *data;
   int32_T *size;
   int32_T allocated;
   int32_T numDimensions;
   boolean_T canFreeData;
} emxArray_real_T;
```

Descripción de los campos de la estructura:

- **data*: Puntero a los datos de tipo <baseTypeName>.
- **size*: Puntero al primer elemento del vector de dimensiones. El largo de este vector es igual al número de dimensiones.
- *allocatedSize*: Número de elementos actualmente en el arreglo de datos.
- *numDimensions*: Número de dimensiones del vector *size*.
- *canFreeData*: Booleano que indica cómo se libera la memoria. Si es *true* MATLAB libera la memoria automáticamente, si es *false* el programa invocador determina cuándo se libera la memoria.

ANEXO D - Estructuras de datos principales

Estas estructuras de datos definen a una escena, se encuentran en el fichero *scene.h*, a continuación se describen las estructuras que se encontraban con el software base.

```
struct scene {

 //Materiales.
 std::vector<material> materials;

 //Vertices.
 std::vector<vec3f> vertices;

 #ifdef NORMAL_POR_VERTICE
        //Normales.
        std::vector<vec3f> normales;
 #endif

 //Superficies.
 std::vector<surface> surfaces;

 // El primer vector referencia a indice de vertice,
 // en el segundo guarda indice de superficie.
 // Tenemos un vector por cada vertice, en cada uno de esos vectores
 // tenemos las superficies que lo usa.
 std::vector<std::vector<int> > vertex_surfaces;

 // Colores de los vertices, se usa para dibujar triangulos
 // con color interpolados.
 std::vector<vec3f> vertex_color;

 //Cantidad de nodos en la jerarquía.
 unsigned int fathers;
};

struct material {
 //Este es el color que se usara.
 vec3f color;
 vec3f emission;
};


struct surface {
 //Indice a vertices.
 std::vector<unsigned int> vertices;

 //Indice a material
 unsigned int material;

 //La normal de la superficie.
 vec3f normal;

 //Nodo padre.
 unsigned int father;

 //Indice de nodo de jerarquia, es el numero de nodo al que pertenece.
```

```
  unsigned int nodeinhierarchy;
};
```

A continuación se detallan estructuras nuevas y cambios a las anteriores en la versión final de RadTR.

```
struct scene {

  //Coordenadas de texturas para cada vertice.
  std::vector<coordTex> coordTexVertices;

  ...
};


struct coordTex{
  bool existe;
  vec2f coord;
};


struct material {
  //Estos colores se guardan para futuros usos.
  vec3f colorAmb;
  vec3f colorDiff;
  vec3f colorSpec;

  //Este es el color que se usara.
  vec3f color;
  vec3f emission;

  //Texturas.
  //El primer vector hace referencia al tipo de textura:
  //0 - aiTextureType_NONE
  //1 - aiTextureType_DIFFUSE
  //2 - aiTextureType_SPECULAR
  //3 - aiTextureType_AMBIENT
  //4 - aiTextureType_EMISSIVE
  //5 - aiTextureType_HEIGHT
  //6 - aiTextureType_NORMALS
  //7 - aiTextureType_SHININESS
  //8 - aiTextureType_OPACITY
  //9 - aiTextureType_DISPLACEMENT
  //10- aiTextureType_LIGHTMAP
  //11-aiTextureType_REFLECTION
  //12- aiTextureType_UNKNOWN
  //El segundo vector tiene una lista de identificadores OpenGL de textura.
  std::vector<std::vector<int> > texturas;
  std::vector<std::vector<char*> > nombreTexturas;
};
```

ANEXO E - Funciones de escena

Estas funciones se encontraban implementadas desde el software base en *scene.cpp*.

```
void calc_vertex_surfaces(scene &s);
```
Calcula para cada vértice las superficies que los contiene.

```
void interpolate_vertex_colors(scene& s);
```
Interpola los colores de las superficies asignando colores a los vértices, de esta forma se dibuja la escena con color interpolado.

```
float area(scene& e, int surf);
```
Retorna el área de una superficie.

```
vec3f normal(scene& e, int surf);
```
Calcula la normal de una superficie.

```
void set_emission(scene &e, const std::vector<vec3f>& surface_emission);
```
Asigna emisión de luz a una escena, en el vector *surface_emission* indica para cada superficie el color que debe poseer, esta función es usada luego de calcular radiosidad por ejemplo.

```
void subdivide(scene& e, hierarchy& hier, float max_area, int level, bool
calcularVertexSurface, bool ordenarPorNodoPadre);
```
Esta función es muy utilizada, divide los triángulos de la escena recursivamente *level* veces como mínimo y continua dividiéndolos hasta llegar un máximo de área de *max_area*. Divide los triángulos en 4 donde cada uno se calcula dividiendo a la mitad cada lado del original, de esta forma siempre existe un triángulo que se encuentra en el centro donde será nodo padre cuando es la primer división (*level* es 0) de la jerarquía. Se utiliza el booleano *calcularVertexSurface* cuando no se desea subdividir más y se desea calcular el array *vertex_surface* de la escena, el booleano *ordenarPorNodoPadre* cuando esta activo ordena las superficies de menor a mayor según el nodo padre que pertenecen (*father*), esto es útil en radiosidad en tiempo real cuando se comprime la matriz V.

Cuando se realiza una subdivisión desde la interfaz gráfica de usuario en realidad se invoca tantas veces la función subdivide como niveles se desea más uno. En el fichero *main.cpp* se encuentra la función subdivide encargada de realizar todas las invocaciones a subdivide de *scene.h* y en la última calcula *vertex_surface* y ordena las superficies por *father*, por último realiza una interpolación de color de los vértices (función *interpolate_vertex_colors*).

ANEXO F - Radiosidad en tiempo real, código en CUDA y C

Código en C para CPU para resolver B=E-Y*V'*E:

```
// B=V'*E
for(int i=0; i<largo; i++){

	int beg = Iarray[i];

	int k = Iarray[i+1] - beg;

	B[i] = 0;
	B[i+largo] = 0;
	B[i+2*largo] = 0;
	for (int j=0; j < k; j++) {
		B[i] += E[beg+j] * Ivalores[beg+j];
		B[i + largo] += E[(beg+j) + alto] * Ivalores[beg+j];
		B[i + 2*largo] += E[(beg+j) + 2*alto] * Ivalores[beg+j];
	}
}

/// B=E-Y*B
unsigned int largoM, altoM;
//M=Y*B
mat->multiplicar_1_3(Yarray, B, largo, alto, 1, largo, largoM, altoM, M);
//B=E-M
mat->resta3(E, M, 1, alto, B);
```

Código en CUDA para GPU para resolver B=V'*E:

```
__global__ void cublasSpMV3Color(int k, float* devPtrB_R, float* devPtrB_G,
float* devPtrB_B, float *E, int *Iarray, float* Ivalores, unsigned int m, un-
signed int n) {

	int idx = blockIdx.x * blockDim.x + threadIdx.x;

	int beg = Iarray[idx];

	if (k == -1)
		k = Iarray[idx + 1] - beg;

	if(idx < m){
		devPtrB_R[idx] = 0;
		devPtrB_G[idx] = 0;
		devPtrB_B[idx] = 0;
		for (int i=0; i < k; i++) {
			devPtrB_R[idx] += E[(beg+i)] * Ivalores[beg+i];
			devPtrB_G[idx] += E[(beg+i) + n] * Ivalores[beg+i];
			devPtrB_B[idx] += E[(beg+i) + 2*n] * Ivalores[beg+i];
		}
	}
}
```

Función que se encarga de lanzar el kernel CUDA:

```
int cublasSpMV4Color(int n, int m, int k,  int* Iarray, float *E, float*
devPtrB_R, float* devPtrB_G, float* devPtrB_B, float* Ivalores, int cantHi-
losxBloque) {

	// Ejecuta bloques de cantHilosxBloque hilos, en caso que no sea multiplo
se lanza un bloque mas, esto causa
	// que los hilos del ultimo bloque terminen sin hacer nada.
	int tamGrid = m/cantHilosxBloque;
	if( (tamGrid * cantHilosxBloque) < m )
		tamGrid ++;

	dim3  grid(tamGrid, 1, 1);
	dim3  threads(cantHilosxBloque, 1, 1);

	cublasSpMV3Color<<<grid,threads>>>(k, devPtrB_R, devPtrB_G, devPtrB_B, E,
Iarray, Ivalores, m, n);

	return 0;
}
```

Cálculo en CUDA de B=E-Y*V'*E:

```
int aplicarIluminacionColor(int m, int n, float* E, float * devPtrE, float*
devPtrYt, int* devPtrV, float* devPtrIvalores, float* B, float* devPtrB, int can-
tHilosxBloque){

	int err;
	cublasStatus stat;


	//Copia n elementos de 'E' en CPU a 'E' en GPU.
	stat =  cublasSetVector (n*3, sizeof(float), (E), 1, devPtrE, 1);
	if (stat != CUBLAS_STATUS_SUCCESS) {
		cublasShutdown();
		return -1;
	}

	// vector ind con paso fijo
	// B=V'*E
	err = cublasSpMV4Color(n, m, -1, devPtrV, devPtrE, devPtrB_R, devPtrB_G,
devPtrB_B, devPtrIvalores, cantHilosxBloque);
	if (err != 0) {
		cublasShutdown();
	 return -1;
	}

	//Separo los componentes de color para realizar 3 calculos.
	for(int i=0; i<n; i++)
		E_Temporal[i] = E[i];
```

```
	stat = cublasSetVector(n, sizeof(float), E_Temporal, 1, devPtrE_R, 1);
	for(int i=0; i<n; i++)
		E_Temporal[i] = E[i + n];
	stat = cublasSetVector(n, sizeof(float), E_Temporal, 1, devPtrE_G, 1);
	for(int i=0; i<n; i++)
		E_Temporal[i] = E[i + 2*n];
	stat = cublasSetVector(n, sizeof(float), E_Temporal, 1, devPtrE_B, 1);

	// E=E-Y*B, notar que recibo Y transpuesta, aplico de nuevo transpuesta
para quedarme con Y.
	cublasSgemv ('T', m, n, -1, devPtrYt, m, devPtrB_R, 1, 1, devPtrE_R, 1);

	cublasSgemv ('T', m, n, -1, devPtrYt, m, devPtrB_G, 1, 1, devPtrE_G, 1);

	cublasSgemv ('T', m, n, -1, devPtrYt, m, devPtrB_B, 1, 1, devPtrE_B, 1);

	stat = cublasGetError();
	if( stat != CUBLAS_STATUS_SUCCESS ){
		cublasShutdown();
		return -1;
	}

	//Copia n elementos de 'E' en GPU a 'B' en CPU.
	stat = cublasGetVector (n, sizeof(float), devPtrE_R, 1, E_Temporal, 1);
	for(int i=0; i<n; i++)
		B[i] = E_Temporal[i];
	stat = cublasGetVector (n, sizeof(float), devPtrE_G, 1, E_Temporal, 1);
	for(int i=0; i<n; i++)
		B[i + n] = E_Temporal[i];
	stat = cublasGetVector (n, sizeof(float), devPtrE_B, 1, E_Temporal, 1);
	for(int i=0; i<n; i++)
		B[i + 2*n] = E_Temporal[i];

	if (stat != CUBLAS_STATUS_SUCCESS) {
		cublasShutdown();
		return -1;
	}

	return 0;
}
```

ANEXO G - Configuración de RadTR 2.0

En esta sección se describe el fichero de configuración *config.h*, se comenta la finalidad y uso de cada variable de configuración:

Variable que define el frustum.

```
#define FAR_FRUSTUM 70.0
#define NEAR_FRUSTUM 0.001
```

Define la cantidad de procesadores de la máquina.

```
#define NUMERO_CORES 2
```

Resolución de la pantalla al calcular factores de forma.

```
#define RESOLUCION_FACTOR_FORMA 256
```

Esta funcionalidad no se encuentra bien implementada, la idea es utilizar normales por vértices en vez de por cara.

```
//Si esta definido trata de usar las normales por vertices.
//Si no se define usara normales por cara.
//#define NORMAL_POR_VERTICE
```

Si está definida utiliza display list al dibujar la escena.

```
#define USAR_DISPLAY_LIST
```

Tiempo de espera entre un frame y otro, en ocasiones si es 0 se produce una degradación del rendimiento.

```
#define MS_ESPERA_FUNC_TIMER 0
```

Si está definida se aplica texturas a la escena.

```
#define APLICAR_TEXTURAS
```

Define la plataforma, es útil para futuras implementaciones en otros sistemas como por ejemplo en Linux.

```
#define WINDOWS
//Para futura implementacion.
```

```
//#define LINUX
```

Otra forma de subdividir la escena, no se utiliza porque produce errores en la lógica de los algoritmos utilizados.

```
//No se usa la division inteligente porque tiene errores.
//#define DIVISION_TRIANGULO_INTELIGENTE
```

Si está definida utiliza display list en paraboloide y hemicubo.

```
//Para utilizar display list en el hemicubo y paraboloide cuando se quiere
dibujar una escena.
#define USAR_DISPLAY_LIST_CALCULO_FACTOR_FORMA
```

Compila con las funcionalidades de CUDA.

```
#define USAR_CUDA
```

Compila con las funcionalidades de MATLAB.

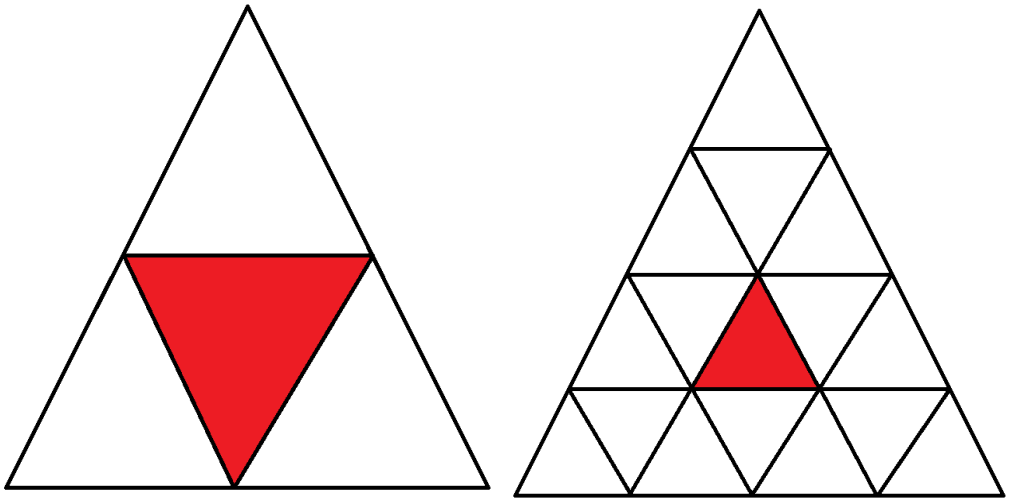
```
#define USAR_MATLAB
```

Si se define guarda una imagen en disco por cada factor de forma calculado al utilizar hemicubo o paraboloide.

```
//#define GUARDAR_IMAGENES_HEMICUBO_PARABOLOIDE
```

ANEXO H - Jerarquía de superficies

La jerarquía de superficies en RadTR 2.0 es utilizada para generar las matrices U y V. Para un grupo de superficies tal que sus factores de forma son similares, la jerarquía indica qué superficie es la que se debe considerar el factor de forma para el conjunto de superficies, se utiliza la coherencia espacial, esto quiere decir que triángulos relativamente cercanos sus factores de forma variarán muy poco. La jerarquía se crea cuando se subdivide la escena, a los triángulos se los divide en 4 triángulos iguales, en la primera iteración de división el triángulo central será el nodo padre de los 4 triángulos, en las sucesivas divisiones el nodo padre siempre será el triángulo central del nodo padre. Con este algoritmo se toma en cuenta que el triángulo central de un conjunto de triángulos será el "representante" del factor de forma de todos los triángulos del conjunto.

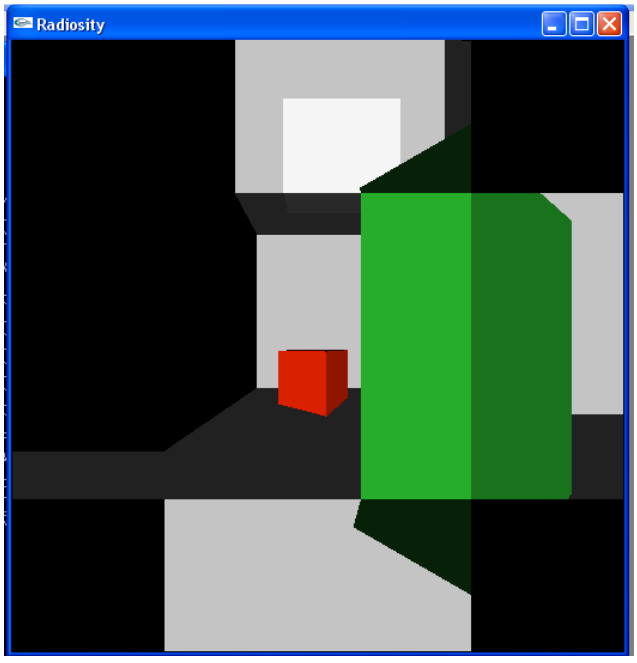


Izquierda, primera iteración de división. Derecha, segunda iteración de división. El triángulo rojo es el nodo padre.

ANEXO I - Hemicubo

Dado que existe cierto peso para cada pixel del hemicubo por la forma prismática del cubo y el ángulo por el que incide la luz, estos pesos se calculan en el momento de la creación de la instancia de la clase hemicube y son almacenados, esto sirve para cuando se utilicen los factores de forma (por ejemplo en el cálculo de radiosidad) no se pierde tiempo en el cálculo de estos pesos.

Para calcular los factores de forma la clase hemicube, se dibuja toda la escena cinco veces, una por cada cara del hemicubo desde la posición del observador, cada dibujo lo realiza en una zona de la pantalla, de esta forma es posible recuperar la pantalla completa desde la tarjeta de video y analizar las 5 caras del hemicubo, cada superficie de la escena debe ser dibujada con un color único y tener desactivado la iluminación, de esta forma en la imagen final se puede calcular fácilmente que superficie es visible desde el observador en cada pixel con solo observar el color del pixel, el análisis de la imagen final solo se reduce en recorrer los pixeles de la imagen y sumar uno a la superficie correspondiente con ese color. Notar que el fondo del mundo es negro y no existen superficies pintadas de negro, esto es para saber cuándo se está observando al vacio (no existe superficie en esa dirección).



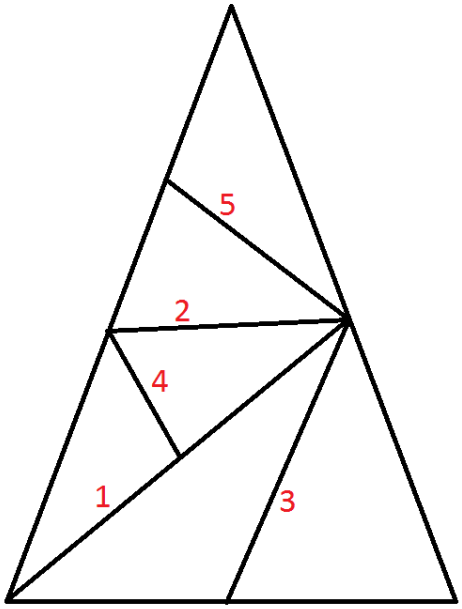
Vista en hemicubo.

ANEXO J - *Shaders* de división de triángulos en paraboloide

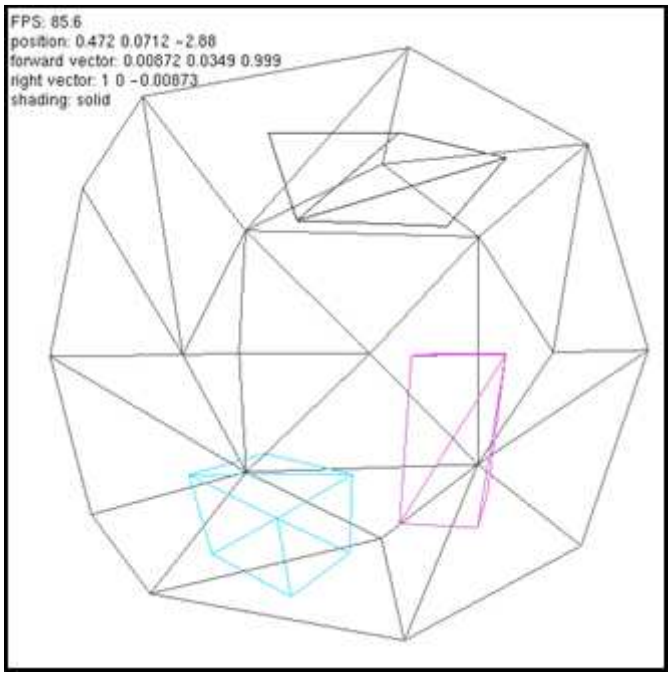
Esta sección trata sobre los *shaders* utilizados en el método del paraboloide para subdividir los triángulos en *geometry shader*.

División de triángulos

Es un algoritmo iterativo, comienza con el triángulo a dividir y en cada iteración elige el triángulo con la arista más larga visualmente, se divide dicho triángulo partiendo la arista larga en dos y así generando dos triángulos a partir del punto medio de la arista. La iteración finaliza cuando todas las aristas de los triángulos tengan un largo menor a cierto umbral o se llegue a una cantidad máxima de triángulos. En caso que se divida un triángulo por una arista interior (ver figura de abajo el corte número 4) se realiza una división del triángulo vecino por la misma arista, de esta forma se evita aristas mal divididas que visualmente genera "agujeros" en la geometría.



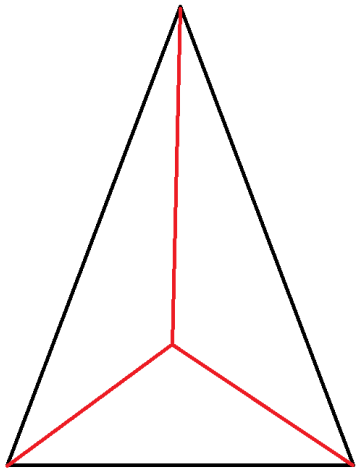
División de triángulos.



Escena con división de triángulos.

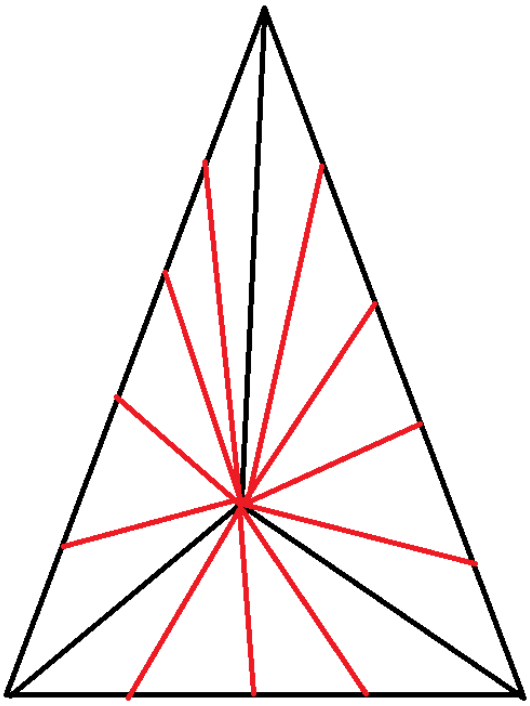
División desde el centro del triángulo

Esta división primero divide la superficie del triángulo en 3 triángulos, cada uno se halla trazando una arista desde el centro del triángulo hacia sus vértices.

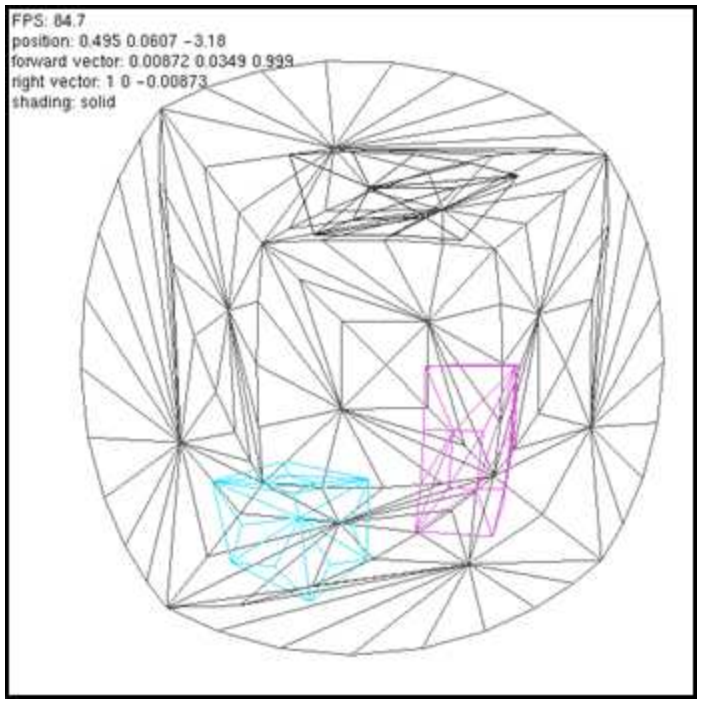


Primer paso de la división de triángulos desde el centro.

Luego para cada subtriángulo realiza tantas divisiones a la arista original hasta obtener una división de largo menor a un umbral, luego genera triángulos donde sus aristas parten del centro hasta las divisiones.

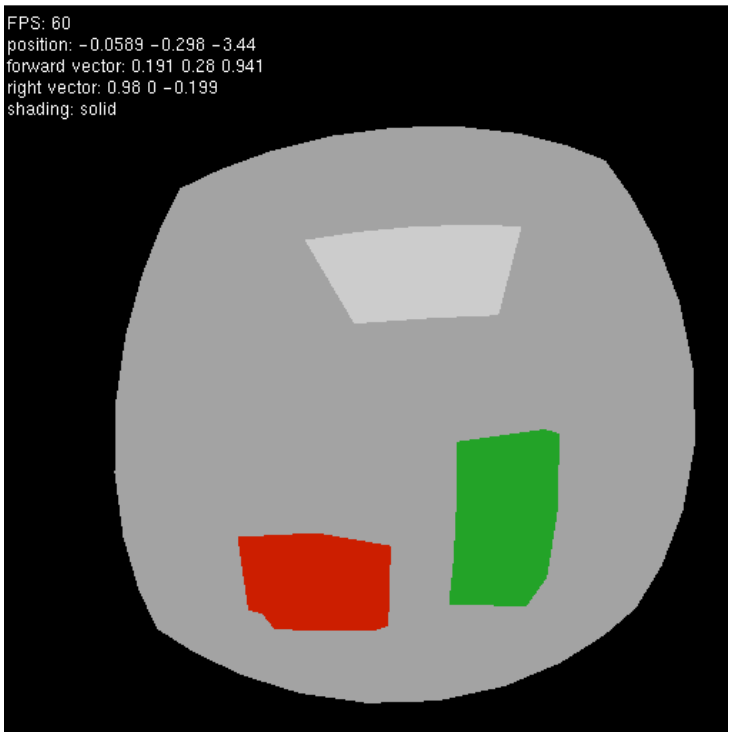


Generación de los triángulos con una arista sobre la arista original.



Escena con división desde el centro del triángulo.

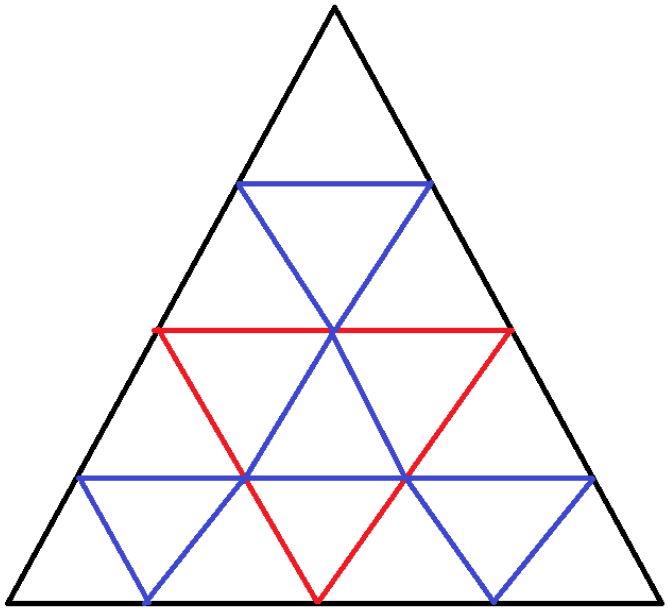
La ventaja de esta división es la facilidad que tiene en asegurar un largo máximo en las aristas exteriores de las superficies y no genera huecos en las aristas compartidas con otros triángulos, tiene como desventaja que genera triángulos demasiados largos y produce errores como se puede observar en la siguiente imagen.



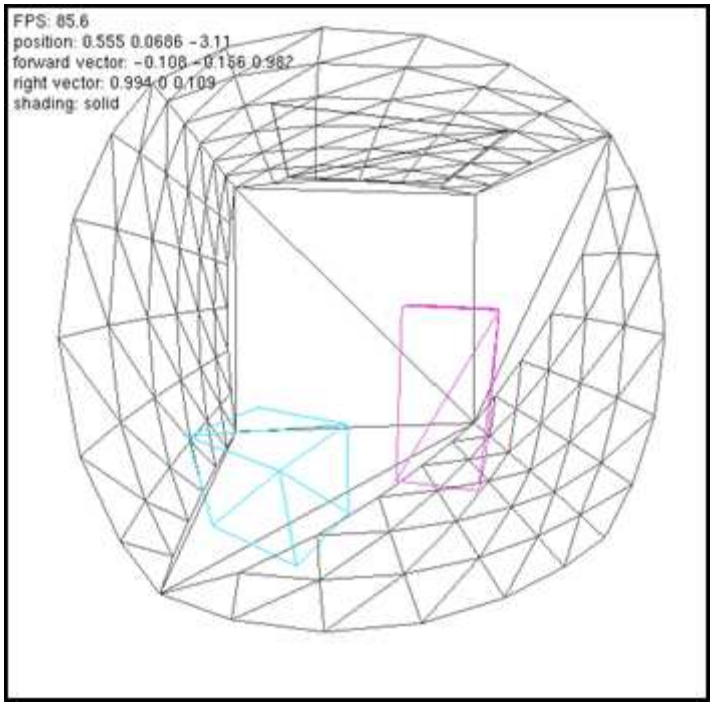
Error de superposición de triángulos, se ve el cubo rojo cortado.

División homogénea

Esta división es igual a la que se realiza cuando se subdivide la escena, cuando un triángulo es demasiado grande en pantalla se lo divide en varios triángulos. La ventaja de esta división es que genera triángulos con la misma proporción que el triángulo original evitando triángulos deformes (demasiados largos). Tiene la desventaja de que genera huecos en las aristas comunes con otros triángulos.



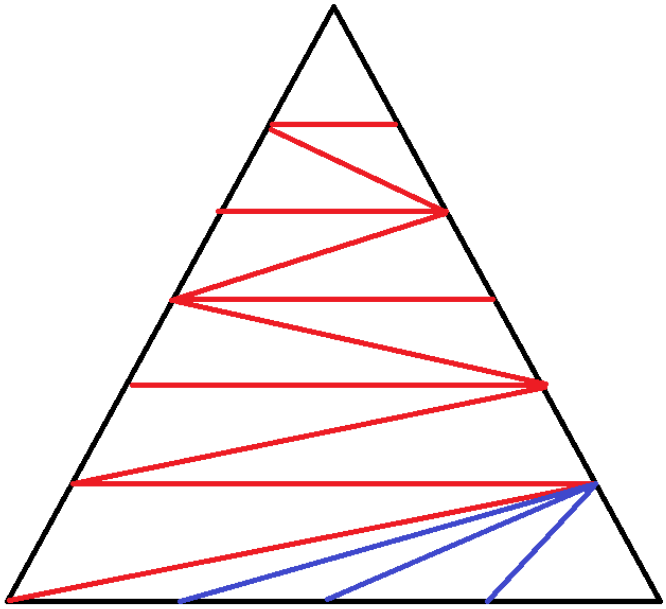
División de triángulos homogéneos.



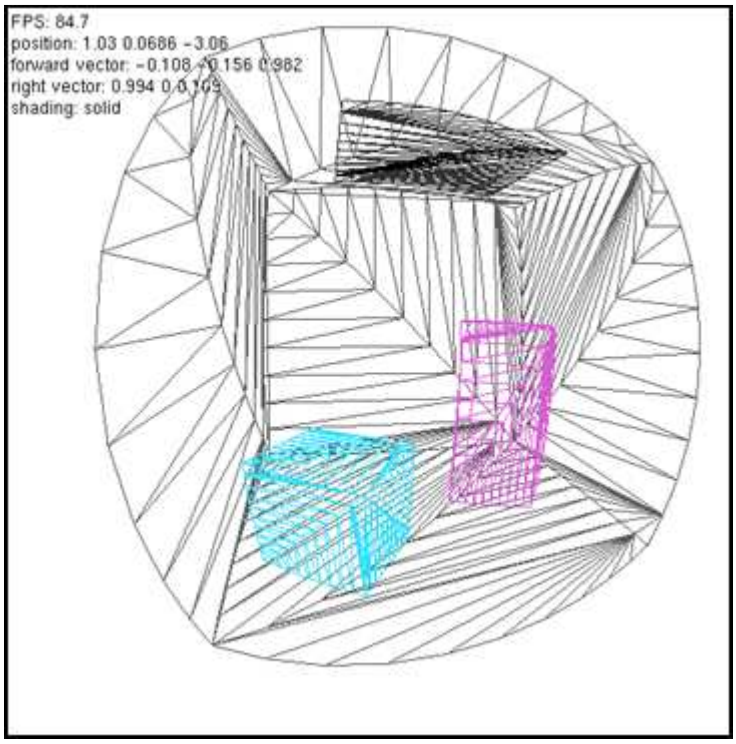
Escena con división de triángulos homogéneos.

División diagonal

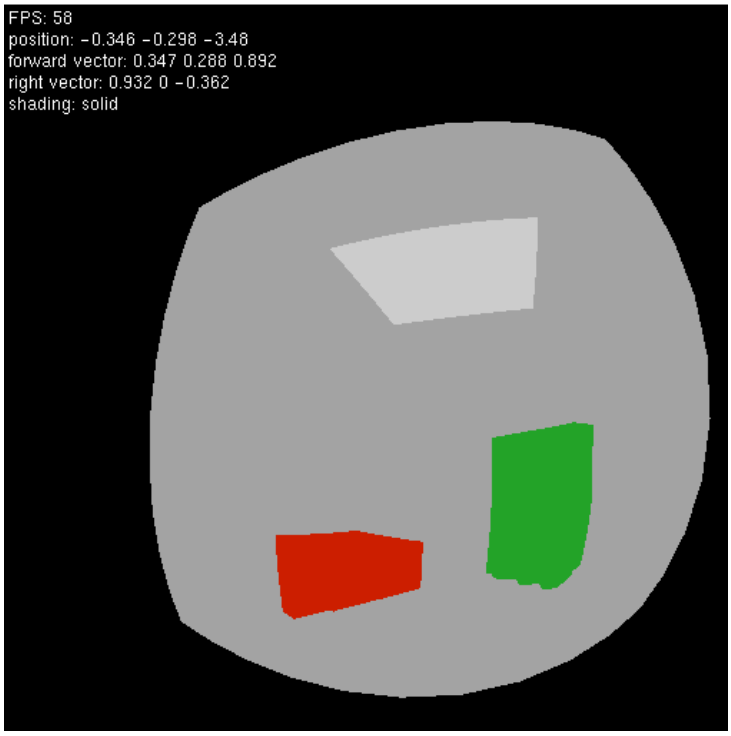
Realiza una división a las aristas generando triángulos en zigzag, con pocos triángulos realiza una buena división de las aristas pero tiene como desventaja que genera triángulos demasiados largo y genera errores en la imagen.



División en diagonal.



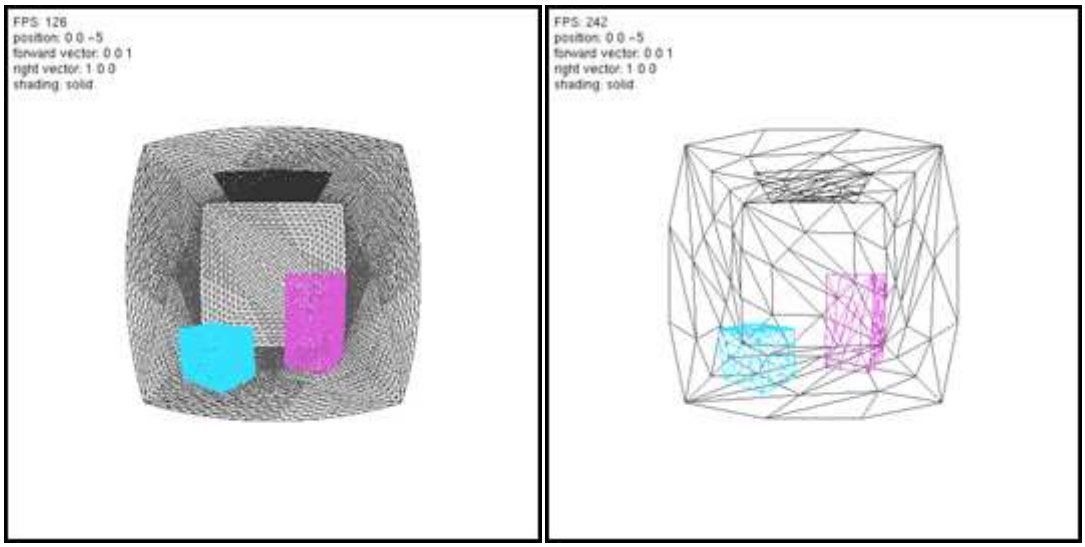
Escena con división el diagonal.



Se ve el error de esta división, el cubo rojo cortado.

División con *Tessellation shader*

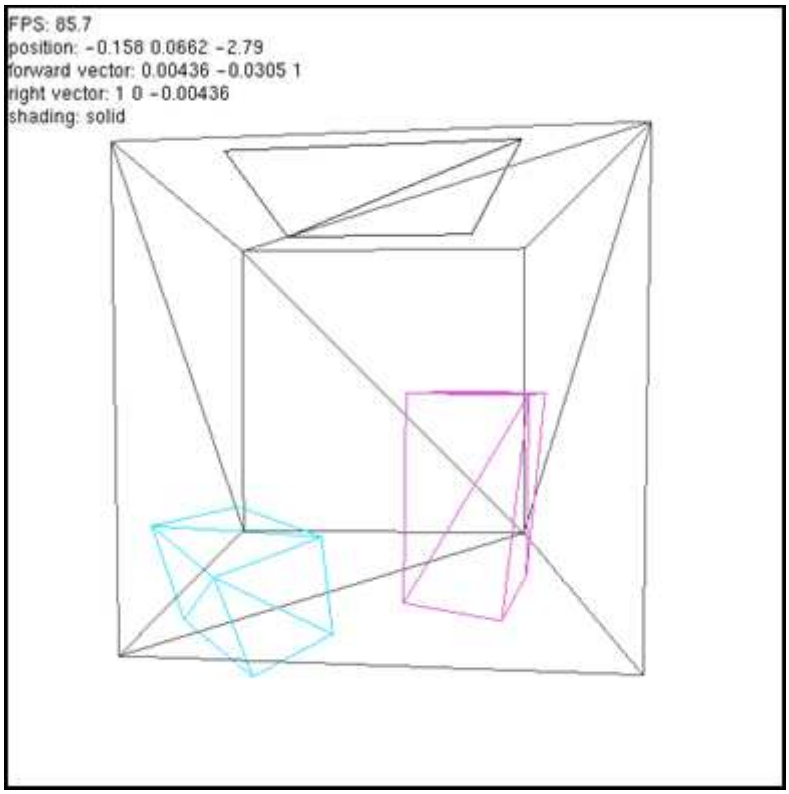
Para este tipo de división se utilizó *tessellation shader*, esta tecnología brinda la posibilidad de programar dos etapas más del pipeline gráfico que se encuentran entre el *vertex* y *geometry shaders* llamadas control y evaluation *shader*. En control *shader* se programa la cantidad de divisiones, existen dos tipos, la interna y la externa, en la interna se realiza una división parecida a la división homogénea ya que divide los triángulos creando puntos internamente, la división externa realiza una división igual que la del centro del triángulo, *tessellation shader* aplica una división utilizando los dos tipos de división. Luego *evaluation shader* genera los puntos de los triángulos teselados. En nuestro *shader* en esta etapa se realiza la proyección en el paraboloide.



Escena con teselado, izquierda utilizando 20 divisiones internas y externas mientras que en la derecha se utiliza 3. La escena original tiene 38 triángulos.

Sin división

Este *shader* se encuentra en la página de NVIDIA [4] y realiza la proyección de los vértices en un paraboloide, no realiza ningún tipo de división. El algoritmo utilizado para proyectar vértices es el utilizado en todos los *shaders* comentados anteriormente. Como es posible observar en la imagen de abajo el cubo rojo parece estar por fuera de la habitación, por esta razón este *shader* no es bueno para utilizarlo con triángulos demasiados grandes y conviene dividir la escena.



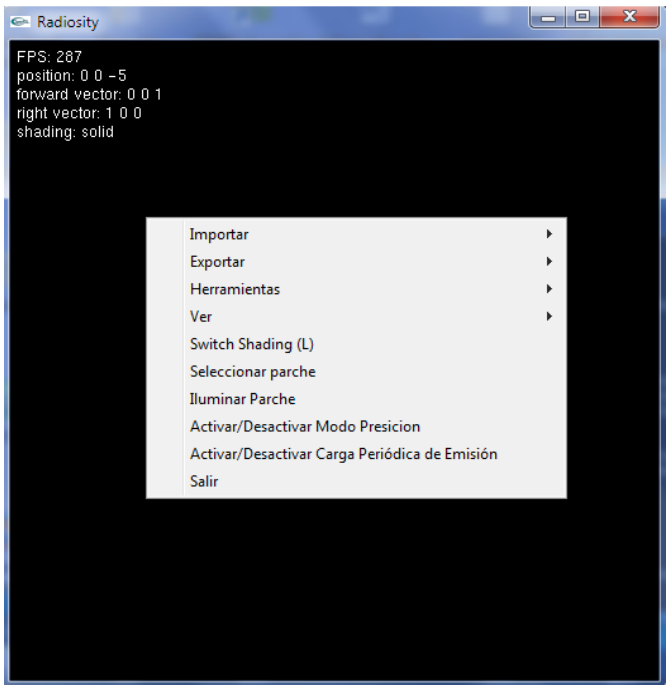
Escena utilizando el shader de NVIDIA, sin división, sin división se ve al cubo rojo saliendo de la habitación cuando no debería.

Anexo K – Manual de usuario

El objetivo de presente anexo es describir el modo de uso de la nueva interfaz gráfica de RadTR 2.0.

Menú

Al menú principal de RadTR 2.0 se accede presionando el botón derecho del mouse en el área de renderizado de la aplicación.



La jerarquía de opciones de dicho menú es la siguiente:

- Importar
    - Cargar emisión (F7)
    - Cargar escena (F2)
- Exportar
    - Generar ficheros (F6)
    - Guardar escena (F3)

- Herramientas
    - Screenshot
    - Calcular radiosidad
    - Radiosidad Tiempo Real
    - Calcular Factor Forma Cielo
    - Animación
    - Subdividir
    - *Shaders*
    - Carga Secuencial
    - Aumentar Iluminación
- Ver
    - Cambiar Wireframe
    - Cambiar Caché
    - Ver Normal
    - Ver Hemicubo
    - Ver Paraboloide
    - Ver Proyección Paralela
    - Mostrar/Ocultar Información
    - Cámara
        - Guardar Estado Actual
        - Cargar Estado
        - Setear Estado
- Switch Shading
- Seleccionar Parche
- Iluminar Parche
- Activar/Desactivar Modo Precisión
- Activar/Desactivar Carga Periódica de Emisión
- Salir

A continuación se describirán brevemente las diferentes funcionalidades.

Importar - Cargar Emisión

Carga desde un fichero una iluminación de la escena. En dicho fichero se encuentra para cada parche la cantidad de luz que emite. Es utilizado cuando los cálculos de iluminación de la escena se realizan por fuera del software RadTR 2.0 y se desea asignar dicha iluminación a la escena.

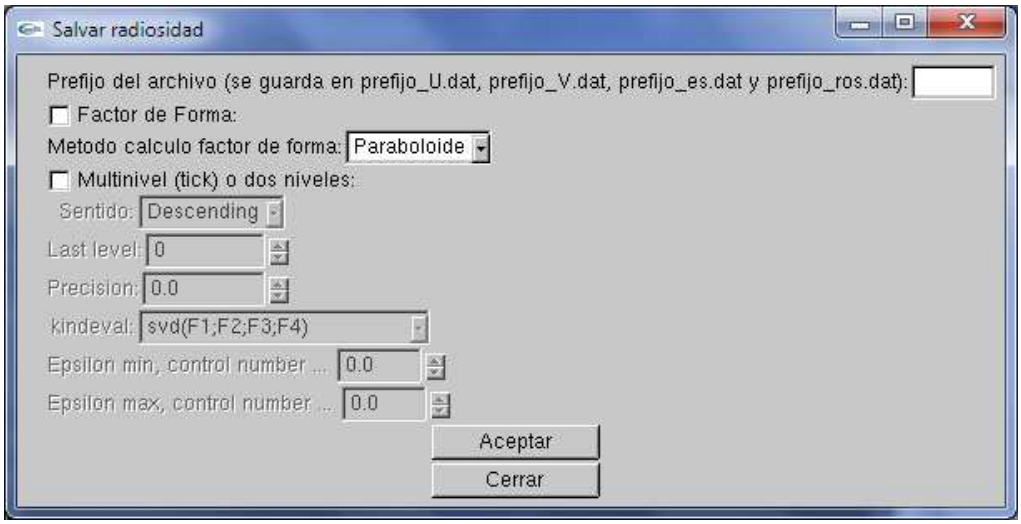
Importar – Cargar Escena

Se carga una nueva escena 3D desde un fichero, el mismo debe tener el formato que soporta la aplicación RadTR 2.0.

Exportar – Generar Ficheros

Esta funcionalidad se utiliza para guardar diferentes datos relacionados con radiosidad, entre ellos se encuentra los factores de forma de la escena (_ffs.dat), la matriz U (_U.dat), la matriz V (_V.dat), la emisión de luz de cada parche (_es.dat), el color de cada superficie (_ros.dat), el área de cada superficie (_area.dat) y la geometría de los triángulos (_triangulo.dat).

Primero se puede especificar un prefijo para los nombres de los ficheros a generar, luego, si se desea calcular los factores de forma, se debe activar dicha sección como también especificar el método a utilizar para generarlos (Paraboloide o Hemicubo). Por último, es posible utilizar multinivel para la generación de las matrices U y V especificando diferentes parámetros o simplemente utilizar dos niveles.



Exportar – Guardar Escena

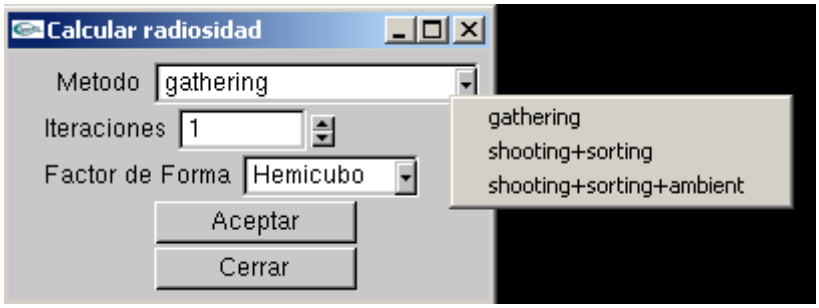
Guarda la escena actual en cualquiera de los formatos soportados por la aplicación (AC3D o 3DS), si se desea almacenar triángulos que poseen emisión de luz se recomienda utilizar AC3D ya que el formato 3DS no guarda este tipo de información.

Herramientas – Screenshot

Realiza una captura de la pantalla de la aplicación y la almacena como una imagen con formato BMP.

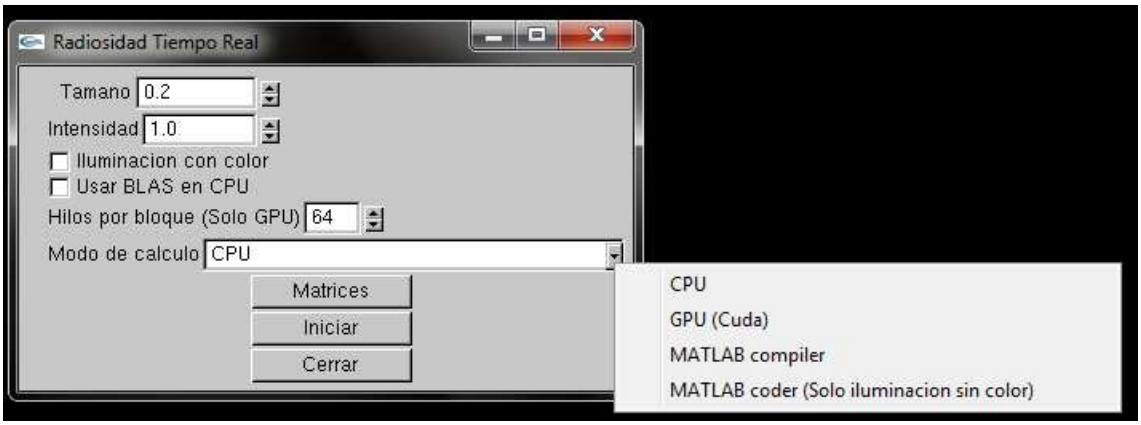
Herramientas - Calcular radiosidad

Realiza el cálculo de radiosidad a la escena actual, se puede escoger uno de los tres métodos disponibles que son Gathering (para cada parche recolecta la luz que le es transmitida desde toda la escena), Shooting+sorting (en cada iteración escoge el parche con más energía lumínica y la dispersa hacia toda la escena) y shooting+sorting+ambient. Se debe especificar la cantidad de iteraciones del algoritmo como también el método a utilizar para el cálculo de los factores de forma (Paraboloide o Hemicubo).



Herramientas - Radiosidad Tiempo Real

Al ejecutar esta opción se despliega la imagen mostrada en la figura.



Tamaño: El tamaño de la "linterna", esto se puede cambiar mientras ejecuta el algoritmo con las teclas '+' y '-'.

Intensidad: Intensidad de la luz de la "linterna".

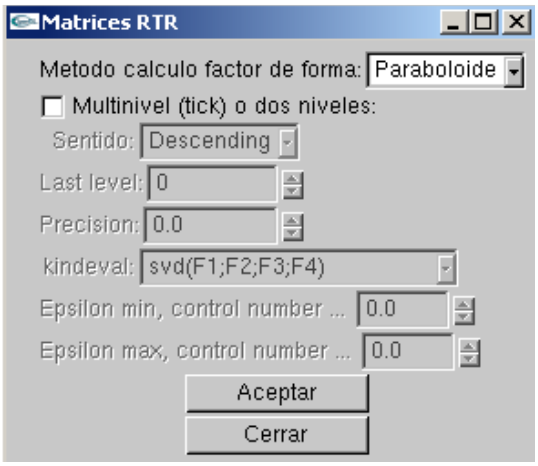
Iluminación con color: Si debe realizar los cálculos teniendo en cuenta los colores de las superficies o simplemente se ignoran y se cálcula con luz monocromática.

Usar BLAS en CPU: Habilita la librería BLAS en la implementación de CPU para utilizarla en la multiplicación de matrices.

Hilos por bloque: Hilos por bloque cuando ejecuta en GPU (CUDA).

Modo de cálculo: Se puede especificar donde se deben realizar los cálculos, en CPU, GPU (CUDA), MATLAB Compiler o MATLAB Coder (este último solo con iluminación sin color).

En la sección de Matrices se asignan parámetros para la generación de las matrices U y V que se realiza al comienzo del algoritmo. Entre las posibilidades que brindan dichos parámetros se encuentra la de asignar el método de cálculo de los factores de forma (Paraboloide o Hemicubo).



Herramientas - Calcular Factor Forma Cielo

Se calculan los factores de forma de la escena desde el punto de vista del cielo, se definen los puntos en el cielo según los parámetros Alfa y Beta, dichos puntos se encuentran horizontalmente a una distancia descripta por el ángulo Alfa y verticalmente por el ángulo Beta. La opción “Generar imágenes” genera una imagen por punto guardando la vista de la escena desde ese punto, “Generar ficheros” genera dos ficheros: uno llamado “<prefijo>_FF”, que guarda una matriz donde en cada fila se almacenan los factores de forma de cada parche para un punto dado (hay tantas filas como puntos en el cielo definidos), y el otro fichero llamado “<prefijo>_InfoPuntos”, que guarda la información de cada punto del cielo (posición, dirección y vector Up). La opción “Aplicar escena” aplica iluminación a la escena como si el cielo estuviera emitiendo luz.

Herramientas – Animación (F11)

Al activarlo cada movimiento que se realice se le pedirá al usuario que guarde la captura de pantalla actual, luego el usuario tendrá una secuencia de imágenes del movimiento realizado.

Herramientas – Subdividir

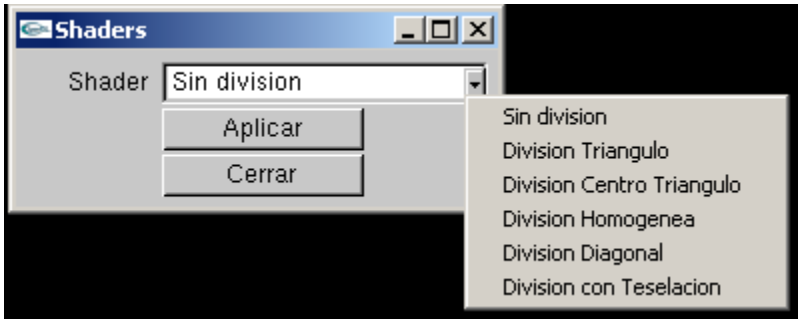
Se aplica subdivisión a la escena. Esta funcionalidad es útil para luego generar las matrices U y V ya que genera toda la estructura jerárquica de superficies. “Niveles” indica la cantidad de tandas de subdivisiones que se aplicarán realmente (Niveles + 1 tandas). En cada una de estas subdivisiones se usará el área máxima definida dividido 2, para la primera división se utilizará “Área máxima”, para la segunda tanda de división se utilizará “Área máxima” / 2, para la tercera tanda “Área máxima” / 4 y así sucesivamente. En cada tanda de subdivisiones se dividirá cada triángulo de la escena hasta que todos tengan de área menos que el área máxima definida para la tanda.

Herramientas – Shaders

Aquí se especifica el *shader* utilizado en el método del Paraboloide (o Hemiesfera) para el cálculo de los factores de forma, los posibles *shaders* son:

- Sin división: No se aplica división de los triángulos, solo aplica transformación de paraboloide a cada vértice. Utiliza *Vertex* y *Fragment Shaders*.
- División Triángulo: Realiza una división tratando de minimizar visualmente el largo de cada arista, luego aplica transformación de paraboloide a todos los vértices. Utiliza *Geometry Shader*.
- División Centro Triángulo: Divide al triángulo desde su centro hacia las aristas, luego aplica transformación de paraboloide. Utiliza *Geometry Shader*.
- División Homogénea: Divide al triángulo en una cantidad específica de triángulos cuando el observador se encuentra cerca, luego aplica transformación de paraboloide. Utiliza *Geometry Shader*.

- División Diagonal: Divide al triángulo en forma de zig-zag y luego aplica transformación de paraboloide. Utiliza *Geometry Shader*.
- División con Teselado: Utiliza *Tessellation Shader* para dividir al triángulo y luego aplica transformación de paraboloide.



Herramientas - Carga Secuencial

Esta funcionalidad permite realizar la carga de archivos de emisión para generar una serie de escenas a partir de los mismos. Para la misma se debe tener un conjunto de archivos cuya nomenclatura consista de un prefijo seguido y un número de secuencia, o sea, <Prefijo Lectura> +<Índice Secuencial>. Se genera una secuencia de escenas correspondientes cuyos nombres son de la forma <Prefijo Escritura> + <Índice Secuencial>.

Al seleccionar esta opción se despliega el formulario de la imagen.

Desde: Número que indica el número inicial de la secuencia de archivos.

Hasta: Número que indica el número final de la secuencia de archivos.

Prefijo Lectura: Texto del prefijo de los archivos a leer.

Prefijo Escritura: Texto del prefijo de los archivos que se crearán.

Path Lectura: Carpeta en la cual se leerán los archivos.

Herramientas - Aumentar Iluminación

Con esta funcionalidad es posible aumentar o disminuir la iluminación de una escena, el valor ingresado es el factor que multiplicará a la emisión de luz de cada parche, con valores mayores a 1 se aumentará la iluminación y con valores menores a 1 se la disminuirá.

Ver - Cambiar Wireframe

Cambia la forma de visualizar los triángulos de la escena, con wireframe se verá solo el contorno de los triángulos (sus aristas) y sin wireframe se verá todo el triángulo.

Ver - Cambiar Caché

Activa y desactiva caché de los factores de forma, en un principio se encuentra desactivado. El caché es útil cuando se requiere los factores de forma ya que será necesario calcularlos una vez para cada parche y así acelerar los algoritmos que los requiere, la memoria necesaria para el caché crece de forma polinomial con la cantidad de triángulos (la cantidad de bytes necesario será: 4*(cantidad_triángulos)^2) por lo tanto para escenas grandes no será posible utilizar caché.

Ver - Ver Normal

Cambia la vista a forma normal, esto es útil cuando la vista está en Paraboloide, Hemicubo o Paralela y se quiere ver de forma normal.

Ver - Ver Hemicubo

Activa la vista en hemicubo, se dibuja la escena en 5 cuadrantes donde cada uno es una cara de un cubo que se encuentra sobre el observador (ver técnica del hemicubo).

Ver - Ver Paraboloide

Activa la vista en paraboloide, dibuja la escena con una transformación en paraboloide (o hemiesfera) desde el punto de vista del observador.

Ver - Ver Proyección Paralela

Activa la vista en paralelo (ortogonal).

Ver - Mostrar/Ocultar Información

Muestra y oculta la información en pantalla, la información mostrada es:

FPS: Cantidad de frame por segundos actuales.

Position: Posición actual del observador.

Forward vector: Vector que apunta hacia atrás de la vista del observador.

Right vector: Vector que apunta hacia la derecha de la vista del observador.

Shading: Modo de dibujado de los triángulos (Ver Switch Shading).

Cuando se está ejecutando radiosidad en tiempo real se agrega:

Tiempo cálculo RTR por segundo: Muestra los milisegundos que consume el algoritmo por segundo y entre paréntesis el tiempo en milisegundos que tarda en calcular la iluminación inicial (Ilum. Inicial) y el tiempo de calcular a partir de la iluminación inicial la iluminación final (Cal. Rad.).

Ver - Cámara - Guardar Estado Actual

Guarda el estado del observador actual con nombre “Estado Posición <numero>”, se guarda la posición y la dirección de vista del observador para que luego pueda ser cargado.

Ver - Cámara - Cargar Estado

Carga un estado del observador, se carga la posición y la dirección de vista.

Ver - Cámara - Setear Estado

Es posible asignar manualmente un estado del observador, para esto se debe especificar un nombre, la posición del observador, el vector que apunta hacia atrás, el vector que apunta hacia la derecha y un último vector que apunta hacia arriba.

Switch Shading

Cambia la forma en que se dibuja los triángulos, también se puede utilizar la tecla L. Las formas posibles son:

Solid: Dibuja los triángulos con su color.

Emission: Dibuja a los triángulos con su emisión de luz.

Interpolated emission: Dibuja a los triángulos con su emisión de luz, a diferencia del anterior se interpola la emisión dando un color más suave.

Color code: El color de los triángulos será el código de color, este modo se utiliza para identificar a los triángulos visualmente por su color.

Solid with the patchs distinguished: Se dibuja los triángulos con color la emisión y se traza con línea blanca sus aristas para distinguir los triángulos.

Seleccionar Parche

Esta funcionalidad posiciona al observador en el centro del triángulo seleccionado y la vista será la normal del mismo, el triángulo seleccionado será el que se le hizo click derecho cuando se abre el menú.

Iluminar Parche

Esta funcionalidad ilumina al parche seleccionado con el click derecha, si ya se encuentra iluminado entonces le quita la iluminación. Es útil para asignar una iluminación a una escena de forma manual.

Activar/Desactivar Modo Precisión

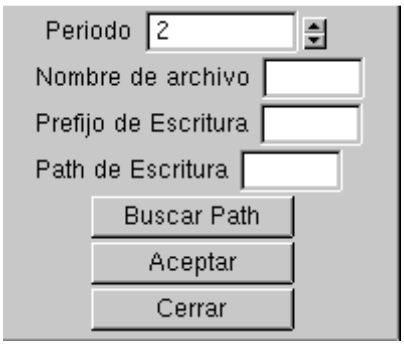
Este modo es útil cuando se quiere realizar movimientos precisos en la posición y vista del observador. Cuando esta activo los movimientos de posición se realizarán de a una unidad. Si se presiona la tecla Shift se hacen movimientos más finos, en posición se mueve 0.01 unidades y los movimientos del mouse también son más precisos.

Activar/Desactivar Carga Periódica de Emisión

Esta funcionalidad permite realizar la carga periódica de un archivo de emisión existente. Cada cierto tiempo determinado previamente, se carga dicho archivo para poder visualizarlo en pantalla y a su vez se almacena una captura de pantalla.

La nomenclatura para los archivos a guardar es la siguiente: <Prefijo de Escritura> + <índice secuencial de la imagen generada> + ".bmp".

Cuando se elije esta opción se muestra el formulario de la figura.



Período: Número que indica la cantidad de segundos que pasarán entre cada carga de emisión.

Nombre de archivo: Nombre del archivo que se cargará periódicamente.

Prefijo de Escritura: Prefijo de los archivos que se irán creando por cada emisión cargada.

Path de Escritura: Carpeta en la cual se guardarán las capturas.

Anexo L – Ejemplos de utilización de MATLAB con RadTR 2.0

Generando librerías compartidas e inclusión en Visual Studio

Para generar la librería compartida en C, desde la aplicación MATLAB o desde la línea de comandos de Windows, previamente ubicados en el directorio donde se encuentra *rtrMat.m*, se introduce el siguiente comando:

```
mcc –l rtrMat.m
```

A partir de este se generan determinados archivos, de los cuales sólo interesan los siguientes:

rtrMat.h

rtrMat.lib

rtrMat.dll

El archivo *rtrMat.h* debe ser incluido en RadTR 2.0, mientras que rtrMat.lib debe ser declarado como entrada al vinculador en Visual Studio. rtrMat.dll debe ubicarse en la misma carpeta que el ejecutable de RadTR 2.0. Se podría construir un script muy simple que compile y mueva los archivos a sus correspondientes directorios, aunque este depende de cómo esté configurado el proyecto en Visual Studio.

Como el código en *rtrMat.m* es un script y no una función, si se modifica no existirán cambios en los archivos *rtrMat.h* y *rtrMat.lib*, ya que no hubo cambios en los cabezales de las funciones. Dichas modificaciones sólo impactarán en *rtrMat.dll*. Por lo tanto, sólo habría que actualizar este último archivo sin tener que compilar de nuevo el proyecto de RadTR 2.0.

Bajo estas condiciones, se encuentra una gran sencillez en el camino de modificar el archivo rtrMat.m y la ejecución de RadTR 2.0, el cual se resume a compilar el código MATLAB y actualizar la librería dinámica.

Ejemplo de aplicación de MATLAB *Coder*

Generación del código

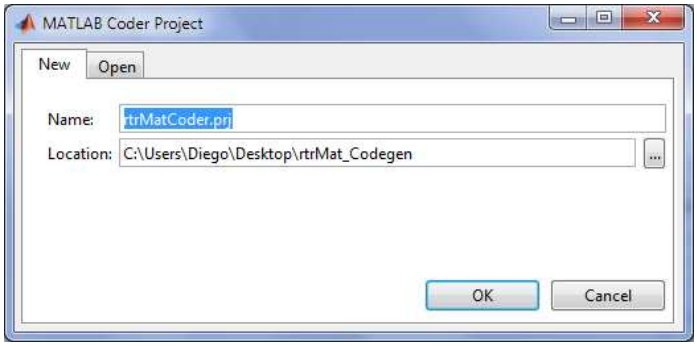
Para este caso, se debe modificar el script contenido en *rtrMat.m* para transformarlo en una función. Para diferenciar, se denomina al nuevo código como *rtrMatCoder.m*. Para generar el código C++ del código MATLAB contenido en dicho archivo se realizan los siguientes pasos.

- Abrir la aplicación MATLAB.
- Posicionar el folder actual de MATLAB en la dirección donde se encuentra el código a partir del cual se va a generar.

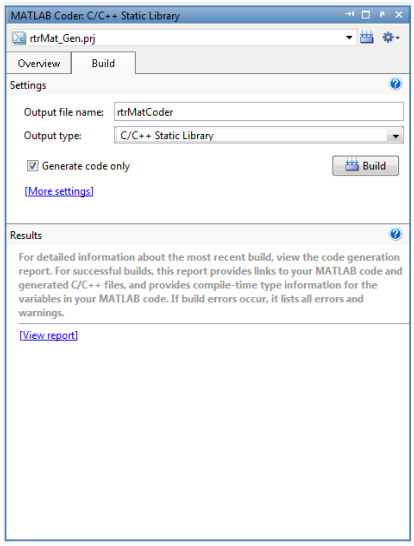
- Ingresar en la línea de comandos de MATLAB:

>> coder

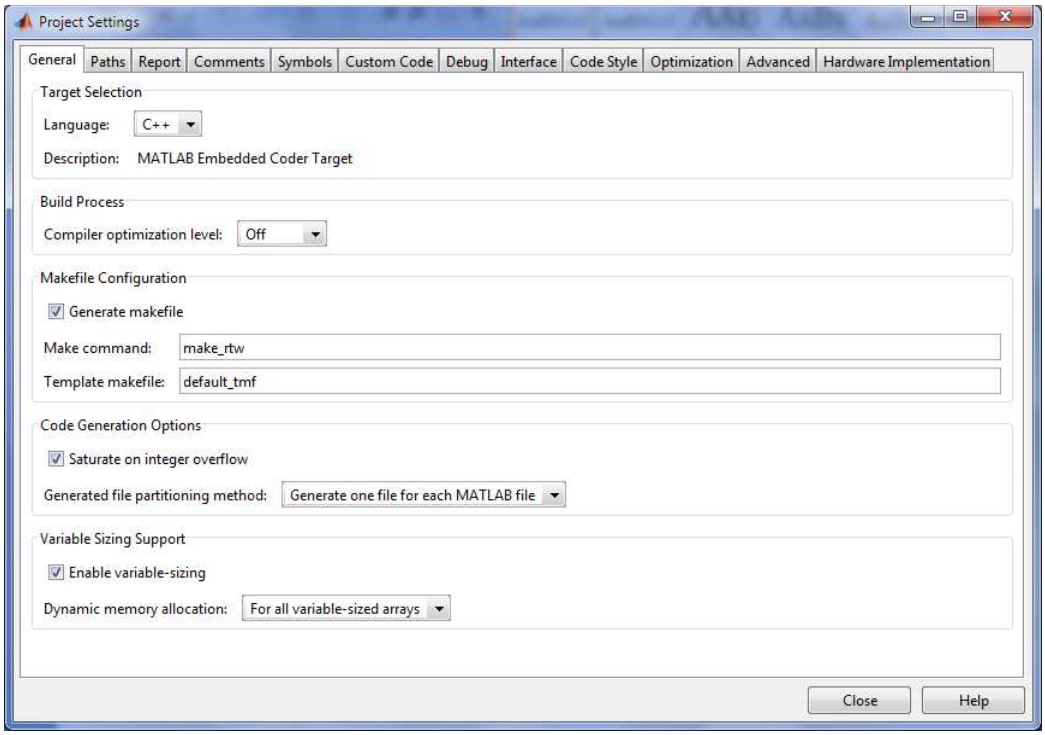
- Allí se despliega una interfaz gráfica en la cual se puede crear un nuevo proyecto de generación. Se elige el nombre el nombre del proyecto *rtrMatCoder.prj* a modo de ejemplo.



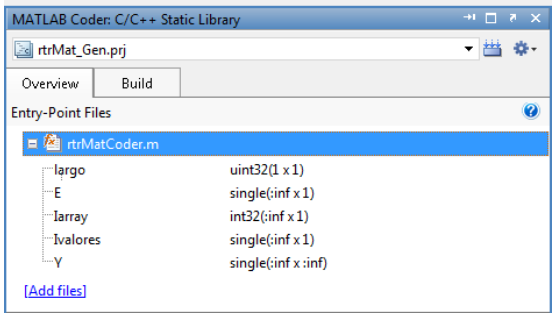
- Al dar *OK* se muestra un panel dentro de MATLAB en el cual se pueden configurar las opciones de generación. Allí se elige la opción "C/C++ Static Library" y se activa la opción el *checkbox* "Generate code only". Luego se hace click en "[More settings]".



- Se despliega una ventana con muchas opciones para configurar en la cual sólo se indicará que el lenguaje de generación sea C++ y que se permita la reserva dinámica de memoria. Esto debe ser así ya que no se conoce a priori que tamaño tendrán los vectores que se le pasen a la función.



- Ahora sólo resta definir la función a partir de la cual se va a realizar la generación. Se selecciona la solapa “Overview” y se agrega el punto de entrada *rtrMatCoder.m.* También allí se debe definir cuidadosamente los tipos de los datos de entrada. En caso que alguna dimensión de algún parámetro sea variable se coloca el valor máximo de la misma. Si se desconoce este máximo se indica como “:inf”, como se hizo en este caso. Hay que tener sumo cuidado en este paso, ya que un tipo mal definido podría causar que el código se genere incorrectamente.



- Luego de realizada toda la configuración ir a la solapa “Build” y presionar el botón “Build”.

Inclusión del código en el proyecto de Visual Studio

Seguidos los pasos anteriores, incorporar el código al proyecto en Visual Studio es muy sencillo, y sólo se limitaría a alojar los archivos en la carpeta adecuada e incluirlos en el proyecto.

El siguiente paso es realizar la invocación a la función generada. Para este cometido interesan las funciones proporcionadas en rtrMatCoder.h y rtrMatCoder_emxAPI.h. En rtrMatCoder.h se encuentra la definición del cabezal de la función que se generó, la cual recibe parámetros cuyos tipos también fueron generados. Para inicializar variables en estos tipos se dispone de la API que se encuentra en rtrMatCoder_emxAPI.h. En el Anexo C se da una breve explicación sobre cómo se generan los arreglos dinámicos.