CEIG'09, San Sebastián, Sept. 9-11 (2009)
C. Andújar and J. LLuch (Editors)

Diseño de descomposiciones espaciales jerárquicas para mallas de triángulos utilizando Geometry Shaders

J.J. Jiménez, A. Martínez, F.R. Feito

Universidad de Jaén. Grupo de Gráficos y Geomática de Jaén. Departamento de Informática.
Edificio A3. Campus las Lagunillas s/n. E-23071. Jaén. Spain

Abstract

Una de las principales ventajas de utilización de la GPU para la implementación de algoritmos geométricos es su especial diseño y adecuación para la realización de operaciones con vectores, su repertorio de operaciones geométricas y su paralelismo a nivel vectorial y de unidades de ejecución. En cambio, este diseño junto al diseño del pipeline gráfico (adecuado para las operaciones de visualización) no lo es tanto para la implementación de las operaciones geométricas más comunes en informática gráfica, ya que las estructuras de datos y algoritmos diseñados para este fin de manera eficiente y efectiva en la CPU deben ser convertidos y codificados en estructuras de datos (normalmente en la forma de texturas) y en algoritmos que no son intuitivos y que deben ser descompuestos en operaciones atómicas, buscando una paralelización de código que en muchos casos no está clara. En este artículo se pretende dar una serie de directrices para la implementación efectiva de algoritmos para la construcción de descomposiciones espaciales jerárquicas utilizando la GPU programable, aprovechando las ventajas que supone la utilización de shaders programables en las etapas del pipeline gráfico, en especial utilizando geometry shaders. Se propondrán algunas soluciones en las que pueden aplicarse dichos algoritmos en una implementación en GPU mediante shaders, y se aplicará a una descomposición espacial mediante Tetra-Trees.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.5]: Computational Geometry and Object Modeling—Computer Graphics [I.3.6]: Methodology and Techniques—

1. Introducción

Uno de los motivos que nos lleva a utilizar la GPU para problemas de propósito general reside en su capacidad para realizar cálculos en forma de flujo de datos formados por vectores y a su alto grado de paralelismo en las operaciones realizadas (a nivel de unidades de ejecución y de operaciones vectoriales), siendo posible además utilizar conjuntamente la CPU y la GPU para resolver este tipo de problemas. Otros motivos son: el alto ancho de banda en las transferencias de información en la memoria de la tarjeta gráfica; el menor coste de la GPU; la rápida evolución del hardware gráfico; y la potencial paralelización de tarjetas gráficas.

Los datos en la GPU son introducidos en forma de flujo de datos, procesados según los programas cargados en el procesador de vértices, de geometría y de fragmentos y el resultado es escrito normalmente en la memoria gráfica. A partir de la implementación OpenGL 3.0 se soporta la doble precisión en coma flotante, lo que permite cálculos precisos.

Uno de los principales inconvenientes que tiene el realizar una implementación de algoritmos geométricos en la GPU consiste en que hay que adaptar las estructuras de datos y la programación a un modelo de programación paralelizado y enfocado a la visualización, con las restricciones que ello conlleva. Además en algunos casos debe mantenerse un flujo bidireccional de información entre la CPU y la GPU, con la consiguiente penalización en el tiempo de ejecución.

Por otra parte, multitud de algoritmos geométricos están basados en una eficiente descomposición espacial de los modelos. Estas descomposiciones permiten tratar con sólo parte de dichos modelos, reduciendo su complejidad. En cierto modo sería deseable aprovechar el paralelismo inherente en la construcción de las estructuras de datos asociadas, así como en la consulta de las mismas por parte de gran cantidad de elementos de forma simultánea (detección de colisión, operaciones booleanas entre sólidos, etc.). Por todo ello pensamos que la utilización de la GPU para tales

DOI: 10.2312/LocalChapterEvents/CEIG/CEIG09/095-104

propósitos nos brinda, por una parte la oportunidad de paralelizar tales tareas, y por otra, la de aumentar la eficiencia de tales operaciones que son un cuello de botella para muchas aplicaciones.

En este artículo se proponen una serie de directrices que permitirán la construcción y posterior consulta de dichas estructuras de datos utilizando los shaders programables, aplicados a mallas de triángulos, de modo que aumente el rendimiento de las aplicaciones, posibilitando la utilización de la GPU para problemas de propósito general, en este caso particular para descomposiciones espaciales jerárquicas. La literatura al respecto es muy escasa, no existiendo trabajos con un enfoque similar; podemos destacar el trabajo de Zhou et. al [ZHWG08] que realiza la construcción de kd-trees en GPU pero utilizando el lenguaje CUDA. El trabajo aquí presentado es fruto de la experiencia en la programación de shaders. Estos conceptos serán aplicados a la construcción de un tipo especial de descomposición espacial denominado Tetra-Tree [Jim06].

A lo largo de este artículo veremos las características de la programación de shaders que permiten su utilización para problemas de propósito general. A continuación se describirán brevemente los conceptos de descomposición espacial utilizados en este artículo, así como los algoritmos de consulta más comunes para estas estructuras de datos. Posteriormente se estudiará la implementación en GPU de algoritmos de descomposición espacial jerárquicos, así como su aplicación a la construcción de Tetra-Trees. Finalmente, en las conclusiones se resumirán las principales aportaciones.

2. El modelo de programación de propósito general en la GPU

En esta sección llevaremos a cabo un estudio sobre las características de la GPU que pueden ser aprovechadas para la implementación de algoritmos de propósito general, así como el tipo de algoritmos que pueden ser llevados a una implementación efectiva en la GPU. Por último daremos unas pautas generales para una implementación eficiente.

2.1. Recursos computacionales avanzados en la GPU

La mayor parte de los procesadores de vértices sólo son capaces de obtener información del vértice actual del flujo de vértices de entrada, no pudiendo leer información de otros vértices distintos del actual. Las tarjetas Nvidia GeForce a partir de la Serie 6 tienen una nueva característica llamada vertex texture fetch (VTF) que permite el acceso aleatorio a memoria de manera que se puede acceder directamente a textura (permitiendo el acceso a los vértices si estos se han almacenado previamente en una *textura de vértices*).

OpenGL es ampliamente utilizado en aplicaciones gráficas, sus versiones unidas a la implementación de las mismas por los fabricantes de tarjetas gráficas determinan las técnicas y operaciones que se pueden realizar. En concreto para la GPU cada nueva versión dota de mayor potencia y versatilidad al cauce gráfico programable. Las versiones de uso actuales comprenden OpenGL 2.1 y 3.0. Las principales características que aportan al cauce gráfico son: nueva versión del *OpenGL Shading Languaje* (GLSL), comandos para definir matrices no cuadradas (aspecto muy importante dentro de la programación de propósito general con la GPU, ya que la mayoría de texturas que representan matrices en la CPU no son cuadradas), el *pixel buffer* y texturas sRGB. En el caso de OpenGL 3.0 podemos destacar: OpenGL Shading Languaje 1.3 (GLSL), *vertex array objects*, *Framebuffer Objects (*FB-Objects*)* [fVIT08] (extensión orientada a GP-GPU para poder realimentar el cauce gráfico de forma intuitiva sin salir de la CPU, lo que proporciona mayor flexibilidad), texturas y buffers con precisión de coma flotante de 32 bits (muy útil en operaciones geométricas precisas) y *texture arrays*.

Las últimas especificaciones de OpenGL incluyen una nueva etapa programable en GPU denominada *geometry shader*. Esta etapa tiene como entrada los vértices obtenidos del procesador de vértices y su relación geométrica con el resto [Yon06]. En este caso la entrada no son vértices sino entidades geométricas (triángulos, líneas, conjuntos de triángulos, etc.). En esta etapa podemos modificar las posiciones y atributos de los vértices teniendo en cuenta su relación geométrica. Un aspecto muy importante que dota de mayor potencia a esta etapa es la capacidad de generar nueva geometría o eliminar la que ya existe. Es la única etapa en la que la entrada no siempre genera una salida de la misma naturaleza. Las operaciones implicadas pueden duplicar geometría (escenarios de mallas idénticas), seleccionar geometría con respecto a criterios geométricos (nivel de detalle, composición, etc.) y también modificar características individuales de los vértices [AL06].

2.2. Características de los algoritmos susceptibles de programación en la GPU

Para poder utilizar la GPU en una determinada implementación es necesario que los datos de entrada sean paralelizables, y que se les aplique los mismos cálculos de manera repetida y de forma independiente entre sí [Fer05]. No todos los problemas son susceptibles de ser resueltos de esta forma, por lo que no todos los algoritmos pueden implementase en la GPU de manera eficiente. La clave está en el paralelismo y la independencia, es decir, no sólo se deben realizar los mismos cálculos a un flujo de elementos, sino que los cálculos de cada elemento deben tener una dependencia pequeña o ninguna con otros elementos.

Los datos de entrada a los algoritmos residentes en el procesador gráfico deben ser adaptados a un conjunto de tipos de datos disponibles. Esto, junto a que los lenguajes de programación actuales de las tarjetas gráficas [Ros06] limitan el conjunto de estructuras de datos a utilizar, reduce en cierto modo el tipo de algoritmos que pueden adaptarse al modelo de programación en la GPU.

2.3. Shaders vs. lenguajes para GP-GPU

Si bien es cierto que en la actualidad existen extensiones de lenguajes de programación orientadas a la programación de propósito general en la GPU (CUDA. AMD-Stream, OpenCL), hay problemas que por su naturaleza ofrecen un enfoque más eficiente en el modelo tradicional de etapas de la GPU. Estos problemas están asociados a operaciones geométricas muy relacionadas con la interacción y visualización de mallas. En estos problemas las variables de entrada suelen tener un significado geométrico o gráfico (puntos, vértices, triángulos, mallas, curvas, etc.) y la operativa hace hincapié en funciones muy relacionadas con la visualización. Para estos problemas es necesario que existan etapas de visualización habituales y su programación se realiza en CPU usando lenguajes de procesamiento gráfico como OpenGL. Al convertir estos problemas a un enfoque basado en GPU no podemos prescindir de las operaciones gráficas. Por ejemplo, un problema de calculo de soluciones a un sistema de ecuaciones diferenciales sería adecuado convertirlo a un problema en GPU usando CUDA [CUD08], pero problemas como el de generación de escenarios con mallas replicadas permiten un enfoque eficiente usando shaders.

2.4. Adaptación de algoritmos a la GPU

Todas las ventajas de uso que hemos visto vienen limitadas actualmente por una serie de inconvenientes que nos llevan a transformar ciertos problemas o algoritmos que están implementados de una forma más natural en la CPU, a una forma en la que puedan usarse de acuerdo a las características tan particulares de la GPU [GÖ5]:

- *Para almacenar datos en la GPU se utilizan texturas, tal y como se hace de forma natural en los arrays en la CPU*: Se puede utilizar una textura o un array de vértices.
- *Los shaders en la GPU simulan los bucles internos en la CPU*: En la CPU se puede utilizar un bucle para iterar operaciones sobre elementos almacenados en un array. En la GPU podemos utilizar operaciones similares en un programa para un shader determinado que se aplicará a todos los elementos de un flujo de elementos de entrada.
- *Escribir en el frame-buffer o en una textura es el modo de obtener realimentación*: Si deseamos que los datos de salida sirvan de entrada para nuevos cálculos debemos escribir estos datos en la memoria gráfica. Para poder escribir en una determinada posición sólo los programas de vértice o de geometría pueden indicar o modificar el fragmento que entrará al procesador de fragmentos.
- *La rasterización de la geometría provoca los cálculos en la GPU*: Para poder utilizar los procesadores programables de la GPU es necesario dibujar, de manera que se llame al programa de vértices con un flujo de vértices y/o al programa de geometría con un flujo de triángulos y se genere finalmente un flujo de fragmentos que pase al programa de fragmentos. Estas activaciones en muchos casos no tendrán nada que ver con geometría ni visualización.
- *Los vértices y/o triángulos son el dominio de entrada de una aplicación*: La forma de suministrar información dinámica a los shaders se realiza mediante el envío de vértices para el caso del *vertex shader* y de triángulos para el caso del *geometry shader*. La información de entrada se suministrará codificada por medio de vértices y propiedades de vértices. La información estática vendrá dada por la información almacenada en texturas.
- *Las coordenadas de vértices controlan el rango de salida de una aplicación*: Como los programas de vértices o de geometría no son capaces de modificar las coordenadas del píxel sobre el que escribir un fragmento, los programas de vértices y de geometría, junto a los vértices de entrada, determinan los pixels que se generarán.

Para aumentar la eficiencia de un programa es necesario la presencia de localidad en las referencias a los datos de memoria, es decir, que los datos referenciados en un breve periodo de tiempo se encuentren en posiciones cercanas.

3. Descomposición espacial jerárquica

En esta sección se describen algunos conceptos asociados a la creación de descomposiciones espaciales jerárquicas, así como algunos algoritmos genéricos que hacen uso de ellas.

3.1. Descomposiciones espaciales

La mayor parte de los algoritmos de descomposición espacial (octrees [Swa93], bsp-trees [NAT90], kd-trees [KHSZ98], etc.) realizan una descomposición jerárquica y adaptativa del espacio que ocupan los objetos, normalmente bajo una estructura de datos en forma de árbol. En cada nodo del árbol generado se puede almacenar información sobre la porción del espacio que representa, un estado de ocupación, información sobre los nodos hijos, y posiblemente otra información adicional sobre la geometría clasificada en dicho nodo (nodos hoja) o incluso a nivel de nodos intermedios.

Utilizando estructuras de datos jerárquicas de este modo, se pueden realizar de forma sencilla ciertas consultas como p.e., dado un punto, obtener en qué descomposición de último nivel se encuentra, o en operaciones booleanas entre sólidos localizar rápidamente la región de interés.

3.2. Algoritmos de construcción y consulta

En general, la construcción de estructuras de datos asociadas a descomposiciones espaciales está basada en la construcción de una jerarquía, que en este artículo vamos a considerar en forma de árbol. Para ello partiremos de un nodo raíz, cuyo volumen asociado contendrá toda la geometría de una malla de triángulos. Éste se dividirá en n_1 nodos hijos, de modo que la geometría del objeto se descomponga en n_1 subconjuntos no necesariamente disjuntos pero que contengan la totalidad de la geometría clasificada en el nodo padre. Cada uno de los nodos resultantes se dividirá en n_i

nodos hijos del mismo modo, hasta alcanzar un criterio de parada (como una profundidad máxima en la jerarquía, o un número mínimo de geometría clasificada en el nodo). Los nodos de último nivel se llamarán nodos hoja. Para reducir la información contenida en la jerarquía suele adoptarse el criterio de sólo almacenar la geometría en los nodos hoja.

Por otra parte, los algoritmos de consulta más habituales para este tipo de estructuras de datos radican en obtener la geometría contenida en un determinado nodo, así como obtener el nodo de máxima profundidad en la jerarquía dada una posición en el espacio. Otras consultas pueden ser obtener los nodos padre o hijos, así como los nodos adyacentes o más cercanos. Fuera del ámbito de consulta también son importantes los algoritmos de actualización de las estructuras de datos asociadas cuando se modifica la geometría de una malla, como ocurre en las deformaciones.

4. Implementación en GPU de algoritmos de descomposición espacial jerárquicos

En el caso que nos ocupa, es conveniente que por un lado el procesamiento realizado en la GPU sea eficiente, y por otro que la representación o codificación de la jerarquía obtenida facilite las operaciones de consulta posteriores.

En esta sección veremos cómo implementar la construcción de una jerarquía según hemos visto en el apartado anterior, de forma similar a como se implementan tradicionalmente dichas jerarquías en CPU, es decir, construyendo una estructura de datos en forma de árbol en cuyos nodos se almacena cierta información [OGL05]. A continuación, debido a las limitaciones que nos impone la arquitectura y el modelo de programación de shaders en GPU, propondremos una implementación radicalmente distinta a este modelo tradicional. Finalmente aplicaremos los conceptos desarrollados a la implementación de Tetra-Trees [Jim06].

4.1. Implementación basada en la codificación de jerarquías de modo tradicional

Antes de definir lo que entendemos por jerarquía tradicional, debemos hacer hincapié en que la mayor parte de los algoritmos implementados en GPU siguen un ciclo consistente en realimentar la información resultado a través de la CPU. Por ejemplo, en la detección de colisión entre una nube de partículas y una malla de triángulos [JOSF06], tanto la clasificación de la malla usando una determinada descomposición espacial, como la clasificación en dicha descomposición espacial y la detección de colisión de una partícula con la parte de la malla correspondiente pueden realizarse en la GPU. En cambio, una vez determinada la colisión de la partícula, esta información es llevada a la CPU que se encarga de determinar acciones posteriores con la malla y/o con la partícula. Además la visualización queda totalmente desligada de los cálculos.

Podemos considerar que una jerarquía tradicional está formada por nodos y relaciones entre nodos (padres e hijos). En cada nodo se almacena la información, en caso necesario, del volumen asociado al nodo y de la geometría contenida en el nodo. Siguiendo este modelo es necesario obtener una codificación de la información contenida en la jerarquía, es decir, codificar la información referente a los distintos nodos de un árbol (geometría del volumen que representa el nodo, geometría de la malla clasificada en el nodo, e información de enlace a los nodos hijos). Esta codificación debe realizarse en texturas de manera que se facilite el acceso a la información, manteniendo la localidad espacial en lo posible, tratando de evitar duplicidades, y posibilitando posteriores modificaciones si fuese posible.

4.1.1. Alternativas de codificación de los datos

Una posible codificación para la jerarquía asociada a una malla de triángulos puede venir dada por una textura que almacene los vértices de la malla (*textura de vértices*) y otra textura (en este caso 2D) que almacene los triángulos clasificados en cada nodo (*textura de triángulos*). Para aumentar la localidad espacial en el acceso a los datos, se puede representar en cada fila de la *textura de triángulos* 2D la información de un nodo del árbol, almacenando el conjunto de triángulos clasificados en dicho nodo. Cada columna puede contener información de un triángulo, es decir, 3 índices a los vértices representados en la *textura de vértices*, junto con el número total de triángulos clasificados en dicho nodo y la información de geometría del volumen asociado. La información de la localización de los nodos hijos puede venir dada mediante una operación aritmética. El caso de los nodos hoja puede ser detectado fácilmente o usar un flag (Figura 1).

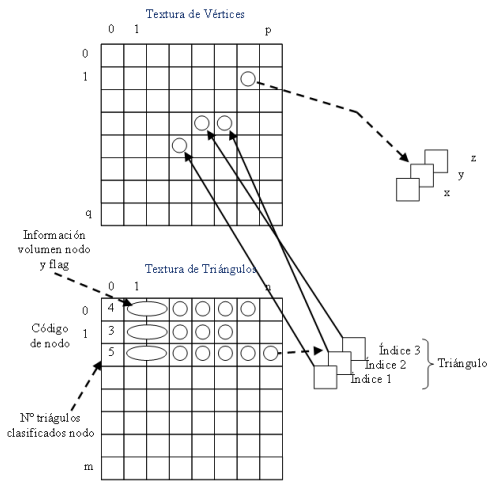

Figura 1: *Representación de la geometría de una malla clasificada en una descomposición espacial utilizando texturas de vértices y de triángulos.*

En la codificación anterior cada nodo intermedio puede

contener información sobre los triángulos clasificados en dicho nodo, si bien en este caso en los primeros niveles sería necesario almacenar gran cantidad de triángulos, número que irá descendiendo conforme la descomposición realizada sea mayor, es decir, conforme aumentemos la profundidad en el árbol. Esto conllevaría que el número de columnas de la *textura de triángulos* fuera tan grande como el número de triángulos de la malla, para poder almacenar en el nodo raíz todos los triángulos, número que ira decreciendo conforme se profundice en el árbol, pero que sería desaprovechado en gran medida a mayores profundidades. En el caso de no almacenar información de la geometría clasificada en los nodos intermedios también sería necesario definir una textura con tantas filas como nodos definidos en la jerarquía, aunque muchos de ellos quedaran "vacíos", disminuyendo así el número de columnas de la *textura de triángulos*.

Una alternativa en la que se reduciría el número de filas de la *textura de triángulos* consistirá en utilizar una cabecera al principio de cada nivel, indicando cuál sería la posición de los siguientes niveles (Figura 2).

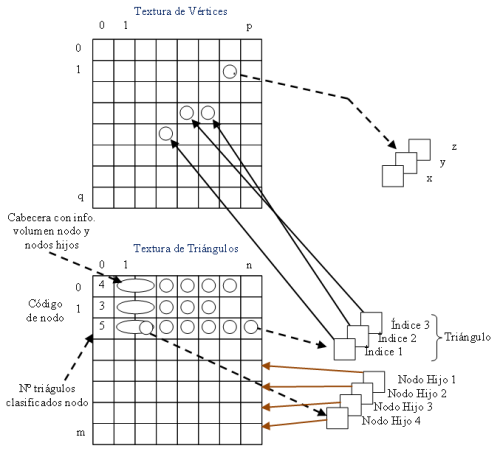

Figura 2: *Representación de la geometría de una malla clasificada en una descomposición espacial utilizando texturas de vértices y de triángulos con infor. de nodos hijos.*

En cambio, podemos utilizar una fila para cada nodo de último nivel, reduciendo así también el número de columnas de la *textura de triángulos* número igual al máximo número de triángulos clasificados en un nodo hoja. Este número podría limitarse a priori añadiendo una cantidad al número teórico de triángulos por nivel. Esto implicaría que sería necesario profundizar todos los nodos a un nivel máximo, obteniendo siempre la descomposición máxima, lo que sería equivalente en el caso de un octree a obtener una rejilla regular, perdiendo las ventajas de realizar una descomposición espacial adaptativa.

Una posible solución radicaría en almacenar en una tercera textura la información de la jerarquía (*textura de jerarquía*), aparte de la *textura de triángulos* (que seguiría teniendo información de los triángulos clasificados por nodo). La *textura de triángulos* contendría en cada fila la información acerca de los triángulos clasificados en cada nodo hoja, no siendo necesario profundizar todas las ramas del árbol al mismo nivel. La *textura de jerarquía* podría contener en cada fila, en el caso de nodos intermedios, información del tipo de nodo, del número de nodos hijos y de la fila en la que se encuentran en la *textura de jerarquía*. En el caso de un nodo hoja, además del tipo de nodo, se almacenaría el número de fila en la *textura de triángulos* con información de la geometría clasificada en el nodo (Figura 3).

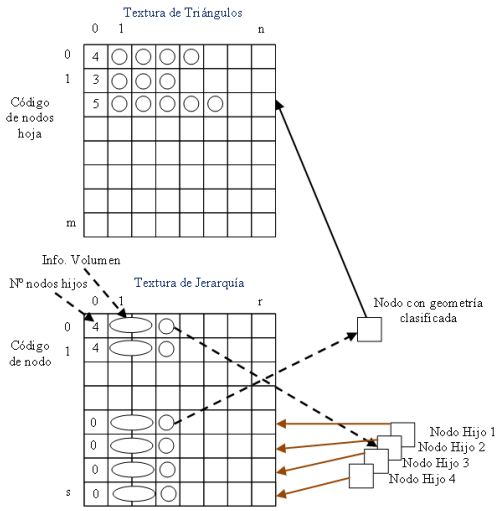

Figura 3: *Representación de la textura de jerarquía.*

En la *textura de jerarquía* podría almacenarse también para cada nodo información sobre el volumen que representa en lugar de almacenar dicha información en la cabecera de cada fila de la *textura de triángulos*.

4.1.2. Construcción en GPU de jerarquías

Veamos a continuación cómo implementar la construcción de las jerarquías anteriores utilizando shaders.

Como ya sabemos, la activación de unidades de ejecución en los shaders viene determinada por el envío a GPU de vértices y/o triángulos. En el caso que nos ocupa nos interesaría provocar la activación de una unidad de ejecución por cada triangulo que queremos clasificar en la jerarquía, e ir construyendo la jerarquía conforme vayan llegando triángulos. Esto podría realizarse en el caso de programas alojados en el *vertex shader* enviando los triángulos codificados en una lista de vértices y propiedades de vértices (como normal y color) de manera que en la lista de vértices se almacene el primer vértice de cada triangulo, en la lista de normales el segundo vértice de cada triángulo, y en la lista de colores el tercer vértice. Una alternativa consistiría en activar

las unidades de ejecución del *geometry shader* enviando directamente como entrada los propios triángulos como geometría a rasterizar (Figura 4).

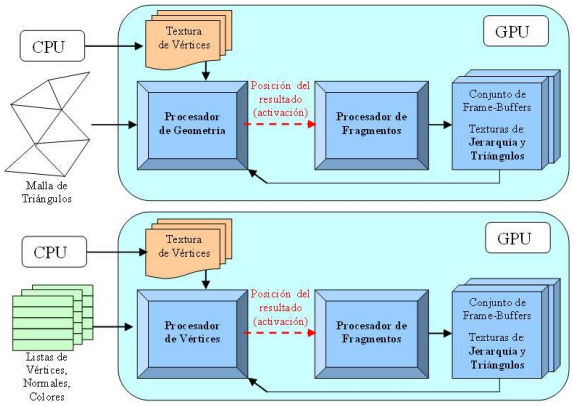

Figura 4: *Implementación de la construcción de una descomposición espacial jerárquica mediante geometry o vertex shader, usando texturas de jerarquía.*

Una vez activada la ejecución de unidades por cada triángulo, habría que clasificarlo y obtener una salida que se escribiría en el frame buffer o en textura. Esta salida consistiría en actualizar las *texturas de triángulos y de jerarquía*.

Debido a la independencia de las unidades de ejecución no es posible compartir información entre unidades, es decir, no se puede utilizar una variable que sea global para todas las unidades de modo que compartan/actualicen la información contenida en dicha variable de manera efectiva. Por ejemplo, para clasificar un triángulo en un determinado volumen sería necesario conocer cuál sería la siguiente posición libre en la fila correspondiente de la *textura de triángulos* para poder escribir en ella. Podríamos utilizar una posición fija en una textura a modo de variable global, de modo que todas las unidades leyeran/escribieran información en esa posición, pero se producirían problemas de sincronismo y de integridad de la información al no existir un mecanismo de sincronización como en el procesamiento paralelo clásico. Por tanto podemos concluir que utilizando este modelo no sería posible construir una *textura de jerarquía* utilizando shaders, debido a la imposibilidad de compartir de manera efectiva información variable entre las unidades de ejecución.

Una alternativa para tratar de construir una jerarquía basada en *texturas de vértices* y *de triángulos* (Figura 1) consistiría en provocar la ejecución de las unidades por niveles en la jerarquía, y no por cada triángulo a clasificar, es decir, provocar la ejecución de tantas unidades asociadas a un determinado shader como nodos haya en un nivel en la jerarquía, repitiendo este proceso para cada nivel desde la raíz a los nodos hoja. Así cada unidad de ejecución clasificaría todos los triángulos previamente clasificados en el nivel anterior, de modo que cada unidad de ejecución escribiría exclusivamente en su fila en la *textura de triángulos*, independientemente de las demás unidades (Figura 5). Sería necesario realizar una pasada por nivel, de modo que se activaran las unidades de ejecución para nodos del mismo nivel, ya que si hubiera nodos de distintos niveles ejecutándose a la vez podría darse el caso de clasificación de triángulos en un nivel de mayor profundidad sin que se hubiera terminado la clasificación de triángulos en el nivel anterior.

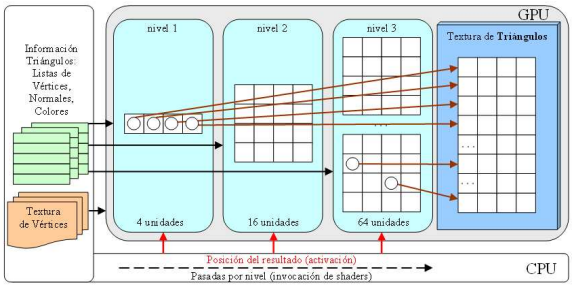

Figura 5: *Implementación de la construcción de una descomposición espacial jerárquica mediante shaders, realizando varias pasadas, una por nivel de jerarquía.*

Además sería necesario almacenar en cada nodo intermedio los triángulos clasificados en dicho nodo para poder llegar a los nodos hoja, lo que nos restaría en versatilidad por la limitación inherente en los tamaños de textura. Al ir realizando pasadas por niveles podríamos conservar la información de los triángulos clasificados por nodo, de padres a hijos, eliminando posteriormente la información de los nodos padre, de modo que finalmente sólo permaneciera la información de los triángulos clasificados a nivel de nodos hoja (Figura 6).

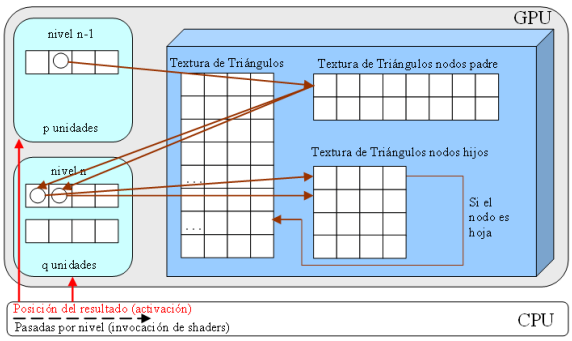

Figura 6: *Triángulos clasificados en texturas auxiliares.*

Para conseguir la clasificación de los triángulos según este modelo se podría provocar la ejecución de *vertex shaders* (o de cualquier otro shader), para lo que sería necesario enviar un vértice de activación por cada nodo del nivel (que podría almacenar el número de nodo). Cuando se determine que no habrá que profundizar más en la jerarquía, se podría

escribir información en una posición clave en la *textura de triángulos* sobre la activación o no de los nodos hijos, de modo que aunque todos los nodos de un nivel pasaran a una unidad de ejecución, el programa consultaría esta información de activación para realizar o no la clasificación. Además los nodos hoja deberán transferir la información de los triángulos clasificados en el nodo a la *textura de triángulos* con la información definitiva. Nuevamente, vemos que la *textura de triángulos* debe tener tantas filas como nodos totales en la jerarquía, incluyendo los intermedios, ya que no se sabe a priori lo que se descenderá en una determinada rama de la jerarquía. En cambio, el número de triángulos clasificados por nodo hoja para la *textura de triángulos* puede restringirse si el criterio de parada incluye no profundizar más cuando el número de triángulos es menor que un umbral.

El principal inconveniente de esta aproximación radica nuevamente en la limitación en los tamaños de las texturas, ya que aunque la *textura de triángulos* final será igual al número máximo de nodos de la jerarquía multiplicado por el número máximo de triángulos clasificados en un nodo hoja, la necesidad de utilizar texturas adicionales de gran tamaño para almacenar la información de los triángulos clasificados en los nodos intermedios, la cantidad de texturas necesarias (tantas como nodos intermedios de cada nivel), y el trasiego de información necesario para actualizar la *textura de triángulos* con la información de los nodos hoja, implica una eficiencia menor de lo esperado así como una limitación en cuanto al tamaño de las mallas de triángulos y niveles de profundidad en la jerarquía.

4.1.3. Utilización para operaciones de consulta

A pesar de las limitaciones anteriores, podemos considerar cómo sería el acceso a las estructuras de datos almacenadas en textura para utilizarlas en diversas aplicaciones.

Consideremos que se ha construido una jerarquía asociada a una malla de triángulos mediante la codificación de *texturas de vértices y de triángulos*, esta última con tantos niveles como nodos en la jerarquía y con información de los triángulos clasificados en los nodos hoja. Cada fila de la *textura de triángulos* representa los triángulos clasificados en un nodo hoja, no existiendo esta información para los nodos intermedios. La información sobre el volumen asociado a cada nodo puede estar almacenada en otra textura o calcularse según convenga.

Veamos cómo se realizarían las principales operaciones de consulta vistas en la sección 3.2.:

- **Dado un nodo, obtención de la geometría clasificada en dicho nodo**: Una vez conocido un número de nodo, sólo habría que acceder a la fila correspondiente en la *textura de triángulos* para obtener los índices de los vértices que forman los triángulos. El número de triángulos estará en la cabecera de esa fila, y para acceder a los vértices del triángulo accederemos mediante los índices anteriores a la *textura de vértices*.
- **Dada una posición en el espacio, obtención del nodo hoja que la contiene**: Podría utilizarse una función que dada una posición y un nivel, obtenga el número de fila correspondiente. Pero a priori no conocemos si en dicho nivel hay o no geometría clasificada, teniendo que consultarlo en la *textura de triángulos*. Debido a este problema habría que comenzar en la raíz del árbol e ir clasificando la posición a través de los hijos hasta llegar a un nodo hoja. Se puede utilizar la operación de obtención de los nodos hijos definida a continuación, consultar o calcular sus volúmenes asociados, clasificar la posición en sus volúmenes, consultar si el nodo hijo por el que descender es o no nodo hoja y descender por la rama correspondiente.
- **Obtención de los nodos hijos y padre**: Dado un nodo, la obtención de los nodos hijos o el nodo padre puede realizarse mediante una función que obtenga la fila correspondiente en la *textura de triángulos*.
- **Actualización de la jerarquía tras una deformación de la malla**: Aunque no es una operación de consulta, una modificación de la geometría de la malla implicaría una reconstrucción total de la jerarquía asociada a la misma.

4.2. Implementación basada en codificación a nivel de triángulos

Vamos a tratar de dar una solución que resuelva los problemas de construcción de jerarquías anteriores y que optimice el acceso a la información de forma más versátil y que pueda utilizarse con mallas de mayor tamaño. Además intentaremos solventar otros de los problemas vistos, como la independencia de la visualización con los cálculos, y la posibilidad de actualización de las jerarquías ante deformaciones.

En esta sección veremos en primer lugar como codificar la información de entrada y resultado de la construcción de la jerarquía, seguido del procedimiento para su construcción. Finalizaremos con los métodos de consulta.

4.2.1. Codificación de la información

En este caso enviaremos información geométrica de la malla como flujo de triángulos de entrada, invocando la rasterización de la geometría, y almacenaremos la información de la jerarquía construida en una textura, pero de un modo distinto al habitual.

La incorporación del procesador de geometría programable al pipeline gráfico nos posibilita obtener información de los triángulos de la malla en cada unidad de ejecución del *geometry shader*. Cuando se envíe la geometría del objeto considerado para su visualización, activaremos una unidad de ejecución en el *geometry shader* por cada triángulo que entre al pipeline gráfico. Cada unidad de ejecución clasificará el triángulo en un determinado nivel de forma independiente a los demás.

No necesitaremos almacenar la información de vértices y

de triángulos como vimos previamente, sino que asociaremos un índice a cada triángulo de la malla. La idea consiste en almacenar información en cada triángulo de los nodos de la jerarquía en los que se encuentra clasificado. Supongamos una malla formada por n triángulos, cada triángulo tendrá asociado un índice que se corresponderá con una determinada posición en una textura de n elementos. En la posición de cada triángulo en la textura se almacenará el código de los nodos del mismo nivel en el que se encuentre clasificado ese triángulo. Por tanto tendremos una textura con n elementos para cada nivel de la jerarquía (*textura de nodos de nivel i*). Así cada posición de una textura de un nivel determinado contendrá el código de los nodos en los que se encuentra almacenado el triángulo correspondiente en ese nivel (Figura 7).

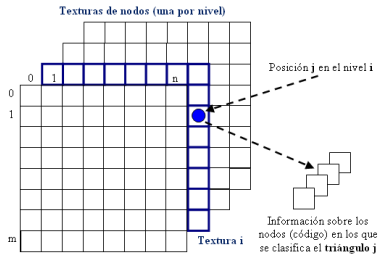

Figura 7: *Texturas de nodos de nivel i.*

Como un triángulo puede estar clasificado en varios nodos de un mismo nivel, el número máximo de nodos en los que es clasificado un triángulo en un determinado nivel no puede saberse a priori, siendo posible que un triángulo esté clasificado en un gran número de nodos en los niveles más profundos del árbol. Para resolver este problema podemos establecer un número máximo de nodos del mismo nivel en el que se encuentra clasificado un triángulo. De este modo, cuando un triángulo en un nivel sea clasificado en más nodos que este máximo, pararemos el proceso de descenso a otro nivel para ese triángulo.

En este caso sería posible enviar a la GPU la información sobre los volúmenes asociados a los nodos codificados en texturas, en lugar de tener que calcularlos a partir del volumen inicial. Un modo de codificar dicha información puede ser mediante una textura 2D que llamaremos *textura de volúmenes*. En esta textura se almacenaría en cada fila la información necesaria para generar el volumen asociado a cada uno de los nodos de un mismo nivel.

4.2.2. Construcción en GPU

La construcción de las texturas asociadas es bastante sencilla. Dada una malla de n triángulos y un número de niveles máximo l para la jerarquía, habrá que enviar la geometría de la malla l veces para su visualización, una por nivel, para que las unidades del *geometry shader* se activen por cada triángulo de la malla. Además de la información de la geometría, en cada triángulo debe codificarse la información del número de triángulo, para que el *geometry shader* pueda escribir en su posición en la textura del nivel correspondiente. Así el *geometry shader* emitirá un vértice con posición el código del triángulo de entrada y con color la codificación de los números de nodo de ese nivel en los que se encuentre clasificado dicho triángulo (Figura 8).

Es decir, para cada pasada de construcción tendremos los resultados del nivel anterior y la información referente a la estructura jerárquica del nivel actual, con ella generaremos al final del pipeline gráfico una nueva matriz de resultados.

Para poder pasar de un nivel al siguiente es necesario cambiar el destino de la escritura mediante Framebuffer Objects [fVIT08]. Esta característica introducida en OpenGL 2.1 y mejorada en OpenGL 3.0 permite modificar la salida del procesador de fragmentos asociándolo con una textura de dimensiones adecuadas. Además esta textura puede ser definida como textura de acceso dentro de los procesadores del cauce gráfico, es decir, estamos permitiendo la realimentación del cauce gráfico sin necesidad de utilizar la comunicación CPU - GPU que siempre supone un cuello de botella en el rendimiento global.

4.2.3. Utilización para operaciones de consulta

Supongamos que se ha construido la jerarquía asociada a una malla de triángulos mediante *texturas de nodos* para cada uno de los niveles de la jerarquía. La información sobre el volumen asociado a cada nodo puede estar almacenada en otra textura o calcularse según convenga.

Al igual que en la sección 4.1.3. veremos cómo implementar algunas de las operaciones de consulta habituales:

- **Dado un nodo, obtención de la geometría clasificada en dicho nodo**: En este caso la obtención de la geometría clasificada por nodo no es tan inmediata, pero si efectiva ya que habría que rasterizar de nuevo la malla. Así, cada unidad de ejecución en el *geometry shader*, ante un triángulo de entrada consultaría en todas las *texturas de nodos* de los distintos niveles si se encuentra el código de nodo entre la información almacenada asociada a la posición de su triángulo, además consultaría si se encuentra el código de nodo de todos sus nodos ascendientes sin incluir la raíz. Este código de nodo sería un valor constante de entrada para todas las unidades de ejecución. En caso de encontrar dicho código de nodo podría p.e. colorear dicho triángulo (feedback visual) o enviar información booleana (0,1) a una nueva textura sobre la clasificación del triángulo en el nodo correspondiente o en sus ancestros.
- **Dada una posición en el espacio, obtención del nodo hoja que la contiene**: Podríamos utilizar una función que dada la posición y nivel obtenga el código de nodo asociado. En cambio, no conocemos a priori si en dicho nivel hay geometría clasificada o no. Si la finalidad de esta consulta es obtener la geometría clasificada en dicho nodo

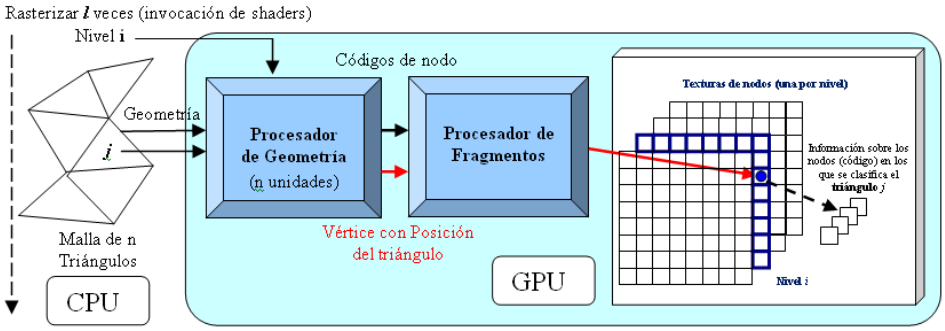

Figura 8: *Construcción de una descomposición espacial usando geometry shaders y texturas de nodos.*

hoja, podemos utilizar el algoritmo descrito anteriormente para obtener la geometría clasificada en un nodo consultando todos los ancestros, por lo que obtendríamos una solución correcta.

- **Obtención de los nodos hijos y padre**: Dado un nodo, la obtención de los nodos hijos o el nodo padre puede realizarse mediante una función que obtenga el código de nodo.
- **Actualización de la jerarquía tras una deformación de la malla**: En este caso, tras localizar los triángulos que han sido modificados por la deformación, simplemente habría que volver a lanzar dichos triángulos para su clasificación, no siendo necesario reconstruir el resto de la información para otros triángulos.

4.3. Implementación de Tetra-Trees

A lo largo de esta sección describiremos la descomposición espacial utilizada en nuestras pruebas, denominada Tetra-Tree [JFS06] [JS08].

Un Tetra-Tree realiza una descomposición recursiva del espacio por medio de tetra-conos. Intuitivamente, un tetra-cono es una región del espacio delimitada por tres planos que intersecan en un punto y que puede ser definida en base a un tetraedro (Figura 9.a).

Un Tetra-Tree está formado por diversos tetra-conos con un origen común, normalmente el centroide del objeto, de modo que estos tetra-conos cubran totalmente el espacio sin solapamientos excepto en su frontera (Figura 9.b). Cuando se construye un Tetra-Tree, los triángulos que forman el objeto son clasificados en cada uno de sus tetra-conos iniciales. Cada tetra-cono se subdivide recursivamente en nuevos tetra-conos (Figura 9.a), normalmente hasta que se alcanza un profundidad máxima en el árbol, o hasta que el número de triángulos clasificados en un tetra-cono sea menor que un valor preestablecido.

En la construcción de un Tetra-Tree, se utilizará una *textura de volumen* que contendrá los puntos que definen cada tetra-cono para todos los niveles contemplados (hasta la máxima profundidad). El centroide del objeto, común a todos los tetra-conos, será una variable de entrada al procesador. La *textura de nodos* se ha implementado según la definición dada para este tipo de textura. En cada posición de la *textura de nodos de nivel i* la componente Z corresponde a la profundidad de la jerarquía y las coordenadas X e Y corresponden al índice del triángulo en la lista de triángulos. Hemos establecido que cada triángulo no podrá ser clasificado en más de cuatro tetra-conos por nivel, obteniendo el criterio de no profundizar más en la jerarquía para un triángulo si se sobrepasa este límite. El código puede verse en la página http://wwwdi.ujaen.es/~juanjo/ttgpu/.

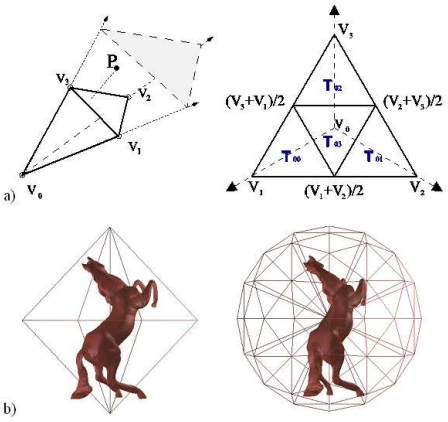

Figura 9: *a) Un tetra-cono y construcción de sub-tetra-conos. b) Tetra-Tree de niveles 1 y 3.*

Como se puede observar en la Figura 10, la velocidad de clasificación de triángulos en GPU supone una escasa penalización en relación a los tiempos de escritura en el *FB-Object*. Los tiempos obtenidos son muy inferiores a los tiempos en CPU, siendo la mejora mucho mayor para grandes mallas y alto número de niveles en la jerarquía. Las pruebas

han sido realizadas en un Intel Quad Core con dos tarjetas Nvidia 8800 GT SLI, con S.O. Ubuntu 8.10.

dragon, 437.645 vértices, 871.414 triángulos				
nivel	Clasif. CPU	Clasif. GPU	Escrit. FB	Clasif GPU + E.FB
2	31,53	0,47	1,12	1,59
3	40,48	0,70	1,62	2,32
4	33,14	0,59	1,84	2,43
5	79,16	1,75	2,01	3,76

parthenon, 52.239 vértices, 104.096 triángulos				
nivel	Clasif. CPU	Clasif. GPU	Escrit. FB	Clasif GPU + E.FB
2	3,19	0,27	0,14	0,41
3	4,37	0,23	0,24	0,47
4	5,98	0,26	0,30	0,56
5	6,13	0,33	0,33	0,66

bunny, 35.947 vértices, 69.451 triángulos				
nivel	Clasif. CPU	Clasif. GPU	Escrit. FB	Clasif GPU + E.FB
2	2,44	0,24	0,09	0,33
3	3,10	0,21	0,18	0,39
4	3,70	0,27	0,18	0,45
5	4,83	0,30	0,23	0,53

Figura 10: *Tiempos en seg. para la clasificación de mallas en Tetra-Trees por niveles usando CPU y GPU. También se muestran los tiempos de escritura en el Frame-Buffer.*

5. Conclusiones

En este artículo proponemos distintos métodos para la codificación en texturas de la información relativa a la descomposición espacial de una malla para su utilización directa en GPU, aprovechando el paralelismo inherente al hardware gráfico. Se proponen además diversos métodos de construcción de la jerarquía asociada también en GPU, viendo las ventajas e inconvenientes de cada método. Se propone una solución basada en *geometry shaders* mediante una codificación de la jerarquía a nivel de triángulos, distinta de la habitual, pero que nos facilita la construcción de dicha jerarquía, así como la realización de diversas operaciones de consulta sobre la misma, lo que nos da mayor versatilidad que otros tipos de métodos, siendo posible de forma simultánea la realización de operaciones sobre la jerarquía y la visualización, todo en GPU. El método propuesto nos permite su extensión e implementación eficiente de otras operaciones, como la modificación a nivel local de la jerarquía, sin la necesidad de reconstruirla completamente.

Pretendemos que este trabajo sirva de referencia para la construcción de descomposiciones espaciales en GPU. Proporcionamos una serie de pautas que pueden facilitar el trabajo de diseño de aplicaciones utilizando shaders, así como una descripción de implementaciones llevadas a cabo de forma efectiva.

Actualmente se está trabajando en el diseño y adaptación de algoritmos para la construcción de jerarquías de volúmenes envolventes, así como en aplicaciones que utilicen en GPU la implementación de las descomposiciones espaciales realizadas y su aplicación a modelos deformables. Por último se están probando implementaciones basadas en lenguajes de propósito general (CUDA) para este tipo de descomposición, pero los resultados parciales obtenidos auguran peores resultados que utilizando la implementación mostrada en este artículo.

6. Reconocimientos

Este artículo ha sido parcialmente subvencionado por el MEC y la Unión Europea (fondos FEDER) a través del proyecto de investigación TIN2007-67474-C03-03, por la Consejera de Innovación Ciencia y Empresa de la Junta de Andalucía a través de los proyectos de investigación P06-TIC-01403 y P07-TIC-02773, y por la Universidad de Jaén a través del proyecto de investigación UJA-08-16-02.

References

[AL06] ANDERSON D., LUKE R.: *GPU GLSL Geometry Shader*. Tech. rep., 2006.

[CUD08] *Nvidia CUDA. Programming Guide*. Nvida, 2008.

[Fer05] *GPU-Gems 2. Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Pearson Education, 2005.

[fVIT08] FOR VISUAL INFORMATION TECHNOLOGY C.: *Example Codes for Shader Model 4.0 (OpenGL 2.1, GLSL 1.20)*. Tech. rep., Centre for Visual Information Technology, 2008.

[GÖ5] GÖDDEKE D.: *GPGPU–Basic Math Tutorial*. Tech. rep., FB Mathematik, Universität Dortmund, Nov. 2005. Ergebnisberichte des Instituts für Angewandte Mathematik, 300.

[JFS06] JIMÉNEZ J. J., FEITO F. R., SEGURA R. J.: Particle oriented collision detection using simplicial coverings and tetra-trees. *Computer Graphics Forum 25*, 1 (2006), 53–68.

[Jim06] JIMÉNEZ J. J.: *Detección de Colisiones Mediante Recubrimientos Simpliciales*. PhD thesis, Dpto. de Lenguajes y Sistemas Informáticos - Universidad de Granada, 2006.

[JOSF06] JIMÉNEZ J. J., OGÁYAR C. J., SEGURA R., FEITO F. R.: Collision detection between a complex solid and a particle cloud assisted by programmable gpu. In *3rd Workshop in Virtual Reality Interactions and Physical Simulation* (2006).

[JS08] JIMÉNEZ J. J., SEGURA R. J.: Collision detection between complex polyhedra. *Computers & Graphics 32*, 4 (2008).

[KHSZ98] KLOSOWSKI J., HELD M., SOWIZRAL J. M. H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *EEE Transactions on Visualization and Computer Graphics* (1998).

[NAT90] NAYLOR B., AMANATIDES J., THIBAULT W.: Merging bsp trees yield polyhedral modeling results. *Proceedings of ACM Siggraph* (1990).

[OGL05] *OpenGL Shading Language Course*. Typhoon L., 2005.

[Ros06] ROST R.: *OpenGL Shading Languaje*. A.Wesley, 2006.

[Swa93] SWAM J.: *Octree-based collision detection with fast neighbor finding*. Tech. rep., OSU/ACCAD-12/93-TR7, 1993.

[Yon06] YONGMING X.: *Geometry Shader Tutorials*. Tech. rep., Visualization and Imaging Research Centre,The Chinese University of Hong Kong, 2006.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph 27*, 5 (2008), 1–11.