



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

TÉCNICA HÍBRIDA DE GENERACIÓN DE IMÁGENES A PARTIR DE MODELOS TRIDIMENSIONALES

Liberto José Camús Valdés

Dirigido por: Miguel Romero Hortelano

Curso: CURSO 2016-2017



TÉCNICA HÍBRIDA DE GENERACIÓN DE IMÁGENES A PARTIR DE MODELOS TRIDIMENSIONALES

Proyecto de Fin de Grado en Ingeniería Informática de modalidad específica

Realizado por: Liberto José Camús Valdés
Dirigido por: Miguel Romero Hortelano

Fecha de lectura y defensa: 12 de junio de 2017

Resumen del proyecto y palabras clave

El presente proyecto tiene como objetivo principal la realización de la implementación de una técnica de generación de imágenes por ordenador híbrida entre la rasterización y el trazado de rayos y la posterior comparación del resultado obtenido con cada una de las dos técnicas clásicas por separado.

Para el desarrollo del mismo, el proyecto se ha dividido en una consecución de etapas que podemos resumir en: inicialmente se analizó la técnica de generación de imágenes por ordenador conocida como rasterización. Este análisis consistió en la implementación de una versión básica de dicha técnica denominada sombreado directo y en la posterior modificación de esta implementación para analizar una segunda versión conocida como sombreado diferido. El sombreado diferido consta de dos pasadas; en la primera se realizan los cálculos dependientes de la geometría y se almacenan los resultados en reservas de memoria. En la segunda pasada se añade la iluminación y se reutilizan los resultados almacenados de la pasada anterior para evitar tener que repetir los cálculos de geometría por cada fuente de luz añadida.

En la siguiente etapa del presente proyecto se implementó y analizó la técnica híbrida que combina la rasterización con el trazado de rayos. La combinación se realizó sustituyendo la segunda pasada del sombreado diferido por un programa codificado en la plataforma de computación paralela heterogénea OpenCL y que se ejecutó en la tarjeta gráfica. Para probar la viabilidad de esta combinación se añadieron a la imagen generada sombras calculadas mediante trazado de rayos en tiempo real.

Posteriormente se analizó un método para acelerar los cálculos del trazado de rayos consistente en ordenar la geometría de la escena en una jerarquía de volúmenes envolventes, minimizando el número de comprobaciones de intersección rayo-triángulo a realizar.

Finalmente, con la finalidad de valorar y comparar el efecto del uso de OpenCL y el cálculo en tarjeta gráfica con respecto a otras opciones disponibles, se implementó la segunda fase de la técnica híbrida en C++ ejecutándose en la CPU.

Palabras clave: OpenCL, Ray Tracing, Trazado de Rayos, Rasterización, Técnica Híbrida, Sombreado Diferido, Paralela, Heterogénea

Hybrid Technique to produce images from three dimensional models

Abstract

The main objective of the present project is the implementation of a hybrid computer imaging technique between rasterization and ray tracing and the subsequent comparison of the result obtained with each of the two classical techniques separately.

For the development of the project, it has been divided into a series of stages that can be summarized in: initially the technique of computer imaging known as rasterization was analyzed. This analysis consisted in the implementation of a basic version of this technique called forward shading and in the subsequent modification of this implementation to analyze a second version known as deferred shading. Deferred shading consists of two passes; In the first one the calculations dependent of the geometry are made and the results are stored in memory reserves. In the second pass lighting is added and the stored results of the previous pass are reused to avoid having to repeat geometry calculations for each added light source.

In the next stage of the present project, the hybrid technique that combines rasterization with ray tracing was implemented and analyzed. The combination was performed by replacing the second pass of the deferred shading by a program coded in the OpenCL heterogeneous parallel computing platform and executed on the graphics card. To test the feasibility of this combination, shadows calculated by real-time ray tracing were added to the generated image.

Later, a method was applied to accelerate ray tracing calculations consisting of ordering the scene geometry in a hierarchy of bounding volumes, minimizing the number of ray-triangle intersection checks to be performed.

Finally, in order to evaluate and compare the effect of the use of OpenCL and the calculation in graphic card with respect to other available options, the second phase of the hybrid technique was implemented in C++ and executed in the CPU.

Keywords: OpenCL, Ray Tracing, Rasterization, Hybrid Technique, Deferred Shading, Parallel, Heterogeneous.

Tabla de contenido

1 Introducción y objetivos del proyecto	11
1.1 Evolución histórica reciente.....	11
1.2 Situación actual y objetivo del siguiente trabajo.....	12
2 Estado del arte	15
3 Revisión tecnológica y criterios de elección	19
3.1 Criterios de elección.....	19
3.2 Herramientas y entorno de desarrollo.....	20
3.3 OpenGL.....	21
3.4 OpenCL.....	22
3.5 OpenMP.....	22
3.6 Bibliotecas de terceros.....	22
3.7 Revisión de C++.....	23
4 Técnicas de renderizado y métodos de aceleración	25
4.1 Rasterización.....	25
4.1.1 Sombreado directo.....	27
4.1.2 Sombreado diferido.....	28
4.2 Trazado de rayos.....	29
4.3 Técnica híbrida.....	32
4.4 Aceleración mediante BVH.....	33
5 Desarrollo	35
5.1 Estrategia de implementación.....	35
5.2 Iteración 1. Sombreado directo.....	35
5.3 Iteración 2. Sombreado diferido.....	39
5.4 Iteración 3. Técnica híbrida con OpenCL.....	41
5.4.1 Implementación.....	41
5.4.2 Optimización y estudio de rendimiento.....	47
5.5 Iteración 4. Aceleración de la técnica híbrida mediante BVH.....	58
5.5.1 Implementación.....	58
5.5.2 Estudio de rendimiento y mejora obtenida al introducir BVH.....	60
5.6 Iteración 5. Técnica híbrida sin OpenCL.....	63
5.6.1 Implementación.....	63
5.6.2 Comparativa de rendimiento con la Iteración 4.....	64
5.7 Diagrama de clases.....	65
6 Comparativa de la técnica híbrida frente a rasterización y trazado de rayos	67
6.1 Comparación de rendimiento.....	67
6.2 Calidad de la imagen obtenida.....	71
7 Conclusiones	77
8 Planificación y presupuesto	79
9 Bibliografía	83

10 Acrónimos y siglas	89
11 Anexos	91
11.1 Descripción de la arquitectura de OpenCL.....	91
11.2 Instrucciones para construir y ejecutar el proyecto.....	97
11.3 Listado de archivos de código fuente y breve descripción de su función.....	100
11.4 Formato del archivo de descripción de escenas.....	103
11.5 Listado de código fuente.....	105
11.5.1 bbox.hpp.....	105
11.5.2 bvh.hpp.....	106
11.5.3 bvh.cpp.....	107
11.5.4 bvhtablenode.hpp.....	109
11.5.5 bvhtableprimitiveinfo.hpp.....	110
11.5.6 camera.hpp.....	110
11.5.7 camera.cpp.....	111
11.5.8 cl_device.hpp.....	113
11.5.9 cl_device.cpp.....	113
11.5.10 clkernelmanager.hpp.....	114
11.5.11 clkernelmanager.cpp.....	115
11.5.12 cl_platform.hpp.....	120
11.5.13 cl_platform.cpp.....	120
11.5.14 config.hpp.....	121
11.5.15 configloader.hpp.....	122
11.5.16 configloader.cpp.....	123
11.5.17 deferredshader.hpp.....	126
11.5.18 deferredshader.cpp.....	127
11.5.19 hybridshader.hpp.....	130
11.5.20 hybridshader.cpp.....	131
11.5.21 hybridshadercpu.hpp.....	135
11.5.22 hybridshadercpu.cpp.....	137
11.5.23 input.hpp.....	143
11.5.24 input.cpp.....	144
11.5.25 main.cpp.....	146
11.5.26 material.hpp.....	147
11.5.27 mesh.hpp.....	148
11.5.28 mesh.cpp.....	149
11.5.29 model.hpp.....	151
11.5.30 model.cpp.....	152
11.5.31 modelloader.hpp.....	153
11.5.32 modelloader.cpp.....	154
11.5.33 pointlight.hpp.....	156
11.5.34 renderengine.hpp.....	156
11.5.35 renderenginecreator.hpp.....	157
11.5.36 shaderloader.hpp.....	157
11.5.37 shaderloader.cpp.....	158
11.5.38 shaderprogram.hpp.....	159
11.5.39 shaderprogram.cpp.....	159
11.5.40 system.hpp.....	160

11.5.41	texture.hpp.....	162
11.5.42	textureloader.hpp.....	162
11.5.43	textureloader.cpp.....	163
11.5.44	texturemanager.hpp.....	164
11.5.45	texturemanager.cpp.....	164
11.5.46	timer.hpp.....	165
11.5.47	triangle.hpp.....	166
11.5.48	vertex.hpp.....	166
11.5.49	window.hpp.....	167
11.5.50	window.cpp.....	168
11.5.51	world.hpp.....	169
11.5.52	world.cpp.....	170
11.5.53	kernels/render.cl.....	170
11.5.54	shaders/gbuffer.vert.....	174
11.5.55	shaders/gbuffer.frag.....	174
11.5.56	shaders/lighting.vert.....	175
11.5.57	shaders/lighting.frag.....	175
11.5.58	CMakeLists.txt.....	176

Índice de figuras

Figura 1: Trazado de Rayos [HEN000].....	11
Figura 2: Captura del juego Driller (1987).....	12
Figura 3: Imagen generada con Mitsuba.....	15
Figura 4: A la derecha, actor generado digitalmente en la película «Logan».....	16
Figura 5: Imagen del cortometraje generado en tiempo real con Unity «Adam».....	17
Figura 6: Imagen generada en tiempo real con «Brigade».....	18
Figura 7: Transformaciones homogéneas de la geometría de la escena.....	25
Figura 8: Rellenado del mapa de bits de un triángulo proyectado en el espacio pantalla.	26
Figura 9: Tubería de renderizado, del libro OpenGL Superbible, 6th ed.....	27
Figura 10: Flujo de datos en sombreado diferido.....	29
Figura 11: Imagen generada con trazado de rayos: «Astro», de Pratik Solanki.....	31
Figura 12: Flujo de datos en la técnica híbrida.....	33
Figura 13: BVH. Imagen extraída de Physically Based Rendering.....	34
Figura 14: Imagen generada mediante la técnica de rasterización sombreado directo....	37
Figura 15: Diagrama de clases simplificado de la implementación del sombreado directo	39
Figura 16: Ejemplo de G-Buffer.....	41
Figura 17: Diagrama de clases del proceso de creación de la clase RenderEngine.....	43
Figura 18: Imagen generada con la técnica híbrida.....	46
Figura 19: Captura de la herramienta CodeXL inspeccionando el kernel "mínimo".....	49
Figura 20: Representación gráfica de la comparativa inicial de rendimiento entre técnica híbrida y sombreado diferido.....	50
Figura 21: Representación del tiempo de ejecución del kernel según el número de puntos de luz.....	51
Figura 22: Captura de CodeXL de un kernel limitado por VGPR.....	51
Figura 23: Captura de CodeXL. Ocupación estimada al 80%.....	55
Figura 24: Diagrama UML de clases completo.....	65
Figura 25: Gráfica de comparación de rendimiento de diferentes técnicas.....	70
Figura 26: Imagen trazada usando «Cycles Renderer».....	71
Figura 27: Imagen generada mediante la técnica híbrida.....	72
Figura 28: Imagen generada con sombreado diferido.....	73
Figura 29: Resaltado de detalles.....	74
Figura 30: Detalles de la proyección de sombras.....	74
Figura 31: Detalle de la iluminación indirecta.....	75
Figura 32: Bump mapping.....	75
Figura 33: Diagrama de Gantt del primer período de desarrollo.....	81
Figura 34: Diagrama de Gantt del segundo período de desarrollo.....	81
Figura 35: Modelo de Plataforma de OpenCL (de Khronos, 2011).....	91
Figura 36: Ejecución paralela de datos 2D en OpenCL.....	92
Figura 37: Modelo de memoria de OpenCL (Khronos, 2011).....	95
Figura 38: Control de concurrencia con event-queueing de OpenCL.....	95

Índice de tablas

Tabla 1: Tamaño de las texturas del G-Buffer.....	42
Tabla 2: Comparativa inicial de rendimiento entre técnica híbrida y sombreado diferido..	50
Tabla 3: Tiempo de ejecución del kernel según el número de puntos de luz.....	51
Tabla 4: Comparación de rendimiento en fps antes y después de optimizar el kernel.....	57
Tabla 5: Rendimiento de la estructura BVH.....	61
Tabla 6: Millones de rayos por segundo evaluados usando BVH.....	62
Tabla 7: Comparación rendimiento ejecución en GPU con ejecución en CPU.....	64
Tabla 8: Conjunto de mediciones de las diferentes técnicas.....	68
Tabla 9: Comparación fps técnica híbrida y sombreado diferido.....	69
Tabla 10: Coste desglosado del ordenador utilizado.....	79
Tabla 11: Tareas realizadas, fechas y tiempo invertido.....	80



ETS de
Ingeniería
Informática

1 Introducción y objetivos del proyecto

1.1 Evolución histórica reciente

Se denomina CGI (Computer generated imagery) a aplicación de los ordenadores para la generación de imágenes. Una de sus modalidades es la generación de imágenes realistas a partir de modelos tridimensionales, que tiene como uno de los primeros y más emblemáticos ejemplos la tetera de Utah, creada por Martin Newell en 1975.

Después de algo más de cuatro décadas e innumerables avances en el campo, hay dos técnicas que han destacado y que se han convertido en las más utilizadas: trazado de rayos y rasterización.

El **trazado de rayos** trata de reproducir el proceso físico por el cual la luz viaja desde las fuentes de luz hasta el ojo que está captando la imagen. Es un proceso que genera imágenes de mucha calidad y muy realistas, pero es computacionalmente muy costoso.

El trazado de rayos se utiliza, por ejemplo, para producir los fotogramas en películas de cine de animación. La generación de cada una de estas imágenes puede llegar a necesitar varias horas de cálculos en granjas de servidores dedicados.

La segunda técnica, **rasterización**, proyecta la geometría descrita a base de vectores sobre un “espacio-pantalla” para más tarde llenar punto a punto el contenido de estos polígonos proyectados. Por ese motivo también se le conoce con el nombre de proyección. La rasterización produce imágenes mucho menos realistas que el trazado de rayos, pero permite la generación de varias imágenes por segundo, por lo que se lleva usando en videojuegos al menos desde finales de los años 80.

Con el paso del tiempo la técnica de la rasterización se ha ido mejorando para conseguir aún más velocidad y un mayor realismo, utilizando diferentes técnicas para

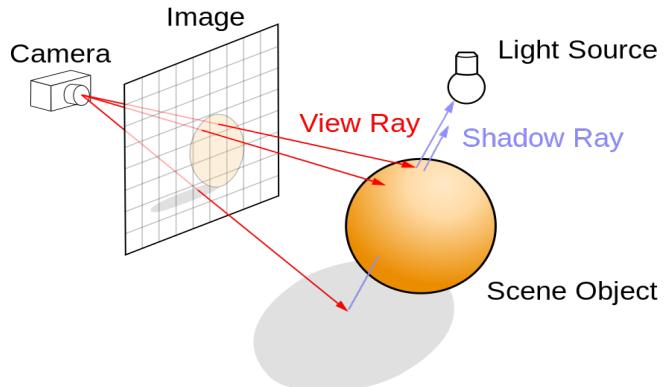


Figura 1: Trazado de Rayos [HEN000]

simular propiedades de la luz y los materiales. También ha aparecido *hardware* diseñado específicamente para acelerar estos cálculos: a principio de los 90 aparecieron las primeras consolas con *hardware* dedicado a la aceleración de los cálculos 3D y en 1997 apareció la primera tarjeta dedicada aceleradora de gráficos 3D para PC. [FUJ00]

La proliferación de este tipo de *hardware*, que en un principio se diseñó para acelerar rutinas gráficas, abrió nuevas posibilidades, naciendo en la primera década de 2000 lo que se denominó General-purpose computing on Graphics Processing Units (GPGPU), que consiste en utilizar las tarjetas gráficas para realizar cálculos de propósito general. El primer experimento, una rutina de multiplicación de matrices, se realizó en 2001 y en 2005 se desarrolló una de las primeras rutinas que corría más rápido en tarjetas gráficas que en CPUs, una descomposición matricial [PLC00].

Para aprovechar esta posibilidad se han desarrollado interfaces de programación de aplicaciones, siendo los dos ejemplos más conocidos CUDA(2007), y OpenCL(2009). Estas APIs han hecho posible que el uso de estas técnicas se extienda masivamente y hoy en día tanto el hardware como el software que lo soporta está disponible en la mayoría de los ordenadores modernos.

1.2 Situación actual y objetivo del siguiente trabajo

Como resultado de la evolución reciente, disponemos de una tremenda potencia de cálculo en las GPU y la posibilidad de usarlas para acelerar la producción de gráficos rasterizados o realizar cálculos de propósito general.

Aunque hay experimentos de trazado de rayos en tiempo real, la capacidad de cómputo es todavía insuficiente para que la calidad sea adecuada, apreciándose efectos de ruido o niebla muy acusados en modelos de cierta complejidad.

Dado que una implementación de trazado de rayos completa en tiempo real todavía no es posible en la práctica, queda la opción de intentar mejorar la calidad de los gráficos

generados mediante rasterización integrando el trazado de rayos para añadir ciertos efectos concretos.

El objetivo principal del presente proyecto es realizar una implementación de esta técnica híbrida, utilizando la computación paralela mediante OpenCL en una GPU y compararla con cada una de las dos técnicas clásicas por separado.

También son objetivo del mismo el proceso de aprendizaje y adquisición de conocimientos sobre los lenguajes y APIs C++, OpenGL, OpenCL y el necesario estudio de las técnicas de rasterización y trazado de rayos, así como una revisión de sus implementaciones más modernas.

2 Estado del arte

La generación de imágenes por ordenador es un área en constante y rápida evolución. En un intento de clasificar los diferentes productos existentes actualmente, podríamos separarlos en tres categorías:

1. Trazadores de rayos *off-line*. Son los programas cuyo objetivo es conseguir imágenes o animaciones de mucha calidad utilizando el trazado de rayos. Suelen necesitar de minutos a horas para la generación de cada imagen. En los últimos años un número cada vez mayor de estos programas utiliza GPGPU para acelerar los cálculos y hay incluso ejemplos de renderizado usando trazado de rayos en tiempo real, aunque el resultado es de menos calidad y «ruidoso» [LUXR000] al mover la cámara hasta que la imagen converge. Como ejemplos de esta categoría se encuentran Cycles Renderer [CYCL000], Octane Render [OCTA000], LuxRender [LUXR001], Mitsuba [MITS000], V-Ray [VRAY000], Arnold [ARNO000] y RenderMan [REND000].



Figura 3: Imagen generada con Mitsuba

Algunos de estos trazadores de rayos, como por ejemplo Arnold, se han usado para añadir objetos, personas o efectos a imágenes reales en películas. En estos casos el

reto no es producir una imagen completa que sea realista, sino integrar objetos sintéticos en imágenes reales sin que se aprecie la diferencia. Otra de sus aplicaciones es la de añadir actores generados por ordenador en películas que no son de animación. Por ejemplo, en la película «*Logan*», se utilizaron modelos digitales de varios de los actores para sustituir las caras de los actores de doblaje por otras generadas mediante trazado de rayos que imitaban a los actores reales. En la figura 4 se puede observar el resultado.



Figura 4: A la derecha, actor generado digitalmente en la película «*Logan*»

Si se desea profundizar más, se puede encontrar una comparación [CYCL001] entre diferentes trazadores de rayos generando imágenes de Blender [BLEN000], en la que se puede observar la calidad de las imágenes y el tiempo empleado, que fue desde los 7 minutos hasta los 30 en las escenas evaluadas.

2. Motores gráficos en tiempo real e interactivos basados en la rasterización, que suelen formar parte del código de motores de videojuegos. Mientras que en la categoría anterior se observa una tendencia a aumentar la velocidad, en este caso aumenta la calidad y el realismo de las imágenes producidas. Ejemplos actuales son Unity [UNIT000], Unreal Engine [UNRE000], Snow Drop [SNOW000], Unigine [UNIG000] y Cryengine [CRYE000].

En los últimos años estos motores han empezado a añadir efectos que hasta ahora eran exclusivos del trazado de rayos. Por ejemplo, desde finales de 2014 tanto Unity como Unreal Engine son capaces de generar imágenes en tiempo real con iluminación

difusa global para objetos estáticos utilizando Enlighten [ENLI000]. Además, pueden generar sombras y oclusión de ambiente mediante *light probes*. También muestran reflejos en «espacio de pantalla», lo que significa que son capaces de mostrar el reflejo de las partes de la escena que están siendo renderizadas. De esta forma, mediante diseños cuidadosamente elegidos, aproximaciones y datos precalculados, se puede conseguir renderizado basado en física de gran calidad. La figura 5 muestra un fotograma del cortometraje generado en tiempo real con Unity «Adam».

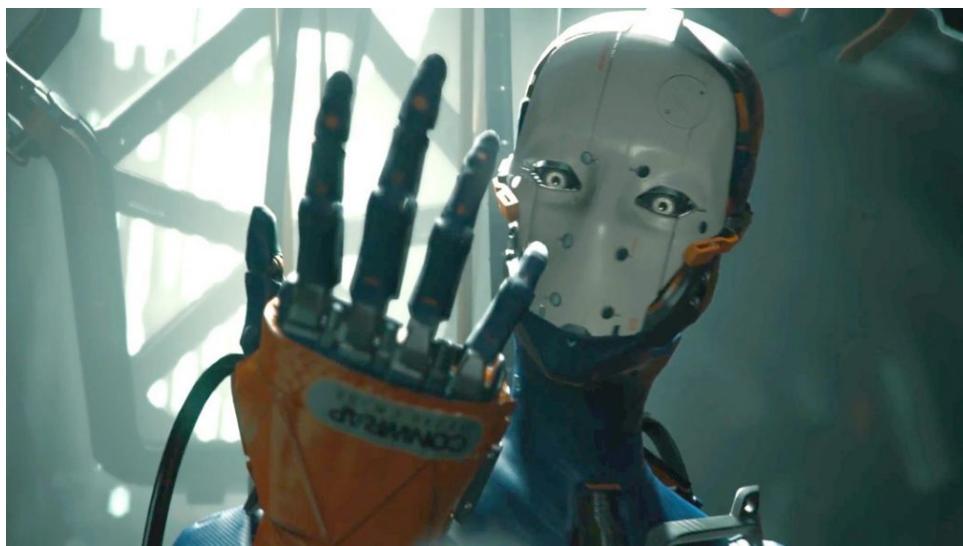


Figura 5: Imagen del cortometraje generado en tiempo real con Unity «Adam»

3. Motores gráficos en tiempo real e interactivos basados en el trazado de rayos.

En la figura 6 se observa una imagen generada en tiempo real con Brigade [BRIG000], uno de los motores más representativos. El principal escollo que se encuentra en esta técnica, mencionado ya en el punto 1, es el ruido que aparece en la imagen al mover la cámara. Cuando el movimiento para se mejora la imagen progresivamente al ir añadiendo más muestras por píxel. Hay avances muy prometedores y los resultados son cada vez mejores; para movimientos lentos se ha conseguido reducir en gran medida este efecto utilizando reprojeción de cámara y vectores de movimiento, como se explica en la conferencia [SWOB000]. En la misma conferencia se realiza un experimento utilizando «Map Bricks» en lugar de «Bounding Volume Hierarchies» para acelerar el cálculo del trazado de rayos en GPU, consiguiendo reflexiones, refracciones y oclusión ambiente en tiempo real.



Figura 6: Imagen generada en tiempo real con «Brigade»

3 Revisión tecnológica y criterios de elección

Para la consecución de los objetivos del presente proyecto, se ha realizado en primer lugar una revisión tecnológica de las herramientas software disponibles actuales que podemos encontrar en el mercado, con el propósito de conocer las opciones disponibles y las ventajas e inconvenientes de cada una de ellas.

A continuación se ha procedido a aplicar una serie de criterios para su elección, los cuales desarrollaremos en el apartado siguiente. De esta forma nos aseguramos una elección conveniente, actual y productiva de las tecnologías que se utilizarán en el desarrollo e implementación de las soluciones descritas en el presente proyecto.

3.1 Criterios de elección

Los criterios aplicados para la elección de una determinada tecnología han sido los siguientes:

- a) La tecnología debe estar establecida y probada en el campo que abarca este trabajo.
- b) Debe tener soporte y que su desarrollo siga activo.
- c) Siempre que sea posible, que sea una tecnología abierta y portable. Idealmente, software libre.
- d) Debe ser de fácil obtención e instalación.

El motivo de adoptar estos criterios es que el carácter experimental del trabajo resida en la combinación de las técnicas de rasterización y trazado de rayos usando OpenCL, evitando en lo posible contratiempos añadidos.

Un criterio añadido es mantener en un mínimo razonable el número de dependencias del proyecto, siempre que no implique un tiempo excesivo para codificar alguna funcionalidad ampliamente disponible y que no está entre los objetivos del proyecto.

3.2 Herramientas y entorno de desarrollo

Para el proceso de desarrollo del presente proyecto he optado por utilizar un ordenador con la distribución Arch Linux del sistema operativo GNU/Linux, puesto que es un entorno de software libre en el que se encuentran disponibles y de fácil instalación las herramientas necesarias. Además, para facilitar la portabilidad, sólo he usado herramientas y bibliotecas multiplataforma, disponibles al menos también para los sistemas operativos Microsoft Windows y Mac OS X.

Las versiones utilizadas de cada herramienta han sido las últimas disponibles en la distribución Arch Linux en cada momento del desarrollo del proyecto. Indico en la presente memoria las últimas utilizadas. Para instalar cada uno de los componentes he usado el gestor de paquetes de la distribución, «pacman» (Package Manager). La instalación de cualquiera de las herramientas o bibliotecas ha sido tan simple como ejecutar: «`pacman -S nombre_de_paquete`». Por ejemplo, para instalar el compilador Clang, basta con teclear «`pacman -S clang`».

El lenguaje de programación utilizado para la parte del programa ejecutable en CPU ha sido C++ en su último estándar publicado al inicio del desarrollo, C++14. He elegido este lenguaje por ser el de uso habitual para interactuar con OpenGL y OpenCL.

He optado por utilizar los compiladores GCC 6.3.1 [GCC000] y Clang 3.9.1 [CLG000], dado que se encuentran disponibles en múltiples plataformas e implementan los últimos estándares de C++. He alternado regularmente entre ambos para asegurarme de que el código sea C++ estándar, no dependiente de un compilador concreto.

Durante el desarrollo del proyecto uso las herramientas de Clang «`clang-format`», para formatear de forma consistente y legible el código, y «`clang-tidy`» para realizar comprobaciones de posibles fallos.

Para gestionar el proceso de construcción (configuración, detección de dependencias, compilación y enlazado) he elegido CMake 3.7.2 [CMA000], por ser independiente del compilador y de la plataforma.

Como herramienta de control de versiones de código fuente he elegido GIT 2.12 [GIT000], puesto que sigue en desarrollo activo y permite llevar un control sencillo de los cambios en el proyecto.

Para la edición del código fuente he usado alternativamente el editor VIM 8.0 [VIM000] y Qt Creator 4.2.1 [QTC000], siendo en este caso la única motivación para la elección una cuestión de gustos personales.

Los modelos utilizados para probar las diferentes técnicas en este proyecto son de libre distribución y han sido obtenidos en la página web de McGuire Graphics Data [MAC000].

El equipamiento utilizado tanto para el desarrollo como para las pruebas de rendimiento ha sido el siguiente:

- CPU: Intel Core i5-4670 (3.4 GHz, 4 núcleos, caché L1 256kB, L2 1MB, L3 6MB)
- Placa base: Gigabyte GA-H87-HD3
- Memoria RAM: 2x8GB Kingston HyperX blu 1600MHz
- Tarjeta gráfica: Sapphire Dual-X R9 285 2GB GDDR5
- Disco duro: Samsung EVO 840 SSD 120GB

3.3 OpenGL

Open Graphics Library [OGL000] es una especificación estándar que define un API para renderizado de gráficos en dos y tres dimensiones. OpenGL fue creado por Silicon Graphics Inc en 1991 y actualmente lo mantiene el grupo Khronos.

En el momento de redactar la presente documentación la última versión es la 4.5, aunque el programa desarrollado funciona con cualquier versión a partir de la 3.3, que está disponible desde 2010.

He realizado la implementación de las técnicas de rasterización en OpenGL, tanto el sombreado directo como el diferido.

Para el estudio de OpenGL he consultado los libros «OpenGL Programming Guide, 8th edition» [SSKL0] y «OpenGL Superbible, 6th edition» [SWH00].

También he seguido el tutorial de Internet «Learn OpenGL» [LOGL00] y he realizado el curso de EDX “UC San DiegoX: CSE167x Computer Graphics” [UCED00]. Este curso es on-line y gratuito, tiene una duración de seis semanas y en él se explican los fundamentos de los gráficos por ordenador.

3.4 OpenCL

Open Computing Language [OCL000] proporciona un interfaz estándar para el desarrollo de aplicaciones paralelas en entornos heterogéneos. La primera versión es de 2009 y se ha convertido en un estándar desarrollado por el grupo Khronos.

En el presente proyecto uso OpenCL para implementar la segunda parte de la técnica híbrida, relativa al trazado de rayos, y ejecutar de forma paralela los algoritmos en la tarjeta gráfica.

Aunque la última versión estable es la 2.1, para construir y ejecutar el programa es suficiente con la versión 1.2, de 2011. También es necesario que la plataforma que ejecute la aplicación disponga de la extensión «cl_khr_gl_sharing» [CLK000].

Para estudiar OpenCL me he basado en los libros «Heterogeneous Computing with OpenCL 2.0» [KMS000] y «OpenCL Programming by Example» [BBH000].

3.5 OpenMP

Open Multi Processing [OMP000] es un interfaz de programación de aplicaciones que proporciona una forma portable de desarrollar aplicaciones paralelas en múltiples lenguajes y arquitecturas. Nació en 1997 para el lenguaje Fortran y actualmente es desarrollado por OpenMP Architecture Review Board.

El uso de OpenMP en el presente proyecto se ha limitado a ejecutar de forma paralela el bucle que recorre los diferentes puntos de las texturas del G-Buffer en la implementación en C++ de la segunda parte de la técnica híbrida.

Para instalar esta tecnología ha sido suficiente instalar la biblioteca «`openmp`» y configurar CMake para que la use, ya que tanto el compilador Clang como GCC lo soportan en las versiones utilizadas.

3.6 Bibliotecas de terceros

GLM versión 0.9.8.4 [GLM000], OpenGL Mathematics es una biblioteca para C++ contenida en archivos de cabecera basada en el lenguaje de programación GLSL (OpenGL Shading Language). Es portable, soporta una gran variedad de compiladores y proporciona clases y métodos similares a los del lenguaje GLSL con tipos de datos directamente compatibles con OpenGL.

GLFW versión 3.2.1 [GLFW00], es una biblioteca para crear y gestionar ventanas y contextos OpenGL. También es un interfaz con la entrada de teclado, ratón y joystick.

GLEW versión 2.0.0 [GLEW00], OpenGL Extension Wrangler Library, es una biblioteca multiplataforma C++ que proporciona un método conveniente de interactuar con las extensiones de OpenGL.

ASSIMP versión 3.3.1 [ASSI00], Open Asset Import Library, proporciona la capacidad de leer una gran variedad de formatos de archivos de descripción de modelos 3D.

FreeImage versión 3.17.0 [FRE000], es una pequeña biblioteca de código abierto que permite leer los formatos de imagen más populares.

3.7 Revisión de C++

El lenguaje de programación C++ ha sufrido cambios importantes en los últimos años. Con la aprobación de los estándares C++11 y C++14 se ha hecho posible un uso más seguro, sencillo y eficiente del lenguaje.

Como referencia de C++ he usado los libros «A Tour of C++» [STR00], «Effective Modern C++» [MEY000], las conferencias [SUT000], [STR001], [STR002], [STR003] y [MEY001] y el artículo de la Wikipedia sobre NRVO (Name Value Return Optimization) y Copy Elision [WIK000].

Después de estudiar todos estos recursos y hacer pruebas con dos compiladores he llegado a las siguientes conclusiones:

1. Usar RAI (Resource Acquisition Is Initialization). Siempre que se tenga que reservar un recurso que deba ser liberado, he creado una clase que adquiere el recurso en el constructor y lo libera en el destructor. Por ejemplo en el manejo de ficheros para cargar texturas o reservas de memoria.
2. Devolver el resultado de funciones por valor. Por ejemplo al crear un vector de vértices en ModelLoader::LoadMeshVertices y devolverlo por valor se produce «Copy Elision» y no tiene ningún coste en cuanto a rendimiento y el código es muy claro.
3. Utilizar la semántica de movimiento para poder pasar valores a los constructores de forma sencilla y eficiente. Por ejemplo, en el caso del constructor de Mesh, los vectores de vértices, normales, etc. se mueven a las variables miembro y así se pueden construir desde una clase factoría de Mesh que carga el modelo usando ASSIMP. De esta forma se libera la clase Mesh de código específico de carga de modelos.

4. Pasar valores a los métodos como «const &» o «const *». No usar new ni delete.
5. En caso de ser necesario, usar unique_ptr, shared_ptr o vectores.

Para conseguir un buen rendimiento tanto en CPU como en tarjetas gráficas actuales hay que tener muy en cuenta el uso de la memoria caché. Sobre este tema he encontrado interesante el artículo [LWN000] y las conferencias [CAR000], [ACT000] y [MEY002].

Como resumen del contenido de los últimos cuatro recursos mencionados, se podría decir que los algoritmos y estructuras de datos deben aprovechar la memoria caché del procesador. Si las estructuras de datos son pequeñas, compactas y lineales y el algoritmo es sencillo y cabe en la caché de instrucciones el rendimiento es órdenes de magnitudes superior al caso contrario. Si se realiza una lectura de datos y se procesa de forma lineal y predecible se aprovecha al máximo la memoria caché y el «prefetching». Por lo tanto, procuraré usar estructuras sencillas y vectores en lugar de listas enlazadas, y elegir algoritmos que las recorran linealmente y de forma predecible los datos. No profundizaré más en este asunto, ya que micro-optimizaciones y un estudio más exhaustivo de rendimiento quedan fuera del objetivo y alcance de este proyecto.

4 Técnicas de renderizado y métodos de aceleración

4.1 Rasterización

La rasterización es una técnica que se basa en la proyección de polígonos situados en un espacio tridimensional sobre el espacio bidimensional de la pantalla y en el posterior recorrido de esa superficie bidimensional proyectada para rellenarla con texturas o colores. La escena que describe la imagen a generar está descrita a base de primitivas en ese espacio tridimensional, generalmente mallas de triángulos.

En la primera fase de la rasterización se toma como entrada los vértices que componen la geometría de la escena y se les aplica las transformaciones necesarias para situarlos en sus posiciones finales en el espacio tridimensional según las tres matrices de transformación definidas: modelo, vista y proyección. Cada una de estas matrices define una posible traslación, rotación y escalado en coordenadas homogéneas. La matriz «*modelo*» sitúa cada modelo en su posición en la escena. Esta posición se conoce como *coordenadas en el mundo*. Después se aplica la transformación «*vista*», que orienta la escena para adecuarla a la posición del punto de vista del ojo del espectador o la cámara. Estas son las llamadas *coordenadas visuales*. Finalmente, mediante la «*matriz de proyección*» se escala y se aplica la perspectiva deseada.

Estas transformaciones matriciales se realizan el el «*vertex shader*» que y las coordenadas finales se guardan en la variable predefinida de OpenGL «*gl_position*»

A continuación se explican estas operaciones matriciales:

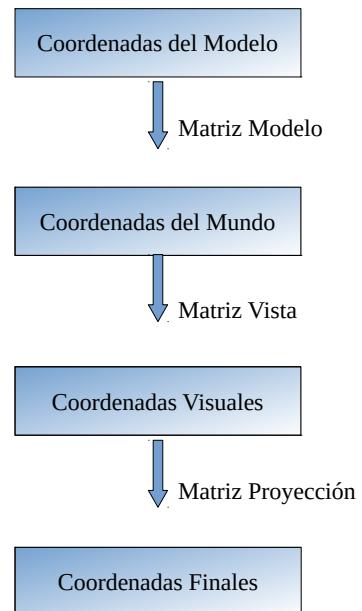


Figura 7: Transformaciones homogéneas de la geometría de la escena

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = M_{\text{Proyección}} \times M_{\text{Vista}} \times M_{\text{Modelo}} \times \begin{bmatrix} x_{\text{modelo}} \\ y_{\text{modelo}} \\ z_{\text{modelo}} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{bmatrix} \times \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \\ v_{31} & v_{32} & v_{33} & v_{34} \\ v_{41} & v_{42} & v_{43} & v_{44} \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \times \begin{bmatrix} x_{\text{modelo}} \\ y_{\text{modelo}} \\ z_{\text{modelo}} \\ 1 \end{bmatrix}$$

La segunda fase consiste en proyectar esos triángulos resultantes, todavía en un espacio tridimensional, sobre el plano de la pantalla, descartando los que caen fuera de la misma y los que no son visibles por estar detrás de otros.

Finalmente se rellena el contenido de los triángulos ya proyectados usando texturas o colores planos. Los vértices de los triángulos definen sus coordenadas en una textura asignada al objeto del que forman parte, de forma que el color de cada punto interior del triángulo se obtiene interpolando las coordenadas en las texturas de los tres vértices.

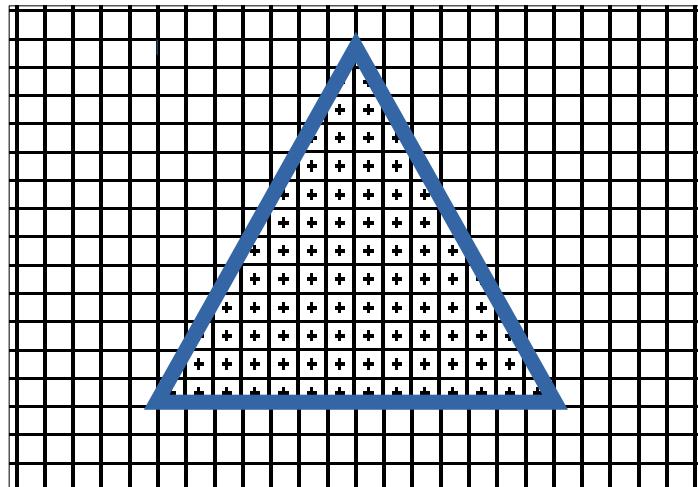


Figura 8: Rellenado del mapa de bits de un triángulo proyectado en el espacio pantalla.

4.1.1 Sombreado directo

La técnica de sombreado directo, o «forward shading» en inglés, es la forma más básica de generar imágenes mediante rasterización con OpenGL. La imagen final se genera haciendo pasar los datos de entrada por una serie de transformaciones, aplicadas por los sombreadores o «shaders», en las que parte o toda la entrada de datos corresponde a la salida del paso anterior, lo que constituye la llamada tubería de renderizado (ver figura 9). Los datos de entrada inicialmente están compuestos por información sobre la geometría del modelo a mostrar, pudiendo definirse una malla de triángulos por sus vértices, normales y coordenadas de esos vértices en la textura correspondiente a ese material. Aunque esta tubería de renderizado puede estar compuesta por varios pasos, de los cuales actualmente en OpenGL hasta cinco son programables, únicamente es obligatorio definir por el programador dos de ellos, pudiendo dejarse los demás en sus valores por defecto.

Estos dos pasos o shaders programables que hay que definir obligatoriamente son el «vertex shader» y el «fragment shader» y son los que se utilizan en este trabajo.

El «vertex shader» corresponde a la primera fase de la rasterización descrita en el apartado anterior, que sitúa un vértice en su posición final en el espacio tridimensional.

El segundo, el «fragment shader», se encarga de definir el color de salida de cada punto de la imagen final generada, tomando como entrada los puntos interiores de los triángulos proyectados sobre el espacio bidimensional de la pantalla. Se rellena el mapa de bits que compone la imagen a partir de los datos vectoriales proyectados. Es en este paso también donde se evalúa la información de iluminación.

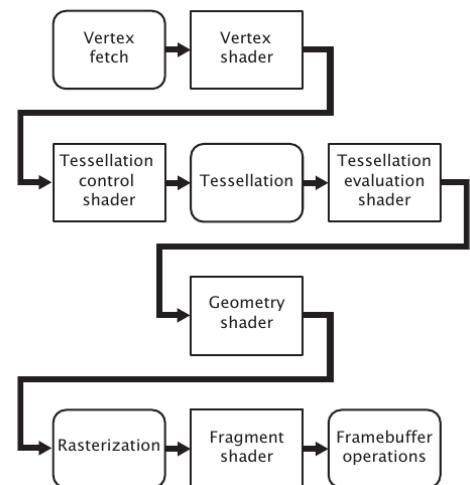


Figura 9: Tubería de renderizado, del libro OpenGL Superbible, 6th ed.

4.1.2 Sombreado diferido

La técnica de sombreado diferido, «deferred shading» en inglés, es una técnica de sombreado en dos pasadas. El sombreado directo, descrito en el apartado anterior, constaba de varias fases, pero genera la imagen de una sola pasada. Por el contrario, en el sombreado diferido, son necesarias dos pasadas, cada una de éstas compuesta por las fases que sean necesarias.

En la primera pasada del sombreado diferido se procesan datos de la geometría de la escena, sin tener en cuenta la iluminación, y se almacena el resultado en una colección de reservas de memoria en forma de texturas denominada G-Buffer (*Graphics Buffer*). Las texturas del G-Buffer están referidas al espacio bidimensional de la pantalla y cada píxel almacena información sobre un píxel en pantalla. Se suele utilizar una textura para posiciones, otra para normales, otra para «albedo», componente especular y cualquier otro dato necesario. El conjunto de estas texturas compone el G-Buffer.

En una segunda pasada se utiliza el G-Buffer junto con información sobre la iluminación para generar la imagen con la iluminación deseada.

Este proceso está descrito en la figura 10, y en la figura 16 se puede observar un ejemplo de G-Buffer utilizado en el presente proyecto.

La ventaja por la que habitualmente se usa esta técnica es que, al retrasar la computación de la iluminación hasta una segunda pasada, se evita tener que repetir todos los cálculos de la primera pasada por cada fuente de luz. Los datos de normales, posiciones y materiales se obtienen una única vez en la primera pasada y se guardan para que se puedan reutilizar en la segunda. El aumento de rendimiento es notable conforme se va incrementando el número de fuentes de luz.

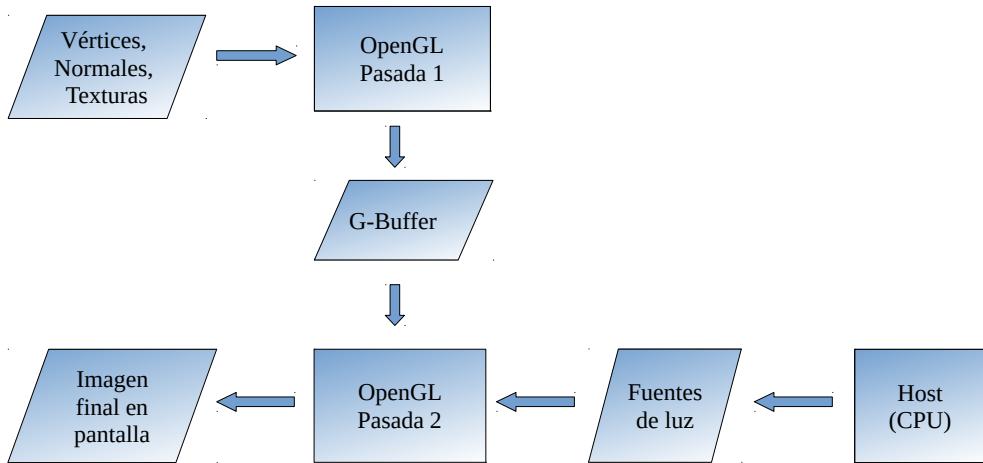


Figura 10: Flujo de datos en sombreado diferido

4.2 Trazado de rayos

Según [SUF00] el trazado de rayos es «una técnica de que genera imágenes por ordenador disparando rayos». La misma fuente da una descripción más detallada, indicando que «los rayos atraviesan los píxeles de la pantalla y se comprueba su intersección con los objetos que conforman la escena. Cuando un rayo choca con un objeto, el programa averigua cuánta luz es reflejada de vuelta en la dirección del rayo para determinar el color del píxel. Utilizando suficientes píxeles, el programa trazador de rayos genera una imagen de los objetos¹». Se puede observar este proceso en la figura 1 (página 11), y un ejemplo de imagen resultante en la figura 11.

Para generar una imagen el programa trazador de rayos utiliza diferentes tipos de rayos. Los que describe el párrafo anterior se denominan rayos primarios y, como indicaba anteriormente, son los que van desde la cámara a la escena pasando por cada uno de los píxeles que componen la imagen. Si sólo se utilizan rayos primarios se obtiene una imagen muy similar a la que se obtendría mediante rasterización.

Para tener en cuenta otros aportes de luz en el cálculo del color de un píxel se utilizan rayos secundarios. Los rayos de sombras son un tipo de rayos secundarios que se lanzan desde el punto en el que el rayo primario choca con un objeto hasta los diferentes puntos de luz que lo iluminan. Si un rayo de sombra choca con otro objeto

¹ Traducción de la versión en inglés de ambas citas realizada por el autor de este proyecto.

de la geometría de la escena en su camino hacia el punto de luz, significará que esa fuente de luz no aportará iluminación a ese objeto en ese punto. Se puede generalizar para superficies emisoras de luz muestreando la superficie emisora con múltiples rayos. De esta forma pueden aparecer zonas de umbra y penumbra en diferentes grados.

Hay otros tipos de rayos secundarios como, por ejemplo, los dedicados a generar iluminación global. La iluminación global es la que se produce cuando algún objeto de la escena, que no es una fuente de luz, contribuye a la iluminación de otro al reflejar parte de la luz que recibe. Las habitaciones blancas son más luminosas que las negras porque las paredes y techo reflejan parte de la energía luminosa que reciben y contribuyen a iluminar el resto de la habitación. Los rayos secundarios pueden llegar a generarse recursivamente, siendo necesario limitar la profundidad de esta recursividad de alguna forma, ya sea fijando un número máximo de rebotes o bien descartando continuar profundizando a partir de un umbral de energía aportada. Otros rayos secundarios sirven para incorporar reflejos o refracciones, entre otros efectos.

Todos estos aportes al color que debe tener un píxel están definidos en la ecuación de renderizado o de transporte de luz, que expresa el equilibrio de energía radiada en la escena. La ecuación de renderizado, sin embargo, es tremadamente compleja y debe resolverse mediante aproximaciones. Una de las aproximaciones más sencilla únicamente tiene en cuenta la iluminación directa, simulando la indirecta mediante un componente de luz difusa constante en toda la escena. Cuanto más fielmente aproxime el programa de trazado de rayos sus cálculos a la ecuación de renderizado, más realista será la imagen producida.

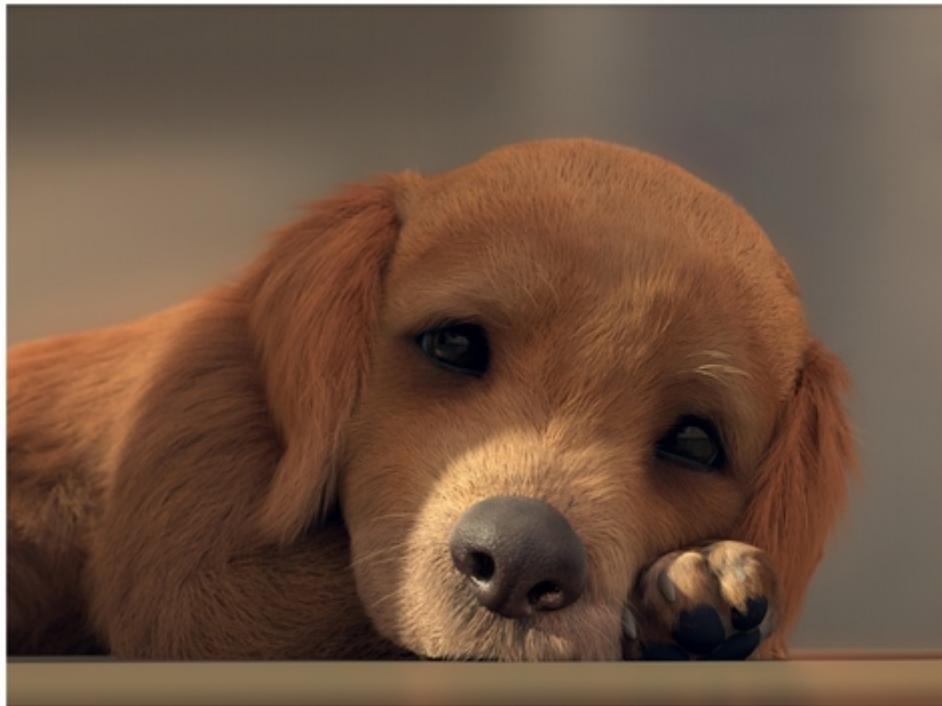


Figura 11: Imagen generada con trazado de rayos: «Astro», de Pratik Solanki

El trazado de rayos sirve de base para diferentes técnicas, como pueden ser radiosidad, *path tracing* unidireccional, *path tracing* bidireccional o mapeado de fotones.

Como se ha explicado con anterioridad, esta técnica permite generar imágenes muy realistas, pero posee el inconveniente de tener un gran coste computacional. Una valoración cuantificada de este coste según el efecto que se quiera reproducir se puede encontrar en [SWOB000]. Esta cuantificación, aunque subjetiva en alguno de sus valores, da una idea del número de rayos por píxel (rpp) que hay que generar para obtener una imagen de una calidad aceptable:

- Rayos primarios: 1 rpp
- Sombras no suavizadas: 1 rpp por punto de luz
- Sombras suaves: 10-20 rpp por fuente de luz
- Reflexiones / refracciones: 1 rpp por rebote
- Oclusión ambiental: 100 rpp
- Path Tracing unidireccional: 500-1000 rpp

Algunos de estos efectos se pueden simular mediante rasterización, pero el resultado no es tan bueno o bien sólo es aplicable en determinadas condiciones y las técnicas utilizadas son mucho más complejas y menos elegantes que sus equivalentes en trazado de rayos.

4.3 Técnica híbrida

La técnica híbrida analizada en el presente proyecto trata de ser un punto intermedio entre la rasterización y el trazado de rayos, tanto en calidad de imagen producida como en tiempo de cálculo invertido. El objetivo es obtener algunos efectos propios del trazado de rayos en imágenes generadas en tiempo real.

Para conseguir este objetivo se genera la imagen en dos pasadas. La primera pasada obtiene el equivalente al resultado obtenido por los rayos primarios en trazado de rayos, pero utilizando rasterización y OpenGL. Esta pasada es, de hecho, la primera pasada del sombreado diferido descrita en el capítulo anterior, que obtiene como producto de sus cálculos el G-Buffer.

En una segunda pasada se añaden otros aportes de la ecuación de renderizado, añadiendo otros tipos de rayos al resultado de la rasterización almacenado en el G-Buffer. En el presente proyecto únicamente se han añadido rayos de sombras, pero podrían añadirse rayos secundarios para calcular reflejos, refracciones e iluminación global. Esta segunda pasada se programa en OpenCL para aprovechar las capacidades de cálculo paralelo de las tarjetas gráficas actuales.

En la figura 12 se muestra una representación gráfica del flujo de datos entre los diferentes sistemas que participan en la técnica híbrida.

El resultado esperado es un aumento de la calidad de imagen generada con respecto a la rasterización, a cambio de un descenso en el número de imágenes generadas por segundo que no sea tan acusado como para evitar su uso en programas interactivos.

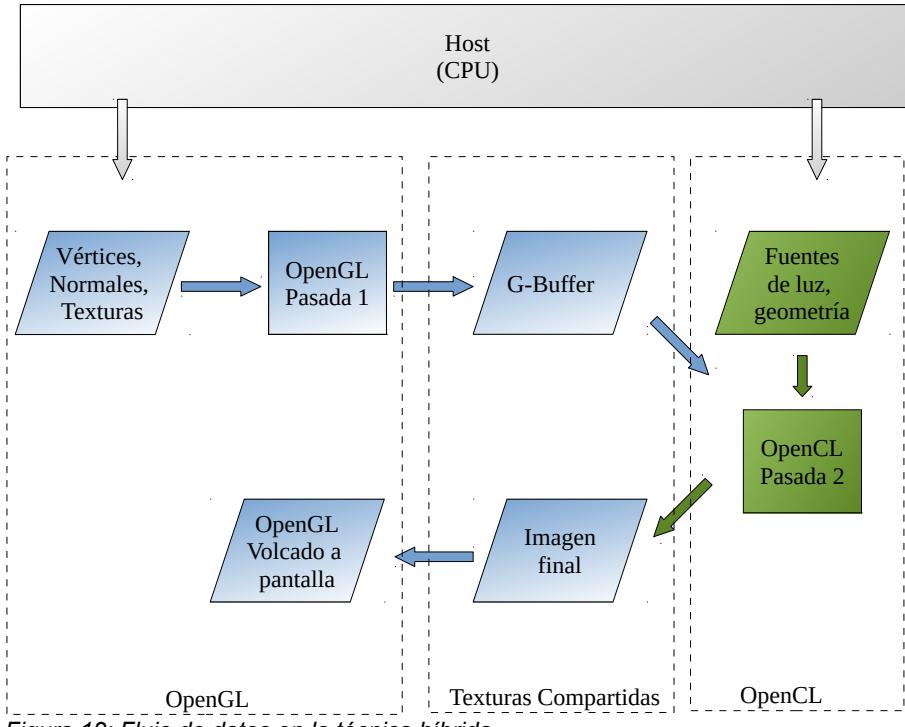


Figura 12: Flujo de datos en la técnica híbrida

4.4 Aceleración mediante BVH

El cálculo computacionalmente más costoso del trazado de rayos es la comprobación de intersecciones de los rayos generados con las primitivas que componen la geometría (en el caso del presente proyecto la geometría de la escena está compuesta únicamente por triángulos). Por lo tanto, es vital optimizar el tiempo invertido en estas comprobaciones.

Hay dos formas realizar esta optimización. La primera es reducir el tiempo empleado en cada comprobación de intersección rayo-tríangulo y la segunda es reducir el número de comprobaciones a realizar.

Sobre la primera aproximación hay varias publicaciones y estudios. Elijo una de las posibilidades por ser la más ampliamente utilizada y ser considerada de las más rápidas: el algoritmo de Möller y Trumbore «Fast, minimum storage ray/triangle intersection» [MT00] de 1997, tal y como está descrito en Physically Based Rendering [PHH00].

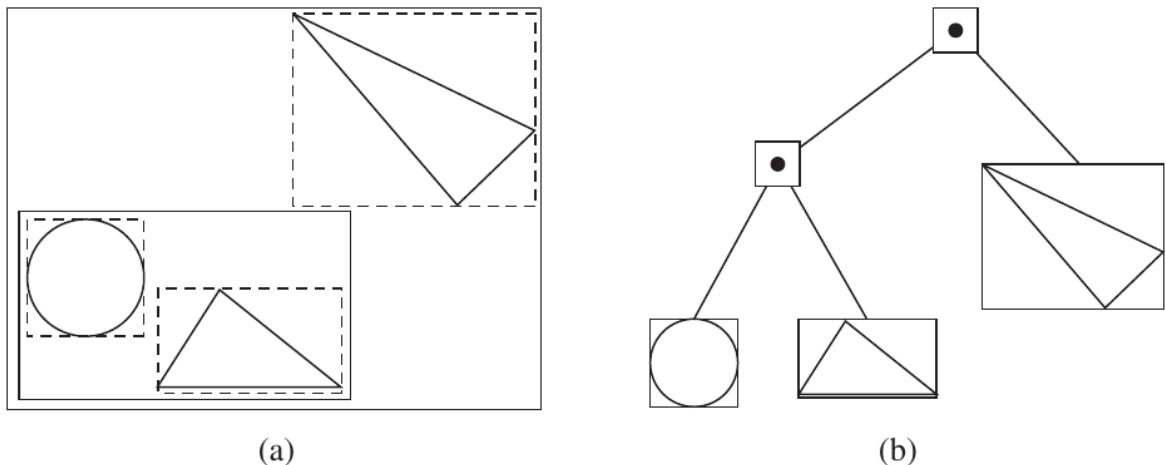


Figura 13: BVH. Imagen extraída de Physically Based Rendering
 a) Conjunto de primitivas, agregadas en jerarquías volúmenes envolventes. Nodos interiores en trazo continuo y nodos hoja en trazo discontinuo.
 b) Árbol BVH resultante.

La segunda aproximación pasa por descartar realizar comprobaciones de parte de la geometría. Una de las técnicas consideradas más útiles para el trazado de rayos es ordenar las primitivas en una jerarquía de volúmenes envolventes, BVH por sus siglas en inglés. En esta técnica se genera un árbol en el que los nodos interiores contienen a otros nodos y los nodos hoja contienen primitivas de la escena. La comprobación de la intersección se empieza desde la raíz del árbol, comprobando todos los nodos hijos. Cuando se comprueba que un rayo no choca con el volumen envolvente perteneciente a un nodo hijo se puede descartar toda la rama que cuelga de éste. Al llegar a un nodo hoja se comprueban las intersecciones con las primitivas que contiene.

5 Desarrollo

5.1 Estrategia de implementación

Para el desarrollo del presente proyecto he optado por utilizar un paradigma iterativo e incremental debido a sus ventajas, ya que la técnica híbrida a implementar se apoya en otras técnicas conocidas y probadas. De esta forma he podido ir construyendo el resultado final mediante una iteración de pasos sencillos, introduciendo cambios en la estructura del programa en las diferentes iteraciones. Al final de cada paso el resultado ha sido un programa capaz de cargar un modelo, mostrarlo en pantalla e interactuar con el usuario en tiempo real.

Las técnicas estudiadas en los capítulos 4.1.1 y 4.1.2, conocidas como sombreado directo y sombreado diferido son técnicas muy conocidas, sobre las que existe abundante literatura y ejemplos. Estas dos iteraciones no han supuesto un problema más allá de su comprensión y correcta implementación. La tercera iteración, la técnica híbrida descrita en el capítulo 4.3, es la que ha requerido más labor de investigación y experimentación.

5.2 Iteración 1. Sombreado directo

El primer paso del desarrollo ha consistido en implementar un programa capaz de generar imágenes en tiempo real a partir de un modelo de descripción de escenas 3D, según la técnica de sombreado directo descrita en el capítulo 4.1.1. El resultado es el esqueleto sobre el que se va a apoyar el trabajo final, ya que muchas de las clases que se definen aquí se reutilizarán en todos los pasos siguientes. Ejemplos de estas clases son la inicialización de OpenGL y la ventana principal, la cámara, la carga de modelos y texturas, el temporizador y la entrada del teclado.

En esta fase del proyecto he implementado el siguiente código en C++:

- Clase **Camera**, en los archivos **camera.cpp** y **camera.hpp**. Contiene los métodos para controlar la posición y orientación de la cámara y sus matrices asociadas: vista y proyección. También le he añadido métodos para que la clase que controla la entrada del usuario a través del teclado y ratón puedan moverla y orientarla. No he contemplado la opción de tener más de una cámara a la vez, aunque, si fuese necesario hacerlo, el cambio no sería complejo. Por simplicidad utilizo ángulos de

Euler XYZ, permitiendo cabeceo (pitch) que es una rotación sobre el eje Y, y es equivalente al gesto de girar la cabeza hacia los lados, y guiñada (yaw), rotación sobre el eje X, que equivale a subir o bajar la vista. También podría haber utilizado matrices homogéneas o cuaternios, pero los ángulos de Euler me parecían la solución conceptualmente más sencilla. Las matriz «*vista*» se obtiene a partir de los ángulos de Euler, y la matriz «*proyección*» a partir de los parámetros distancia más cercana (*zNear*), más lejana (*zFar*), y grados sexagesimales de amplitud del campo de visión vertical (Field of view Y: *fovy*). Ambas se calculan usando la biblioteca GLM.

- Clase **Input**, en los archivos `input.cpp` e `input.hpp`. Se encarga de obtener la entrada de teclado y ratón. Utilizo la API de GLFW. De esta forma me aseguro de que sea portable entre diferentes plataformas. Para intentar que quede lo más limpio posible utilizo funciones lambda para los callbacks de GLFW, evitando funciones en C que estén fuera de las clases definidas. Además, como necesito acceso a los métodos y objetos contenidos en la clase `Input`, utilizo las funciones de GLFW `glfwSetWindowUserPointer` y `glfwGetWindowUserPointer`, que permiten guardar un puntero de tipo «`void`» (sin tipo) para poder acceder a él más tarde. En este puntero guardo la dirección del objeto `Input`.
- Clase **Window**, en los archivos `window.cpp` y `window.hpp`. Utiliza las API de GLFW para inicializar la ventana en la que se mostrará la aplicación. También define varios parámetros de OpenGL, como la eliminación de caras ocultas o la espera del refresco de monitor.
- Clase **Model**, en los archivos `model.cpp` y `model.hpp`. Contiene un modelo a dibujar. Un modelo está compuesto por uno o varios objetos denominados `Mesh` (clase `Mesh`, definida en `mesh.cpp` y `mesh.hpp`), cada uno de los cuales es una malla de triángulos a la que se asocia un material y texturas. Para dibujar un modelo hay que dibujar todos sus mallas. La clase `Model` tiene un método denominado `draw`, que realiza una iteración llamando al método del mismo nombre de cada uno de los objetos `Mesh` que contiene. También contiene la matriz «`modelo`», para permitir situar el modelo en la posición de la escena necesaria. La elección de esta estructura imita la estructura que utiliza ASSIMP y facilita el intercambio de estructuras con OpenGL. En estos momentos no se contempla la opción de cargar y se mostrar más de un modelo a la vez.

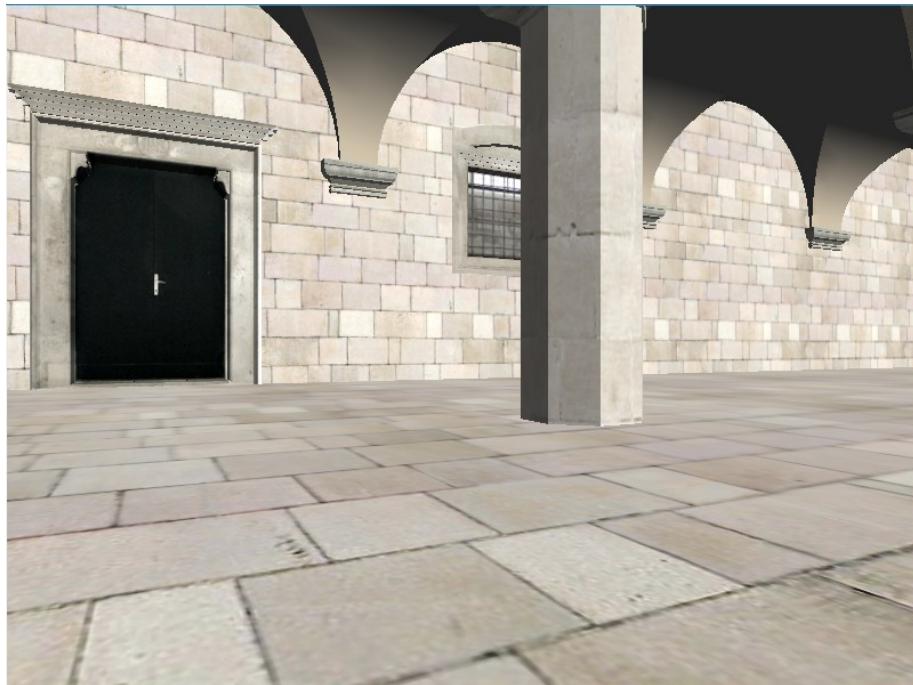


Figura 14: Imagen generada mediante la técnica de rasterización sombreado directo

- Clase **ModelLoader**, en los archivos **modelloader.cpp** y **modelloader.hpp**. Lectura de disco de los modelos a mostrar. La llamada a ModelLoader con un nombre de archivo devuelve un objeto de tipo Model ya cargado en memoria. Siempre que se cargue un objeto de disco he procurado separar su lectura de disco del objeto en sí. Así la lógica de funcionamiento y la de lectura de disco están separadas en archivos diferentes, lo que redunda en un código más legible.
- Clase **Texture**, en el archivo **texture.hpp**. Las texturas se almacenan en memoria de la tarjeta de vídeo, por lo que en la clase Texture únicamente se guarda el identificador único con el que OpenGL nombra a esa textura y el identificador del «uniform» en el shader. Sólo es necesario acceder a las texturas desde OpenGL, por lo que es suficiente con guardar estos datos.
- Clase **TextureLoader**, definida en los archivos **textureloader.cpp** y **textureloader.hpp** y clase **TextureManager**, definida en **texturemanager.cpp** y **texturemanager.hpp**. Lectura de disco de texturas. TextureLoader carga una imagen usando la biblioteca FreeImage. En caso de que la imagen no estuviese en formato RGBA se realiza en esta clase la transformación, ya que todas las texturas que se usan en este programa están en este formato. TextureManager implementa

una optimización para no cargar más de una vez cada textura. Es un caché de texturas. El motivo es que normalmente hay muchas mallas que tienen la misma textura asociada y es un desperdicio cargarla de forma duplicada.

- Clase **Material**, en el archivo `material.hpp`. Guarda las propiedades de un material, tales como reflejo difuso de luz , especular y texturas.
- Clase **World**, definida en `world.cpp` y `world.hpp`: Define el objeto «mundo», que contiene los modelos, luces, cámara y demás objetos que describen la escena.
- Función principal, **Main**, definida en `main.cpp`: Contiene la inicialización y el bucle principal del programa.
- Clase **Timer**, definida en `timer.hpp`. Define un temporizador. Obtiene la diferencia en milisegundos entre llamadas a la función update. Utilizo las API de GLFW, no las del sistema operativo para que el código sea portable.
- Clase **System**, definida en `system.hpp`: Encapsula todas las funciones relacionadas con el sistema. Contiene el objeto temporizador Timer, la ventana de la aplicación Window y la entrada del teclado y ratón Input.
- Clase **ShaderProgram**, definida en `shaderprogram.hpp` y `shaderprogram.cpp`. Contiene un programa GLSL, compuesto por un «vertex shader» y un «program shader». Proporciona los métodos para usar ese programa y para acceder a sus variables globales, uniforms.
- Clase **ShaderLoader**, definida en `shaderloader.hpp` y `shaderloader.cpp`. Lee de disco los shaders, los compila y enlaza para poder ejecutarlos. Se le indica los nombres de los archivos en disco de los dos shaders que se usan en el programa, «vertex» y «fragment» y devuelve un objeto del tipo `ShaderProgram`.

También defino los dos «shaders» que necesita OpenGL en el lenguaje GLSL, «vertex shader» en `shaders/shader.vert` y «fragment shader» en `shaders/shader.frag`.

En la figura 15 se muestra un diagrama de clases de esta implementación. Para simplificar la representación no he incluido las clases auxiliares encargadas de la lectura de disco.

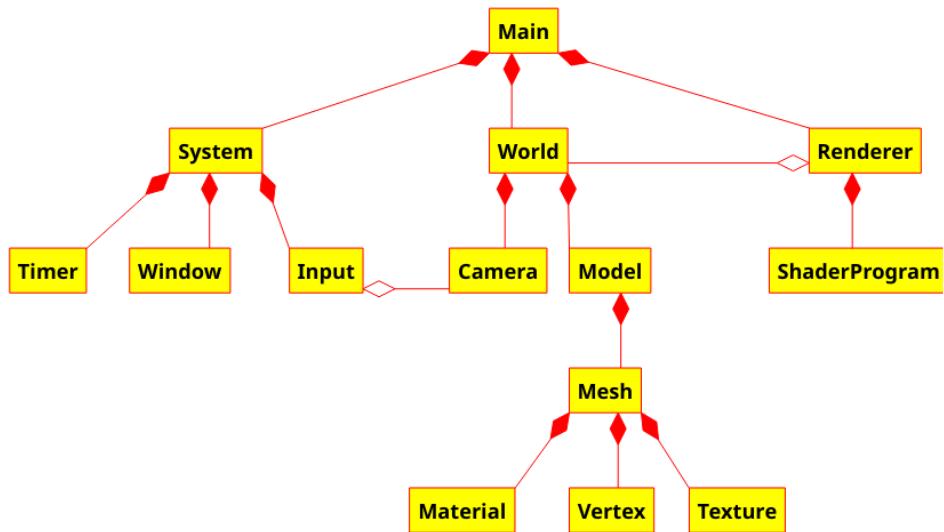


Figura 15: Diagrama de clases simplificado de la implementación del sombreado directo

5.3 Iteración 2. Sombreado diferido

En esta iteración modifíco la forma en la que se genera la imagen a mostrar para usar la técnica sombreado diferido descrita en el capítulo 4.1.2. También añado puntos de luz con atenuación y el componente especular a la ecuación de renderizado. Este método proporciona una forma de exportar los resultados de los cálculos realizados mediante rasterización y OpenGL; datos que en la siguiente iteración se usarán en OpenCL.

En la figura 16 se muestra un ejemplo de G-Buffer generado durante esta fase. Se ha separado el componente especular a una textura propia con la finalidad de poder mostrarlo en esta presentación, ya que en el G-Buffer real éste componente está contenido en el canal alfa de la textura que alberga el albedo. El G-Buffer utilizado en realidad está compuesto únicamente por tres texturas.

Para implementar esta técnica he introducido las siguientes modificaciones en el código:

- Duplicar el número de «shaders». Cada pareja se encarga de una de las pasadas del sombreado diferido. La primera, «shaders/gbuffer.vert» y «shaders/gbuffer.frag» genera el G-Buffer. La segunda,

«shaders/lighting.vert» y «shaders/lighting.frag» utiliza el G-Buffer y la información sobre puntos de luz para generar la imagen final.

- Sustituir la función de renderizado por la clase `DeferredShader` (`deferredshader.cpp` y `deferredshader.hpp`). Esta clase carga las dos parejas de shaders descritas anteriormente. Además, en la inicialización, se genera el framebuffer de OpenGL y las texturas para el G-Buffer. Cada vez que se llame a la función `render` se realizan las dos pasadas correspondientes a las dos parejas de «shaders». La información de los puntos de luz se hace llegar a OpenGL mediante *uniforms*, que son variables globales de GLSL. Defino un número máximo fijo de puntos de luz posibles. Generar un número variable de puntos de luz en un shader requeriría que la información sobre los puntos de luz se guardase en una estructura de memoria específica o bien escribir el código de ese shader dinámicamente, compilándolo con el número de luces ya definido como una constante. La causa es que GLSL no soporta arrays con una dimensión definida por una variable. Para evitar estas complejidades he definido el array de puntos de luz con un número máximo de luces fijo en el propio shader.

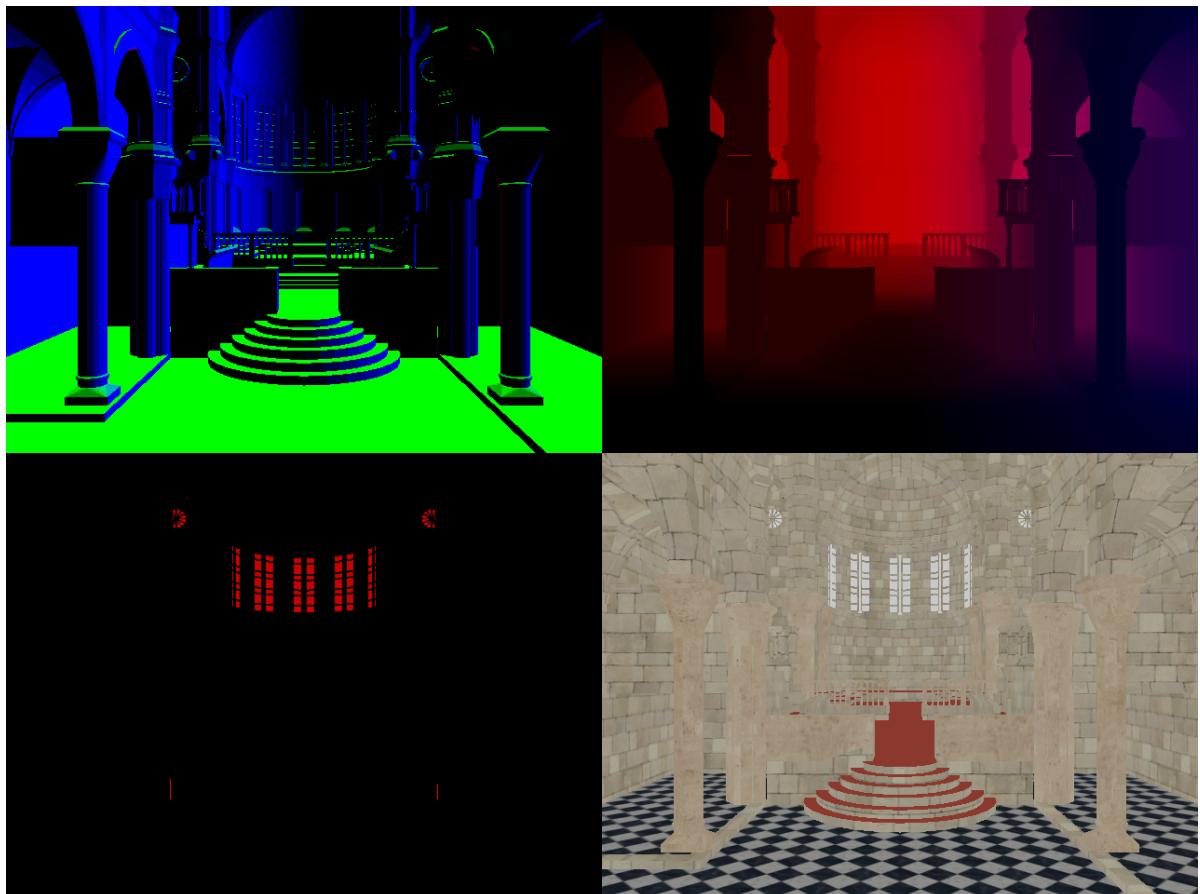


Figura 16: Ejemplo de G-Buffer.

De arriba a abajo, de izquierda a derecha: normales, posiciones, componente especular y albedo

- Añadir la clase `PointLight`, en el archivo `pointlight.hpp`. Contiene los valores que definen un punto de luz. Su color, intensidad difusa y especular, valores de atenuación lineal y cuadrática. Añado un vector de puntos de luz a la clase `World`.

5.4 Iteración 3. Técnica híbrida con OpenCL

5.4.1 Implementación

La técnica híbrida desarrollada consiste en sustituir la segunda pasada del sombreado diferido, que estaba implementada en OpenGL y se encargaba del cálculo del efecto de la iluminación, por una nueva que use OpenCL, según lo descrito en el capítulo 4.3.

En este punto se hace necesario, por lo tanto, un intercambio de información entre OpenGL y OpenCL. El G-Buffer que genera OpenGL tiene que ponerse a disposición

de OpenCL para que éste pueda realizar los cálculos de iluminación. De la misma forma, la imagen resultante generada en OpenCL debe ser visible por OpenGL, puesto que será el encargado de trasladarla a la pantalla.

Hay varias formas de realizar este intercambio de información. Un buen resumen se encuentra en este artículo del sitio web de Intel [SHE00], de Maxim Shevtsov. Las tres opciones que explica son:

- Compartición directa de una textura usando `clCreateFromGLTexture`.
- Crear un Pixel Buffer Object (PBO) intermedio para la textura OpenGL usando `clCreateFromGLBuffer`, actualizar el buffer con OpenCL y copiar el resultado a la textura.
- Mapear la textura GL con `glMapBuffer`, pasando el puntero resultante del host como un buffer OpenCL para procesarlo, y copiar el resultado de vuelta con `glUnmapBuffer`.

Cada uno de estos tres métodos tiene ventajas y desventajas. Los dos primeros requieren la extensión «`cl_khr_gl_sharing`», mientras que el tercero no requiere ninguna extensión. El primero es el único que realiza la compartición sin copiar los datos. Por lo tanto, siempre que todos los cálculos se realicen en el mismo dispositivo, es el más rápido. Sin embargo, nos restringe a texturas que sean transferibles de un sistema a otro, lo que nos obliga a trabajar con 4 canales RGBA en coma flotante o byte sin signo.

Antes de tomar una decisión, he realizado el siguiente cálculo, para obtener el tamaño mínimo de texturas a compartir, teniendo en cuenta las 3 texturas del G-Buffer más la de la imagen final:

Textura	Canales	Tipo de dato	Bytes tipo dato	Bytes por píxel
Albedo + especular	4	byte	1	4
Posiciones	3	float	4	12
Normales	3	byte	1	3
Imagen final	3	byte	1	3
			Total	22

Tabla 1: Tamaño de las texturas del G-Buffer

Suponiendo una resolución razonable de 1024x768 píxeles, para la generación de cada imagen se necesitan $1024 \times 768 \times 22 = 17.301.504$ bytes, aproximadamente 16,5 megabytes.

El objetivo normal de una animación en tiempo real es igualar la tasa de refresco de un monitor, que actualmente es, como mínimo, de 60Hz. Si nos marcamos ese objetivo de 60 imágenes por segundo, habría que copiar 1.038.090.240 bytes por segundo, aproximadamente 0,97 GB/s.

Según las especificaciones de la tarjeta gráfica usada en este trabajo, una AMD R9 285, de gama media y de casi dos años de antigüedad, el ancho de banda de su memoria es de 176 GB/s, por lo que, a priori, esa copia no debería provocar una merma importante del rendimiento.

Sin embargo, como las texturas disponibles con la extensión «cl_khr_g1_sharing» son adecuadas y la extensión necesaria está disponible, elijo el primer método. Puesto que los otros dos métodos no presentan ninguna ventaja adicional para este trabajo, se evita esa copia innecesaria de al menos 1GB por segundo, tenga efecto visible o no en el rendimiento.

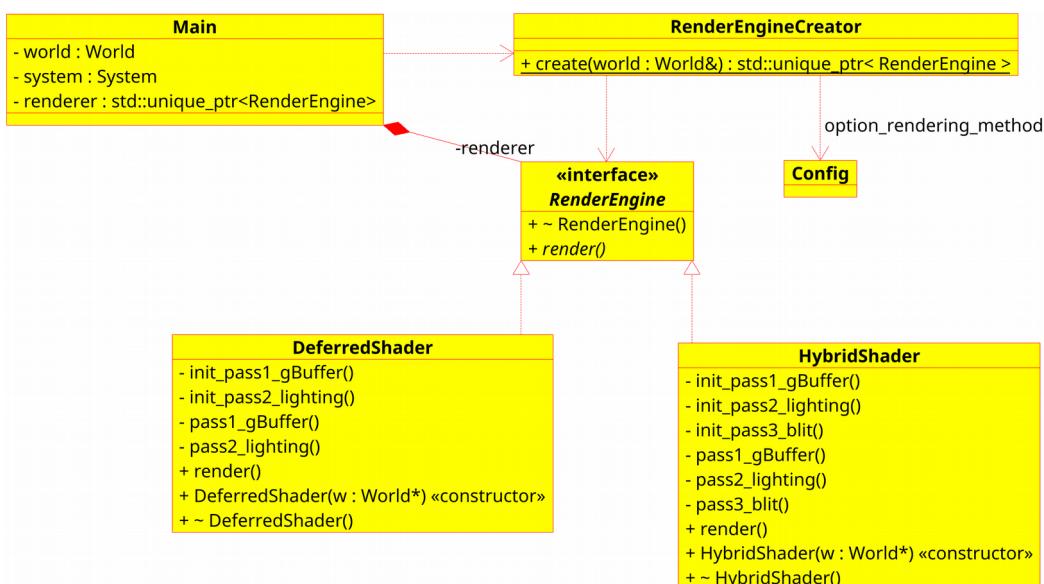


Figura 17: Diagrama de clases del proceso de creación de la clase `RenderEngine`.
Implementa el patrón de diseño «Factory Method»

Decido mantener el código de la iteración 2 funcional, creando una plantilla de C++ (template) y dando la opción de elegir qué método se quiere usar durante la compilación, con la intención de facilitar mediciones posteriores y comparativas entre los dos métodos. Más adelante vuelvo sobre este punto y sustituyo la plantilla por una clase abstracta y herencia. Este cambio se debe a que la plantilla proporciona polimorfismo en tiempo de compilación y he preferido el polimorfismo en tiempo de ejecución, que permite no tener que volver a compilar el proyecto para cambiar el método de renderizado, pudiendo incluso cambiarse éste durante la ejecución del

programa. Implemento este polimorfismo dinámico mediante un «Factory Method» [GOF000]. En la figura 17 se puede observar el diagrama de clases.

Implemento las siguientes clases nuevas:

- **RenderEngine**, en el archivo `renderengine.hpp`: Clase abstracta que establece qué interfaz deben cumplir los métodos de renderizado que se definan en el programa: deben tener un método `render`, al que se llamará para realizar el renderizado.
- **RenderEngineCreator**, en el archivo `renderenginecreator.hpp`. Implementa el patrón de diseño «Factory Method», de forma que en la función principal `main` no es necesario incluir ningún código específico de la técnica de renderizado a utilizar. Es esta clase, apoyándose en el parámetro `option_rendering_method` del objeto `Config`, la que instancia la subclase adecuada.
- **DeferredShader**, en los archivos `deferredshader.cpp` y `deferredshader.hpp`: Modifico la clase desarrollada en el apartado anterior para que herede de `RenderEngine` y cumpla su interfaz.
- **HybridShader**, en los archivos `hybridshader.cpp` e `hybridshader.hpp`: Clase que alberga el código para utilizar la técnica híbrida. Es una subclase de `RenderEngine`.
- **Config**, en el archivo `config.hpp`: Alberga parámetros de configuración.
- **CL_Device**, en los archivos `cl_device.cpp` y `cl_device.hpp`: Código para obtener información de los dispositivos OpenCL en el sistema.
- **CL_Platform**, en los archivos `cl_platform.cpp` y `cl_platform.hpp`: Código para obtener información de las plataformas OpenCL disponibles en el sistema.
- **CLKernelManager**, en los archivos `clkernelmanager.cpp` y `clkernelmanager.hpp`: Administrador de kernels de OpenCL y métodos asociados. Realiza la inicialización, lee kernels del disco y los compila, y contiene el contexto, la cola de mensajes y demás datos propios de OpenCL. Permite llamar al kernel compilado.
- kernel de OpenCL «`kernels/render.cl`», que realiza la segunda pasada del renderizado, calculando los efectos de la iluminación y las sombras.

Para poder usar la extensión «`cl_khr_gl_sharing`» es necesario asociar el puntero de la función al código de la misma de la biblioteca. Este proceso es necesario para usar cualquier función de las extensiones. Un ejemplo práctico se puede encontrar en el ejemplo «SimpleGL» del SDK 3.0 de AMD para OpenCL.

(SDK/samples/opencl/cl/1.x/SimpleGL/). Basándome en ese ejemplo, implemento el código del archivo «`clkernelmanager.cpp`»:

```
typedef CL_API_ENTRY cl_int(CL_API_CALL *clGetGLContextInfoKHR_fn)(  
    const cl_context_properties *properties, cl_gl_context_info param_name,  
    size_t param_value_size, void *param_value, size_t *param_value_size_ret);  
// Rename references to this dynamically linked function to avoid  
// collision with static link version  
#define clGetGLContextInfoKHR clGetGLContextInfoKHR_proc  
static clGetGLContextInfoKHR_fn clGetGLContextInfoKHR;
```

También es necesario crear el contexto de OpenCL a partir del de OpenGL. Esto tiene la desventaja de que hay que introducir código no portable, usando las macros de preprocesador para poder soportar más de una arquitectura. Dado que se limita a unas pocas líneas de código incluyo la parte específica de Linux, Windows y Mac OS X siguiendo el ejemplo mencionado anteriormente. El motivo es que el acceso a ese contexto OpenGL no se hace a través de GLFW, sino que es necesario usar llamadas específicas del sistema de ventanas nativo.

Queda otra textura que hay que compartir entre ambas tecnologías. Es la que alberga la imagen final generada en OpenCL y que debe ser volcada a la pantalla por OpenGL. Realizo la compartición de la misma forma que para el G-Buffer y adapto el código para que se haga la copia de esa textura al *framebuffer* por defecto, haciendo simplemente un volcado de la textura a la memoria de pantalla.

Dado que tanto OpenGL como OpenCL van a acceder a las mismas texturas, es necesario sincronizar su acceso. Antes de ejecutar el *kernel* de OpenCL bloquee las texturas y al finalizar las libere, llamando a «`enqueueAcquireGLObjects`» y «`enqueueReleaseGLObjects`», respectivamente.

Además, puesto que las llamadas a OpenCL son asíncronas, incluyo un evento que indica cuándo termina la ejecución del *kernel* de OpenCL y espero a este evento antes de liberar las texturas.

Después de estos cambios, consigo el mismo resultado que en el sombreado diferido, pero sustituyendo la segunda pasada por el código en OpenCL.

El siguiente paso es calcular las sombras utilizando técnicas de trazado de rayos. Este cálculo consiste en definir un rayo, que es un vector con un origen definido, desde el píxel que queremos saber si está iluminado hasta el punto de luz que supuestamente lo ilumina. Si este vector sufre alguna intersección con algún elemento de la geometría de la escena entre esos dos puntos, la luz no llegará a iluminar ese píxel. En caso

contrario, se tiene en cuenta el efecto de esa luz en la iluminación del píxel que estamos comprobando.

Para poder realizar estas comprobaciones es necesario disponer de la información de la geometría de la escena, además de los puntos de luz. Esta información está ya almacenada en OpenGL, pero en un formato en el que me ha sido imposible compartir con OpenCL. El problema es que cada malla de triángulos está almacenada en un *buffer* diferente, correspondiente a un Mesh, según lo explicado en el apartado 5.2, dedicado al sombreado directo. Cada *buffer* tendría que ser pasado al kernel como un parámetro, pero eso implicaría crear un *kernel* con un número variable de parámetros, lo que no está permitido según la especificación de OpenCL. Después de valorar posibles soluciones, decidí duplicar la información de la geometría y volver sobre este punto en un momento posterior. Como explico en el apartado siguiente, esta decisión finalmente es acertada, ya que, en cualquier caso, es necesario usar una estructura para acelerar el recorrido de las comprobaciones de intersección rayo-geometría, por lo que los datos de OpenGL no nos son útiles por su organización.



Figura 18: Imagen generada con la técnica híbrida.
En ella se pueden observar las sombras proyectadas

Además, únicamente es necesario cargar la geometría en la inicialización del *kernel*, a no ser que parte de ésta cambie. En este momento sólo planteo escenas estáticas, por lo que no hay una reducción de rendimiento por este motivo.

Una vez el *kernel* tiene la información de la geometría, realiza la comprobación de cada píxel con cada triángulo de la geometría por punto de luz, para comprobar si ésta llega a iluminar el píxel.

El desarrollo de un algoritmo eficiente para comprobar intersecciones rayo-triángulo ha sido objeto de varias publicaciones y estudios y constituye por sí sólo material para un trabajo independiente. Por lo tanto, opto por apoyarme en uno de los algoritmos conocidos, adaptando su implementación.

Baso el algoritmo de intersección rayo-triángulo en el algoritmo de Möller y Trumbore «Fast, minimum storage ray/triangle intersection» [MT000] de 1997, tal y como está descrito en Physically Based Rendering [PHH00], haciendo las adaptaciones necesarias para pasarlo de C++ a C para OpenCL. Las modificaciones consisten en los tipos de datos, como float3 para vectores de tres dimensiones en coma flotante, la estructura que contiene el tipo de datos «Ray», los nombres de las operaciones vectoriales y los argumentos que recibe la función, ya que hay punteros a estructuras a memoria global.

Dejo para más adelante el estudio de si este algoritmo es el más idóneo y de optimizaciones posibles, como dejar calculados previamente los lados de los triángulos, ya que en estos momentos lo más importante es la técnica en su conjunto y el uso de OpenCL, no este algoritmo en concreto.

Una vez depurado y funcionando el programa, observo que el resultado es visualmente correcto, pero tarda varios segundos en generar cada imagen. Es necesario reducir el número de comprobaciones rayo-triángulo por píxel y fuente de luz, tarea descrita en el capítulo 5.5.

5.4.2 Optimización y estudio de rendimiento

Con el fin de realizar el estudio de rendimiento, comienzo por identificar cuál es una cota superior razonable. Para obtener dicha cota comprobar el rendimiento de un *kernel* mínimo, que únicamente copia la textura «albedo» del G-Buffer a la textura final:

```
_kernel void render(__read_only image2d_t g_albedo_spec,
```

```

__read_only image2d_t g_position,
__read_only image2d_t g_normal,
__write_only image2d_t output,
const SceneAttribs attrs, const float3 view_position,
__constant PointLight *point_lights, const int point_lights_nr,
__global const Triangle *triangles ) {
int2 coord = (int2)(get_global_id(0), get_global_id(1));
const float3 diffuse = read_imagef(g_albedo_spec, imageSampler, coord).xyz;
write_imagef(output, coord, (float4)(diffuse, 0.0f));
}
  
```

En esta versión ignoro todos los demás parámetros y compruebo el rendimiento con varias resoluciones del modelo «Sponza», descrito en el capítulo 6.1. Este *kernel* sería el más rápido posible, ya que no realiza ningún cálculo ni accede a ningún valor más allá del albedo.

La figura 19 es una captura del programa CodeXL y corresponde a una ejecución del programa a una resolución de 1024x768 píxeles. Se observa que entre dos llamadas consecutivas de ejecución del *kernel* pasan aproximadamente 1,5 milisegundos, pese a que la ejecución en sí es de unos 56 microsegundos.

También se observa que desde que llamamos a `clSetKernelArg` hasta que finaliza `clReleaseEvent` pasa aproximadamente 1 milisegundo. Probando otra ejecución, esta vez a 320x240, se obtiene de nuevo como resultado 1 milisegundo. Por lo tanto, concluyo que el proceso mínimo que se necesita realizar para ejecutar el *kernel* (actualizar parámetros, adquirir texturas, lanzar la ejecución, liberar texturas y esperar a que finalice) está cerca de 1 milisegundo. No podemos esperar, por lo tanto, más de 1000FPS el en mejor de los casos, suponiendo que la ejecución de OpenGL y otras tareas del programa no tarden absolutamente nada. Dado que en realidad estos procesos sí tienen una duración apreciable, se confirma el resultado de las mediciones realizadas en estas condiciones, que arrojan valores de 600 a 800FPS a una resolución de 1024x768 píxeles y unos 990FPS a una resolución de 320x240 píxeles.

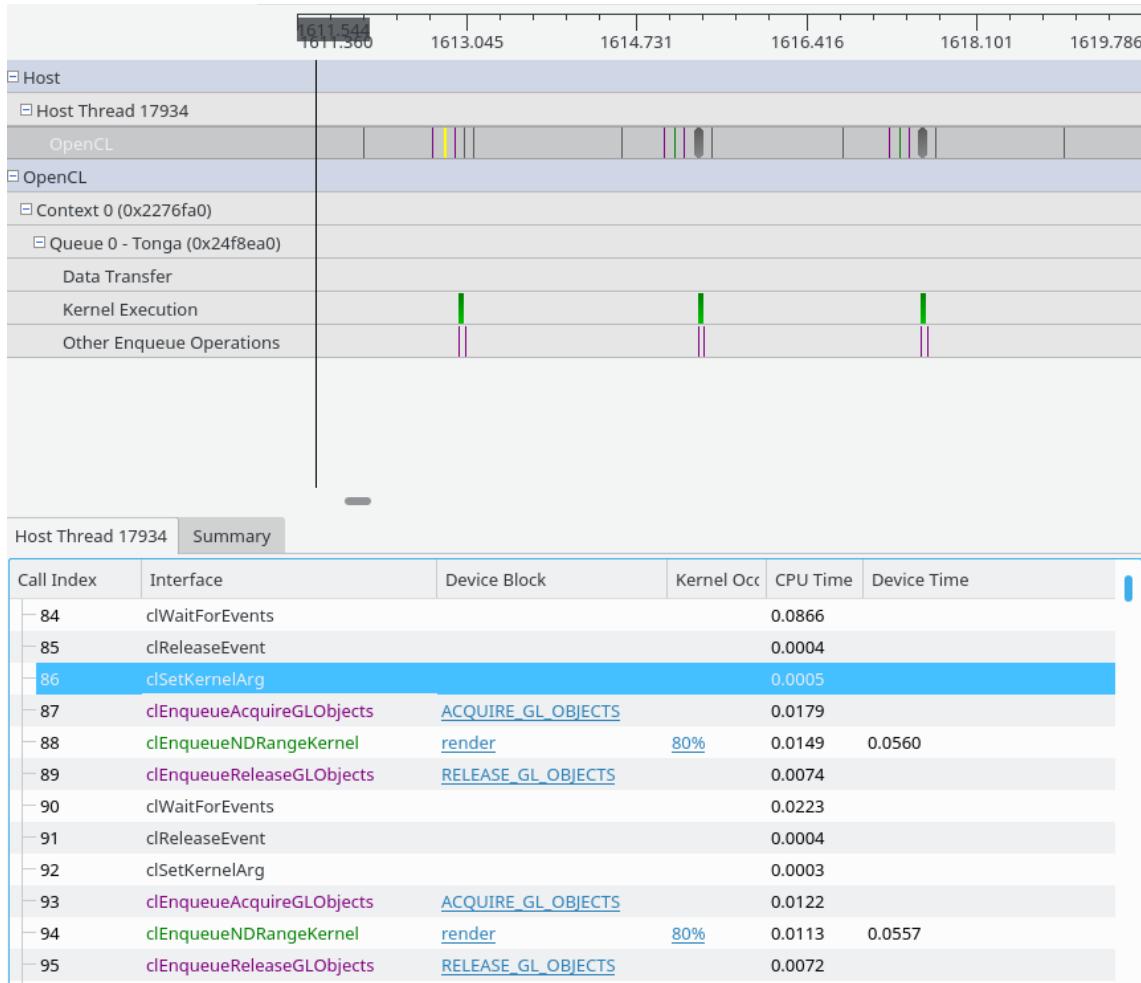


Figura 19: Captura de la herramienta CodeXL inspeccionando el kernel "mínimo"
Tiempos en milisegundos

Es razonable, por lo tanto, esperar para este modelo valores **inferiores a 700FPS** a una resolución de 1024x768 con este hardware.

Realizo las mismas comprobaciones con el modelo «Sibenik» [MAC000] y arroja valores equivalentes. De nuevo el tiempo utilizado por OpenCL es de alrededor de 1 milisegundo. Sin embargo, OpenGL requiere más tiempo para producir las texturas del G-Buffer y las imágenes por segundo bajan hasta las 300 – 400. Este resultado es lógico, puesto el proceso de generación del G-Buffer es el único en estas condiciones que se ve afectado por la complejidad de la geometría.

Una vez realizada esta comprobación, paso a medir la ejecución sin la generación de sombras a diferentes resoluciones y a compararla con el sombreado diferido.

LUCES	FPS MEDIO OPENCL	FPS MEDIO DIFERIDO
1	503,1	995,9
3	498,9	997,1
6	487,4	984,1
9	331,6	972,0
12	328,5	962,4
15	292,5	937,5
18	270,0	925,7

Tabla 2: Comparativa inicial de rendimiento entre técnica híbrida y sombreado diferido

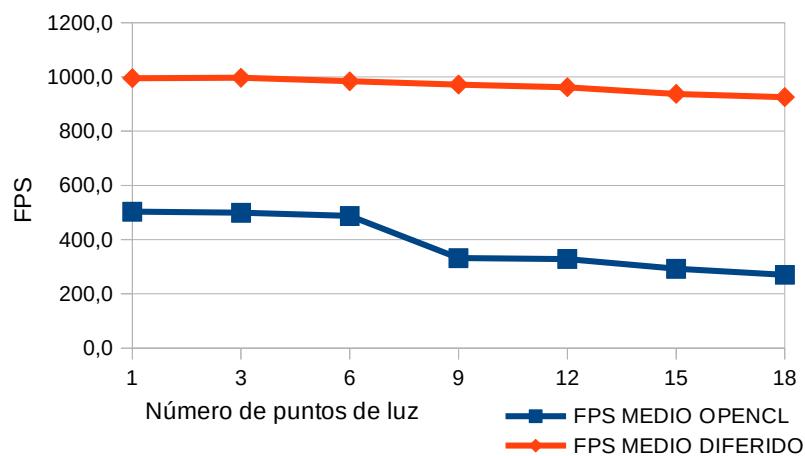


Figura 20: Representación gráfica de la comparativa inicial de rendimiento entre técnica híbrida y sombreado diferido

Debido al coste de sincronización y compartición de texturas sería lógico que la recta de OpenCL fuese paralela, aunque por debajo, a la de OpenGL. Sin embargo, se observa cómo la reducción de rendimiento por punto de luz es mucho mayor en OpenCL que en OpenGL. Para obtener valores más concretos utilice de nuevo la herramienta CodeXL, para obtener el tiempo exacto de ejecución del *kernel* según cambia el número de fuentes de luz.

Desarrollo

N.º Luces	Tiempo ejecución Kernel (ms)
1	0,387
3	0,440
6	0,791
9	1,143
12	1,150
15	1,844
18	2,199

Tabla 3: Tiempo de ejecución del kernel según el número de puntos de luz

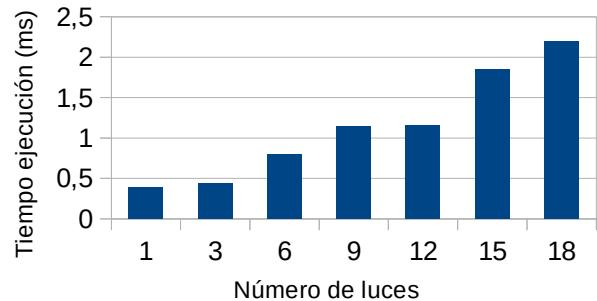


Figura 21: Representación del tiempo de ejecución del kernel según el número de puntos de luz

También en la herramienta CodeXL, se aprecia que el factor que está limitando el rendimiento es el número de VGPR (Vector General Purpose Register) disponibles (figura 22).

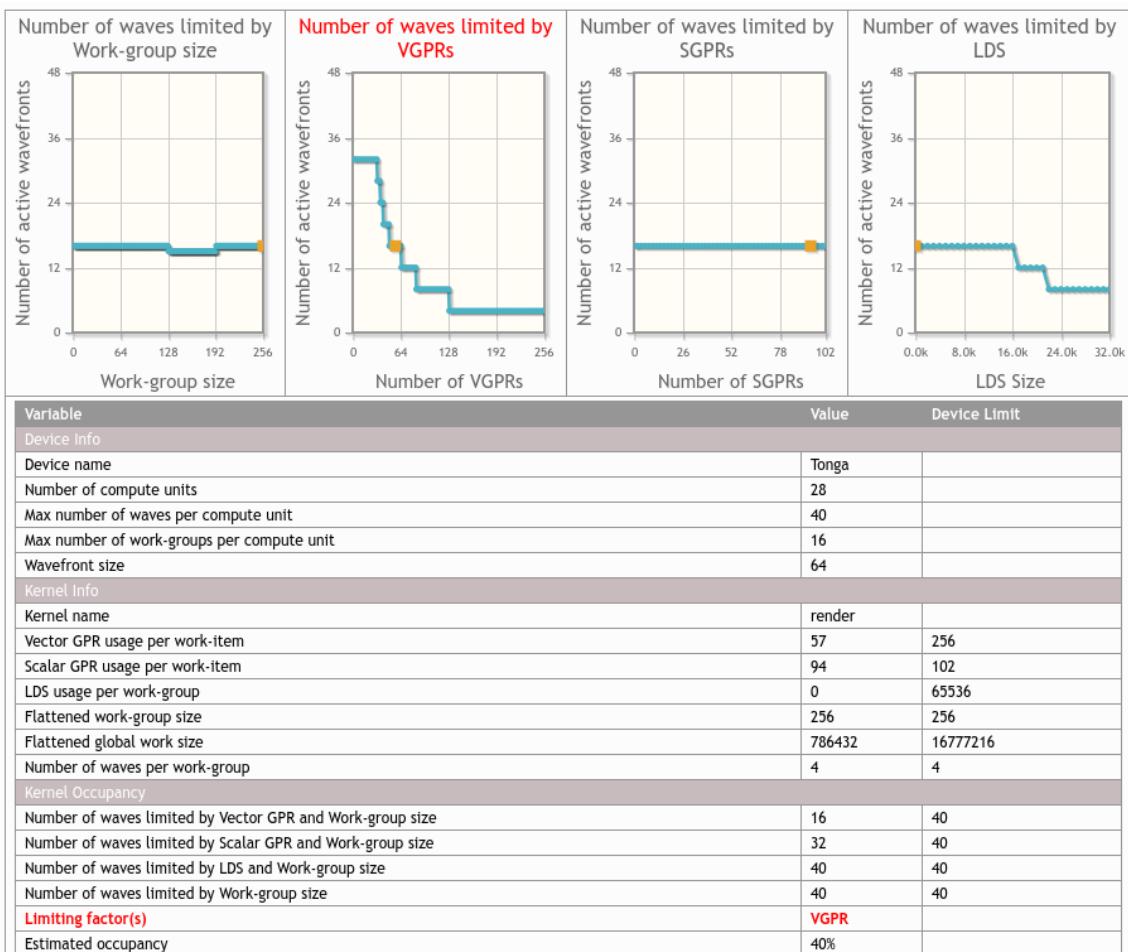


Figura 22: Captura de CodeXL de un kernel limitado por VGPR.

Para entender el significado de la figura 22 hay que tener en cuenta la arquitectura de OpenCL, de la que se puede encontrar un resumen en el anexo 11.1. Cabe recordar que un *wavefront* corresponde con el número de hilos con el que se lanza un *work group*, que a su vez está compuesto por *work items*. Como se indica en el capítulo 10 de [KMS00], dedicado al estudio de rendimiento y depuración de código en OpenCL, el número de *wavefronts* que puede ser asignado a una *compute unit* está limitado por los recursos disponibles en ésta. Más concretamente, está limitado por la cantidad de memoria local y registros vectoriales y escalares de los que dispone esa *compute unit*. El rendimiento de *kernels* que realizan muchas transacciones de memoria se beneficia de tener muchos *waves* lanzados. El motivo es que se utilizan los registros para ocultar la latencia de memoria: mientras un hilo está esperando a que se cargue en un registro un dato alojado en memoria global, la *compute unit* puede ocuparse de realizar cálculos de otros hilos, evitando así el tiempo de inactividad. En el caso de que el *kernel* realice muchas operaciones pero poco acceso a memoria, conviene tener pocos *waves* concurrentes, para un mejor aprovechamiento de la memoria caché.

En un uso normal, para aumentar el rendimiento, conviene hacer un uso equilibrado de los recursos y utilizar tanto los registros vectoriales como los escalares y memoria local.

Dado el tamaño de las texturas a tratar y el número de ciclos por segundo requeridos, parece que nos encontramos en el caso en el que se están utilizando los registros de propósito general para ocultar la latencia de acceso a memoria. Realizo varias comprobaciones para asegurarme e intentar mejorar el rendimiento.

En un primer paso intento reducir el uso de VGPR. Compruebo que el bucle del *kernel* en el que se recorren los puntos de luz utiliza una y otra vez los valores de componente difuso, normal, posición y valor especular de ese píxel. Por lo tanto, no se puede liberar o reutilizar ninguno de los vectores hasta que finalice la última iteración del bucle.

Una opción para usar menos VGPR es mover a memoria local las posiciones de memoria global que se vayan a reutilizar en lugar de almacenarlas en registros vectoriales. Es una forma habitual de optimización en OpenCL para *kernels* en los que se comparten datos entre *work items*, ya que se sustituyen accesos repetidos a memoria global por accesos a memoria local, sincronizados debidamente con *barriers*. Aunque no es el caso en el que nos encontramos, realizo la prueba, generando un array en memoria local en el que guardo los valores de albedo, normal y posición para

ese píxel. En lugar de usar la memoria privada del *work item*, se usa memoria local y se liberan VGPR:

```

// Calculates point lights contribution to shading a specific point
// on a surface
float3
pointLightsColor(__constant PointLight *point_lights, const int point_lights_nr,
                 const float3 view_pos, __local float3 *values,
                 // const float3 surface_pos, const float3 surface_normal,
                 // const float3 surface_diffuse,
                 const float surface_specular, bool shadows_enabled,
                 __global const Triangle *triangles,
                 __global const BVHNode *nodes) {

    float3 color = (float3)(0.0f, 0.0f, 0.0f);
    const float3 view_dir = normalize(view_pos - values[POS]);

    for (int i = 0; i < point_lights_nr; i++) {
        __constant PointLight *pLight = &point_lights[i];

        // Attenuation
        const float dist = distance(pLight->p_position, values[POS]);
        const float attenuation = 1.0f / (1.0f + (pLight->linear * dist) +
                                         (pLight->quadratic * dist * dist));

        const float3 light_dir = normalize(pLight->p_position - values[POS]);
        if (shadows_enabled) {
            Ray r = (Ray){values[POS], light_dir, EPSILON, dist};
            if (intersects(&r, triangles, nodes))
                continue;
        }

        // Diffuse Lambertian
        const float ang = max(dot(values[NOR], light_dir), 0.0f);
        color += (ang * values[ALB] * pLight->p_color) * attenuation;

        // Specular Blinn-Phong
        const float3 halfway_dir = normalize(light_dir + view_dir);
        const float spec = pow(max(dot(values[NOR], halfway_dir), 0.0f), 16.0f);
        const float3 specular = pLight->p_color * spec * surface_specular;
        color += specular * attenuation;
    }

    return color;
}

// Main kernel
kernel void
render(__read_only image2d_t g_albedo_spec, __read_only image2d_t g_position,
       __read_only image2d_t g_normal, __write_only image2d_t output,
       const SceneAttribs attribs, const float3 view_position,
       __constant PointLight *point_lights, const int point_lights_nr,
       __global const Triangle *triangles, __global const BVHNode *nodes) {

    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    __local float3 values[3 * 256];
    int coord_local = 3 * (get_local_id(1) * get_local_size(0) + get_local_id(0));
    values[coord_local + ALB] =
        read_imagef(g_albedo_spec, imageSampler, coord).xyz;
    const float specular = read_imagef(g_albedo_spec, imageSampler, coord).w;
    values[coord_local + POS] = read_imagef(g_position, imageSampler, coord).xyz;
    const float3 norm = read_imagef(g_normal, imageSampler, coord).xyz;
    values[coord_local + NOR] = normalize(2 * norm - (float3)(1, 1, 1));

    // Ambient light
}

```

```

float3 color = values[coord_local + ALB] * attribs.ambient;

// Point Lights (diffuse + specular)
color += pointLightsColor(
    point_lights, point_lights_nr, view_position, &values[coord_local],
    specular, (bool)attribs.shadows_enabled, triangles, nodes);

// Write result
write_imagef(output, coord, (float4)(color.x, color.y, color.z, 0.0f));
}
  
```

No ha sido necesario sincronizar el acceso con *barriers*, puesto que no se comparten datos entre hilos. El resultado de la prueba es que el rendimiento baja a casi la mitad, ya que el acceso a memoria local es más lento que el acceso a memoria privada y en este caso no hemos reducido el número de accesos a memoria global. Además, el uso de memoria local empieza a limitar también el número de *wave fronts* lanzados simultáneamente.

En otro intento de reducir el uso de VGPR, modiflico el *kernel* para que el número de punto de luz sea la tercera dimensión del grupo de trabajo y elimino el bucle que recorre los puntos de luz. También modiflico la llamada de `clEnqueueNDRangeKernel` de la misma forma para reflejar las tres dimensiones en lugar de dos. Se producirán más cálculos por cada píxel, ya que hasta ahora hay ciertos cálculos que se realizan fuera del bucle que recorre los puntos de luz, pero se liberarán antes los recursos. Además, retraso la carga de datos de memoria global a VGPR hasta que son necesarios:

```

kernel void
render(__read_only image2d_t g_albedo_spec, __read_only image2d_t g_position,
       __read_only image2d_t g_normal, __write_only image2d_t output,
       const SceneAttribs attribs, const float3 view_pos,
       __constant PointLight *point_lights, const int point_lights_nr,
       __global const Triangle *triangles, __global const BVHNode *nodes) {

    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    __constant PointLight *pLight = &point_lights[get_global_id(2)];

    // Attenuation
    const float3 surface_pos = read_imagef(g_position, imageSampler, coord).xyz;
    const float dist = distance(pLight->p_position, surface_pos);
    const float attenuation = 1.0f / (1.0f + (pLight->linear * dist) +
                                      (pLight->quadratic * dist * dist));

    const float3 light_dir = normalize(pLight->p_position - surface_pos);

    // Diffuse Lambertian
    float3 color = read_imagef(g_albedo_spec, imageSampler, coord).xyz;
    float3 surface_normal = read_imagef(g_normal, imageSampler, coord).xyz;
    surface_normal = normalize(2 * surface_normal - (float3)(1, 1, 1));
    const float ang = max(dot(surface_normal, light_dir), 0.0f);
    color = (ang * pLight->p_color * attenuation + attribs.ambient) * color;

    // Specular Blinn-Phong
    const float surface_specular =
        read_imagef(g_albedo_spec, imageSampler, coord).w;

    const float3 view_dir = normalize(view_pos - surface_pos);
  
```

```

const float3 halfway_dir = normalize(light_dir + view_dir);
const float spec = pow(max(dot(surface_normal, halfway_dir), 0.0f), 16.0f);
const float3 specular = pLight->p_color * spec * surface_specular;
color += specular * attenuation;

// Write result
write_imagef(output, coord, (float4)(color.x, color.y, color.z, 0.0f));
}

```

El resultado está representado en la figura 23. Se reduce el uso de VGPR de 57 a 6 para un único punto de luz, llegando a una ocupación estimada del 80% y finalizando la ejecución en 0,156 ms. Este resultado supone una mejora en tiempo del 60%, lo que parece muy prometedor.

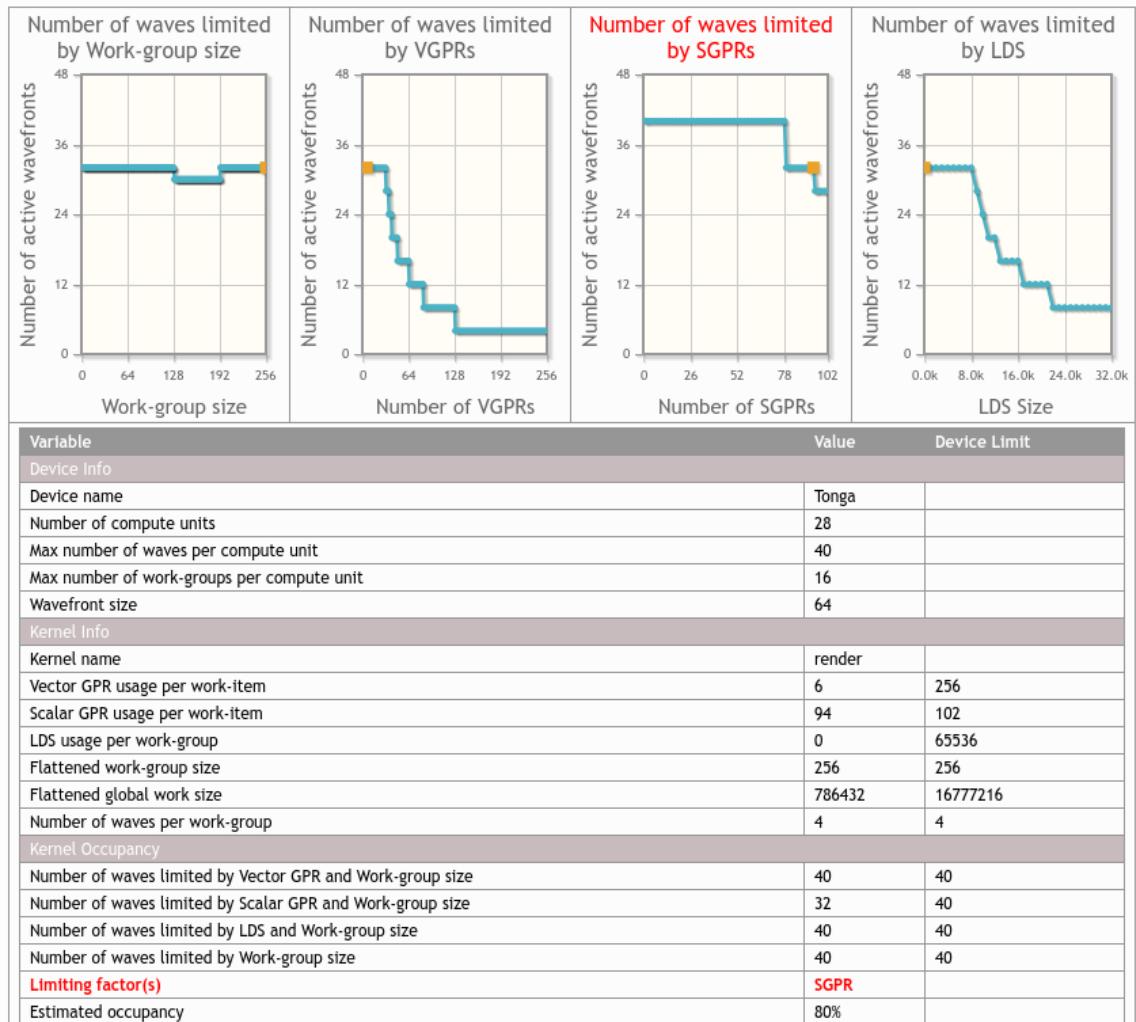


Figura 23: Captura de CodeXL. Ocupación estimada al 80%

Sin embargo, para utilizar más de una fuente de luz hay que sincronizar los diferentes *work items* que van a escribir en un determinado píxel, ya que de no hacerlo, se sobrescribiría una y otra vez el píxel por cada punto de luz, sin combinar los efectos. Podría utilizarse memoria local con este propósito si todos los puntos de luz estuviesen en el mismo *work group*, pero no conocemos cuántos puntos de luz existirán y cada dispositivo tiene unos límites en cuanto al número de *work items* por *work group*. Por lo tanto, dado que es muy probable que no se dé esta condición, se puede utilizar un patrón «reducción» [\[AMD0001\]](#). En este patrón se guarda el resultado de cada cálculo en una posición fija de un *array* definida por unas coordenadas únicas para cada dato, que en este caso son posición x, posición y y número de punto de luz. Una vez generado este array se recorre combinando sus valores.

Codifico la primera fase, guardando de forma independiente el aporte de luz de cada fuente en un *buffer* generado con el tamaño: ancho * alto * número luces * tamaño tipo float * 3:

```
__kernel void render(__read_only image2d_t g_albedo_spec,
                    __read_only image2d_t g_position,
                    __read_only image2d_t g_normal,
                    __write_only image2d_t output, const SceneAttribs attribs,
                    const float3 view_pos, __constant PointLight *point_lights,
                    const int point_lights_nr,
                    __global const Triangle *triangles,
                    __global const BVHNode *nodes, __global float3 *output2) {

    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    __constant PointLight *pLight = &point_lights[get_global_id(2)];

    // Attenuation
    const float3 surface_pos = read_imagef(g_position, imageSampler, coord).xyz;
    const float dist = distance(pLight->p_position, surface_pos);
    const float attenuation = 1.0f / (1.0f + (pLight->linear * dist) +
                                      (pLight->quadratic * dist * dist));

    const float3 light_dir = normalize(pLight->p_position - surface_pos);

    // Diffuse Lambertian
    float3 color = read_imagef(g_albedo_spec, imageSampler, coord).xyz;
    float3 surface_normal = read_imagef(g_normal, imageSampler, coord).xyz;
    surface_normal = normalize(2 * surface_normal - (float3)(1, 1, 1));
    const float ang = max(dot(surface_normal, light_dir), 0.0f);
    color = (ang * pLight->p_color * attenuation + attribs.ambient) * color;

    // Specular Blinn-Phong
    const float surface_specular =
        read_imagef(g_albedo_spec, imageSampler, coord).w;
    const float3 view_dir = normalize(view_pos - surface_pos);
    const float3 halfway_dir = normalize(light_dir + view_dir);
    const float spec = pow(max(dot(surface_normal, halfway_dir), 0.0f), 16.0f);
    const float3 specular = pLight->p_color * spec * surface_specular;
    color += specular * attenuation;

    // Write result
    output2[(get_global_id(1) * get_global_size(0) + get_global_id(0)) *
            get_global_size(2) +
            get_global_id(2)] = color;
}
```

El resultado es que, sin llegar a realizar el paso de unificación de resultados, la ejecución del *kernel* con 18 luces pasa de 2,20ms a 22,9ms. Según CodeXL hay una ocupación del 80%, pero el factor limitante es el número de SGPR. Realizando la prueba de copiar el albedo sin realizar ninguna otra operación, consigo 70FPS a 1024x768, lo que sería la cota superior usando esta técnica. La explicación es que el nuevo *buffer* introducido para implementar el patrón *reducción* está moviendo 1024x768x18x3x4x70 bytes, unos 11GB, por segundo. Dado que el problema parecía ser excesivo uso de memoria global, esta prueba no ha hecho más que empeorar el resultado y confirmar que éste es realmente un factor que está limitando el rendimiento.

Finalmente decidí mantener la estructura original del *kernel* y añadir otro tipo de optimizaciones. Recupero el *kernel* inicial y añado dos optimizaciones: primero sustituyo las llamadas a “normalize” por “fast_normalize” y “distance” por “fast_distance”, puesto que la pérdida de precisión no es apreciable en el resultado y es preferible aumentar la velocidad. En segundo lugar comprobar si el componente especular es cero para evitar los cálculos asociados a ese aporte de energía cuando no sean necesarios. En escenas en las que no todos los objetos tienen brillo especular se evita un número importante de cálculos por cada fuente de luz.

Realizo de nuevo mediciones una vez implementadas estas optimizaciones. El resultado está representado en la tabla 4:

LUCES	FPS INICIAL	FPS OPTIMIZADO
1	503,1	533,8
3	498,9	516,6
6	487,4	511,9
9	331,6	508,8
12	328,5	495,6
15	292,5	491,2
18	270,0	486,2

Tabla 4: Comparación de rendimiento en fps antes y después de optimizar el *kernel*

En la nueva versión se realizan menos cálculos y de forma más rápida, liberando antes VGPR para ser usados en la carga de datos desde memoria global. Decido continuar adelante utilizando esta versión, ya que es la más sencilla y la que mejor rendimiento proporciona con más de un punto de luz.

El estudio de rendimiento de la técnica híbrida con la generación de sombras activada se encuentra desarrollado en el capítulo 5.5.2.

5.5 Iteración 4. Aceleración de la técnica híbrida mediante BVH

5.5.1 Implementación

Para procesar la información de sombras en el paso anterior se realizaba una comprobación por cada píxel y punto de luz con toda la geometría de la escena. Uno de los modelos utilizados en las pruebas, el patio de Sponza de Crytek, que se considera hoy en día una escena sencilla, tiene algo más de 260.000 triángulos. Para una resolución de 1024x768 píxeles, supone 204.472.320.000 comprobaciones de intersección rayo-triángulo por imagen y luz. Esto explica la aparente lentitud.

Resulta evidente que hay que reducir el número de comprobaciones a realizar. Existen varias técnicas que tienen esta finalidad, siendo la más habitual el uso de alguna estructura que particione el espacio de la escena y permita descartar grupos de triángulos que estén contenidos en zonas por las que no pasa el rayo que queremos comprobar.

Hay varias estructuras de este tipo conocidas, como por ejemplo: árboles binarios, arboles de ocho dimensiones (octrees), árboles de k-dimensiones (kdtrees) o jerarquías de volúmenes de recubrimiento (Bounding Volume Hierarchies).

Después de consultar en varias fuentes, entre ellas el libro Physically Based Rendering [PHH00] he decidido implementar la estructura BVH, descrita anteriormente en el capítulo 4.4. El motivo es que, según el citado libro, el rendimiento es bueno en todo tipo de escenas y es fácil «serializar» o «aplanar» el árbol resultante para convertirlo en un array. Aunque el rendimiento que proporciona el Kd-tree también es muy bueno, el tiempo necesario para generar la estructura es mucho mayor.

Realizo la implementación en la CPU con C++, de forma que se construye el árbol al cargar la escena y se pone a disposición del código OpenCL. La ventaja es que se simplifica su generación, pero tiene el inconveniente de no ser apto para escenas en las que la geometría es dinámica, ya que tendría que volver a generarse el árbol BVH cada vez que cambie algún elemento de la geometría. Sí se permiten, sin embargo, fuentes de luz móviles.

La serialización mencionada anteriormente es necesaria para poder copiar la estructura completa a OpenCL en forma de array. Para compartir entre la memoria de CPU y GPU listas enlazadas (o, para ser más exactos, entre memoria del host y del dispositivo) sería necesario OpenCL 2.0, que proporciona Shared Virtual Memory (SVM). Puesto que hasta ahora no había ningún motivo para esa exigencia, opto por no añadirla ahora. Además, el recorrido de listas enlazadas puede suponer una merma importante del rendimiento con respecto a recorrer un *array*, al provocar un fallo de caché cada salto de nodo.

Para la creación del BVH introduzco los siguientes cambios en el código:

- Añado la clase **BVH**: (bvh.hpp bvh.cpp) Código para generar la estructura en árbol y más tarde «aplanarla»
- Añado la clase **Bbox** (bbox.hpp): Métodos y propiedades de un objeto tipo caja envolvente.
- Modifico el *kernel* “kernels/render.cl” para que pueda recorrer la estructura generada en lugar de comprobar todos los triángulos.

De nuevo me baso en los algoritmos de «Physically Based Rendering» [PHH00] para la primera aproximación al problema, aunque valoro modificar el tipo de estructura o los algoritmos en refinamientos posteriores.

Realizo varias modificaciones sobre lo propuesto en el libro. Como no dispongo de código específico para gestionar la reserva de memoria, utilizo `make_unique` para reservar la memoria en punteros únicos. Además, cambio el punto en el que se realiza dicha reserva. Sigo la norma de que sea el propietario de un objeto dinámico el que reserve y libere su memoria. Por lo tanto, primero reservo memoria para los dos nodos hijos y más tarde los relleno. Otro problema con el que me encuentro es que en el paso de la estructura resultante del *host* al contexto de OpenCL se corrompen los datos. Finalmente cambio las estructuras para que se usen datos específicos de OpenCL, como `cl_float3` para vectores de tres componentes de coma flotante. Descubro más tarde que en el *host* `cl_float3` es un alias a `cl_float4` y que siempre se usan cuatro componentes en la implementación con la que estoy trabajando, lo que explica el problema sufrido.

En esta fase también decido que para facilitar las pruebas sería conveniente poder generar archivos que contengan configuración y descripción de la escena a mostrar. Genero un tipo de archivo muy simple en el que se define la resolución de la ventana, las propiedades de la cámara, puntos de luz, nombre y ruta de los shaders, modelo y

kernel de OpenCL a cargar, etc. Lo implemento en las clases **Config**(config.hpp) y **ConfigLoader**(configloader.hpp, configloader.cpp). El formato de este archivo está descrito en el anexo 11.4.

5.5.2 Estudio de rendimiento y mejora obtenida al introducir BVH

Para estudiar el efecto de introducir la estructura de aceleración BVH realizo varias mediciones antes y después de su implementación. Una vez más, utilizo el modelo «Sponza» a una resolución de 1024x768 píxeles, esta vez con un único punto de luz.

Resulta interesante tener en cuenta el número de rayos por segundo que el programa está siendo capaz de evaluar. Para añadir el efecto de sombras no suavizadas se usa 1 rayo por píxel (*rpp*) y punto de luz, que va desde el píxel hasta el punto de luz y se debe comprobar si choca con alguno de los triángulos de la escena. Puesto que la resolución de esta prueba ha sido de 1024x768 y un sólo punto de luz, para la generación de cada imagen es necesario evaluar 786.432 rayos.

Las mediciones realizadas antes de incluir BVH dan como resultado una media de 4500ms en producir cada imagen, lo que equivale a 0,22fps, o 17.000 rayos por segundo.

En las mediciones en las mismas condiciones después de implementar BVH se necesitan unos 9,26ms para generar cada imagen, equivalente a 108fps y 84.930.000 rayos por segundo. Por lo tanto, introducir BVH ha hecho que la generación de imágenes sea aproximadamente **500 veces** más rápida.

Realizo varias pruebas para evaluar la variación del rendimiento de la estructura BVH en función del número máximo de primitivas que almacena cada nodo hoja. En la implementación tomada como referencia del libro «Physically Based Rendering» [PHH00] en cada nodo hoja se guarda una única primitiva. Realizo el experimento con varios valores diferentes para este parámetro, con el mismo modelo, con 18 puntos de luz y una resolución de 1280x720 píxeles:

Primitivas	fps
1	4,41
2	4,64
3	4,66
4	4,69
5	4,53
50	3,16
500	1,13

Tabla 5: Rendimiento de la estructura BVH

Siendo, en la tabla 5, «Primitivas» el número máximo de primitivas por nodo hoja y «fps» el número de imágenes generadas por segundo.

Al aumentar el número de primitivas por nodo hoja se reduce el número de nodos y niveles del árbol BVH, por lo que su estructura ocupa menos memoria y cuesta menos recorrerla. Si el número de primitivas no es muy alto, también se almacenan en posiciones contiguas de memoria primitivas que, con mucha probabilidad, van a tener que comprobarse en su conjunto en caso de que se tenga que comprobar una de ellas, ya que están muy próximas en su localización espacial. Sin embargo, al aumentar aún más el número de primitivas por nodo hoja y reducir el número de nodos BVH a recorrer, se realizan más comprobaciones rayo-tríángulo, siendo el caso límite la situación anterior a implementar BVH.

Por lo tanto, después de valorar los resultados, decidí que cada nodo hoja almacenase 4 primitivas

Para realizar una medición del número de rayos generados y evaluados por segundo usando BVH, ejecuté en varias ocasiones el programa eliminando el cálculo de componente difuso y especular con el modelo «Sponza», que consta de 262.267 triángulos a una resolución de 1024x768 píxeles. En la tabla 6 se encuentra el resultado. La columna «rpp» indica el número de rayos que ha sido necesario generar por píxel de la escena. En este caso, que se usan para obtener sombras no suavizadas, por cada píxel es necesario un rayo por punto de luz. Multiplicando «rpp» por la resolución obtenemos en número de rayos por imagen (*frame*). Para que las cantidades sean manejables he usado como unidad el millón de rayos o Mega-Ray (10⁶). Las siguientes dos columnas tienen en cuenta únicamente el tiempo de ejecución del *kernel* de OpenCL y miden su rendimiento. Ese sería el número de rayos por segundo si no se tuviese que hacer nada más para la construcción de cada imagen. Aunque esta es una buena evaluación del *kernel*, tomo medidas de los

frames por segundo (fps) reales, para obtener el número de rayos por segundo que se están generando en realidad.

GEOMETRÍA			KERNEL		TOTAL ESCENA	
Luces	rpp	Mrayos/frame	ms ejecución	Mrayos/s	fps	Mrayos/s
1	1	0,79	9,88	79,60	94,7	74,46
9	9	7,08	85,29	82,99	10,9	76,93
18	18	14,16	172,56	82,03	5,4	76,72

Tabla 6: Millones de rayos por segundo evaluados usando BVH

De los resultados se extrae que aproximadamente el 93% del tiempo necesario para generar una imagen es consumido por el *kernel* de OpenCL, dedicándose éste para esta prueba únicamente a comprobar la visibilidad de puntos de luz desde cada uno de los píxeles de la escena.

Pese a lo impresionante de los números y la mejora obtenidos, el recorrido de la estructura BVH presenta un problema al ejecutarse con alto grado de paralelismo. Los diferentes hilos de un *wavefront* se ejecutan anclados paso por paso. Si hay código que se ejecuta de forma condicional y sólo uno de los hilos cumple la condición, los demás hilos esperan hasta que todos vuelven a llegar al mismo punto de la función. Por lo tanto, cuando se está recorriendo el árbol y un hilo encuentra que tiene que comprobar los triángulos de un nodo hoja, todos los demás que no estén realizando la misma comprobación están desactivados, sin realizar ningún cálculo. Este efecto se conoce como divergencia en la ejecución.

Para intentar reducir la divergencia, implemento una modificación en el la función del *kernel* «*intersects*», de forma que primero se recorre todo el árbol comprobando qué nodos hoja chocan con el rayo a comprobar y se guardan los resultados en una reserva de memoria. En una segunda fase se recorre el listado de nodos hoja almacenado y se realizan todas las comprobaciones rayo-triángulo con las primitivas que contienen éstos. En la nueva implementación se evitan muchos saltos condicionales y se reduce la divergencia. El resultado, sin embargo, es algo más lento que la implementación anterior. La explicación es que, en la nueva implementación, se pierde la oportunidad de dejar de comprobar una rama del árbol tan pronto como se encuentre la primera intersección rayo-triángulo. El algoritmo es mucho más predecible, pero también realiza mucho más trabajo. Esta implementación sería adecuada para reflexiones y refracciones, ya que hay que encontrar el triángulo más cercano a un punto y por lo tanto hay que comprobar toda la geometría, pero no para sombras, ya que en este caso es suficiente con saber si hay alguna intersección,

aunque no sea la más cercana y se puede finalizar la comprobación tan pronto como se encuentre una coincidencia.

Añado posteriormente una última optimización para evitar tener que recorrer la estructura BVH en ciertos casos en los que no va a tener un resultado apreciable. En caso de que las sombras estén habilitadas, si el aporte de un punto de luz es menor que cierto valor, descarto ese punto de luz. Experimentalmente ajusto ese valor a 0,02. Es decir, si se va a aportar menos de un 2% de la intensidad de la fuente de luz, la descarto. Es la comprobación del *kernel*:

```
if (attenuation < ATTENUATION_SENSITIVITY && shadows_enabled)
    continue;
```

5.6 Iteración 5. Técnica híbrida sin OpenCL

5.6.1 Implementación

Como prueba de concepto he portado la funcionalidad realizada por el *kernel* de OpenCL «render.cl» a C++. La intención es comparar el rendimiento del código programado en OpenCL y ejecutado en una GPU con código en C++ ejecutado en CPU.

Las texturas del G-Buffer se copian de la tarjeta gráfica al *host* y en éste, usando C++, se realizan los cálculos de iluminación y sombras. Finalmente, la imagen generada se vuelve a copiar a la tarjeta gráfica para que se vuelque a la pantalla.

Estas operaciones se realizan la clase HybridShaderCPU, añadida al proyecto en este punto. A partir de este momento hay tres métodos de renderizado posibles en el programa: Sombreado diferido, técnica híbrida con OpenCL en GPU y técnica híbrida en C++ en CPU.

Para poder llevar los datos del G-Buffer de OpenGL a la CPU utilice la función `glReadpixels`, que lee uno de los anexos de color del *framebuffer*.

También uso OpenMP, descrito en el capítulo 3.5, para que trabajen todos los núcleos disponibles de la CPU, ejecutando de forma paralela el bucle que recorre los los diferentes píxeles que componen una línea de la imagen final. Para usar OpenMP añado la directiva `#pragma omp parallel for schedule(dynamic)` antes del bucle que recorre cada línea de la imagen a generar.

Añado la misma optimización que en la implementación en OpenCL para descartar los cálculos de fuentes de luz que estén atenuadas debido a su distancia más allá de un umbral, que fijo en un 2%.

5.6.2 Comparativa de rendimiento con la Iteración 4

La técnica híbrida implementada en CPU está claramente en desventaja respecto a la versión en OpenCL y GPU. El primer motivo es la necesidad de copiar las texturas del G-Buffer y la imagen generada entre la CPU y la GPU en cada *frame*. Esta copia, además, tiene que hacerse por el bus PCI-E, que es más lento de lo que sería la copia dentro de la tarjeta gráfica, en el caso de que no lo evitásemos gracias a la extensión `cl_khr_g1_sharing`.

Por otro lado, la CPU utilizada, Intel Core i5-4670 a 3,4GHz, tiene 4 núcleos y 85,57 GFLOPS, según [OCA0000] mientras que la GPU tiene 3.290 GFLOPS según [TPU0000], lo que supone una capacidad de cálculo casi 40 veces superior.

Estos dos factores explican el resultado obtenido, independientemente de que se pudiese conseguir un rendimiento algo mayor en la CPU con las optimizaciones adecuadas.

LUCES	FACTOR	FPS GPU (OpenCL)	FPS CPU (OpenMP)
1	28,71	297,3	10,4
3	23,17	157,4	6,8
6	40,01	113,1	2,8
9	25,40	51,3	2,0
12	27,61	36,3	1,3
15	30,41	30,1	1,0
18	30,46	22,8	0,7

Tabla 7: Comparación rendimiento ejecución en GPU con ejecución en CPU

En la tabla 7 muestro de forma resumida las mediciones realizadas, indicando en la columna «FACTOR» el número de veces que es más rápida la implementación en OpenCL y ejecutada en GPU respecto de la implementación en C++ ejecutada en CPU.

5.7 Diagrama de clases

En la figura 24 se muestra el diagrama UML de clases completo del programa. Se incluyen relaciones de dependencia, agregación, composición y herencia.

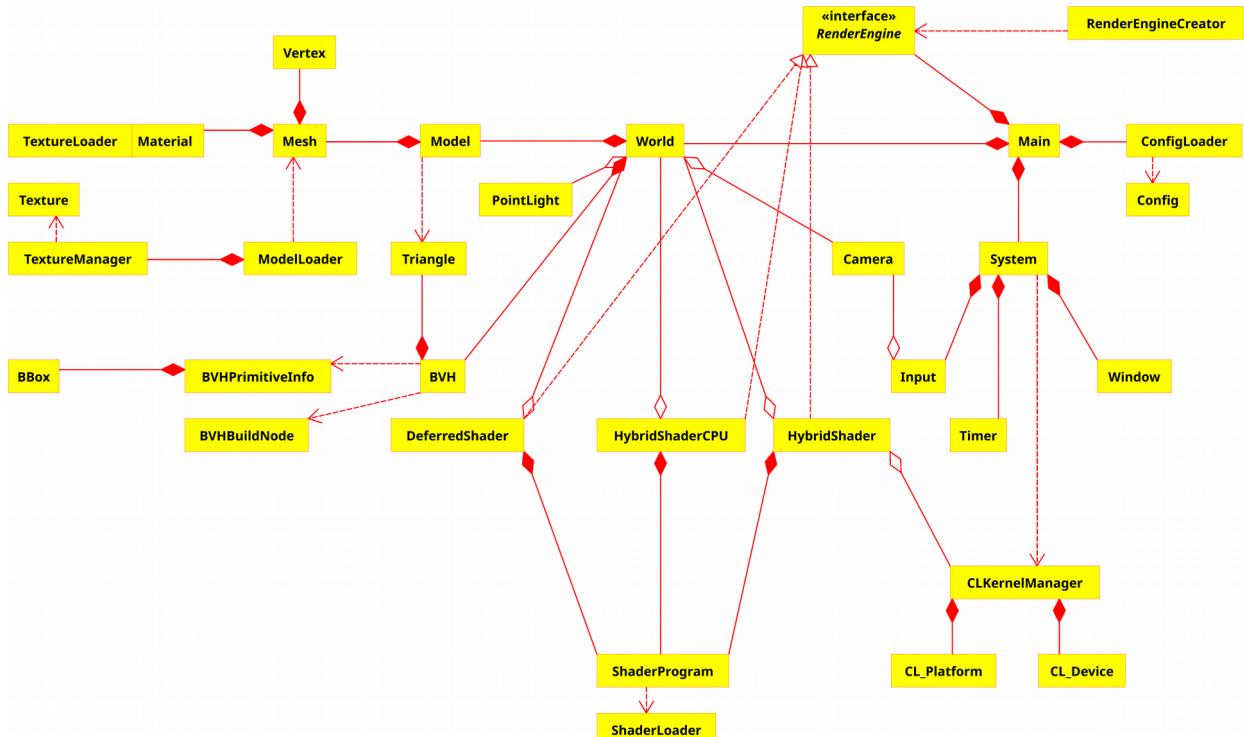


Figura 24: Diagrama UML de clases completo

En el diagrama se utiliza la notación estándar UML, en la que se representan las relaciones de la siguiente forma:

- Herencia: Línea de trazo discontinuo acabada en un triángulo sin relleno que va de la subclase a la superclase.
- Agregación: Línea de trazo continuo que comienza con un rombo vacío en el objeto que contenedor.
- Composición: Línea de trazo continuo que comienza con un rombo relleno en el objeto contenedor.
- Dependencia: Línea de trazo discontinuo que acaba en una flecha y que parte del objeto dependiente.

6 Comparativa de la técnica híbrida frente a rasterización y trazado de rayos

6.1 Comparación de rendimiento

El estudio de rendimiento ha sido realizado en el equipo utilizado para el desarrollo del proyecto, descrito en el apartado 3.2 . Repito aquí sus características:

- CPU: Intel Core i5-4670 (3.4 GHz, 4 núcleos, caché L1 256kB, L2 1MB, L3 6MB)
- Placa base: Gigabyte GA-H87-HD3
- Memoria RAM: 2x8GB Kingston HyperX blu 1600MHz
- Tarjeta gráfica: Sapphire Dual-X R9 285 2GB GDDR5
- Disco duro: Samsung EVO 840 SSD 120GB

El procedimiento seguido para obtener los datos de rendimiento ha consistido en tomar cinco mediciones por cada dato a recabar, eliminar las mediciones mayor y menor y obtener la media de las tres restantes. El objetivo es obtener datos representativos del rendimiento y eliminar anomalías estadísticas.

El modelo utilizado es «The Atrium Sponza Palace, Dubrovnik», en su versión de la empresa Crytek, modificado por Morgan McGuire [MAC000]. Este modelo está compuesto por 262.267 triángulos.

Las mediciones se han realizado a una resolución de 1024x768 píxeles, con la sincronización de refresco del monitor deshabilitado para evitar falsear el resultado debido al límite de 60Hz del monitor utilizado.

La tabla 8 muestra el resultado de las mediciones realizadas. La columna «Luces» indica el número de fuentes de luz presentes en la escena. La columna «Método» indica la técnica de renderizado utilizada: OpenCL para la técnica híbrida en tarjeta gráfica desarrollada en el capítulo 5.4, OpenMP para la técnica híbrida en C++ ejecutándose en CPU del capítulo 5.6 y «Diferido» para la técnica de sombreado diferido desarrollada en el capítulo 5.3.

La columna «FPS» indica el número de imágenes generadas por segundo y las tres columnas siguientes expresan el tiempo, en milisegundos, que se tardó en generar

una imagen durante esa ejecución del programa: «T_min» es el tiempo mínimo, «T_medio» es promedio de todas las imágenes generadas en esa ejecución y «T_max» el tiempo máximo.

Las mediciones se realizaron en primer lugar con el cálculo de sombras activo, excepto en el caso de la técnica sombreado diferido, puesto que no implementa esa funcionalidad. Posteriormente se repitieron las mediciones sin utilizar la técnica de trazado de rayos para generar sombras.

LUCES	MÉTODO	SOMBRA ACTIVAS				SOMBRA INACTIVAS			
		FPS	T_min	T_medio	T_max	FPS	T_min	T_medio	T_max
1	OpenCL	297,3	1,0	3,4	11,3	541,3	1,0	1,8	4,7
	OpenMP	10,4	51,3	96,6	170,0	27,4	35,0	36,5	61,7
	Diferido	-	-	-	-	995,9	1,0	1,0	3,7
3	OpenCL	157,4	3,0	6,4	16,7	515,9	1,0	1,9	4,0
	OpenMP	6,8	43,7	147,2	393,3	23,3	39,7	42,9	85,7
	Diferido	-	-	-	-	997,1	1,0	1,0	3,3
6	OpenCL	113,1	3,7	8,8	27,0	515,3	1,0	1,9	5,3
	OpenMP	2,8	200,0	353,9	614,0	20,7	47,0	48,3	75,5
	Diferido	-	-	-	-	984,1	1,0	1,0	4,3
9	OpenCL	51,3	10,3	19,5	34,7	503,6	1,0	2,0	3,7
	OpenMP	2,0	328,3	495,2	746,0	17,4	54,0	57,3	90,7
	Diferido	-	-	-	-	980,1	1,0	1,0	5,0
12	OpenCL	36,3	12,3	27,5	46,7	494,0	2,0	2,0	4,7
	OpenMP	1,3	588,7	759,6	946,0	15,5	62,0	64,6	100,7
	Diferido	-	-	-	-	962,4	1,0	1,0	6,0
15	OpenCL	30,1	16,7	33,3	61,3	491,5	2,0	2,0	4,3
	OpenMP	1,0	792,3	1011,7	1284,3	13,7	71,0	72,9	129,0
	Diferido	-	-	-	-	960,3	1,0	1,0	5,3
18	OpenCL	22,8	25,0	43,8	69,3	485,5	2,0	2,1	4,3
	OpenMP	0,7	1016,0	1334,1	1563,7	12,2	79,0	82,0	117,0
	Diferido	-	-	-	-	935,9	1,0	1,1	6,0

Tabla 8: Conjunto de mediciones de las diferentes técnicas

La comparación que resulta más importante para el objetivo del presente trabajo es la que tiene en cuenta el rendimiento del sombreado diferido sin realizar cálculos de sombras, y la de la técnica híbrida con OpenCL con sombras activas. De esta comparación se puede extraer el coste que supone la inclusión de un efecto calculado con trazado de rayos, como son las sombras, en la generación de cada imagen. Extrayendo únicamente esos datos de la tabla anterior, y calculando el número de veces que es más rápido el sombreado diferido que la técnica híbrida (columna «Factor»), se obtiene el resultado de la tabla número 9:

De dicha tabla se concluye que la técnica híbrida es entre dos y tres veces más lenta que el sombreado diferido **por cada punto de luz** añadido. Entre 6 y 9 puntos de luz se consiguen aproximadamente los 60fps que hemos considerado como requerimiento para que la animación generada proporcione sensación de fluidez, pero con 12 puntos de luz los *frames* por segundo ya caen casi a la mitad de lo deseado.

N.º Luces	Factor	Técnica híbrida (fps)	Sombreado diferido (fps)
1	3,3	297,3	995,9
3	6,3	157,4	997,1
6	8,7	113,1	984,1
9	19,1	51,3	980,1
12	26,5	36,3	962,4
15	31,9	30,1	960,3
18	41,0	22,8	935,9

Tabla 9: Comparación fps técnica híbrida y sombreado diferido

Por lo tanto, parece deseable aprovechar la generación de sombras únicamente en casos en los que el resultado sea visualmente atractivo. Con este dato en mente, podría ser interesante la opción de activar selectivamente la generación de sombras por punto de luz. De esta forma se podría evitar el coste computacional de calcular sombras de puntos de luz con una atenuación muy rápida y que no aporten nada visualmente a la escena.

Aunque el rendimiento de la técnica híbrida sea muy inferior al sombreado diferido, su utilización nos permite, por ejemplo, añadir un punto de luz que simule la luz solar y generar sombras realistas a casi 300 fps.

El renderizado mediante trazado de rayos de la misma escena con un punto de luz utilizando Blender 2.78c en el mismo *hardware* tardó 25 minutos, lo que lo descarta completamente como método interactivo. Por lo tanto, no se han tomado más mediciones de rendimiento para esta técnica.

La figura 25 muestra la representación gráfica de la tabla 8. En ella «S» significa con cálculo de sombras mediante trazado de rayos y «no-S» con el cálculo desactivado.

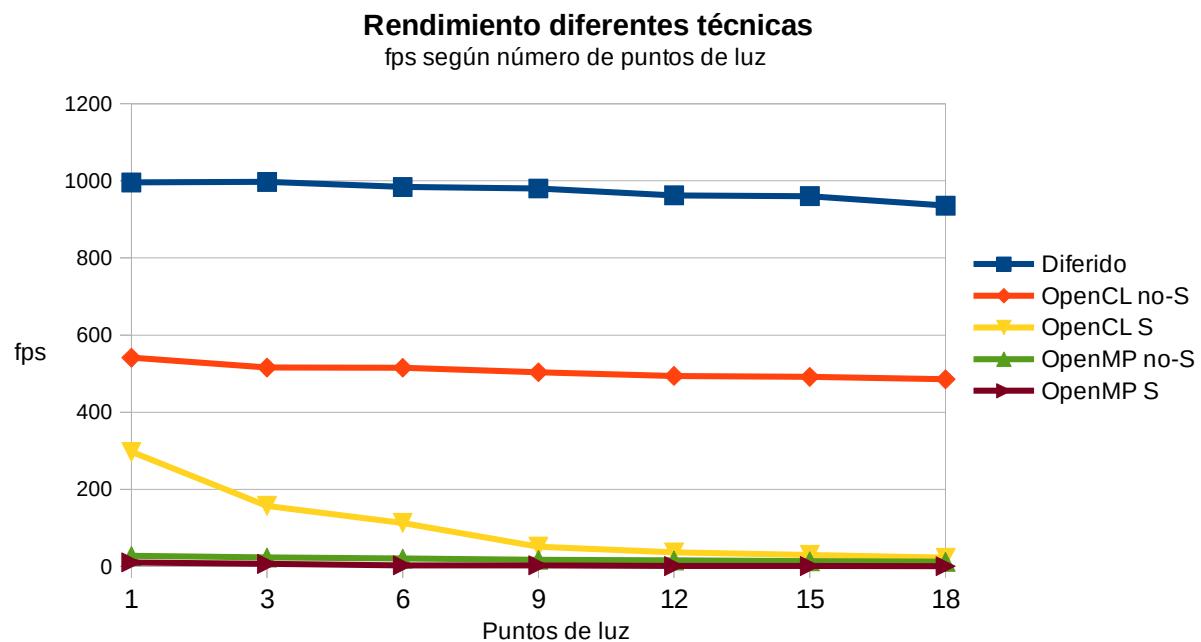


Figura 25: Gráfica de comparación de rendimiento de diferentes técnicas

Por último, cabe mencionar que la técnica híbrida en CPU proporciona un rendimiento tan bajo que sólo es útil como prueba de concepto, o quizás para prototipado rápido de nuevas funcionalidades.

6.2 Calidad de la imagen obtenida

A continuación se muestra una comparación de la calidad de imagen obtenida utilizando el trazado de rayos, el sombreado diferido y la técnica híbrida.

Las imágenes se han generado utilizando el modelo descrito en el apartado anterior, a una resolución de 1920x1080 píxeles, que corresponde con la resolución estándar FullHD de los televisores actuales. La escena está iluminada únicamente mediante un punto de luz en la parte superior a suficiente distancia, imitando la luz solar. No hay fuentes de luz con atenuación, puesto que el resultado es muy similar en las tres técnicas y su cálculo no presenta dificultad. En el caso del trazado de rayos, se le ha dado al punto de luz un tamaño pequeño, aunque no despreciable, por lo que se comporta como una superficie emisora. De esta forma se producen algunas sombras con bordes suaves. Esta opción no está implementada en las técnicas en tiempo real debido al coste computacional que implica el muestreo de las superficies emisoras de luz.

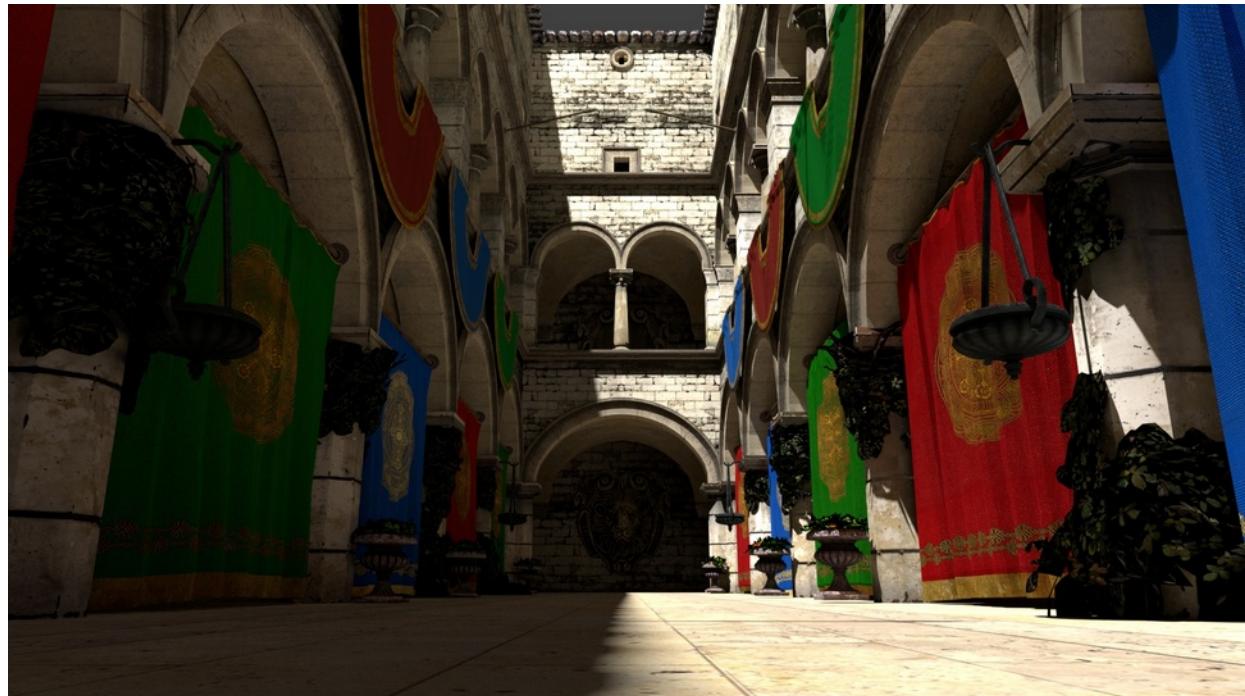


Figura 26: Imagen trazada usando «Cycles Renderer»

La imagen tomada como referencia para el trazado de rayos se obtuvo importando el modelo en el programa Blender [BLEN000] , convirtiendo las texturas mediante el add-

on «Material, Materials Utils Specials» y trazando finalmente la escena mediante «Cycles Renderer» [CYCL000]. La generación de la imagen tardó 1h 05min. El resultado se puede observar en la figura 26.

Para obtener la imagen generada con la técnica híbrida se utilizó el programa desarrollado en el presente proyecto. Puesto que únicamente está implementada la iluminación directa, se añadió un factor de iluminación ambiente difusa constante para toda la escena para emular la iluminación global. Ese es el motivo por el que las zonas en sombra no se muestran completamente negras. El resultado se puede observar en la figura 27.



Figura 27: Imagen generada mediante la técnica híbrida

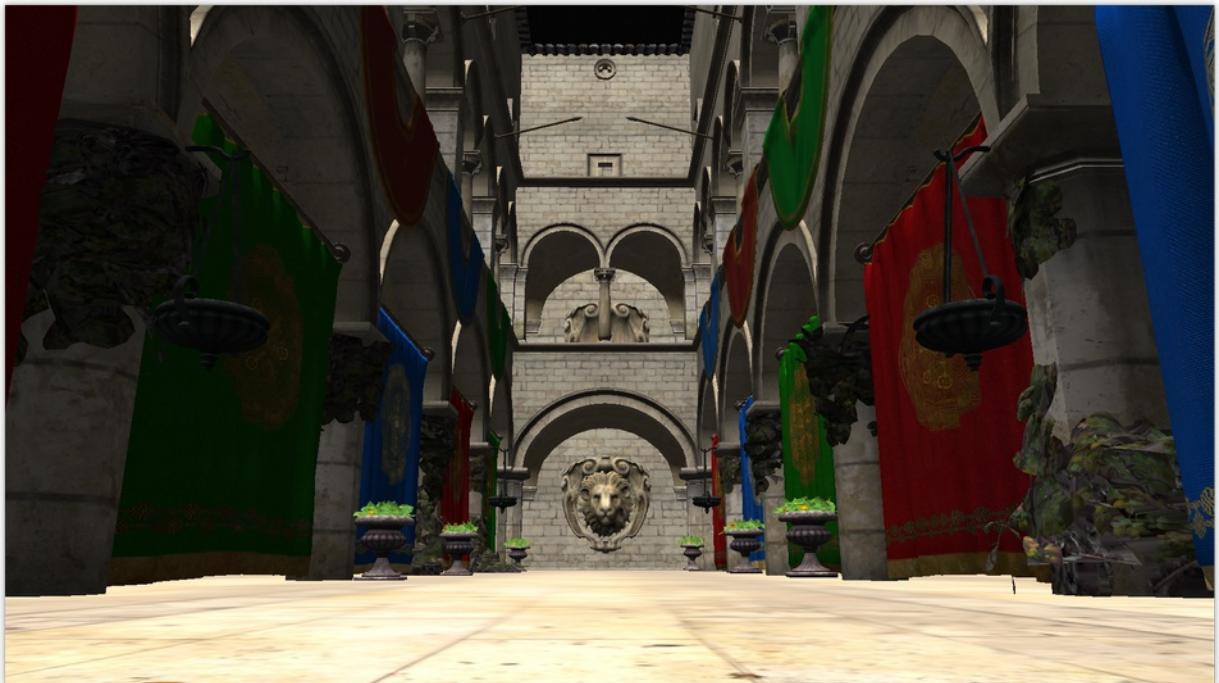


Figura 28: Imagen generada con sombreado diferido

Finalmente, en la figura 28 se observa el resultado obtenido usando sombreado diferido. En este caso no hay sombras proyectadas, por lo que únicamente se toma en cuenta para calcular la mayor o menor iluminación de un punto la normal al foco de luz y la iluminación ambiente.

Resulta interesante comparar los detalles resaltados en la figura 29. En esta figura se han resaltado tres zonas, numeradas del 1 al 3 y delimitadas por rectángulos de trazo discontinuo.

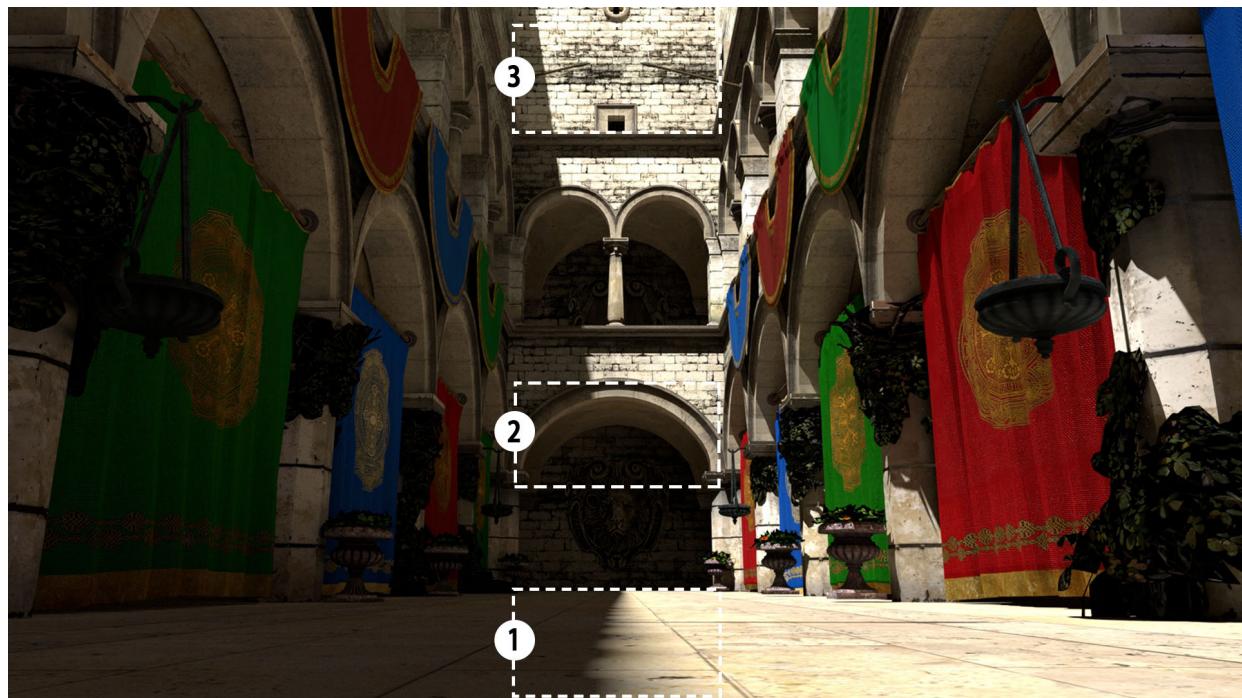


Figura 29: Resultado de detalles

En el detalle **número 1**, mostrado de forma ampliada en la figura 30, se muestra el resultado de las sombras proyectadas por el tejado del edificio sobre el suelo del patio. El trazado de rayos produce un resultado mucho más suave y realista, mientras que la técnica híbrida, calculada con 1 rayo por píxel y sin efecto *antialias*, genera bordes abruptos y *pixelados*.



Trazado de rayos



Técnica híbrida



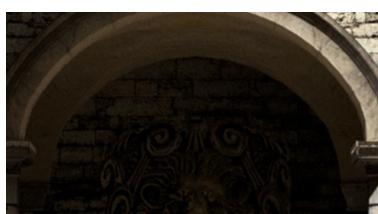
Rasterización

Figura 30: Detalles de la proyección de sombras

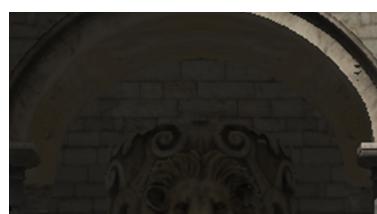
En este detalle la comparación más útil es la que enfrenta el resultado generado por un trazador de rayos *off-line* con la técnica híbrida en tiempo real. Aunque existen técnicas para la generación de sombras proyectadas mediante rasterización como el *shadow mapping*, la implementación de técnicas de rasterización que imiten el trazado

de rayos se aleja del objetivo del presente proyecto, en el que la rasterización se utiliza únicamente para sustituir a los rayos primarios.

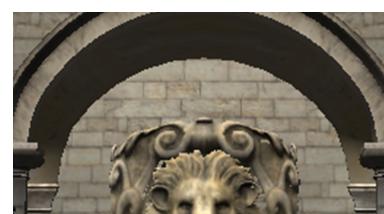
En el detalle **número 2**, ampliado en la figura 31, se puede observar el efecto de la iluminación global. Mientras que en el trazado de rayos hay diferentes grados de iluminación debido a la iluminación indirecta, es decir, a la contribución de cada objeto de la escena a la iluminación del resto, en la técnica híbrida hay únicamente zonas en sombra, con iluminación constante ambiente, o iluminadas por el punto de luz. En el detalle tomado de la imagen generada por rasterización no hay zonas en sombra. La mayor o menor iluminación depende únicamente, de nuevo, del ángulo formado por la normal de la superficie iluminada con la dirección de la luz incidente. Tanto la técnica híbrida como la rasterización producen un efecto más «plano».



Trazado de rayos



Técnica híbrida



Rasterización

Figura 31: Detalle de la iluminación indirecta

Finalmente, en el **detalle 3**, se observa que los bloques de mampostería poseen un relieve mucho más realista en la imagen obtenida por trazado de rayos. En este caso no debe achacarse la diferencia a limitaciones de las técnicas utilizadas, sino a la implementación desarrollada en el presente proyecto, puesto que no se ha incluido código para aplicar «*bump mapping*».



Trazado de rayos



Técnica híbrida



Rasterización

Figura 32: Bump mapping

7 Conclusiones

Con el desarrollo del presente proyecto se ha comprobado la viabilidad del uso de la técnica híbrida de generación de imágenes en tiempo real a partir de modelos tridimensionales. El rendimiento de la implementación realizada y del hardware utilizado permite calcular en un modelo de más de 250.000 triángulos hasta 6-9 rayos por píxel sin que se pierda la sensación de fluidez. De esta forma, y según los costes en rpp explicados en el capítulo 4.2, sería posible añadir un número limitado de puntos de luz y reflejos/refracciones con un número aceptable de rebotes.

Sin embargo, con el rendimiento actual no sería posible, en principio, añadir otros efectos más costosos como sombras suaves, superficies de luz, oclusión ambiental o iluminación global, que dotan de un gran realismo a la imagen generada. La inclusión de estos efectos requeriría una mayor optimización y el estudio de técnicas alternativas.

La programación paralela mediante GPGPU supone un gran avance ya que, como se vio en el capítulo 5.6.2, proporciona una potencia de cálculo muy superior al uso de CPU en ciertos algoritmos. Únicamente gracias a la utilización de computación paralela en la tarjeta gráfica ha sido posible alcanzar un rendimiento suficiente para la utilización de forma interactiva de la técnica híbrida.

Durante el desarrollo de este proyecto ha sido necesario estudiar en profundidad y comprender OpenCL, puesto que esta potencia de cálculo no es accesible de forma automática para el programador. Para utilizarla es necesario conocer la arquitectura para la que se está programando y diseñar los algoritmos cuidadosamente para no desaprovechar los recursos. Además, el uso de algoritmos que de forma histórica se han comportado muy bien en CPUs y ejecutándose con pocos hilos no siempre es lo más adecuado para su implementación en plataformas masivamente paralelas. El recorrido de la estructura de aceleración BVH, por ejemplo, posee una alta divergencia inherente. Mientras en [KTE0000] se estudia como paliar este problema usando códigos Morton, otros autores proponen soluciones diferentes pensadas desde cero para su implementación paralela, como el uso de *Brick Maps* en [SWOB000] o un algoritmo de trazado de rayos paralelo divide y vencerás en [RAVI000].

La compartición de datos entre OpenGL y OpenCL también impone un coste no desdenable. Sin embargo, el hardware de las GPU actuales está diseñado y

optimizado para la rasterización, por lo que su uso para el cálculo de rayos primarios brinda más ventajas que inconvenientes.

La computación paralela, impulsada por los recientes avances tanto en *hardware* como en interfaces de programación de aplicaciones, abre la puerta a nuevas técnicas y genera la necesidad de buscar algoritmos que aprovechen sus características al máximo. Una de estas nuevas opciones es la técnica híbrida entre rasterización y trazado de rayos que, como se ha mostrado en el presente proyecto, es viable y capaz de producir resultados en tiempo real.

8 Planificación y presupuesto

A continuación realizo una estimación del coste total de desarrollo del presente proyecto, en el que incluyo el coste de adquisición de materiales necesarios y el de mano de obra.

En el presupuesto del material necesario para la ejecución del presente proyecto he incluido el ordenador utilizado para el desarrollo del programa, la realización de las pruebas de rendimiento y la confección de la documentación, así como los manuales y bibliografía específicos adquiridos con la finalidad de servir de material de estudio y consulta.

El ordenador utilizado fue adquirido por componentes. En la tabla siguiente detallo el coste de cada uno de los componentes, así como el importe total de adquisición:

COMPONENTE	PRECIO (€)
CPU: Intel Core i5-4670	185
Placa base Gigabyte GA-H87-HD3	88,55
Memoria RAM: 2x8GB Kingston HyperX blu	136,62
Tarjeta gráfica: Sapphire Dual-X R9 285 2GB	214,49
Disco duro: Samsung EVO 840 SSD 120GB	85,14
Caja ATX	57,32
Fuente de alimentación OCZ 600w	56,9
Monitor, teclado, ratón	125
TOTAL	949,02

Tabla 10: Coste desglosado del ordenador utilizado

El total del coste de adquisición de los libros utilizados fue de 264,69€, repartido de la siguiente forma:

- [STR00] A Tour of C++ 28,03€
- [STR00] Effective Modern C++ 27,99€
- [KMS00] Heterogeneous Computing with OpenCL 2.0 53,95€
- [SUF00] Ray Tracing from the Ground Up 76,63€
- [PHH00] Physically Based Rendering 78,09€

El resto de libros utilizados han sido consultados de la biblioteca o bien ya formaban parte de mi colección anteriormente a empezar este trabajo.

Por lo tanto, se puede establecer un coste total de los materiales de 1213,71€

Incluyo a continuación, en la tabla 11, un detalle de las actividades realizadas durante la ejecución del presente proyecto, así como sus fechas de inicio y duración en horas. Es necesario reseñar que el desarrollo del presente trabajo comenzó en el verano de 2015, siendo interrumpido durante el curso 2015-2016 para finalizar las asignaturas pendientes y retomado en septiembre de 2016.

Nombre	Horas	Inicio	Descripción
Tarea 1	20	01/06/15	Estudio de C++, lectura de A Tour of C++
Tarea 2	40	22/06/15	Estudio de OpenGL. Curso EDX CSE167x Computer Graphics
Tarea 3	5	13/07/15	Preparación del entorno de programación C++ y OpenGL
Tarea 4	40	15/07/15	Desarrollo de la iteración 1 – Sombreado directo
Tarea 5	30	26/08/15	Desarrollo de la iteración 2 – Sombreado diferido
Tarea 6	40	08/09/15	Estudio de OpenCL, libro Heterogeneous Computing with OpenCL
Tarea 7	10	20/10/15	Implementación del movimiento de la cámara
Tarea 8	30	24/10/16	Desarrollo de la iteración 3-a – Compartición datos OpenGL- OpenCL
Tarea 9	40	01/11/16	Estudio del libro Ray Tracing From the ground Up
Tarea 10	20	15/12/16	Estudio del libro Physically Based Rendering
Tarea 11	20	02/01/17	Desarrollo de la iteración 3-b – Sombras
Tarea 12	10	16/01/17	Estudio de estructuras aceleradoras
Tarea 13	20	23/01/17	Desarrollo de la iteración 4 – BVH
Tarea 14	10	06/02/17	Implementación del fichero de configuración y carga de escenas
Tarea 15	8	09/02/17	Desarrollo de la iteración 5 – C++ y CPU
Tarea 16	7	20/02/17	Pruebas y corrección de errores
Tarea 17	10	27/02/17	Ajuste tamaño del G-Buffer
Tarea 18	60	15/03/17	Mediciones de rendimiento, depuración, pruebas y optimización
Tarea 19	120	23/01/17	Redacción de la memoria

Tabla 11: Tareas realizadas, fechas y tiempo invertido

Horas totales: 540 h

Incluyo a continuación una representación en diagramas de Gantt de las tareas realizadas durante los dos períodos, en las figuras 33 y 34.

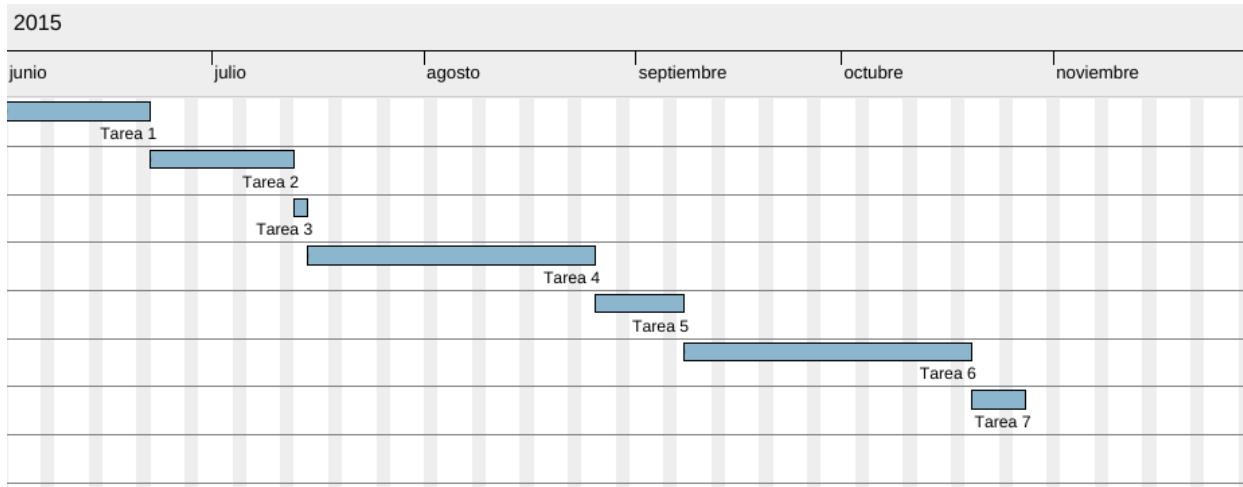


Figura 33: Diagrama de Gantt del primer período de desarrollo

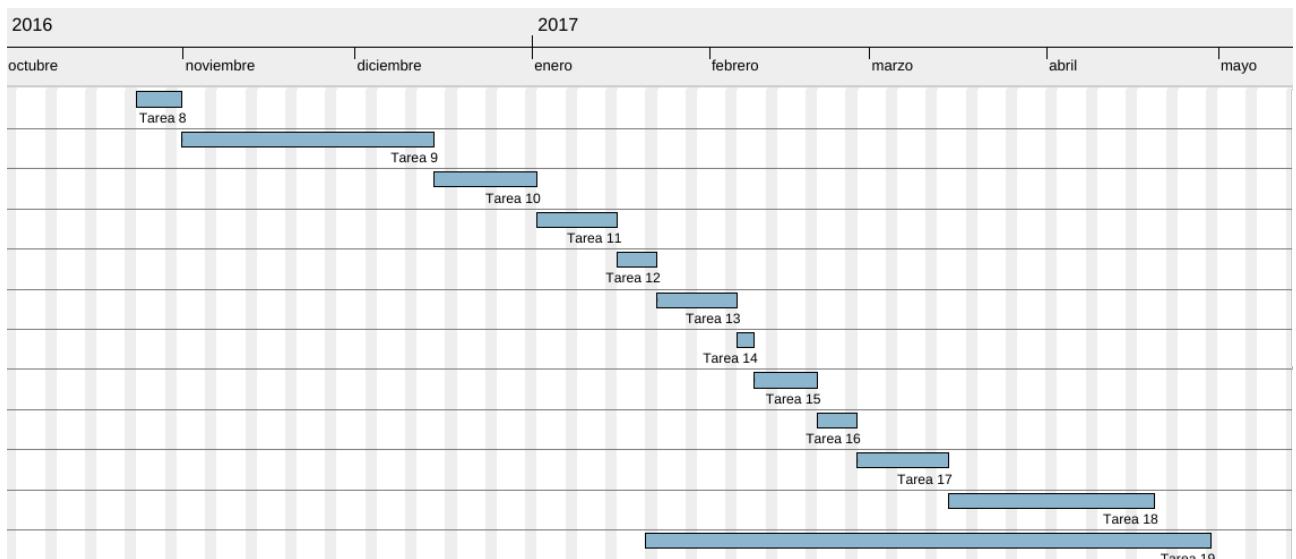


Figura 34: Diagrama de Gantt del segundo período de desarrollo

Para calcular el coste de mano de obra he supuesto un coste de programador-analista de 20€ / hora. He incluido todo el tiempo dedicado al proyecto, incluso el de documentación y formación. Por lo tanto, el coste de mano de obra sería de 20 €/h x 540 h= 10.800€

Teniendo en cuenta el coste de materiales más del de mano de obra, el **coste total** del proyecto es de 1.213,71€ + 10.800€ = **12.013,71€**.

9 Bibliografía

ACT000: Data Oriented Design and C++. Acton, Mike, CPPCON , 2014,
<https://www.youtube.com/watch?v=rX0ItVEVjHc>

AMD0001: OpenCL™ Optimization Case Study: Simple Reductions. Catanzaro, Bryan, AMD Developer Central, 2010, <http://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-simple-reductions/>

ARNO000: Arnold. Varios autores, Solid Angle, S.L., 2017,
<http://www.solidangle.com/arnold/>

ASSI00: ASSIMP, Open Asset Import Library . Varios autores, Assimp Development Team, 2017, <http://assimp.org>

BBH000: OpenCL Programming by Example. Bhattacharyya,Koushik. Banger, Ravishekhar., Packt Publishing, 2013. ISBN:9781849692342,
<https://www.packtpub.com/application-development/opencl-programming-example>

BLEN000: Blender, software libre modelador 3D. Varios autores, Blender Foundation, 2017, <https://www.blender.org/>

BRIG000: Brigade. OTOY Inc., OTOY Inc. , 2017, <https://home.otoy.com/render/brigade/>

CAR000: Efficiency with Algorithms, Performance with Data Structures. Carruth, Chandler, CPPCON , 2014, <https://www.youtube.com/watch?v=fHNmRkzxHWs>

CLG000: Compilador Clang. Varios autores, The LLVM Foundation, 2017,
<http://clang.llvm.org>

CLK000: Extensión «cl_khr_gl_sharing» para compartición de datos entre OpenCL y OpenGL. Varios autores, Khronos Group, 2011,
https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/cl_khr_gl_sharing.html

CMA000: Herramienta de construcción de proyectos CMake. Varios autores, Kitware, 2017, <https://cmake.org>

CRYE000: Cryengine. Varios autores, Crytek GmbH, 2017, <https://www.cryengine.com/>

CYCL000: Cycles Renderer. Varios autores, Blender Foundation, 2017,
<https://www.cycles-renderer.org/>

CYCL001: Render Engine Comparison: Cycles vs The Rest. Baldi, Dario, Blender Guru, 2015, <https://www.blenderguru.com/articles/render-engine-comparison-cycles-vs-giants/>

- ENLI000: Englighten, iluminación global multi-plataforma. Varios autores, Geomerics Ltd., 2017, <http://www.geomerics.com/enlighten/>
- FRE000: Free Image, una biblioteca para soportar formatos de imagen populares. Drolon, Hervé, www, 2015, <http://freeimage.sourceforge.net/>
- FUJ00: Fujitsu Develops World's First Three Dimensional Geometry Processor. Umano, Asako, Fujitsu Limited, 1997, <http://pr.fujitsu.com/jp/news/1997/Jul/2e.html>
- GCC000: Compilador GCC. Varios autores, Free Software Foundation, Inc., 2017, <http://gcc.gnu.org>
- GIT000: Sistema de control de versiones distribuido GIT. Varios autores, Software Freedom Conservancy, 2017, <https://git-scm.com/>
- GLEW00: GLEW, The OpenGL Extension Wrangler Library. Varios autores, GLEW Team, 2016, <http://glew.sourceforge.net/>
- GLFW00: GLFW, infraestructura portable para el desarrollo de aplicaciones gráficas. Varios autores, GLFW Team, 2016, <http://www.glfw.org/>
- GLM000: Biblioteca OpenGL Mathematics, una biblioteca de matemáticas en C++ para programación gráfica. Varios Autores, G-Truc Creation, 2017, <http://glm.g-truc.net/0.9.8/index.html>
- GOF000: Design Patterns. Elements of Reusable Object-Oriented Software. Gamma, Erich. Helm, Richard. Johnson, Ralph. Vlissides, John, Addison-Wesley, 1995. ISBN:9780201633610, <https://www.pearsonhighered.com/program/Gamma-Design-Patterns-Elements-of-Reusable-Object-Oriented-Software/PGM14333.html>
- HEN000: Imagen GFDL del proceso de trazado de rayos. Henrik, Wikimedia Commons, 2008, <https://commons.wikimedia.org/w/index.php?curid=3869326>)
- KMSP00: Heterogeneous Computing with OpenCL 2.0. Kaeli, David. Mistry, Perhaad. Schaa, Dana. Ping Zhang, Dong, Morgan Kaufmann, 2015. ISBN:9780128014141, <https://www.elsevier.com/books/heterogeneous-computing-with-opencl-20/kaeli/978-0-12-801414-1>
- KTE0000: Thinking Parallel, Part III: Tree Construction on the GPU. Karras, Tero, NVIDIA Corporation, 2012, <https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/>
- LOGL00: Learn OpenGL, tutorial del uso de OpenGL con ejemplos. de Vries, Joey, www, 2017, <http://www.learnopengl.com>
- LUXR000: Renderizado en tiempo real con Luxrender. Bucciarelli, David, Youtube, 2016, <https://www.youtube.com/watch?v=fmitplI9jxs>

- LUXR001: Programa trazador de rayos «LuxRender». Varios autores, LuxRender, 2017, http://www.luxrender.net/en_GB/index
- LWN000: What every programmer should know about memory. Drepper, Ulrich, Linux Weekly News, 2007, <https://lwn.net/Articles/250967/>
- MAC000: Repositorio de modelos de libre distribución de McGuire Graphics Data . McGuire, Morgan, McGuire Graphics Data, 2011, <http://graphics.cs.williams.edu/data>
- MEY000: Effective Modern C++. Meyers, Scott, O'Reilly, 2014. ISBN:9781491903995, <http://www.aristeia.com/books.html>
- MEY001: Type deduction and why you care. Meyers, Scott, , 2014, <https://www.youtube.com/watch?v=wQxj20X-tIU>
- MEY002: CPU Caches and Why You Care. Meyers, Scott, CODE::DIVE, 2014, <https://www.youtube.com/watch?v=WDIkqP4JbkE>
- MITS000: Mitsuba - Physically Based Renderer. Jakob, Wenzel, Mitsuba, 2017, <https://www.mitsuba-renderer.org/>
- MT000: Fast, Minimum Storage Ray/Triangle Intersection. Möller, Thomas. Trumbore, Ben., Journal of Graphics Tools, 1997, <http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf>
- OCA0000: Haswell: Core i7 4770K and Core i5 4670K. Büchel, Marc, Ocaholic, 2013, <https://ocaholic.ch/modules/smartersection/item.php?itemid=1005&page=6>
- OCL000: OpenCL. Open Computing Library. Varios autores, Khronos Group, 2017, <https://www.khronos.org/opencl/>
- OCL001: An Introduction to the OpenCL Programming Model. Tompson, Jonhatan. Schlachter, Kristofer, NYU: Media Research Lab, 2012, <http://cims.nyu.edu/~schlacht/OpenCLModel.pdf>
- OCTA000: OctaneRender - Real-time 3D Rendering. OTOY Inc., OTOY Inc., 2017, <https://home.otoy.com/render/octane-render/>
- OGL000: Open Graphics Library. Varios autores, Khronos Group, 2017, <https://www.opengl.org/>
- OMP000: OpenMP (Open Multi-Processing). Varios autores, The OpenMP ARB (Architecture Review Boards), 2017, <http://www.openmp.org/>
- PHH00: Physically Based Rendering. Pharr, Matt. Humphreys, Greg, Morgan Kaufmann, 2010. ISBN:9780123750792, <http://www.pbrt.org/>

PLC00: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming . Peng Dua, Rick Webera, Piotr Luszczeka, , , Stanimire Tomova, Gregory Petersona, Jack Dongarra, Elsevier, 2012, <http://www.sciencedirect.com/science/article/pii/S0167819111001335>

QTC000: Entorno de desarrollo integrado Qt Creator. Varios autores, The Qt Company, 2017, https://wiki.qt.io/Qt_Creator

RAVI000: Parallel Divide and Conquer Ray Tracing. Ravichandran, Srinath, Dartmouth department of Computer Science, 2013, <http://www.cs.dartmouth.edu/~sriravic/>

REND000: RenderMan. Varios autores, Pixar Inc., 2017, <https://renderman.pixar.com/view/renderman>

SHE00: OpenCL™ and OpenGL* Interoperability Tutorial. Shevtsov, Maxim, Intel Developer Zone, 2014, <https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial>

SNOW000: Snow Drop. Massive Entertainment, Ubisoft, 2017, <http://www.massive.se/>

SSKL0: OpenGL Programming Guide, 8th ed.. Shreiner, Dave. Sellers, Graham. Kessenich, John. Licea-Kane, Bill, Addison Wesley, 2013. ISBN:9780321773036,

STR00: A Tour of C++. Bjarne Stroustrup , Addison Wesley, 2014. ISBN:9730321958310, <http://www.stroustrup.com/Tour.html>

STR001: Make Simple Tasks Simple!. Stroustrup, Bjarne, CPPCON, 2014, <https://www.youtube.com/watch?v=nesCaocNjtQ>

STR002: What - if anything - have we learned from C++?. Stroustrup, Bjarne, Curry On, 2015, https://www.youtube.com/watch?v=2egL4y_VpYg

STR003: The Essence of C++. Stroustrup, Bjarne, , 2014, <https://www.youtube.com/watch?v=86xWVb4XlyE>

SUF00: Ray Tracing from the Ground Up. Suffern, Kevin, A.K.Peters, LTD, 2007. ISBN:9781568812724, <http://www.raytracegroundup.com/>

SUT000: Back to the Basics! Essentials of Modern C++ Style. Sutter, Herb, CPPCON, 2014, <https://www.youtube.com/watch?v=xnqTKD8uD64>

SWH00: OpenGL SuperBible, 6th ed. Sellers, Graham. Wright, Richard S. Jr. Haemel, Nicholas, Addison Wesley, 2014. ISBN:9780321902948, <http://www.openglsuperbible.com/>

SWOB000: A Really Realtime Raytracer. Swoboda, Matt, NVScene, 2014, <https://www.youtube.com/watch?v=JSr6wvkvgM0>

TPU0000: AMD Radeon R9 285 specifications. Autor desconocido, TechPowerup, 2017, <https://www.techpowerup.com/gpudb/2609/radeon-r9-285>

UCED00: Computer Graphics. CSE167x. Ramamoorthi, Ravi, UC San Diego, 2017,
<https://www.edx.org/course/computer-graphics-uc-san-diegox-cse167x-1>

UNIG000: Unigine. Varios autores, Unigine Corp., 2017, <https://unigine.com/>

UNIT000: Unity. Varios autores, Unity Technologies, 2017, <https://unity3d.com/es/>

UNRE000: Unreal Engine. Varios autores, Epic Games, Inc., 2017,
<https://www.unrealengine.com/>

VIM000: Editor de textos VIM. Moolenaar, Bram, , 2017, <http://www.vim.org>

VRAY000: V-Ray. Varios autores, Chaos Software Ltd, 2017,
<https://www.chaosgroup.com/>

WIK000: Artículo de la Wikipedia "Name Value Return Optimization". Varios, Wikimedia Foundation, 2016, https://en.wikipedia.org/wiki/Return_value_optimization

10 Acrónimos y siglas

- AMD:** Advanced Micro Devices
- API:** Application Programming Interface
- ASSIMP:** Open Asset Import Library
- BSD:** Berkeley Software Distribution
- BVH:** Bounding Volume Hierarchy
- CGI:** Computer generated imagery
- CPU:** Central Processing Unit
- CU:** Compute Unit
- CUDA:** Compute Unified Device Architecture
- DRAM:** Dynamic Random Access Memory
- GDDR5:** Double data rate type five synchronous graphics random-access memory
- FPS:** Frames per second.
- G-buffer:** Graphics Buffer
- GB:** Gigabytes
- GCC:** GNU Compiler Collection
- GFLOPS:** Giga floating point operations per second
- GLM:** Open Graphics Library Mathematics
- GLEW:** Open Graphics Library Extensions Wrangler
- GLSL:** OpenGL Shading Language
- GNU:** Gnu's not Unix
- GPU:** Graphics Processing Unit
- GPGPU:** General-purpose computing on Graphics Processing Units
- NRVO:** Name Value Return Optimization
- OpenCL:** Open Computing Language
- OpenGL:** Open Graphics Library
- OpenMP:** Open Multi-Processing
- Pixel:** Picture element
- PBO:** píxel-Buffer-Object
- OS:** Operative System
- UML:** Unified Modelling Language
- RAII:** Resource Acquisition Is Initialization
- RGBA:** Red Green Blue Alpha
- RPP:** Rayos por píxel (Rays per pixel)
- SDK:** Software Development Kit

SGPR: Scalar General Purpose Register

SIMT: Single Instruction, Multiple Thread

SVM: Shared Virtual Memory

VGPR: Vector General Purpose Register

11 Anexos

11.1 Descripción de la arquitectura de OpenCL

A continuación se incluye un breve resumen de la arquitectura de OpenCL basado en [OCL001], que amplía la información ofrecida en el capítulo 3.4.

OpenCL es un estándar e interfaz de programación de aplicaciones de Khronos Group [OCL000] para computación paralela heterogénea en *hardware* multi-plataforma y multi-dispositivo. Proporciona una abstracción de alto nivel para rutinas de *hardware* de bajo nivel, así como modelos de memoria y ejecución consistentes para la ejecución de código de forma masivamente paralela. La ventaja de esta capa de abstracción es su capacidad de escalado, ya que puede producir código que se ejecute en *hardware* que va desde los simples microcontroladores embebidos, o CPUs de Intel o AMD, hasta GPGPU masivamente paralelas, sin necesidad de modificar el código.

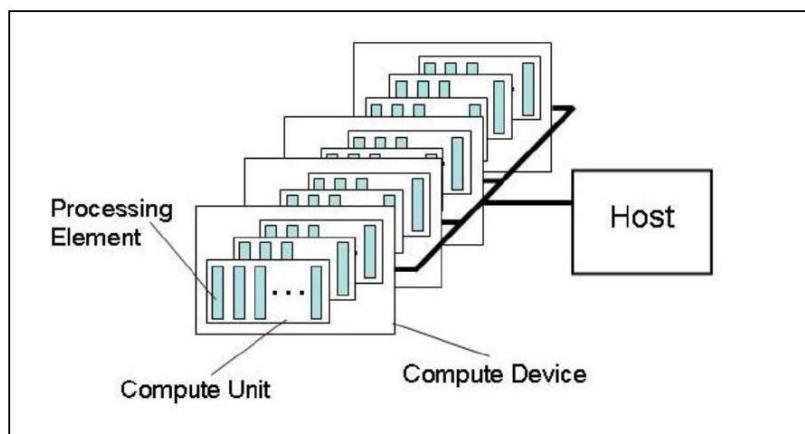


Figura 35: Modelo de Plataforma de OpenCL (de Khronos, 2011)

La figura 35 muestra un resumen de la arquitectura de OpenCL. Un «Host» basado en CPU controla múltiples «Compute Devices» o dispositivos de computación (por ejemplo, CPU y GPU son dispositivos de computación diferentes). Cada uno de estos dispositivos de grano grueso están formados por múltiples «Compute Units» o unidades de computación (equivalentes a los núcleos de una CPU) y dentro de éstos se encuentran múltiples «Processing Elements» o elementos de procesado. En el nivel más bajo, estos elementos de procesado ejecutan *kernel*s de OpenCL.

La definición específica de las unidades de computación depende del fabricante del *hardware*.

Modelo de ejecución

En el nivel más alto de la jerarquía de ejecución de OpenCL, el *host* utiliza el interfaz de programación de aplicaciones (API) para consultar y seleccionar los dispositivos de computación, enviarles tareas y administrar la carga de trabajo a través de contextos y colas. Por otro lado, en el nivel más bajo de la jerarquía se encuentran los *kernels* de OpenCL ejecutándose en los elementos de procesado. Dichos *kernels* están escritos en C de OpenCL y se ejecutan en paralelo en un dominio de computación de N-dimensiones predefinido. A cada elemento de ejecución independiente en este dominio se le denomina «*work-item*» o elemento de trabajo. Estos «*work-items*» están agrupados en «*work-groups*» o grupos de trabajo, los cuales son independientes unos de los otros. En la figura 36 se muestra un resumen de esta estructura.

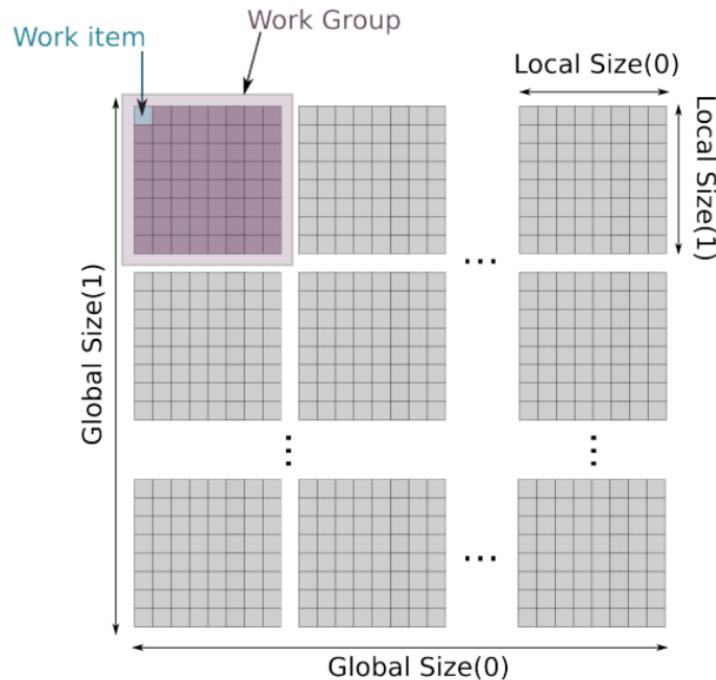


Figura 36: Ejecución paralela de datos 2D en OpenCL

De acuerdo con la documentación, el modelo de ejecución consiste en “paralelismo de datos e hilos de grano fino, anidado en paralelismo de datos y tareas de grano grueso”. La programación paralela de datos consiste en definir el dominio de ejecución de cada hebra sobre cierta región de una estructura de datos u objeto, habitualmente

mediante un rango de índices en un array de NxN, donde la ejecución en esas subregiones se considera independiente, tal y como se muestra en la figura 36. El modelo alternativo es programación de tareas paralelas, donde la concurrencia es explotada a través de dominios de paralelismo a nivel de tareas. OpenCL utiliza ambos modelos aunque, dado que el acceso a memoria global es lento, es necesario ser cuidadoso y escribir código de *kernel*s que tenga en cuenta los rendimientos de acceso a ciertas localizaciones de la jerarquía de memoria. De esta forma, los *work-groups* pueden ser divididos por programación paralela a nivel de tareas, dado que las hebras dentro de un *work-group* pueden compartir memoria local, pero es más habitual que sean subdominios en una estructura de datos más grande dado que esto mejora el acceso a memoria *hardware*, ya que transferir memoria desde DRAM a la región de memoria global de la GPU es lento, como también lo es transferir memoria global de la GPU a memoria local de un *work-group*.

Dado que se produce la ejecución concurrente de cientos de hebras, lo que resulta en un aumento lineal de ancho de banda de entrada-salida, NVIDIA usa una arquitectura Single-Instruction, Multiple-Thread (SIMT). Una llamada a una instrucción en esta arquitectura ejecuta código idéntico en diferentes hebras en paralelo y cada hebra ejecuta el código sobre diferentes datos. Este esquema reduce el ancho de banda de entrada-salida y permite una lógica de ejecución de hebras más compacta. La arquitectura de AMD sigue un modelo muy similar, aunque la nomenclatura es diferente.

Con la arquitectura descrita hasta ahora, se puede resumir los pasos de ejecución de una aplicación OpenCL:

1. En primer lugar una CPU *host* define un dominio computacional sobre una región de memoria DRAM mediante un array de N-dimensiones. Cada índice de este array de N-dimensiones será un *work-item* y cada *work-item* ejecutará el mismo *kernel*.
2. El *host* define la agrupación de estos *work-items* en *work-groups*. Cada *work-item* perteneciente a un *work-group* se ejecutará concurrentemente en una *compute unit* y compartirán memoria local. Estos *work-groups* se situarán en una *work-queue*.
3. El *hardware* transferirá la memoria DRAM a la región global de la RAM de la GPU y ejecutará cada *work-group* de la *work-queue*.

4. En el *hardware* NVIDIA se ejecutarán 32 hebras a la vez (lo que llaman «*warp group*»). Si el *work-group* contiene más hebras, ésta serán serializadas, lo que tiene implicaciones obvias sobre la consistencia de la memoria local. El equivalente en AMD es el *wavefront*.

Cada *processing element* ejecuta código puramente secuencial. No hay predicción de bifurcaciones ni ejecución especulativa, así que las instrucciones en una hebra son ejecutadas en orden. Es más, algunos saltos condicionales requerirán que se ejecuten ambas ramas de ejecución, y serán luego multiplexadas para producir el resultado final.

Modelo de memoria

El modelo de memoria de OpenCL, mostrado en la figura 37, está estructurado para imitar la configuración física de memoria del *hardware* de AMD y NVIDIA, aunque no hay una correspondencia completa puesto que ambas compañías definen la jerarquía de memoria de sus productos de forma diferente. Sin embargo, la estructura consistente en memoria global frente memoria local por *work-group* sí es consistente en ambas plataformas. Es más, en ambas el nivel de ejecución más bajo tiene un pequeño espacio de memoria privada para registros del programa.

Los *work-groups* se pueden comunicar a través de memoria compartida y primitivas de sincronización, aunque su acceso a memoria es independiente de la de otros *work-groups*. Se trata en esencia de un modelo de ejecución de datos en paralelo, donde el dominio de unidades de ejecución independientes está enlazado y definido por los patrones de acceso de memoria subyacentes. Para estos grupos, OpenCL implementa un modelo de memoria compartida de consistencia relajada. Hay excepciones, y algunos *compute devices*, sobre todo CPUs, pueden ejecutar *kernels* de computación de tareas paralelas. Sin embargo, la mayor parte de las aplicaciones en *hardware* GPGPU ejecutarán tareas estrictamente de ejecución de datos en paralelo.

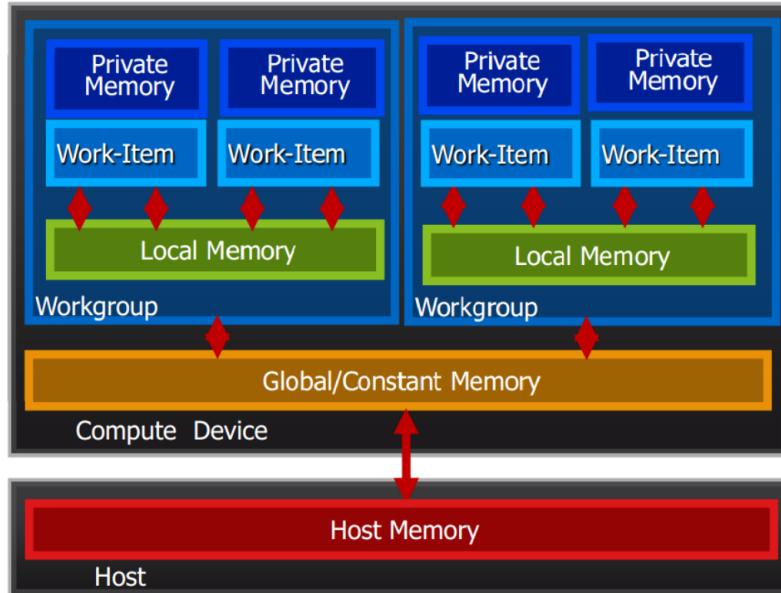


Figura 37: Modelo de memoria de OpenCL (Khronos, 2011)

Una cuestión a tener en cuenta cuando se programa un *kernel* de OpenCL es que el acceso a los bloques de memoria DRAM, global y local no está protegido de ninguna forma. Esto significa que las violaciones de segmento, cuando los *work-items* acceden a memoria fuera de su propio almacenamiento global, no son notificadas. La consecuencia es que memoria GPU reservada por el sistema operativo puede ser sobrescrita de forma no intencionada, lo que puede resultar en comportamientos que van desde parpadeos benignos de pantalla hasta pantallazos azules de la muerte (BSOD) y cuelgues a nivel del sistema operativo.

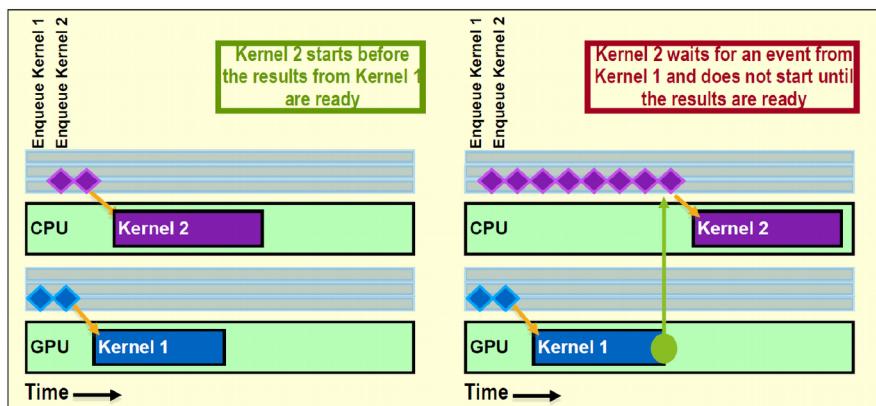


Figura 38: Control de concurrencia con event-queueing de OpenCL

Es posible utilizar *fences* (vallas) de memoria dentro de las hebras de un *work-group*, así como *barriers* (barreras) de sincronización para hebras al nivel de *work-item* (entre hebras individuales en un *processing element*), así como a nivel de *work-group* (para sincronización de grano grueso entre *work-groups*). En el lado del *host*, las funciones API con bloqueo pueden esperar a que ciertos eventos se completen, tales como eventos en una *queue*, o eventos específicos. Utilizando este control de eventos de grano grueso el *host* puede decidir ejecutar tareas en paralelo o secuencialmente sobre varios dispositivos diferentes (figura 38), dependiendo en cómo se sitúen los marcadores en el *work-queue*.

11.2 Instrucciones para construir y ejecutar el proyecto

El proyecto ha sido desarrollado usando el sistema operativo GNU/Linux, sin embargo está compuesto únicamente por componentes portables, por lo que es posible construirlo y ejecutarlo al menos en los sistemas operativos GNU/Linux, Mac OS X y Microsoft Windows.

Se ha realizado la compilación tanto en GNU/Linux como en Microsoft Windows y a continuación se detalla el proceso seguido.

GNU/Linux

Partiendo de la base de que hay un entorno gráfico instalado y los drivers y el *runtime* de OpenCL adecuados para la tarjeta gráfica a utilizar, el primer paso es instalar las bibliotecas de terceros y los compiladores. En la distribución utilizada, ArchLinux, el comando a utilizar es el siguiente:

```
pacman -S base-devel clang gcc cmake glew glfw-x11 freeimage assimp openmp
```

Después es necesario copiar el código fuente a un directorio, por ejemplo ~/proyecto/src y crear el directorio en el que se construirá el proyecto:

```
mkdir ~/proyecto/build
```

Lo siguiente es generar el archivo que servirá para construir el proyecto. En este caso, usando «makefile»:

```
cd ~/proyecto/build
cmake -G Unix Makefiles ../src
```

Este proceso debe detectar que todas las dependencias están instaladas y generar el «makefile» adecuado. Si no ha dado ningún error:

```
make
```

Y el proyecto se compilará y enlazará.

Microsoft Windows

En este caso no hay un comando para instalar todas las dependencias, por lo que el proceso es más complicado y susceptible a errores. Puesto que es muy dependiente de la tarjeta gráfica a utilizar, también partimos del punto en el que están instalados los drivers y el *runtime* de OpenCL.

Se ha realizado la compilación utilizando Visual Studio Community 2017 en el sistema operativo Microsoft Windows 10 64 bits.

Para cada una de las dependencias descritas en el capítulo 3.6, descargar el código fuente de la página del proyecto y descomprimirlo en una ruta temporal. Lanzar cmake-gui, indicando como ruta de entrada el archivo descargado descomprimido y como ruta de salida otra carpeta temporal. Pulsar en configurar. Seleccionar un generador adecuado, por ejemplo Visual Studio 15 64bit y configurar la ruta de instalación. Seleccionar el mismo prefijo para todas las dependencias. Por ejemplo, usar C:\lab\libs Pulsar en generar y luego en abrir proyecto. Se abrirá Visual Studio. Pinchar el “Construir” y luego construir en el proyecto “INSTALL”.

Una vez estén todas las dependencias instaladas en la misma ruta, editar el archivo CmakeLists.txt que se encuentra en el directorio del código fuente para actualizar esta variable, llamada LIBS_PREFIX. Copiar también la carpeta “cmake” que se proporciona junto con el código fuente a la misma carpeta en la que se han instalado las dependencias. En esta carpeta se encuentran archivos para que Cmake pueda encontrar las dependencias que no proporcionan integración con Cmake.

Finalmente, volver a ejecutar cmake-gui, esta vez tomando como origen la carpeta «src» del proyecto y realizar los mismos pasos de configuración y apertura del proyecto realizados hasta ahora. En este punto se debería poder construir el proyecto obteniendo como resultado un ejecutable.

Ejecución del programa

Una vez generado el ejecutable, se podrá lanzar pasándole como último parámetro el nombre de un archivo de descripción de escena, cuya especificación se encuentra en el anexo 11.4.

El programa admite los siguientes parámetros:

--deferred : Utiliza el método diferido de renderizado

--hybrid : Utiliza la técnica híbrida de renderizado codificada en OpenCL ejecutándose en GPU.

--cpu : Utiliza la técnica híbrida de renderizado codificada en C++ ejecutándose en CPU.

--noshadows : No realiza el cálculo de sombras proyectadas en los métodos de técnica híbrida

--nocapturemouse : No captura el puntero del ratón. Útil para depuración.

El dispositivo OpenCL debe soportar la extensión «cl_krh_gl_sharing» para poder ejecutar el programa usando la técnica híbrida ejecutada en GPU.

Durante la ejecución del programa es posible rotar la cámara utilizando el ratón y moverla utilizando los cursores del teclado. Además, pulsando “l” mostrará en consola la localización de la cámara, “p” mostrará las estadísticas de rendimiento, “r” reseteará las estadísticas, “s” activará/desactivará el cálculo de sombras, “d” pasará al vuelo a usar el método diferido, “h” pasará a usar el método híbrido y “c” pasará a utilizar el método híbrido en CPU.

11.3 Listado de archivos de código fuente y breve descripción de su función

- bbox** (bbox.hpp) - Métodos y propiedades de un objeto tipo caja envolvente.
- bvh** (bvh.hpp bvh.cpp) - Código para generar la estructura de jerarquía de volúmenes envolventes en árbol y más tarde «aplanarla» a un *array*.
- camera** (camera.hpp camera.cpp) - Controla la posición y orientación de la cámara y sus matrices asociadas: vista y proyección
- cl_device** (cl_device.hpp cl_device.cpp) - Obtiene información de los dispositivos OpenCL pertenecientes a una plataforma disponibles en el sistema.
- clkernelmanager** (clkernelmanager.hpp clkernelmanager.cpp) - Realiza la detección e inicialización de OpenCL. También carga y compila *kernels* de disco, además de tener métodos para llamar a un *kernel*, generar reservas de memoria y especificar los argumentos en la llamada a ejecución.
- cl_platform** (cl_platform.hpp cl_platform.cpp) - Obtiene información de las plataformas OpenCL disponibles en el sistema.
- config** (config.hpp) - Alberga parámetros de configuración y de estado del sistema.
- configloader** (configloader.hpp configloader.cpp) - Permite la lectura de disco de un archivo de descripción de configuración y escena, con el formato descrito en el capítulo 11.4.
- deferredshader** (deferredshader.hpp deferredshader.cpp) - Implementa el renderizado mediante el sombreado diferido. Es una subclase de RenderEngine.
- hybridshader** (hybridshader.hpp hybridshader.cpp) - Implementa el método de renderizado híbrido mediante OpenCL. Es una subclase de RenderEngine.
- hybridshadercpu** (hybridshadercpu.hpp hybridshadercpu.cpp) - Implementa el método de renderizado híbrido en C++ y ejecutándose en CPU. Es una subclase de RenderEngine.
- input** (input.hpp input.cpp) - Encapsula la recogida de datos de los dispositivos de entrada: ratón y teclado.
- main** (main.cpp) - Función principal. Llama a la inicialización y contiene el bucle principal del programa.
- material** (material.hpp) - Guarda las propiedades de un material, tales como intensidad de componente difuso, especular o sus texturas.
- mesh** (mesh.hpp mesh.cpp) - Un Mesh es una malla de triángulos con un material asociado. Esta clase encapsula sus datos y métodos asociados, tales como el encargado de «pintar» la malla.
- model** (model.hpp model.cpp) - Descripción de un modelo tridimensional y métodos asociados. Está compuesto de uno o varios Mesh.
- modelloader** (modelloader.hpp modelloader.cpp) - Lectura del disco de los modelos a mostrar.
- pointlight** (pointlight.hpp) - Contiene los valores que definen un punto emisor de luz.

- renderenginecreator** (`renderenginecreator.hpp`) - Implementa el patrón de diseño «Factory Method», de forma que en la función principal Main no es necesario incluir ningún código específico del método de renderizado a utilizar.
- renderengine** (`renderengine.hpp`) - Clase abstracta que define el interfaz que deben cumplir todos los métodos de renderizado que se definan en el programa.
- shaderloader** (`shaderloader.hpp` `shaderloader.cpp`) - Lee del disco los *shaders*, los compila y los enlaza para poder ejecutarlos.
- shaderprogram** (`shaderprogram.hpp` `shaderprogram.cpp`) - Contiene un programa GLSL, compuesto por un *vertex shader* y un *fragment shader*.
- system** (`system.hpp`) - Encapsula todas las funciones relacionadas con el sistema. Contiene los objetos temporizador, ventana de la aplicación y entrada del teclado y ratón
- texture** (`texture.hpp`) - Guarda la relación entre el *uniform* y el identificador único con el que OpenGL identifica una textura.
- textureloader** (`textureloader.hpp` `textureloader.cpp`) - Carga una imagen del disco utilizando la biblioteca FreeImage.
- texturemanager** (`texturemanager.hpp` `texturemanager.cpp`) - Caché de texturas, para evitar leer más de una vez de disco la misma textura.
- timer** (`timer.hpp`) - Temporizador independiente de la plataforma. Devuelve la diferencia en milisegundos entre llamadas al método update.
- triangle** (`triangle.hpp`) - Contiene un triángulo definido por las posiciones de sus vértices en un formato directamente compatible con OpenCL.
- vertex** (`vertex.hpp`) - Contiene un vértice en un formato compatible con OpenGL. El vértice incluye información de posición, posición en la textura y vector normal.
- window** (`window.hpp` `window.cpp`) - Utiliza las API de GLFW para inicializar la ventana en la que se mostrará la aplicación. También define varios parámetros de OpenGL, como la eliminación de caras ocultas o la espera del refresco de monitor.
- world** (`world.hpp` `world.cpp`) - Define el objeto «Mundo», que contiene los modelos, fuentes de luz, cámara y demás objetos de componen la escena.
- kernels/render.cl** (`kernels/render.cl`) - Código OpenCL encargado de la segunda fase de la técnica híbrida, que se ejecuta en la tarjeta gráfica.
- shaders/gbuffer.frag** (`shaders/gbuffer.vert`) - Código GLSL que corresponde al «*fragment shader*» de la primera pasada del sombreado diferido, encargada de generar el G-Buffer.
- shaders/gbuffer.vert** (`shaders/gbuffer.frag`) - Código GLSL que corresponde al «*vertex shader*» de la primera pasada del sombreado diferido, encargada de generar el G-Buffer.
- shaders/lighting.frag** (`shaders/lighting.vert`) - Código GLSL que corresponde al «*fragment shader*» de la segunda pasada del sombreado diferido, encargada de realizar los cálculos de sombreado.
- shaders/lighting.vert** (`shaders/lighting.frag`) - Código GLSL que corresponde al «*vertex shader*» de la segunda pasada del sombreado diferido, encargada de realizar los cálculos de sombreado.

11.4 Formato del archivo de descripción de escenas

El programa desarrollado en el presente proyecto utiliza un formato propio para establecer ciertos parámetros de configuración y definir la escena a mostrar. El formato es intencionadamente simple para evitar añadir alguna dependencia que permita tratar otros formatos más complejos, como xml o json.

Los diferentes valores están separados por cualquier número de espacios y los caracteres “//” al principio de una línea la marcarán como comentario, por lo que se ignorará.

A continuación se incluye un ejemplo de archivo de descripción de escena:

```
window 1024 768
model models/sponzacyr/sponza.obj

gbuffer_shader shaders/gbuffer
lighting_shader shaders/lighting
lighting_kernel kernels/render.cl

//      position      znear  zfar    pitch   yaw
camera -1100.0f 150.0f -40.0f   0.1f   3000.0f  0.0f   0.0f

//      speed   zoom   sensitivity
movement 400.0f 40.0f 0.25f

// ambient light intensity
ambient 0.10

//      position      color      diffuse   linear   quadratic
pointlight -1300.0f 1900.0f 160.0f 1.0f 1.0f 1.0f   0.0f   0.0f
pointlight -1000.0f 150.0f -350.0f 1.0f 1.0f 1.0f   0.5f   0.001f   0.00008f
```

Siendo la siguiente la descripción de los parámetros necesarios:

window <ancho> <alto> - Ancho y alto especifican el tamaño de la ventana del programa, en píxeles

model <ruta> - Ruta y nombre del archivo de modelo a utilizar en la escena. El formato debe ser compatible con ASSIMP.

gbuffer_shader <ruta> - Ruta y nombre de los archivos (sin extensión) del *vertex shader* y *fragment shader* para la primera pasada del sombreado diferido y técnica híbrida. Se buscará el *vertex shader* añadiendo la extensión *.vert* y el *fragment shader* añadiendo la extensión *.frag*.

lighting_shader <ruta> - Ruta y nombre de los archivos (sin extensión) del *vertex shader* y *fragment shader* para la segunda pasada del sombreado diferido. Se buscará el *vertex shader* añadiendo la extensión *.vert* y el *fragment shader* añadiendo la extensión *.frag*.

lighting_kernel <ruta> - Ruta y nombre del archivo con extensión en el que se encuentra el *kernel* de OpenCL «*render*». Los parámetros del *kernel* deben coincidir con los que se utilizan en el programa.

camera <posición> <zNear> <zFar> <cabeceo> <guiñada> - Datos de la cámara. La posición se especificará mediante tres números en coma flotante, que representan las coordenadas x, y y z. Znear indica la distancia más cercana a la que es visible un objeto y zFar la más lejana. Cabeceo y guiñada especifican la orientación de la cámara con ángulos de Euler.

movement <velocidad> <zoom> <sensibilidad> - Velocidad, zoom inicial y sensibilidad del movimiento del ratón.

ambient <factor> - Factor constante de iluminación difusa global.

pointlight <posición> <color> <difuso> <atenuación lineal> <atenuación cuadrática> - Datos de cada punto de luz. La posición especifica las coordenadas mediante tres números en coma flotante, color representa los valores RGB del color del punto de luz, en un intervalo de 0 a 1. Difuso expresa la intensidad del punto de luz y los dos siguientes factores definen la atenuación del punto de luz.

11.5 Listado de código fuente

11.5.1 bbox.hpp

```
*****
* BBox
*
* Bounding box implementation
*
* 2017 - Liberto Camús
* ****
#ifndef BBOX_H
#define BBOX_H

#include <algorithm>
#include <glm/glm.hpp>

<**
* @brief The BBox class
*
* Contains a bounding box volume and related methods
*/
class BBox {
public:
    glm::vec3 pMin, pMax; // Further apart vertices

    BBox()
        : pMin{INFINITY, INFINITY, INFINITY}, pMax{-INFINITY, -INFINITY,
          -INFINITY} {}

    BBox(const glm::vec3 &p1, const glm::vec3 &p2)
        : pMin{std::min(p1.x, p2.x), std::min(p1.y, p2.y), std::min(p1.z, p2.z)},
          pMax{std::max(p1.x, p2.x), std::max(p1.y, p2.y), std::max(p1.z, p2.z)} {}

    <**
     * @brief Union of a bounding box and a point
     * @param b Bounding box
     * @param p Point to stretch the bounding box to
     * @return New bounding box with the union
    */
    friend BBox Union(const BBox &b, const glm::vec3 &p) {
        BBox ret = b;
        ret.pMin.x = std::min(b.pMin.x, p.x);
        ret.pMin.y = std::min(b.pMin.y, p.y);
        ret.pMin.z = std::min(b.pMin.z, p.z);
        ret.pMax.x = std::max(b.pMax.x, p.x);
        ret.pMax.y = std::max(b.pMax.y, p.y);
        ret.pMax.z = std::max(b.pMax.z, p.z);
        return ret;
    }

    <**
     * @brief Union of two bounding boxes
     * @param b First bbox
     * @param b2 Second bbox
     * @return New bbox which is the union of the two
    */
    friend BBox Union(const BBox &b, const BBox &b2) {
        BBox ret = b;
        ret.pMin.x = std::min(b.pMin.x, b2.pMin.x);
        ret.pMin.y = std::min(b.pMin.y, b2.pMin.y);
        ret.pMin.z = std::min(b.pMin.z, b2.pMin.z);
        ret.pMax.x = std::max(b.pMax.x, b2.pMax.x);
        ret.pMax.y = std::max(b.pMax.y, b2.pMax.y);
        ret.pMax.z = std::max(b.pMax.z, b2.pMax.z);
        return ret;
    }
}
```

```

}

/*
 * @brief MaximumExtent
 * Calculates the largest dimension of the cube
 * @return 0 -> x, 1 -> and 2 -> z
 */
unsigned char MaximumExtent() const {
    glm::vec3 diag = pMax - pMin; /// Diagonal
    if (diag.x > diag.y && diag.x > diag.z)
        return 0;
    else if (diag.y > diag.z)
        return 1;

    return 2;
}

#endif // BBOX_H


```

11.5.2 bvh.hpp

```

*****
* BVH
*
* Bounding Volume Hierarchy creation code.
* Creates a BVH tree and if flattens it to an array
* Based on algorith from "Physically based Rendering,
* 2nd edition"
*
* TODO: Use SAH
*
* 2017 - Liberto Camús
*****/

#ifndef BVH_H
#define BVH_H

#include "bbox.hpp"
#include "bvhlinenode.hpp"
#include <memory>
#include <vector>

class Triangle;
struct BVHTreeNode;
class BVHPrimitiveInfo;

class BVH {
public:
    size_t totalNodes = 0;
    std::vector<Triangle> triangles;
    std::unique_ptr<BVHLinearnode[]> nodes_array;
    void init();

private:
    static constexpr unsigned int PRIMITIVES_PER_NODE = 4;

    // Recursive method to build the BVH tree. Uses "createLeaf"
    void recursiveBuild(BVHTreeNode *node,
                        std::vector<BVHPrimitiveInfo> &buildData, size_t start,
                        size_t end, std::vector<Triangle> &orderedPrims);

    // Creates a leaf for the tree structure
    void createLeaf(BVHTreeNode *node, std::vector<BVHPrimitiveInfo> &buildData,
                   size_t start, size_t end, std::vector<Triangle> &orderedPrims,
                   BBox &bbox, unsigned char nPrimitives);
}

```

```

// Recursive method that fills "nodes_array" with the contents of the
// BVH tree.
size_t flattenBVHTree(const BVHTreeNode *node, size_t &offset,
                      size_t &max_depth, size_t depth);
};

#endif // BVH_H

```

11.5.3 bvh.cpp

```

/****************************************************************************
 * BVH
 *
 * Bounding Volume Hierarchy creation code.
 * Creates a BVH tree and if flattens it to an array
 * Based on algorith from "Physically based Rendering,
 * 2nd edition"
 *
 * TODO: Use SAH
 *
 * 2017 - Liberto Camús
 * ****
#include "bvh.hpp"
#include "bvhprimitiveinfo.hpp"
#include "triangle.hpp"

#include <iostream>
#include <memory>

struct BVHTreeNode {
    BBox bounds;
    std::unique_ptr<BVHTreeNode> c0, c1;
    size_t firstPrimOffset;
    unsigned char splitAxis, nPrimitives;
    BVHTreeNode() {}

    void createChildren() {
        c0 = std::make_unique<BVHTreeNode>();
        c1 = std::make_unique<BVHTreeNode>();
    }

    BVHTreeNode *getChild0() const { return c0.get(); }
    BVHTreeNode *getChild1() const { return c1.get(); }
    void InitLeaf(size_t first, unsigned char n, const BBox &b) {
        firstPrimOffset = first;
        nPrimitives = n;
        bounds = b;
    }
    void InitInterior(unsigned char axis) {
        bounds = Union(c0->bounds, c1->bounds);
        splitAxis = axis;
        nPrimitives = 0;
    }
};

void BVH::init() {
    // Initialize buildData array for triangles (primitive number,
    // bbox and centroid)
    std::vector<BVHPrimitiveInfo> buildData;
    std::vector<Triangle> orderedPrims;
    buildData.reserve(triangles.size());
    for (size_t i = 0; i < triangles.size(); ++i) {
        BBox bbox = triangles[i].getBbox();
        buildData.push_back(BVHPrimitiveInfo(i, bbox));
    }

    // Recursively build BVH tree for triangles
    std::unique_ptr<BVHTreeNode> root = std::make_unique<BVHTreeNode>();
}

```

```

recursiveBuild(root.get(), buildData, 0, triangles.size(), orderedPrims);
std::cout << "Nodes:" << totalNodes << "\n";
triangles.swap(orderedPrims);

// Compute representation of depth-first traversal of BVH tree
nodes_array = std::make_unique<BVHLinearNode[]>(totalNodes);
size_t offset{0}, maxlevel{1};
flattenBVHTree(root.get(), offset, maxlevel, 1);

// Print info
std::cout << "Size of each BVH node:" << sizeof(BVHLinearNode)
      << " bytes \n";
std::cout << "Size of BVH structure:"
      << totalNodes * sizeof(BVHLinearNode) / 1024 << " KB\n";
std::cout << "Size of primitives:"
      << triangles.size() * sizeof(Triangle) / 1024 << " KB\n";
std::cout << "Max levels:" << maxlevel << "\n";
std::cout << "Number of nodes:" << totalNodes << "\n";
std::cout << "Number of primitives: " << triangles.size() << "\n";
}

size_t BVH::flattenBVHTree(const BVHTreeNode *node, size_t &offset,
                           size_t &max_depth, size_t depth) {
    // Update max depth if necessary
    if (max_depth < depth)
        max_depth = depth;

    // Pointer to current node in the array
    BVHLinearNode *linearNode = &nodes_array[offset];
    auto my_offset = offset;
    ++offset;

    // Update bounds
    linearNode->pMin = {
        {node->bounds.pMin.x, node->bounds.pMin.y, node->bounds.pMin.z}};
    linearNode->pMax = {
        {node->bounds.pMax.x, node->bounds.pMax.y, node->bounds.pMax.z}};

    // Fill leaf or continue recursion
    if (node->nPrimitives > 0) {
        // Leaf
        linearNode->primitivesOffset = static_cast<cl_uint>(node->firstPrimOffset);
        linearNode->nPrimitives = node->nPrimitives;
    } else {
        // Create interior flattened BVH node
        linearNode->axis = node->splitAxis;
        linearNode->nPrimitives = 0;
        flattenBVHTree(node->getChild0(), offset, max_depth, depth + 1);
        linearNode->secondChildOffset = static_cast<cl_uint>(
            flattenBVHTree(node->getChild1(), offset, max_depth, depth + 1));
    }
}

return my_offset;
}

void BVH::createLeaf(BVHTreeNode *node,
                     std::vector<BVHPrimitiveInfo> &buildData, size_t start,
                     size_t end, std::vector<Triangle> &orderedPrims,
                     BBox &bbox, unsigned char nPrimitives) {
    size_t firstPrimOffset = orderedPrims.size();
    for (size_t i = start; i < end; ++i) {
        size_t primNum = buildData[i].primitiveNumber;
        orderedPrims.push_back(triangles[primNum]);
    }
    node->InitLeaf(firstPrimOffset, nPrimitives, bbox);
}

void BVH::recursiveBuild(BVHTreeNode *node,
                        std::vector<BVHPrimitiveInfo> &buildData, size_t start,
                        size_t end, std::vector<Triangle> &orderedPrims) {
    // Update total nodes
}

```

```

++this->totalNodes;

// Compute bounds of all primitives in BVH node
BBox bbox;
for (size_t i = start; i < end; ++i)
  bbox = Union(bbox, buildData[i].bounds);
size_t nPrimitives = end - start;

if (nPrimitives < PRIMITIVES_PER_NODE) {
  createLeaf(node, buildData, start, end, orderedPrims, bbox,
             static_cast<unsigned char>(nPrimitives));
  return;
}

// Compute bound of primitive centroids, choose split dimension dim
BBox centroidBounds;
for (size_t i = start; i < end; ++i)
  centroidBounds = Union(centroidBounds, buildData[i].centroid);
auto dim = centroidBounds.MaximumExtent();

// Partition primitives into two sets and build children
size_t mid = (start + end + 1) / 2; // Truncates towards zero
if (centroidBounds.pMax[dim] != centroidBounds.pMin[dim] && nPrimitives > 2) {
  // Partition primitives through node's midpoint
  float pmid = .5f * (centroidBounds.pMin[dim] + centroidBounds.pMax[dim]);
  BVHPrimitiveInfo *midPtr = std::partition(
    &buildData[start], &buildData[end - 1] + 1,
    [&pmid, &dim](const auto &a) { return a.centroid[dim] < pmid; });
  mid = midPtr - &buildData[0];
}
node->createChildren();
recursiveBuild(node->getChild0(), buildData, start, mid, orderedPrims);
recursiveBuild(node->getChild1(), buildData, mid, end, orderedPrims);
node->InitInterior(dim);
}

```

11.5.4 bvhlinarnode.hpp

```

/****************************************************************************
 * BVHLinearNode
 *
 * Contains a node of the "linear" BVH structure,
 * prepared to be stored in an array and passed to
 * OpenCL
 *
 * 2017 - Liberto Camús
 * ****
 */

#ifndef BVHLINEARNODE_H
#define BVHLINEARNODE_H

#include "CL/cl.h"

class BVHLinearNode {
public:
  // BBox bounds;
  cl_float3 pMin;           // 16 bytes
  cl_float3 pMax;           // 16 bytes
  union {
    cl_uint primitivesOffset; // leaf
    cl_uint secondChildOffset; // interior
  };
  cl_uchar nPrimitives; // 0 -> Interior . 1 byte
  cl_uchar axis;        // interior node: xyz. 1 byte
};

#endif // BVHLINEARNODE_H

```

11.5.5 bvhprimitiveinfo.hpp

```
*****
 * BVHPrimitiveInfo
 *
 * Contains info about a primitive (triangle):
 * Its number in a "primitives" array, its centroid and
 * bounding box.
 * Used to build the BVH tree
 *
 * 2017 - Liberto Camús
 *****/
#ifndef BVHPRIMITIVEINFO_H
#define BVHPRIMITIVEINFO_H

#include "bbox.hpp"
#include <glm/glm.hpp>

class BVHPrimitiveInfo {
public:
    BVHPrimitiveInfo(size_t pn, const BBox &b) : primitiveNumber(pn), bounds(b) {
        centroid = .5f * b.pMin + .5f * b.pMax;
    }
    size_t primitiveNumber;
    glm::vec3 centroid;
    BBox bounds;
};

#endif // BVHPRIMITIVEINFO_H
```

11.5.6 camera.hpp

```
*****
 * Camera
 *
 * Controls camera position, orientation and related
 * matrices (Projection and View).
 *
 * Includes methods for moving, zooming and changing
 * orientation.
 *
 * 2017 - Liberto Camús
 *****/
#ifndef CAMERA_H
#define CAMERA_H

#include "config.hpp"

#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

```


/***
 * @brief The Camera class
 */
class Camera {
private:
    /// Constants
    static float constexpr MIN_PITCH{-89.0f};
    static float constexpr MAX_PITCH{89.0f};
    static float constexpr MIN_FOVY{1.0f};
    static float constexpr MAX_FOVY{45.0f};

    /// Camera state
    float fovy;           /// Vertical field of view angle
    float z_near, z_far;  /// Min and Max distance to cull
    glm::vec3 position;   /// Position
    GLfloat yaw, pitch;  /// Orientation, Euler angles
    GLfloat mov_speed;   /// Camera options

    /// Calculated from state
    glm::vec3 front_vector, up_vector, right_vector; /// Unit vectors
    glm::mat4 view_matrix, proj_matrix;               /// Matrices

    /// Update calculated properties
    void update_vectors();

public:
    /// Possible movement directions
    enum camera_movement { FORWARD, BACKWARD, LEFT, RIGHT };

    /// Constructor
    Camera(glm::vec3 position = glm::vec3(Config::camera_position.x,
                                            Config::camera_position.y,
                                            Config::camera_position.z),
           GLfloat yaw = Config::camera_yaw, GLfloat pitch = Config::camera_pitch)
        : fovy(Config::movement_zoom), z_near(Config::camera_znear),
          z_far(Config::camera_zfar), position(position), yaw(yaw), pitch(pitch),
          mov_speed(Config::movement_speed / 1000.0f), // convert to units/ms
          front_vector(glm::vec3(0.0f, 0.0f, -1.0f)),
          view_matrix(glm::lookAt(this->position,
                                 this->position + this->front_vector,
                                 this->up_vector)),
          proj_matrix(glm::perspective(glm::radians(fovy), Config::window_ratio,
                                       z_near, z_far)) {
        this->update_vectors();
    }

    /// Movement methods
    void change_fovy(GLfloat delta_fovy);
    void rotate(GLfloat delta_yaw, GLfloat delta_pitch);
    void move(camera_movement direction, GLfloat delta_time);

    /// Information methods
    const glm::mat4 &getViewMatrix() const { return view_matrix; }
    const glm::mat4 &getProjectionMatrix() const { return proj_matrix; }
    const glm::vec3 &getPosition() const { return position; }
};

#endif // CAMERA_H


```

11.5.7 camera.cpp

```


*****
* Camera
*
* Controls camera position, orientation and related
* matrices (Projection and View).
*
* Includes methods for moving, zooming and changing
* orientation.
*


```

```

* 2017 - Liberto Camús
* ****

#include "camera.hpp"
#include <iostream>

<**
 * @brief Camera::move Update Camera position
 * Processes input received from any keyboard-like input system. Accepts input
 * parameter in the form of camera defined ENUM (to abstract it from windowing
 * systems
 * @param direction camera defined enumeration
 * @param delta_time time in milliseconds since last movement
 */
void Camera::move(camera_movement direction, GLfloat delta_time) {
    glm::vec3 movement_vector;

    if (direction == FORWARD)
        movement_vector = this->front_vector;
    else if (direction == BACKWARD)
        movement_vector = -this->front_vector;
    else if (direction == LEFT)
        movement_vector = -this->right_vector;
    else if (direction == RIGHT)
        movement_vector = this->right_vector;

    this->position += this->mov_speed * delta_time * movement_vector;
    this->update_vectors();
}

// 
<**
 * @brief Camera::rotate Rotate Camera
 * Processes input received from a mouse input system. Expects the offset value
 * in both the x and y direction.
 * @param delta_yaw
 * @param delta_pitch
 */
void Camera::rotate(GLfloat delta_yaw, GLfloat delta_pitch) {
    // Update Camera's Euler angles
    this->yaw += delta_yaw;
    // Make sure that when pitch is out of bounds, screen doesn't get flipped
    this->pitch = glm::clamp(this->pitch + delta_pitch, MIN_PITCH, MAX_PITCH);

    // Update Front, Right and Up Vectors using the updated Euler angles
    this->update_vectors();
}

<**
 * @brief Camera::change_fovy
 * Changes the vertical field of view angle. Equivalent to zooming
 * @param delta_fovy Signed value to shrink or stretch angle (sexagesimal)
 * Valid angle values are 1 to 45
 */
void Camera::change_fovy(GLfloat delta_fovy) {
    /// Update field of view angle. Clamp to 1-45 degrees
    this->fovy = glm::clamp(this->fovy - delta_fovy, MIN_FOVY, MAX_FOVY);

    /// Update projection matrix with new fov angle
    proj_matrix =
        glm::perspective(glm::radians(fovy), Config::window_ratio, z_near, z_far);
}

<**
 * @brief Camera::update_vectors
 * Updates viewMatrix and the front, right and up vector based on
 * the Camera's Euler angles
 */
void Camera::update_vectors() {
    constexpr glm::vec3 upVector = glm::vec3(0.0f, 1.0f, 0.0f);
}

```

```

/// Calculate the new Front vector
glm::vec3 front = {
    cos(glm::radians(this->yaw)) * cos(glm::radians(this->pitch)),
    sin(glm::radians(this->pitch)),
    sin(glm::radians(this->yaw)) * cos(glm::radians(this->pitch))};
this->front_vector = glm::normalize(front);

/// Also re-calculate the Right and Up vector
this->right_vector = glm::normalize(glm::cross(this->front_vector, upVector));
this->up_vector =
    glm::normalize(glm::cross(this->right_vector, this->front_vector));

/// Update view matrix
this->view_matrix = glm::lookAt(
    this->position, this->position + this->front_vector, this->up_vector);
}
  
```

11.5.8 cl_device.hpp

```

*****
* CL_Device
*
* Obtains information about OpenCL devices in the
* platform
*
* 2017 - Liberto Camús
*****
#ifndef CL_DEVICE_HPP
#define CL_DEVICE_HPP
#include <string>
#include <vector>

#include "CL/cl.h"

class CL_Device {
    std::string name;
    cl_device_id device_id;

public:
    explicit CL_Device(cl_device_id);
    CL_Device() : device_id(0) {}

    const cl_device_id &getId() const { return device_id; }
    const std::string &getName() const { return name; }

    static std::vector<cl_device_id> get_devices_ids(cl_platform_id platform_id);
};

#endif // CL_DEVICE_HPP
  
```

11.5.9 cl_device.cpp

```

*****
* CL_Device
*
* Obtains information about OpenCL devices in the
* system
*
* 2017 - Liberto Camús
*****
#include "cl_device.hpp"
#include <iostream>
  
```

```
#include <string>

<**
 * @brief CL_Device::CL_Device
 * Obtains information about a specified OpenCL device
 * @param id specific OpenCL device
 */
CL_Device::CL_Device(cl_device_id id) : device_id{id} {
    size_t arg_size;
    clGetDeviceInfo(device_id, CL_DEVICE_NAME, 0, NULL, &arg_size);
    name.resize(arg_size);
    clGetDeviceInfo(device_id, CL_DEVICE_NAME, arg_size, &name[0], NULL);
}

<**
 * @brief CL_Device::get_devices_ids
 * Obtains OpenCL devices in a specified platform
 * @param platform_id OpenCL id of the platform
 * @return Vector of identifiers of OpenCL devices
 */
std::vector<cl_device_id>
CL_Device::get_devices_ids(cl_platform_id platform_id) {
    cl_uint n_dev;
    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 0, nullptr, &n_dev);
    if (n_dev == 0) {
        std::cout << "No GPUs detected. Falling back to CPU" << endl;
        clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 0, nullptr, &n_dev);
        auto ids = std::vector<cl_device_id>(n_dev);
        clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, n_dev, ids.data(), nullptr);
        return ids;
    } else {
        auto ids = std::vector<cl_device_id>(n_dev);
        clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, n_dev, ids.data(), nullptr);
        return ids;
    }
}
```

11.5.10 clkernelmanager.hpp

```
*****
* CLKernelManager
*
* Encapsulates OpenCL initialization and functions
* Also loads and compiles a kernel.
*
* 2017 - Liberto Camús
*****
#ifndef CL_KERNELMANAGER_H
#define CL_KERNELMANAGER_H

#include "CL/cl.h"
#include "cl_device.hpp"
#include "cl_platform.hpp"
#include <CL/cl_gl_ext.h>
#include <GL/glew.h> // for GLuint
#include <string>

class CLKernelManager {
private:
    CL_Platform platform;
    CL_Device device;
    cl_context ctxt;
    cl_command_queue cmdQueue;
    cl_program program;
    cl_kernel kernel;
    bool checkForCLGLSharing(cl_device_id dev_id);
    std::string readFile(const std::string &path);
    bool createSharedContext(cl_platform_id plat_id);
```

```

public:
    CLKernelManager();
    ~CLKernelManager();

    cl_mem createFromGLTexture(GLuint GLtexture, cl_mem_flags mem_flags,
                               const std::string &str);
    cl_mem createFromGLBuffer(GLuint GLBuffer, cl_mem_flags mem_flags,
                               const std::string &str);
    cl_mem createBufferReadOnly(size_t size, void *ptr) {
        return clCreateBuffer(ctx, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size,
                             ptr, nullptr);
    }
    cl_mem createBuffer(size_t size, void *ptr) {
        return clCreateBuffer(ctx, CL_MEM_READ_WRITE, size, ptr, nullptr);
    }

    void enqueueAcquireGLObjects(uint count, const cl_mem *cl_shared_objects) {
        clEnqueueAcquireGLObjects(cmdQueue, count, cl_shared_objects, 0, nullptr,
                                  nullptr);
    }
    void enqueueReleaseGLObjects(uint count, const cl_mem *cl_shared_objects) {
        clEnqueueReleaseGLObjects(cmdQueue, count, cl_shared_objects, 0, nullptr,
                                  nullptr);
    }
    void finish() { clFinish(cmdQueue); }

    void loadKernelFromFile(const std::string &path);
    void setKernelArg(cl_uint index, size_t size, const void *value);
    void executeKernel(cl_event *event, size_t third);

};

#endif // CL_KERNELMANAGER_H

```

11.5.11 clkernelmanager.cpp

```

*****
* CLKernelManager
*
* Encapsulates OpenCL initialization and functions
* Also loads and compiles a kernel.
*
* 2017 - Liberto Camús
*****
#include "clkernelmanager.hpp"

#if defined(_WIN32)
#define GLFW_EXPOSE_NATIVE_WIN32
#define GLFW_EXPOSE_NATIVE_WGL
#define OVR_OS_WIN32
#elif defined(__APPLE__)
#define GLFW_EXPOSE_NATIVE_COCOA
#define GLFW_EXPOSE_NATIVE_NSGL
#define OVR_OS_MAC
#elif defined(__linux__)
#define GLFW_EXPOSE_NATIVE_X11
#define GLFW_EXPOSE_NATIVE_GLX
#define OVR_OS_LINUX
#endif

#include <fstream>
#include <iostream>
#include <sstream>

#include "cl_device.hpp"
#include "config.hpp"
#include <CL/cl_gl.h>
#include <CL/cl_gl_ext.h>
#include <GLFW/glfw3.h>
#include <GLFW/glfw3native.h>

```

```
#include <cstring>
#include <string>

// Use extension cl_khr_gl_sharing
typedef CL_API_ENTRY cl_int(CL_API_CALL *clGetGLContextInfoKHR_fn)(
    const cl_context_properties *properties, cl_gl_context_info param_name,
    size_t param_value_size, void *param_value, size_t *param_value_size_ret);

// Rename references to this dynamically linked function to avoid
// collision with static link version
#define clGetGLContextInfoKHR clGetGLContextInfoKHR_proc
static clGetGLContextInfoKHR_fn clGetGLContextInfoKHR;

/***
 * @brief CLKernelManager::CLKernelManager
 */
CLKernelManager::CLKernelManager() {
    Config::option_opencl_available = false;

    auto plat_ids = CL_Platform::get_platforms_ids();
    if (plat_ids.size() == 0) {
        std::cout << "No OpenCL platforms detected\n";
        return;
    }

    for (cl_platform_id &plat_id : plat_ids) {
        std::cout << "Detecting devices on platform \n";
        CL_Platform plat(plat_id);

        // Try to find a device which supports cl_khr_gl_sharing in this
        // platform
        bool platform_has_capable_devices = false;
        for (cl_device_id &dev_id : CL_Device::get_devices_ids(plat_id)) {
            CL_Device dev(dev_id);
            if (checkForCLGLSharing(dev_id) == true) {
                platform_has_capable_devices = true;
                break;
            }
        }

        // Didn't find one. Skip this platform
        if (!platform_has_capable_devices) {
            std::cout << "Couldn't find extension cl_khr_gl_sharing in platform.\n";
            continue;
        }

        // Try to get proc address for clGetGLContextInfoKHR
        if (!clGetGLContextInfoKHR) {
            clGetGLContextInfoKHR =
                (clGetGLContextInfoKHR_fn)clGetExtensionFunctionAddressForPlatform(
                    plat_id, "clGetGLContextInfoKHR");
            if (!clGetGLContextInfoKHR) {
                std::cout << "Failed to query proc address for clGetGLContextInfoKHR";
                continue;
            }
        }

        // Try to create a shared context with OpenGL
        if (!createSharedContext(plat_id))
            continue;

        // Create Command Queue
        cl_int status;
        cmdQueue =
            clCreateCommandQueueWithProperties(ctx, device.getId(), 0, &status);
        if (status != CL_SUCCESS) {
            std::cout << "Error creating OpenCL command queue\n";
            continue;
        }
        Config::option_opencl_available = true;
    }
}
```

```

        break;
    }

/***
 * @brief CLKernelManager::createSharedContext
 * @param plat_id
 * @return
 */
bool CLKernelManager::createSharedContext(cl_platform_id plat_id) {

#ifndef linux
    cl_context_properties properties[] = {
        CL_GL_CONTEXT_KHR,
        (cl_context_properties)glfwGetGLXContext(glfwGetCurrentContext()),
        CL_GLX_DISPLAY_KHR,
        (cl_context_properties)glfwGetX11Display(),
        CL_CONTEXT_PLATFORM,
        (cl_context_properties)plat_id,
        0};
#elif defined _WIN32
    cl_context_properties properties[] = {
        CL_GL_CONTEXT_KHR,
        (cl_context_properties)wglGetCurrentContext(),
        CL_WGL_HDC_KHR,
        (cl_context_properties)wglGetCurrentDC(),
        CL_CONTEXT_PLATFORM,
        (cl_context_properties)plat_id,
        0};
#elif defined TARGET_OS_MAC
    CGLContextObj glContext = CGLGetCurrentContext();
    CGLShareGroupObj shareGroup = CGLGetShareGroup(glContext);
    cl_context_properties properties[] = {
        CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE,
        (cl_context_properties)shareGroup, 0};
#endif

    cl_device_id cl_dev;
    size_t size;
    cl_int result =
        clGetGLContextInfoKHR(properties, CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
                              sizeof(cl_device_id), &cl_dev, &size);
    if (result != CL_SUCCESS) {
        std::cout << "Failed to obtain OpenCL Device from GL Device\n";
        return false;
    }

    device = CL_Device(cl_dev);
    cl_int status;
    this->ctxt = clCreateContext(properties, 1, &device.getId(), nullptr, nullptr,
                                 &status);

    return status == CL_SUCCESS;
}

cl_mem CLKernelManager::createFromGLBuffer(GLuint GLBuffer,
                                           cl_mem_flags mem_flags,
                                           const std::string &str) {
    cl_int err;
    cl_mem buffer = clCreateFromGLBuffer(ctxt, mem_flags, GLBuffer, &err);
    if (!buffer || err != CL_SUCCESS) {
        std::cout << "Error acquiring CL texture from OpenGL " << str << "\n";
    }

    return buffer;
}

cl_mem CLKernelManager::createFromGLTexture(GLuint GLtexture,
                                             cl_mem_flags mem_flags,
                                             const std::string &str) {
    cl_int err;

```

```

cl_mem texture =
    clCreateFromGLTexture(ctx, mem_flags, GL_TEXTURE_2D, 0, GLtexture, &err);
if (!texture || err != CL_SUCCESS) {
    std::cout << "Error acquiring CL texture from OpenGL " << str << "\n";
    switch (err) {
        case CL_INVALID_CONTEXT:
            std::cout << "CL_INVALID_CONTEXT\n";
            break;
        case CL_INVALID_VALUE:
            std::cout << "CL_INVALID_VALUE\n";
            break;
        case CL_INVALID_MIP_LEVEL:
            std::cout << "CL_INVALID_MIP_LEVEL\n";
            break;
        case CL_INVALID_GL_OBJECT:
            std::cout << "CL_INVALID_GL_OBJECT\n";
            break;
        case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
            std::cout << "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR\n";
            break;
        case CL_INVALID_OPERATION:
            std::cout << "CL_INVALID_OPERATION\n";
            break;
        case CL_OUT_OF_RESOURCES:
            std::cout << "CL_OUT_OF_RESOURCES\n";
            break;
        case CL_OUT_OF_HOST_MEMORY:
            std::cout << "CL_OUT_OF_HOST_MEMORY\n";
            break;
    }
    exit(-1);
}
return texture;
}

bool CLKernelManager::checkForCLGLSharing(cl_device_id dev_id) {
    std::string extension_string(1024, ' ');
    cl_int status =
        clGetDeviceInfo(dev_id, CL_DEVICE_EXTENSIONS, 1024 * sizeof(char),
                        (void *)extension_string.data(), nullptr);

    if (status != CL_SUCCESS) {
        std::cout << "Error querying for cl_khr_gl_sharing extension\n";
        return false;
    }

    std::string::size_type n = extension_string.find("cl_khr_gl_sharing");
    if (n == std::string::npos)
        return false;

    return true;
}

CLKernelManager::~CLKernelManager() {
    if (Config::option_openc1_available) {
        clReleaseProgram(program);
        clReleaseKernel(kernel);
        clReleaseCommandQueue(cmdQueue);
        clReleaseContext(ctx);
    }
}

void CLKernelManager::loadKernelFromFile(const std::string &path) {
    std::string code = readFile(path);
    size_t p_source_len = code.size();
    cl_int status;
    const char *data = code.data();
    program = clCreateProgramWithSource(ctx, 1, &data, &p_source_len, &status);

    if (status != CL_SUCCESS) {
        std::cout << "Error creating program: " << path << "\n";
        return;
    }
}

```

Anexos

```

}

status = clBuildProgram(program, 1, &device.getId(), "-cl-fast-relaxed-math",
                       nullptr, nullptr);
if (status != CL_SUCCESS) {
    std::cout << "Error building program: " << path << "\n";

    size_t buildLogSize = 0;
    clGetProgramBuildInfo(program, device.getId(), CL_PROGRAM_BUILD_LOG,
                          buildLogSize, nullptr, &buildLogSize);

    auto buildLog = std::make_unique<char[]>(buildLogSize);

    clGetProgramBuildInfo(program, device.getId(), CL_PROGRAM_BUILD_LOG,
                          buildLogSize, buildLog.get(), nullptr);

    std::cout << buildLog.get() << std::endl;
    std::exit(-1);
}

status = clCreateKernelsInProgram(program, 1, &kernel, nullptr);
if (status != CL_SUCCESS) {
    std::cout << "Unable to create kernel from program\n";
    return;
}

std::cout << "OpenCL kernel loaded:" << path << "\n";
}

void CLKernelManager::setKernelArg(cl_uint index, size_t size,
                                   const void *value) {
    clSetKernelArg(kernel, index, size, value);
}

void CLKernelManager::executeKernel(cl_event *event, size_t third) {
    size_t globalWorkSize[] = {Config::window_width, Config::window_height,
                             third};
    // size_t localWorkSize[] = {1, 1, third};
    cl_int status =
        clEnqueueNDRangeKernel(cmdQueue, kernel, 3, nullptr, globalWorkSize,
                               nullptr, // localWorkSize,
                               0, 0, event);
    if (status != CL_SUCCESS) {
        std::cout << "Error running kernel\n";
    }
}

std::string CLKernelManager::readFile(const std::string &path) {
    std::string code{""};
    try {
        std::ifstream file;
        file.exceptions(std::ifstream::failbit | std::ifstream::badbit);
        file.open(path);
        std::stringstream fStream;
        fStream << file.rdbuf();
        file.close();
        code = fStream.str();
    } catch (std::ifstream::failure &e) {
        std::cerr << "Error reading OpenCL file \n";
        std::cerr << "File: " << path << std::endl;
        std::exit(1);
    }
    return code;
}

```

11.5.12 cl_platform.hpp

```
*****
* CL_Platform
*
* Obtains information about OpenCL platforms in the
* system
*
* 2017 - Liberto Camús
* ****
#ifndef CL_PLATFORM_H
#define CL_PLATFORM_H
#include <memory>
#include <vector>

#include "CL/cl.h"

class CL_Platform {
private:
    cl_platform_id platform_id;
    std::string vendor_name;
    std::string name;
    std::string version;
    std::string profile;

public:
    CL_Platform(cl_platform_id id);
    CL_Platform() : platform_id(0) {}
    static std::vector<cl_platform_id> get_platforms_ids();
};

#endif // CL_PLATFORM_H
```

11.5.13 cl_platform.cpp

```
*****
* CL_Platform
*
* Obtains information about OpenCL platforms in the
* system
*
* 2017 - Liberto Camús
* ****
#include "cl_platform.hpp"
#include <iostream>

std::vector<cl_platform_id> CL_Platform::get_platforms_ids() {
    cl_uint n_pat;
    cl_int status = clGetPlatformIDs(0, nullptr, &n_pat);
    if (status != CL_SUCCESS) {
        std::cout << "Couldn't find any OpenCL platforms." << std::endl;
        return std::vector<cl_platform_id>();
    }

    std::cout << "Found " << n_pat << " OpenCL platforms." << std::endl;
    auto ids = std::vector<cl_platform_id>(n_pat);
    clGetPlatformIDs(n_pat, ids.data(), nullptr);

    return ids;
}

CL_Platform::CL_Platform(cl_platform_id id) : platform_id{id} {
    size_t arg_size;
    // PLATFORM ID
```

```

std::cout << "Platform ID:" << id << std::endl;

// PLATFORM NAME
clGetPlatformInfo(platform_id, CL_PLATFORM_NAME, 0, nullptr, &arg_size);
name.resize(arg_size);
clGetPlatformInfo(platform_id, CL_PLATFORM_NAME, arg_size, &name.front(),
                  nullptr);
std::cout << "OpenCL platform:" << name << std::endl;

// VENDOR NAME
clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR, 0, nullptr, &arg_size);
vendor_name.resize(arg_size);
clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR, arg_size,
                  &vendor_name.front(), nullptr);
std::cout << "OpenCL vendor:" << vendor_name << std::endl;

// PROFILE
clGetPlatformInfo(platform_id, CL_PLATFORM_PROFILE, 0, nullptr, &arg_size);
profile.resize(arg_size);
clGetPlatformInfo(platform_id, CL_PLATFORM_PROFILE, arg_size,
                  &profile.front(), nullptr);
std::cout << "OpenCL profile:" << profile << std::endl;

// VERSION
clGetPlatformInfo(platform_id, CL_PLATFORM_VERSION, 0, nullptr, &arg_size);
version.resize(arg_size);
clGetPlatformInfo(platform_id, CL_PLATFORM_VERSION, arg_size,
                  &version.front(), nullptr);
std::cout << "OpenCL version:" << version << std::endl;
}

```

11.5.14 config.hpp

```

*****
* Config
*
* Configuration and scene descripton values
*
* 2017 - Liberto Camús
*****
#ifndef CONFIG_H
#define CONFIG_H
#include <glm/glm.hpp>
#include <string>
#include <vector>

class Config {
public:
    enum RENDERING_METHOD { HYBRID, DEFERRED, HYBRID_CPU };

    // window
    static unsigned int window_width;
    static unsigned int window_height;
    static float window_ratio;

    // Shaders / kernels
    static std::string gbuffer_shader_vert;
    static std::string gbuffer_shader_frag;
    static std::string lighting_shader_vert;
    static std::string lighting_shader_frag;
    static std::string lighting_kernel;

    // Camera
    static glm::vec3 camera_position;
    static float camera_znear;
};

```

```

static float camera_zfar;
static float camera_pitch;
static float camera_yaw;

// Movement
static float movement_speed;
static float movement_zoom;
static float movement_fovy_step;
static float movement_sensitivity;

// Scene
static std::vector<std::string> models;
static std::vector<std::vector<float>> point_lights;
static float ambient;

// Options
static bool option_shadows_enabled;
static bool option_normal_mapping_enabled;
static bool option_rendering_method_change_requested;
static bool option_no_capture_mouse;
static bool option_opengl_available;
static bool option_statistics_requested;
static bool option_reset_statistics_requested;

// Rendering method
static RENDERING_METHOD rendering_method;
};

#endif // CONFIG_H

```

11.5.15 configloader.hpp

```

*****
* ConfigLoader
*
* Initialization of "Config.hpp" values. Reads and
* parses configuration files.
*
* 2017 - Liberto Camús
*****
```

```

#ifndef CONFIGLOADER_H
#define CONFIGLOADER_H
#include <string>
#include <vector>

class ConfigLoader {
public:
    ConfigLoader(const std::string &file);
    ~ConfigLoader();

    enum CONFIG_LOADER_POINT_LIGHT {

    };

private:
    bool setWindow(const std::vector<std::string> &);
    bool setModel(const std::vector<std::string> &);
    bool setGBufferShader(const std::vector<std::string> &);
    bool setLightingShader(const std::vector<std::string> &);
    bool setLightingKernel(const std::vector<std::string> &);
    bool setCamera(const std::vector<std::string> &);
    bool setMovement(const std::vector<std::string> &);
    bool setPointLight(const std::vector<std::string> &);
};

```

```

bool setAmbientLightIntensity(const std::vector<std::string> &);

std::vector<std::string> readFile(const std::string &path);
};

#endif // CONFIGLOADER_H

```

11.5.16 configloader.cpp

```

*****
* ConfigLoader
*
* Initialization of "Config.hpp" values. Reads and
* parses configuration files.
*
* 2017 - Liberto Camús
*****
```

```

#include "configloader.hpp"
#include "config.hpp"

#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>

// Window
uint Config::window_width = 0;
uint Config::window_height = 0;
float Config::window_ratio = 0.0f;

// Shaders / kernels
std::string Config::gbuffer_shader_vert = "";
std::string Config::gbuffer_shader_frag = "";
std::string Config::lighting_shader_vert = "";
std::string Config::lighting_shader_frag = "";
std::string Config::lighting_kernel = "";

// Camera
glm::vec3 Config::camera_position = glm::vec3{0.0f, 0.0f, 0.0f};
float Config::camera_znear = 0.0f;
float Config::camera_zfar = 0.0f;
float Config::camera_pitch = 0.0f;
float Config::camera_yaw = 0.0f;

// Movement
float Config::movement_speed = 0.0f;
float Config::movement_zoom = 0.0f;
float Config::movement_sensitivity = 0.0f;
float Config::movement_fovy_step = 3.0f;

// Scene
std::vector<std::string> Config::models;
std::vector<std::vector<float>> Config::point_lights;
float Config::ambient = 0.0f;

// Runtime Options
bool Config::option_shadows_enabled = true;
bool Config::option_normal_mapping_enabled = false;
bool Config::option_rendering_method_change_requested = false;

```

```

bool Config::option_no_capture_mouse = false;
bool Config::option_opencl_available = true;
bool Config::option_statistics_requested = false;
bool Config::option_reset_statistics_requested = false;

// Rendering Method
Config::RENDERING_METHOD Config::rendering_method = Config::HYBRID;

ConfigLoader::ConfigLoader(const std::string &file) {
    std::cout << "Reading configuration file: " << file << "\n";
    std::vector<std::string> data = readFile(file);

    unsigned int line_nr = 0;
    bool file_ok = true;
    for (std::string &line : data) {
        ++line_nr;
        if (line.size() < 2)
            continue;
        if (line[0] == '/' & line[1] == '/')
            continue;

        std::vector<std::string> result;
        std::istringstream iss(line);
        for (std::string t; iss >> t;)
            result.push_back(t);

        if (result.size() == 0)
            continue;
        std::string title = result.at(0);

        if (title.compare("window") == 0) {
            file_ok = setWindow(result);
        } else if (title.compare("model") == 0) {
            file_ok = setModel(result);
        } else if (title.compare("gbuffer_shader") == 0) {
            file_ok = setGBufferShader(result);
        } else if (title.compare("lighting_shader") == 0) {
            file_ok = setLightingShader(result);
        } else if (title.compare("lighting_kernel") == 0) {
            file_ok = setLightingKernel(result);
        } else if (title.compare("camera") == 0) {
            file_ok = setCamera(result);
        } else if (title.compare("movement") == 0) {
            file_ok = setMovement(result);
        } else if (title.compare("ambient") == 0) {
            file_ok = setAmbientLightIntensity(result);
        } else if (title.compare("pointlight") == 0) {
            file_ok = setPointLight(result);
        }
    }
    if (!file_ok) {
        std::cout << "Error reading configuration file " << file << "\n";
        std::cout << " in line " << line_nr << "\n";
        exit(1);
    }
}

bool ConfigLoader::setWindow(const std::vector<std::string> &v) {
    if (v.size() < 3) {
        std::cout << "Too few arguments for window size. Expected \n";
        std::cout << "Window <width> <height> \n";
        return false;
    }
    Config::window_width = std::atoi(v.at(1).c_str());
    Config::window_height = std::atoi(v.at(2).c_str());
    Config::window_ratio =
        (float)Config::window_width / (float)Config::window_height;

    return true;
}

bool ConfigLoader::setModel(const std::vector<std::string> &v) {

```

```

if (v.size() < 2) {
    std::cout << "Too few arguments for Model number. Expected \n";
    std::cout << "model <path> \n";
    return false;
}
Config::models.push_back(v.at(1));
return true;
}

bool ConfigLoader::setGBufferShader(const std::vector<std::string> &v) {
    if (v.size() < 2) {
        std::cout << "Too few arguments for GBuffer shader name. Expected \n";
        std::cout << "gbuffer_shader <name> \n";
        return false;
    }
    Config::gbuffer_shader_frag = v.at(1) + ".frag";
    Config::gbuffer_shader_vert = v.at(1) + ".vert";
    return true;
}

bool ConfigLoader::setLightingShader(const std::vector<std::string> &v) {
    if (v.size() < 2) {
        std::cout << "Too few arguments for lighting shader name. Expected \n";
        std::cout << "lighting_shader <name> \n";
        return false;
    }
    Config::lighting_shader_frag = v.at(1) + ".frag";
    Config::lighting_shader_vert = v.at(1) + ".vert";
    return true;
}

bool ConfigLoader::setLightingKernel(const std::vector<std::string> &v) {
    if (v.size() < 2) {
        std::cout << "Too few arguments for lighting kernel name. Expected \n";
        std::cout << "lighting_kernel <name> \n";
        return false;
    }
    Config::lighting_kernel = v.at(1);
    return true;
}

bool ConfigLoader::setCamera(const std::vector<std::string> &v) {
    enum pos : size_t {
        pos_x = 1, pos_y, pos_z, znear, zfar, pitch, yaw, pos_end
    };
    if (v.size() < pos_end) {
        std::cout << "Too few arguments for camera parameters. Expected \n";
        std::cout << "camera <pos_x> <pos_y> <pos_z> <znear> <zfar> <pitch> <yaw> \n";
        return false;
    }
    Config::camera_position = glm::vec3((float)std::atof(v.at(pos_x).c_str()),
                                         (float)std::atof(v.at(pos_y).c_str()),
                                         (float)std::atof(v.at(pos_z).c_str()));
    Config::camera_znear = (float)std::atof(v.at(znear).c_str());
    Config::camera_zfar = (float)std::atof(v.at(zfar).c_str());
    Config::camera_pitch = (float)std::atof(v.at(pitch).c_str());
    Config::camera_yaw = (float)std::atof(v.at(yaw).c_str());
    return true;
}

bool ConfigLoader::setMovement(const std::vector<std::string> &v) {
    enum pos : size_t { speed = 1, zoom, sensitivity, pos_end };
    if (v.size() < pos_end) {
        std::cout << "Too few arguments for movement parameters. Expected \n";
        std::cout << "movement <speed> <zoom> <sensitivity>\n";
        return false;
    }

    Config::movement_speed = (float)std::atof(v.at(speed).c_str());
    Config::movement_zoom = (float)std::atof(v.at(zoom).c_str());
    Config::movement_sensitivity = (float)std::atof(v.at(sensitivity).c_str());
    return true;
}

```

```

bool ConfigLoader::setPointLight(const std::vector<std::string> &v) {
    enum pos : size_t {
        pos_x = 1, pos_y, pos_z, color_r, color_g, color_b, intensity, linear,
        quadratic, pos_end };
    if (v.size() < pos_end) {
        std::cout << "Too few arguments for point light parameters. Expected \n";
        std::cout << "pointlight <pos_x> <pos_y> <pos_z> <color_r> <color_g> "
              << "<color_b> <intensity> <linear> <quadratic>\n";
        return false;
    }

    std::vector<float> v_f;
    v_f.push_back((float)std::atof(v.at(pos_x).c_str())); // pos_x
    v_f.push_back((float)std::atof(v.at(pos_y).c_str())); // pos_y
    v_f.push_back((float)std::atof(v.at(pos_z).c_str())); // pos_z
    v_f.push_back((float)std::atof(v.at(color_r).c_str())); // color_r
    v_f.push_back((float)std::atof(v.at(color_g).c_str())); // color_g
    v_f.push_back((float)std::atof(v.at(color_b).c_str())); // color_b
    v_f.push_back((float)std::atof(v.at(intensity).c_str())); // intensity
    v_f.push_back((float)std::atof(v.at(linear).c_str())); // linear
    v_f.push_back((float)std::atof(v.at(quadratic).c_str())); // quadratic

    Config::point_lights.emplace_back(v_f);
    return true;
}

bool ConfigLoader::setAmbientLightIntensity(const std::vector<std::string> &v) {
    if (v.size() < 2) {
        std::cout << "Too few arguments for ambient value. Expected \n";
        std::cout << "ambient <number> \n";
        return false;
    }
    Config::ambient = (float)std::atof(v.at(1).c_str());
    return true;
}

std::vector<std::string> ConfigLoader::readFile(const std::string &path) {
    std::vector<std::string> res;
    std::string line;

    std::ifstream fich(path);
    if (!fich) {
        std::cout << "Error opening file:" << path << "\n";
        exit(-1);
    }
    while (std::getline(fich, line))
        res.push_back(line);

    return res;
}

ConfigLoader::~ConfigLoader() {}

```

11.5.17 deferredshader.hpp

```

*****
* _DeferredShader
*
* Renders the scene using OpenGL deferred shader
* algorithm. No OpenCL involved.
* One of the two alternative methods implemented
* (the other one is hybridshader)
*
* 2017 - Liberto Camús
*****
#ifndef DEFERREDSHADER_H
#define DEFERREDSHADER_H

```

```
#include "renderengine.hpp"
#include "shaderprogram.hpp" // for ShaderProgram
#include <GL/glew.h> // for GLuint
#include <vector> // for vector

class World;

class DeferredShader : public RenderEngine {
private:
    GLuint gBuffer; // Framebuffer
    GLuint gNormal, gPosition, gAlbedoSpec; // Color attachments
    GLuint rboDepth;
    GLuint quadVAO = 0;
    GLuint quadVBO;

    ShaderProgram gBufferShader;
    ShaderProgram lightingPassShader;
    World *world;

    void init_pass1_gBuffer();
    void init_pass2_lighting();
    void pass1_gBuffer();
    void pass2_lighting();

public:
    void render();
    // const ShaderProgram &getModelShader() { return gBufferShader; }

    DeferredShader(World *w);
    ~DeferredShader();
};

#endif // DEFERREDSHADER_H
```

11.5.18 deferredshader.cpp

```
*****
* DeferredShader
*
* Renders the scene using OpenGL deferred shader
* algorithm. No OpenCL involved.
* One of the two alternative methods implemented
* (the other one is hybridshader)
*
* 2017 - Liberto Camús
*****
#include "deferredshader.hpp"
#include "camera.hpp"
#include "model.hpp"
#include "shaderprogram.hpp"
#include "window.hpp"
#include "world.hpp"
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <iostream>
#include <string>

DeferredShader::DeferredShader(World *w)
: gBufferShader{"shaders/gbuffer.vert", "shaders/gbuffer.frag"}, 
  lightingPassShader{"shaders/lighting.vert", "shaders/lighting.frag"}, 
  world{w} {

    world->initModelsUniforms(gBufferShader);

    init_pass1_gBuffer();
    init_pass2_lighting();
```

}

```

DeferredShader::~DeferredShader() {
    glDeleteTextures(1, &gPosition);
    glDeleteTextures(1, &gNormal);
    glDeleteTextures(1, &gAlbedoSpec);
    glDeleteRenderbuffers(1, &rboDepth);
    glDeleteFramebuffers(1, &gBuffer);
}

// Initialize GBuffer -> textures(albedo+specular, normals, positions)
// and render buffer
void DeferredShader::init_pass1_gBuffer() {
    gBufferShader.use();
    glGenFramebuffers(1, &gBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);

    // - Position color buffer
    glGenTextures(1, &gPosition);
    glBindTexture(GL_TEXTURE_2D, gPosition);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_FLOAT, nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                          gPosition, 0);

    // - Normal color buffer
    glGenTextures(1, &gNormal);
    glBindTexture(GL_TEXTURE_2D, gNormal);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8,
                nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
                          gNormal, 0);

    // - Color + Specular color buffer
    glGenTextures(1, &gAlbedoSpec);
    glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8,
                nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D,
                          gAlbedoSpec, 0);

    // - Tell OpenGL which color attachments we'll use (of this framebuffer) for
    // rendering
    GLuint attachments[3] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
                            GL_COLOR_ATTACHMENT2};
    glDrawBuffers(3, attachments);
    glGenRenderbuffers(1, &rboDepth);
    glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                         Config::window_width, Config::window_height);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                             GL_RENDERBUFFER, rboDepth);

    // - Finally check if framebuffer is complete
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
        std::cout << "Framebuffer not complete!" << std::endl;
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

// Initialize lighting pass -> uniforms for textures
// quad for rendering final scene
// pass point lights information to the shader

```

```

void DeferredShader::init_pass2_lighting() {
    lightingPassShader.use();
    glUniform1i(lightingPassShader.getUniformLocation("gPosition"), 0);
    glUniform1i(lightingPassShader.getUniformLocation("gNormal"), 1);
    glUniform1i(lightingPassShader.getUniformLocation("gAlbedoSpec"), 2);
    glUniform1f(lightingPassShader.getUniformLocation("ambient"),
               Config::ambient);

    constexpr GLfloat quadVertices[] = {
        // Positions           // Texture Coords
        -1.0f,  1.0f,  0.0f,  0.0f,  1.0f,
        -1.0f, -1.0f,  0.0f,  0.0f,  0.0f,
        1.0f,  1.0f,  0.0f,  1.0f,  1.0f,
        1.0f, -1.0f,  0.0f,  1.0f,  0.0f,
    };
    // Setup plane VAO
    glGenVertexArrays(1, &quadVAO);
    glGenBuffers(1, &quadVBO);
    glBindVertexArray(quadVAO);
    glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), &quadVertices,
                 GL_STATIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat),
                          (GLvoid *)nullptr);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat),
                          (GLvoid *) (3 * sizeof(GLfloat)));
}

// Pass point light information to the shader
auto pLights = world->getPointLights();
glUniform1i(lightingPassShader.getUniformLocation("NR_LIGHTS"),
            static_cast<GLint>(world->getPointLightsNr()));
for (unsigned int i = 0; i < pLights.size(); i++) {
    PointLight p = pLights[i];
    glm::vec3 lightPos = {p.position.x, p.position.y, p.position.z};
    glm::vec3 lightCol = {p.color.x, p.color.y, p.color.z};

    glUniform3fv(lightingPassShader.getUniformLocation(
                    "lights[" + std::to_string(i) + "].Position"),
                 1, &lightPos[0]);
    glUniform3fv(lightingPassShader.getUniformLocation(
                    "lights[" + std::to_string(i) + "].Color"),
                 1, &lightCol[0]);
    glUniform1f(lightingPassShader.getUniformLocation(
                    "lights[" + std::to_string(i) + "].Linear"),
                p.linear);
    glUniform1f(lightingPassShader.getUniformLocation(
                    "lights[" + std::to_string(i) + "].Quadratic"),
                p.quadratic);
    glUniform1f(lightingPassShader.getUniformLocation(
                    "lights[" + std::to_string(i) + "].Intensity"),
                p.intensity);
}
}

// Main method, called once every frame
void DeferredShader::render() {
    pass1_gBuffer();
    pass2_lighting();
}

// Produces GBuffer textures
void DeferredShader::pass1_gBuffer() {
    glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
    glClearColor(0.05f, 0.05f, 0.05f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    gBufferShader.use();

    const Camera *camera = world->getCamera();
    const std::vector<Model> *models = world->getModels();
}

```

```

auto vMat = camera->getViewMatrix();
auto loc = gBufferShader.getUniformLocation("view");
glUniformMatrix4fv(loc, 1, GL_FALSE, glm::value_ptr(vMat));

auto projMatrix = camera->getProjectionMatrix();
auto projLoc = gBufferShader.getUniformLocation("projection");
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projMatrix));

for (auto &m : *models) {
    glUniformMatrix4fv(gBufferShader.getUniformLocation("model"), 1, GL_FALSE,
                       glm::value_ptr(m.getModelMatrix()));
    m.draw(gBufferShader);
}

// Renders the scene using light information and GBuffer textures
void DeferredShader::pass2_lighting() {
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    lightingPassShader.use();

    // Update camera position
    auto camPos = world->getCamera()->getPosition();
    glUniform3fv(lightingPassShader.getUniformLocation("viewPos"), 1, &camPos[0]);

    // Bind GBuffer textures as input
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, gPosition);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, gNormal);
    glActiveTexture(GL_TEXTURE2);
    glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);

    // Draw the scene
    glBindVertexArray(quadVAO);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glBindVertexArray(0);
}

```

11.5.19 hybridshader.hpp

```

*****
* HybridShader
*
* Renders the scene in 3 passes:
* - Generate GBuffer with OpenGL
* - Generate final Scene texture with OpenCL using
*   GBuffer from previous pass and geometry (BVH) and lighting information
* - Blit the result of the second pass to the render buffer (openGL)
*
* 2017 - Liberto Camús
*****
#ifndef HYBRIDSHADER_H
#define HYBRIDSHADER_H

#include "clkernelmanager.hpp"
#include "renderengine.hpp"
#include "shaderprogram.hpp" // for ShaderProgram
#include <GL/glew.h> // for GLuint
#include <glm/glm.hpp>
#include <vector> // for vector
class Camera;
class Model;
class World;

class HybridShader : public RenderEngine {
private:

```

```

CLKernelManager *opencl;

// GBUFFER FRAMEBUFFER
GLuint gBuffer; // Framebuffer
GLuint gNormal, gPosition, gAlbedoSpec; // Color attachments
GLuint rboDepth;

// OPENCL_RENDER FRAMEBUFFER
GLuint gSceneBuffer; // Framebuffer
GLuint gSceneTexture; // Color attachment

// OPENGL-OPENCL Shared Textures
enum CL_SHARED_OBJECTS : unsigned int {
    GPOSITION = 0,
    GALBEDOSPEC,
    GNORMAL,
    GSCEENE,
    CL_SHARED_OBJECTS_COUNT
};
cl_mem cl_shared_objects[CL_SHARED_OBJECTS::CL_SHARED_OBJECTS_COUNT];

cl_mem cl_nodesvh, cl_primitives;
// OPENCL kernel argument: Point Lights. The might change
// from frame to frame, so we update it.
cl_mem cl_point_lights;

ShaderProgram gBufferShader;
World *world;

void init_pass1_gBuffer();
void init_pass2_lighting();
void init_pass3_blt();
void update_kernel_args();
void pass1_gBuffer();
void pass2_lighting();
void pass3_blt();

glm::vec3 lastCameraPosition{0, 0, 0};

const std::string GBUFFER_POSITION_TEXTURE = "gPosition";
const std::string GBUFFER_ALBEDO_SPEC_TEXTURE = "gAlbedoSpec";
const std::string GBUFFER_NORMAL_TEXTURE = "gNormal";
const std::string GBUFFER_SCENE_TEXTURE = "gScene";

const std::string UNIFORM_MODEL_MATRIX = "model";
const std::string UNIFORM_VIEW_MATRIX = "view";
const std::string UNIFORM_PROJECTION_MATRIX = "projection";

public:
    void render();
    HybridShader(World *w, CLKernelManager *cl);
    ~HybridShader();
};

#endif // HYBRIDSHADER_H

```

11.5.20 hybridshader.cpp

```

*****
* HybridShader
*
* Renders the scene in 3 passes:
* - Generate GBuffer with OpenGL
* - Generate final Scene texture with OpenCL using

```

```

*      GBuffer from previous pass and geometry (BVH) and lighting information
* - Blit the result of the second pass to the render buffer (openGL)
*
* 2017 - Liberto Camús
* ****
#include "hybridshader.hpp"
#include "camera.hpp"
#include "config.hpp"
#include "model.hpp"
#include "shaderprogram.hpp"
#include "window.hpp"
#include "world.hpp"
#include <CL/cl_gl.h>
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <iostream>
#include <string>

HybridShader::HybridShader(World *w, CLKernelManager *cl)
    : opencl{cl}, gBufferShader{Config::gbuffer_shader_vert.c_str(),
                                Config::gbuffer_shader_frag.c_str()},
      world{w} {

    // Initialize uniforms for the newly created Shaders
    world->initModelsUniforms(gBufferShader);

    // Initialize GBuffer and Scene textures
    init_pass1_gBuffer(); // GBuffer
    init_pass3.blit(); // Scene

    // We need GBuffer and Scene texture before initializing OpenCL
    init_pass2_lighting();
}

void HybridShader::render() {
    pass1_gBuffer();
    update_kernel_args();
    pass2_lighting();
    pass3.blit();
}

void HybridShader::init_pass1_gBuffer() {
    gBufferShader.use();
    glGenFramebuffers(1, &gBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);

    // - Position color buffer
    glGenTextures(1, &gPosition);
    glBindTexture(GL_TEXTURE_2D, gPosition);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_FLOAT, nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                          gPosition, 0);

    // - Normal color buffer
    glGenTextures(1, &gNormal);
    glBindTexture(GL_TEXTURE_2D, gNormal);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8,
                nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
                          gNormal, 0);

    // - Color + Specular color buffer
    glGenTextures(1, &gAlbedoSpec);
    glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,

```

```

        Config::window_height, 0, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8,
        nullptr);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D,
                      gAlbedoSpec, 0);

// - Tell OpenGL which color attachments we'll use (of this framebuffer) for
// rendering
GLuint attachments[3] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
                          GL_COLOR_ATTACHMENT2};
glDrawBuffers(3, attachments);
glGenRenderbuffers(1, &rboDepth);
glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                      Config::window_width, Config::window_height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, rboDepth);

// - Finally check if framebuffer is complete
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
    std::cout << "Framebuffer not complete!" << std::endl;
}
 glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

// Load OpenCL program and compile it, initialize CL memory objects,
// kernel arguments...
void HybridShader::init_pass2_lighting() {
    // Make accessible to OpenCL textures created with OpenGL
    cl_shared_objects[CL_SHARED_OBJECTS::GPOSITION] = opencl->createFromGLTexture(
        gPosition, CL_MEM_READ_ONLY, GBUFFER_POSITION_TEXTURE);
    cl_shared_objects[CL_SHARED_OBJECTS::GALBEDOSPEC] =
        opencl->createFromGLTexture(gAlbedoSpec, CL_MEM_READ_ONLY,
                                    GBUFFER_ALBEDO_SPEC_TEXTURE);
    cl_shared_objects[CL_SHARED_OBJECTS::GNORMAL] = opencl->createFromGLTexture(
        gNormal, CL_MEM_READ_ONLY, GBUFFER_NORMAL_TEXTURE);
    cl_shared_objects[CL_SHARED_OBJECTS::GSCENE] = opencl->createFromGLTexture(
        gSceneTexture, CL_MEM_WRITE_ONLY, GBUFFER_SCENE_TEXTURE);

    // Create point lights memory structure
    auto lPos = world->getPointLights();
    cl_point_lights = opencl->createBufferReadOnly(
        sizeof(PointLight) * lPos.size(), lPos.data());
    cl_int nr_point_lights = static_cast<cl_int>(world->getPointLightsNr());

    // Create geometry memory structures (BVH Nodes + Triangles)
    cl_nodesbvh = opencl->createBufferReadOnly(
        world->bvh.totalNodes * sizeof(BVHLinearNode),
        (void *)&world->bvh.nodes_array[0]);
    cl_primitives = opencl->createBufferReadOnly(
        world->bvh.triangles.size() * sizeof(Triangle),
        (void *)&world->bvh.triangles.data());

    // Set kernel arguments
    opencl->setKernelArg(0, sizeof(cl_mem),
                          &cl_shared_objects[CL_SHARED_OBJECTS::GALBEDOSPEC]);
    opencl->setKernelArg(1, sizeof(cl_mem),
                          &cl_shared_objects[CL_SHARED_OBJECTS::GPOSITION]);
    opencl->setKernelArg(2, sizeof(cl_mem),
                          &cl_shared_objects[CL_SHARED_OBJECTS::GNORMAL]);
    opencl->setKernelArg(3, sizeof(cl_mem),
                          &cl_shared_objects[CL_SHARED_OBJECTS::GSCENE]);
    opencl->setKernelArg(4, sizeof(world->scene_attribs),
                          (void *)&world->scene_attribs);
    opencl->setKernelArg(6, sizeof(cl_mem), &cl_point_lights);
    opencl->setKernelArg(7, sizeof(cl_int), &nr_point_lights);
    opencl->setKernelArg(8, sizeof(cl_mem), &cl_primitives);
    opencl->setKernelArg(9, sizeof(cl_mem), &cl_nodesbvh);
}

// Initialize the framebuffer which will show the scene

```

```

void HybridShader::init_pass3.blit() {
    glGenFramebuffers(1, &gSceneBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, gSceneBuffer);

    // - Scene will be rendered here
    glGenTextures(1, &gSceneTexture);
    glBindTexture(GL_TEXTURE_2D, gSceneTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
                 Config::window_height, 0, GL_RGBA, GL_UNSIGNED_BYTE, nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                          gSceneTexture, 0);

    GLuint attachments[1] = {GL_COLOR_ATTACHMENT0};
    glDrawBuffers(1, attachments);

    // - Finally check if framebuffer is complete
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
        std::cout << "Framebuffer not complete!" << std::endl;
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

void HybridShader::update_kernel_args() {
    // Update scene attributes
    opencl->setKernelArg(4, sizeof(world->scene_attribs),
                           (void *)&world->scene_attribs);

    // Update camera position
    glm::vec3 pos = world->getCamera()->getPosition();
    if (lastCameraPosition != pos) {
        cl_float3 p = cl_float3{pos.x, pos.y, pos.z};
        lastCameraPosition = pos;
        opencl->setKernelArg(5, sizeof(cl_float3), &p);
    }
}

// Update the GBuffer
void HybridShader::pass1_gBuffer() {
    glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
    glClearColor(0.00f, 0.00f, 0.00f, 0.00f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    gBufferShader.use();

    const Camera *camera = world->getCamera();
    const std::vector<Model> *models = world->getModels();

    auto vMat = camera->getViewMatrix();
    auto loc = gBufferShader.getUniformLocation(UNIFORM_VIEW_MATRIX);
    glUniformMatrix4fv(loc, 1, GL_FALSE, glm::value_ptr(vMat));

    auto projMatrix = camera->getProjectionMatrix();
    auto projLoc = gBufferShader.getUniformLocation(UNIFORM_PROJECTION_MATRIX);
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projMatrix));

    auto uniform_model_matrix =
        gBufferShader.getUniformLocation(UNIFORM_MODEL_MATRIX);
    for (auto &m : *models) {
        glUniformMatrix4fv(uniform_model_matrix, 1, GL_FALSE,
                           glm::value_ptr(m.getModelMatrix()));
        m.draw(gBufferShader);
    }
}

// Generate final image from GBuffer + Geometry & Light information
// Update variable kernel arguments
void HybridShader::pass2_lighting() {
    // Sync with OpenGL
}

```

```

glFinish();
opencl->enqueueAcquireGLObjects(CL_SHARED_OBJECTS::CL_SHARED_OBJECTS_COUNT,
                                  cl_shared_objects);

// Execute kernel
cl_event kernel_event;
opencl->executeKernel(&kernel_event, 1);
opencl->enqueueReleaseGLObjects(CL_SHARED_OBJECTS::CL_SHARED_OBJECTS_COUNT,
                                 cl_shared_objects);

// Sync with OpenGL
clWaitForEvents(1, &kernel_event);
clReleaseEvent(kernel_event);
}

// Copy the final image to the framebuffer
void HybridShader::pass3_blt() {
    glBindFramebuffer(GL_READ_FRAMEBUFFER, gSceneBuffer);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
    glBlitFramebuffer(0, 0, Config::window_width, Config::window_height, 0, 0,
                      Config::window_width, Config::window_height,
                      GL_COLOR_BUFFER_BIT, GL_LINEAR);
}

HybridShader::~HybridShader() {
    clReleaseMemObject(cl_shared_objects[CL_SHARED_OBJECTS::GALBEDOSPEC]);
    clReleaseMemObject(cl_shared_objects[CL_SHARED_OBJECTS::GPOSITION]);
    clReleaseMemObject(cl_shared_objects[CL_SHARED_OBJECTS::GNORMAL]);
    clReleaseMemObject(cl_shared_objects[CL_SHARED_OBJECTS::GSCENE]);
    clReleaseMemObject(cl_primitives);
    clReleaseMemObject(cl_nodesbvh);
    clReleaseMemObject(cl_point_lights);

    glDeleteTextures(1, &gPosition);
    glDeleteTextures(1, &gNormal);
    glDeleteTextures(1, &gAlbedoSpec);
    glDeleteTextures(1, &gSceneTexture);
    glDeleteRenderbuffers(1, &rboDepth);
    glDeleteFramebuffers(1, &gSceneBuffer);
    glDeleteFramebuffers(1, &gBuffer);
}
}

```

11.5.21 hybridshadercpu.hpp

```

/****************************************************************************
 * HybridShaderCPU
 *
 * Renders the scene in 3 passes:
 * - Generate GBuffer with OpenGL
 * - Generate final Scene texture with OpenCL using
 *   GBuffer from previous pass and geometry (BVH) and lighting information
 * - Blit the result of the second pass to the render buffer (OpenGL)
 *
 * 2017 - Liberto Camús
 * *****/
#ifndef HYBRIDSHADERCPU_H
#define HYBRIDSHADERCPU_H

#include "renderengine.hpp"
#include "shaderprogram.hpp" // for ShaderProgram
#include <GL/glew.h>           // for GLuint
#include <glm/glm.hpp>
#include <memory>
#include <vector> // for vector
#include <vector>
class Camera;
class Model;
class World;

```

```

class HybridShaderCPU : public RenderEngine {
private:
    class _BVHNode {
    public:
        glm::vec3 bounds_pMin;
        glm::vec3 bounds_pMax;
        union {
            unsigned int primitivesOffset; // leaf
            unsigned int secondChildOffset; // interior
        } uf;
        unsigned char nPrimitives; // 0 -> Interior
        unsigned char axis; // interior node: xyz
    };
};

class _Triangle {
public:
    glm::vec3 p0, p1, p2;
};

class _Ray {
public:
    glm::vec3 o; // origin
    glm::vec3 d; // direction
    float mint;
    float maxt;
};
};

static constexpr float EPSILON = 0.001f;
static constexpr float ATTENUATION_SENSIBILITY = 0.02f;

std::vector<_BVHNode> bvhnodes;
std::vector<_Triangle> triangles;

bool intersects(const _Ray *ray);
bool test_ray_bbox(const _Ray *ray, const _BVHNode *node,
                   const glm::vec3 *invDir);
bool test_ray_triangle(const _Triangle *tri, const _Ray *ray);

// GBUFFER FRAMEBUFFER
GLuint gBuffer; // Framebuffer
GLuint gNormal, gPosition, gAlbedoSpec; // Color attachments
GLuint rboDepth;

// OPENCL RENDER FRAMEBUFFER
GLuint gSceneBuffer; // Framebuffer
GLuint gSceneTexture; // Color attachment

// 
std::vector<GLubyte> gNormal_text;
std::vector<GLfloat> gPosition_text;
std::vector<GLubyte> gAlbedoSpec_text;
std::vector<GLubyte> gScene_text;

ShaderProgram gBufferShader;
World *world;

void init_pass1_gBuffer();
void init_pass2_lighting();
void init_pass3_blit();
void pass1_gBuffer();
void pass2_lighting();
void pass3_blit();

const std::string GBUFFER_POSITION_TEXTURE = "gPosition";
const std::string GBUFFER_ALBEDO_SPEC_TEXTURE = "gAlbedoSpec";
const std::string GBUFFER_NORMAL_TEXTURE = "gNormal";
const std::string GBUFFER_SCENE_TEXTURE = "gScene";

```

```

const std::string UNIFORM_MODEL_MATRIX = "model";
const std::string UNIFORM_VIEW_MATRIX = "view";
const std::string UNIFORM_PROJECTION_MATRIX = "projection";

glm::vec3 pointLightsColor(const glm::vec3 &position, const glm::vec3 &normal,
                           const glm::vec3 &albedo, float specular,
                           const glm::vec3 &viewDir);

void init_geometry();
// unsigned int checks=0;
public:
  void render();
  // const ShaderProgram &getModelShader() { return gBufferShader; }

  HybridShaderCPU(World *w);
  ~HybridShaderCPU();
};

#endif // HYBRIDSHADERCPU_H

```

11.5.22 hybridshadercpu.cpp

```

/*********************************************
 * HybridShaderCPU
 *
 * Renders the scene in 3 passes:
 * - Generate GBuffer with OpenGL
 * - Generate final Scene texture with OpenCL using
 *   GBuffer from previous pass and geometry (BVH) and lighting information
 * - Blit the result of the second pass to the render buffer (openGL)
 *
 * 2017 - Liberto Camús
 * *****/
#include "hybridshadercpu.hpp"
#include "camera.hpp"
#include "config.hpp"
#include "model.hpp"
#include "shaderprogram.hpp"
#include "window.hpp"
#include "world.hpp"
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <iostream>
#include <string>
#include <utility>
//#include <omp.h>

HybridShaderCPU::HybridShaderCPU(World *w)
  : gBufferShader{Config::gbuffer_shader_vert.c_str(),
                  Config::gbuffer_shader_frag.c_str()},
    world{w} {

  // Initialize uniforms for the newly created Shaders
  world->initModelsUniforms(gBufferShader);

  // Initialize GBuffer and Scene textures
  init_pass1_gBuffer(); // GBuffer
  init_pass3.blit(); // Scene

  // We need GBuffer and Scene texture before initializing OpenCL
  init_pass2_lighting();

  init_geometry();
}

void HybridShaderCPU::init_geometry() {

```

```

for (unsigned int i = 0; i < world->bvh.totalNodes; i++) {
    BVHLinearNode clnode = world->bvh.nodes_array[i];

    bvhnodes.emplace_back(
        _BVHNode{glm::vec3(clnode.pMin.x, clnode.pMin.y, clnode.pMin.z),
                  glm::vec3(clnode.pMax.x, clnode.pMax.y, clnode.pMax.z),
                  {clnode.primitivesOffset},
                  clnode.nPrimitives,
                  clnode.axis});
}

for (Triangle cltriangle : world->bvh.triangles) {
    triangles.emplace_back({_Triangle{
        glm::vec3(cltriangle.v1.x, cltriangle.v1.y, cltriangle.v1.z),
        glm::vec3(cltriangle.v2.x, cltriangle.v2.y, cltriangle.v2.z),
        glm::vec3(cltriangle.v3.x, cltriangle.v3.y, cltriangle.v3.z)}});
}

void HybridShaderCPU::render() {
    pass1_gBuffer();
    pass2_lighting();
    pass3_blit();
}

void HybridShaderCPU::init_pass1_gBuffer() {
    gBufferShader.use();
    glGenFramebuffers(1, &gBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);

    // - Position color buffer
    glGenTextures(1, &gPosition);
    glBindTexture(GL_TEXTURE_2D, gPosition);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_FLOAT, nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                          gPosition, 0);

    // - Normal color buffer
    glGenTextures(1, &gNormal);
    glBindTexture(GL_TEXTURE_2D, gNormal);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8,
                nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
                          gNormal, 0);

    // - Color + Specular color buffer
    glGenTextures(1, &gAlbedoSpec);
    glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8,
                nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D,
                          gAlbedoSpec, 0);

    // - Tell OpenGL which color attachments we'll use (of this framebuffer) for
    // rendering
    GLuint attachments[3] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
                            GL_COLOR_ATTACHMENT2};
    glDrawBuffers(3, attachments);
    glGenRenderbuffers(1, &rboDepth);
    glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                         Config::window_width, Config::window_height);
}

```

Anexos

```

glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                         GL_RENDERBUFFER, rboDepth);

// - Finally check if framebuffer is complete
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
    std::cout << "Framebuffer not complete!" << std::endl;
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

// Reserve host memory for GBuffer and Scene
void HybridShaderCPU::init_pass2_lighting() {
    gAlbedoSpec_text.reserve(Config::window_height * Config::window_width *
                           sizeof(GLubyte) * 4);
    gNormal_text.reserve(Config::window_height * Config::window_width *
                         sizeof(GLubyte) * 4);
    gPosition_text.reserve(Config::window_height * Config::window_width *
                          sizeof(GLfloat) * 4);
    gScene_text.reserve(Config::window_height * Config::window_width *
                       sizeof(GLubyte) * 4);
}

// Initialize the framebuffer which will show the scene
void HybridShaderCPU::init_pass3_blt() {
    glGenFramebuffers(1, &gSceneBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, gSceneBuffer);

    // - Scene will be rendered here
    glGenTextures(1, &gSceneTexture);
    glBindTexture(GL_TEXTURE_2D, gSceneTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
                Config::window_height, 0, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8,
                nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                          gSceneTexture, 0);

    GLuint attachments[1] = {GL_COLOR_ATTACHMENT0};
    glDrawBuffers(1, attachments);

    // - Finally check if framebuffer is complete
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
        std::cout << "Framebuffer not complete!" << std::endl;
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

// Update the GBuffer
void HybridShaderCPU::pass1_gBuffer() {
    glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
    glClearColor(0.05f, 0.05f, 0.05f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    gBufferShader.use();

    const Camera *camera = world->getCamera();
    const std::vector<Model> *models = world->getModels();

    auto vMat = camera->getViewMatrix();
    auto loc = gBufferShader.getUniformLocation(UNIFORM_VIEW_MATRIX);
    glUniformMatrix4fv(loc, 1, GL_FALSE, glm::value_ptr(vMat));

    auto projMatrix = camera->getProjectionMatrix();
    auto projLoc = gBufferShader.getUniformLocation(UNIFORM_PROJECTION_MATRIX);
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projMatrix));

    for (auto &m : *models) {
        glUniformMatrix4fv(gBufferShader.getUniformLocation(UNIFORM_MODEL_MATRIX),
                           1, GL_FALSE, glm::value_ptr(m.getModelMatrix()));
    }
}

```

```

        m.draw(gBufferShader);
    }

glm::vec3 cltoglm(cl_float3 a) { return glm::vec3(a.x, a.y, a.z); }

// Generate final image from GBUFFER + Geometry & Light information
void HybridShaderCPU::pass2_lighting() {
    // Bring GBUFFER from GPU to HOST
    gBufferShader.use();
    glBindFramebuffer(GL_READ_FRAMEBUFFER, rboDepth);
    glReadBuffer(GL_COLOR_ATTACHMENT0);
    glReadPixels(0, 0, Config::window_width, Config::window_height, GL_RGBA,
                GL_FLOAT, gPosition_text.data());
    glReadBuffer(GL_COLOR_ATTACHMENT1);
    glReadPixels(0, 0, Config::window_width, Config::window_height, GL_RGBA,
                GL_UNSIGNED_BYTE, gNormal_text.data());
    glReadBuffer(GL_COLOR_ATTACHMENT2);
    glReadPixels(0, 0, Config::window_width, Config::window_height, GL_RGBA,
                GL_UNSIGNED_BYTE, gAlbedoSpec_text.data());

    // Loop through rows and columns
    glm::vec3 view_pos = world->getCamera()->getPosition();

    unsigned int yoffset = 0;

    for (unsigned int y = 0; y < Config::window_height; y++) {
#pragma omp parallel for schedule(dynamic)
        for (unsigned int x = 0; x < Config::window_width; x++) {
            unsigned int xoffset = yoffset + x * 4;
            // Get vectors to albedo, normal, color...
            glm::vec3 albedo =
                glm::vec3(gAlbedoSpec_text[xoffset], gAlbedoSpec_text[xoffset + 1],
                          gAlbedoSpec_text[xoffset + 2]) /
                255.0f;
            float specular = (float)(gAlbedoSpec_text[xoffset + 3]) / 255.0f;

            glm::vec3 normal =
                glm::vec3(gNormal_text[xoffset], gNormal_text[xoffset + 1],
                          gNormal_text[xoffset + 2]) /
                255.0f;
            normal = glm::normalize(normal * 2.0f - glm::vec3(1.0f, 1.0f, 1.0f));

            glm::vec3 position =
                glm::vec3(gPosition_text[xoffset], gPosition_text[xoffset + 1],
                          gPosition_text[xoffset + 2]);
            glm::vec3 ambient = albedo * world->scene_attribs.ambient;
            glm::vec3 viewDir = glm::normalize(view_pos - position);

            // Ambient light
            glm::vec3 out = ambient;

            // Point Lights
            out += pointLightsColor(position, normal, albedo, specular, viewDir);

            // Output
            gScene_text[xoffset] = (GLubyte)(out.x * 255);
            gScene_text[xoffset + 1] = (GLubyte)(out.y * 255);
            gScene_text[xoffset + 2] = (GLubyte)(out.z * 255);
            gScene_text[xoffset + 3] = 0;
        }
        yoffset += Config::window_width * 4;
    }

    // std::cout << "BVH bounding box checks this frame:" << checks << "\n";
    // checks=0;
    // Update GPU Scene texture
}

```

Anexos

```

glBindTexture(GL_TEXTURE_2D, gSceneTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Config::window_width,
            Config::window_height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
            gScene_text.data());
}

glm::vec3 HybridShaderCPU::pointLightsColor(const glm::vec3 &position,
                                              const glm::vec3 &normal,
                                              const glm::vec3 &albedo,
                                              float specular,
                                              const glm::vec3 &viewDir) {
    glm::vec3 out{0.0f, 0.0f, 0.0f};

    // Compute each point light contribution
    for (const PointLight &l : world->getPointLights()) {

        glm::vec3 lpos = cltologlm(l.position);
        glm::vec3 lcol = cltologlm(l.color);
        glm::vec3 lightDir = glm::normalize(lpos - position);
        float dist = glm::distance(lpos, position);
        // Attenuation
        float attenuation = static_cast<float>(
            1.0 / (1.0 + l.linear * dist + l.quadratic * dist * dist));

        if (world->scene_attribs.shadowsEnabled &&
            attenuation > ATTENUATION_SENSIBILITY) {
            _Ray r = _Ray{position, lightDir, EPSILON, dist};
            if (intersects(&r))
                continue;
        }

        // Diffuse
        glm::vec3 diffuse =
            albedo * glm::max(glm::dot(normal, lightDir), 0.0f) * lcol;

        // Specular
        glm::vec3 l_spec;
        if (specular > 0) {
            glm::vec3 halfwayDir = glm::normalize(lightDir + viewDir);
            float spec =
                glm::pow(glm::max(glm::dot(normal, halfwayDir), 0.0f), 16.0f);
            l_spec = lcol * spec * specular;
        }

        out += (diffuse + l_spec) * attenuation;
    }
    return out;
}

// Copy the final image to the framebuffer
void HybridShaderCPU::pass3_blt() {
    glBindFramebuffer(GL_READ_FRAMEBUFFER, gSceneBuffer);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
    glBlitFramebuffer(0, 0, Config::window_width, Config::window_height, 0, 0,
                      Config::window_width, Config::window_height,
                      GL_COLOR_BUFFER_BIT, GL_NEAREST);
}

HybridShaderCPU::~HybridShaderCPU() {
    glDeleteTextures(1, &gPosition);
    glDeleteTextures(1, &gNormal);
    glDeleteTextures(1, &gAlbedoSpec);
    glDeleteTextures(1, &gSceneTexture);
    glDeleteRenderbuffers(1, &rboDepth);
    glDeleteFramebuffers(1, &gSceneBuffer);
    glDeleteFramebuffers(1, &gBuffer);
}

// Checks ray-triangle intersection

```

```

// Based on algorithm in "Physically based rendering, 2nd edition"
bool HybridShaderCPU::test_ray_triangle(const _Triangle *tri, const _Ray *ray) {
    glm::vec3 e1 = tri->p1 - tri->p0;
    glm::vec3 e2 = tri->p2 - tri->p0;

    glm::vec3 s1 = glm::cross(ray->d, e2);
    float a = glm::dot(s1, e1);
    if (a > -EPSILON && a < EPSILON)
        return false;

    float invDivisor = 1.0f / a;

    glm::vec3 d = ray->o - tri->p0;
    float b1 = invDivisor * glm::dot(d, s1);
    if (b1 < 0.0f || b1 > 1.0f)
        return false;

    glm::vec3 s2 = glm::cross(d, e1);
    float b2 = invDivisor * glm::dot(ray->d, s2);
    if (b2 < 0.0f || b1 + b2 > 1.0f)
        return false;

    float t = invDivisor * glm::dot(e2, s2);
    if (t < ray->mint || t > ray->maxt)
        return false;
    else
        return true;
}

// Checks ray-bounding box intersection
// Based on algorithm in "Physically based rendering, 2nd edition"
bool HybridShaderCPU::test_ray_bbox(const _Ray *ray, const _BVHNode *node,
                                     const glm::vec3 *invDir) {

    // checks++;
    float tmin = ray->mint;
    float tmax = ray->maxt;

    glm::vec3 tNear = (node->bounds_pMin - ray->o) * (*invDir);
    glm::vec3 tFar = (node->bounds_pMax - ray->o) * (*invDir);

    if (tNear.x > tFar.x)
        std::swap(tNear.x, tFar.x);
    tmin = tNear.x > tmin ? tNear.x : tmin;
    tmax = tFar.x < tmax ? tFar.x : tmax;
    if (tmin > tmax)
        return false;

    if (tNear.y > tFar.y)
        std::swap(tNear.y, tFar.y);
    tmin = tNear.y > tmin ? tNear.y : tmin;
    tmax = tFar.y < tmax ? tFar.y : tmax;
    if (tmin > tmax)
        return false;

    if (tNear.z > tFar.z)
        std::swap(tNear.z, tFar.z);
    tmin = tNear.z > tmin ? tNear.z : tmin;
    tmax = tFar.z < tmax ? tFar.z : tmax;
    if (tmin > tmax)
        return false;

    return true;
}

// Checks if a ray intersects with a collection of triangles, using
// a BVH tree to accelerate the process.

```

```
// Based on algorithm in "Physically based rendering, 2nd edition"
bool HybridShaderCPU::intersects(const _Ray *ray) {

    int todoOffset = 0, nodeNum = 0;
    int todo[64];

    glm::vec3 invDir =
        glm::vec3(1.0f / ray->d.x, 1.0f / ray->d.y, 1.0f / ray->d.z);
    int dirIsNeg[3] = {invDir.x < 0, invDir.y < 0, invDir.z < 0};

    while (true) {
        const _BVHNode *node = &bvhnodes[nodeNum];
        // Check ray against BVH node
        if (test_ray_bbox(ray, node, &invDir)) {
            if (node->nPrimitives > 0) {
                // Intersect ray with primitives in leaf BVH node
                for (int i = 0; i < node->nPrimitives; i++)
                    if (test_ray_triangle(&triangles[node->uf.primitivesOffset + i], ray))
                        return true;
                if (todoOffset == 0)
                    break;
                nodeNum = todo[--todoOffset];
            } else {
                // Put far BVH node on todo stack, advance to near node
                if (dirIsNeg[node->axis]) {
                    todo[todoOffset++] = nodeNum + 1;
                    nodeNum = node->uf.secondChildOffset;
                } else {
                    todo[todoOffset++] = node->uf.secondChildOffset;
                    nodeNum = nodeNum + 1;
                }
            }
        } else {
            if (todoOffset == 0)
                break;
            nodeNum = todo[--todoOffset];
        }
    }
    return false;
}
```

11.5.23 input.hpp

```
*****
* Input
*
* Manages input from keyboard and mouse. Calls Camera
* methods to update its position and orientation.
* Uses "glfwGetWindowUserPointer" to store a pointer
* to itself in the Window object and thus be able
* to access its methods and properties from the callback code.
*
* 2017 - Liberto Camús
*****
#ifndef INPUT_H
#define INPUT_H

#include <iostream>
#include <memory>
#define GLEW_STATIC
#include <GL/glew.h>
#include <GLFW/glfw3.h>

#include "camera.hpp"

class Input {
public:
    Input(GLFWwindow *w);
    Input(GLFWwindow *w, Camera *c);
```

```

~Input() {}
void setCamera(Camera *c) { camera = c; }
void move(GLfloat delta_time);

private:
    Camera *camera;
    bool key_pressed[1024];
    static constexpr double NOT_INITIALIZED = -1;
    double lastX = NOT_INITIALIZED, lastY = NOT_INITIALIZED;

    void init(GLFWwindow *window);
    void set_key_callback(GLFWwindow *window);
    void set_scroll_callback(GLFWwindow *window);
    void set_mouse_callback(GLFWwindow *window);
};

#endif // INPUT_H

```

11.5.24 input.cpp

```

*****
* Input
*
* Manages input from keyboard and mouse. Calls Camera
* methods to update its position and orientation.
* Uses "glfwGetWindowUserPointer" to store a pointer
* to itself in the Window object and thus be able
* to access its methods and properties from the callback code.
*
* 2017 - Liberto Camús
*****
#include "input.hpp"
#include "camera.hpp"

#include <cstring>

Input::Input(GLFWwindow *window) { init(window); }

Input::Input(GLFWwindow *window, Camera *c) : camera{c} { init(window); }

/**
 * @brief Input::init
 * Object initialization
 * @param window window which handles the event
 */
void Input::init(GLFWwindow *window) {
    /// Save pointer to «this» into «GL window»
    glfwSetWindowUserPointer(window, reinterpret_cast<void *>(this));
    /// Initialize array to zero
    std::memset(key_pressed, false, sizeof key_pressed);
    /// Set callbacks
    set_key_callback(window);
    set_mouse_callback(window);
    set_scroll_callback(window);
}

/**
 * @brief Input::move
 * Check cursor keys and move camera accordingly
 * @param delta_time Time in ms since last movement
 */
void Input::move(GLfloat delta_time) {
    if (key_pressed[GLFW_KEY_UP])
        camera->move(Camera::FORWARD, delta_time);
    if (key_pressed[GLFW_KEY_DOWN])
        camera->move(Camera::BACKWARD, delta_time);
    if (key_pressed[GLFW_KEY_LEFT])
        camera->move(Camera::LEFT, delta_time);
}

```

```

    if (key_pressed[GLFW_KEY_RIGHT])
        camera->move(Camera::RIGHT, delta_time);
}

//  

/**  

 * @brief Input::set_key_callback  

 * Is called whenever a key is pressed/released via GLFW  

 * Handles keys that have to react only once when pressed  

 * @param window Window which handles the event  

 */
void Input::set_key_callback(GLFWwindow *window) {
    glfwSetKeyCallback(window, [](GLFWwindow *window, int key, int /*scancode*/,
                                int action, int /*mode*/) {
        auto thiz = reinterpret_cast<Input *>(glfwGetWindowUserPointer(window));
        if (action == GLFW_PRESS) {
            switch (key) {
                case GLFW_KEY_ESCAPE:
                case GLFW_KEY_Q:
                    glfwSetWindowShouldClose(window, GL_TRUE);
                    break;
                case GLFW_KEY_S:
                    Config::option_shadows_enabled = !Config::option_shadows_enabled;
                    std::cout << "Shadows "
                           << ((Config::option_shadows_enabled) ? "ON" : "OFF") << "\n";
                    break;
                case GLFW_KEY_C:
                    if (Config::rendering_method != Config::HYBRID_CPU) {
                        Config::option_rendering_method_change_requested = true;
                        Config::rendering_method = Config::HYBRID_CPU;
                    }
                    break;
                case GLFW_KEY_H:
                    if (Config::rendering_method != Config::HYBRID) {
                        if (Config::option_opencl_available) {
                            Config::option_rendering_method_change_requested = true;
                            Config::rendering_method = Config::HYBRID;
                        } else
                            std::cout << "OpenCL Shader not available\n";
                    }
                    break;
                case GLFW_KEY_D:
                    if (Config::rendering_method != Config::DEFERRED) {
                        Config::option_rendering_method_change_requested = true;
                        Config::rendering_method = Config::DEFERRED;
                    }
                    break;
                case GLFW_KEY_P:
                    Config::option_statistics_requested = true;
                    break;
                case GLFW_KEY_R:
                    Config::option_reset_statistics_requested = true;
                    break;
                case GLFW_KEY_L:
                    glm::vec3 pos = thiz->camera->getPosition();
                    std::cout << "Camera location: " << pos.x << ", " << pos.y << ", "
                           << pos.z << "\n";
                    break;
            }
        }
        // Update array of key states
        if (key >= 0 && key < 1024) {
            if (action == GLFW_PRESS)
                thiz->key_pressed[key] = true;
            else if (action == GLFW_RELEASE)
                thiz->key_pressed[key] = false;
        }
    });
}

/**  

 * @brief Input::set_mouse_callback  

 * Sets mouse movement callback. Rotates the camera around  

 * @param window Window which handles the event

```

```

/*
void Input::set_mouse_callback(GLFWwindow *window) {
    glfwSetCursorPosCallback(
        window, [](GLFWwindow *window, double xpos, double ypos) {

            auto thiz = reinterpret_cast<Input *>(glfwGetWindowUserPointer(window));
            /// Initialize mouse position in first call
            if (thiz->lastX == NOT_INITIALIZED) {
                thiz->lastX = xpos;
                thiz->lastY = ypos;
            }

            /// Calculate x and y offset
            double xoffset = xpos - thiz->lastX;
            /// Reversed since y-coordinates go from bottom to left
            double yoffset = thiz->lastY - ypos;

            /// Update las position
            thiz->lastX = xpos;
            thiz->lastY = ypos;

            /// Rotate the camera
            thiz->camera->rotate(
                static_cast<float>(xoffset) * Config::movement_sensitivity,
                static_cast<float>(yoffset) * Config::movement_sensitivity);
        });
}

/**
 * @brief Input::set_scroll_callback
 * Sets the callback used when mouse scroll wheel event is received
 * Updates camera's vertical field of view angle (fovy) and behaves
 * as a zoom
 * @param window Window which handles the event
 */
void Input::set_scroll_callback(GLFWwindow *window) {
    glfwSetScrollCallback(
        window, [](GLFWwindow *window, double /*xoffset*/, double yoffset) {
            auto thiz = reinterpret_cast<Input *>(glfwGetWindowUserPointer(window));
            thiz->camera->change_fovy(static_cast<float>(yoffset) *
                                         Config::movement_fovy_step);
        });
}

```

11.5.25 main.cpp

```

*****
* Main
*
* Main method and loop
*
* 2017 - Liberto Camús
* ****
#include "configloader.hpp"
#include "renderenginecreator.hpp"
#include "system.hpp"
#include "world.hpp"

constexpr const char *USAGE_STR = "\
Usage: ashpool <options> configuration_file
Example: ashpool --cpu --noshadows sponza.ash
Options:
    --cpu: Use Hybrid Shader on CPU
    --hybrid: Use Hybrid Shader with OpenCL
    --deferred: Use Deferred Shader
    --noshadows: Do not calculate shadows with Hybrid shaders
    --nocapturemouse: Do not capture mouse pointer.
";

```

```

void parseCLI(int argc, char *argv[]) {
    for (int i = 1; i < (argc - 1); i++) {
        std::string option{argv[i]};
        if (option.compare("--cpu") == 0)
            Config::rendering_method = Config::HYBRID_CPU;
        else if (option.compare("--deferred") == 0)
            Config::rendering_method = Config::DEFERRED;
        else if (option.compare("--hybrid") == 0)
            Config::rendering_method = Config::HYBRID;
        else if (option.compare("--noshadows") == 0)
            Config::option_shadows_enabled = false;
        else if (option.compare("--nocapturemouse") == 0)
            Config::option_no_capture_mouse = true;
    }
}

int main(int argc, char *argv[]) {
    // Check number of command line interface arguments
    if (argc < 2) {
        std::cout << USAGE_STR;
        exit(-1);
    }

    // Load configuration and scene description
    ConfigLoader loadconfig(argv[argc - 1]);
    // Check command line interface arguments
    parseCLI(argc, argv);

    // Initialize engine objects
    System system;
    system.initOpenCL();
    if (Config::rendering_method == Config::HYBRID &&
        !Config::option_opencl_available) {
        std::cout << "Required OpenCL capabilities not available. \n";
        exit(1);
    }
    World world;
    world.init();
    std::unique_ptr<RenderEngine> renderer =
        RenderEngineCreator::create(world, system);
    system.setCamera(world.getCamera());

    // Main loop
    while (!system.exitRequested()) {
        // Update timers, keyboard events...
        system.update();
        // Update geometry and scene options
        world.update();
        // Render the scene
        renderer->render();
        // Change Rendering Method if necessary
        if (Config::option_rendering_method_change_requested) {
            renderer.reset(nullptr);
            renderer = RenderEngineCreator::create(world, system);
            system.resetStatistics();
        }
    }
    system.clmanager.reset(nullptr);
    return 0;
}

```

11.5.26 material.hpp

```

*****
* Material
*
* Stores material properties. Diffuse, ambient,
* specular...
*
* 2017 - Liberto Camús
*****

```

```
#ifndef MATERIAL_H
#define MATERIAL_H

#include <GL/glew.h>
#include <glm/glm.hpp>

class Material {
public:
    glm::vec4 diffuse{0.8f, 0.8f, 0.8f, 1.0f};
    glm::vec4 ambient{0.2f, 0.2f, 0.2f, 1.0f};
    glm::vec4 specular{0.0f, 0.0f, 0.0f, 1.0f};
    glm::vec4 emissive{0.0f, 0.0f, 0.0f, 1.0f};

    GLuint shininess_uniform{0};
    GLuint diffuse_uniform{0};
    GLuint ambient_uniform{0};
    GLuint specular_uniform{0};
    GLuint tex_count_uniform{0};

    float shininess{0.0f};
    int texCount{0};
};

#endif // MATERIAL_H
```

11.5.27 mesh.hpp

```
*****
* Mesh
*
* Set of triangles, material properties, textures
* and its methods for drawing and updating OpenGL
* shaders.
* A Model is made out of Meshes.
*
* 2017 - Liberto Camús
*****
#ifndef MESH_H
#define MESH_H

#include "material.hpp"
#include "texture.hpp"
#include "vertex.hpp"
#include <GL/glew.h> // for GLuint
#include <algorithm> // for move
#include <vector> // for vector

#include <iostream>

class ShaderProgram;

class Mesh {
public:
    Mesh() = delete;
    Mesh(Mesh &other)
        : vertices(std::move(other.vertices)), indices(std::move(other.indices)),
          specular_textures(std::move(other.specular_textures)),
          diffuse_textures(std::move(other.diffuse_textures)),
          normal_textures(std::move(other.normal_textures)),
          material(std::move(other.material)), VAO(other.VAO), VBO(other.VBO),
          EBO(other.EBO), indicesSize(other.indicesSize) {
        other.VBO = 0;
        other.EBO = 0;
        other.VAO = 0;
        other.indicesSize = 0;
    }
}
```

```

Mesh(std::vector<Vertex> &&vertices, std::vector<GLuint> &&indices,
     std::vector<Texture> &&specular_textures,
     std::vector<Texture> &&diffuse_textures,
     std::vector<Texture> &&Normal_Textures, Material &&material);
~Mesh();

Mesh &operator=(const Mesh &other) = delete;
Mesh &operator=(const Mesh &&other) = delete;
Mesh(const Mesh &other) = delete;

void refreshUniforms(const ShaderProgram &shader);
void draw(const ShaderProgram &shader) const;

const std::vector<Vertex> &getVertices() const { return vertices; }
const std::vector<GLuint> &getIndices() const { return indices; }

private:
    std::vector<Vertex> vertices;
    std::vector<GLuint> indices;
    std::vector<Texture> specular_textures;
    std::vector<Texture> diffuse_textures;
    std::vector<Texture> normal_textures;
    Material material;
    GLuint VAO{0}, VBO{0}, EBO{0};
    GLsizei indicesSize{0};

    void setupMesh();
};

#endif // MESH_H

```

11.5.28 mesh.cpp

```

/****************************************************************************
 * Mesh
 *
 * Set of triangles, material properties, textures
 * and its methods for drawing and updating OpenGL
 * shaders.
 * A Model is made out of Meshes.
 *
 * 2017 - Liberto Camús
 * ****
#include "mesh.hpp"
#include "config.hpp"
#include "shaderprogram.hpp"    // for Shader
#include "texturemanager.hpp"   // for TextureManager
#include <GL/glew.h>           // for GLuint, glBindVertexArray, etc
#include <assimp/color4.h>       // for aiColor4D
#include <assimp/material.h>     // for aiGetMaterialColor, etc
#include <assimp/material.inl>    // for aiMaterial::GetTextureCount
#include <assimp/mesh.h>         // for aiMesh, aiFace
#include <assimp/scene.h>        // for aiScene
#include <assimp/vector3.h>       // for aiVector3t, aiVector3D
#include <ext/alloc_traits.h>      // for __alloc_traits<>::value_type
#include <iostream>
#include <iterator> // for back_insert_iterator, etc
#include <sstream> // for stringstream, basic_ostream
#include <stddef.h> // for offsetof

Mesh::Mesh(std::vector<Vertex> &&Vertices, std::vector<GLuint> &&Indices,
           std::vector<Texture> &&Specular_Textures,
           std::vector<Texture> &&Diffuse_Textures,
           std::vector<Texture> &&Normal_Textures, Material &&mtl)
: vertices{std::move(Vertices)}, indices{std::move(Indices)},
  specular_textures{std::move(Specular_Textures)},
  diffuse_textures{std::move(Diffuse_Textures)},
  normal_textures{std::move(Normal_Textures)}, material{std::move(mtl)} {
    this->setupMesh();
}

```

```

void Mesh::refreshUniforms(const ShaderProgram &shader) {
    static constexpr auto diffuse_name = "texture_diffuse";
    static constexpr auto specular_name = "texture_specular";
    static constexpr auto normal_name = "texture_normal";

    for (GLuint i = 0; i < diffuse_textures.size(); i++)
        diffuse_textures[i].uniformId =
            shader.getUniformLocation(diffuse_name + std::to_string(i));

    for (GLuint i = 0; i < specular_textures.size(); i++)
        specular_textures[i].uniformId =
            shader.getUniformLocation(specular_name + std::to_string(i));

    for (GLuint i = 0; i < normal_textures.size(); i++)
        normal_textures[i].uniformId =
            shader.getUniformLocation(normal_name + std::to_string(i));

    material.shininess_uniform = shader.getUniformLocation("material.shininess");
    material.diffuse_uniform = shader.getUniformLocation("material.diffuse");
    material.ambient_uniform = shader.getUniformLocation("material.ambient");
    material.specular_uniform = shader.getUniformLocation("material.specular");
    material.tex_count_uniform = shader.getUniformLocation("material.tex_count");
}

void Mesh::draw(const ShaderProgram &shader) const {
    shader.use();

    glUniform1i(material.tex_count_uniform, 0);
    // Bind textures
    int i = 0;
    for (auto &tex : diffuse_textures) {
        glActiveTexture(GL_TEXTURE0 + i);
        glUniform1i(tex.uniformId, i);
        glBindTexture(GL_TEXTURE_2D, tex.id);
        glUniform1i(material.tex_count_uniform, 1);
        i = i + 1;
    }
    for (auto &tex : specular_textures) {
        glActiveTexture(GL_TEXTURE0 + i);
        glUniform1i(tex.uniformId, i);
        glBindTexture(GL_TEXTURE_2D, tex.id);
        i = i + 1;
    }

    for (auto &tex : normal_textures) {
        glActiveTexture(GL_TEXTURE0 + i);
        glUniform1i(tex.uniformId, i);
        glBindTexture(GL_TEXTURE_2D, tex.id);
        i = i + 1;
    }

    GLuint normal_mapping = shader.getUniformLocation("options_normal_mapping");
    glUniform1i(normal_mapping, Config::option_normal_mapping_enabled);

    // Bind material
    glUniform1f(material.shininess_uniform, material.shininess);
    glUniform3f(material.diffuse_uniform, material.diffuse.x, material.diffuse.y,
               material.diffuse.z);
    glUniform3f(material.ambient_uniform, material.ambient.x, material.ambient.y,
               material.ambient.z);
    glUniform1f(material.specular_uniform, material.specular.x);
    glUniform1i(material.tex_count_uniform, material.texCount);
    // glUniform1f(shader.getUniformLocation("material_specular"), 0.9f);

    // Draw
    glBindVertexArray(this->VAO);
    glDrawElements(GL_TRIANGLES, this->indicesSize, GL_UNSIGNED_INT, nullptr);
}

```

```

void Mesh::setupMesh() {
    this->indicesSize = static_cast<GLsizei>(indices.size());

    // Create buffers/arrays
    glGenVertexArrays(1, &this->VAO);
    glGenBuffers(1, &this->VBO);
    glGenBuffers(1, &this->EBO);

    glBindVertexArray(this->VAO);

    // Load data into vertex buffers
    auto sizeof_vertex = sizeof(Vertex);
    glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
    glBufferData(GL_ARRAY_BUFFER, this->vertices.size() * sizeof_vertex,
                 &this->vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->indices.size() * sizeof(GLuint),
                 &this->indices[0], GL_STATIC_DRAW);

    // Set the vertex attribute pointers
    // Vertex Positions
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof_vertex,
                          (const GLvoid *)nullptr);
    // Vertex Normals
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof_vertex,
                          (const GLvoid *)offsetof(Vertex, Normal));
    // Vertex Texture Coords
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof_vertex,
                          (const GLvoid *)offsetof(Vertex, TexCoords));

    glBindVertexArray(0);
}

Mesh::~Mesh() {
    if (vertices.empty())
        return;

    if (this->VAO != 0)
        glDeleteVertexArrays(1, &this->VAO);

    if (this->VBO != 0)
        glDeleteBuffers(1, &this->VBO);

    if (this->EBO != 0)
        glDeleteBuffers(1, &this->EBO);
}

```

11.5.29 model.hpp

```

*****
* Model
*
* Set of meshes which constitutes an entity.
* The model contains the "ModelMatrix", to translate,
* spin and scale it.
* Has a method for drawing itself (calling each
* mesh draw methods)
*
* 2017 - Liberto Camús
*****

```

```
#ifndef MODEL_H
#define MODEL_H

#include <CL/cl_gl.h>
#include <glm/glm.hpp>
#include <vector> // for vector

#include "mesh.hpp" // for Mesh
#include "triangle.hpp"
class ShaderProgram;

class Model {
public:
    Model() = delete;
    Model(std::vector<Mesh> &&m) : meshes{std::move(m)} {}
    Model &operator=(const Model &other) = delete;
    Model(const Model &other) = delete;
    Model(Model &&other)
        : meshes{std::move(other.meshes)}, modelMatrix{
            std::move(other.modelMatrix)} {}

    ~Model() = default;

    void draw(const ShaderProgram &shader) const;
    const glm::mat4 &getModelMatrix() const { return modelMatrix; }

    // Updates/sets information about its meshes into the OpenGL shader
    void refreshUniforms(const ShaderProgram &shader);

    // Exports its geometry (the vertices of all of its meshes)
    // in a format to be used by the BVH acceleration structure
    // and exported to OpenCL
    std::vector<Triangle> ExportTriangles();

private:
    std::vector<Mesh> meshes;
    glm::mat4 modelMatrix{};
};

#endif // MODEL_H
```

11.5.30 model.cpp

```
*****
* Model
*
* Set of meshes which constitutes an entity.
* The model contains the "ModelMatrix", to translate,
* spin and scale it.
* Has a method for drawing itself (calling each
* mesh draw methods)
*
* 2017 - Liberto Camús
* ****
#include "model.hpp"
#include "mesh.hpp"
#include "triangle.hpp"
#include "vertex.hpp"

#include "config.hpp"

void Model::draw(const ShaderProgram &shader) const {
    for (const auto &mesh : meshes)
        mesh.draw(shader);
}
```

Anexos

```
// Updates/sets information about its meshes into the OpenGL shader
void Model::refreshUniforms(const ShaderProgram &shader) {
    for (auto &mesh : meshes)
        mesh.refreshUniforms(shader);
}

// Exports its geometry (the vertices of all of its meshes)
// in a format to be used by the BVH acceleration structure
// and exported to OpenCL
std::vector<Triangle> Model::ExportTriangles() {
    std::vector<Triangle> t;
    size_t n_vertices{0}, n_triangles{0};
    for (Mesh &mesh : meshes) {
        std::vector<GLuint> ind = mesh.getIndices();
        std::vector<Vertex> vertices = mesh.getVertices();
        n_vertices += vertices.size();
        n_triangles += ind.size() / 3;
        for (size_t i = 0; i < ind.size() - 3; i += 3) {
            t.emplace_back(Triangle{vertices[ind[i]].Position,
                                   vertices[ind[i + 1]].Position,
                                   vertices[ind[i + 2]].Position});
        }
    }
    std::cout << "Triangles: " << n_triangles << "\n";
    std::cout << "Vertices: " << n_vertices << "\n";
}

return t;
}
```

11.5.31 modelloader.hpp

```
*****
* ModelLoader
*
* Loads a model from a file into a Model object.
* Uses ASSIMP to import different possible formats and it
* loads meshes, textures and material properties
*
* 2017 - Liberto Camús
*****
#ifndef MODELLOADER_H
#define MODELLOADER_H

#include <GL/glew.h>
#include <string>
#include <vector>

#include "texturemanager.hpp"
#include "vertex.hpp"

struct aiScene;
struct aiMesh;
struct aiMaterial;
class Model;
class Mesh;
class Material;

class ModelLoader {

public:
    ModelLoader() = default;
    Model loadModel(const std::string &path);

private:
    std::vector<Mesh> loadMeshes(const aiScene *scene,
```

```

        const std::string &directory);
Material loadMaterial(const aiMaterial *mtl) const;
std::vector<Vertex> loadMeshVertices(const aiMesh *mesh) const;
std::vector<GLuint> loadMeshIndices(const aiMesh *mesh) const;
TextureManager mTextureManager;
};

#endif // MODELLOADER_H

```

11.5.32 modelloader.cpp

```

*****
* ModelLoader
*
* Loads a model from a file into a Model object.
* Uses ASSIMP to import different possible formats and it
* loads meshes, textures and material properties
*
* 2017 - Liberto Camús
*****
#include "modelloader.hpp"
#include "material.hpp"
#include "mesh.hpp"
#include "model.hpp"
#include "shaderprogram.hpp" // for Shader
#include "texture.hpp"
#include "texturemanager.hpp" // for TextureManager
#include "vertex.hpp"
#include <GL/glew.h> // for GLuint, glBindVertexArray, etc
#include <assimp/Importer.hpp> // for Importer
#include <assimp/color4.h> // for aiColor4D
#include <assimp/material.h> // for aiGetMaterialColor, etc
#include <assimp/material.inl> // for aiMaterial::GetTextureCount
#include <assimp/mesh.h> // for aiMesh, aiFace
#include <assimp/postprocess.h>
#include <assimp/scene.h> // for aiScene
#include <assimp/vector3.h> // for aiVector3t, aiVector3D
#include <ext/alloc_traits.h> // for __alloc_traits<>::value_type
#include <iostream>
#include <iterator> // for back_insert_iterator, etc
#include <sstream> // for stringstream, basic_ostream
#include <stddef.h> // for offsetof

Model ModelLoader::loadModel(const std::string &path) {
    // Read file
    std::cout << "Start importing model" << std::endl;
    Assimp::Importer importer;
    auto directory = path.substr(0, path.find_last_of('/'));
    const aiScene *scene = importer.ReadFile(
        path, aiProcess_GenSmoothNormals | aiProcess_Triangulate);

    // Check for errors
    if (!scene || scene->mFlags == AI_SCENE_FLAGS_INCOMPLETE ||
        !scene->mRootNode) {
        std::cout << "Error importing model " << importer.GetErrorString()
            << std::endl;
        exit(2);
    }

    // Return new model
    return Model(loadMeshes(scene, directory));
}

std::vector<Mesh> ModelLoader::loadMeshes(const aiScene *scene,
                                            const std::string &directory) {

    std::vector<Mesh> meshes;
    std::cout << "Reading " << scene->mNumMeshes << " meshes" << std::endl;
    meshes.reserve(scene->mNumMeshes);
}

```

Anexos

```

for (GLuint i = 0; i < scene->mNumMeshes; i++) {
    auto m = scene->mMeshes[i];
    aiMaterial *mat = scene->mMaterials[m->mMaterialIndex];

    meshes.emplace_back(loadMeshVertices(m), loadMeshIndices(m),
                        mTextureManager.loadMaterialTextures(
                            mat, aiTextureType_SPECULAR, directory),
                        mTextureManager.loadMaterialTextures(
                            mat, aiTextureType_DIFFUSE, directory),
                        std::vector<Texture>(), loadMaterial(mat));
}

std::cout << "Model successfully imported." << std::endl;
return meshes;
}

std::vector<Vertex> ModelLoader::loadMeshVertices(const aiMesh *mesh) const {

    std::vector<Vertex> vertices;
    vertices.reserve(mesh->mNumVertices);

    for (GLuint i = 0; i < mesh->mNumVertices; i++) {
        auto vv = &mesh->mVertices[i];
        auto vn = &mesh->mNormals[i];
        // Texture Coordinates
        if (mesh->mTextureCoords[0]) {
            // Load only 1st texture coordinates set. It could have up to 8.
            auto vt = &mesh->mTextureCoords[0][i];

            vertices.emplace_back(glm::vec3(vv->x, vv->y, vv->z),
                                  glm::vec3(vn->x, vn->y, vn->z),
                                  glm::vec2(vt->x, vt->y));
        } else {
            vertices.emplace_back(glm::vec3(vv->x, vv->y, vv->z),
                                  glm::vec3(vn->x, vn->y, vn->z),
                                  glm::vec2(0.0f, 0.0f));
        }
    }
    return vertices;
}

std::vector<GLuint> ModelLoader::loadMeshIndices(const aiMesh *mesh) const {

    std::vector<GLuint> indices;
    indices.reserve(3 * mesh->mNumFaces);

    for (GLuint i = 0; i < mesh->mNumFaces; i++) {
        aiFace *face = &mesh->mFaces[i];
        std::copy_n(&face->mIndices[0], face->mNumIndices, back_inserter(indices));
    }

    return indices;
}

Material ModelLoader::loadMaterial(const aiMaterial *mtl) const {

    Material mat;

    mat.texCount = 0;

    if (mtl->GetTextureCount(aiTextureType_DIFFUSE) > 0)
        mat.texCount = 1;

    aiColor4D color;
    if (AI_SUCCESS == aiGetMaterialColor(mtl, AI_MATKEY_COLOR_DIFFUSE, &color))
        mat.diffuse = glm::vec4(color.r, color.g, color.b, color.a);
}

```

```

if (AI_SUCCESS == aiGetMaterialColor(mtl, AI_MATKEY_COLOR_AMBIENT, &color))
    mat.ambient = glm::vec4(color.r, color.g, color.b, color.a);

if (AI_SUCCESS == aiGetMaterialColor(mtl, AI_MATKEY_COLOR_SPECULAR, &color))
    mat.specular = glm::vec4(color.r, color.g, color.b, color.a);

if (AI_SUCCESS == aiGetMaterialColor(mtl, AI_MATKEY_COLOR_EMISSIVE, &color))
    mat.emissive = glm::vec4(color.r, color.g, color.b, color.a);

return mat;
}

```

11.5.33 pointlight.hpp

```

/*********************************************************************
* PointLight
*
* Contains properties of a Point Light in a format
* suitable to be shared with OpenCL.
* Stores position, color, attenuation values...
*
* 2017 - Liberto Camús
* *****/
#ifndef POINTLIGHT_H
#define POINTLIGHT_H

#include <CL/cl_gl.h>

class PointLight {
public:
    PointLight(float x, float y, float z)
        : position{{x, y, z}}, color{{1.0f, 1.0f, 1.0f}}, intensity{0.8f},
          linear{0.001f}, quadratic{0.00001f} {}
    cl_float3 position;
    cl_float3 color;
    cl_float intensity;
    cl_float linear;
    cl_float quadratic;
};

#endif // POINTLIGHT_H

```

11.5.34 renderengine.hpp

```

/*********************************************************************
* RenderEngine
*
* Abstract class intended to be the interface every
* rendering method should inherit from.
* Currently there are three possibilities: Deferred
* Shader (full OpenGL), Hybrid Shader (OpenGL+OpenCL)
* and CPU Hybrid Shader (OpenGL+OpenMP)
*
* 2017 - Liberto Camús
* *****/
#ifndef RENDERENGINE_H
#define RENDERENGINE_H

class RenderEngine {
public:
    virtual void render() = 0;
    virtual ~RenderEngine() {}
};


```

```
#endif // RENDERENGINE_H
```

11.5.35 renderenginecreator.hpp

```
*****
* RenderEngineCreator
*
* Factory Method to create the a rendering engine
* based on Config::rendering_method enum.
* Currently there are three possibilities: Deferred
* Shader (full OpenGL), Hybrid Shader (OpenGL+OpenCL)
* and CPU Hybrid Shader (OpenGL+OpenMP)
*
* 2017 - Liberto Camús
*****
#ifndef RENDERENGINECREATOR_H
#define RENDERENGINECREATOR_H

#include "config.hpp"
#include "deferredshader.hpp"
#include "hybridshader.hpp"
#include "hybridshadercpu.hpp"
#include "renderengine.hpp"
#include "system.hpp"
#include <iostream>
#include <memory>
class World;

class RenderEngineCreator {
public:
    static std::unique_ptr<RenderEngine> create(World &world, System &system) {
        std::unique_ptr<RenderEngine> ren;
        switch (Config::rendering_method) {
            case Config::HYBRID:
                if (!Config::option_opencl_available) {
                    std::cout << "OpenCL not available\n";
                    exit(1);
                }

                ren = std::make_unique<HybridShader>(&world, system.clmanager.get());
                std::cout << "Creating Hybrid Shader\n";
                break;
            case Config::HYBRID_CPU:
                ren = std::make_unique<HybridShaderCPU>(&world);
                std::cout << "Creating Hybrid CPU Shader\n";
                break;
            default:
                ren = std::make_unique<DeferredShader>(&world);
                std::cout << "Creating Deferred Shader\n";
        }
        Config::option_rendering_method_change_requested = false;
        return ren;
    }
};

#endif // RENDERENGINECREATOR_H
```

11.5.36 shaderloader.hpp

```
*****
* ShaderLoader
*
* Loads a GLSL OpenGL shader from a disk file and
```

```

* compiles it.
*
* 2017 - Liberto Camús
* ****
#ifndef SHADERLOADER_H
#define SHADERLOADER_H

#include <GL/glew.h> // for GLuint, GLchar
#include <string> // for string

class ShaderLoader {
public:
    ShaderLoader(const std::string &path, GLuint shader_type);
    ~ShaderLoader();
    GLuint getId() { return mId; }

private:
    GLuint mId;
    std::string readFile(const std::string &path);
    void compile(const GLchar *, GLuint);
};

#endif // SHADERLOADER_H

```

11.5.37 shaderloader.cpp

```

*****
* ShaderLoader
*
* Loads a GLSL OpenGL shader from a disk file and
* compiles it.
*
* 2017 - Liberto Camús
* ****
#include "shaderloader.hpp"
#include <fstream>
#include <iostream>
#include <sstream>

ShaderLoader::ShaderLoader(const std::string &path, GLuint shader_type)
    : mId{0} {
    std::string vertexCodeStr = readFile(path);
    const GLchar *code = vertexCodeStr.c_str();
    compile(code, shader_type);
}

ShaderLoader::~ShaderLoader() {
    if (mId != 0)
        glDeleteShader(mId);
}

void ShaderLoader::compile(const GLchar *code, GLuint shader_type) {
    // Compile shader
    mId = glCreateShader(shader_type);
    glShaderSource(mId, 1, &code, NULL);
    glCompileShader(mId);

    // Check the result of compilation
    GLint success;
    glGetShaderiv(mId, GL_COMPILE_STATUS, &success);
    if (!success) {
        GLchar infoLog[512];
        glGetShaderInfoLog(mId, 512, NULL, infoLog);
        std::cerr << "Error compiling GLSL shader\n" << infoLog;
        std::exit(1);
}

```

```

}

std::string ShaderLoader::readFile(const std::string &path) {
    std::string code{""};
    try {
        std::ifstream file;
        file.exceptions(std::ifstream::failbit | std::ifstream::badbit);
        file.open(path);
        std::stringstream fStream;
        fStream << file.rdbuf();
        file.close();
        code = fStream.str();
    } catch (std::ifstream::failure &e) {
        std::cerr << "Error loading GLSL shader from disk\n";
        std::cerr << "File: " << path << std::endl;
        std::exit(1);
    }
    return code;
}

```

11.5.38 shaderprogram.hpp

```

*****
* ShaderProgram
*
* It stores a Shader program and allows to use it.
* During its initialization loads a vertex and a fragment
* shaders (using ShaderLoader) and creates the program,
* linking them.
*
* 2017 - Liberto Camús
*****
#ifndef SHADERPROGRAM_H
#define SHADERPROGRAM_H

#include <GL/glew.h> // for GLuint, GLchar
#include <string> // for string

class ShaderProgram {
public:
    ShaderProgram(const GLchar *vertexSourcePath,
                  const GLchar *fragmentSourcePath);
    ~ShaderProgram();
    void use() const;
    GLuint getUniformLocation(const std::string &name) const;

private:
    GLuint mProgram;
    void createProgram(GLuint, GLuint);
};

#endif // SHADERPROGRAM_H

```

11.5.39 shaderprogram.cpp

```

*****
* ShaderProgram
*
* It stores a Shader program and allows to use it.
* During its initialization loads a vertex and a fragment
* shaders (using ShaderLoader) and creates the program,
* linking them.
*
* 2017 - Liberto Camús
*****

```

```
#include "shaderprogram.hpp"
#include "shaderloader.hpp"
#include <fstream>
#include <iostream>
#include <sstream>

ShaderProgram::ShaderProgram(const GLchar *vertexPath,
                             const GLchar *fragmentPath) {
    auto vertexShader = ShaderLoader(vertexPath, GL_VERTEX_SHADER);
    auto fragmentShader = ShaderLoader(fragmentPath, GL_FRAGMENT_SHADER);
    createProgram(vertexShader.getId(), fragmentShader.getId());
}

GLuint ShaderProgram::getUniformLocation(const std::string &name) const {
    return glGetUniformLocation(mProgram, name.data());
}

ShaderProgram::~ShaderProgram() {
    if (mProgram != 0)
        glDeleteProgram(mProgram);
}

void ShaderProgram::createProgram(GLuint vertexId, GLuint fragmentId) {
    mProgram = glCreateProgram();
    glAttachShader(mProgram, vertexId);
    glAttachShader(mProgram, fragmentId);
    glLinkProgram(mProgram);

    // Check the result
    GLint success = 0;
    glGetProgramiv(mProgram, GL_LINK_STATUS, &success);
    if (!success) {
        GLchar infoLog[512];
        glGetProgramInfoLog(mProgram, 512, NULL, infoLog);
        std::cerr << "Error linking OpenGL shaders \n" << infoLog << std::endl;
        std::exit(1);
    }
}

void ShaderProgram::use() const { glUseProgram(mProgram); }
```

11.5.40 system.hpp

```
*****
* System
*
* Contains the timer, input and window objects
*
* 2017 - Liberto Camús
* *****/
#ifndef SYSTEM_H
#define SYSTEM_H

#include "clkernelmanager.hpp"
#include "input.hpp"
#include "timer.hpp"
#include "window.hpp"
#include <memory>

class System {
public:
    std::unique_ptr<CLKernelManager> clmanager;

    System() : input{window.createInput()} {}

    void update() {
```

```

window.pollEvents();
timer.update();
window.swapBuffers();
input.move(static_cast<GLfloat>(timer.getDelta()) * 1000); /// We need ms
updateStatistics();

if (Config::option_statistics_requested) {
    Config::option_statistics_requested = false;
    printStatistics();
}
if (Config::option_reset_statistics_requested) {
    resetStatistics();
    Config::option_reset_statistics_requested = false;
}

nr_frames++;

void initOpenCL() {
    clmanager = std::make_unique<CLKernelManager>();
    if (Config::option_opencl_available)
        clmanager->loadKernelFromFile(Config::lighting_kernel);
}

void resetStatistics() {
    max_time = 0;
    min_time = 999999;
    total_time = 0;
    nr_frames = 0;
}

void setCamera(Camera *c) { input.setCamera(c); }
bool exitRequested() { return window.shouldClose(); }
void printStatistics() {
    // nr_frames -= 1;
    std::cout << "Rendering method: ";
    switch (Config::rendering_method) {
    case Config::DEFERRED:
        std::cout << "Deferred OpenGL shader\n";
        break;
    case Config::HYBRID:
        std::cout << "Hybrid OpenCL shader\n";
        break;
    default:
        std::cout << "Hybrid CPU shader\n";
        break;
    }
    std::cout << "Average FPS: "
        << (double)nr_frames * 1000 / (double)total_time << "\n";
    std::cout << "Nr of frames: " << nr_frames << "\n";
    std::cout << "Min time: " << min_time << " (ms)\n";
    std::cout << "Max time: " << max_time << " (ms)\n";
}

private:
    Window window;
    Input input;
    Timer timer;

    unsigned int max_time = 0;
    unsigned int min_time = 999999;
    unsigned int total_time = 0;
    unsigned int nr_frames = 0;

    void updateStatistics() {
        if (nr_frames == 0)
            return;

        double delta = timer.getDelta() * 1000; // ms
        max_time = max_time > delta ? max_time : (unsigned int)delta;
    }
}

```

```
    min_time = min_time < delta ? min_time : (unsigned int)delta;
    total_time += static_cast<unsigned int>(delta);
}
#endif // SYSTEM_H
```

11.5.41 texture.hpp

```
*****
* Texture
*
* Contains OpenGL texture id and its type
* (diffuse, specular, normal...)
*
* 2017 - Liberto Camús
*****
#ifndef TEXTURE_H
#define TEXTURE_H

#include <GL/glew.h> // for GLuint

enum class TextureType { diffuse, specular, normal };

class Texture {
public:
    GLuint id;
    GLuint uniformId;
};

#endif // TEXTURE_H
```

11.5.42 textureloader.hpp

```
*****
* TextureLoader
*
* Loads a texture from disk into memory and converts it
* to RGBA format if it wasn't already.
* Uses FreeImage library
*
* 2017 - Liberto Camús
*****
#ifndef TEXTURELOADER_H
#define TEXTURELOADER_H

#include <GL/glew.h> // for GLbyte
#include <memory> // for unique_ptr
#include <string> // for string
struct FIBITMAP;

class TextureLoader {
public:
    explicit TextureLoader(const std::string &filename);
    ~TextureLoader();
    const GLbyte *getImagePtr() const;
    unsigned int getWidth() const { return width; }
    unsigned int getHeight() const { return height; }

private:
    FIBITMAP *dib{nullptr};
    unsigned int width{0};
    unsigned int height{0};
};


```

```
#endif // TEXTURELOADER_H
```

11.5.43 textureloader.cpp

```
*****
* TextureLoader
*
* Loads a texture from disk into memory and converts it
* to RGBA format if it wasn't already.
* Uses FreeImage library
*
* 2017 - Liberto Camús
*****
#include "textureloader.hpp"
#include <FreeImage.h> // for BYTE, FreeImage_Unload, etc
#include <GL/glew.h> // for GLbyte
#include <iostream>
#include <memory> // for unique_ptr, make_unique
#include <string> // for string

TextureLoader::TextureLoader(const std::string &filename) {
    // Check the file signature and deduce its format
    auto fif = FreeImage_GetFileType(filename.c_str(), 0);
    if (fif == FIF_UNKNOWN)
        fif = FreeImage_GetFIFFFromFilename(filename.c_str());
    if (fif == FIF_UNKNOWN) {
        std::cout << "Unknown image type. Error loading texture " << filename
              << "\n";
        return;
    }

    // check that the plugin has reading capabilities and load the file
    if (FreeImage_FIFSupportsReading(fif))
        dib = FreeImage_Load(fif, filename.c_str());
    // if the image failed to load, return failure
    if (!dib) {
        std::cout << "Error loading texture " << filename << "\n";
        return;
    }

    // Convert non-32 bit images
    if (FreeImage_GetBPP(dib) != 32) {
        FIBITMAP *oldDib = dib;
        dib = FreeImage_ConvertTo32Bits(oldDib);
        FreeImage_Unload(oldDib);
    }

    BYTE *bits = FreeImage_GetBits(dib);
    width = FreeImage_GetWidth(dib);
    height = FreeImage_GetHeight(dib);

    // Test for errors
    if ((bits == nullptr) || (width == 0) || (height == 0))
        std::cout << "Error loading texture " << filename << "\n";
}

const GLbyte *TextureLoader::getImagePtr() const {
    return reinterpret_cast<const signed char *>(FreeImage_GetBits(dib));
}

TextureLoader::~TextureLoader() {
    if (dib)
        FreeImage_Unload(dib);
}
```

11.5.44 texturemanager.hpp

```
*****
* TextureManager
*
* Loads textures corresponding to ASSIMP materials from disk,
* using "TextureLoader", uploads it to OpenGL GPU memory and
* returns a "Texture" object with its type and OpenGL id.
* Stores internally already loaded textures to avoid loading a
* texture more than once.
*
* 2017 - Liberto Camús
* ****
#ifndef TEXTUREMANAGER_H
#define TEXTUREMANAGER_H

#include "mesh.hpp"          // for Texture
#include <GL/glew.h>         // for GLbyte, GLint, GLuint
#include <assimp/material.h> // for aiMaterial (ptr only), aiTextureType
#include <string>             // for string
#include <vector>             // for vector

class TextureManager {
public:
    std::vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type,
                                                const std::string &directory);

private:
    // Struct for internal use to avoid loading duplicated textures
    struct TextureLoaded {
        GLuint id;
        std::string path;
    };

    std::vector<TextureLoaded> textures_loaded;
    GLint TextureFromFile(const std::string &filename) const;
};

#endif // TEXTUREMANAGER_H
```

11.5.45 texturemanager.cpp

```
*****
* TextureManager
*
* Loads textures corresponding to ASSIMP materials from disk,
* using "TextureLoader", uploads it to OpenGL GPU memory and
* returns a "Texture" object with its type and OpenGL id.
* Stores internally already loaded textures to avoid loading a
* texture more than once.
*
* 2017 - Liberto Camús
* ****
#include "texturemanager.hpp"
#include "textureloader.hpp" // for TextureLoader
#include "vertex.hpp"
#include <GL/glew.h> // for GL_TEXTURE_2D, glTexParameteri, etc
#include <algorithm>
#include <string> // for allocator, string, operator+, etc
#include <vector> // for vector

std::vector<Texture>
TextureManager::loadMaterialTextures(aiMaterial *mat, aiTextureType aiType,
                                    const std::string &directory) {

    std::vector<Texture> textures;
```

```

for (GLuint i = 0; i < mat->GetTextureCount(aiType); i++) {
    // Get texture name
    aiString str;
    mat->GetTexture(aiType, i, &str);
    auto path = std::string(str.C_Str());

    // Find out if we have already loaded it.
    auto found =
        std::find_if(begin(textures_loaded), end(textures_loaded),
                     [&path](TextureLoaded &t) { return t.path == path; });

    if (found != end(textures_loaded)) {
        // The texture was already loaded
        textures.push_back(Texture{(*found).id, 0});
    } else {
        // Load a new texture
        TextureLoaded texture;
        texture.id = TextureFromFile(directory + "/" + path);
        texture.path = path;
        textures.push_back(Texture{texture.id, 0});
        textures_loaded.push_back(std::move(texture));
    }
}

return textures;
}

GLint TextureManager::TextureFromFile(const std::string &filename) const {

    // Generate texture ID and load texture data
    GLuint textureID = 0;
    glGenTextures(1, &textureID);

    // Load the image
    TextureLoader image{filename};
    if (image.getImagePtr() == nullptr) {
        std::cout << "Failed loading " << filename << "\n";
    }

    // Assign texture to ID
    glBindTexture(GL_TEXTURE_2D, textureID);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, image.getWidth(), image.getHeight(),
                0, GL_BGRA, GL_UNSIGNED_BYTE, image.getImagePtr());
    glGenerateMipmap(GL_TEXTURE_2D);

    glBindTexture(GL_TEXTURE_2D, 0);
    std::cout << "Imported texture " << filename << "\n";

    return textureID;
}

```

11.5.46 timer.hpp

```

*****
* Timer
*
* Keeps track of time elapsed between calls to its
* "update" method.
*
* 2017 - Liberto Camús
*****

```

```
#ifndef TIMER_H
#define TIMER_H
#include <GL/glew.h> // for GLfloat
#include <GLFW/glfw3.h> // for glfwGetTime

class Timer {
public:
    Timer() : lastTime{glfwGetTime()} {}
    ~Timer() {}
    void update() {
        double currentTime = glfwGetTime();
        deltaTime = currentTime - lastTime;
        lastTime = currentTime;
    }
    double getDelta() const { return deltaTime; }

private:
    double lastTime, deltaTime;
};

#endif // TIMER_H
```

11.5.47 triangle.hpp

```
#ifndef TRIANGLE_H
#define TRIANGLE_H
#include <glm/glm.hpp>

#include "bbox.hpp"
#include <CL/cl_gl.h>

class Triangle {
public:
    Triangle(glm::vec3 &vert1, glm::vec3 &vert2, glm::vec3 &vert3)
        : v1{{vert1.x, vert1.y, vert1.z}}, v2{{vert2.x, vert2.y, vert2.z}},
          v3{{vert3.x, vert3.y, vert3.z}} {}

    cl_float3 v1, v2, v3;

    BBox getBbox() const {
        return Union(BBox(glm::vec3(v1.x, v1.y, v1.z), glm::vec3(v2.x, v2.y, v2.z)),
                    glm::vec3(v3.x, v3.y, v3.z));
    }
};

#endif // TRIANGLE_H

#endif // TRIANGLE_H
```

11.5.48 vertex.hpp

```
*****
* Vertex
*
* Contains a vertex in a format suitable to be passed
* to OpenGL. Stores its position, normal and texture
* coordinates
*
* 2017 - Liberto Camús
* *****/
#ifndef VERTEX_H
```

```
#define VERTEX_H
#include <GL/glew.h>      // for GLuint
#include <algorithm>        // for move
#include <assimp/types.h> // for aiString
#include <glm/glm.hpp>
#include <string> // for string
#include <vector> // for vector

class Vertex {
public:
    Vertex() = default;
    Vertex(Vertex &&other)
        : Position(std::move(other.Position)), Normal(std::move(other.Normal)),
          TexCoords(std::move(other.TexCoords)) {}
    Vertex(glm::vec3 pos, glm::vec3 nor, glm::vec2 tex)
        : Position(std::move(pos)), Normal(std::move(nor)),
          TexCoords(std::move(tex)) {}
    Vertex &operator=(const Vertex &other) = default;
    Vertex(const Vertex &other) = default;

    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
};

#endif // VERTEX_H
```

11.5.49 window.hpp

```
*****
* Window
*
* Desktop window initialization, event polling and swaping
* methods. Uses GLFW3 and GLEW
*
* 2017 - Liberto Camús
*****
#ifndef WINDOW_H
#define WINDOW_H

// GLEW
// #define GLEW_STATIC
#include <GL/glew.h>

// GLFW
#include "config.hpp"
#include <GLFW/glfw3.h>

class Input;

class Window {
public:
    Window();
    ~Window();
    bool shouldClose();
    void pollEvents();
    void swapBuffers();
    Input createInput();

    void move();

    GLuint WIDTH = Config::window_width;
    GLuint HEIGHT = Config::window_height;
    float RATIO = (float)WIDTH / (float)HEIGHT;
```

```
private:
    GLFWwindow *window;
};

#endif // WINDOW_H
```

11.5.50 window.cpp

```
*****  

* Window  

*  

* Desktop window initialization, event polling and swaping  

* methods. Uses GLFW3 and GLEW  

*  

* 2017 - Liberto Camús  

* *****  

#include "window.hpp"
#include "input.hpp"
#include <functional>

Window::Window() {
    // Init GLFW
    if (!glfwInit()) {
        std::cerr << "GLFW Error initializing" << std::endl;
        std::exit(1);
    }

    // Set all the required options for GLFW
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

    window = glfwCreateWindow(WIDTH, HEIGHT, "ASHPOOL", nullptr, nullptr);
    if (window == nullptr) {
        std::cerr << "GLFW Error creating window" << std::endl;
        exit(1);
    }
    glfwMakeContextCurrent(window);
    glfwSwapInterval(0); // 0-> no sync, more than 60fps possible

    // Options
    if (Config::option_no_capture_mouse == true)
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
    else
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

    // Set this to true so GLEW knows to use a modern approach to retrieving
    // function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    GLenum glewres = glewInit();
    if (GLEW_OK != glewres) {
        std::cerr << "Error: " << glewGetString(glewres) << std::endl;
        std::exit(1);
    }

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);

    glEnable(GL_CULL_FACE);
    glEnable(GL_ALPHA);
    // glBindFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    // glEnable(GL_BLEND);
    glCullFace(GL_FRONT);
    glFrontFace(GL_CW);
```

```

    glEnable(GL_DEPTH_TEST);

Input Window::createInput() { return Input(window); }

Window::~Window() { glfwTerminate(); }

bool Window::shouldClose() { return glfwWindowShouldClose(window); }

void Window::pollEvents() { glfwPollEvents(); }

void Window::swapBuffers() { glfwSwapBuffers(window); }

```

11.5.51 world.hpp

```

/****************************************************************************
 * World
 *
 * Main Scene object. Contains its models, lights,
 * cameras, BVH geometry structure and options.
 *
 * 2017 - Liberto Camús
 * ****
#ifndef WORLD_H
#define WORLD_H

#include "model.hpp"
#include <memory> // for unique_ptr
#include <vector> // for vector
class Camera;
class ShaderProgram;
#include "bvh.hpp"
#include "pointlight.hpp"
#include "triangle.hpp"

class World {
public:
    struct SceneAttrs {
        cl_float ambient;
        cl_bool shadowsEnabled;
    };
    SceneAttrs scene_attrs;

    World();
    ~World() {}
    void init();
    void initModelsUniforms(ShaderProgram &shader);

    Camera *getCamera() const { return camera.get(); }
    const std::vector<Model> *getModels() const { return &models; }
    size_t getPointLightsNr() const { return PointLights.size(); }
    const std::vector<PointLight> &getPointLights() const { return PointLights; }
    void update();
    BVH bvh;

private:
    std::unique_ptr<Camera> camera;
    std::vector<Model> models;
    // lights
    std::vector<PointLight> PointLights;
};

#endif // WORLD_H

```

11.5.52 world.cpp

```
*****
* World
*
* Main Scene object. Contains its models, lights,
* cameras, BVH geometry structure and options.
*
* 2017 - Liberto Camús
*****
```

```
#include "world.hpp"
#include "camera.hpp"      // for Camera
#include "model.hpp"        // for Model
#include "modelloader.hpp"  // for ModelLoader
#include <memory>           // for make_unique, allocator
#include <iostream>
```

```
World::World()
    : scene_attrs{Config::ambient, Config::option_shadows_enabled},
    camera{std::make_unique<Camera>()} {
```

```
// Load models
ModelLoader mLoader;
for (std::string model : Config::models)
    models.emplace_back(mLoader.loadModel(model));
```

```
// Export its geometry to the BVH acceleration structure
for (auto &m : models) {
    auto t = m.ExportTriangles();
    std::copy_n(t.data(), t.size(), back_inserter(bvh.triangles));
}
```

```
// Initialize BVH
bvh.init();
```

```
}
```

```
void World::initModelsUniforms(ShaderProgram &shader) {
    for (auto &m : models) {
        m.refreshUniforms(shader);
    };
}
```

```
void World::init() {
    // Load Point Lights from "Config"
    enum pos : size_t { pos_x = 0, pos_y, pos_z, color_r, color_g, color_b,
        intensity, linear, quadratic, pos_end };

    for (std::vector<float> result : Config::point_lights) {
        PointLight p(result[pos_x], result[pos_y], result[pos_z]);
        p.color = cl_float3{{result[color_r], result[color_g], result[color_b]}};
        p.intensity = result[intensity];
        p.linear = result[linear];
        p.quadratic = result[quadratic];
        PointLights.push_back(p);
    }
}
```

```
void World::update() {
    scene_attrs.shadowsEnabled = Config::option_shadows_enabled;
}
```

11.5.53 kernels/render.cl

```
*****
* render.cl
*
* Uses G-Buffer textures (Diffuse+Specular, Positions, Normals),
```

```

* a Bounding Volume Hierarchy containing the geometry (triangles) and
* lighting information to produce a rendered texture.
*
* 2017 - Liberto Camús
*****constant sampler_t imageSampler =
CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;

// Minimum distance to avoid self-intersection
constant float EPSILON = 0.001f;
// Threshold to discard light contribution
constant float ATTENUATION_SENSITIVITY = 0.02f;

// Struct of each Bounding Volume Hierarchy node
typedef struct _BVHNode {
    float3 bounds_pMin;
    float3 bounds_pMax;
    union {
        uint primitivesOffset; // leaf
        uint secondChildOffset; // interior
    } uf;
    uchar nPrimitives; // 0 -> Interior
    uchar axis; // interior node: xyz
} BVHNode;

typedef struct _Ray {
    float3 o; // origin
    float3 d; // direction
    float mint;
    float maxt;
} Ray;

typedef struct _Triangle { float3 p0, p1, p2; } Triangle;

typedef struct _PointLight {
    float3 p_position;
    float3 p_color;
    float intensity;
    float linear;
    float quadratic;
} PointLight;

typedef struct _SceneAttribs {
    float ambient;
    int shadows_enabled;
} SceneAttribs;

// Checks ray-triangle intersection
// Based on algorithm in "Physically based rendering, 2nd edition"
bool test_ray_triangle(__global const Triangle *tri, const Ray *ray) {
    float3 e1 = tri->p1 - tri->p0;
    float3 e2 = tri->p2 - tri->p0;

    float3 s1 = cross(ray->d, e2);
    float a = dot(s1, e1);
    if (a > -EPSILON && a < EPSILON)
        return false;

    float invDivisor = 1.0f / a;

    float3 d = ray->o - tri->p0;
    float b1 = invDivisor * dot(d, s1);
    if (b1 < 0.0f || b1 > 1.0f)
        return false;

    float3 s2 = cross(d, e1);
    float b2 = invDivisor * dot(ray->d, s2);
    if (b2 < 0.0f || b1 + b2 > 1.0f)
        return false;
}

```

```

float t = invDivisor * dot(e2, s2);
if (t < ray->mint || t > ray->maxt)
    return false;
else
    return true;
}

// Checks ray-bounding box intersection
// Based on algorithm in "Physically based rendering, 2nd edition"
bool test_ray_bbox(const Ray *ray, __global const BVHNode *node,
                   const float3 *invDir) {

    float3 tNear = (node->bounds_pMin - ray->o) * (*invDir);
    float3 tFar = (node->bounds_pMax - ray->o) * (*invDir);

    float tmin = max(ray->mint, min(tNear.x, tFar.x));
    float tmax = min(ray->maxt, max(tNear.x, tFar.x));
    if (tmin > tmax)
        return false;

    tmin = max(tmin, min(tNear.y, tFar.y));
    tmax = min(tmax, max(tNear.y, tFar.y));
    if (tmin > tmax)
        return false;

    tmin = max(tmin, min(tNear.z, tFar.z));
    tmax = min(tmax, max(tNear.z, tFar.z));
    if (tmin > tmax)
        return false;

    return true;
}

// Checks if a ray intersects with a collection of triangles, using
// a BVH tree to accelerate the process.
// Based on algorithm in "Physically based rendering, 2nd edition"
bool intersects(const Ray *ray, __global const Triangle *triangles,
                 __global const BVHNode *nodes) {

    int todoOffset = 0, nodeNum = 0;
    int todo[64];

    float3 invDir = (float3)(1.0f / ray->d.x, 1.0f / ray->d.y, 1.0f / ray->d.z);
    // int dirIsNeg[3] = {invDir.x < 0, invDir.y < 0, invDir.z < 0};

    while (true) {
        const __global BVHNode *node = &nodes[nodeNum];
        // Check ray against BVH node
        if (test_ray_bbox(ray, node, &invDir)) {
            if (node->nPrimitives > 0) {
                // Intersect ray with primitives in leaf BVH node
                for (int i = 0; i < node->nPrimitives; i++)
                    if (test_ray_triangle(&triangles[node->uf.primitivesOffset + i], ray))
                        return true;
                if (todoOffset == 0)
                    break;
                nodeNum = todo[--todoOffset];
            } else {
                // Put far BVH node on todo stack, advance to near node
                // if (dirIsNeg[node->axis]) {
                todo[todoOffset++] = nodeNum + 1;
                nodeNum = node->uf.secondChildOffset;
                //}
                // todo[todoOffset++] = node->uf.secondChildOffset;
                // nodeNum = nodeNum + 1;
                //}
            }
        } else {
            if (todoOffset == 0)
                break;
        }
    }
}

```

```

    nodeNum = todo[--todoOffset];
}
return false;
}

// Calculates point lights contribution to shading a specific point
// on a surface
float3 pointLightsColor(__constant PointLight *point_lights,
                        const int point_lights_nr, const float3 view_pos,
                        const float3 surface_pos, const float3 surface_normal,
                        const float3 surface_diffuse,
                        const float surface_specular, bool shadows_enabled,
                        __global const Triangle *triangles,
                        __global const BVHNode *nodes) {

float3 color = (float3)(0.0f, 0.0f, 0.0f);
const float3 view_dir = fast_normalize(view_pos - surface_pos);

for (int i = 0; i < point_lights_nr; i++) {
  __constant PointLight *pLight = &point_lights[i];

  // Attenuation
  const float dist = fast_distance(pLight->p_position, surface_pos);
  const float attenuation = 1.0f / (1.0f + (pLight->linear * dist) +
                                    (pLight->quadratic * dist * dist));
  if (attenuation < ATTENUATION_SENSITIVITY && shadows_enabled)
    continue;

  const float3 light_dir = fast_normalize(pLight->p_position - surface_pos);
  if (shadows_enabled) {
    Ray r = (Ray){surface_pos, light_dir, EPSILON, dist};
    if (intersects(&r, triangles, nodes))
      continue;
  }

  // Diffuse Lambertian
  const float ang = max(dot(surface_normal, light_dir), 0.0f);
  const float3 diffuse =
    ang * surface_diffuse * pLight->p_color * pLight->intensity;
  color += diffuse * attenuation;

  // Specular Blinn-Phong
  if (surface_specular > 0.0f) {
    const float3 halfway_dir = fast_normalize(light_dir + view_dir);
    const float spec =
      pow(max(dot(surface_normal, halfway_dir), 0.0f), 16.0f);
    const float3 specular = pLight->p_color * spec * surface_specular;
    color += specular * attenuation;
  }
}

return color;
}

//
// Main function. Kernel "render"
//
__kernel void
render(__read_only image2d_t g_albedo_spec, __read_only image2d_t g_position,
       __read_only image2d_t g_normal, __write_only image2d_t output,
       const SceneAttrs attrs, const float3 view_position,
       __constant PointLight *point_lights, const int point_lights_nr,
       __global const Triangle *triangles, __global const BVHNode *nodes) {

int2 coord = (int2)(get_global_id(0), get_global_id(1));
const float3 diffuse = read_imagef(g_albedo_spec, imageSampler, coord).xyz;
const float specular = read_imagef(g_albedo_spec, imageSampler, coord).w;
const float3 pos = read_imagef(g_position, imageSampler, coord).xyz;

```

```

const float3 norm = read_imagef(g_normal, imageSampler, coord).xyz;
const float3 normal = fast_normalize(2 * norm - (float3)(1, 1, 1));

// Ambient light
float3 color = diffuse * attribs.ambient;

// Point Lights (diffuse + specular)
color += pointLightsColor(point_lights, point_lights_nr, view_position, pos,
                           normal, diffuse, specular,
                           (bool)attribs.shadows_enabled, triangles, nodes);

// Write result
write_imagef(output, coord, (float4)(color.x, color.y, color.z, 0.0f));
}

```

11.5.54 shaders/gbuffer.vert

```

#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

out vec3 FragPos;
out vec2 TexCoords;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    vec4 worldPos = model * vec4(position, 1.0f);
    mat3 normalMatrix = transpose(inverse(mat3(model)));

    FragPos = worldPos.xyz;
    TexCoords = texCoords;
    Normal = normalMatrix * normal;

    gl_Position = projection * view * worldPos;
}

```

11.5.55 shaders/gbuffer.frag

```

#version 330 core
layout(location = 0) out vec4 gPosition;
layout(location = 1) out vec4 gNormal;
layout(location = 2) out vec4 gAlbedoSpec;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;

struct Material {
    vec3 ambient;
    vec3 diffuse;
    float specular;
    float shininess;
    int tex_count;
};

uniform Material material;
uniform sampler2D texture_diffuse0;

```

```

uniform sampler2D texture_specular0;
uniform sampler2D texture_normal0;

void main() {
    // Store the per-fragment normals into the gbuffer - Change [-1..1] to [0..1]
    gNormal = 0.5 * (vec4(1.0, 1.0, 1.0, 0.0) + vec4(normalize(Normal), 0.0f));

    // Store position
    gPosition = vec4(FragPos, 0);

    // Store albedo & specular
    gAlbedoSpec = vec4(material.diffuse, material.specular);

    // Use albedo texture if present
    if (material.tex_count > 0) {
        vec4 texColor = texture(texture_diffuse0, TexCoords).rgba;
        // handle transparency
        if (texColor.a < 0.1)
            discard;
        gAlbedoSpec.rgb *= texColor.rgb;
    }
}

```

11.5.56 shaders/lighting.vert

```

#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoords;

out vec2 TexCoords;

void main()
{
    gl_Position = vec4(position.rgb, 1.0f);
    TexCoords = texCoords;
}

```

11.5.57 shaders/lighting.frag

```

#version 330 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;

struct Light {
    vec3 Position;
    vec3 Color;
    float Linear;
    float Quadratic;
    float Shininess;
};
const int MAX_LIGHTS = 25;
uniform int NR_LIGHTS;
uniform Light lights[MAX_LIGHTS];
uniform vec3 viewPos;
uniform float ambient;

void main() {
    // Retrieve data from gbuffer
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
}

```

```

// Change [0..1] to [-1..1]
Normal = normalize(2 * Normal - vec3(1, 1, 1));
vec3 Diffuse = texture(gAlbedoSpec, TexCoords).rgb;
float Specular = texture(gAlbedoSpec, TexCoords).a;
// Then calculate lighting as usual
vec3 lighting = Diffuse * ambient;

vec3 viewDir = normalize(viewPos - FragPos);
for (int i = 0; i < NR_LIGHTS; ++i) {
    // Diffuse
    vec3 lightDir = normalize(lights[i].Position - FragPos);
    vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse * lights[i].Color *
        lights[i].Shininess;
    // Specular
    vec3 halfwayDir = normalize(lightDir + viewDir);
    float spec = pow(max(dot(Normal, halfwayDir), 0.0), 16.0);
    vec3 specular = lights[i].Color * spec * Specular;
    // Attenuation
    float distance = length(lights[i].Position - FragPos);
    float attenuation = 1.0 / (1.0 + lights[i].Linear * distance +
        lights[i].Quadratic * distance * distance);
    diffuse *= attenuation;
    specular *= attenuation;
    lighting += diffuse + specular;
}
FragColor = vec4(lighting, 1.0);
}
  
```

11.5.58 CMakeLists.txt

`cmake_minimum_required(VERSION 3.1)`

```

project(ashpool)
set(CMAKE_CXX_STANDARD 14)

IF(WIN32)
    set(LIBS_PREFIX "C:/lab/libs/")
    set(glm3_DIR "${LIBS_PREFIX}GLFW/lib/cmake/glfw3")
    set(GLM_ROOT_DIR "${LIBS_PREFIX}/glm")
    set(FREEIMAGE_ROOT_DIR "${LIBS_PREFIX}/FreeImage")
    set(OpenCL_LIBRARY
        "C:/Program Files (x86)/AMD APP SDK/3.0/lib/x86_64/OpenCL.lib")
    set(OpenCL_INCLUDE_DIR "C:/Program Files (x86)/AMD APP SDK/3.0/include")
    set(CMAKE_PREFIX_PATH "${LIBS_PREFIX}/GLEW" "${LIBS_PREFIX}/FreeImage" "${LIBS_PREFIX}/Assimp/lib/cmake/assimp-3.3")
    set(CMAKE_LIBRARY_PATH "${LIBS_PREFIX}/GLEW/lib/Release/x64/" "${LIBS_PREFIX}/FreeImage/Dist/x64")
ELSE()
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3 -Wall -Wextra -pedantic -fno-omit-frame-pointer -fno-rtti -ggdb -Wconversion")
ENDIF()

set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
    "${LIBS_PREFIX}/cmake"
    "${CMAKE_SOURCE_DIR}/cmake/Modules"
    "${CMAKE_CURRENT_SOURCE_DIR}/../cmake/
    "${CMAKE_CURRENT_SOURCE_DIR}/Modules")

add_executable(ashpool
    bbox.hpp
    bvh.hpp bvh.cpp
    bvhlinernode.hpp
    bvhprimitiveinfo.hpp
    camera.hpp camera.cpp
    cl_device.cpp cl_device.hpp
    clkernelmanager.cpp clkernelmanager.hpp
    cl_platform.cpp cl_platform.hpp
    config.hpp
    configloader.cpp configloader.hpp
    deferredshader.cpp deferredshader.hpp
    hybridshader.cpp hybridshader.hpp)
  
```

Anexos

```

hybridshadercpu.cpp hybridshadercpu.hpp
input.hpp input.cpp
main.cpp
material.hpp
mesh.cpp mesh.hpp
model.hpp model.cpp
modelloader.cpp modelloader.hpp
pointlight.hpp
renderengine.hpp
renderenginecreator.hpp
shaderloader.cpp shaderloader.hpp
shaderprogram.cpp shaderprogram.hpp
system.hpp
texture.hpp
textureloader.cpp textureloader.hpp
texturemanager.cpp texturemanager.hpp
timer.hpp
triangle.hpp
vertex.hpp
window.cpp window.hpp
world.cpp world.hpp
)

find_package(ASSIMP REQUIRED)
find_package(glfw3 3.2 REQUIRED)
find_package(OpenGL REQUIRED)
find_package(GLEW REQUIRED)
find_package(glm REQUIRED)
find_package(FreeImage REQUIRED)
find_package(OpenCL REQUIRED)

find_package(OpenMP)
if (OPENMP_FOUND)
    set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
endif()

if(WIN32)
    SET(ASSIMP_LIBRERIA ${ASSIMP_LIBRARY_DIRS}/${ASSIMP_LIBRARIES})
ELSE()
    SET(ASSIMP_LIBRERIA ${ASSIMP_LIBRARIES})
endif()

include_directories( ${OPENGL_INCLUDE_DIRS}
${GLFW_INCLUDE_DIRS}
${GLEW_INCLUDE_DIRS}
${GLM_INCLUDE_DIRS}
${ASSIMP_INCLUDE_DIRS}
${FREEIMAGE_INCLUDE_PATH}
${OpenCL_INCLUDE_DIRS}
)

target_link_libraries(ashpool
${OPENGL_LIBRARIES}
${OPENGL_gl_LIBRARY}
glfw ${GLFW_LIBRARIES}
GLEW::GLEW
${GLEW_LIBRARY}
${GLM_LIBRARY}
${ASSIMP_LIBRERIA}
${FREEIMAGE_LIBRARIES}
${OpenCL_LIBRARIES}
)

```

