TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Framework para abstracción de entornos heterogéneos CPU/GPU

Optimización de renderización por cálculo de occlusion culling en paralelo

Autor
Juan Carlos Pulido Poveda

Director
Alejandro José León Salas

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Junio de 2019

Índice

Índice de figuras

Introducción

La proliferación de multicores y Graphic Processing Units (GPUs) ha permitido desarrollar un modelo de programación paralelo que aísla de la programación a bajo nivel de la GPU mediante OpenGL Shading Language. Este modelo de programación es común a OpenCL y CUDA. Sin embargo, la planificación desde el punto de vista de los Sistemas Operativos de los recursos, tanto multicore como GPUs, de cara a las aplicaciones no ha llegado a un modelo ampliamente aceptado.

En este proyecto se desarrollará un framework que proporcione recursos de computación a las aplicaciones de manera que les permita aislarse de los detalles de bajo nivel a la vez que les proporcione de herramientas para poder adaptar la asignación de los recursos a sus necesidades reales. En este caso nos centraremos en intentar optimizar el la optimización del proceso de renderizado de escenas obviando aquellos objetos que estén ocultos por otros de forma que no se desperdicie tiempo en cálculos que luego no serán realmente útiles.

Este trabajo viene de la motivación provocada por la falta de contenido abierto que existe en el ámbito de los gráficos, si bien es cierto que existen librerías que ayudan a la realización de dicha tarea, todas suelen ser de código cerrado o soluciones que hayan implementado directamente los desarrolladores de los distintos motores gráficos que se utilizan en la actualidad ya sea Unreal Engine o Unity. Los cuales aún siendo gratuitos de usar, no podemos saber qué algoritmos/tecnologías los componen ni si pueden ser mejoradas o adaptadas para las próximas tecnologías.

De ahí este intento de crear algo que pueda ser ampliado por la comunidad para liberarse de las grandes empresas detrás de los nombrados motores gráficos construyendo software moderno y que se aproveche de las tecnologías de paralelismo que disponemos en la actualidad.

Estructura del documento

Primero realizaremos un repaso a cómo se encuentra el panorama actual en lo que a computación altamente paralela se refiere (CUDA, OpenCL) y como las distintas librerías de gráficos actuales nos permiten reprogramar el pipeline gráfico a nuestro gusto obteniendo un gran rendimiento

Después se hará análisis exhaustivo de nuestro problema desde la perspectiva de ingeniería del software y los requisitos que acarrea. Se describirá la metodología usada en este proyecto, las fases que se han llevado a cabo y los distintos problemas y contratiempos que se hayan encontrado en el camino a la hora de la planificación. Más adelante se proveerá de una especificación formal tanto de objetivos como los distintos tipos de requisitos funcionales, no funcionales y de información del proyecto Luego nos adentraremos en la parte de planificación del proyecto, las fases que se plantearon al principio y las distintas fases que le siguieron junto con las correcciones que se hubo de hacer.

Por consiguiente se habrá de discutir el diseño de la solución y cómo subdividir el problema para poder adaptarlo de la manera más eficiente sin sacrificar la simplicidad y usabilidad de las distintas partes de la librería. En el siguiente capítulo se hablará de las distintas decisiones tomadas basadas en un análisis previo de mercado y lo que ofrece cada tecnología disponible.

Por último encontraremos varias partes consecutivas hablando tanto de la implementación final, preparación del entorno, el diseño de las pruebas del software y los distintos casos de uso donde debería ser útil seguido por las conclusiones y líneas de futuro para el proyecto.

Estado del arte

Conceptos fundamentales

Inicios

Para poder enfrentarnos al problema primero debemos entender perfectamente la arquitectura que tienen las tarjetas gráficas (GPU de aquí en adelante) también llamados aceleradores gráficos fueron diseñaron para solventar los distintos problemas que fueron surgiendo los cuáles se vieron que no eran como los que la computación tradicional había tenido siempre en mente a la hora de construir y diseñar los procesadores.

Las GPU nacieron de la necesidad de computación en gráficos en los años 90 donde se vió que el computar cada uno de los píxeles de la pantalla, un problema *O(n*m)* siendo *n* la altura de la pantalla y *m* la anchura, que es altamente paralelizable como explican ya que los valores de los píxeles no tienen ninguna dependencia entre ellos mismos.

Las grandes empresas como NVIDIA y ATI empezaron a diseñar chips que contenían muchos procesadores simples pero optimizados para el pipeline gráfico que no es más que una serie de pasos que se deben de seguir para transformar una serie de vértices y caras definidas por el programador hasta llegar a presentar los colores de los píxeles por pantalla.

Según McClanahan (2010) el hecho de que el pipeline gráfico fue pensado y diseñado con anterioridad hizo que las empresas pudieran fijarse a dicho modelo proporcionando así un ecosistema más consistente entre distintos fabricantes permitiendo así una mayor aceleración de acogida de las librerías de gráficos. Todo esto

provocó una explosión en el apartado de la computación gráfica haciendo que pudiera evolucionar de una manera mucho más rápida.

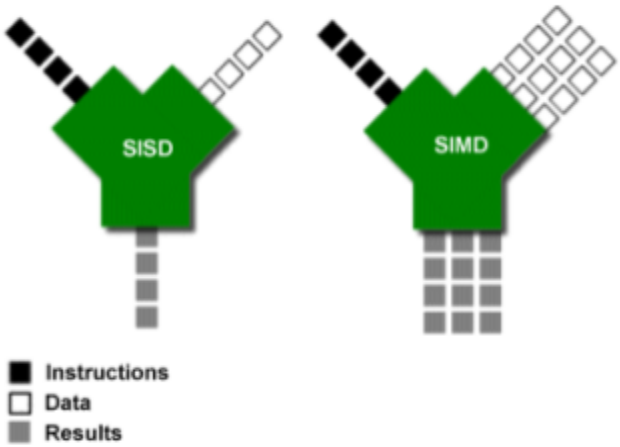


Figura 1 - Single Instruction Multiple Data (SIMD) architecture (Arstechnica, 2000)

Estos procesadores se encargaban de aplicar las mismas instrucciones (Single Instruction) a los distintos valores, normalmente vértices y caras, almacenados en memoria (Multiple Data) para obtener al final un valor RGB que será mostrado en pantalla como en la figura 1 se explica. De esta forma se aceleraría notablemente el proceso de renderizado con respecto a la CPU.

Evolución de las GPUs

Al cabo del tiempo las GPUs y librerías gráficas fueron evolucionando para ser más flexibles y poder reprogramar el pipeline gráfico para permitir más personalización del cómputo de los gráficos. Es aquí cuando surgen los lenguajes de Shading de las distintas librerías gráficas OpenGL y DirectX cada uno con OpenGL Shading Language (OpenGLSL) y High Level Shader Language (HLSL) respectivamente.

En este caso hablaremos de las respectivas partes de código abierto ya que son aquellas que podemos saber perfectamente cómo funcionan y dejaremos otras soluciones como DirectX de Microsoft de lado debido a su naturaleza cerrada y exclusiva.

En OpenGL se introdujo la capacidad de los shaders en la versión 2.0 en el año

2004 como ya se explica en el documento de especificación de GLSL Language Specification, Version 1.10.59 (Kessenich & Baldwin & Rost, 2004) se creó con la motivación de que el nivel de computación que se podía hacer por vértice o fragment había avanzado tanto que el mecanismo tradicional que había seguido OpenGL se había quedado obsoleto y no era capaz de llevar las tarjetas gráficas a su máxima capacidad. También perseguían aplicar los mismo conceptos que los lenguajes tradicionales como C++ habían seguido, buscando proporcionar una plataforma común que permitiera al programador crear código portable y de alto nivel sin tener que preocuparse de dónde iba a ser ejecutado ni otros aspectos de bajo nivel.

General Purpose GPU Computing (GPGPU)

Todos estos avances en el campo permitieron crear el siguiente nivel de abstracción y es que las tarjetas gráficas pudieran hacer cómputo no solo de vértices, iluminación, etc sino que fueran capaces de computar en paralelo cualquier cosa que se pudiera definir o lo que es lo mismo General Purpose GPU computing (GPGPU). Para perseguir este fin se crearon varias librerías para proporcionar al programador con herramientas: OpenCL del lado del código abierto y CUDA del lado de NVIDIA.

En la especificación de la versión 1.0 de OpenCL (Khronos OpenCL Working Group, 2009) se nos habla de cómo en la actualidad más que nunca se necesitan herramientas que permitan a los programadores extraer el máximo beneficio de las distintas plataformas de cómputo heterogéneas como pueden ser las CPUs y las GPUs desde un bajo nivel de abstracción con alto rendimiento haciendo uso del alto paralelismo que esta nos proporciona. CUDA también persigue la misma misión que OpenCL en este sentido, solo que es exclusivo de las GPUs de NVIDIA y por lo tanto proporciona un poco más de atención al detalle ya que no tiene que adaptarse a todos los dispositivos del mercado aunque este punto se tratará más adelante.

Análisis de Frameworks con ambos enfoques

Una vez planteado el espectro de recursos que disponemos en la actualidad lo interesante sería usar distintos recursos en función del tipo de problema que nos enfrentamos, por ejemplo si nuestro problema es de generar árboles y mantenerlos ordenados como se hacen en las bases de datos nos interesa usar CPUs rápidas con todos las memorias caches que tienen para ir lo más rápido posible sin embargo nos enfrentamos a un problema donde debemos de ejecutar miles de iteraciones sobre datos que son independientes unos de otros, lo lógico es usar la GPU y aprovechar al máximo el alto grado de paralelismo que esta nos proporciona.

Esta visión se llama enfoque híbrido o enfoque de sistemas heterogéneos que tiene en cuenta las distintas naturalezas de los componentes para asignar los trabajos de una forma u otra. El problema es que en la actualidad no hay muchas ofertas de librerías que permitan hacer esto en los distintos aspectos de la computación. Encontramos librerías como Heterogenerous Programming Libray (HPL) pero no son todo lo grandes, estables y extendidas que necesitamos en la actualidad.

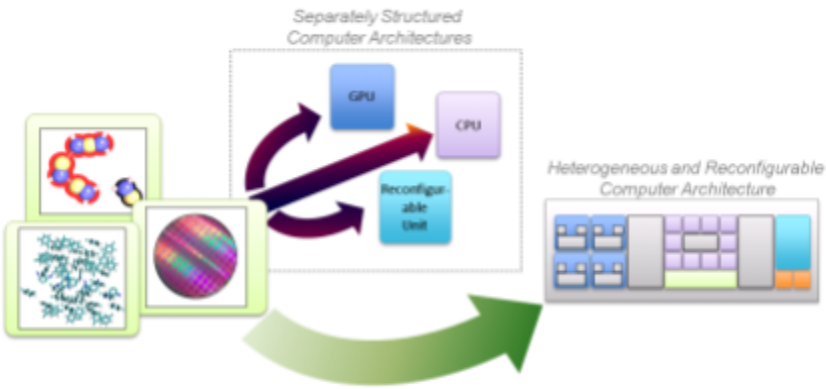


Figura 2 - Heterogeneous Computing (Institut für Technische Informatik, 2017)

Crítica de las posibilidades ofertadas

En este sentido la respuesta es sencilla ya que las ofertas que se disponen en la actualidad son pocas o inexistentes.

Como se ha comentado antes existen algunas librerías que facilitan la tarea de implementar algoritmos que aprovechen las capacidades de un sistema heterogéneo pero ahí acaba la oferta. No hay librerías completas con algoritmos para cómputo de gráficos en general y que aprovechen al máximo los recursos.

En lo que a Occlusion Culling se refiere hasta ahora se han inventado ciertos algoritmos y trucos que solventan ciertos problemas de los comentados como ya nos habla Bazhenov (2017) en su post donde expone todas las formas que existen hasta la fecha para calcular la oclusión de objetos además de las ventajas y desventajas que suponen cada una de ellas. Las formas son las siguientes:

1. Software occlusion culling inventada por Intel y optimizada para sus procesadores que tienen como ventaja su baja latencia pero no puede con muchos objetos en escena

2. Depth buffer que vuelve a renderizar la escena completa a una menor resolución pero en vez de almacenar los valores RGB se almacena la distancia hacia la cámara de cada pixel. Sus ventajas es que funciona bien con sombras y escenas dinámicas pero como contrapartida la GPU debe volver renderizar la escena para saber qué objetos son visibles, lo cual es un contrasentido.

Todo esto nos lleva a pensar que estamos en un área menos investigada y estudiada debido a la novedad del GPGPU computing ocasionando que no esté tan extendido

Nuestra Propuesta

Nuestra propuesta consiste en crear un framework para aprovechar los sistemas heterogéneos de forma eficiente en lo que a geometría de los gráficos se refiere. En específico el problema de realizar el Occlusion Culling donde dado una posición de cámara y la dirección donde mira se hará una poda de objetos que están ocultos por otros desde la perspectiva de la cámara.

Para ello se hará uso de las capacidades de la CPU para particionar el espacio mediante un Árbol Octree y la GPU para comprobar si los objetos son ocluidos por otros, el algoritmo se explicará con profundidad en el siguiente apartado. De esta forma obtendremos un enfoque híbrido de nuestro problema utilizando lo mejor de cada dispositivo.

Más adelante se irá ampliando el framework poco a poco intentando abordar los distintos problemas con la misma mentalidad de distinguir si un problema puede ser abordado mejor por una CPU de cómputo general o si por el contrario sería mejor paralelizar lo máximo posible y aprovechar la gran capacidad de la GPU en ese sentido.

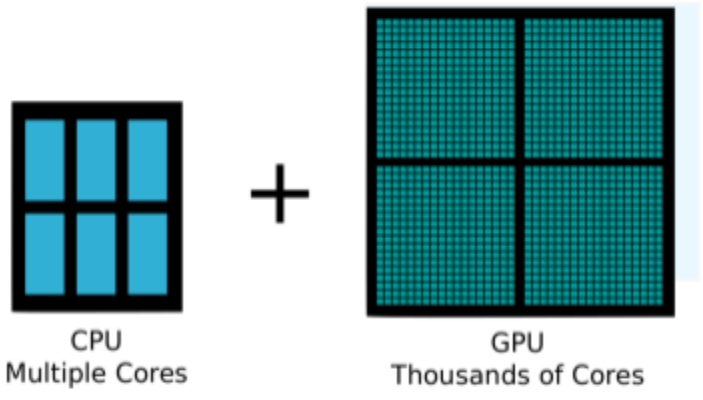


Figura 3 - CPU & GPU architecture (Profesionalreview, 2017)

Análisis del problema

Análisis de requisitos

El objetivo de este proyecto es el de empezar lo que sería una librería/framework que resuelva distintos problemas gráficos que nos podamos encontrar siempre teniendo en mente las capacidades de los dispositivos que disponemos en los equipos actuales, es decir de aprovechar todas las optimizaciones de la CPU para cómputo complejo y las virtudes de la GPU para solventar problemas con muchísimos datos que se puedan hacer un uso eficiente de los miles de threads que nos proveen las GPUs.

Sabiendo esto nos adentraremos en los requisitos que se buscan cumplir en este proyecto:

- La librería proveerá de un patrón de clases para facilitar la integración de código GPU y código ejecutado en CPU en futuras ampliaciones.
- Se deberá de hacer las mínimas asunciones posibles de las capacidades del hardware para facilitar la integración del software
- La librería hará uso de la GPU para tareas repetitivas e independientes entre sí.
- La librería debe de calcular correctamente cuando un objeto está ocluido o no.
- Debe de ser lo suficientemente rápida como para poder ejecutarse en tiempo real y suponer una mejora en la cantidad fotogramas por segundo renderizados.

Planificación

Para realizar la planificación se ha tenido en cuenta las distintas tecnologías que se habrían de aprender y practicar, el tiempo de resolución de algoritmos eficientes para la GPU, las personas implicadas y el tiempo del que disponemos. Después de haber realizado una estimación de las partes anteriores ha tocado hacer la elección de metodología de trabajo ágil que se más adecue a la situación actual.

En lo que a recursos hardware se refiere se deberá de tener una tarjeta gráfica de NVIDIA que soporte CUDA 3.0 (a partir GTX 600) como mínimo y en cuanto al software debemos de instalar algunas librerías de código abierto para el cómputo de geometría y el SDK de CUDA. Debido a la falta de manejo con dichas herramientas se ha estimado un tiempo para el aprendizaje de las mismas y añadido un tiempo para posibles imprevistos en la implementación/integración de las mismas.

También se han tenido en cuenta a las personas que trabajan en el proyecto y el tiempo disponible para poder concertar citas cada cierto tiempo. Las personas involucradas en el proyecto han sido: el tutor del proyecto y yo. Mi parte tratará de realizar gran parte del diseño e implementación del proyecto.Mientras que la labor del tutor será la de guiar y proporcionar consejo/experiencia sobre las distintas dudas que vayan surgiendo con el paso del tiempo.

En cuanto al tiempo disponible dependerá de las demás asignaturas escogidas por lo que deberemos de permitir cierta flexibilidad en todos las fases para acomodar las distintas variaciones de tiempo disponible que puedan ocurrir a lo largo del tiempo. Así como también puede ocurrir con el tutor del proyecto cuyo tiempo de disponibilidad también oscile dependiendo de otros aspectos ajenos al proyecto.

Después de haber hecho un análisis de los distintos factores del proyecto se debe de elegir una metodología de trabajo. Debido a la complejidad del proyecto usaremos una metodología ágil de las estudiadas en las asignaturas de ingeniería del software que nos proporcione flexibilidad y rapidez para adaptarnos a los cambios.

Hemos optado por seguir una metodología de prototipado con algunas características de la metodología scrum. La razón es clara ya que al enfrentarnos a un problema de alta complejidad lo podemos subdividir en problemas más simples e ir ampliando incrementalmente. Un ejemplo es primero realizar las estructuras de datos encargadas de hacer la subdivisión espacial para más tarde hacer los algoritmos paralelos de cálculo de oclusiones. De esta forma vamos completando nuestro objetivo final poco a poco.

Se han determinado las siguientes fases para la realización de este proyecto:

1. Investigación inicial
    1.1. Aprendizaje CUDA
    1.2. Aprendizaje Estructuras de particionado de espacio
2. Diseño del software
    2.1. Diseño de algoritmos
    2.2. Diseño de clases
    2.3. Adaptación de librerías a usar
3. Implementación
    3.1. Implementación Octree
    3.2. Implementar la parte de CUDA
4. Redacción del TFG
5. Realización de la presentación

Resolución de problemas gráficos

Detección de oclusión de objetos de forma paralela

Para empezar a pensar una solución debemos de simplificar todos los modelos de los objetos a los mismas características. Lo primero que pensamos es en una primitiva simple de las que ya estudiaba Ericson (2004, p. 76) en Real-Time Collision Detection donde se nos expone las características deseables en los Bounding Volumes. Las distintas características que encontramos son varias: se ajuste bien al modelo, sean fáciles de computar, fáciles de computar el test de colisión contra ellas mismas como otras primitivas como planos y puntos.

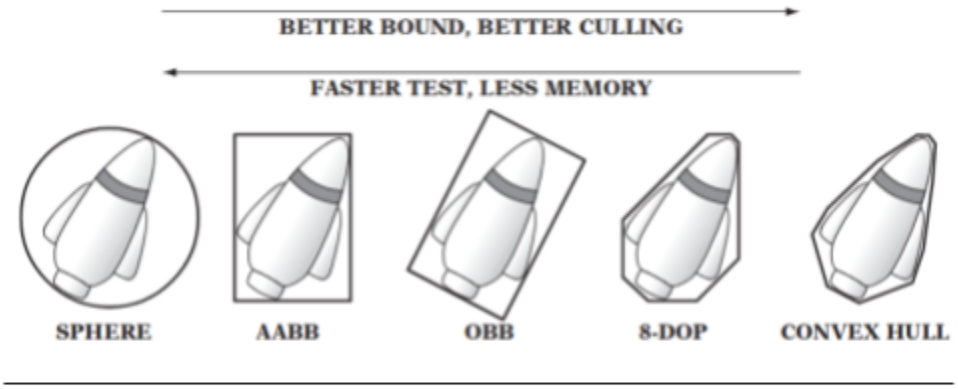


Figura 4 - Comparación de las distintas primitivas (Ericson, 2004, p. 77).

Se debe elegir una opción que sea eficiente a la hora de hacer rotaciones de objetos y simple a la vez para que la GPU no tenga muchos problemas a la hora de procesar miles de ellas. Esto nos deja con la opción de bounding Sphere, las cajas alineadas con los ejes (AABB) y las cajas alineadas con objeto (OBB). Al estudiar nuestro problema con más profundidad vemos que las cajas alineadas con los ejes no nos sirven ya que cada vez que se rotase la cámara deberían de recalcularse cada una de ellas para que estas siguiesen englobando al objeto pero a la vez también continuasen alineadas con los ejes lo que las hace una mala opción de implementación.

Ahora solo nos quedarían las esferas contra las OBB y en este caso podemos ver rápidamente que las esferas producirán muchos falsos positivos haciendo que el algoritmo no funcione correctamente ocasionando al final problemas de flickering, donde los objetos desaparezcan y aparezcan rápidamente, como también problemas de popping donde objetos que deberían ser renderizados no lo estén siendo debido a que la esfera englobante no nos está proporcionando la precisión necesaria.

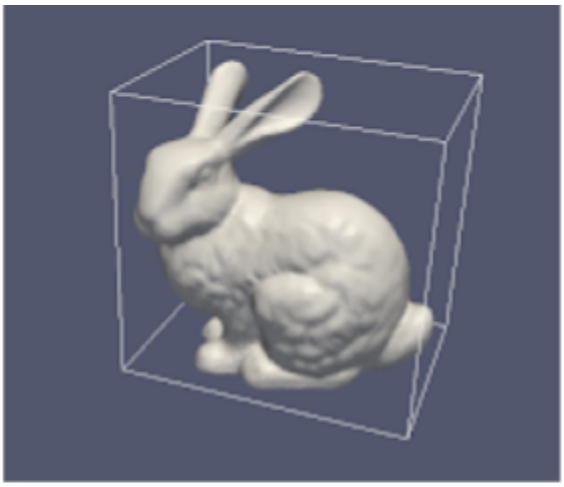


Figura 5 - Object Oriented Bounding Box
(Gregson, 2011)

Después de haber elegido nuestra estructura englobante ahora nos tocaría diseñar el algoritmo para detectar si una caja tapa a otra dado un punto que representa la posición de la cámara y un vector con la dirección de la misma.

Para empezar debemos de dejar claro cierto vocabulario que se usará en el pseudocódigo para entender bien cuando nos referimos a uno u a otro. El algoritmo a definir tendrá como entrada dos vectores: ***camera_pos*** y ***camera_dir*** que definirá la posición y dirección de la cámara en un espacio 3D, también obtendrá las 8 esquinas del objeto que comprobar si ocluye o no, será referenciado como ***occluder***. Por último tenemos las 8 esquinas el objeto que queremos comprobar si es ocluido por occluder, esta vez llamado ***occludee***. El algoritmo devolverá True o False en función si occluder oculta completamente occludee o no.

Un primer acercamiento al algoritmo sería calcular las 6 normales que definen las caras de los planos occluder y comprobar si los rayos definido por cada una de las esquinas del occludee intersecan con los planos o no.

En las siguientes figuras podemos ver una representación del problema que estamos tratando de resolver y los distintos casos posibles con los que nos encontramos

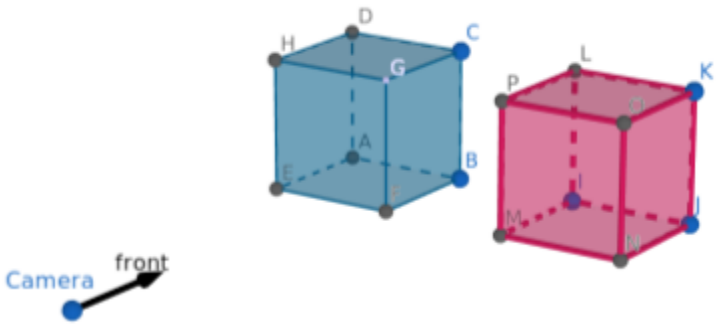


Figura 6 - Caso sin oclusión

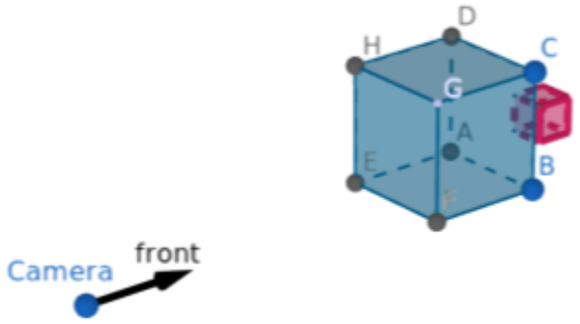


Figura 7 - Caso con oclusión

Después de haber revisado los casos posibles nos damos cuenta que comprobar los 6 planos no es correcto ya que solo debemos comprobar aquellas caras que sean visibles desde la cámara. Esto no resulta mayor problema ya que solo debemos de resolver la siguiente ecuación ('·' representa el producto escalar) :

- $(V_0 - P) \cdot N \geq 0$
  - V_0 es un punto del plano N
  - P es la posición de la cámara
  - N es la normal del plano

Ahora nos tocaría calcular si la intersección del rayo con el plano cae dentro del cubo o no debido a que los planos son infinitos y comprobar si intersecan o no nos asegura que el punto está detrás del cubo.

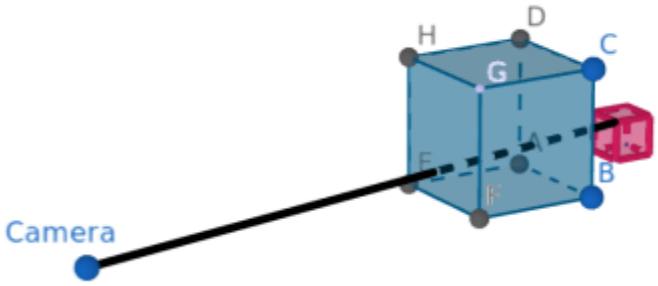


Figura 8 - Rayo desde puntos de occludee a cámara

Para comprobar si un punto cae dentro de la cara del cubo que se está computando he seguido la idea de Pradhan, S. (2018) donde nos explica para comprobar si un punto cae dentro de un rectángulo una forma sencilla y rápida es ver si el área de los 4 triángulos que se forman entre el punto y las 4 esquinas del rectángulo a comprobar coinciden. En las siguientes ilustraciones se ven claramente los 4 triángulos formados y que la suma de sus áreas es igual que el área del rectángulo.

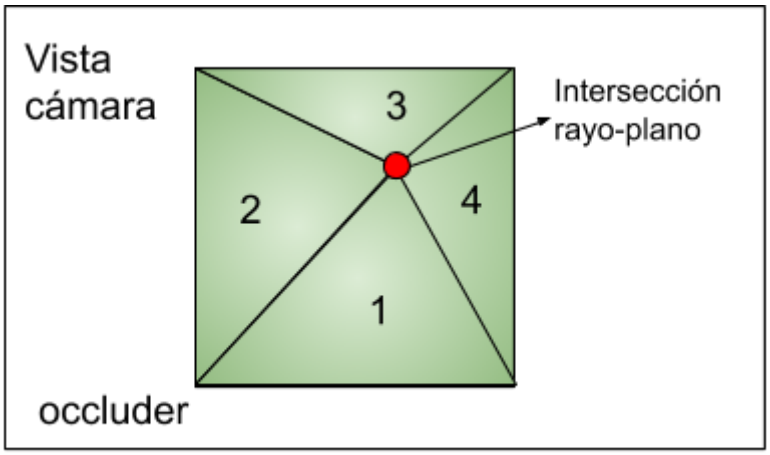


Figura 9 - Comprobación de intersección punto ocluido

Como se puede ver el área de los triángulos 1 + 2 + 3 + 4 es igual que la del rectángulo que los contiene. Por tanto solo nos queda calcular el área de un triángulo dado 3 puntos y la ecuación es sencilla ('×' representa el producto vectorial y '||' representa el módulo del vector):

- $área = \frac{1}{2}|\overline{AB} \times \overline{AC}|$
  - A, B y C son los tres puntos que definen al triángulo
  - $\overline{AB}\ y\ \overline{AC}$ son los vectores que indican que empiezan en A y terminan en B y C respectivamente

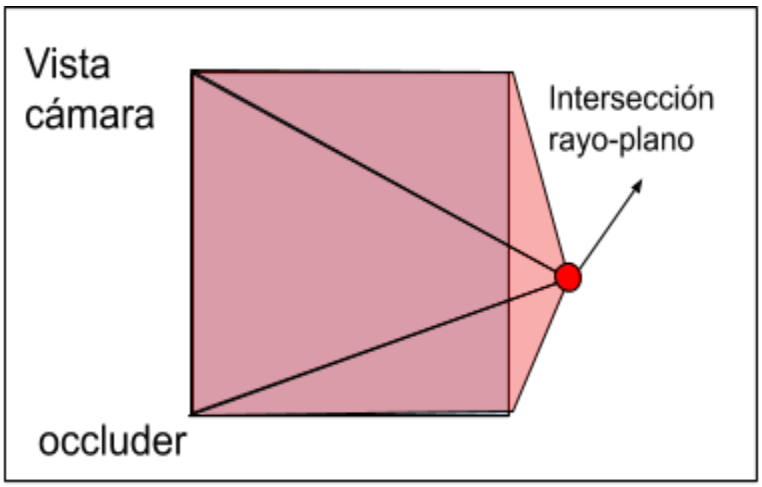


Figura 10 - Comprobación de intersección punto no ocluido

Vemos que claramente si el punto no se encuentra dentro del rectángulo la suma de las áreas no es igual a la del rectángulo. Ahora ya podemos calcular correctamente cuando un punto está ocluido por una cara del cubo y por extensión podemos comprobar si un cubo ocluye todos los puntos de otro. Lo que significa que podemos determinar si un cubo es visible desde un punto de perspectiva o no completando nuestro objetivo y resolviendo nuestro problema.

Especificación de Requisitos

Objetivos

Los objetivos de esta librería ya han sido listados con anterioridad en otras secciones, ahora nos los limitaremos a juntarlos y especificarlos con más detalle.

1. Solucionar problemas gráficos mediante el uso de recursos en los sistemas heterogéneos
2. Hacer uso de alto paralelismo de la GPU
3. Definir una jerarquía de clases o patrón a seguir para poder extender la librería de una forma consistente para solucionar más problemas en el futuro
4. Solucionar el problema del cálculo de occlusion culling de gran cantidad de objetos de forma lo suficientemente eficiente como para poder hacerse en tiempo real

Requisitos Funcionales

- Debe calcular correctamente de las oclusiones de las figuras
- Debe usar el paralelismo de la GPU
- Debe de hacerse la serialización correctamente
- Debe de evitar comportamiento indeterminado por concurrencia

Requisitos no funcionales

- Debe de ser fácilmente extensible
- Debe de ser rápido
- Debe de usar la orientación de objetos en la medida de lo posible (CUDA no permite la creación de clases)
- Debe de facilitar la integración GPU y CPU
- Debe aprovechar los mecanismos caché de CPU y GPU
- Debe ser escalable
- Debe ser usable por otros proyectos

Requisitos de información

- Debe aceptar modelos de ‘sopa de triángulos’
- Debe aceptar gran cantidad de objetos

Planificación

Metodología empleada

Como ya se ha empezado a explicar en otra sección anterior se ha optado por seguir una metodología de prototipado con algunas características de la metodología scrum. La razón es clara ya que al enfrentarnos a un problema de alta complejidad lo podemos subdividir en problemas más simples e ir ampliando incrementalmente. Un ejemplo es primero realizar las estructuras de datos encargadas de hacer la subdivisión espacial para más tarde hacer los algoritmos paralelos de cálculo de oclusiones. De esta forma vamos completando nuestro objetivo final poco a poco.

Los prototipos se han estructurado de la siguiente forma:

1. Prototipo del árbol Octree debe:
    a. Añadir correctamente los objetos insertados
    b. Debe de subdividir los nodos cuando se cumpla la condición de máximo nº de objetos por cada nodo
    c. Debe mantener correctamente ordenada la lista enlazada de nodos por nivel, de forma que se pueda iterar todos los nodos de todos los niveles.
2. Prototipo de Occlusion culling:
    a. Se deben de calcular bien las normales de las caras del cubo
    b. Se debe de computar correctamente el back face culling para evitar hacer comprobaciones erróneas
    c. Se debe de calcular correctamente el área de los triángulos para discernir si un punto se encuentra ocluido o no.
3. Integración de ambos:
    a. El Octree debe pasarle correctamente los objetos a la función encargada de hacer Occlusion culling.

Fases de desarrollo

Se han determinado las siguientes fases para la realización de este proyecto:

1. Investigación inicial
2. Diseño del software
3. Implementación
4. Redacción del TFG
5. Realización de la presentación

Como es lógico primero se ha de estudiar y empezar investigar sobre el contenido que ya existe y qué es lo que todavía no se ha diseñado para saber cuál será el lugar que nuestro software deberá ocupar. Esta fase nos llevará ciertas semanas para poder averiguar qué soluciones de código abierto existen y podemos usar. Una vez estudiado las soluciones disponibles se habrá de estudiar nuevas herramientas a usar como son CUDA u otras librerías para el proyecto.

Después nos encontramos con la fase de diseño del software donde debemos de crear una jerarquía de clases útil para nuestro problema además de ir pensando las distintas estructuras que representarán cada una de ellas. Por ejemplo en este paso se debe de ir pensando una forma eficiente de almacenar los objetos del árbol para luego poder hacer copias a la GPU de la forma más eficiente aprovechando todos los niveles de caché disponibles. Esta fase será más costosa que la de investigación debido a su mayor complejidad y esfuerzo requerido.

La siguiente fase que nos encontramos es la de implementación. En esta fase se encuentra el grueso del trabajo y es por ello que se ha dejado más tiempo para permitir cierta flexibilidad con los plazos debido a que esta fase es la que más contratiempos puede presentar.

Más adelante nos encontramos la redacción de este documento. Esta fase no debería presentar ningún contratiempo sólo cierto trabajo constante para poder escribir el documento correctamente junto con todas las referencias y bibliografía necesaria.

Finalmente nos quedaría la fase de creación de la presentación que debería la fase más simple ya que solo habrá que recopilar la información extraída en el documento y ponerla presentable y visible de forma que sea fácilmente entendible en la presentación. Esta fase tampoco debería presentar ningún contratiempo ya que solo se deberá concentrar la información recogida en el documento.

En la siguiente página tenemos una disposición visual de cómo se distribuye el tiempo para cada una de las tareas propuestas y qué porcentaje se ha completado de cada una de las mismas.

Fases del Proyecto

WBS	Task description	Finish date	Progress
1	Investigación inicial	04/04/2019	100%
2	Aprendizaje CUDA	04/04/2019	100%
3	Aprendizaje Estructuras de particio	04/04/2019	100%
4	Diseño del software	04/05/2019	100%
5	Adaptacion de librerías	14/04/2019	100%
6	Diseño de algoritmos	26/04/2019	100%
7	Diseño de clases	04/05/2019	100%
8	Implementación	06/06/2019	100%
9	Implementación Octree	19/05/2019	100%
10	Implementar la parte de CUDA	06/06/2019	100%
11	Redacción del TFG	17/06/2019	100%
12	Realización de la presentación	23/06/2019	0%

Figura 11 - Diagrama las distintas fases del proyecto

Planificación Inicial

Según la planificación inicial el proyecto nos debería durar 112 días de los cuales el tiempo se distribuirá de la siguiente forma:

Investigación	28.57%
Diseño	26.78%
Implementación	32.14%
Redacción documento	6.25%
Presentación	4.46%

Figura 12 - Tabla de distribución del tiempo según las fases

Como podemos el proceso de investigación se lleva una gran porción del tiempo debido a que es clave aprender bien las tecnologías a usar en el proyecto además de estudiar bien las soluciones que ya están disponibles para no repetir trabajo que ya han hecho otros y es perfectamente usable/aplicable en nuestro proyecto.

Obviamente se reserva más del 50% para el diseño + implementación del proyecto ya que es donde se encuentra el grueso del proyecto y es lo que le dará peso sobre la importancia que tendrá realmente o no en la realidad. Si sumamos todos los porcentajes vemos que no llegan exactamente a 100% y de ahí viene el periodo de flexibilidad reservado para los posibles contratiempos en la implementación del proyecto.

Por último encontramos el bloque de redacción del proyecto y presentación que conllevan alrededor de 10% del total del proyecto ya que puede llevar cierto esfuerzo pero debería suponer un trabajo mayor como los anteriores.

Presupuesto

Tabla de gastos

Tipo de gasto	Precio por Unidad	Importe Total
GASTO DE PERSONAL		
Gasto de contratación de personal	2064 €/mes	7840€
GASTOS DE EJECUCIÓN		
Costes de adquisición de material inventariable		
Ordenadores	1500€	1500€
Costes de adquisición de material fungible		
Herramientas y programas varios	Gratuitos	0€
Costes de investigación y aprendizaje	2240€	2240€
Costes de consultoría	Gratuitos	0€
GASTOS COMPLEMENTARIOS		
Desplazamientos	Gratuitos	0€
TOTAL INCENTIVO SOLICITADO		**11580€**

Figura 13 - Tabla de gastos del proyecto

- Para realizar la estimación del salario se ha usado el salario medio de un junior software engineer en España propuesto por glassdoor
- Los costes de consultoría son 0 debido a que son consultas al tutor del proyecto
- Desplazamientos son siempre a pie por tanto gratuitos

Evoluciones de la planificación inicial

Gracias a la investigación inicial se ha encontrado una librería de código abierto que nos va a ahorrar mucho trabajo en la creación de clases en lo referente a cálculo de geometría. Además está creada con un patrón de orientación a objetos por lo que será muy fácil de agregar al proyecto.

Después de haber aprendido los distintos detalles de la programación altamente paralela de CUDA hemos aprendido que es importante que haya poca divergencia entre las hebras para obtener el mejor rendimiento posible. La divergencia se refiere a dos tipos de comportamientos divergentes:

- Divergencia de código donde las hebras se encuentran ejecutando líneas de código distintas causando una importante disminución de la velocidad de la GPU y por ende aumentando la latencia del software.

- Divergencia de datos, donde cada hebra está tratando datos alojados muy lejanamente causando un *thrashing* continuo de la caché al pedir variables muy lejanas en memoria haciendo fallar todos los niveles de caché disponibles disminuyendo una vez más el rendimiento

En la parte de implementación se ha encontrado con ciertos fallos de segmento que han tenido que ir siendo depurados poco a poco hasta que la clase Octree no causa ninguno y todos los objetos se almacenan correctamente. Las operaciones básicas (CRUD) se realizan con normalidad y no rompen el árbol después de ser ejecutadas.

La redacción del proyecto avanza con normalidad y parece continuar correctamente y no presenta ningún contratiempo.

Diseño

Arquitectura del Software

Como arquitectura software se usará un modelo cliente servidor donde la librería actúa como servidor a la que se solo se le debe de proporcionar los objetos que estarán en escena y cada vez que queramos que se nos compute la oclusión de los objetos deberemos de darle la posición y dirección de la cámara junto con el frustum de visión.

De esta forma obtenemos un sistema altamente encapsulado en el que el cliente no debe de preocuparse de ninguno de los detalles que debe mantener la librería internamente. Aunque no es un cliente y servidor al uso ya que ambos se encontrarían en el mismo equipo pero la filosofía detrás de dicho modelo sigue presente.

- El servidor se encarga de manejar todas las estructuras y estados necesarios para completar una función de una forma eficiente.

- El cliente solo debe de preocuparse de proporcionar los datos necesarios para inicializar el primer estado del servidor y luego hacer las peticiones necesarias.

El hecho de que el patrón usual no se siga, donde el servidor y el cliente están en equipos diferentes, no quiere decir que no se siga respetando la filosofía. Es por ello que no creo que costase mucho modificar la librería existente para permitir que ambos se encuentren en equipos distintos, pero en este caso sería inviable separarlos debido a que la latencia de la red supondría que el software no pudiera cumplir con su cometido de ser usado para entornos 3D en tiempo real.

Diagramas UML

En las siguientes páginas se dispondrán una serie de figuras con los distintos diagramas necesarios para poder especificar cómo estará dispuesto el software.

Primero veremos un diagrama mostrando qué clases componen la librería y cada uno de los atributos junto con la visibilidad, lo que es un diagrama de clases al uso. Se ha de decir que hay clases en blanco debido a que pertenecen a otra librería y por tanto desde nuestra perspectiva son cajas negras de las cuales solo sabemos que métodos las componen y cómo usarlas, pero no sabemos qué atributos las componen y por tanto no podemos describirlas en el diagrama de clases.

Después veremos los distintos diagramas de secuencia donde se dispondrá todas los mensajes que se generan cuando se realiza una llamada a cada uno de los métodos. Esto nos ayudará a comprender y delimitar muy bien el comportamiento de cada uno de los métodos y la complejidad de cada uno de ellos.

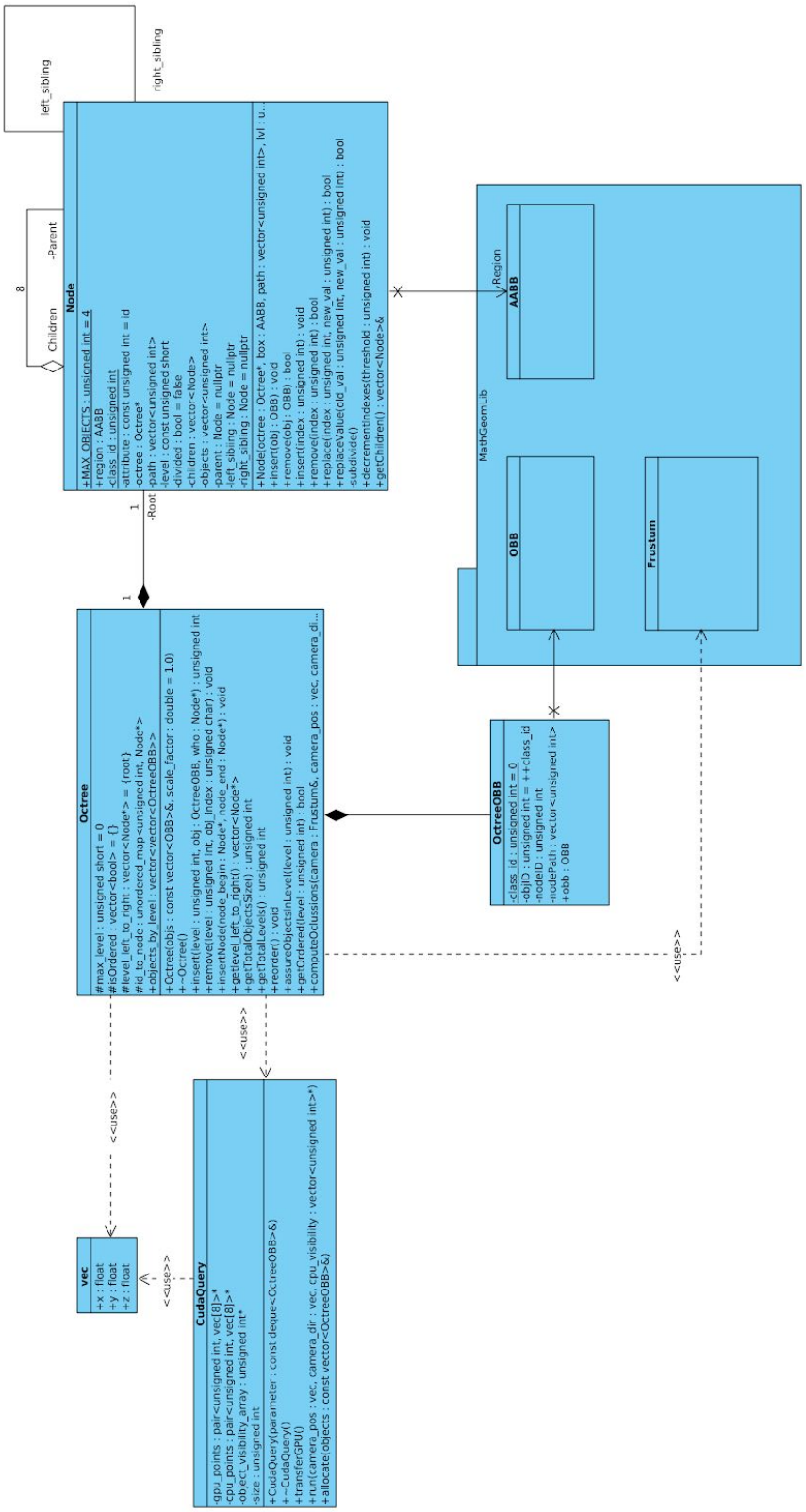


Figura 14 - Diagrama de clases de la librería

Diagramas de Octree

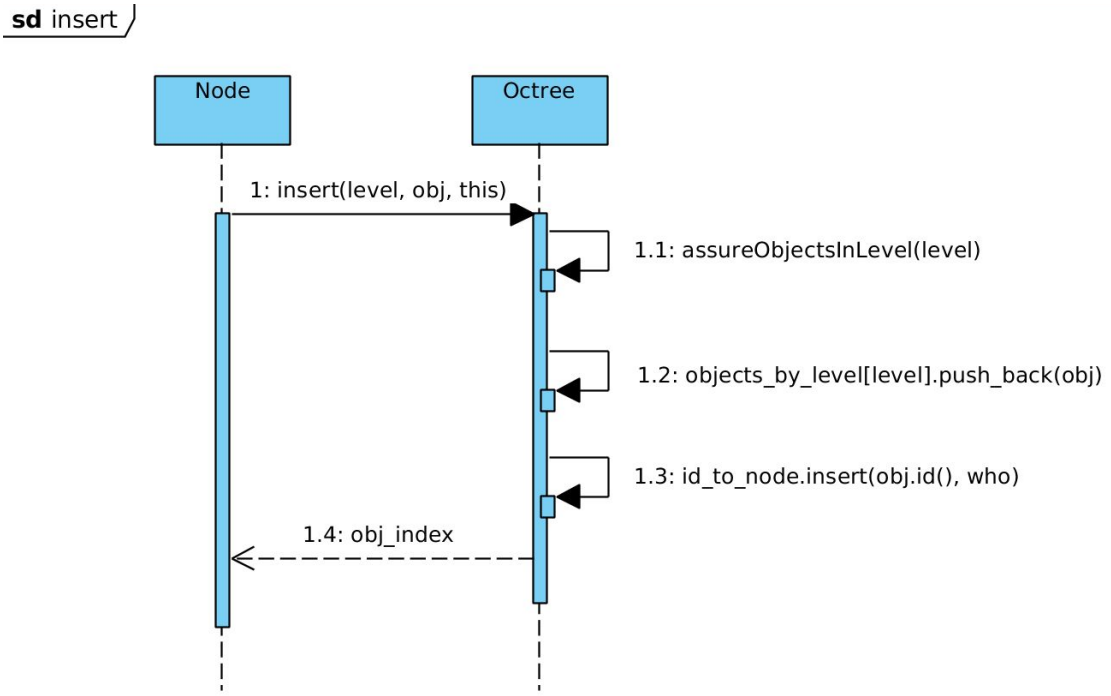


Figura 15 - Diagrama de secuencia del método insert de Octree

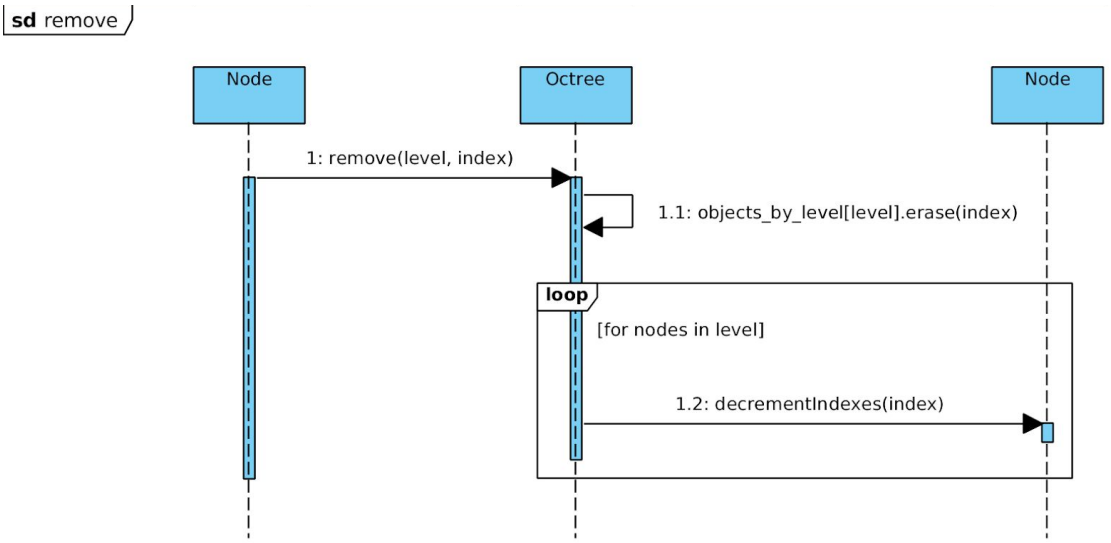


Figura 16 - Diagrama de secuencia del método remove de Octree

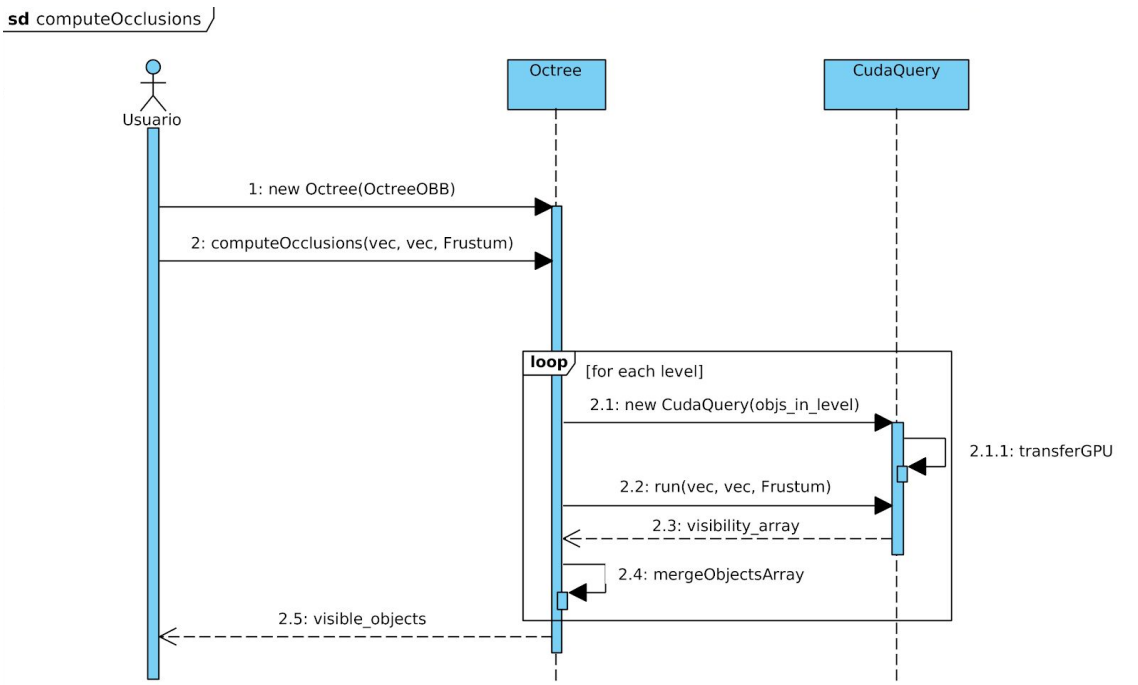


Figura 17 - Diagrama de secuencia del método computeOcclusions de Octree

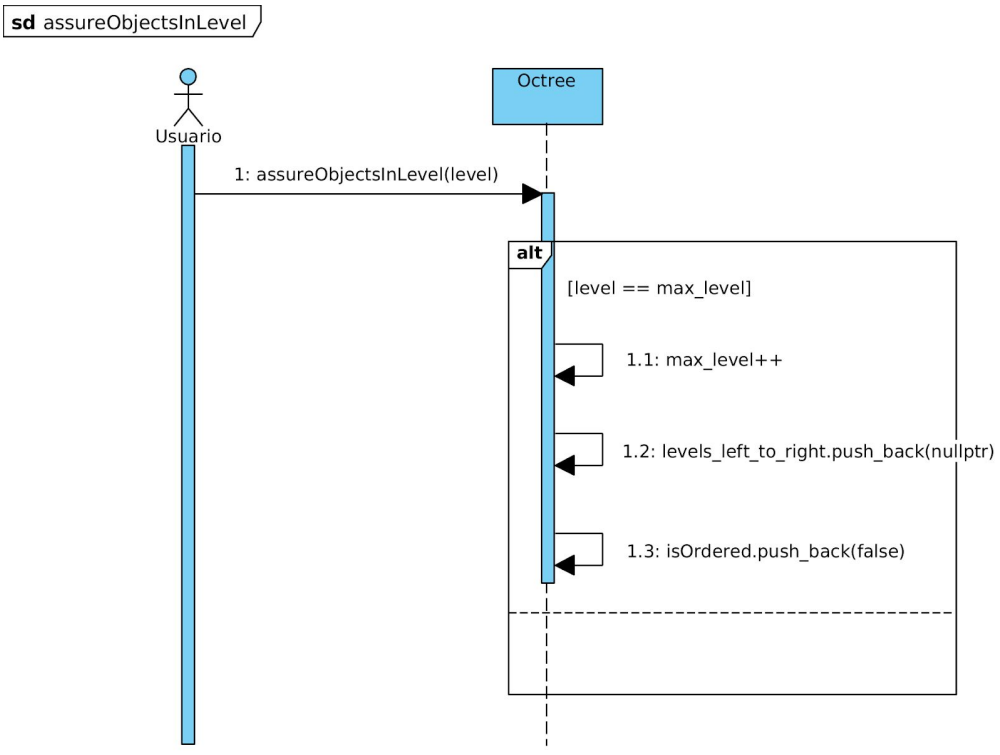


Figura 18 - Diagrama de secuencia del método assureObjectsInLevel de Octree

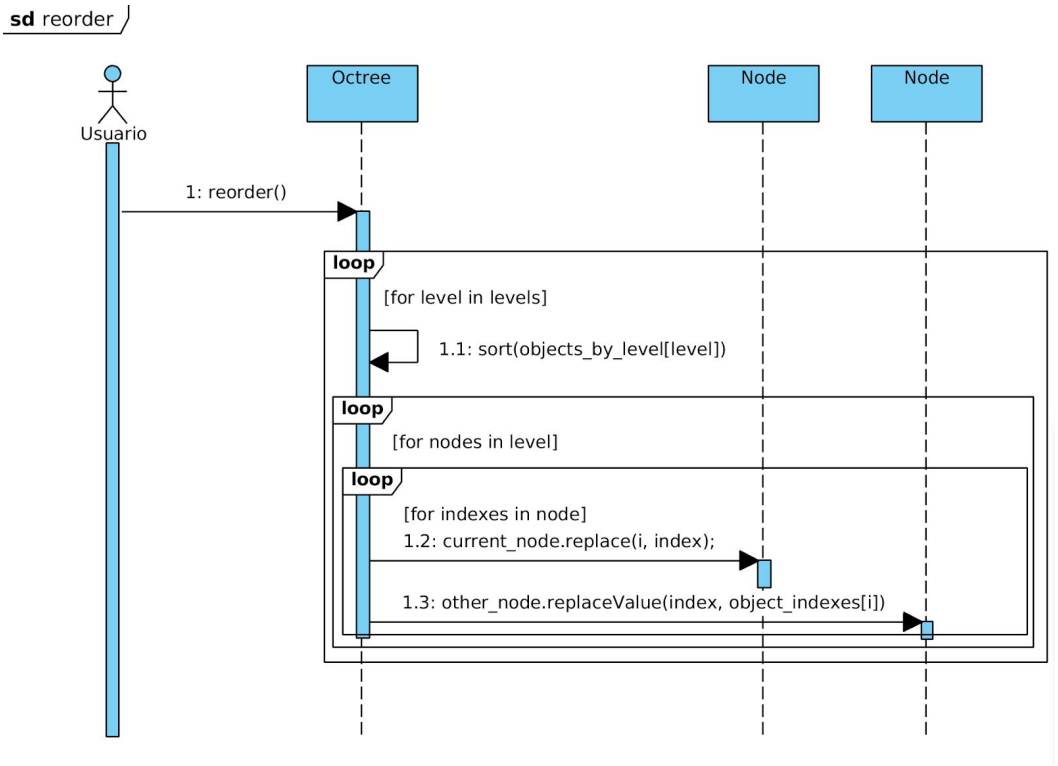


Figura 19 - Diagrama de secuencia del método reorder de Octree

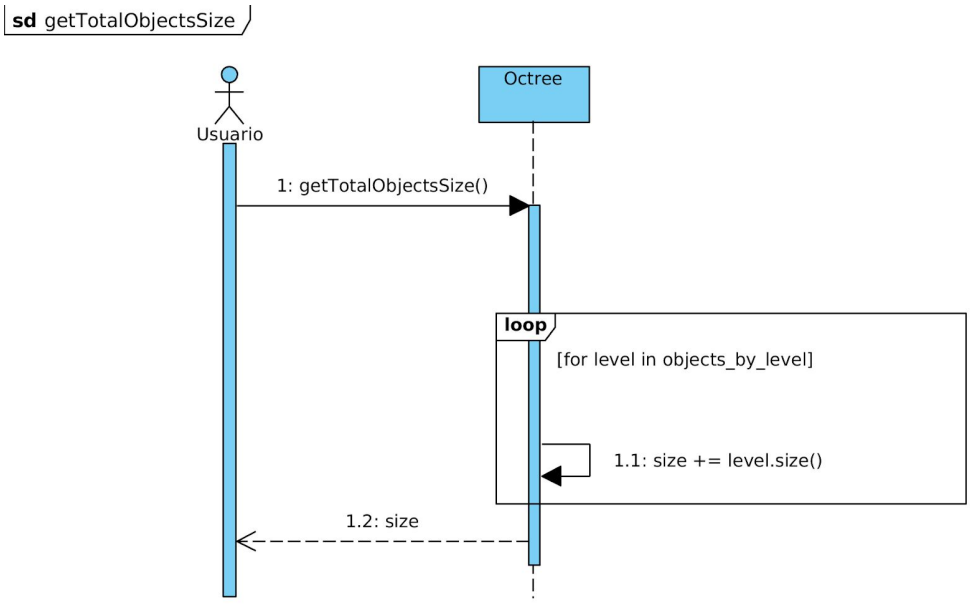


Figura 20 - Diagrama de secuencia del método getTotalObjectsSize de Octree

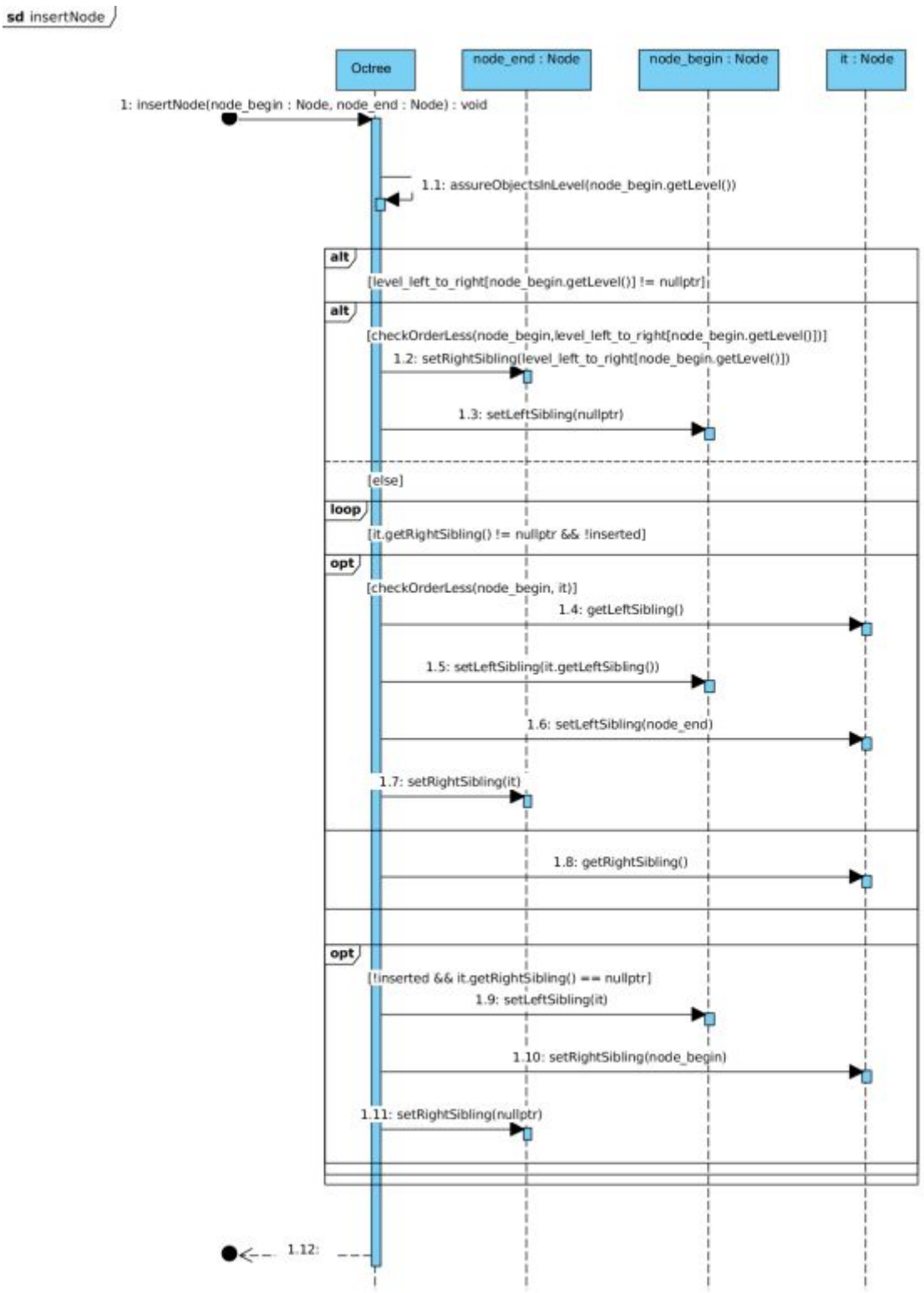


Figura 21 - Diagrama de secuencia del método insertNode de Octree

Diagramas Node

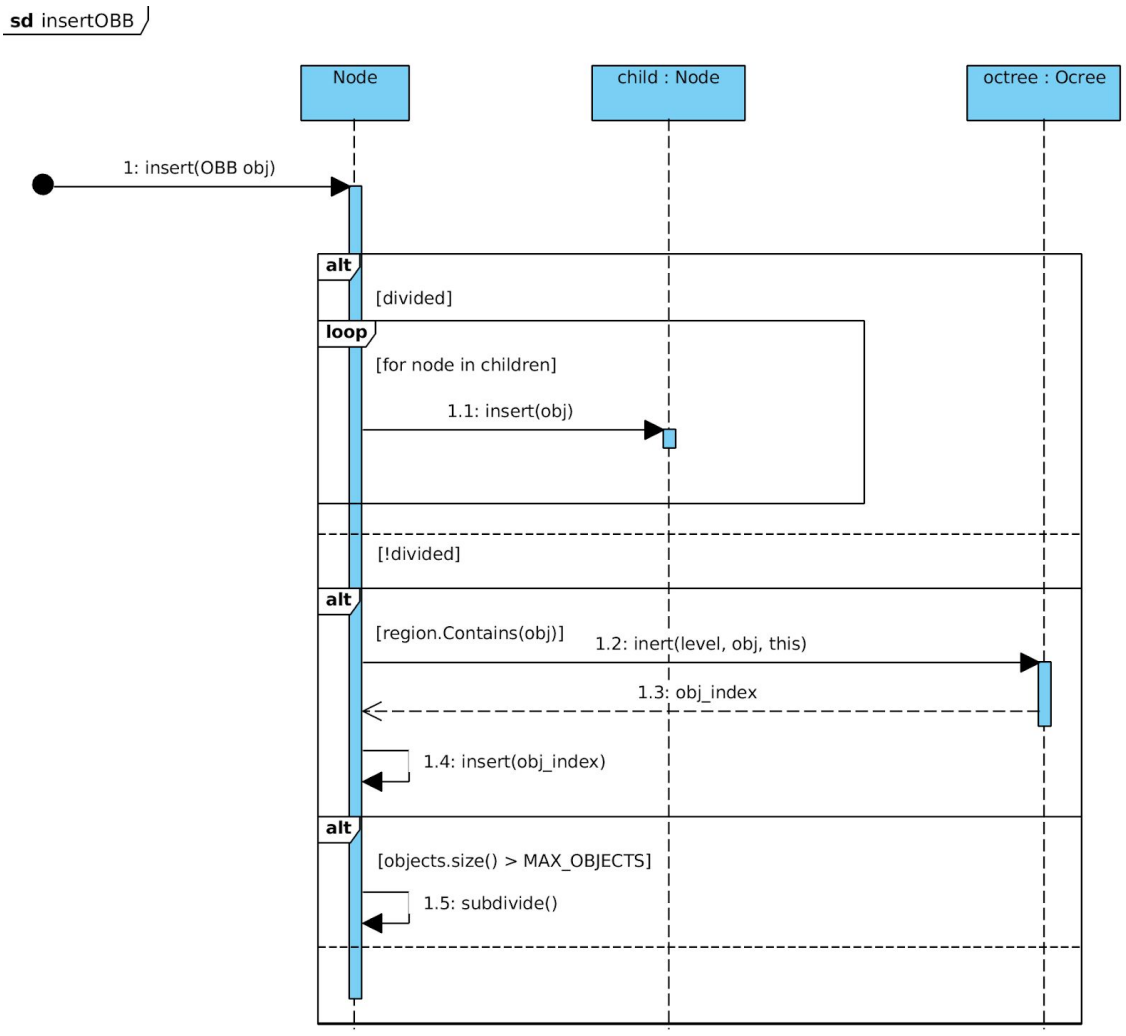


Figura 22 - Diagrama de secuencia del método insertOBB de Node

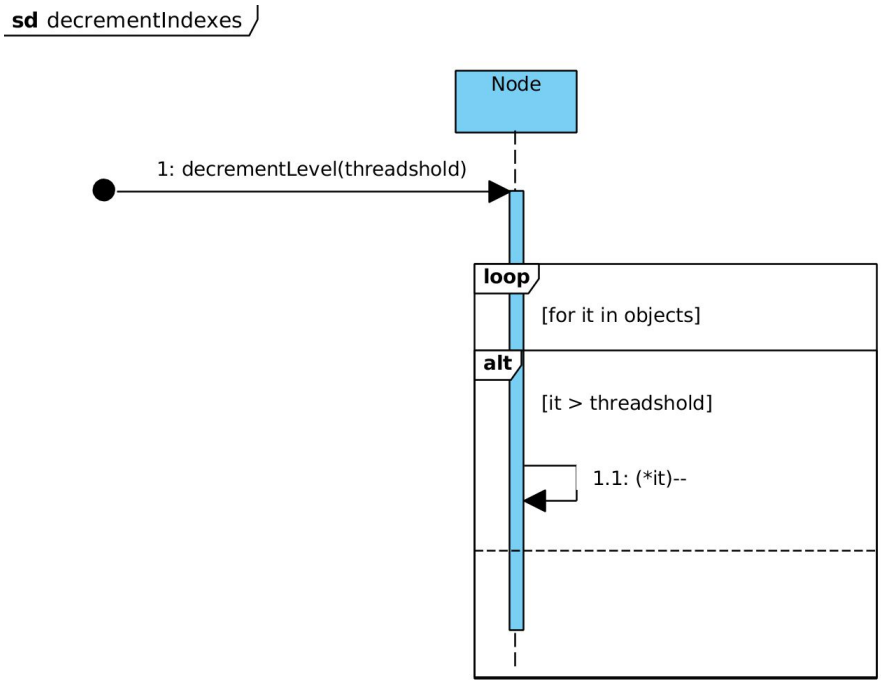


Figura 23 - Diagrama de secuencia del método decrementIndexes de Node

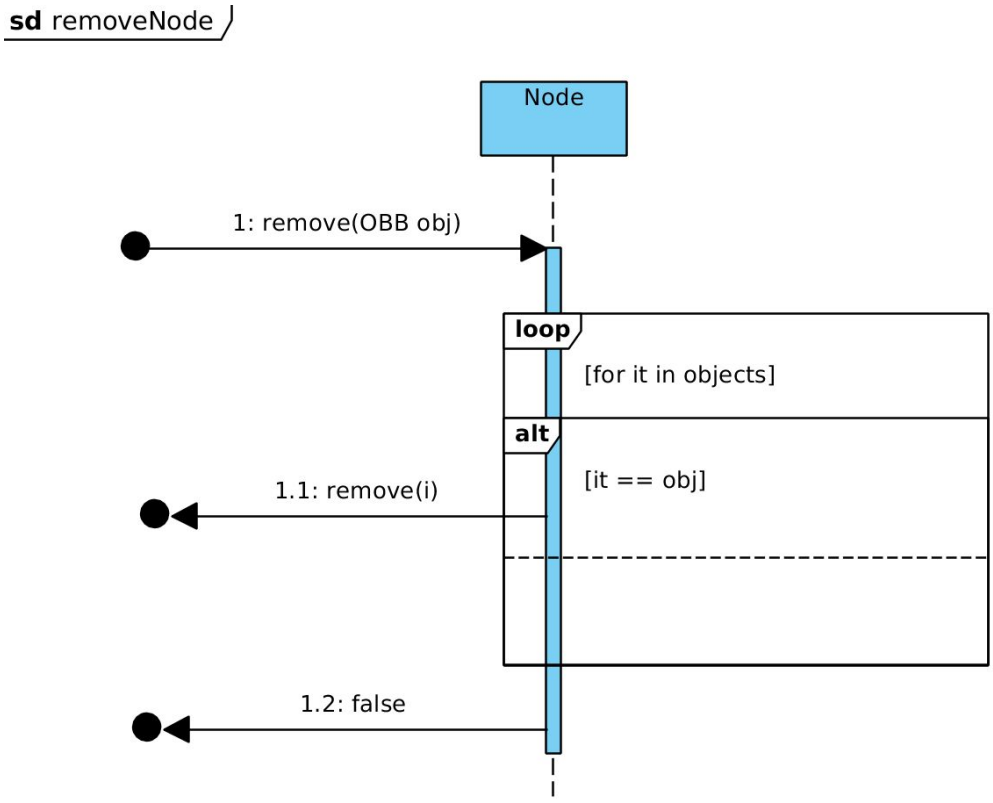


Figura 24 - Diagrama de secuencia del método remove de Node

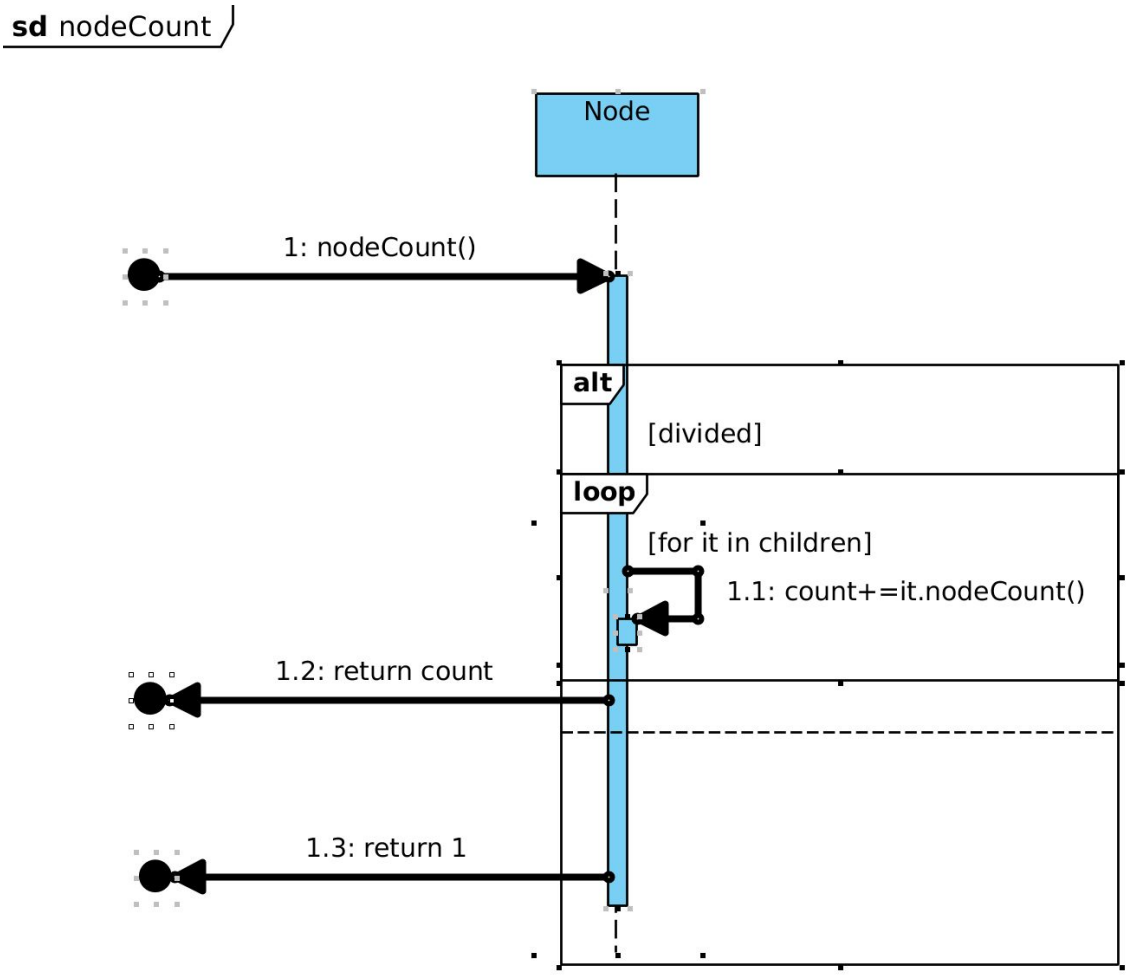


Figura 25 - Diagrama de secuencia del método nodeCount de Node

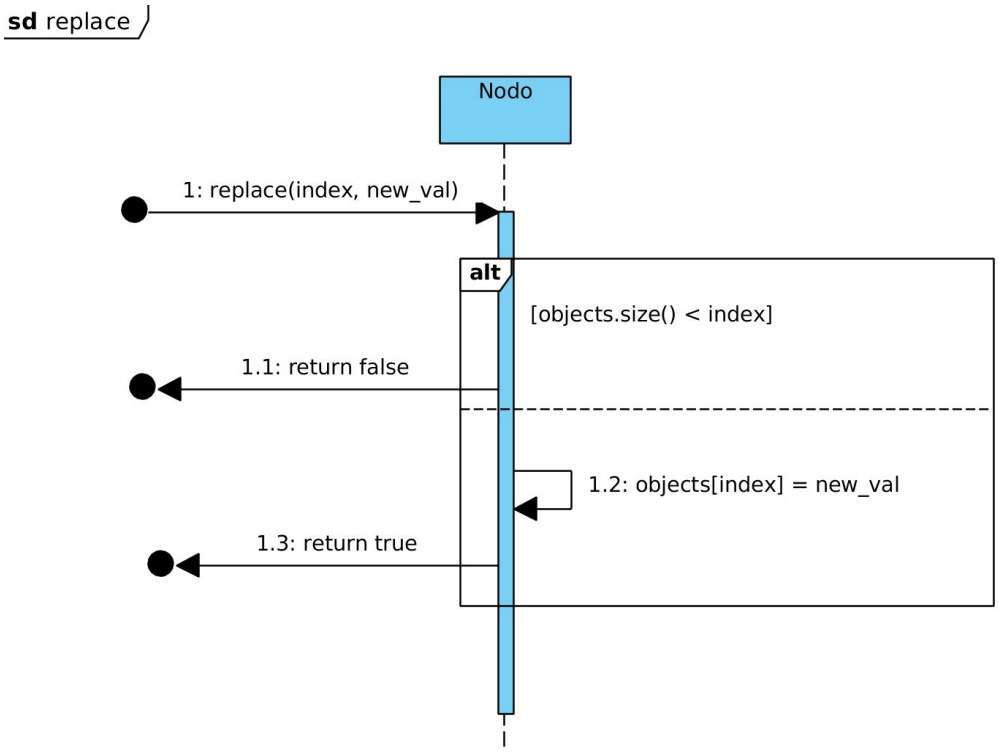


Figura 26 - Diagrama de secuencia del método replace de Node

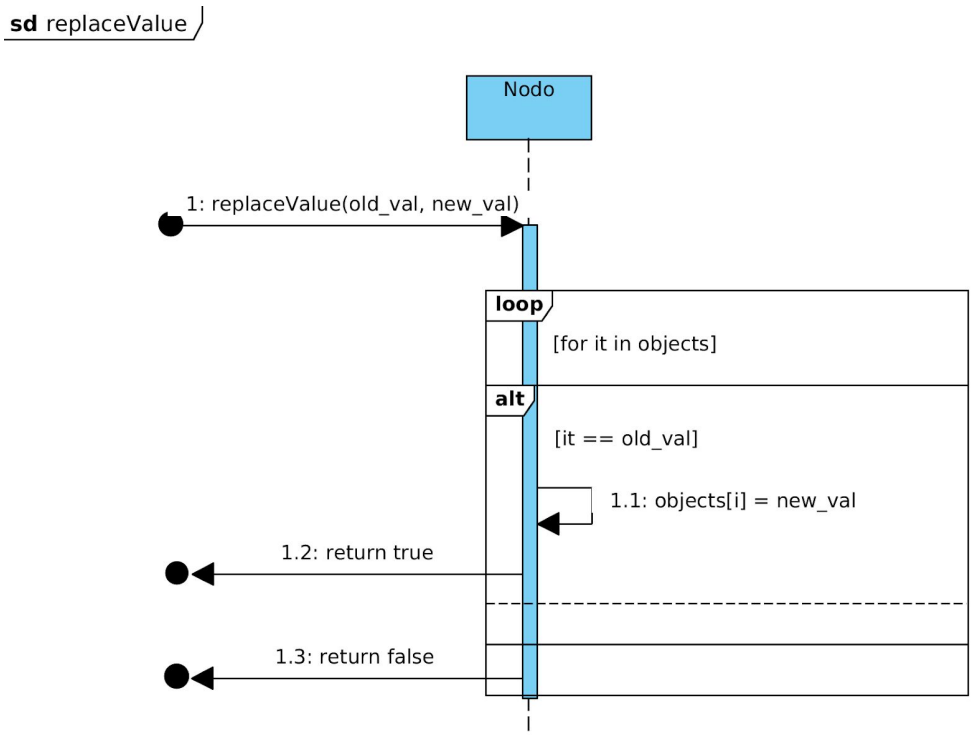


Figura 27 - Diagrama de secuencia del método replaceValue de Node

Análisis del mercado y decisiones tecnológicas

Después de haber estudiado las posibilidades ofertadas tanto como en librerías relacionadas como plataformas para la programación con alto grado de paralelismo entre ellas CUDA y OpenCL se ha decidido hacer uso de CUDA de NVIDIA debido a las siguientes razones:

1. Un mayor grado de afinamiento del código debido a que no debe de correr en las distintas plataformas como pueden ser GPU integradas en CPU, GPU discretas e incluso CPU que sí debe de asegurar OpenCL. Esto hace que CUDA pueda optimizar y definir mejor qué comportamiento queremos que tenga nuestro código, con posibilidad de lanzar hebras a un grado más fino para aprovechar mejor los distintos niveles de caché disponibles, etc.

2. Un ecosistema más consistente y maduro debido a que hay una empresa específica y responsable detrás del mismo, por lo que se dispone de gran cantidad de foros e información en la red al alcance del programador sabiendo que el código que escriba podrá ser ejecutado por todas las GPUs que soporten CUDA siempre y cuando no utilicen algunos aspectos muy específicos que no son soportados por todas las GPUs.

Es claro que el talón de Aquiles de OpenCL es su intento de ser universal haciendo como contrapartida que se pierdan cierto grado de control más fino que resulta ser realmente importante para la resolución de problemas. Queda meridiano que este grado de control es importante ya que de otra forma CUDA no hubiese avanzado y habría quedado desbancado por la alternativa de código abierto. Por estas razones que en este proyecto se hará uso del software de NVIDIA para implementar todo aquello que deba ser ejecutado por la GPU para intentar obtener el máximo rendimiento posible.

Implementación

Plataforma de desarrollo

La plataforma de desarrollo será:

- Linux 4.15.0-51-generic
- Ubuntu 18.04
- CUDA 9.0
- gcc version 6.5.0
- GNU Make 4.1
- NVIDIA drivers 390.116

No se hará uso de ningún tipo de IDE para maximizar la portabilidad del proyecto y tampoco se utilizarán librerías que fuercen a tomar decisiones sobre lo abierto o cerrado que debe de ser el código que las use, es decir **se evitarán librerías con licencia GNU GPL 2.0 y 3.0.**

Librerías usadas:

- **MathGeoLib** https://github.com/juj/MathGeoLib
- **Catch2** https://github.com/catchorg/Catch2

MathGeoLib es una gran librería de código abierto con orientación a objetos que facilita todo lo relacionado con el cómputo de geometría entre distintos tipos de objetos implementados. Incluye todas las clases necesarias para este proyecto AABB, OBB, Frustum y muchas más de una forma muy elegante.
Agradecimientos al usuario juj por crear esta librería y liberar el código con una licencia Apache-2.0, de las más permisivas que existen.

Instalación y configuración

Para poder usar el software solo debemos de tener instalados los correspondientes drivers de NVIDIA para nuestra GPU. En ubuntu el proceso de instalación ha sido altamente simplificado para que todo el mundo pueda instalarlos sin mayor problema. En la siguiente web se nos explica con imágenes paso a paso cómo hacerlo: [Ubuntu Linux Install Nvidia Driver (Latest Proprietary Driver)]

1. Para instalar los drivers solo necesitamos los siguientes comandos.

```
$> sudo apt install nvidia-driver-390
```

2. Debemos de reiniciar para confirmar que todo se ha instalado correctamente.

```
$> sudo reboot
```

Para poder compilar/usar la librería debemos de tener actualizado gcc y make a una versión reciente, para este paso un simple ***sudo apt update -y && sudo apt upgrade*** nos servirá para cumplir estos objetivos.

El paso que nos puede presentar algún problema es el de instalar CUDA ya que debemos de descargar un script hecho por NVIDIA y seguir los pasos que se nos exponen en la terminal. En el siguiente gist de github se nos explica con gran detalle. [Install CUDA 9.0]

- Sólo debemos utilizar los siguientes comandos para instalar CUDA 9.0

```
$> cd
$> wget
https://developer.nvidia.com/compute/cuda/9.0/Prod/local_instal
lers/cuda_9.0.176_384.81_linux-run
$> chmod +x cuda_9.0.176_384.81_linux-run
$> ./cuda_9.0.176_384.81_linux-run --extract=$HOME
```

Desarrollo

Este problema no ha sido solucionado por mí sino que solo se ha aprovechado conocimiento ya estudiado con anterioridad para realizar una variación de la versión tradicional del Octree. El octree es una forma tradicional de particionar el espacio y organizar objetos dentro de dicho espacio de un forma eficiente

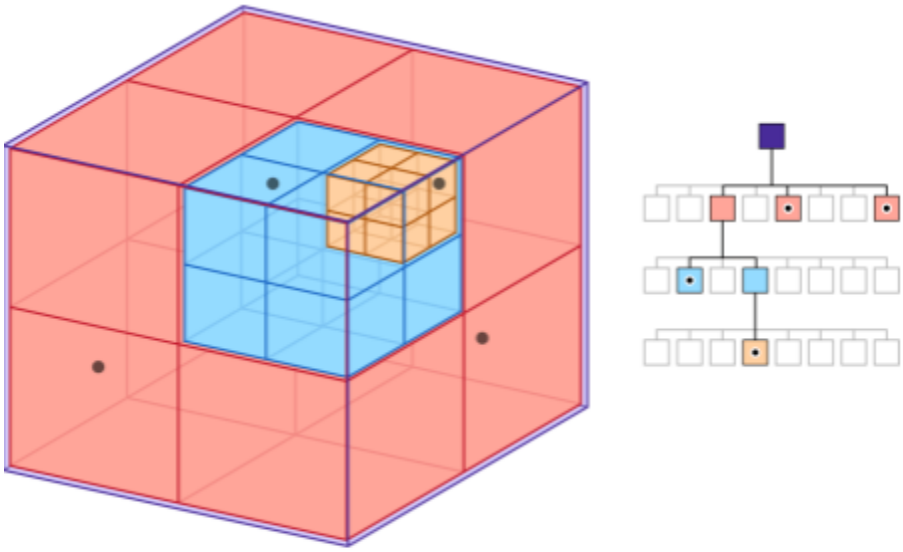


Figura 28 - Octree Space Partitioning (Apple, n.d.)

El Octree funciona de la siguiente forma, dado un espacio y una serie de puntos u objetos y cuando este cubo llega a un límite de objetos se subdivide en 8 sub-cubos y se distribuyen dichos objetos en los hijos del cubo actual. De esta forma se consigue una estructura eficiente a la hora de hacer cómputo sobre dichos objetos en función de su posición ya que se puede hacer una pasada rápida en anchura y en el momento que un cubo no cumple nuestro requisito ya podemos descartar todos los hijos a partir de este.

En nuestro caso se ha realizado una variación del Octree donde los objetos no se almacenan en los nodos del árbol sino que estos tienen un índice a un vector donde todos los objetos se almacenan de forma contigua. Esta idea se aplica también en los árboles b+ de las bases de datos donde todos los datos de los árboles binarios.

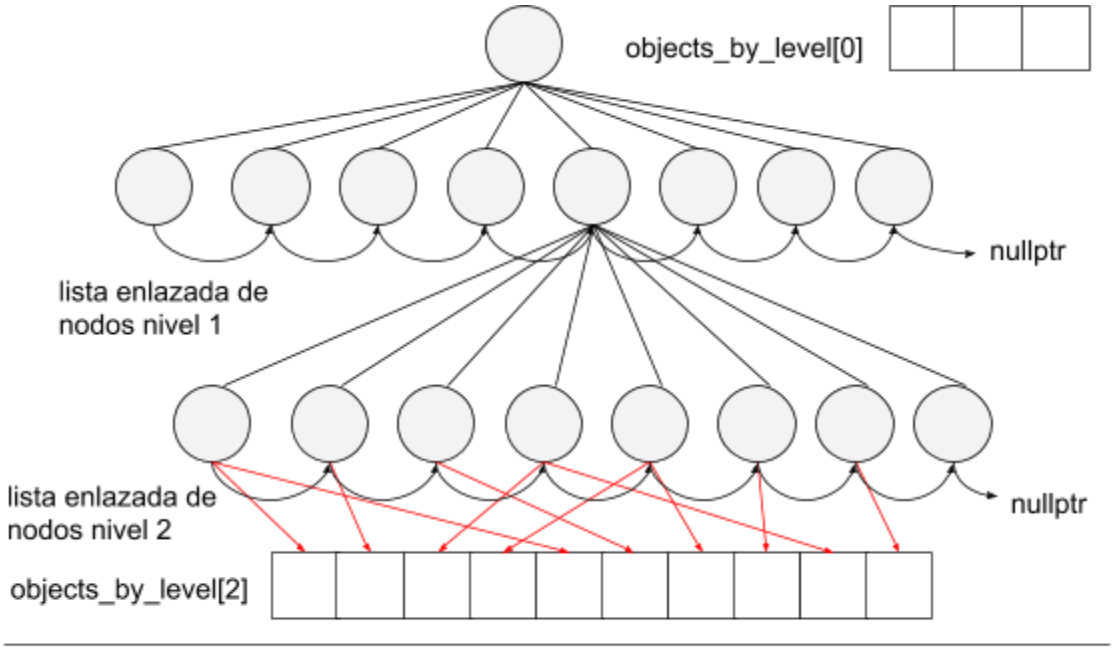


Figura 29 - Diseño del Árbol Octree

De esta forma cuando tengamos que copiar los objetos desde CPU a GPU podremos aprovechar al máximo los distintos niveles de caché de los dispositivos y minimizar la latencia al máximo posible.

Juntando ambas características obtenemos una estructura optimizada para nuestro problema. Podremos ser capaces de hacer una poda rápida de los objetos que no estén ni siquiera dentro del frustum de visión de la cámara y luego realizar la copia de los objetos minimizando la latencia y maximizando el ancho de banda.

Además en la clase Octree se mantiene una lista de punteros al primer nodo de cada nivel para poder iterar los niveles en anchura de forma rápida ya que podemos iterar fácilmente sobre los **right_sibling** de cada uno de los nodos hasta que sea null. Ya que los nodos también forman una lista doblemente enlazada entre ellos.

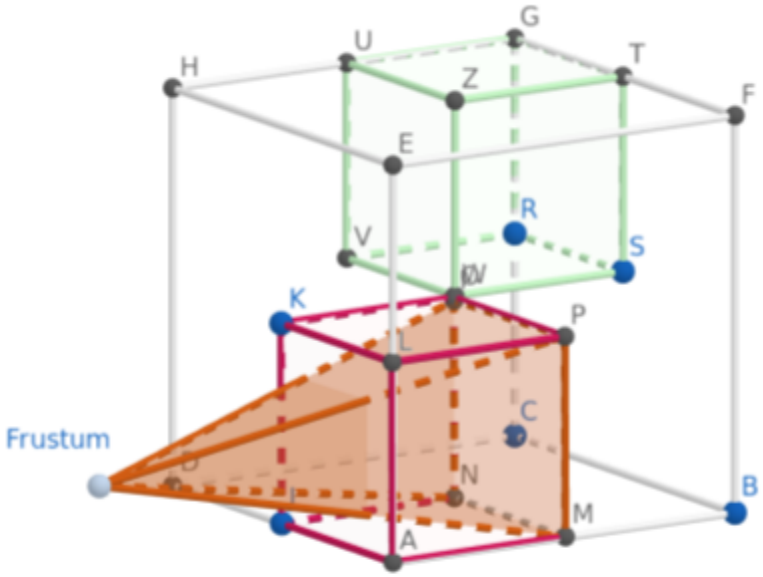


Figura 30 - Poda de objetos por Octree

Esta optimización hace que para comprobar la oclusión no tengamos que comprobar todos los objetos contra todos, es decir $O(n^2)$ sino que dependerá del nº de sub-cubos interseque con el frustum.

Luego mediante la alta paralelización que nos permite la GPU podremos bajar este peor caso de $O(n^2)$ a $O(n)$ donde cada block se encargará de un objeto y se comparará contra los demás para probar si es ocluido o no

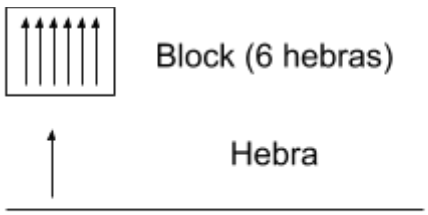


Figura 31 - Parámetros del Kernel

La ejecución del kernel funciona de la siguiente forma:

1. Se lanzan tanto blocks como objetos.
2. Cada block se compone de 6 hebras para comprobar cada una de las caras del cubo de forma paralela.

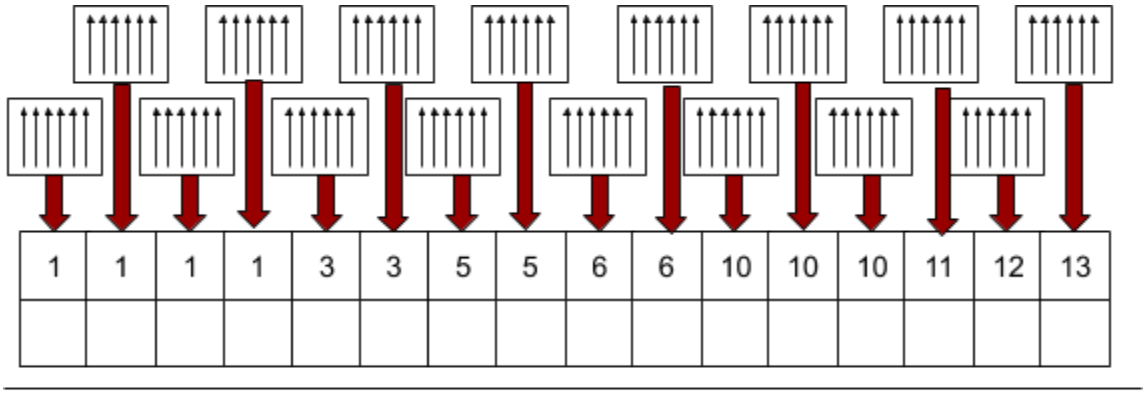


Figura 32 - Ejecución del Kernel

La ejecución de cada uno de los bloques funcionan así:

1. Se va comprobando hacia la derecha mientras su **group_id** sea el mismo que el de su bloque
2. Una vez hemos llegado al final del vector o el **group_id** no coincide se empiezan a comprobar hacia la izquierda
3. Cada vez que se realiza una iteración se actualiza de forma atómica un vector global compartido a todos los bloques

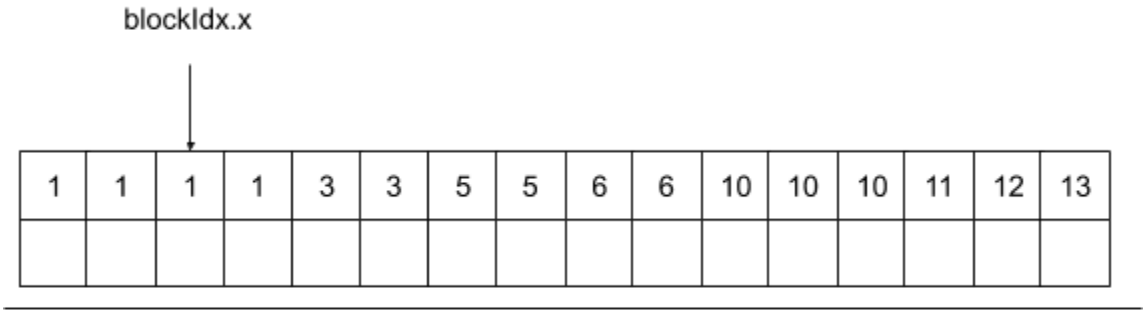


Figura 33 - Algoritmo iteración 0

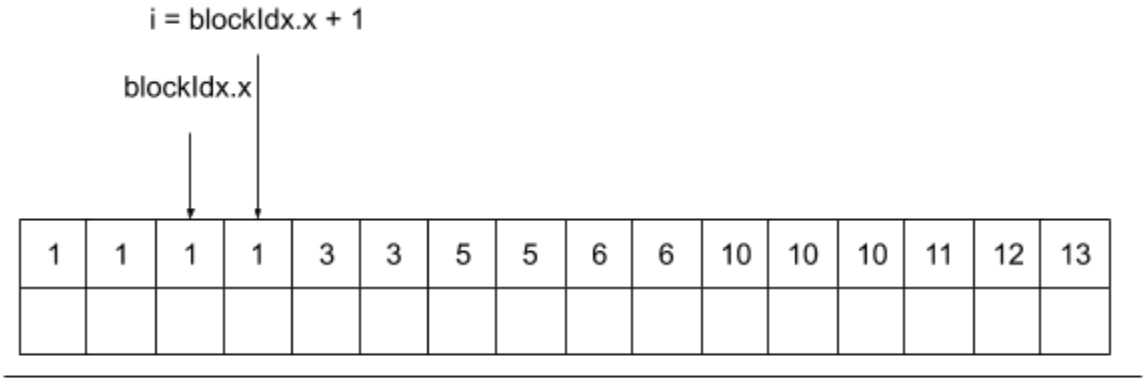


Figura 34 - Algoritmo iteración 1

Como la siguiente iteración el group_id no coincide ($1 \neq 3$) se pasa al bucle que itera en el otro sentido.

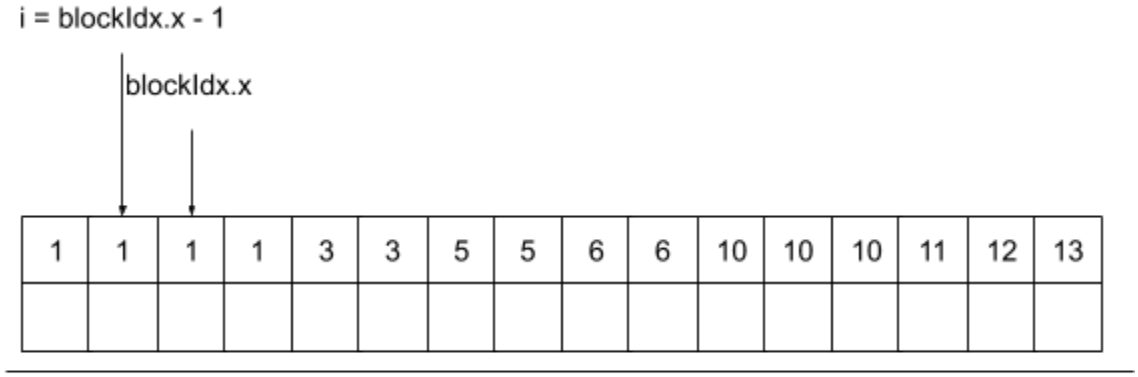


Figura 35 - Algoritmo iteración 2

Figura 36 - Algoritmo iteración 2

Como la siguiente iteración $i < 0$ se acabaría el bucle y ya habríamos de terminado la comprobación de la oclusión todos los objetos.

Pseudocódigo

Pseudocódigo del bucle que itera hacia derecha:

```
if(i < size){
  // Iteración hacia la derecha
  while(occluder.group == occludee.group){
     // Se llama a la función que chequea dos OBB
      checkFaces(camera_pos, camera_dir, occluder, occludee, visibility);

     unsigned int any_visible = 0;
      for (unsigned int p = 0; p < 8 && !any_visible; ++p) {
            any_visible |= visibility[p];
      }

      if(!any_visible){
            // Si ningún punto de occludee es visible
            // el objeto no es visible y no debe ser renderizado
            atomicAnd(&object_visibility_array[i],0);
      }

      // Reinicio del array de visibilidad de los puntos
      visibility[0] = 1;
      visibility[1] = 1;
      visibility[2] = 1;
      visibility[3] = 1;
      visibility[4] = 1;
      visibility[5] = 1;
      visibility[6] = 1;
      visibility[7] = 1;

      if((++i) == size)
         break;
  }
}
```

Solo nos quedaría subir al siguiente nivel del Octree y hacer un merge con los objetos visibles de este nivel con los del nivel superior cambiando el group_id por el del padre de cada uno de los nodos que poseen estos objetos.

Pruebas del software

Las pruebas que nuestro software debe seguir no son muy complejas en este caso las podemos dividir en dos:

1. Octree:
    a. El octree debe insertar correctamente los objetos en el nodo correspondiente
    b. Los nodos deben subdividirse correctamente y pasar sus objetos a sus correspondientes hijos
    c. La lista enlazada de nodos debe estar correctamente en ordenada
    d. Los por nivel deben estar agrupados antes de mandarse a computar a GPU

2. Occlusion Culling:
    a. Se debe calcular correctamente la oclusión de objetos

Para los tests unitarios se va a utilizar Catch2 un framework de unit testing para c++ recomendado por la comunidad por su sencillez y rapidez a la hora de probar software. Además para descargarlo solo debemos de instalar el *single header version* del framework e incluirlo en nuestro proyecto.

De esta forma podremos asegurar que nuestro código siempre funciona aunque se hagan cambios en la implementación ya que en el momento que fallemos los test nos lo dirán. Además este es el proceso habitual en la industria así que nos permitirá familiarizarnos con esta forma de trabajar y facilitar la automatización de compilación de nuevas versiones y la llamada integración continua.

El make realiza la ejecución de los tests automáticamente

```
tests/octree_tests.cpp:95: PASSED:
  REQUIRE( sorted )
with expansion:
  true

tests/octree_tests.cpp:83: PASSED:
  REQUIRE( sorted )
with expansion:
  true

tests/octree_tests.cpp:95: PASSED:
  REQUIRE( sorted )
with expansion:
  true

tests/octree_tests.cpp:83: PASSED:
  REQUIRE( sorted )
with expansion:
  true

tests/octree_tests.cpp:95: PASSED:
  REQUIRE( sorted )
with expansion:
  true

tests/octree_tests.cpp:83: PASSED:
  REQUIRE( sorted )
with expansion:
  true

tests/octree_tests.cpp:95: PASSED:
  REQUIRE( sorted )
with expansion:
  true

tests/octree_tests.cpp:97: PASSED:
  REQUIRE( objs_size == objs.size() )
with expansion:
  1000 (0x3e8) == 1000 (0x3e8)

[CudaQuery] - Computation of 3 objects took 0.051478 ms
-------------------------------------------------------------------------------
Occlusions culling computation
-------------------------------------------------------------------------------
tests/cudaquery_tests.cpp:8
...............................................................................

tests/cudaquery_tests.cpp:43: PASSED:
  REQUIRE( visible_objs.size() == 2 )
with expansion:
  2 == 2

===============================================================================
All tests passed (15 assertions in 3 test cases)
```

Figura 37 - Imagen de ejecución de los tests

Casos de uso y discusión crítica

Esta librería ha sido implementada con el mínimo de dependencias posibles para que sea altamente portable y que cualquiera pueda integrarla en su programa o motor gráfico personalizado. Sólo se necesitará un entorno con soporte de CUDA como ya se ha explicado en la sección anterior.

Gracias a que no se han usado librerías con licencias que fuerzan a liberar el código para permitir que cualquiera pueda usarlo en cualquier situación sin tener que lidiar con los distintos problemas que suponen algunas de ellas. En este caso quien use este código solo debe de dar crédito de que ha usado esta librería y quién es el autor de la misma como las dependencias que esta conlleva.

Esta librería conlleva una licencia MIT que es compatible con todas las demás dependencias. Esto significa que cualquiera puede usarla tanto uso comercial como privado sin tener que negociar una licencia diferente con el autor del código. Siempre se debe dar crédito al autor por su trabajo en el diseño y desarrollo de la librería al igual que se hace en este proyecto con las librerías de terceros.

En la siguiente sección se hará un análisis más crítico de cuando debemos usar la librería y cuando no nos merece la pena. Todo esto mediante un estudio del tiempo de cómputo que supone nuestra librería y los beneficios que puede proponer.

Conclusiones y Trabajo futuro

Ahora revisaremos las gráficas obtenidas después de lanzar una escena de esferas de radio aleatorio entre [1-10) y situadas en el cubo de 100x100x100. En este caso se ha situado una cámara en el (0,0,0) con dirección (1,1,1) para poder captar todos las esferas en la escena.

El nº de objetos lanzados se ha ido variando para poder extraer conclusiones como cuantos objetos es capaz de manejar nuestra estructura de datos y el algoritmo de cómputo de oclusión en paralelo.

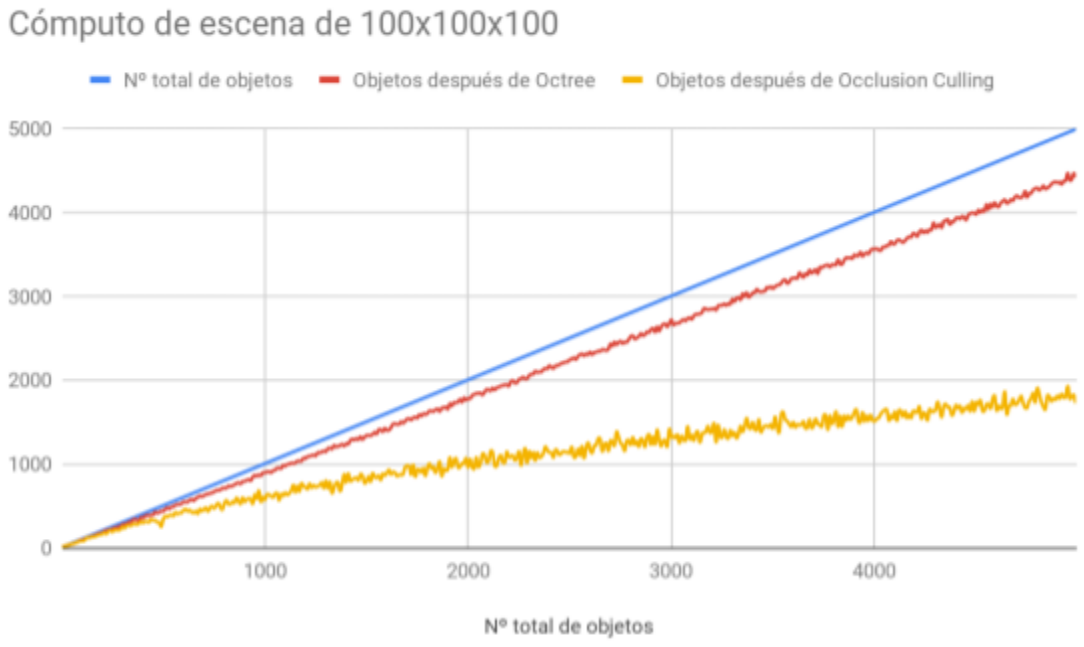


Figura 38 - Gráfica de poda de objetos

En la gráfica podemos ver que cantidad de objetos nos ahorra cada uno de los pasos:

1. La poda del octree nos ahorra computar todos aquellos objetos que ni siquiera estén en el cono de visión de la cámara. Se puede ver que es una poda menor pero muy rápida.
2. El paso que de verdad nos poda objetos es el de Occlusion culling, en el que podemos ver que en nuestra escena hay muchos objetos que no serán visibles desde la perspectiva de la cámara y por tanto una pérdida de tiempo el renderizarlos

Ahora echaremos un vistazo a la gráfica de milisegundos que tarda en completarse nuestra query en función del nº de objetos.

Figura 39 - Gráfica de tiempo de cómputo según nº de objetos

Podemos ver que para una pequeña cantidad de objetos la latencia que ocasiona nuestro algoritmo es mínima pero conforme se va aumentando la cantidad de objetos esta crece mucho más rápidamente probablemente debido a que hay tantos objetos que la GPU no tiene tantas hebras como para poder computarlos todos completamente en paralelo.

Por último veremos una gráfica donde se mostrará el máximo teórico de fotogramas por segundo que se podrá obtener debido a la latencia que este proceso introduce. Es decir que sólo nuestro algoritmo ya limita a esa cantidad incluso suponiendo que el proceso de renderizado fuese instantáneo y no costase nada, de ahí lo de límite teórico.

Figura 40 - Gráfica límite teórico de fps según nº de objetos

Se puede apreciar que para un número de objetos menor que 500 podemos suponer sin mucho error que nuestro software no causará un cuello de botella a la hora de renderizar objetos ya que el máximo teórico es 200 fps lo cual es raramente alcanzado por los programas/juegos 3D normalmente buscando 60 fps y 90 fps en Realidad Virtual.

Sin embargo cuando nos acercamos a los 1000 objetos en escena el límite sería 60 fps por lo que ahí sí empezará a suponer un cuello de botella en el momento de renderizar y en esos casos el programador deberá de decidir si es más eficiente intentar comprobarlos para no renderizarlos o por si el contrario son objetos simples y el hecho de renderizarlos no supone un *major drawback* para ese caso.

Mis recomendación es que no se dude en usarlo cuando tengamos escenas de 500 objetos o menos y que cuando la cantidad de objetos sea mayor a 500 hagamos una prueba de si de verdad merece la pena para nuestro caso específico. Destacar que todas las pruebas han sido ejecutadas en el siguiente entorno:

CPU	i7-7700HQ
Cantidad de RAM	16 GB
GPU	NVIDIA GTX 1050
Sistema Operativo	Ubuntu 18.04
GPU Drivers	NVIDIA drivers 390.116

Figura 41 - Tabla de los datos del equipo donde se han ejecutado los benchmarks

Como líneas de futuro debemos dejar claro que se deben de ampliar la cantidad de problemas que esta librería puede resolver para mejorar de una forma no intrusiva el rendimiento de los programas que se dediquen a renderizar escenas complejas en 3D.

También sería necesario revisar la implementación del código paralelo de CUDA por alguien que sea experto en la materia de forma que se minimice al máximo posible la latencia del código paralelo ya que esa parte es crítica en lo que a rendimiento se refiere. Esto significa que una pequeña mejora puede significar un gran cambio en la latencia introducida por nuestra librería.

Según mi poca experiencia con CUDA tengo entendido que hay formas simples de mejorar la latencia los kernels y aprovechar mejor el ancho de banda con el dispositivo. Un ejemplo sería lanzar los kernels copiando incrementalmente los buffers de memoria en vez de esperar a copiar todo el buffer de objetos y luego lanzar todas las hebras a la vez. Estoy seguro que habrá muchas otras técnicas de minimizar esta latencia y obtener el máximo rendimiento posible.

También se puede mejorar el algoritmo de detección de oclusiones debido a que un único bounding box puede ocasionar falsos positivos con objetos que se alejen a la morfología del modelo, para ello sería mejor utilizar una jerarquía de cajas englobantes. Esta solución haría que nuestro software se adapte mejor a muchos modelos 3D sin importar su morfología y su dimensiones.

Con todo esto hemos construido el inicio de una librería que podría suponer un gran cambio en el panorama actual si la comunidad la acoge y se implementa en el máximo número de programas posibles, es por esto que decido liberar el código con una licencia nada restrictiva como es la Apache 2.0. Esta licencia permite utilizar el código en todos los casos posibles, tanto para uso comercial como uso privado del software sin tener que negociar ninguna licencia especial con el creador. Aunque siempre se debe de dar crédito al autor de que la librería ha usada.

Espero que este proyecto sea útil para el campo y que mejore el área de la computación de gráficos.

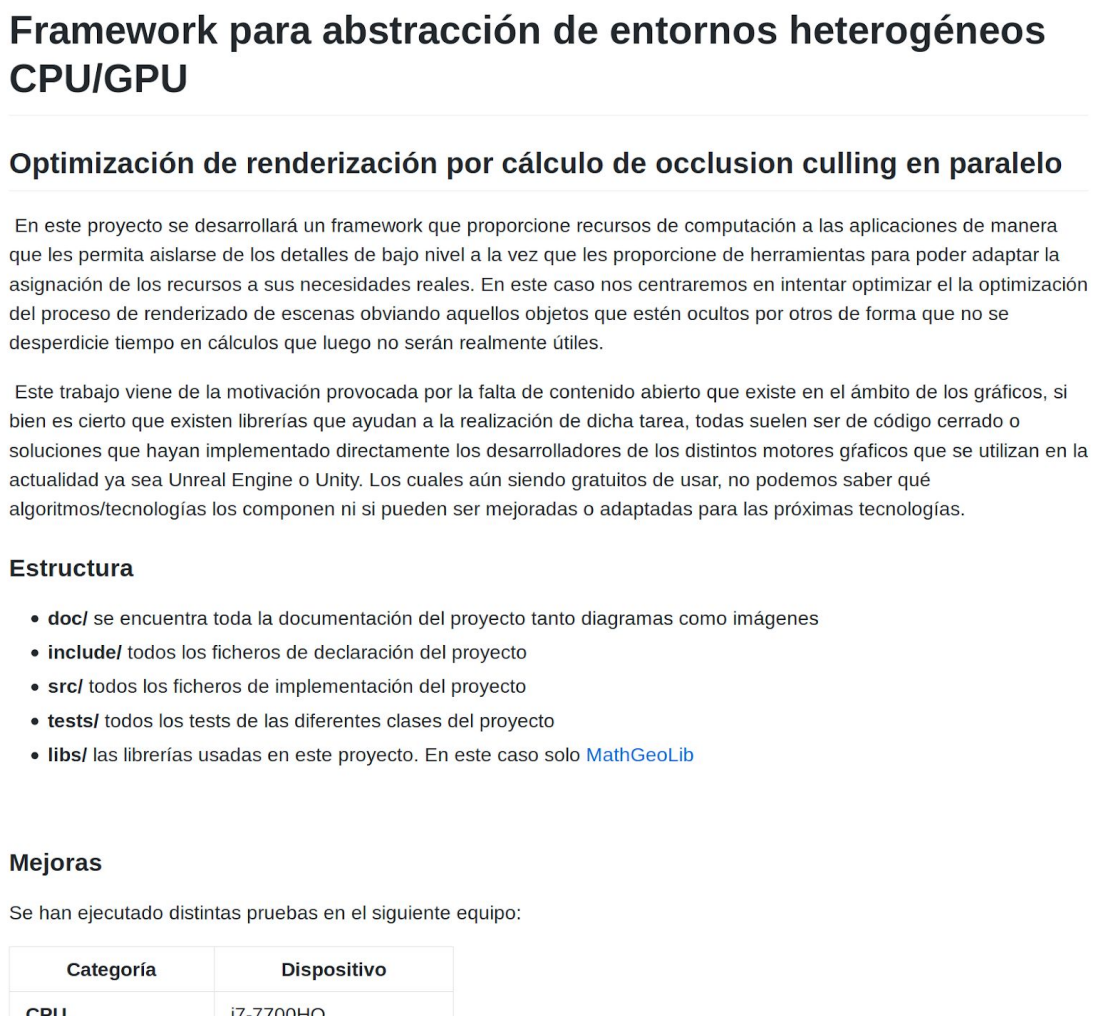
Con esto finalizo mi trabajo de fin de grado.

Juan Carlos Pulido Poveda

Anexos

A1. Instalación de la aplicación Github

Para descargar la librería de Github solo debemos entrar aquí y descargar el zip como se haría normalmente y seguir las instrucciones explicadas en el apartado de instalación y configuración



Framework para abstracción de entornos heterogéneos CPU/GPU

Optimización de renderización por cálculo de occlusion culling en paralelo

En este proyecto se desarrollará un framework que proporcione recursos de computación a las aplicaciones de manera que les permita aislarse de los detalles de bajo nivel a la vez que les proporcione de herramientas para poder adaptar la asignación de los recursos a sus necesidades reales. En este caso nos centraremos en intentar optimizar el la optimización del proceso de renderizado de escenas obviando aquellos objetos que estén ocultos por otros de forma que no se desperdicie tiempo en cálculos que luego no serán realmente útiles.

Este trabajo viene de la motivación provocada por la falta de contenido abierto que existe en el ámbito de los gráficos, si bien es cierto que existen librerías que ayudan a la realización de dicha tarea, todas suelen ser de código cerrado o soluciones que hayan implementado directamente los desarrolladores de los distintos motores gráficos que se utilizan en la actualidad ya sea Unreal Engine o Unity. Los cuales aún siendo gratuitos de usar, no podemos saber qué algoritmos/tecnologías los componen ni si pueden ser mejoradas o adaptadas para las próximas tecnologías.

Estructura

- **doc/** se encuentra toda la documentación del proyecto tanto diagramas como imágenes
- **include/** todos los ficheros de declaración del proyecto
- **src/** todos los ficheros de implementación del proyecto
- **tests/** todos los tests de las diferentes clases del proyecto
- **libs/** las librerías usadas en este proyecto. En este caso solo MathGeoLib

Mejoras

Se han ejecutado distintas pruebas en el siguiente equipo:

Categoría	Dispositivo
CPU	i7-7700HQ

Figura 42 - Imagen del repositorio github

Bibliografía

McClanahan, C. (2010). History and evolution of gpu architecture. *A Survey Paper*, 9.

Arstechnica, J. S. (2000, March 22). SIMD Basics [Digital image]. https://cdn.arstechnica.net/cpu/1q00/simd/figure6.gif

Kessenich, J., Baldwin, D., & Rost, R. (2004). The opengl shading language. *Language version 1*.

Munshi, A. (2009, August). The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)* (pp. 1-314). IEEE.

Fraguela, B. (2016). *fraguela/hpl*. [online] GitHub. Available at: https://github.com/fraguela/hpl

Institut für Technische Informatik (2017). *Heterogeneous Computing*. [image] Available at:https://www.iti.uni-stuttgart.de/fileadmin/rami/files/projects/simtech/heterogen_mapping_overview.png

Profesionalreview (2017). *CPU-GPU*. [image] Available at: https://www.profesionalreview.com/wp-content/uploads/2017/06/En-qu%C3%A9-se-diferencia-la-CPU-de-la-GPU.png

Pradhan, S. (2018). *Check whether a given point lies inside a rectangle or not - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/check-whether-given-point-lies-inside-rectangle-not/

Gregson, J. (2011). [image] Available at: http://1.bp.blogspot.com/-LyIb4cQihWs/TZkQS15uR-I/AAAAAAAAAAU/ZG38yDOqTqA/s200/OBB_triangles.png [Accessed 6 Jun. 2019].

Ericson, C. (2004). *Real-time collision detection*. CRC Press.

Apple (n.d.). *Octree Space Partitioning*. [image] Available at: https://developer.apple.com/documentation/gameplaykit/gkoctree [Accessed 6 Jun. 2019].

GitHub. (2019). catchorg/Catch2. [online] Available at: https://github.com/catchorg/Catch2

Bazhenov, K. (2017). GPU Driven Occlusion Culling in Life is Feudal. [online] Bazhenovc.github.io. Available at: https://bazhenovc.github.io/blog/post/gpu-driven-occlusion-culling-slides-lif/ [Accessed 9 Jun. 2019].

Glassdoor.com. (2019). *Software Engineer Salaries in Barcelona, Spain*. [online] Available at: https://www.glassdoor.com/Salaries/barcelona-junior-software-engineer-salary-SRCH_IL.0,9_IM1015_KO10,34.htm [Accessed 11 Jun. 2019].