UNIVERSIDAD DE MURCIA

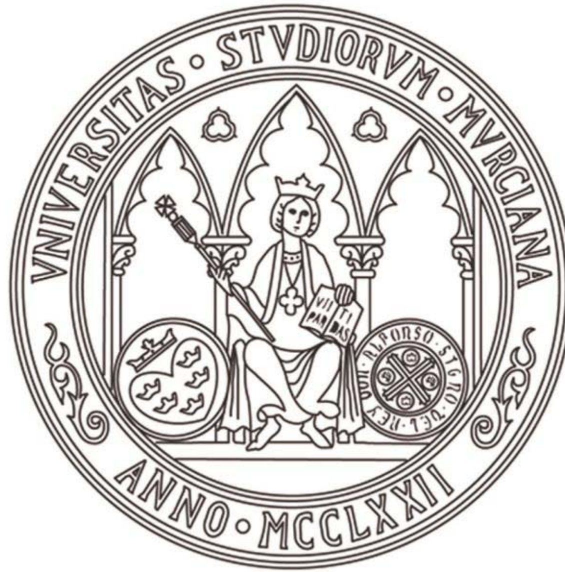
ESCUELA INTERNACIONAL DE DOCTORADO

TESIS DOCTORAL

Diseño de GPUs eficientes energéticamente explotando la coherencia entre fotogramas y optimizando los accesos a memoria

D. David Corbalán Navarro

2023



UNIVERSIDAD DE MURCIA

ESCUELA INTERNACIONAL DE DOCTORADO

TESIS DOCTORAL

Diseño de GPUs eficientes energéticamente explotando la coherencia entre fotogramas y optimizando los accesos a memoria

Autor: D. David Corbalán Navarro

Directores: D. Juan Luis Aragón Alcaraz y D. Antonio González Colás

UNIVERSIDAD DE
MURCIA

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD
DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR
Aprobado por la Comisión General de Doctorado el 19-10-2022

D./Dña. David Corbalán Navarro

doctorando del Programa de Doctorado en

Programa de Doctorado en Informática

de la Escuela Internacional de Doctorado de la Universidad Murcia, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

Diseño de GPUs eficientes energéticamente explotando la coherencia entre fotogramas y optimizando los accesos a memoria

y dirigida por,

D./Dña. Juan Luis Aragón Alcaraz

D./Dña. Antonio González Colás

DECLARO QUE:

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita, cuando se han utilizado sus resultados o publicaciones.

Si la tesis hubiera sido autorizada como tesis por compendio de publicaciones o incluyese 1 o 2 publicaciones (como prevé el artículo 29.8 del reglamento), declarar que cuenta con:

- *La aceptación por escrito de los coautores de las publicaciones de que el doctorando las presente como parte de la tesis.*
- *En su caso, la renuncia por escrito de los coautores no doctores de dichos trabajos a presentarlos como parte de otras tesis doctorales en la Universidad de Murcia o en cualquier otra universidad.*

Del mismo modo, asumo ante la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada, en caso de plagio, de conformidad con el ordenamiento jurídico vigente.

En Murcia, a 16 de marzo de 2023

Fdo.: David Corbalán Navarro

Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la primera página de la tesis presentada para la obtención del título de Doctor.

Información básica sobre protección de sus datos personales aportados	
Responsable:	Universidad de Murcia. Avenida teniente Flomesta, 5. Edificio de la Convalecencia. 30003; Murcia. Delegado de Protección de Datos: dpd@um.es
Legitimación:	La Universidad de Murcia se encuentra legitimada para el tratamiento de sus datos por ser necesario para el cumplimiento de una obligación legal aplicable al responsable del tratamiento. art. 6.1.c) del Reglamento General de Protección de Datos
Finalidad:	Gestionar su declaración de autoría y originalidad
Destinatarios:	No se prevén comunicaciones de datos
Derechos:	Los interesados pueden ejercer sus derechos de acceso, rectificación, cancelación, oposición, limitación del tratamiento, olvido y portabilidad a través del procedimiento establecido a tal efecto en el Registro Electrónico o mediante la presentación de la correspondiente solicitud en las Oficinas de Asistencia en Materia de Registro de la Universidad de Murcia

Agradecimientos

En primer lugar, me gustaría darle las gracias a mi familia por su apoyo constante e incondicional a lo largo, no sólo de la realización de esta Tesis Doctoral, sino de toda mi vida y de las decisiones que he ido tomando, estuviera o no equivocado. Siempre habéis estado a mi lado, en mis mejores y peores momentos. No tengo palabras para expresar lo que habéis significado para mí durante todo este tiempo.

Cómo no, también quiero agradecer a mis amigos de toda la vida, Aure, Mario, Fran y Pablo, todos los momentos que he pasado con vosotros que, aunque no hayan podido ser muchos durante la realización de esta Tesis, han sido intensos e inolvidables... como acostumbramos que sean. Vosotros habéis sido la carga de positividad que necesitaba en los momentos más complicados. Por supuesto, también me gustaría agradecer el apoyo de mi círculo cercano: Francis, Antonio José, Nico, Gabriel y David.

También quiero agradecer a mis directores de Tesis, Juan Luis Aragón y Antonio González, toda la confianza que habéis puesto en mí a la hora de dirigir esta Tesis Doctoral. Con vosotros he descubierto el mundo de la investigación y lo he vivido en primera persona. Para mí, una experiencia que ha cambiado mi vida profesional y, por supuesto, de la que siempre os estaré agradecido. Gracias también por haber estado ahí en los *deadlines* más difíciles, aunque fuera a escasos minutos de que se cerrara el plazo para subir el *paper*. Al hilo de esto, perdón por haber apurado tantísimo tanto los plazos. También, me gustaría agradecer la gran ayuda de Joan-Manuel Parcerisa, por haber estado ahí también.

También quiero darle las gracias al grupo ARCO de Barcelona por haberme dejado formar parte de ese maravilloso grupo de investigación, con el que he com-

partido tan buenos momentos. Gracias a José María, Enrique, Martí, Diya, Marc, Franyell, Andreas, Albert, Jordi, Josep, Dennis, Pedro Henrique, Raúl, Jorge, Rodrigo, Aurora, Mojtaba, Nitesh, Imad, Mehdi, Mohamad y Bahareh.

Por supuesto, no podría olvidarme de mis compañeros de Tesis, y amigos, Paco y Eduardo, por todos esos momentos en el laboratorio de la Facultad, intercambiando conocimientos, hablando de cualquier tema, jugando a la diana, clavando una placa base del año de la polca en un tablacho de madera y anclándolo a la pared... lo típico vamos. Sin vosotros esta Tesis Doctoral no habría sido posible, pues vuestro apoyo y los conocimientos que he aprendido gracias a vosotros han sido imprescindibles.

También me gustaría agradecer a todo el departamento y al grupo CAPS el agradable ambiente de trabajo que he tenido durante este tiempo. A Paco Plana, Alberto, Rubén, Juan Manuel, Lorenzo, Juan, Antonio Javier, José Manuel, Manolo, Diego, Ricardo, Antonio Flores, Pedro Enrique, Gregorio, Félix, Óscar, Ashkan, Sawan, Víctor, Pilar, Gema, Sebastián y Paco Guil.

Como no podría ser menos, me gustaría agradecer a la conserjería, y a quienes eventualmente pasan por ahí, esos momentos de desconexión completamente necesarios. A Michelle, Judit, Esther, Carmen, Mateo, Antonio Hernández, Paco Montoya, Josefa y María Luisa. Pero especialmente a ti, Nana, por haberme cuidado tanto durante todos estos años. Sin todos esos deliciosos platos, hacía tiempo ya que habría desfallecido en el intento.

Y por último, y no por ello menos importante, gracias también a la gente de Voptica, los Joses, Santi, Lucía, Consu, Luis, Sunil, Shoaib, Rosa, Pedro y, en especial, a Pablo Artal por darme la oportunidad de formar parte de esta gran familia.

Gracias.

A mi madre

Resumen

El uso de dispositivos móviles tales como *smartphones*, *tablets* o *smartwatches* se ha extendido de una manera desenfrenada durante los últimos años, de tal forma que hoy en día forman parte de nuestra vida cotidiana. A su vez, los usuarios demandan cada vez más funcionalidades y capacidades como, por ejemplo, más rendimiento en el apartado de videojuegos. En este caso, los videojuegos se mueven gracias al procesador gráfico del dispositivo, conocido como GPU (*Graphics Processing Unit*) y aunque éstos precisan del uso de la CPU para procesar otro tipo de tareas no gráficas como el motor físico o la inteligencia artificial, la mayor parte del trabajo va destinada a la GPU. A medida que los videojuegos se vuelven más complejos en cuanto a la calidad de los gráficos y la resolución de visualización, las GPUs han que ser más potentes para moverlos de una manera fluida, de modo que no afecte negativamente a la experiencia de usuario. Sin embargo, este aumento de rendimiento conlleva un aumento del consumo energético, lo que reduce drásticamente la batería de los dispositivos. Es por esto que el diseño de nuevas microarquitecturas energéticamente eficientes para GPUs móviles es un ámbito de investigación muy activo y en continuo desarrollo.

En esta Tesis se proponen, diseñan, implementan y evalúan tres técnicas para GPUs de dispositivos móviles destinadas a reducir el consumo energético. Para lograrlo, se aprovecha el principio de coherencia temporal entre fotogramas, el cual establece que dos fotogramas cercanos en el tiempo deben ser similares debido a la naturaleza inherentemente continua del movimiento de los objetos en una escena. Así pues, si dos fotogramas son similares en apariencia, los cómputos necesarios para generarlos también deberían serlo. De este modo, se pueden aprovechar cálcu-

los del fotograma anterior, ahorrando así energía y aumentando el rendimiento.

El proceso para dibujar un fotograma se denomina renderizado. Este proceso se lleva a cabo en forma de *pipeline*, es decir, por etapas que pueden solaparse en el tiempo. Existen esencialmente dos formas de renderizar una escena: de manera global y directa, habitualmente denominada *Immediate Mode Rendering* (IMR); o por pequeños trozos de pantalla denominados *tiles*, también conocida como *Tile-based Rendering* (TBR). En esta Tesis nos centramos en la segunda opción. Una arquitectura TBR procesa la escena por pequeños cuadrados de tamaño fijo llamados *tiles*. Éstos son generalmente pequeños (ej., 32x32 píxeles), de modo que pueden caber en pequeñas memorias *on-chip* de acceso rápido, reduciendo así el tráfico a memoria principal y el consecuente gasto energético. Esto último es lo que hace que TBR sea una arquitectura especialmente atractiva para dispositivos móviles. En TBR, el renderizado se lleva a cabo en tres fases esenciales, a saber: procesado de la geometría, generación de *tiles* y renderizado de fragmentos. El procesado de la geometría, llevado a cabo por el *Geometry Pipeline*, se encarga de manipular la geometría de la escena y llevarla al plano de la pantalla por medio de la proyección de vértices. La generación de *tiles*, llevada a cabo por el *Tiling Engine*, consiste en dividir la geometría resultante de la escena (triángulos en esencia) en los mencionados *tiles* por medio de la generación de listas de triángulos que solapan con cada *tile* de la pantalla. El renderizado de fragmentos, llevado a cabo por el *Raster Pipeline*, consiste en convertir la geometría perteneciente a cada *tile* en lo que se denominan fragmentos, que finalmente son visibles (en forma de píxeles) en la pantalla. Los fragmentos no son más que el resultado de discretizar el área de un triángulo en puntos indivisibles que llevan asociados consigo una lista de atributos (color, posición, normales, etc.).

Como propuestas de esta Tesis, en primer lugar se presenta Ω-Test como solución a un problema denominado *overdraw*, el cual se explica a continuación. Una de las tareas que se llevan a cabo durante el *Raster Pipeline* es el test de profundidad (o Z-Test), que consiste en determinar la visibilidad de cada fragmento en función de su profundidad. Para lograrlo, se almacena por cada píxel de la pantalla la profundidad del fragmento más cercano a la cámara hasta ese momento. A esta estructura se le denomina Z-Buffer. Cuando un fragmento llega a esta etapa, se compara su

profundidad con la correspondiente (por posición en la pantalla) almacenada en el Z-Buffer, y si ésta es más cercana a la cámara, entonces se dibuja en la pantalla y se actualiza el valor de profundidad en el Z-Buffer. De lo contrario, el fragmento queda descartado por estar ocluido, con lo que no será mostrado en la imagen final. Este esquema presenta un problema, y es que si los objetos de una escena se procesan de atrás hacia delante, entonces todos (o casi todos) los fragmentos van a pasar el Z-Test porque la profundidad actual siempre va a ser más cercana a la cámara que la anterior. De este modo, para una posición determinada de la pantalla se han tenido que dibujar varios fragmentos, a pesar de que sólo uno sea visible finalmente, lo cual supone un desperdicio importante de recursos. Al grado de fragmentos sobredibujados por posición se denomina *overdraw*, y éste supone un 37.7 % en promedio (dados los *benchmarks* evaluados). Para paliar este problema de manera eficaz, Ω-Test aprovecha el Z-Buffer del fotograma previo (lo que se denomina Ω-Table), en vez de partir de cero. Esto hace que, de manera prematura, se descarten muchos más fragmentos, reduciendo así el *overdraw* de la escena. Sin embargo, la información de la que se parte es obviamente imprecisa (a pesar de ser muy parecida), por lo que se pueden producir errores para los cuales Ω-Test incorpora un eficiente mecanismo de corrección, dando lugar a una imagen exactamente igual a la original. Con esto se ha logrado reducir el sombreado proveniente del *overdraw* en un 32.7 % de media, dando lugar a un aumento en el rendimiento del 16.3 % y una reducción del consumo energético del 15.2 % de media.

Como segunda propuesta, se presenta Triangle-Dropping, una técnica que ahorra el procesamiento de primitivas ocluidas en la fase de geometría. A pesar de que con el Z-Test podamos eliminar fragmentos de objetos que quedan ocluidos por otros, su geometría se procesa igualmente. El problema de procesar esta geometría es el tráfico a memoria que supone en una arquitectura TBR. Durante la fase de *Tiling Engine*, la geometría de los *tiles* se almacena en una estructura de datos denominada *Parameter Buffer*. Ésta se almacena en memoria principal puesto que el *Tiling Engine* hace de mediador entre la fase de *Geometry Pipeline* y *Raster Pipeline* para transferir estos *tiles*. Por lo tanto, la geometría ocluida que se almacena en esta estructura es un desperdicio puesto que genera tráfico a memoria innecesario. En los *benchmarks* evaluados se ha podido observar que, de media, el 37.7 % de la geo-

metría de una escena se acaba escribiendo en el *Parameter Buffer*, de la cual el 60.1 % está completamente ocluida. Para reducir drásticamente el número de triángulos ocluidos, Triangle-Dropping predice la visibilidad de los triángulos del fotograma actual aprovechando la del fotograma anterior. Para lograrlo, se almacena en una estructura denominada *Frame Visibility Buffer* un bit por cada triángulo de la escena que indica si ha sido visible o no, de modo que se usa en el siguiente fotograma como indicador de si debe ser procesado o no. Una vez más, la información usada es imprecisa, por lo que la imagen final puede contener errores. Sin embargo, a diferencia de Ω-Test, Triangle Dropping no precisa de un mecanismo para corregir *todos* los errores, dada la naturaleza de la etapa tan temprana en la que se aplica. Es por eso que esta técnica es especialmente conservadora a la hora de eliminar triángulos, dando lugar a una imagen *casi* igual a la original. Gracias a esta serie de optimizaciones, Triangle Dropping consigue reducir un 31.4 % de la geometría presente en el *Parameter Buffer*, lo cual supone un 57 % de la geometría ocluida, dando lugar así a un aumento del rendimiento general del 20.2 %, al mismo tiempo que reduce el consumo energético en un 14.5 % de media.

Finalmente, se propone DTM-NUCA (Dynamic Texture Mapping-NUCA), una técnica capaz de aumentar la capacidad efectiva de las cachés de texturas de los núcleos de procesamiento de fragmentos (los denominados *Fragment Processors*). En una jerarquía de memoria clásica compuesta por varios núcleos de procesamiento, las cachés de primer nivel presentan un alto grado de replicación de bloques de memoria. Esto no es necesariamente negativo para el rendimiento, pues gracias a esto se reduce la latencia media de acceso a texturas. Sin embargo, también reduce de manera drástica la capacidad efectiva del conjunto de cachés de primer nivel, perjudicando así al rendimiento general. Para aprovechar de una manera más eficiente el conjunto de cachés de texturas de los Fragment Processors, DTM-NUCA implementa un mecanismo basado en una arquitectura NUCA (*Non-Uniform Cache Access*) que permite compartir bloques de texturas entre Fragment Processors. Además, DTM-NUCA permite replicación cuando ésta es beneficiosa, de modo que se reducen las latencias medias de acceso a texturas. Para lograrlo, DTM-NUCA incorpora una estructura de datos denominada *Affinity Table* que almacena de manera dinámica el *Fragment Processor* propietario de cada rango de bloques de texturas

(denominados *buckets*). Para determinar el mejor propietario a un *bucket*, se almacena el número de accesos a *bucket* por *Fragment Processor*, de modo que el que más accesos acumule (en un rango de tiempo denominado *epoch*) es el mejor candidato a poseer el citado *bucket*. Respecto a la *Affinity Table*, DTM-NUCA se presenta en dos versiones: centralizada o distribuida. La versión centralizada implementa una única *Affinity Table* central a la que acceden todos los *Fragment Processors*. Mientras que en la versión distribuida, cada *Fragment Processor* posee su versión de la *Affinity Table*, y todas éstas se sincronizan cada cierto tiempo para garantizar que la información que posee cada una está siempre actualizada. Con este esquema de mapeo dinámico se consigue aumentar el rendimiento en un 16.9 % para el caso de la *Affinity Table* centralizada, y en un 15.7 % para el caso de la versión distribuida. Además, se consigue reducir el consumo de energía en un 11.1 % y en un 10.3 % para las configuraciones centralizada y distribuida respectivamente.

En definitiva, a lo largo de esta Tesis se han propuesto, desarrollado y evaluado técnicas micro-arquitecturales avanzadas enfocadas a GPUs móviles con el fin de reducir considerablemente su consumo energético y aumentar el rendimiento, haciendo que la experiencia de usuario sea mucho más satisfactoria.

Summary

The use of mobile devices such as smartphones, tablets or smartwatches has become rampant in recent years, so much so that they are now part of our daily lives. in such a rampant manner over the last few years that they are now part of our daily lives. At the same time, users are demanding more and more functionalities and capabilities, such as, for example, more performance in the video game section. In this case, video games are driven by the device's graphics processor, known as the GPU (Graphics Processing Unit) and although they require the use of the CPU to process other non-graphical tasks such as the physics engine or artificial intelligence, most of the work goes to the GPU. As video games become more complex in terms of graphics quality and display resolution, GPUs have to be more powerful to move them in a smooth way, so that it does not affect the user experience. However, this increase in performance leads to an increase in power consumption, which drastically reduces the battery life of the devices. This is why the design of new energy-efficient microarchitectures for mobile GPUs is a very active and continuously developing area of research.

In this Thesis, three techniques for mobile device GPUs are proposed, designed, implemented and evaluated to reduce power consumption. To achieve this, we take advantage of the principle of temporal coherence between frames, which states that two frames close in time should be similar due to the inherently continuous nature of the motion of objects in a scene. Thus, if two frames are similar in appearance, the computations required to generate them should also be similar. In this way, computations from the previous frame can be leveraged, thus saving energy and increasing performance.

The process of drawing a frame is called rendering. This process is carried out in the form of a pipeline, i.e. in stages that may overlap in time. There are essentially two ways of rendering a scene: globally and directly, usually called Immediate Mode Rendering (IMR); or by small pieces of screen calledtiles, also known as Tile-based Rendering (TBR). In this Thesis we focus on the second option. A TBR architecture processes the scene by small fixed-size squares called tiles. These are generally small (e.g., 32x32 pixels), so that they can fit in small fast-access on-chip memories, thus reducing the traffic to main memory and the consequent energy expenditure. The latter is what makes TBR a particularly attractive architecture for mobile devices. In TBR, rendering is carried out in three essential phases, namely: geometry processing, rendering of the geometry, generation of tiles and rendering of fragments. Geometry processing, carried out by the Geometry Pipeline, is responsible for manipulating the scene geometry and bringing it to the screen plane by means of vertex projection. The generation of tiles, carried out by the Tiling Engine, consists of dividing the resulting scene geometry (essentially triangles) into the aforementioned tiles by generating lists of triangles that overlap with each tile on the screen. Fragment rendering, carried out by the Raster Pipeline, consists of converting the geometry belonging to each tile into what are called fragments, which are finally visible (in the form of pixels) on the screen. Fragments are nothing more than the result of discretizing the area of a triangle into indivisible points that carry with them a list of attributes (color, position, normals, etc.).

As proposals of this Thesis, first of all, we present the Ω-Test as a solution to a problem called overdraw, which is explained below. One of the tasks carried out during the Raster Pipeline is the depth test (or Z-Test), which consists of determining the visibility of each fragment as a function of its depth. To achieve this, the depth of the fragment closest to the camera up to that moment is stored for each pixel of the screen. This structure is called Z-Buffer. When a fragment reaches this stage, its depth is compared with the corresponding depth (by position on the screen) stored in the Z-Buffer, and if this is closer to the camera, then it is drawn on the screen and the depth value in the Z-Buffer is updated. Otherwise, the fragment is discarded as occluded, so it will not be displayed in the final image. This scheme presents a problem, and that is that if the objects in a scene are processed

from back to front, then all (or almost all) fragments will pass the Z-Test because the current depth will always be closer to the camera than the previous one. Thus, for a given screen position, several fragments have had to be drawn, even though only one is finally visible, which is a significant waste of resources. The degree of overdrawn fragments per position is called overdraw, and this is 37.7 % on average (given the benchmarks evaluated). To effectively alleviate this problem, Ω-Test leverages the Z-Buffer of the previous frame (referred to as the Ω-Buffer), rather than starting from scratch. This causes many more fragments to be prematurely discarded, thus reducing the overdraw of the scene. However, the starting information is obviously inaccurate (despite being very similar), so errors can occur, for which Ω-Test incorporates an efficient error correction mechanism, resulting in an image exactly the same as the original. This has resulted in a reduction in shading from the overdraw by an average of 32.7 %, leading to a performance increase of 16.3 % and a reduction in power consumption of 15.2 % on average.

As a second proposal, Triangle-Dropping, a technique that saves the processing of occluded primitives in the geometry phase, is presented. On average in a scene of the evaluated benchmarks, 37.7 % of the primitives are written to memory, which on average account for 60.1 % of the primitives that end up being written to memory (37.7 %) according to the evaluated benchmarks. Although with the Z-Test we can eliminate objects that are occluded by others, their geometry is still processed. The problem with processing this geometry is the memory traffic it involves in a TBR architecture. During the Tiling Engine phase, the geometry of the objects is stored in a data structure called Parameter Buffer. This is stored in main memory since the Tiling Engine mediates between the Geometry Pipeline and Raster Pipeline phase to transfer these tiles. Therefore, the occluded geometry stored in this structure is wasteful since it generates unnecessary memory traffic. In the evaluated benchmarks it has been observed that, on average, 37.7 % of the geometry of a scene ends up being written in the Parameter Buffer, of which 60.1 % is completely occluded. To drastically reduce the number of occluded triangles, Triangle-Dropping predicts the visibility of the current frame's triangles by leveraging that of the previous frame. To achieve this, a bit is stored in a structure called Frame Visibility Buffer for each triangle in the scene indicating whether it has been visible

or not, so that it is used in the next frame as an indicator of whether it should be processed or not. Again, the information used is imprecise, so the final image may contain errors. However, unlike Ω-Test, Triangle Dropping does not require a mechanism to correct all errors, given the nature of the very early stage at which it is applied. This is why this technique is especially conservative in removing triangles, resulting in an image that is almost identical to the original. Thanks to this series of optimizations, Triangle Dropping manages to reduce 31.3 % of the geometry present in the Parameter Buffer, which is 57 % of the occluded geometry, thus resulting in an overall performance increase of 20.2 %, while reducing power consumption by 14.5 % on average.

Finally, DTM-NUCA (Dynamic Texture Mapping-NUCA), a technique capable of increasing the effective capacity of the texture caches of the fragment processing cores (the so-called fragment processors), is proposed. In a classical memory hierarchy consisting of several processing cores, the first-level caches have a high degree of memory block replication. This is not necessarily negative for performance, as this reduces the average texture access latency. However, it also drastically reduces the effective capacity of the top-level cache set, thus hurting overall performance. To make more efficient use of the Fragment Processors' texture cache pool, DTM-NUCA implements a mechanism based on a NUCA (Non-Uniform Cache Access) architecture that allows texture blocks to be shared between Fragment Processors. In addition, DTM-NUCA allows replication when it is beneficial, so that average texture access latencies are reduced. To achieve this, DTM-NUCA incorporates a data structure called Affinity Table that dynamically stores the owner of each range of texture blocks (called buckets). To determine the best owner to a bucket, the number of accesses to bucket per Fragment Processor is stored, so that the one that accumulates more accesses (in a time range called epoch) is the best candidate to own the mentioned bucket. Regarding the Affinity Table, DTM-NUCA comes in two versions: centralized or distributed. The centralized version implements a single central Affinity Table accessed by all the Fragment Processors. While in the distributed version, each Fragment Processor has its own version of the Affinity Table, and all of them are synchronized from time to time to ensure that the information held by each one is always up to date. This dynamic mapping scheme

increases performance by 16.9 % in the case of the centralized Affinity Table, and by 15.7 % in the case of the distributed version. In addition, power consumption is reduced by 11.1 % and 10.3 % for the centralized and distributed configurations respectively.

In short, throughout this Thesis, advanced micro-architectural techniques focused on mobile GPUs have been proposed, developed and evaluated in order to considerably reduce their power consumption and increase performance, making the user experience much more satisfactory.

Índice general

1. Introducción **1**

1.1. Problemas detectados y motivación 4

1.1.1. Fragmentos ocluidos *(overdraw)* 5

1.1.2. Geometría ocluida 7

1.1.3. Replicación de texturas 8

1.2. Contribuciones de la Tesis 10

1.2.1. Ω-Test 10

1.2.2. Triangle Dropping 10

1.2.3. DTM-NUCA 11

1.2.4. Publicaciones 12

2. Contexto y estado del arte **15**

2.1. El *pipeline* gráfico 15

2.1.1. Procesamiento de los vértices de la escena 20

2.1.2. Teselaciones 31

2.1.3. Procesamiento de la Geometría 32

2.1.4. Ensamblado de Primitivas 33

2.1.5. *Clipping & Culling* 34

2.1.6. *Rasterizer* 34
2.1.7. *Early Z-Test* 40
2.1.8. Procesado y sombreado de fragmentos 42
2.1.9. *Late Z-Test* 50
2.1.10. *Color Blending* 52
2.1.11. *Framebuffers* 53
2.2. Cómo se implementa el *pipeline* gráfico en *hardware* 57
2.2.1. Arquitecturas *Immediate Mode Rendering* (IMR) 57
2.2.2. Arquitecturas *Tile-based Rendering* (TBR) 66
2.3. Ejemplos de GPUs móviles comerciales 70
2.3.1. PowerVR (arquitectura TBDR) 71
2.3.2. Mali 73

3. Metodología de trabajo **79**
3.1. Herramientas y entorno de simulación 79
3.2. Benchmarks 83

4. Ω-Test **95**
4.1. Implementación y funcionamiento 97
4.2. Manejo eficiente de los errores 103
4.2.1. Posibles escenarios que pueden provocar errores iniciales . . . 103
4.2.2. Mitigando los errores: el margen delta (δ) 106
4.2.3. Detección y corrección de errores 109
4.3. Resultados experimentales 114
4.3.1. Reducción del *overdraw* 114
4.3.2. Rendimiento y análisis del tráfico a memoria 117

4.3.3. Ahorro de energía 119
4.4. Trabajos relacionados 120
4.4.1. Técnicas basadas en *Deferred Rendering* 122
4.4.2. Técnicas basadas en *HiZ occlusion culling* 124
4.4.3. Otros trabajos relacionados que aprovechan la coherencia entre fotogramas 125
4.5. Conclusiones 126

5. Triangle Dropping **127**
5.1. Identificación de comandos OpenGL entre fotogramas 131
5.2. Comprobación de la visibilidad de un Triángulo 133
5.3. Cómputo de la visibilidad durante el Raster Pipeline 135
5.4. Intervalo de refresco 137
5.5. El Command Matcher 137
5.6. Manejo de primitivas con visibilidad intermitente 140
5.7. Mecanismo de intervalo de refresco dinámico 141
5.8. Resultados 142
5.8.1. Impacto en la calidad de la imagen 142
5.8.2. Reducción de la geometría 144
5.8.3. Reducción del tráfico a memoria DRAM 144
5.8.4. Rendimiento y eficiencia energética 146
5.9. Trabajos relacionados 149
5.10. Conclusiones 154

6. DTM-NUCA **157**
6.1. Arquitecturas NUCA convencionales 159

6.2. Una organización NUCA para las Cachés de Texturas 161
6.2.1. Relación de propiedad de bloques de texturas basada en *buckets* 161
6.2.2. Seguimiento dinámico de la propiedad 164
6.3. Opciones de diseño de DTM-NUCA 166
6.3.1. Affinity Table centralizada 166
6.3.2. Affinity Table distribuida 167
6.4. Ajuste de parámetros de DTM-NUCA 168
6.4.1. Determinando el mejor tamaño de página para el esquema de agrupamiento de DTM-NUCA 169
6.4.2. Número de *buckets* de la Affinity Table 169
6.4.3. Determinando la longitud de época 170
6.5. Resultados experimentales y discusión 173
6.6. Trabajos relacionados 178
6.7. Conclusiones 180

7. Conclusiones y vías futuras **183**
7.1. Conclusiones 183
7.2. Vías futuras 186

Índice de figuras

1.1. Desglose del consumo energético de los componentes de una GPU. . 5

1.2. Mapa de calor representando el *overdraw* de un fotograma del juego Hot Wheels. Las áreas más oscuras corresponden con un factor más bajo de *overdraw*; mientras que las zonas más claras corresponden con un mayor solapamiento de primitivas y, por tanto, a un mayor grado de *overdraw*. 6

1.3. Fotograma de una escena del juego Hot Wheels renderizada en modo *wireframe* donde se observa la cantidad de geometría ocluida. A la izquierda, toda la geometría procesada que cae dentro de la pantalla (27554 triángulos en total); a la derecha, sólo la geometría que es visible finalmente (tan sólo 10439 triángulos), excluyendo aquellas primitivas que están completamente ocluidas. 7

1.4. Grado de replicación de texturas en los *benchmarks* estudiados para una GPU con 4 *Fragment Processors*. 9

2.1. Modelo definido por índices implícitos. 17

2.2. Tipos de primitivas en OpenGL. 18

2.3. Representación estructural de un modelo o malla. 20

2.4. Esquema del *pipeline* gráfico. 21

2.5. Modelo de caja negra de un *vertex shader*. 22

2.6. Ejemplo de *frustum volume* en gris claro. En gris más oscuro, un objeto dentro de él que sería visible. 28
2.7. Mapeo del cubo unitario al dispositivo de visualización. 29
2.8. Ejemplo ilustrativo del flujo de ejecución de la etapa *Primitive Assembly*. 33
2.9. Representación gráfica del producto vectorial de dos vectores `a` y `b`, y la importancia que tiene el orden en el que se aplica sobre la dirección del vector resultante. 37
2.10. Ejemplo de *edge functions* definidas por los lados de un triángulo. . . 37
2.11. Ejemplo de obtención de las coordenadas baricéntricas de un fragmento que pertenece a un triángulo. 39
2.12. Representación del problema de la perspectiva. 40
2.13. Ejemplo del proceso de *texture mapping* con una textura bidimensional. 46
2.14. Ejemplo de *texture mipmapping* con 4 niveles de detalle (LOD). 49
2.15. Implementación del *pipeline* gráfico de una arquitectura basada en IMR. 58
2.16. Implementación del *pipeline* gráfico de una arquitectura TBR. 67
2.17. Estructura interna del Parameter Buffer de una GPU basada en TBR. 68
2.18. Esquema de la arquitectura TBDR de PowerVR. 71
2.19. Esquema general de la arquitectura Valhall de Mali. 74
2.20. Diagrama de un *Shader Core* de la arquitectura Valhall de Mali. 76

3.1. Implementación del *pipeline* gráfico de una arquitectura TBDR usado en TEAPOT. 80
3.2. Flujo de ejecución del entorno de simulación TEAPOT. 81
3.3. Fotograma del videojuego Beach Buggy Racing. 84
3.4. Fotograma del videojuego Derby Destruction. 85
3.5. Fotograma del videojuego Gravity. 86

3.6. Fotograma del videojuego Hell Rider. 87

3.7. Fotograma del videojuego Hot Wheels. 87

3.8. Fotograma del videojuego Hot Wheels. 88

3.9. Fotograma del videojuego Snipper 3D. 89

3.10. Fotograma del videojuego Sonic Dash. 90

3.11. Fotograma del videojuego Counter Strike. 91

3.12. Fotograma del videojuego 300. 92

3.13. Fotograma del videojuego Captain America. 93

3.14. Fotograma del videojuego Vegas Crime Simulator. 93

3.15. Fotograma del videojuego Temple Run. 94

4.1. Desglose de fragmentos sombreados en una escena, clasificados en visibles y ocluidos. 96

4.2. Implementación de Ω-Test sobre una arquitectura TBR. Las nuevas estructuras están delimitadas por una línea de puntos para diferenciarlas mejor 98

4.3. Ejemplo del esquema de compresión basado en *coarsening* usando un *tile* de 9x9 píxeles y un factor de *coarsening* de 3x3. El Z-Buffer se procesa en lotes no solapados de 3x3 píxeles, tal y como establece el factor de *coarsening*. Por cada lote se computa el máximo y se almacena en el Ω-Buffer. Una vez procesados todos los lotes de 3x3, el Ω-Buffer se escribe en la posición correspondiente de la Ω-Table. . . . 101

4.4. Estudio del impacto del *coarsening* y sus funciones de agregado sobre el *overdraw* y los errores iniciales en Ω-Test. 102

4.5. Conjunto de escenarios que podrían producir errores potenciales usando la técnica Ω-Test. (a) representa la escena inicial (fotograma i) con dos primitivas que se solapan renderizadas usando el algoritmo del pintor, es decir, de atrás hacia delante. (b)-(f) representan el fotograma $i+1$ con los posibles movimientos que puede realizar la primitiva B. Los casos (b) y (c) muestran la primitiva B alejándose de la cámara, lo que conlleva a errores potenciales en la imagen como se puede observar. En el caso (b) no se utiliza el margen δ, por lo que se producen errores en todo el área que ocupa la primitiva B sobre A. En el caso (c) sí que se utiliza el margen δ, y como puede observarse, los errores relativos a esta área desaparecen, quedando sólo unos pocos errores en el borde de la primitiva A. En el caso (d) se muestra la primitiva B acercándose a la cámara. Este caso no produce ningún error, puesto que los valores actuales de profundidad (los del Z-Buffer) son más cercanos que los del Ω-Buffer, por lo que se pasan ambos tests. En los casos (e) y (f) la primitiva B se mueve lateralmente. En el caso (e) no se aplica *coarsening*, lo que produce errores en la parte de la primitiva A que antes cubría la primitiva B, y produce *overdraw* en la parte hacia la que se mueve. En el caso (f) se utiliza el *coarsening* con la función de agregado *máximo*, lo que evita los errores provocados en el caso (e), puesto que los valores de profundidad del Ω-Buffer tienden a ser más lejanos que los del Z-Buffer, gracias al uso de la función *máximo* como función de agregado. 104

4.6. Estudio del ratio *overdraw*/errores analizando varios valores para δ: 0.5, 0.05, 0.005, 0.0005 y 0. Los resultados están normalizados al *baseline*. La última barra de cada *benchmark* (dyn) corresponde a la implementación del mecanismo de margen δ dinámico. 107

4.7. Implementación de la fase de corrección. Cualquier número diferente de -1 en cualquier celda del E-Buffer indica un error a corregir en dicha posición. En este ejemplo concreto hay 4 errores en el *quad fragment* situado en la esquina superior izquierda, correspondientes a las primitivas 5, 8 y 4. 111

4.8. Ejemplo del estado final de las colas *Primitive-ID queue* y *Position-XY queue* dado un E-Buffer de 4x4 píxeles (4 *quad fragments*). 112

4.9. Desglose de los fragmentos renderizados para los *benchmarks* evaluados, comparando la arquitectura base, VRO y Ω-Test. 115

4.10. Comparación del rendimiento entre VRO, Ω-Test y una implementación HSR perfecta. Los resultados de tiempo de ejecución están normalizados respecto a la arquitectura base. 118

4.11. Accesos a la Texture Cache desglosados en aciertos y fallos que van a ir a la L2, normalizados respecto a la arquitectura base. 119

4.12. Cantidad total de accesos a DRAM (normalizados respecto a la arquitectura base) para los *benchmarks* evaluados. 120

4.13. Ahorro de energía logrado por Ω-Test en una arquitectura TBR, normalizado respecto a la arquitectura base. 121

4.14. Ahorro en términos de EDP obtenidos gracias a Ω-Test respecto a la arquitectura base. 121

5.1. Caracterización de la geometría de una escena, desglosada en primitivas que caen dentro del *frustum* de la cámara (tanto visibles como ocluidas) y primitivas que caen fuera de éste. 128

5.2. Implementación de Triangle Dropping en una arquitectura TBDR. Las unidades adicionales aparecen sombreadas. 130

5.3. Esquema de la implementación *hardware* y la lógica del Command Matcher y el Command Buffer. 138

5.4. Reducción de geometría usando Triangle Dropping, desglosado en primitivas transparentes y opacas. 145

5.5. Reducción del tráfico a memoria DRAM obtenida por Triangle Dropping, desglosada en tráfico provocado por el Parameter Buffer y tráfico causado por otros tipos de datos. 146

5.6. Desglose de la fracción de accesos al Parameter Buffer que acaban yendo a memoria DRAM. 147

5.7. *Speedup* obtenido por Triangle Dropping sobre una arquitectura TBDR. 147

5.8. Ahorro del consumo energético logrado por Triangle Dropping sobre una arquitectura TBDR. 148

5.9. Ahorro del EDP logrado por Triangle Dropping sobre una arquitectura TBDR. 149

6.1. Arquitectura NUCA convencional con nodos en una red de interconexión con topología de matriz de 2x2. 161

6.2. Esquema de agrupamiento de bloques de texturas en buckets. Este ejemplo muestra un espacio de direcciones formado por 16 bloques (a_0-a_{15}) las cuales están agrupadas (módulo 4) en 4 buckets (b_0-b_3). . 163

6.3. Diagrama de implementación de la Affinity Table considerando en caso de 4 *Fragment Processors* y 8 *buckets*. 163

6.4. Red de interconexión para una implementación DTM-NUCA de 9 nodos, en la que cada nodo consiste en un *Fragment Processor* y su correspondiente Texture Cache local. (a) Esquema centralizado con la Affinity Table en el centro de la red. (b) Esquema distribuido con una Affinity Table por nodo. 168

6.5. Análisis del impacto del tamaño de página sobre los accesos a texturas, desglosados en accesos locales y remotos (normalizados respecto al *baseline*). 169

6.6. Análisis del impacto del tamaño de página sobre los cambios de afinidad. 170

6.7. Análisis del impacto del tamaño de la Affinity Table sobre el rendimiento en términos de accesos locales a texturas (normalizados respecto al *baseline*). 171

6.8. Análisis del impacto de la longitud de época sobre el rendimiento en términos de accesos locales a texturas (normalizados respecto al *baseline*). . . . 172
6.9. Análisis de los cambios de afinidad en función del tamaño de época (normalizados respecto al total de accesos a texturas). . . . 172
6.10. Accesos normalizados a la caché L2 para los diferentes diseños de NUCA con respecto a la arquitectura de referencia (*baseline*). . . . 175
6.11. Accesos a L2 normalizados de DTM-NUCA respecto al *baseline* con diferentes tamaños de Texture Cache. . . . 175
6.12. Accesos locales y remotos a la Texture Cache para las organizaciones NUCA evaluadas. . . . 176
6.13. Aceleración de las organizaciones NUCA evaluadas respecto al *baseline*. . . . 177
6.14. Energía consumida por las organizaciones NUCA evaluadas normalizada al *baseline*. . . . 178
6.15. Escalabilidad de DTM-NUCA para un número de nodos creciente. . . . 178

Índice de tablas

3.1. Parámetros generales de simulación usados en el simulador ciclo a ciclo. 82

3.2. Caracterización de los *benchmarks* evaluados. 83

4.1. Necesidades de almacenamiento de la Ω-Table para los diferentes factores de *coarsening* posibles en un *tile* de 16x16, asumiendo una resolución de pantalla de 1280x720 píxeles. 101

4.2. Parámetros de simulación usados en Ω-Test. 115

5.1. Parámetros de simulación específicos de Triangle Dropping usados para su evaluación. 142

5.2. Calidad de imagen y vídeo de los *benchmarks* evaluados usando las métricas MSSIM y VSSIM. 144

6.1. Parámetros de simulación usados en DTM-NUCA. 173

1

Introducción

El uso de dispositivos móviles tales como teléfonos inteligentes (también conocidos como *smartphones*), tabletas y relojes inteligentes (también conocidos como *smartwatches*) ha ido aumentando a lo largo de los últimos años debido a sus grandes capacidades, facilidad de uso y autonomía. Tanto es así que forman una parte esencial de nuestro día a día. Con el avance de la tecnología, estos dispositivos son cada vez más potentes, pudiendo llegar a desempeñar las mismas funciones que antes eran exclusivamente llevadas a cabo por un computador de sobremesa. En este sentido, una cantidad importante del segmento de los videojuegos se ha desplazado al sector de los dispositivos móviles. Lo que antes era una tarea llevada a cabo únicamente por las videoconsolas, ordenadores de sobremesa o portátiles de gama alta, ahora es posible realizarlas en dispositivos móviles. A este incremento de potencia de cálculo se le suma la constante demanda de los usuarios consumidores de estas aplicaciones, quienes cada vez exigen gráficos más realistas. Esta

evolución ha traído consigo un desafío importante que lleva investigándose durante muchos años, que es el consumo energético, la duración de las baterías y el calentamiento del dispositivo. Todos estos juegan un papel crucial en la experiencia final de usuario, por lo que es importante proponer soluciones eficientes a estos problemas.

Un dispositivo móvil moderno incorpora un SoC (de sus siglas en inglés, *System-on-a-Chip*) que a su vez incorpora todos los componentes de procesamiento necesarios para su funcionamiento, como son la CPU, la GPU, los módulos de banda base (o *baseband*), GPS, WiFi, Bluetooth, o la tarjeta de sonido. Numerosos trabajos de investigación [9, 7, 22, 65, 62] señalan a la GPU como uno de los mayores contribuidores en el consumo de energía de un SoC móvil, por lo tanto, es importante mejorar la eficiencia de ésta.

Para mejorar la eficiencia energética de una GPU móvil es necesario conocer su funcionamiento básico. La interfaz mediante la que se programa una GPU es una API (de sus siglas en inglés, *Application Program Interface*). Ésta define cómo se van a renderizar las escenas en la pantalla del dispositivo, es decir, los procedimientos internos para lograr tal objetivo. De este modo, se define una capa de abstracción entre el dispositivo *hardware* y el programador, lo que implica una mayor comodidad de programación y compatibilidad con una gran variedad de implementaciones de GPUs. Usando esta interfaz el programador dibuja una escena sin importar cómo la GPU lo va a realizar internamente.

Las APIs gráficas más conocidas son OpenGL [78] (de Khronos Group), Vulkan[77] (también de Khronos Group) y Direct3D[1] (de Microsoft). En el caso concreto de OpenGL, un sistema gestor de gráficos crea un contexto, que no es más que una máquina de estados que el programador manipula y configura para renderizar escenas. Dentro de este contexto se ejecuta una secuencia de comandos que modifica esa máquina de estados. Existen dos tipos de comandos: los comandos de estado y los comandos de dibujado o llamadas de dibujado (conocidas generalmente como *draw calls*). Los primeros son aquéllos que modifican y preparan la máquina de estados para la renderización de la escena, pero que no disparan dicho procedimiento.

[1]Perteneciente al conjunto de APIs de DirectX, también propiedad de Microsoft.

Estos comandos pueden ser, por ejemplo, una transferencia de memoria de la CPU a la GPU (cargar modelos, texturas, programas...), la compilación de un programa o la definición de funciones de dibujado. Por otro lado, los comandos de dibujado disparan el proceso de renderización propiamente dicho. Éstos son los que más tiempo de cómputo requieren puesto que activan los componentes de la GPU para dibujar la escena en la pantalla. Generalmente, este tipo de comandos consiste en dibujar un modelo (u objeto). Un modelo está compuesto por primitivas, que a su vez están compuestas por vértices, que a su vez están compuestas por atributos, como pueden ser la posición, el color o las normales de cada uno de los vértices. Al conjunto de primitivas de una escena se le denomina *geometría*. Cada primitiva da lugar a un conjunto de elementos del tamaño de un píxel denominados fragmentos, que contienen una interpolación de los atributos de los vértices a los que pertenecen. Para dar más realismo a las primitivas que se dibujan, generalmente se hace uso de una o más texturas, de modo que los fragmentos resultantes usan esta información para determinar su color final. A este proceso se le denomina texturizado. Una textura es una imagen generalmente de 2 dimensiones compuesta por elementos del tamaño de un píxel, denominados *texels* (del inglés, *Texture Elements*). Tras el proceso de texturizado, y otros procesos más, se obtiene finalmente el color del fragmento, que será mostrado en la pantalla. En la Sección 2.1 se mostrarán más detalles sobre el proceso de renderizado, sus principales etapas y sus implementaciones más típicas.

El objetivo principal de esta Tesis es proponer, implementar y evaluar técnicas microarquitectónicas diseñadas específicamente para GPUs móviles con el fin de mejorar la experiencia de usuario de un dispositivo móvil, haciendo uso para ello de técnicas *hardware* que maximicen la vida de la batería al mismo tiempo que se aumenta el rendimiento. De este modo, se pretende mejorar la eficiencia energética de estos dispositivos. Las técnicas propuestas en esta Tesis están enfocadas a mejorar la eficiencia energética de los dispositivos móviles, centrándose principalmente en la GPU como mayor contribuidor del consumo energético de un dispositivo móvil que ejecuta aplicaciones gráficas, es decir, durante la ejecución de videojuegos. Al mismo tiempo, las técnicas propuestas permitirán mejorar el rendimiento de una GPU móvil, mejorando así considerablemente la experiencia de usuario.

Para lograr estos objetivos, estas técnicas se centran en eliminar principalmente el trabajo inútil, es decir, eliminar procedimientos internos que no afectan (o afectan muy poco) a la imagen final que se visualiza en la pantalla del dispositivo, que es lo que el usuario percibe. Por otro lado, se pretende aumentar el rendimiento mejorando la organización y comportamiento de los componentes internos de la GPU, así como la jerarquía de memoria en la que se apoya. Uno de los pilares fundamentales sobre los que se basa esta Tesis es la coherencia entre fotogramas, también conocida como coherencia *frame-to-frame*. Esta característica inherente de los videojuegos establece que dos fotogramas de una secuencia consecutivos en el tiempo se asemejan. Esto significa que si dos imágenes son parecidas, los cálculos realizados para generarlas también lo serán y, por lo tanto, es factible la reutilización de ciertos cálculos (o conocimientos sobre lo que ha ocurrido durante la renderización de un fotograma) a la hora de procesar el fotograma siguiente, reduciendo así la carga de trabajo de la GPU.

1.1. Problemas detectados y motivación

La motivación principal de esta Tesis es la existencia de ciertos patrones en la forma de dibujar escenas en los videojuegos estudiados que se han visto como una oportunidad de mejorar el rendimiento y reducir el consumo energético de la GPU. En el caso de este último, se pueden identificar dos grandes contribuidores en el consumo energético, que son los procesadores de fragmentos (véase Sección 2.1.8) y el subsistema de memoria DRAM. La Figura 1.1 muestra un desglose del consumo energético de los componentes de una GPU que más energía gastan. Como se puede observar, el subsistema de memoria consume más de la mitad (53 %), llevándose la mayor parte del consumo. Con un 42 % del consumo energético le siguen los procesadores de fragmentos y, el otro 5 % restante se corresponde con el resto de componentes (caché L2, cachés de primer nivel, procesadores de vértices, etc.).

Por otra parte, se ha observado que gran parte de las escenas dibujadas presentan elementos que no se visualizan en la imagen final del dispositivo, lo que da lugar a que la GPU realice trabajo inútil y, por tanto, derive en un consumo energético innecesario. En la mayoría de los casos, este problema es muy difícil

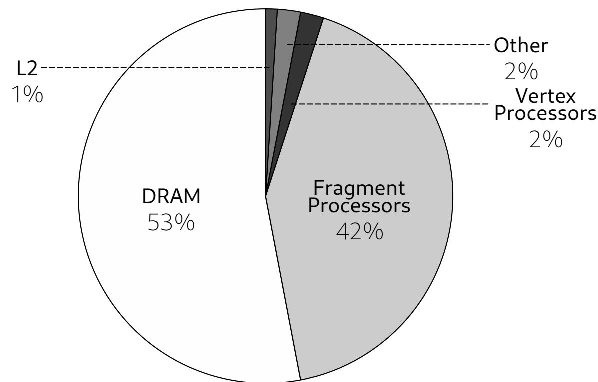


Figura 1.1: Desglose del consumo energético de los componentes de una GPU.

de resolver desde el punto de vista del programador puesto que no dispone de la información necesaria para evitarlo, e intentar resolverlo mediante mecanismos *software* requeriría de una cantidad de recursos y procesamiento extra que haría inviable tal optimización. En otras ocasiones, este escenario se presenta por una serie de malas prácticas que se desvinculan de las recomendaciones, o bien del fabricante de GPUs, o bien de la API que se usa. Cuanto más compleja es una escena, más difícil es evitar estas situaciones. Es por esto que existen en la literatura numerosas propuestas enfocadas a ahorrar este tipo de regiones de trabajo inútiles de manera programática. En las siguientes secciones se describen los principales problemas observados y las soluciones que proponemos para combatirlos y mitigarlos.

1.1.1. Fragmentos ocluidos *(overdraw)*

El hecho de que una escena sea compleja acarrea consigo que se visualice una gran cantidad de primitivas por lo que, por pura estadística, es muy probable que haya primitivas que se solapen en la pantalla. Esto tiene como consecuencia que habrá fragmentos que se solapen, siendo visibles únicamente aquéllos que están más cerca de la cámara. A este fenómeno se le conoce como *overdraw* (o sobredibujado) porque implica que en la pantalla se están dibujando más fragmentos de los necesarios para obtener la misma imagen. O dicho de otro modo, para un píxel concreto de la imagen final, se ha dibujado más de un fragmento. El grado de *overdraw* dependerá de la cantidad de fragmentos que haya ocultos en la escena. De

este modo, se define el factor de *overdraw* como la cantidad de fragmentos ocultos respecto al total. Intentar eliminar fragmentos ocultos de manera programática es muy costoso por lo que generalmente se hace a un grano más grueso, esto es, a nivel de primitiva o incluso de objeto.

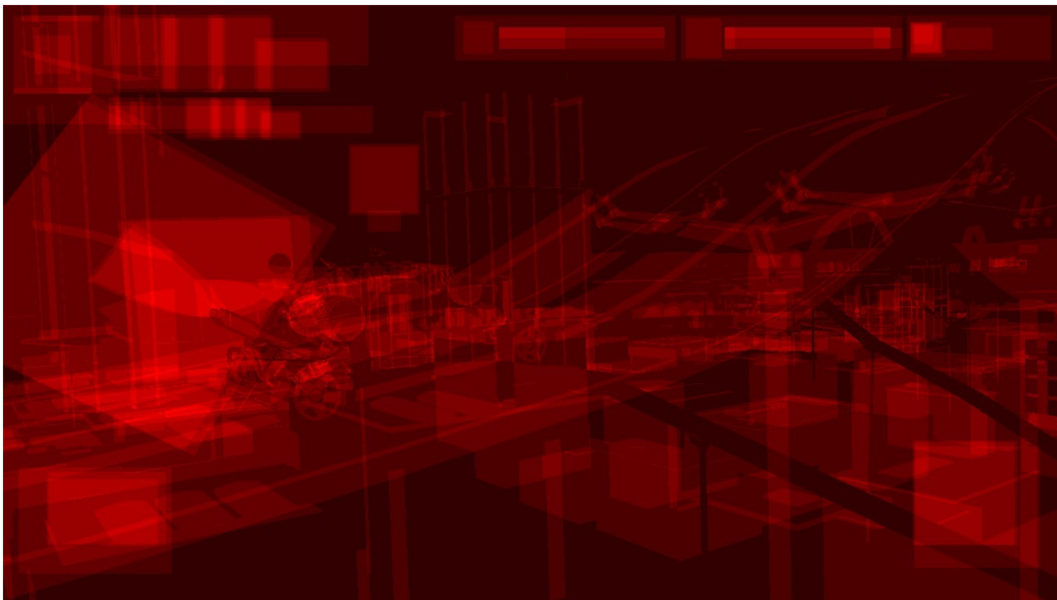


Figura 1.2: Mapa de calor representando el *overdraw* de un fotograma del juego Hot Wheels. Las áreas más oscuras corresponden con un factor más bajo de *overdraw*; mientras que las zonas más claras corresponden con un mayor solapamiento de primitivas y, por tanto, a un mayor grado de *overdraw*.

En la Figura 1.2 se puede observar la cantidad de *overdraw* en una escena, representado como un mapa de calor, de uno de los videojuegos que se han estudiado en esta Tesis.

Para hacer frente al problema del *overdraw*, en esta Tesis se ha propuesto la técnica Ω-Test que será detallada en el Capítulo 4. A grandes rasgos, en lugar de partir de cero, Ω-Test aprovecha la información de profundidad (almacenada en el denominado Z-Buffer) del fotograma anterior con el fin de eliminar fragmentos de una manera más temprana, reduciendo así considerablemente el *overdraw* de la escena.

1.1.2. Geometría ocluida

Como se ha mencionado antes, durante el proceso de renderizado de una escena compleja se dibujan muchas primitivas pero sólo unas cuantas son visibles finalmente en la pantalla que el usuario visualiza, mientras que el resto, o están completamente ocluidas por otras, o directamente están fuera de la pantalla. A este fenómeno se le conoce como *geometría ocluida*, que aparte de causar un posible *overdraw*, también acarrea un gasto no despreciable en el procesamiento de la geometría por parte de la GPU. Por lo tanto, si se consigue reducir o eliminar directamente el procesamiento de estas primitivas, disminuirá el consumo energético y aumentará el rendimiento.

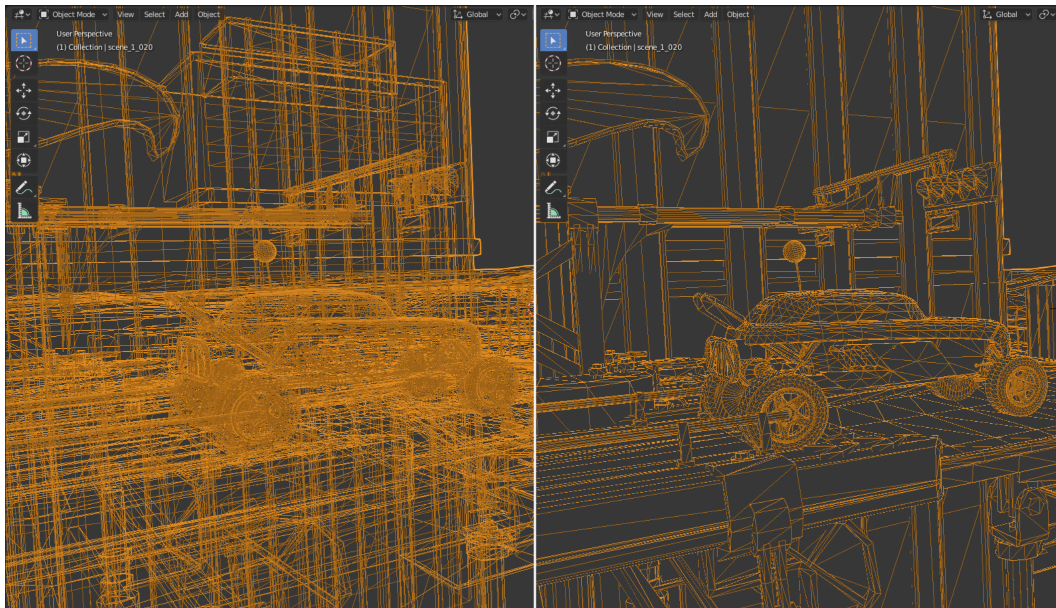


Figura 1.3: Fotograma de una escena del juego Hot Wheels renderizada en modo *wireframe* donde se observa la cantidad de geometría ocluida. A la izquierda, toda la geometría procesada que cae dentro de la pantalla (27554 triángulos en total); a la derecha, sólo la geometría que es visible finalmente (tan sólo 10439 triángulos), excluyendo aquellas primitivas que están completamente ocluidas.

La Figura 1.3 muestra una escena, también del juego Hot Wheels, donde se puede observar la cantidad de geometría ocluida. Gracias al renderizado en *wireframe* se puede observar fácilmente la cantidad de primitivas ocultas. En esta imagen, de un total de 27554 triángulos, tan sólo se visualizan en la pantalla 10439.

Al igual que pasa con el *overdraw*, a mayor complejidad de una escena, mayor cantidad de geometría ocluida habrá. Eliminar esta cantidad de trabajo inútil a nivel *software* no es tan complejo como eliminar *overdraw* debido a su gruesa granularidad, pero aun así requiere de una información costosa de obtener. Por otro lado, existen soluciones híbridas que combinan tanto información *software* como la proporcionada por el *hardware* para hacer frente a este problema.

En esta Tesis se propone la técnica denominada *Triangle Dropping*, detallada y evaluada en el Capítulo 5, que aborda el problema de la geometría ocluida y es puramente *hardware*, no siendo necesaria la intervención del programador ni que la aplicación tenga que preocuparse de esta tarea. La idea que subyace detrás de esta técnica es el aprovechamiento de la visibilidad de las primitivas en el fotograma anterior. En este sentido, y dado que existe coherencia entre fotogramas, las primitivas que fueron ocluidas en el fotograma previo, es muy probable que también lo sean en el actual. Así pues, se implementa un mecanismo capaz de capturar la visibilidad de las primitivas de un fotograma para usarla en el siguiente y eliminar así las primitivas que fueron ocluidas. Es importante destacar que esta segunda propuesta trabaja a nivel de primitiva y, por tanto, actúa de forma mucho más temprana en el *pipeline* gráfico, esto es, a nivel de geometría de la escena, a diferencia de Ω-Test que actúa a nivel de fragmento en una etapa más tardía.

1.1.3. Replicación de texturas

A medida que aumenta el número de nodos de un sistema multicore aumenta la replicación entre sus cachés privadas. Esto es algo natural conforme aumenta el grado de compartición y los datos que referencia cada uno de estos núcleos están cercanos en el espacio. Este aspecto no tiene que ser necesariamente negativo para el rendimiento de un sistema multicore, pues ayuda en gran medida a reducir la contención en niveles superiores de la jerarquía de memoria, al mismo tiempo que disminuye la latencia de acceso medio al subsistema de memoria. Sin embargo, y a pesar de este aumento en la capacidad del nivel privado, la mayor parte está ocupada por bloques replicados que podrían tener, en su lugar, otros bloques del *working set* del procesador al que pertenece.

En el caso de las GPUs, existe mucha replicación de datos en las cachés de primer nivel de los denominados *Fragment Processors* (son los núcleos de procesamiento encargados de renderizar cada uno de los fragmentos que componen la escena, como se explicará en la Sección 2.2.1) puesto que éstos también acceden a ellas de manera privada sin compartir datos. Esta replicación se produce porque generalmente estos procesadores acceden a los mismos bloques de texturas en el tiempo. La Figura 1.4 muestra una idea de la cantidad de replicación que existe en una GPU con cuatro *Fragment Processors*. Aquí se puede observar que casi la mitad (el 49.19 %) de los bloques de texturas se replican en las cuatro cachés, lo que significa una reducción de la capacidad efectiva en un factor de 4 para estos bloques. Esto obviamente es un escenario muy negativo para el rendimiento.

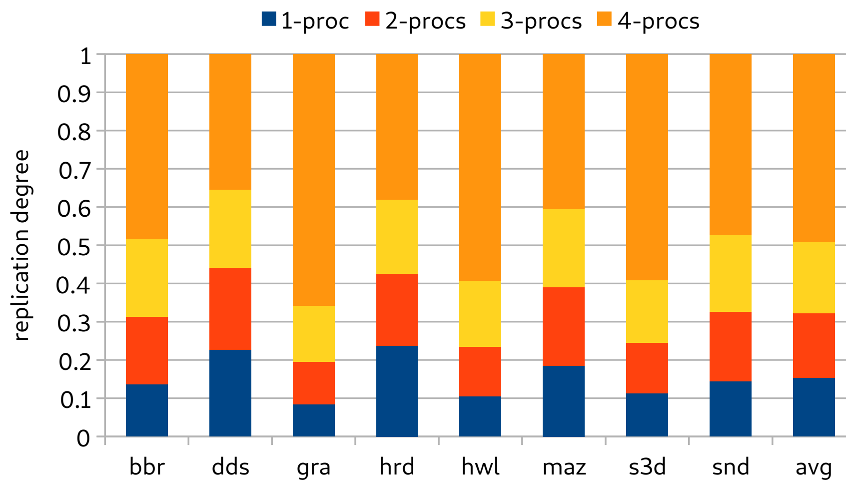


Figura 1.4: Grado de replicación de texturas en los *benchmarks* estudiados para una GPU con 4 *Fragment Processors*.

Para superar este problema, en esta Tesis se propone la técnica denominada DTM-NUCA y que se expone en el Capítulo 6. A grandes rasgos, DTM-NUCA es un esquema NUCA diseñado exclusivamente para las cachés de texturas de los procesadores de fragmentos de una GPU, que permite compartir bloques de texturas entre cores reduciendo así el grado de replicación y aumentando la capacidad efectiva del conjunto de cachés de texturas.

1.2. Contribuciones de la Tesis

Como se ha mencionado anteriormente, para hacer frente a los problemas descritos en la Sección 1.1, se han propuesto en esta Tesis diferentes técnicas a nivel de microarquitectura diseñadas a medida para las GPUs móviles. En esta Sección describiremos brevemente estas técnicas.

1.2.1. Ω-Test

Ω-Test es una técnica enfocada a reducir el *overdraw* haciendo uso de la coherencia entre fotogramas. La idea es aprovechar el Z-Buffer del fotograma previo para el fotograma actual, de modo que el Early Z-Test pueda eliminar más fragmentos al estar más actualizado su Z-Buffer. En una GPU convencional, esta información se desecha de un fotograma a otro, lo que provoca que se tenga que construir desde cero cada vez que comienza el procesamiento de cada nuevo fotograma. Por otro lado, si se aprovecha dicha información del Z-Buffer en bruto puede llevar a errores puesto que, a pesar de ser muy parecidos, no son exactamente iguales, lo que llevará al Early Z-Test a descartar fragmentos que no debía. Para evitar dicha situación, Ω-Test incorpora un mecanismo de corrección de errores que permite recuperar la información de la imagen que ha sido determinada de manera especulativa debido al uso de esta información del fotograma previo, y que se explicará más adelante. Dicho mecanismo es liviano en *hardware*, lo que incurre en una sobrecarga muy baja. Es importante aclarar que, en todo momento, las imágenes que produce Ω-Test son idénticas a las originales, es decir, no se producen errores finales en la imagen.

En definitiva, el mecanismo propuesto Ω-Test obtiene una mejora de rendimiento de 16.3 %, una reducción del EDP de 26.42 %, y una reducción del consumo de energía de 15.17 %.

1.2.2. Triangle Dropping

Triangle Dropping es una técnica para GPUs móviles enfocada a reducir la geometría ocluida por medio del aprovechamiento de la coherencia entre fotogra-

mas, reduciendo así el consumo energético y aumentando el rendimiento general. La motivación principal de esta técnica reside en el hecho de que, para nuestro conjunto de *benchmarks*, el 60 % de las primitivas que están dentro de la escena están completamente ocluidas por otras primitivas. Incluso habiendo una caché para el Parameter Buffer[2] (la Tile Cache), alrededor del 46 % de los accesos a DRAM son provocados por este *buffer*[3]. Triangle Dropping aprovecha la información de la visibilidad computada durante el Raster Pipeline, para eliminar de manera prematura primitivas en el fotograma siguiente. Como es lógico, la información de visibilidad de las primitivas no tiene por qué ser la misma de un fotograma a otro debido a que hay continuos cambios en la escena. Es por eso que este mecanismo puede provocar errores en la imagen final. Para mitigarlos, Triangle Dropping usa una serie de mecanismos conservativos que mantienen la calidad de imagen en rangos aceptables para el ojo humano. Respecto a la mejora que supone, Triangle Dropping consigue un aumento del rendimiento de 20.2 %, una reducción media del consumo energético de 14.5 %, y una reducción del EDP de 28.2 %.

1.2.3. DTM-NUCA

DTM-NUCA (de sus siglas Dynamic Texture Mapping-NUCA) es un esquema NUCA[4] diseñado para las cachés de texturas privadas (L1) de una GPU móvil, enfocada a incrementar la capacidad efectiva general y a disminuir la latencia media de accesos atacando la replicación de datos. De este modo, un bloque que no se encuentre en una caché de texturas local (L1) se puede servir por una caché de texturas remota (esto es, asociada a otro *Fragment Processor*) que lo contenga a un coste mucho menor que el de acceder a la caché compartida L2 (que se corresponde con la LLC en una GPU convencional).

Para ello, DTM-NUCA incorpora una pequeña tabla de mapeo dinámico, llamada Affinity Table, cuyo tamaño es totalmente independiente del tamaño de la caché compartida L2, a diferencia de una organización NUCA convencional en la

[2] Esta estructura se almacena en memoria DRAM y se encarga de almacenar la geometría procesada. Para más detalles, véase Sección 2.2.2.

[3] Para los *benchmarks* evaluados, la tasa de fallos media de la Tile Cache es del 80.81 % para las escrituras y del 26.17 % para las lecturas.

[4] Non-Uniform Cache Access.

que el directorio debe almacenar el estado de cada bloque de la caché L2. En nuestro caso, el mejor candidato a propietario para un determinado conjunto de bloques se determina de manera dinámica y se almacena en la denominada Affinity Table para maximizar los accesos locales. Además, este mecanismo permite un cierto grado de replicación para favorecer los accesos locales en casos en los que existe una clara contención, pero sin perjudicar el rendimiento puesto que se pierde muy poca capacidad con dicha replicación. DTM-NUCA se presenta en dos variantes. Una con una Affinity Table *centralizada* y otra con la Affinity Table *distribuida* entre todos los nodos de procesamiento. Esta segunda organización tiene como resultado una reducción de la presión sobre la caché L2 del 41.8 % del total de accesos a dicha caché. Por otro lado, se maximizan los accesos locales en un 73.9 % de media con el fin de disminuir la latencia media de acceso. Como consecuencia, DTM-NUCA consigue obtener una mejora media del rendimiento de 16.9 % y una reducción media del consumo energético de 7.6 % respecto a una organización convencional compuesta por cachés de texturas privadas de primer nivel.

1.2.4. Publicaciones

Finalmente, la realización de esta Tesis ha dado como resultado las siguientes publicaciones (en orden de realización):

- D. Corbalán-Navarro, J.L. Aragón, M. Anglada, E. de Lucas, J.M. Parcerisa y A. González. "Omega-Test: A Predictive Early-Z Culling to Improve the Graphics Pipeline Energy-Efficiency". *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, issue 14, pp. 4375-4388, Diciembre de 2022. [18]

- D. Corbalán-Navarro, J.L. Aragón, M. Anglada, E. de Lucas, J.M. Parcerisa y A. González. "Improving the Energy Efficiency of the Graphics Pipeline by Reducing Overshading". *Actas de las XXXI Jornadas de Paralelismo*, pp. 125-134, Málaga, Septiembre de 2021. [63]

- D. Corbalán-Navarro, J.L. Aragón, M. Anglada, J.M. Parcerisa y A. González. "Triangle Dropping: An Occluded-Geometry Predictor for Energy-efficient

Mobile GPUs". *ACM Transactions on Architecture and Code Optimization*, vol. 19, issue 3, article 39, pp. 1-20, Septiembre de 2022. [19]

- D. Corbalán-Navarro, J.L. Aragón, J.M. Parcerisa y A. González. "DTM-NUCA: Dynamic Texture Mapping-NUCA for Energy-Efficient Graphics Rendering". *Proc. of the 30th IEEE Euromicro International Conference on Parallel, Distributed and Network-based Computing* **(PDP)**, Valladolid, Spain, pp. 144-151, Marzo de 2022. [20]

- J. Ortiz-Escribano, D. Corbalán-Navarro, J.L. Aragón y A. González. "MEG-sim: A Novel Methodology for Efficient Simulation of Graphics Workloads in GPUs". *Proc. of the 2022 IEEE International Symposium on Performance Analysis of Systems and Software* **(ISPASS)**, Singapore, Singapore, pp. 69-78, Mayo de 2022. [21]

- Finalmente, y en proceso de revisión se encuentra: D. Corbalán-Navarro, J.L. Aragón, J.M. Parcerisa y A. González. "A Novel NUCA Organization for Dynamic Texture Mapping in Mobile GPUs". *Microprocessors and Microsystems: Embedded Hardware Design* **(MICPRO)**.

2

Contexto y estado del arte

2.1. El *pipeline* gráfico

En el ámbito de la arquitectura de computadores es muy común encontrarse con el término *pipeline*. Éste hace referencia a cómo el flujo de datos de entrada de un sistema es procesado hasta generar la salida deseada. Un *pipeline* está compuesto por varias etapas que transforman los datos de entrada. Cada etapa consta de una o varias entradas y una salida que irá a la etapa siguiente estableciendo una cadena. Idealmente, cada etapa hace una función determinada, y actúa como una caja negra, es decir, su implementación interna es indiferente siempre y cuando genere la salida deseada. La idea que subyace detrás de este término es que si conseguimos dividir un proceso en múltiples etapas, se puede mejorar el rendimiento global del sistema dado que la ejecución de éstas se puede llevar a cabo simultáneamente en el tiempo, solapando tareas.

El objetivo final del *pipeline* gráfico es obtener una imagen en pantalla a partir de representaciones de modelos situados en una escena. Es decir, los datos de entrada son modelos que representan una escena, y la salida es una imagen en la pantalla del dispositivo que visualiza el usuario. A este proceso se le conoce con el nombre de proceso de *renderización* o *renderizado*. Se puede ver también como una caja negra que recibe como entrada modelos de objetos y escenas (definidos en un formato específico) y obtiene como salida una imagen plana bidimensional formada por colores[1]. En esta Sección cubriremos los aspectos relacionados con el funcionamiento lógico de los componentes que forman el *pipeline* gráfico así como las configuraciones más típicas del mismo y sus consecuencias.

Para poder entender el proceso de renderización de una escena, es necesario entender primero qué es un modelo de un objeto. Un modelo (o malla) de un objeto es un conjunto ordenado de vértices que forman primitivas, y está generalmente almacenado en la memoria principal de la GPU. Por ejemplo, si queremos definir un modelo que consista simplemente en un hexágono, podríamos hacerlo definiendo 6 vértices y una primitiva que contenga esos 6 vértices. Así pues, una primitiva consiste en un conjunto de vértices previamente definidos. Por lo tanto, el requisito mínimo para representar un modelo es que contenga vértices y primitivas. Existen muchas maneras de representar estructuralmente un modelo. En OpenGL, por ejemplo, hay dos formas[2]: mediante un único buffer de vértices (denominado *Vertex Buffer Object*, o simplemente VBO); o mediante un VBO más un búffer de elementos (denominado *Element Buffer Object*, o simplemente EBO), que forman a su vez primitivas. En el primer caso, no hay una estructura explícita que defina las primitivas, por lo que éstas se definen de manera implícita sobre la marcha a medida que se definen los vértices. Esto tiene un problema, y es que si más de una primitiva comparte vértices con otra, éstos han de definirse varias veces, tantas como primitivas haya usando ese vértice. La Figura 2.1 muestra este escenario, en el que un modelo compuesto por 5 triángulos repite los vértices comunes entre éstos.

[1]Realmente se puede generar más de una imagen de salida. Por ejemplo, en OpenGL se pueden generar hasta 4 imágenes (que se denominan *framebuffers*)

[2]Existe una tercera forma de definir un modelo en OpenGL, que es mediante el uso de *Vertex Array Object* (o VAO). Esta representación combina VBO y EBO en una única estructura y, además, incluye el formato de los atributos que se le van a pasar a los *shaders*.

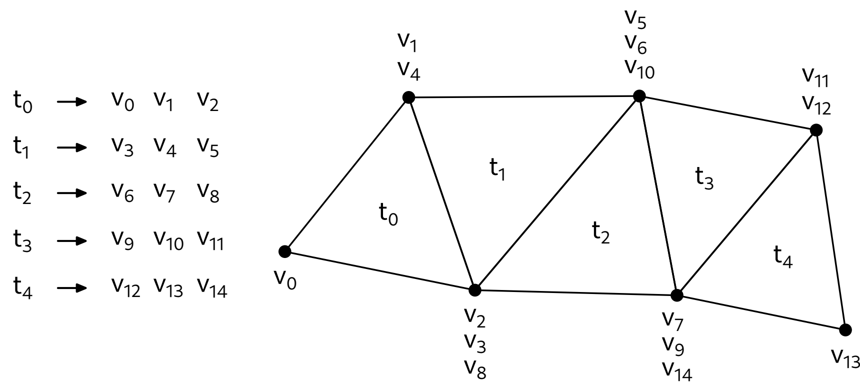


Figura 2.1: Modelo definido por índices implícitos.

En la segunda forma, el EBO define las primitivas del modelo de manera explícita. Cada elemento de este *buffer* contiene un puntero a un vértice del VBO en forma de índice. De este modo, las primitivas que tienen en común un vértice sólo tienen que contener un elemento que apunte a un vértice que se ha definido una única vez, optimizando así el espacio. Es importante destacar que lo que finalmente determina las primitivas de un modelo son los elementos del EBO y no los vértices del VBO. Dicho de otro modo, si hay algún vértice en el VBO que no es referenciado por ningún elemento del EBO, dicho vértice no se visualizará en el modelo final. En el caso de modelos de objetos que se definen únicamente mediante el VBO, los elementos se corresponden con los vértices del VBO en el mismo orden. Es decir, si tenemos un VBO formado por n vértices, de manera implícita tendrá un EBO con n elementos en el que el elemento i-ésimo apunta al vértice i del VBO.

Como ya se ha mencionado, una primitiva está formada por un conjunto de vértices. Existen tres *primitivas base*, a partir de las cuales se pueden obtener todas las primitivas imaginables. Éstas son: el punto, la línea y el triángulo. El *punto* consiste únicamente en un vértice. La *línea* consiste en dos vértices que se unen para formar un segmento recto. El *triángulo* consiste en tres vértices. Para generar una primitiva a partir de vértices (o índices a éstos), es necesario definir el tipo de primitiva que va a usar el modelo. El tipo de primitiva determina la forma en la que los vértices van a generar las primitivas del modelo.

La Figura 2.2 muestra los diferentes tipos de primitiva que podemos encontrarnos en OpenGL.

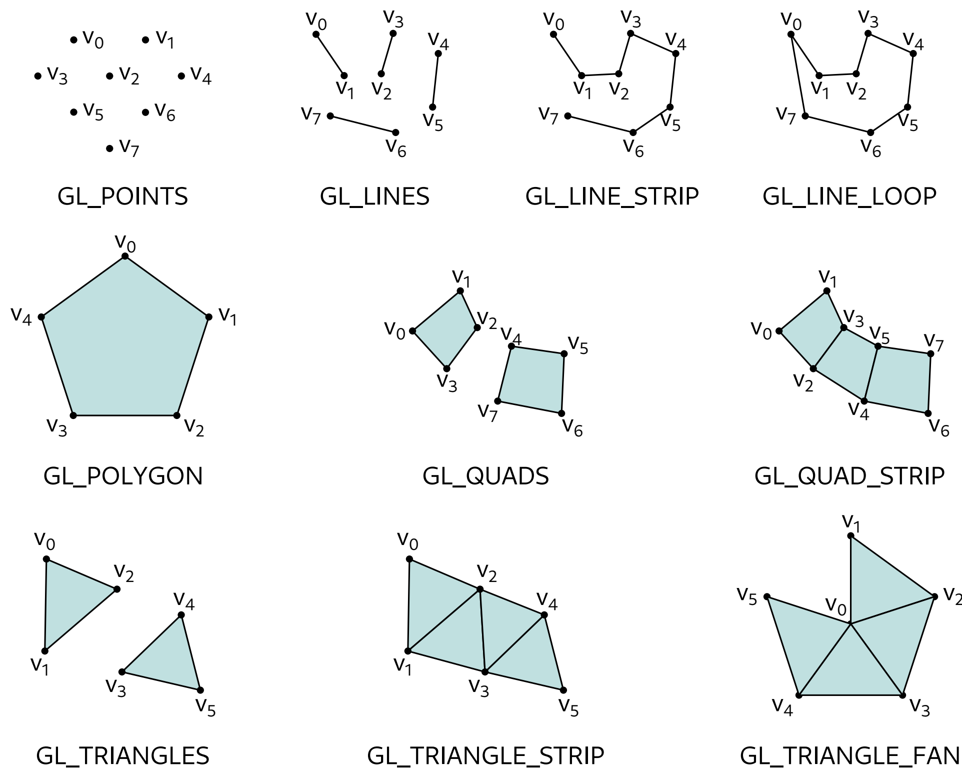


Figura 2.2: Tipos de primitivas en OpenGL.

El tipo de primitiva más simple que se puede definir para un modelo es el *punto* (`GL_POINTS`). Este tipo de primitiva lo único que hace es generar un punto por cada elemento del EBO del modelo.

Dentro de la primitiva base de la *línea*, encontramos tres formas compuestas: líneas sueltas (GL_LINES), la tira de líneas (`GL_LINE_STRIP`) y el bucle de líneas (`GL_LINE_LOOP`). La primera genera una línea por cada par de vértices. La segunda genera una línea con los dos primeros elementos del EBO. Luego, por cada elemento i-ésimo, genera otra línea formada por el vértice $i-1$ y el vértice i. La tercera es igual, pero cierra el bucle de líneas juntando el vértice $n-1$ con el vértice 0.

Dentro de la primitiva base *triángulo* encontramos varios tipos de primitiva compuesta, a saber: `GL_POLYGON`, `GL_QUADS`, `GL_QUAD_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` y `GL_TRIANGLE_FAN`. La primitiva `GL_POLYGON` define un polígono convexo que va desde el primer vértice hasta el último, cerrando el

bucle con el primer vértice. Este tipo de primitiva se declaró como obsoleta a partir de la versión 3.1 de OpenGL (core) porque estaba muy limitada[3], generaba artefactos en las escenas y podía ser reemplazada fácilmente por `GL_TRIANGLE_FAN`. El tipo `GL_QUADS` forma un cuadrángulo por cada 4 elementos seguidos del EBO. El tipo `GL_QUAD_STRIP` también forma primitivas en cuadrángulos, pero de manera enlazada. De este modo, los cuatro primeros elementos del EBO forman el primer cuadrángulo, después, por cada dos elementos i e $i+1$ del EBO se forma otro cuadrángulo con los vértices $i-2$, $i-1$, i e $i+1$. El tipo `GL_TRIANGLES` forma un triángulo por cada 3 elementos del EBO. Es el tipo de primitiva más usado por su flexibilidad y potencia, pues con éste se pueden generar todas las formas geométricas (exceptuando formas curvadas, que deberán ser trianguladas en su caso). El tipo `GL_TRIANGLE_STRIP` también genera triángulos, pero de manera enlazada. De este modo, el primer triángulo del modelo se forma con los tres primeros elementos del EBO, después, por cada vértice i del EBO, se forma otro triángulo compuesto por los elementos $i-2$, $i-1$ e i. Finalmente, el tipo de primitiva `GL_TRIANGLE_FAN` genera también primitivas en forma de triángulos, también de manera enlazada, pero esta vez, todos los triángulos tienen como vértice común el apuntado por el primer elemento del EBO. De este modo, el primer triángulo se forma con los tres primeros elementos del EBO, y después, por cada elemento i, se genera otro triángulo formado por los vértices 0, $i-1$ e i.

Los vértices son estructuras de datos cuya definición depende del programador. Un vértice no es más que un conjunto de atributos que lo definen, como por ejemplo la posición, el color, la normal o las coordenadas de textura. Además, el programador puede definir más atributos que le puedan ser de utilidad a la hora de dotar de efectos al objeto a dibujar. Por ejemplo, es muy común usar animaciones en modelos de personas, animales o criaturas, que al moverse cambian sus vértices de posición, pero no todos se mueven la misma cantidad. Por lo tanto, un atributo muy útil a la hora de hacer estas transformaciones es el peso de cada vértice a la hora de deformar el modelo. Nótese que los atributos pueden ser desde un escalar hasta un vector de cuatro elementos. Por ejemplo, la posición está definida en un espacio generalmente tridimensional, por lo que se usan las componentes `xyz`.

[3] Pues sólo era posible dibujar formas convexas.

Para los colores, es muy común usar las tres componentes básicas `rgb` (rojo, verde y azul) y, aparte, una componente que define el grado de transparencia (comúnmente llamada *alpha*), por lo que generalmente el color se define como un vector de cuatro elementos (`rgba`). Obviamente, ni todas las posiciones están restringidas a 3 componentes, ni todos los colores a 4 componentes, pues estas decisiones las toma el programador.

La Figura 2.3 muestra la estructura de un modelo en el que los vértices tienen 4 atributos, y las primitivas están formadas por 4 vértices (usa un tipo de primitiva `GL_QUADS`). Una vez ya tenemos una representación de los modelos de objetos, éstos son procesados por el *pipeline* gráfico, que consta de dos fases: el *Geometry Pipeline* y el *Raster Pipeline*. La primera, también denominada como fase de geometría, consiste en procesar la geometría de los modelos de la escena, de modo que todos los objetos son proyectados en la pantalla (bidimensional) del dispositivo. La segunda fase consiste en calcular el color de cada uno de los píxeles de la imagen final a partir de la geometría ya proyectada en la pantalla. La Figura 2.4 muestra las etapas de las que se compone cada fase del *pipeline* gráfico.

2.1.1. Procesamiento de los vértices de la escena

La primera etapa del *Geometry Pipeline* es el procesamiento de los vértices (etapa *Vertex Processing*) de entrada de cada modelo de la escena. Aquí los atributos de los vértices son manipulados por medio de un programa denominado *vertex shader*, definido por el programador. Así pues, esta etapa es programable. Un *vertex shader* recibe como entrada un vértice (definido por sus atributos) y una serie de

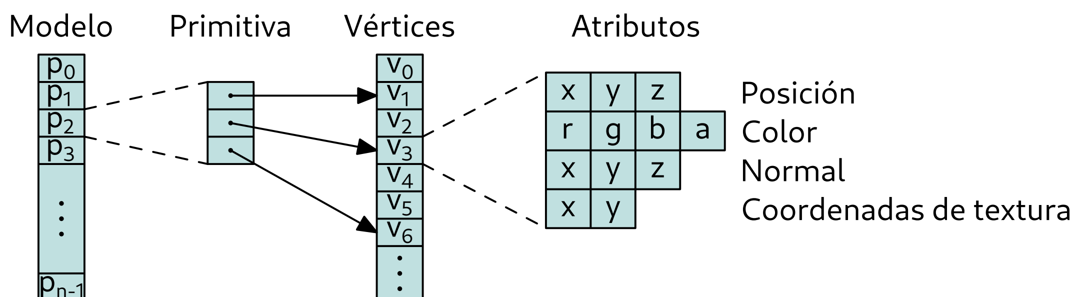


Figura 2.3: Representación estructural de un modelo o malla.

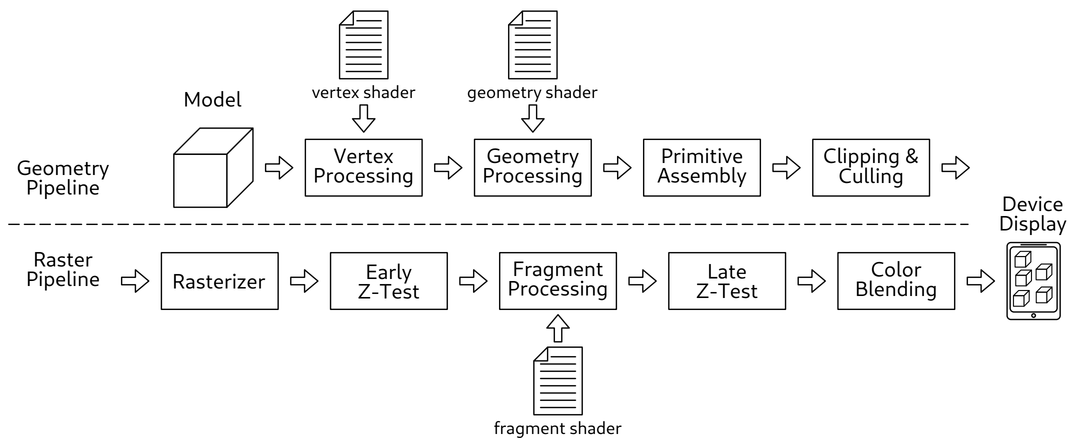


Figura 2.4: Esquema del *pipeline* gráfico.

argumentos opcionales denominados *uniforms* o uniformes que se envían desde el programa principal de la CPU, y da como salida la posición del vértice como un vector de 4 componentes[4] y una serie de atributos, que no tienen por qué ser los mismos que los del vértice original de entrada. Por otro lado, los *vertex shaders* tienen una serie de variables implícitas (*built-in variables*) que se pueden usar dentro del programa (como por ejemplo, el índice del vértice que se está procesando). La Figura 2.5 muestra un esquema de caja negra de un *vertex shader*. En este ejemplo se pueden apreciar tres componentes esenciales, a saber: el vértice de entrada, el *vertex shader* y el vértice de salida. Como puede observarse en este ejemplo, el vértice de entrada tiene dos atributos: uno de posición y otro de coordenadas de textura; denominados `pos`, que tiene dos canales y `tex` que tiene 3 canales, respectivamente. Cuando el *vertex shader* procesa este vértice, se genera el vértice de salida que, como puede observarse, es muy diferente en varios aspectos. En primer lugar, éste tiene dos atributos nuevos respecto al original, uno para determinar el color con 4 canales, y otro para definir un vector normal con 3 canales, denominados `col` y `norm` respectivamente. Además, la posición ha cambiado a `pos´`. También puede apreciarse que se ha descartado el atributo de coordenadas de textura.

El lenguaje de programación de los *vertex shader* depende de la API que se use. Por ejemplo, en OpenGL se denomina GLSL (de *GL Shader Languaje*). Hay que

[4]En realidad, en OpenGL hay dos más (`gl_PointSize` y `gl_ClipDistance`), pero no son tan importantes como la posición.

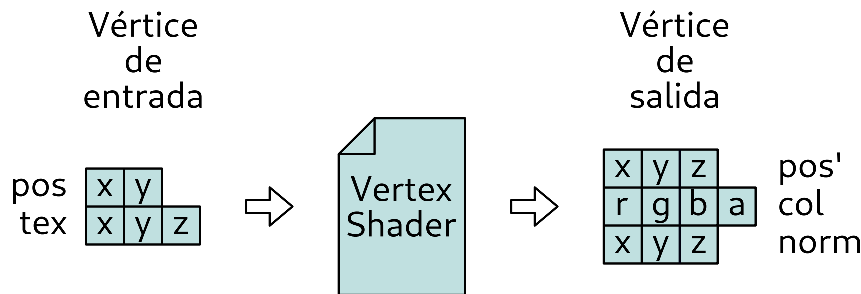


Figura 2.5: Modelo de caja negra de un *vertex shader*.

tener en cuenta que tanto los atributos de entrada como las *uniforms*, así como los atributos de salida del *vertex shader* son opcionales. Lo único que es obligatorio es la posición de salida del vértice, que está definida como un vector de 4 componentes (`xyzw`). Así pues, el *vertex shader* se encarga de manipular los atributos de entrada. En esta Sección, sólo cubriremos la parte de las transformaciones de la posición de los vértices.

Transformaciones geométricas

El objetivo de la etapa *Vertex Processing* es proyectar los vértices de los modelos en la pantalla bidimensional del dispositivo. Para ello, las posiciones de los vértices tienen que pasar por una serie de transformaciones geométricas. Existen muchas maneras de transformar un vértice, pero en esta Sección lo haremos mediante el uso de matrices. Este método es ampliamente usado en lenguajes de programación de *shaders* (como GLSL) porque se adapta bien al modelo computacional de la arquitectura de una GPU pues, al final, realizar operaciones con matrices es computacionalmente sencillo y fácilmente paralelizable. Otra ventaja que ofrecen las matrices es que una única matriz puede agrupar las transformaciones de muchas matrices. Esto permite realizar múltiples transformaciones geométricas mediante una simple multiplicación de una matriz por un vector.

Lo primero que hay que hacer es definir la posición de un vértice. Aunque realmente la posición de un vértice en el espacio se define con un vector de tres dimensiones xyz, en este caso vamos a añadir una cuarta dimensión que denominaremos w. Esta componente extra tiene su explicación y, como veremos más adelante, tiene

que ver con la proyección, el *clipping* y la rasterización. La Ecuación 2.1 muestra la definición de la posición de un vértice en el espacio.

$$\vec{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix} \tag{2.1}$$

Así pues, para transformar un vértice necesitamos multiplicarlo por una matriz de 4x4 elementos como la que se define en la Ecuación 2.2.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \tag{2.2}$$

Es importante notar el orden en el que se realizan dichas transformaciones, pues al contrario de lo que se puede pensar por intuición, éstas se realizan de derecha a izquierda. Por ejemplo, si tenemos tres matrices de transformación, a saber: A, B y C, un vector de posición $\vec{p}$, y queremos obtener un vector $\vec{q}$ cuyo resultado sea el de aplicar primero la transformación A, luego B y luego C, entonces la operación sería la que se muestra en la Ecuación 2.3.

$$\vec{q} = C \cdot B \cdot A \cdot \vec{p} \tag{2.3}$$

Por supuesto, como se ha mencionado anteriormente, esta operación de transformación se puede realizar juntando todas las transformaciones en una matriz D y luego multiplicándola por el vector a transformar, tal y como se muestra en la Ecuación 2.4.

$$\begin{aligned} D &= C \cdot B \cdot A \\ \vec{q} &= D \cdot \vec{p} \end{aligned} \tag{2.4}$$

Un objeto pasa por varios sistemas de coordenadas antes de ser renderizado. Para convertir de un sistema a otro se usan transformaciones que, como hemos visto, se pueden definir mediante matrices. Antes de ver los diferentes sistemas de coordenadas por los que pasa un objeto, vamos a ver los tipos de transformación que se pueden realizar mediante matrices[5]. Las transformaciones más típicas son: identidad, translación, escalado, rotación y proyección.

La transformación más sencilla que hay es la de identidad. Dado un vector $\vec{p}$ en el espacio, el resultado de aplicar la identidad es el mismo vector $\vec{p}$. La Ecuación 2.5 muestra esta operación. Si nos fijamos en esta operación, vemos que realmente la fila de la matriz nos indica qué componente del vector queremos manipular (si x, y, z o w), mientras que la columna nos indica el peso de cada componente. Por ejemplo, en la matriz identidad, la primera fila sólo tiene el primer elemento a 1. Al ser la primera fila, estamos estableciendo el valor de la componente x, que es la primera del vector. Esta fila nos indica qué cantidad de cada componente queremos establecerle a la componente x. El primer elemento (columna) nos indica que queremos 1 de la componente x, el segundo indica 0 de la componente y, el tercero 0 de la componente z y el cuarto 0 de la componente w. Por lo tanto, la matriz identidad, por cada componente selecciona únicamente el componente que le corresponde al vector original, por lo que no lo altera.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix} \tag{2.5}$$

Respecto a la componente w, cabe destacar la importancia que tiene a la hora de diferenciar si el vector es de posición (punto) o de dirección. En este aspecto, si la componente w es 1, entonces el vector se define como punto, mientras que si vale 0, entonces se trata de un vector de dirección.

Para definir la matriz de translación, necesitamos una operación que sume a cada componente del punto original la cantidad deseada de un vector $\vec{p}$. Dado

[5] En este Capítulo sólo veremos las transformaciones más comunes, pero existen muchas más.

que los puntos siempre tienen su componente w a 1, usaremos ésta para determinar la cantidad de movimiento en cada componente. La Ecuación 2.6 muestra esta transformación. Como se puede observar, dado que la componente w es siempre 1 podemos establecer en cada fila la cantidad de movimiento que queremos en cada componente. La última columna nos está indicando qué cantidad de w queremos en cada componente. En la primera fila, que es la que define la componente x del punto $\vec{p}$, estamos indicando que queremos 1 de la propia componente x del punto $\vec{p}$ más 1 de la componente x del vector $\vec{v}$. Y así con el resto de componentes. Es importante destacar que si aplicamos una transformación de translación a un vector, obtenemos el mismo vector, independientemente de la cantidad de movimiento establecida. Esto se debe a que la componente w en los vectores es 0 y, por lo tanto, el vector de movimiento no se añade. Esto es útil cuando estamos transformando modelos que tienen normales. Las normales son vectores, y no deben alterarse con movimientos de translación.

$$\begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{pmatrix} \tag{2.6}$$

La transformación de escalado consiste en alterar el tamaño del espacio vectorial teniendo como referencia el origen de coordenadas. Así pues, esta transformación consiste, en esencia, en multiplicar cada una de las componentes del vector por un escalar. Cada componente se puede multiplicar por un valor distinto, dando lugar a efectos de estiramiento o contracción. La Ecuación 2.7 muestra esta operación.

$$\begin{pmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x * v_x \\ p_y * v_y \\ p_z * v_z \\ 1 \end{pmatrix} \tag{2.7}$$

La rotación es una transformación un poco más compleja en tanto que consis-

te en la agregación de varias transformaciones. Aunque estas transformaciones son más eficientes, precisas y simples con el uso de *quaterions* [41, 72], el uso de matrices en este ámbito es necesario dado que al final se necesita pasar una matriz al *vertex shader*. Para solventar este problema, generalmente se hacen conversiones de *quaternion* a matriz. La transformación de rotación consiste en tres sub-transformaciones de rotación, una por cada eje del espacio ortogonal[6]. Cada eje define su propio ángulo, y al conjunto de todos estos se le denomina ángulos de Euler. De este modo, tendremos una transformación de rotación en el eje x, otra en el eje y y otra en el eje z. Para juntar estas tres transformaciones, basta con computar la multiplicación de estas tres matrices de transformación. Las Ecuaciones 2.8, 2.9 y 2.10 corresponden a las matrices de transformación de los ejes x, y y z, definidas como R_x, R_y y R_z respectivamente.

$$R_x(\theta_x) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta_x) & -sen(\theta_x) & 0 \\ 0 & sen(\theta_x) & cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.8}$$

$$R_y(\theta_y) = \begin{pmatrix} cos(\theta_y) & 0 & sen(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -sen(\theta_y) & 0 & cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.9}$$

$$R_z(\theta_z) = \begin{pmatrix} cos(\theta_z) & -sen(\theta_z) & 0 & 0 \\ sen(\theta_z) & cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.10}$$

Así pues, la matriz de rotación final que junta los ángulos θ_x, θ_y y θ_z es el producto de estas tres matrices, tal y como muestra la Ecuación 2.11.

[6]Ésta es una forma de representarla, que es mediante las transformadas esenciales de los ángulos de Euler, pero en realidad existen muchas más.

$$R(\theta_x, \theta_y, \theta_z) = R_x(\theta_x) \cdot R_y(\theta_y) \cdot R_z(\theta_z) \tag{2.11}$$

Hay que tener en cuenta que el orden de las transformaciones importa, tal y como se ha mencionado antes. Esta combinación de matrices de cada ángulo de Euler para definir las rotaciones tiene un problema. Consideremos los planos definidos por cada eje. Si, por ejemplo, el plano definido por el eje x se rota de tal manera que se vuelve paralelo al del eje y, es decir, los ejes se alinean, entonces rotar sobre el eje x es equivalente a rotar sobre el eje y, por lo que se pierde la libertad de rotar sobre el eje x de este sistema. A este problema se le conoce como *gimbal lock* (también conocido como bloqueo de cardán), y se soluciona mediante el uso de *quaternions* en lugar de matrices [59, 4, 68].

Finalmente, en lo que a transformaciones se refiere, tenemos la matriz de proyección, cuyo objetivo es transformar los vértices de un espacio tridimensional a un espacio bidimensional. Aquí existen dos opciones de transformación: proyección ortogonal y proyección de perspectiva. La primera consiste simplemente en proyectar los puntos del espacio en un plano usando líneas paralelas a éste. Dado que este espacio ya es así en el sistema de coordenadas original, esta matriz se puede definir simplemente como una transformación de escalado (Ecuación 2.7) que se ajuste a la pantalla del dispositivo sobre la que se va a renderizar la escena. La segunda consiste en proyectar los puntos de la escena en un plano perpendicular a la línea de visión de la cámara, usando para ello líneas que van desde el centro de la cámara (el punto de fuga) hasta el punto que se desea proyectar. Para definir esto con una matriz, es necesario acotar el volumen de visión de la cámara, es decir, todo aquello que se encuentra dentro del campo de visión de ésta. A este volumen se le conoce como *frustum volume*, y se puede definir mediante dos planos y un rectángulo de visión. Los planos se denominan *near* (el más cercano a la cámara) y *far* (el más lejano). En el plano *near* es donde se proyectan los puntos. El rectángulo se define dentro del plano *near* estableciendo los límites superior, inferior, derecho e izquierdo, que se denominan en la literatura como *top*, *bottom*, *right* y *left* respectivamente. Desde el punto de fuga hasta cada una de las esquinas de este rectángulo se trazan rectas que definen finalmente el *frustum volume*. La Figura 2.6 muestra

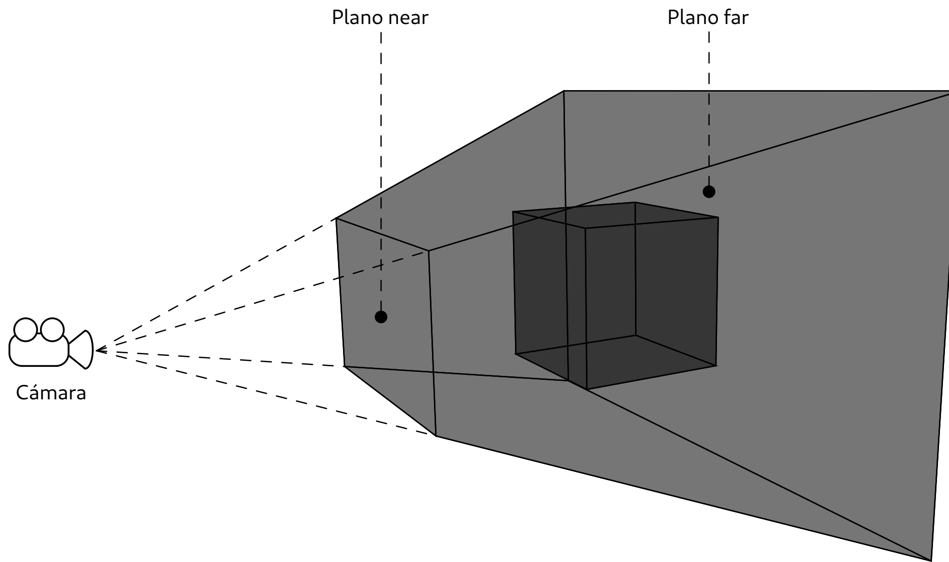


Figura 2.6: Ejemplo de *frustum volume* en gris claro. En gris más oscuro, un objeto dentro de él que sería visible.

una representación del *frustum volume* en el espacio, donde se pueden apreciar los planos *near* y *far*, y que las aristas laterales de dicha forma geométrica convergen en la cámara, que sería el punto de fuga de la escena.

Teniendo en cuenta estos parámetros, la matriz de proyección de perspectiva se define como se ve en la Ecuación 2.12 [49], siendo n y f las distancias del punto de fuga a los planos *near* y *far* respectivamente, y t, b, r y l los límites del rectángulo sobre el que se quiere proyectar, a saber, *top*, *bottom*, *right* y *left* respectivamente.

$$P_{persp.}(n,f,t,b,r,l) = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{2.12}$$

Sistemas de coordenadas

El objetivo final del Geometry Pipeline es ubicar los objetos visibles dentro de lo que se conoce como cubo unitario. El cubo unitario es un sistema de coordenadas

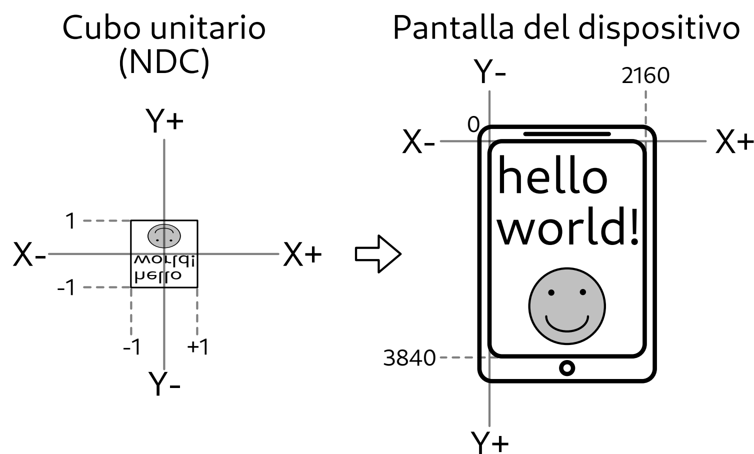


Figura 2.7: Mapeo del cubo unitario al dispositivo de visualización.

que se mapea directamente al dispositivo donde se va a visualizar la escena final. Dependiendo de la API gráfica, éste tiene unos orígenes y límites u otros. Por ejemplo, en OpenGL se considera que este cubo tiene como origen de coordenadas el punto $(0, 0, 0)$ (centro del dispositivo), y el rango del mismo va desde $(-1, -1, -1)$ hasta $(1, 1, 1)$. A este sistema de coordenadas también se le conoce como sistema NDC (de sus siglas *Normalized Device Coordinates*). La Figura 2.7 representa este mapeo de cubo unitario a dispositivo. En este ejemplo podemos ver el cubo unitario en coordenadas NDC que se mapea a una pantalla UltraHD (2160x3840 píxeles) de un dispositivo móvil. Aquí es importante notar que el ratio de aspecto en coordenadas NDC puede ser diferente al de la pantalla del dispositivo, de ahí que en la figura se vea un poco aplastada la imagen (aparte de invertida). También es importante destacar que los orígenes de coordenadas también difieren, pues mientras que en NDC el origen es el centro del espacio a visualizar, en las pantallas generalmente es la esquina superior izquierda. Respecto a la orientación del eje que define la profundidad (eje z), también depende de la API gráfica. En el caso de OpenGL, se define como el eje z positivo, de modo que la profundidad 0 significa lo más cercano a la cámara (situado en el plano *near*), mientras que profundidad 1 significa el punto visible más lejano (contenido en el plano *far*).

Con todas las transformaciones que hemos visto en esta sección se puede convertir de un sistema de coordenadas a otro. Así pues, los modelos de la escena pa-

san por varios sistemas hasta ser proyectados en la pantalla. Los vértices de un modelo, inicialmente, se encuentran en lo que se denominan coordenadas de modelo. En este sistema el origen es el centro del modelo y se definen las transformaciones que se van a efectuar sobre éste. Por ejemplo, si queremos que el modelo tenga un tamaño más grande, le aplicamos una matriz de escalado. Con una transformación de translación podemos mover el centro del objeto, de modo que cuando apliquemos una rotación se tome el nuevo centro como pivote. Una vez transformado el objeto, hay que colocarlo dentro de la escena. A este nuevo sistema de coordenadas se le denomina coordenadas del mundo, en el que el centro es la escena en sí misma. Por ejemplo, si estamos renderizando un campo de fútbol, el centro de este sistema sería el centro del campo. Para pasar un objeto a coordenadas del mundo tenemos que decidir en qué localización de este sistema queremos situar el modelo, y entonces, aplicar una transformación de translación (Ecuación 2.6) en función de la posición. Una vez se han situado todos los modelos en la escena, hay que colocar la cámara. Para ello, dado que no existe el concepto de cámara como tal, sino que todo lo que se dibuja finalmente en la pantalla es lo que aparece en el cubo unitario, lo que se hace es mover toda la escena hasta que la parte en la que queramos que esté la cámara caiga dentro del cubo unitario. Dicho de otro modo, para colocar la cámara hay que aplicar una transformación de translación opuesta a la verdadera localización de la misma en la escena. Respecto a la rotación, también es necesario que se rote toda la escena pertinentemente y, nuevamente, en sentido opuesto por el mismo motivo. Una vez colocada la cámara se pasa a coordenadas NDC. Aquí hay dos opciones, a saber: realizar una transformación de proyección o una transformación ortogonal, tal y como se ha explicado en la Sección 2.1.1.

Como se puede observar, el sistema de coordenadas NDC tiene sólo tres dimensiones, en vez de 4. Esto significa que tenemos que convertir de un espacio vectorial a otro. Esta transformación es muy sencilla, pues simplemente consisten en dividir cada componente del vector entre la componente w, algo que también se le conoce como *perspective divide*. Es importante no hacer esta división de manera explícita en el *vertex shader*, pues de hacerlo, resultaría en inconsistencias en la imagen final. Existe un espacio intermedio antes de llegar al NDC, que se denomina *Clipping Coordinates* (o CC) (véase Sección 2.1.5), que está definido en las

cuatro dimensiones y sirve para hacer de manera correcta el proceso de *clipping* de primitivas.

2.1.2. Teselaciones

Con frecuencia es necesario subdividir superficies de objetos para conseguir efectos más realistas y/o modelos con más detalle. Aquí es donde entran en juego las teselaciones (más ampliamente conocidas como *tessellations*), cuyo objetivo es disminuir las necesidades de almacenamiento de un objeto que necesita ser subdividido.

Para entender este concepto, vamos a poner un ejemplo sencillo. Imaginemos que queremos renderizar el mar con una superficie que simula tener olas. Para lograrlo, empezaríamos creando una superficie plana, y la subdividiríamos en cuadrados más pequeños hasta conseguir un nivel de detalle considerable. Si queremos reproducir el movimiento de las olas necesitaremos hacer muchas subdivisiones que nos permitan deformar con flexibilidad la superficie de la malla creada. Como se ha visto anteriormente, el modelo generalmente se almacena en la memoria de la GPU, para que ésta pueda acceder a él y renderizarlo. A medida que vamos subdividiendo la superficie, el número de vértices va aumentando y, por lo tanto, las necesidades de almacenamiento. Esto puede provocar mucho tráfico a memoria para leer el modelo. Este tráfico es inútil dado que las subdivisiones de las superficies del modelo se pueden obtener mediante una sencilla función lineal, en vez de ser almacenadas en memoria. Para resolver este problema, se añade esta etapa al *pipeline* gráfico, de modo que las subdivisiones se crean sobre la marcha sin necesidad de almacenar los vértices en memoria.

Así pues, esta etapa recibe primitivas[7] como entrada, y obtiene un número igual o superior de primitivas de salida. Al número de subdivisiones de una superficie o primitiva se le denomina grado de teselación. El proceso de teselación se lleva a cabo mediante tres subprocedimientos bien definidos, a saber: control de la teselación, generación de primitivas de salida y evaluación de la teselación.

[7] En esta etapa, OpenGL denomina a las primitivas como parches (o *patches*).

El control de teselación se lleva a cabo mediante un programa definido por el usuario denominado *Tessellation Control Shader* (o TCS), cuyo objetivo es determinar el grado de teselación de las primitivas de entrada y modificarlas si es necesario (transformar, añadir o quitar vértices). Además, se pueden definir parámetros que definen la subdivisión de la superficie. Esta etapa es opcional, y si no se define ningún TCS, los parches (primitivas en esencia) se pasan directamente a la etapa de generación de primitivas.

La generación de primitivas de salida es una etapa fija que se encarga de generar una serie de primitivas dada la primitiva de entrada. Para ello, se hace uso de una función de subdivisión que utiliza los parámetros definidos en la etapa TCS.

Finalmente, la evaluación de la teselación se lleva a cabo mediante la ejecución de otro programa definido por el usuario denominado *Tessellation Evaluation Shader* (o TES). Esta etapa se encarga de procesar los atributos resultantes de los vértices generados durante la etapa anterior y los parámetros del TCS y computar los valores finales de los vértices. Aquí, por ejemplo, se recompuntan las normales para los efectos de sombreado, las coordenadas de textura o los colores de los vértices. Esta etapa se puede observar como un *vertex shader* que se ejecuta varias veces por vértice (dependiendo del grado de teselación). Para que la teselación tenga lugar, esta etapa es obligatoria, pues sin ésta no tiene sentido la etapa de generación de vértices.

2.1.3. Procesamiento de la Geometría

Una vez ya están todos los vértices procesados y en coordenadas NDC, éstos se ensamblan en forma de primitiva. Como ya se ha visto, una primitiva está compuesta por varios índices a vértices y un tipo de primitiva. Esta etapa (*Geometry Processing*) se encarga de agrupar estos vértices y manipular la geometría de la primitiva de entrada dada. De este modo, se pueden añadir vértices nuevos y emitir primitivas que en la geometría original de entrada no estaban. Para ello, se ejecuta otro programa denominado *geometry shader*, cuyo objetivo es procesar la primitiva y modificarla acorde a las necesidades de la escena determinada por el diseñador. Dado que este tipo de efectos no es muy común en escenas, esta etapa es opcional.

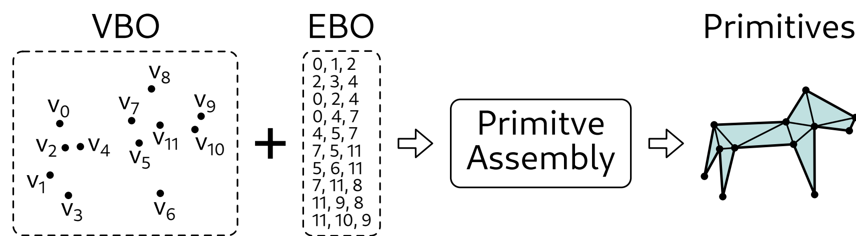


Figura 2.8: Ejemplo ilustrativo del flujo de ejecución de la etapa *Primitive Assembly*.

Generalmente, en esta etapa se puede añadir un nuevo vértice con una llamada a `EmitVertex` y, una vez formada la primitiva deseada, llamar a `EndPrimitive`. Nótese que el tipo de primitiva de entrada no tiene por qué ser el mismo que el de salida. Este aspecto se especifica también en el *geometry shader*.

2.1.4. Ensamblado de Primitivas

Para renderizar una escena, primero es necesario convertir la geometría en primitivas básicas, es decir, en puntos, líneas y triángulos. Esto se debe a que estas primitivas base son más fáciles de dibujar que las primitivas compuestas. Así pues, esta etapa se encarga de realizar esta conversión. En el caso de los puntos (`GL_POINTS`), esta etapa simplemente los pasa sin procesar a la siguiente etapa. En el caso de las líneas (`GL_LINE_STRIP` y `GL_LINE_LOOP`), cada pareja posible de vértices adyacentes genera una línea. El resto de primitivas se descomponen en triángulos, proceso también conocido como triangulación de la geometría.

La Figura 2.8 muestra el funcionamiento de esta unidad funcional. En este ejemplo se puede ver cómo a esta etapa llegan tanto el VBO (la información de los vértices) como el EBO (la información de los índices a esos vértices). Con esta información y el tipo de primitiva se construyen las primitivas de salida. En este caso, el tipo de primitiva es GL_TRIANGLES, porque por cada tres elementos se emite un triángulo independiente.

2.1.5. *Clipping & Culling*

Una vez triangulada toda la geometría, se recorta para ajustarla al cubo unitario, que es todo lo que será finalmente visible en la pantalla. Aquí, los triángulos que están completamente fuera del cubo unitario se descartan (*culling*), de modo que no pasan a la siguiente etapa. De igual manera, aquellos triángulos que muestran su cara trasera a la línea de visión también se descartan. A este proceso se le denomina *back-face culling*. Qué es una cara delantera o trasera dependerá de la configuración inicial del contexto de OpenGL. Esto se determina mediante el orden en el que están definidos los vértices del triángulo. En este aspecto tenemos dos opciones: orden *horario* u orden *antihorario*. Por ejemplo, se puede considerar que si los vértices de un triángulo están colocados en orden horario, ésta corresponde con la cara visible. Por otro lado, los triángulos que están contenidos completamente dentro del cubo unitario pasan por esta etapa sin modificaciones. En el caso de triángulos que están parcialmente dentro del cubo unitario hay que recortarlos (*clipping*) de modo que se genere un polígono que no tenga partes fuera de dicho cubo. Existen gran cantidad de algoritmos para realizar estos recortes [79, 38, 82, 85].

Como hemos mencionado antes, antes del espacio NDC está el espacio CC, cuyas coordenadas están definidas en las cuatro dimensiones de un punto. Esto tiene su explicación, y es que si usáramos las coordenadas en NDC, los puntos que se encuentran detrás de la cámara se encontrarían dentro del cubo unitario, pues al tener éstos las componentes z y w negativas, la división daría un número positivo. Es por esto que se usa el sistema de coordenadas CC, en el que se determina si un punto en las cuatro dimensiones está dentro si se cumple la condición definida por la Ecuación 2.13.

$$-w \leq (x, y, z) \leq w \tag{2.13}$$

2.1.6. *Rasterizer*

El *Rasterizer* es la primera etapa del *Raster Pipeline*, y se encarga de discretizar los triángulos de la escena en elementos del tamaño de un píxel denominados frag-

mentos. Para que esto tenga lugar, hace falta hacer un par de conversiones implícitas en el *pipeline* gráfico. Si recordamos lo que pasaba en la etapa *Vertex Processing*, nos damos cuenta de que a la salida de ésta las coordenadas de todos los vértices están en coordenadas NDC, cuyos valores oscilan en el rango $[0, 1]$. Dado que la pantalla se encuentra en otros rangos[8], es necesario una conversión de espacios. A este nuevo sistema de coordenadas se le denomina coordenadas de la pantalla (o *screen coordinates*) Esta conversión se hace de manera implícita[9] multiplicando las coordenadas de los vértices por un factor. La segunda conversión es opcional, pero por motivos de rendimiento (y sobre todo en *hardware*) es conveniente realizarla. Se trata de cambiar el formato de las coordenadas, pasando de punto flotante de simple precisión (32 bits) a un formato de punto fijo (o *fixed point*) de la misma precisión. Esto se debe a que las operaciones con estos números son computacionalmente más simples y, por lo tanto, menos costosas de implementar en *hardware*. Además, la velocidad a la que se realizan estas operaciones con números en este formato es muy superior.

Una vez convertidos los vértices a coordenadas de la pantalla se procede a rasterizar los triángulos. Este proceso consiste en generar una lista de fragmentos (equivalentes a píxeles) para cada triángulo. Existe una gran cantidad de esquemas para generar estas listas de fragmentos. Probablemente, la más sencilla sea la del *test del punto dentro del triángulo*, que dado un punto concreto en la pantalla y un triángulo definido por tres vértices determina si el punto está o no dentro del triángulo. El algoritmo es muy sencillo, y se describe en el pseudo-código 2.1. Como puede observarse, este algoritmo es muy ineficiente, pues por cada triángulo hay que recorrer todos los píxeles de la pantalla, incluso si el triángulo a rasterizar ocupa unos pocos píxeles. Obviamente, existen otros esquemas mucho más eficientes como, por ejemplo, computar previamente una *bounding box* del triángulo y sólo recorrer ese trozo de pantalla o, generar únicamente los fragmentos que pertenecen a ese triángulo usando una variante del algoritmo de Bresenham [14, 13, 15].

```
1 for (const triangle of Scene)
2 {
```

[8]Por ejemplo, en una pantalla con resolución FullHD, el rango de dibujado es de 1920x1080 píxeles.

[9]En OpenGL se usan los parámetros definidos por el usuario durante la llamada a `glViewport`.

```
3     for (let y = 0; y < Screen.height; y++)
4         for (let x = 0; x < Screen.width; x++)
5         {
6             if (triangle.has(x, y))
7                 return triangle.interpolate(x, y);
8         }
9 }
```

Código 2.1: Algoritmo de rasterización más básico.

Para determinar si un punto se encuentra o no dentro de un triángulo se utilizan lo que se conoce como *edge functions*, que aprovechan las propiedades del producto vectorial y que fueron definidas por primera vez en [69]. Una *edge function* es una función que dada una recta y un punto, determina en qué lado de la recta está el punto: si a la derecha o a la izquierda. Para ello se computa el producto vectorial entre el vector director de la recta y el vector definido por un punto de la recta y el propio punto a evaluar. El signo de este producto vectorial indica el lado respecto al segmento del triángulo en el que se encuentra el punto.

La Figura 2.9 muestra el resultado del producto vectorial entre dos vectores, a y b. Se puede observar que el resultado es otro vector perpendicular al plano que forman los dos vectores originales, y cuya magnitud es igual al área del paralelepípedo formado por estos dos vectores. Como puede observarse, dependiendo del orden en el que se aplica esta operación, el vector tiene una dirección u otra, y ahí es donde está el potencial de esta operación, pues esta nos indica si un punto está a la derecha o a la izquierda de una recta.

Es importante notar que la dirección de los vectores determina el signo, por lo que hay que establecer una convención a la hora de computar las *edge functions*. Para determinar si un punto se encuentra dentro de un triángulo o no, lo único que hay que hacer es computar las *edge functions* de las tres rectas que definen los lados del triángulo con el punto deseado y observar si todos los resultados tienen el mismo signo. Si es así, entonces el punto está dentro del triángulo. Para definir las *edge functions* de los lados del triángulo, se escoge un orden arbitrario de los vértices: horario o antihorario. Por ejemplo, supongamos un triángulo formado por los vértices A, B y C, tal y como se muestra en la Figura 2.10. Vamos a escoger un orden

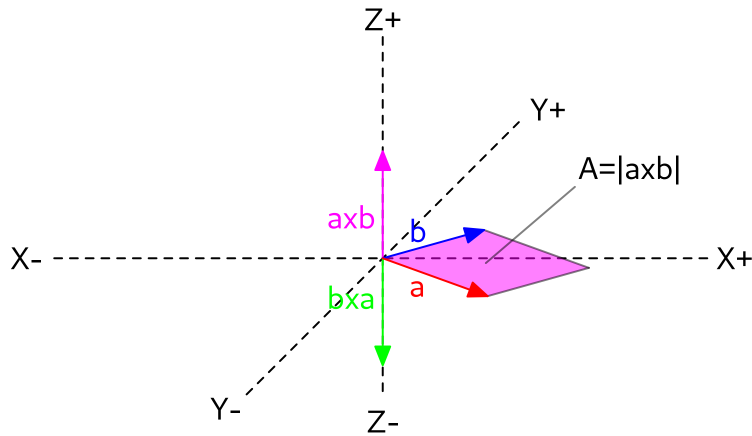


Figura 2.9: Representación gráfica del producto vectorial de dos vectores `a` y `b`, y la importancia que tiene el orden en el que se aplica sobre la dirección del vector resultante.

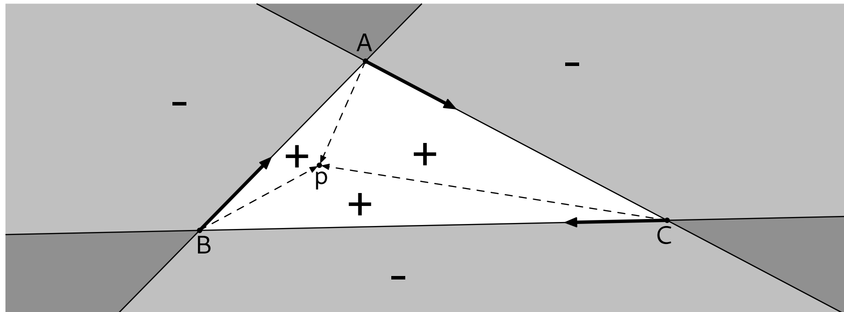


Figura 2.10: Ejemplo de *edge functions* definidas por los lados de un triángulo.

horario para los vértices, de modo que generamos las *edge equations* definidas por las rectas definidas por los vectores AC, CB y BA. De este modo, si multiplicamos el punto a evaluar por cada una de las tres *edge functions*, tenemos que si el resultado de las tres operaciones es positivo, entonces el punto se encuentra dentro del triángulo. En caso contrario, el punto está fuera del triángulo. Es por esto que es importante el orden en el que se consideran los vértices y, también, el orden en el que se multiplican los vectores.

Una vez definida esta función que determina si un punto está o no dentro de un triángulo, falta definir los puntos a evaluar. Dado que la pantalla sigue definida en números reales, tenemos que determinar la cantidad de puntos a evaluar dentro de la misma. En este aspecto también existen muchas formas de hacerlo, siendo

la más sencilla la de evaluar un único punto por cada píxel de la pantalla[10]. Cada uno de estos puntos se sitúa en el centro del píxel. De este modo, si tenemos una pantalla de 1920x1080 píxeles tenemos 2,073,600 puntos a evaluar. Por cada uno de estos puntos se evalúa la función definida anteriormente, y si la función devuelve verdadero, entonces se genera un fragmento. El resultado de este recorrido es una lista de píxeles que se sitúan dentro del triángulo dado. Obviamente, éste no es el esquema más eficiente, ya que a simple vista se puede optimizar definiendo una caja que englobe el triángulo (lo que se denomina *bounding box*), en vez de la pantalla entera, ahorrando así muchos tests.

Obviamente, con esto no es suficiente, pues los triángulos tienen muchos más atributos definidos más allá de un valor booleano. Al igual que los vértices, los fragmentos llevan asociados consigo una lista de atributos derivada de los vértices del triángulo al que pertenecen. Para obtener estos atributos se ejecuta un procedimiento denominado interpolación de atributos, en el que los atributos de cada vértice son interpolados en función de la distancia del fragmento generado a cada vértice del triángulo. Para obtener estos parámetros de interpolación se computan lo que se conoce como coordenadas baricéntricas. Dicho sistema consta de tres dimensiones que determinan qué fracción de los atributos de cada vértice debe tener el fragmento generado. Para calcular estas coordenadas es necesario primero dividir el triángulo en tres subtriángulos tal y como muestra la Figura 2.11. El área de cada uno de estos subtriángulos supone una fracción del total, que se corresponde con la coordenada baricéntrica del vértice opuesto a dicha área. En la Figura 2.11 se pueden observar tres áreas, a saber: A_1, A_2 y A_3, que corresponden a los vértices V_1, V_2 y V_3 respectivamente. Estas áreas están normalizadas a la del triángulo inicial, lo que significa que la suma de las tres da la unidad. Así pues, cada fracción corresponde a una coordenada baricéntrica. Si observamos con detenimiento el resultado de evaluar un punto con una *edge funciton*, enseguida nos damos cuenta de que coincide con el área definida por el paralelepípedo formado por el vector director de la recta y el vector definido por el vértice del triángulo y el punto a

[10]Existen otros esquemas que evalúan varios puntos dentro de un píxel para evitar lo que se conoce como *aliasing*, un problema que se caracteriza por generar dientes de sierra en las aristas de los triángulos pequeños. Dichos esquemas se conocen como *Multi Sampling Anti aliasing* (o MSAA) debido a que evalúan múltiples muestras (*samples*) por píxel, dando lugar a lo que se conoce como *antialiasing*.

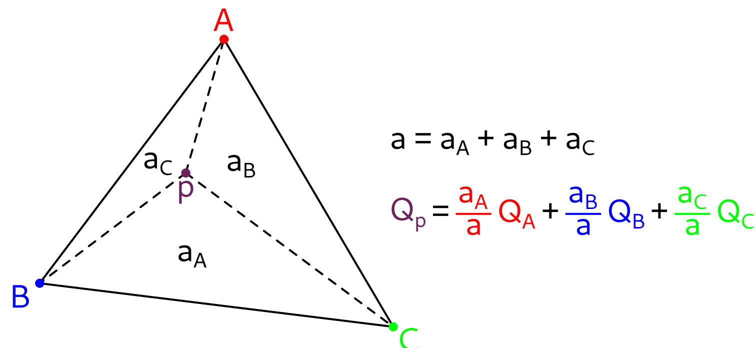


Figura 2.11: Ejemplo de obtención de las coordenadas baricéntricas de un fragmento que pertenece a un triángulo.

evaluar, simplemente por la propiedad del producto vectorial. Si este resultado lo dividimos entre dos obtenemos el área del triángulo.

A pesar de que con este esquema se pueden obtener las coordenadas baricéntricas de un punto determinado dentro de un triángulo, éste sufre de un problema derivado de la perspectiva. La Figura 2.12 representa un claro ejemplo en el que una operación de interpolación de este tipo devolvería un resultado indeseado. Aquí se puede observar un segmento muy oblicuo a la cámara, el cual es proyectado en el plano de proyección. Este segmento consta de dos vértices, cada uno con un atributo de valor conocido, a saber, a_1 y a_2, respectivamente. Los atributos de los puntos proyectados tienen el mismo valor que sin proyectar. Como la interpolación se lleva a cabo en los vértices proyectados y no en los originales, lo lógico sería interpolar linealmente. Para que se vea más claro, aquí hemos cogido como ejemplo la mitad del segmento proyectado. Si interpolamos linealmente, obtendremos el 50 % de peso para cada atributo, obteniendo así el atributo nuevo, a_e. Sin embargo, si nos fijamos en el punto que se proyecta en el segmento original, no coincide con el 50 % de dicho segmento, por lo que los pesos reales del atributo a_i no coinciden, dando así un resultado erróneo. Este problema se agrava aún más cuanto más oblicuo es el segmento (o superficie) respecto a la cámara.

Para solucionar el problema, se aplica un factor de corrección de perspectiva (también conocido como *perspective correction*), de modo que se tiene en cuenta la profundidad de los vértices a interpolar para dar lugar a una imagen más realista.

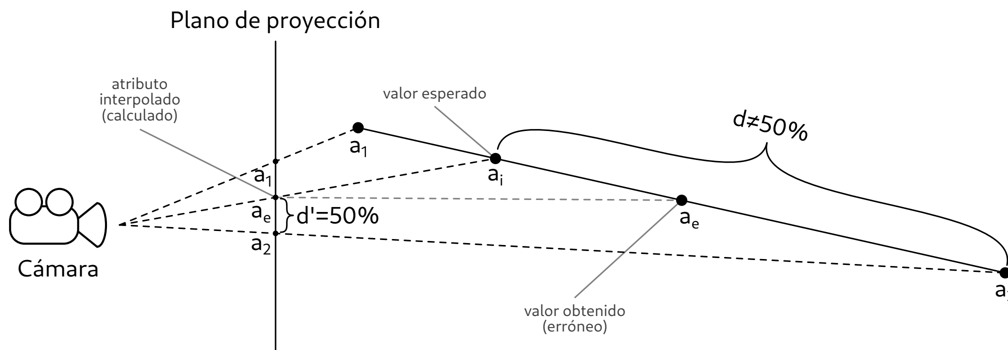


Figura 2.12: Representación del problema de la perspectiva.

Tal y como se ha dicho en la Sección 2.1.1, los vértices cuentan con un canal extra (denominado w), para hacer el *clipping* y la corrección de la perspectiva. Esta componente nos indica el grado de proyección de un vértice respecto a la cámara, de modo que al hacer el cómputo de las coordenadas baricéntricas, ésta se usa como componente de peso de cada vértice.

Volviendo a los métodos de rasterización, existe un esquema mucho más eficiente para generar fragmentos y sus correspondientes coordenadas baricéntricas denominado el método *scanline* [2]. Este método consiste en recorrer el triángulo moviendo una línea horizontal desde el vértice superior del triángulo hacia el inferior, determinando en todo momento los incrementos y límites izquierdo y derecho entre los que se generan los fragmentos. Para determinar estos incrementos y límites se usa el *algoritmo de Bresenham* [13], para lo cual es necesario, nuevamente, ordenar los vértices del triángulo, pero esta vez de arriba hacia abajo. Para determinar las coordenadas baricéntricas basta con computarlas una vez, y computar los incrementos de éstas en cada movimiento en el eje X e Y, lo que se denomina incrementos. De este modo, cada vez que se mueva la *scanline* o el eje X se usan estos incrementos en vez de recomputar las coordenadas baricéntricas.

2.1.7. *Early Z-Test*

Hasta esta etapa del *pipeline* gráfico se ha determinado si un punto pertenece a un triángulo o no, y sus coordenadas baricéntricas respecto a éste para obtener sus atributos, pero no se ha determinado si es visible o no en la pantalla por haber sido

ocluido por otras primitivas. De esto se encarga la etapa *Early Z-Test*, cuyo único objetivo es resolver el conocido problema de la visibilidad.

En una escena podemos encontrarnos una gran cantidad de primitivas, pero sólo unas cuantas son finalmente visibles dentro de la pantalla del dispositivo, es decir, el usuario sólo percibe unas pocas. Esto se debe a que hay primitivas que solapan con otras, dando lugar a primitivas ocluidas. Determinar la visibilidad de una manera computacionalmente eficiente es un tema que ha dado mucho de qué hablar en la literatura, pues han sido muchas las propuestas al respecto. El método más comúnmente aplicado en GPUs es el que se conoce como Z-Buffer. El Z-Buffer es una estructura de datos de las mismas dimensiones que la pantalla del dispositivo, pero en vez de almacenar colores, por cada posición almacena la información de profundidad del fragmento correspondiente más cercano a la cámara[11]. Dado que la posición de un vértice es un atributo más, se interpola durante la rasterización (etapa anterior) y llega a esta etapa. Además, dado que las posiciones de estos vértices están en coordenadas NDC, sus valores oscilan también en el rango $[0, 1]$, significando 0 el *near plane* del *frustum view* y 1 el *far plane*. En esta etapa se compara la profundidad del fragmento entrante con la profundidad que le corresponde en el Z-Buffer. A esta comparación se le denomina Z-Test, y es una función que el programador puede seleccionar. Dicho de otro modo, el Z-Test consiste en aplicar a cada fragmento que llega una función que recibe como entrada la profundidad del fragmento entrante y la que ya hay almacenada en el Z-Buffer para esa posición de pantalla, y como resultado da un valor booleano indicando si el fragmento es visible o no, es decir, si pasa a la siguiente etapa del *pipeline*. Si el fragmento supera el Z-Test, entonces se actualiza la profundidad correspondiente del Z-Buffer con la del fragmento, y éste es pasado a la siguiente etapa. En caso contrario, es descartado en tanto que ha quedado ocluido por una primitiva previa y no es visible en la imagen final. Es importante notar que esta es la implementación mínima necesaria para obtener una imagen correcta. Existen otros esquemas mucho más complejos para determinar la visibilidad de una escena de una forma más eficiente, y que son objeto de estudio de esta Tesis.

[11]En realidad, esto depende de la función que se establezca como Z-Test, pero la más común es escoger el fragmento más cercano a la cámara (función *less than* $<$).

El término *early* de esta etapa se debe a que permite descartar de una manera más temprana fragmentos, evitando así la etapa que más coste tiene en energía el *pipeline* gráfico, que es la de *Fragment Processing* (véase la Sección 2.1.8).

Otra cosa a tener en cuenta son las propiedades del comando (objeto de la escena) emitido respecto a esta etapa. En este aspecto encontramos dos parámetros booleanos, a saber: *Z-Test enabled* y *Z-Write enabled*. El primero de éstos nos indica si el fragmento entrante tiene que ser evaluado en esta etapa. Si está desactivado, entonces el fragmento pasa directamente a la etapa *Fragment Processing*, sin comprobar su profundidad. Si está activado, entonces pasa por esta etapa, y además se evalúa el segundo parámetro (*Z-Write enabled*). Este parámetro nos indica si debemos actualizar el Z-Buffer tras un test exitoso, es decir, que el fragmento pase el Z-Test. Esta *flag* generalmente está desactivada para las primitivas transparentes puesto que no es deseable que una primitiva de este tipo actualice la profundidad[12].

2.1.8. Procesado y sombreado de fragmentos

Ésta es, con vasta diferencia, la etapa más compleja, costosa computacionalmente e importante del *pipeline* gráfico y, probablemente, de la que menos documentación se puede encontrar al respecto sobre su implementación en *hardware*. De ahí el recelo de los principales fabricantes de GPUs a mostrar sus detalles de implementación *hardware*.

A esta etapa (denominada *Fragment Processing*) llegan los fragmentos que han superado el test de visibilidad de la etapa anterior (*Early Z-Test*), y se procesan por medio de la ejecución de un programa definido por el usuario denominado *fragment shader*. Este programa recibe como entrada un fragmento en forma de atributos y obtiene como salida uno o varios colores, que generalmente están definidos en el espacio RGBA. A la ejecución de este programa también se le conoce como proceso de sombreado o *shading*. Durante este proceso, los atributos del fragmento

[12]En este sentido es muy importante el orden en el que se renderizan las primitivas, y esto depende del programador. Generalmente, los motores gráficos dibujan primero las primitivas opacas y luego las transparentes, porque así se realiza la mezcla correctamente. Sin embargo, también hay que tener en cuenta el orden en el que las propias primitivas transparentes se dibujan. Lo óptimo sería ordenar las primitivas de detrás hacia delante, para dar lugar a una mezcla realista.

entrante se manipulan para obtener finalmente un color o una lista de colores. El programador puede definir un número de *buffers* sobre los que dibujar, de ahí que la salida del *fragment shader* pueda contener más de un color, dependiendo de los *buffers* de destino. En OpenGL, por defecto, si no especifica ninguno, se dibuja en el `FRONT_BUFFER`.

Generalmente, en esta etapa se encuentran varios procesadores de fragmentos, también denominados *Fragment Processors*. Esto se debe a la alta complejidad de las operaciones realizadas y al impacto que esto tiene en su productividad en el *pipeline* gráfico, pues es necesario que esta etapa no sea un cuello de botella. Esto quiere decir que mientras que un *Fragment Processor* es capaz de procesar x fragmentos por unidad de tiempo, la etapa *Early Z-Test*, por ejemplo, es capaz de procesar $10x$, por lo que si sólo ponemos un *Fragment Processor* en esta etapa supondrá un cuello de botella importante. Dado que la productividad de un *Fragment Processor* depende de la complejidad del *fragment shader* a ejecutar, es difícil determinar un número exacto de *Fragment Processors* en esta etapa. Por lo que el diseñador de la GPU debe determinar una cantidad en función de varios *benchmarks* de casos de uso común. Como cualquier sistema multiprocesador, la etapa de *Fragment Shading* de una GPU cuenta con un planificador de fragmentos que determina qué *Fragment Processor* va a procesar el fragmento de entrada. Dependiendo de la información usada para determinar el *Fragment Processor* de destino, los planificadores se clasifican en estáticos o dinámicos. Un planificador estático sólo tiene en cuenta información intrínseca del fragmento entrante, es decir, no tiene en cuenta variables externas como el número de fragmentos que ha procesado cada *Fragment Processor*, la ocupación actual de los mismos o cualquier otro tipo de contadores. En esta clase de planificadores, el más comúnmente usado es el planificador basado en la posición del fragmento dentro de la pantalla. Este planificador tiene una ventaja clara, y es que no es necesario reordenar la salida de fragmentos, por lo que un fragmento A que ha entrado antes que un fragmento B, puede salir del *Fragment Processor* después de que se haya procesado el fragmento B sin alterar la imagen final. Dicho de otro modo, estos planificadores permiten la ejecución fuera de orden de los fragmentos sin que ello afecte a la imagen final.

A nivel *hardware*, este planificador simplifica mucho la lógica a implementar,

pues se ahorra una etapa que requiere de memoria para ordenar los fragmentos de salida. Esto es equivalente a asignar una región concreta de la pantalla a cada *Fragment Processor*. Dentro de esta clase de planificadores estáticos, encontramos diferentes esquemas de asignación, que atienden a un diseño (o *layout*) concreto de la pantalla. Por otro lado, los planificadores dinámicos, además de la información intrínseca de los fragmentos entrantes, también tienen en cuenta el estado actual del *pipeline* gráfico y otras variables externas, generalmente, las que tienen que ver con la etapa *Fragment Processing*. Dentro de esta clase de planificadores, el más conocido es el planificador basado en la ocupación de los *Fragment Processors*. Este planificador simplemente asigna el fragmento entrante al primer *Fragment Processor* libre y, en caso de empate, a aquél con el identificador más bajo. El problema que tiene este planificador es que la carga de trabajo no es homogénea entre *Fragment Processors*, pues se asigna siempre más cantidad al primero de todos (al que tiene identificador 0).

Otro planificador muy usado sigue una política *round robin*. Aquí, el control de asignación lo lleva una variable que se incrementa cada vez que llega un fragmento a la etapa *Fragment Processing*, y se reinicia cuando llega al máximo número de *Fragment Processors* de esta etapa. Con este planificador se soluciona el problema del reparto de la carga, pero tiene el inconveniente de que puede provocar esperas, pues si se le asigna un fragmento a un procesador que está ocupado, mientras hay otros libres, no se están aprovechando bien los recursos. Por eso, lo que se suele implementar es un esquema mixto de estos dos planificadores dinámicos. La ventaja de estos planificadores es que se cuenta con muchísima más información para planificar los fragmentos, pudiendo así aprovechar al máximo los recursos *hardware*, sin que queden *Fragment Processors* libres y balanceando lo máximo posible la carga de trabajo. Sin embargo, a diferencia de los planificadores estáticos, éstos requieren forzosamente de un mecanismo de reordenación de fragmentos que garantice el orden original de entrada, para no incurrir en errores en la imagen final. Esto requiere de estructuras ROBs (*ReOrder Buffers*), por ejemplo, que son muy costosas en *hardware* y energía, por lo que generalmente no se implementan en GPUs móviles, en las que las decisiones de diseño están dominadas principalmente por el consumo energético.

Por otro lado, dado que el procesamiento de fragmentos se realiza de una manera independiente, no es necesario *hardware* para sincronizar los *Fragment Processors*. Esto simplifica el diseño y la forma de programarlos.

En esta etapa es donde se realizan la gran mayoría de efectos que se encuentran en una escena, como por ejemplo, los modelos de sombreado, los reflejos dinámicos, o el coloreado de superficies. Probablemente, una de las funciones más importantes, usadas y costosas computacionalmente llevadas a cabo en esta etapa sea el proceso de texturizado. Una escena compuesta únicamente por polígonos con colores planos (o incluso sombreados con degradados) no da como resultado una imagen muy realista, pues da sensación de simplicidad. Es por esto que para dotar de más realismo a la escena se aplica una textura sobre estas superficies ya coloreadas y sombreadas. Una textura es un mapa de bits que almacena propiedades de la textura por píxel, como por ejemplo, el color, la profundidad o la normal en un determinado punto de la superficie. A cada píxel de una textura se le denomina *texel* (de *TEXture ELement*, elemento de la textura), que no es más que un color cuyo espacio depende de la definición de la textura (R, G, B , RG, RGB, RGBA...)[13]. El objetivo del texturizado es dotar de textura a los fragmentos de una superficie. O dicho de otro modo, el proceso de texturizado consiste en asignar un *texel* (o una mezcla de éstos) a cada fragmento de la superficie a dibujar. A este proceso se le denomina *texture mapping* (mapeo de textura). En la Figura 2.13 se puede ver en qué consiste dicho proceso.

En esta imagen tenemos una textura bidimensional de 8x8 *texels*, y un triángulo al que se le aplica esa textura. Como ya se ha visto anteriormente, los vértices llevan asociados consigo atributos. Un atributo importante para realizar el mapeo de texturas es el vector de coordenadas de texturas (o *texture coordinates*). Este atributo tiene las mismas dimensiones que la textura a mapear, e indica qué parte de la textura se va a mapear a ese vértice. Si nos fijamos en la Figura 2.13, vemos que a cada vértice le corresponde una coordenada en la textura. Como se ha mencionado antes, el *Rasterizer* se encarga de interpolar atributos, por lo que los fragmentos que hay intermedios cogen los atributos interpolados de cada vértice en función

[13]Existen esquemas más complejos, como por ejemplo el *bump mapping* [66, 47, 28], que incluyen profundidades y normales para dar una sensación más realista en presencia de efectos lumínicos.

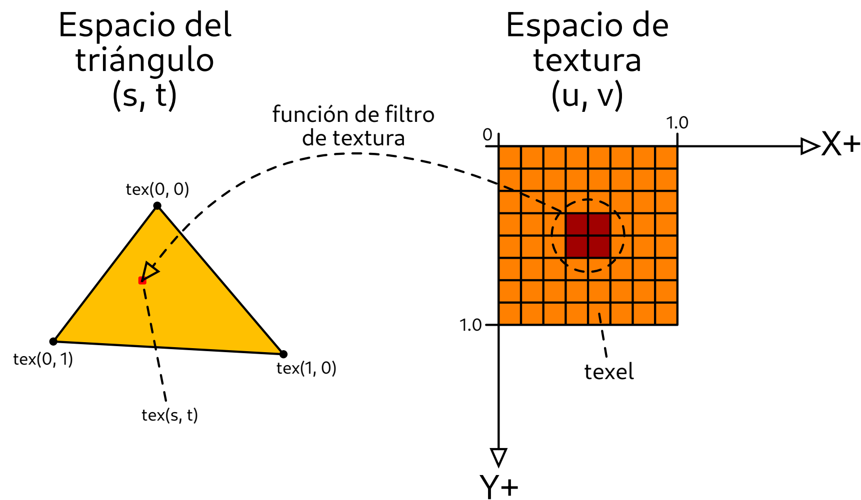


Figura 2.13: Ejemplo del proceso de *texture mapping* con una textura bidimensional.

de su posición. Las coordenadas de texturas también se interpolan, por lo que para acceder a un *texel* concreto de la textura basta con consultar este atributo en el fragmento. Este proceso puede verse como una conversión entre espacios de coordenadas, de coordenadas del triángulo (s, t) a coordenadas de la textura (u, v). Las coordenadas de una textura bidimensional se definen generalmente en el rango $[0, 1]$, significando el punto $(0, 0)$ la esquina superior-izquierda de la imagen, y el punto $(1, 1)$ la esquina inferior-derecha. Valores más allá o por debajo de la unidad se consideran de una manera u otra atendiendo al modo de envoltorio *wrapping mode* de la textura. Por ejemplo, se puede definir un patrón de repetición (`GL_REPEAT` en OpenGL) en el que sólo se escoge la parte decimal de la coordenada de textura, dando lugar a un patrón de mosaicos. Otro patrón es el de *clamping* (`GL_CLAMP_TO_EDGE` en OpenGL). Aquí, las coordenadas se acotan a los límites de la textura, por lo que si una coordenada se sale de los rangos, ésta se establece al máximo o al mínimo (según corresponda) de la textura.

Otra forma ampliamente usada para mapear texturas es la conocida como *cubemap*. Aquí las coordenadas de textura están definidas en tres dimensiones. El mapeo consiste en lo siguiente. Se define un cubo en el rango $[(-1, -1, -1), (1, 1, 1)]$ (como el cubo unitario), cuyo origen es el punto $(0, 0, 0)$. Para cada lado de este cubo se establece una textura bidimensional. Para seleccionar un *texel*, se traza una línea

recta cuyo vector director es la coordenada de textura, se computa la intersección con los 6 planos del cubo, y se escoge el punto más cercano.

Si por cada fragmento que se renderiza sólo escogemos un único *texel*, el efecto que se obtiene es muy poco realista, pues genera cambios muy bruscos de color en los que se ve claramente el cambio de *texel* (artefacto conocido como *blocking*). Por otro lado, un *texel* no siempre va a corresponder a un fragmento (y viceversa). Además, es muy poco probable que el punto seleccionado del *texel* caiga justo en el centro del mismo a la hora de hacer el mapeo de texturas. Por estos motivos es necesario el uso de filtros de texturas. Como se ha visto, dado un fragmento que contiene unas coordenadas de texturas se obtiene un *texel* accediendo a dichas coordenadas en el espacio de la textura. Pero si nos fijamos bien en la Figura 2.13, la conversión del espacio (s, t) al espacio (u, v) no siempre guarda una relación uno a uno. Esto quiere decir que un fragmento en el espacio (s, t) puede mapear varios *texels* del espacio (u, v), y viceversa, un *texel* en el espacio (u, v) mapee varios fragmentos del espacio (s, t). Para manejar estas situaciones se aplica lo que se conoce como un filtro de texturas, un proceso también conocido como *texture filtering*. La idea de este procedimiento consiste en usar la información, no sólo de un *texel*, sino de varios *texels* vecinos con el fin de interpolarlos y obtener un efecto mucho más gradual y realista. Dependiendo de la relación de mapeo, un filtro puede ser de minimización o de magnificación. Imaginemos una superficie muy cerca de la cámara. Aquí varios fragmentos mapean el mismo *texel*, por lo que se necesitaría un filtro de magnificación, también conocido como *texture magnification filter*. Existen varias opciones para realizar este mapeo, siendo el más sencillo el esquema *nearest-neighbor*, que consiste en escoger únicamente el *texel* cuyo centro sea más cercano a la coordenada mapeada. Este esquema genera el ya mencionado efecto de *blocking*. Otro esquema más realista de magnificación es el filtro bilineal o *bilinear filtering*. Esta función consiste en coger los cuatro *texels* vecinos más cercanos al punto mapeado, e interpolar sus colores en función de la distancia a cada uno de estos *texels*. Este filtro crea un efecto de gradiente entre *texels*, lo cual es un efecto mucho más suave y realista que el anteriormente visto. Ahora imaginémonos otro caso en el que un triángulo está lejos de la cámara. Acorde a la Figura 2.13 un fragmento en el espacio (s, t) correspondería a varios *texels* del espacio (u, v).

Para este tipo de situaciones se utiliza un filtro de minimización, también conocido como *texture minification filter*. Aquí lo que se utiliza es un método conocido como *texture mipmapping*, que consiste en generar texturas más pequeñas a partir de una textura original. A cada textura derivada se le asigna un nivel de detalle en función de su tamaño, siendo el nivel 0 la textura original. Cada nivel, las dimensiones de la textura se dividen entre 2. El número de niveles de *mipmap* los define el usuario (generalmente hasta que la textura de un nivel tenga un tamaño muy pequeño). La Figura 2.14 muestra un ejemplo de *texture mipmapping* en la que se parte de una textura de 512x512 *texels* y se generan 4 niveles de *mipmap*. El nivel 1 es una textura de 256x256 *texels*, el nivel 2 es una textura derivada de 128x128 *texels*, y el nivel 3 es una textura de tan sólo 64x64 *texels*. Para aplicar un filtro de minimización primero se tiene que determinar el nivel de detalle, también conocido como *Level Of Detail* (LOD), que irá en función de la distancia al plano *near* y a la separación de los vértices del triángulo a renderizar. Una vez determinado el LOD, se accede a la textura del nivel correspondiente y se obtiene el *texel*. Si nos fijamos, aunque hayamos accedido a un único *texel*, el color resultante realmente se ha obtenido anteriormente de varios téxels porque se han obtenido texturas más pequeñas aplicando una interpolación de colores. Es importante destacar que, además del efecto conseguido, el *mipmapping* ofrece grandes beneficios al proceso de renderizado, pues al acceder a texturas más pequeñas el patrón de mapeo es más *cache friendly* que si se accediera a la textura de resolución completa. Esto significa que se aprovecha mucho mejor la localidad espacial de las texturas accedidas durante el mapeo y, por lo tanto, el proceso de mapeo de texturas es mucho más eficiente.

Un filtro de textura más sofisticado es el filtro trilineal, también conocido como *trilinear filtering*. Este filtro consiste en determinar un LOD intermedio entre dos niveles, es decir, ahora el LOD no es una variable discreta (0, 1, 2...) sino que es una variable continua, lo que quiere decir que queda comprendida entre dos niveles. Esta determinación se consigue del mismo modo, es decir, en función de la distancia al plano *near* y la separación de los vértices del triángulo. Lo que se hace es computar en cada nivel un filtro bilineal, con lo que se obtienen dos colores y, luego, estos colores se interpolan en función del LOD continuo. Ni qué decir tiene que este filtro es más complejo que los anteriores, pero ofrece una calidad de imagen

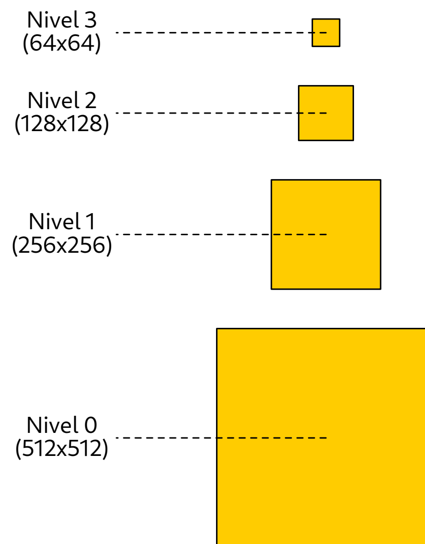


Figura 2.14: Ejemplo de *texture mipmapping* con 4 niveles de detalle (LOD).

muy superior[14].

Aunque los filtros bilineal y trilineal sean muy completos sufren de un problema de resolución cuando las primitivas que se visualizan son muy oblicuas a la cámara. En este aspecto estos filtros para los fragmentos lejanos escogerán niveles de detalle muy bajos, penalizando la resolución en el eje x. Para solucionar este problema se usa lo que se conoce como filtro anisotrópico (o *anisotropic filter*) que, aparte de generar los mimaps de diferentes resoluciones, también genera diferentes resoluciones no cuadradas de la imagen para cada eje, dando lugar a niveles de *mipmap* rectangulares.

Finalmente, otra función importante que puede desempeñar un *fragment shader* es la eliminación de fragmentos condicional o incondicionalmente, algo conocido también como *fragment killer*.

Como se puede ver, esta etapa es muy compleja, lo que implica que su consumo energético es muy elevado. Es por esto que hay una etapa previa que trata de eliminar de manera prematura fragmentos que no son visibles, la etapa *Early Z-Test* (véase la Sección 2.1.7).

[14] Al igual que sucede en muchos aspectos que tienen que ver con la calidad de la imagen, en lo que se refiere a filtros de texturas también existe un *trade off* entre rendimiento y calidad.

2.1.9. *Late Z-Test*

Aunque ya exista un test de visibilidad previo a la etapa *Fragment Processing*, en el proceso se pueden modificar las profundidades de los fragmentos por medio de la invocación de un *fragment shader*. Esto, automáticamente, anularía el test de visibilidad previo (*Early Z-Test*) dado que estaría usando información desactualizada y, por lo tanto, el resultado de dicho test no sería fiable. Esto podría dar como resultado una imagen errónea. Es por eso, que para este tipo de casos, es necesario un test de visibilidad posterior al *fragment shader*, también ampliamente conocido como *Late Z-Test*. Para lograr tal modificación, el lenguaje de *shading* subyacente debe exponer al programador una interfaz, función o variable de contexto que le permita actualizar dicho valor. En OpenGL, esto se hace modificando la variable de salida del *fragment shader* `gl_FragDepth`. Dado que este patrón invalida el *Early Z-Test*, es necesario proveer al *pipeline* de un método para saltarse dicho test y pasar directamente a la etapa *Fragment Processing*. Veámoslo con dos ejemplos ilustrativos. Imaginemos que tenemos dos objetos, a saber, una pelota y una pared. La pelota inicialmente se encuentra delante de la pared, por lo que es visible para la cámara. Si primero dibujamos la pared y luego la pelota, entonces se producirá *overdraw* debido a que los fragmentos de ambos objetos pasarían el *Early Z-Test*, pero la pelota se vería delante de la pared. Ahora, supongamos que durante la etapa *Fragment Processing*, un *fragment shader* hace que la pelota de repente esté por detrás de la pared. Dado que los fragmentos de la pelota ya han pasado el *Early Z-Test*, ésta sería visible pese a que esté detrás de la pared. Este efecto, obviamente, es indeseable y es por eso que se usa esta etapa de test de visibilidad posterior. Ahora supongamos el ejemplo contrario, es decir, la pelota está detrás de la pared. El orden de dibujado sigue siendo el mismo. Aquí sucedería que en el momento de dibujar la pelota, ningún fragmento pasaría el *Early Z-Test* puesto que ya habría fragmentos más cercanos a la cámara en el Z-Buffer, los de la pared. Entonces, si más tarde el *fragment shader* de la pelota quisiera modificar los valores de profundidad de sus fragmentos, no tendría oportunidad dado que no llegarían a la etapa de *Fragment Processing*, provocando así, nuevamente, una imagen errónea. Así pues, para evitar dicha situación el *driver* ofrece un mecanismo de desactivación del *EarlyZ-Test*, que consiste en deshabilitarlo cuando se detecte un *fragment shader* de este tipo en el

comando a procesar. De este modo, si se detecta que la variable `gl_FragDepth` se modifica en el código del *fragment shader*, entonces se deshabilita la etapa *Early Z-Test*, por lo que los fragmentos entrantes a dicha etapa pasan directamente a la etapa de *Fragment Processing*. Basta con que aparezca una sentencia que modifique esta variable para que se deshabilite el *Early Z-Test*, incluso si el código es inalcanzable durante el flujo de ejecución.

Otro factor a tener en cuenta es que si en el *fragment shader* se detecta una sentencia que modifique de manera estática esta variable, ésta debe ser escrita en todos los caminos del código, de modo que cuando acabe la ejecución se haya modificado de manera explícita esta variable. Por ejemplo, si en una sentencia `if` modificamos esta variable, tenemos que modificarla explícitamente también en el camino del `else`.

Para no perder mucho potencial en el *Early Z-Test*, a partir de la versión 4.20 de GLSL (*OpenGL Shader Language*) se implementó una mejora que permite indicar de antemano qué valor tendrá la variable `gl_FragDepth` respecto a la profundidad original del fragmento tras ejecutarse el *fragment shader*. Esta condición se indica mediante un modificador durante la declaración de la variable en el *fragment shader*. Las posibles condiciones son las siguientes:

- `any`: no hay restricciones respecto al valor final de la variable `gl_FragDepth`. Ésta sería equivalente a no aplicar ninguna optimización.
- `greater`: la variable `gl_FragDepth`, tras la ejecución del *fragment shader* será superior a la profundidad original del fragmento.
- `less`: tras la ejecución del *fragment shader*, la variable `gl_FragDepth` será inferior a la profundidad original del fragmento.
- `unchanged`: la variable `gl_FragDepth` será igual a la profundidad original del fragmento.

Ni qué decir tiene que si se viola esta condición durante la ejecución del *fragment shader* el comportamiento de éste respecto a las profundidades es indefinido y, por lo tanto, no fiable. Es decir, si por ejemplo declaramos en el *fragment shader*

que el valor de `gl_FragDepth` va a ser menor que la profundidad original tras la ejecución del *fragment shader*, y tras ejecutarse resulta que esta variable es mayor que la profundidad original del fragmento, entonces el resultado es indefinido, es decir, muy probablemente erróneo. Esto se debe a que el compilador de *fragment shaders*, junto con el *hardware* subyacente usan estos *hints* para optimizar ciertas situaciones por medio de asertos y cribar más fragmentos en esta etapa. Es por eso que el programador debe evitar, en la medida de lo posible, usar el *hint* `any`, puesto que esto no le dice absolutamente nada (de cara a optimizar) ni al compilador de *fragment shaders* ni al *hardware*.

En definitiva, a pesar de que esta etapa no mejora el rendimiento, como sí lo hace la etapa *Early Z-Test*, garantiza la visibilidad de los fragmentos dando lugar a una imagen correcta. Esta etapa también usa el mismo Z-Buffer que la etapa *Early Z-Test*, de hecho, la funcionalidad que implementa es exactamente la misma, pero aplicada en un momento del *pipeline* gráfico diferente.

2.1.10. *Color Blending*

Esta etapa se encarga de mezclar el color entrante proveniente de la etapa *Fragment Processing* con el contenido de lo que se conoce como *Color Buffer*.

A esta etapa ya no llegan fragmentos como tal, sino que llegan colores en el espacio RGBA, que es el resultado de aplicar la función de *shading* a un fragmento. Aquí sólo llegan los colores cuyos fragmentos son visibles en la escena hasta el momento, para lo cual se ha efectuado tanto el *Early Z-Test*, como los *fragment killers* pertinentes, como el *Late Z-Test*.

La función que desempeña esta etapa es la de mezclar los colores entrantes con el *Color Buffer* existente. El *Color Buffer* es una estructura *hardware* que almacena una imagen bidimensional formada por colores, generalmente en el espacio RGBA, que se corresponde con la información que muestra finalmente el dispositivo de visualización, como puede ser la pantalla de un ordenador o un *smartphone*. Como se verá en la Sección 2.1.11, el *Color Buffer* es un *buffer* más de lo que se denomina *framebuffer*.

Aquí se comprueba otra propiedad asociada al comando que se está renderizando, que es el atributo *blending enabled*, que indica si se debe hacer dicha mezcla o no. Si este *flag* está desactivado, entonces el color se escribe directamente en el *Color Buffer*, sin tener en cuenta el color que había previamente en esa posición. Si está activado, entonces se procede a hacer la mezcla, para lo cual se consultan otras propiedades que determinan la forma en la que se hace la mezcla de los colores. En esta etapa también se implementan las transparencias de las primitivas (atendiendo a si tienen el *flag blending enabled* activado), pero no es su único uso.

Para entender el concepto de *blending* hay que tener en cuenta tres componentes de las que depende la función de *blending*, a saber: color de entrada, color ya existente en el *Color Buffer* y función de *blending*. La función de *blending*, como su nombre sugiere, consiste en mezclar los dos colores de entrada ya mencionados aplicando una función, dando lugar a un color de salida que se escribe en la posición correspondiente del *Color Buffer*. Existen muchas funciones de *blending* y configuraciones. Una de las más comunes es el *alpha blending*, que se encarga de generar los efectos de transparencias propiamente dichos, atendiendo al canal *alpha* de un color en el espacio RGBA. Cuanto más alto sea el factor *alpha*, más opaco es el fragmento, siendo 1 la opacidad total. Por el contrario, cuanto más bajo sea el factor *alpha*, más transparente será el fragmento, siendo 0 la transparencia absoluta.

2.1.11. *Framebuffers*

Cuando hablamos de renderización hablamos de dibujar objetos en la pantalla del dispositivo, pero realmente, durante el proceso de renderización podemos decidir en qué *lienzo* dibujar los resultados. Esto quiere decir que no es necesario en absoluto que la escena se dibuje inmediatamente en la pantalla del dispositivo[15]. Estos *lienzos* reciben en OpenGL el nombre de *framebuffers* y están compuestos por varias estructuras de datos que se utilizan durante el proceso de renderización. Dado que estos *buffers* se usan para renderizar escenas, éstos se conocen también bajo el nombre de *render targets* (objetivos de renderizado). Existen dos tipos de *frame-*

[15] De hecho, no es lo común, pues en vez de dibujar la escena sobre la marcha, se usa un *buffer* intermedio donde se dibuja toda la escena y, una vez completado el renderizado, se vuelca al *framebuffer* de la pantalla. A esta técnica, como se verá más adelante se le denomina *double buffering*.

buffer en OpenGL, que son los que se explicarán en esta Sección.

Por defecto, durante la creación del contexto de OpenGL se crea lo que se denomina el *default framebuffer*, que es el que se usará para volcar la imagen final en la pantalla del dispositivo. Por lo tanto, este *framebuffer* está fuertemente acoplado al dispositivo de visualización. Asimismo, los parámetros que lo definen, como las dimensiones, el formato de color o la profundidad, dependen intrínsecamente del sistema subyacente, que generalmente es el sistema de gestión de ventanas o el decorador de la interfaz gráfica que, a su vez, también está acoplado al *hardware* dispositivo de visualización. Estos parámetros también se determinan durante la creación del contexto de OpenGL[16]. Por lo tanto, este *framebuffer* se asocia directamente a la pantalla del dispositivo[17], y todo lo que se renderice en éste será finalmente dibujado en dicha pantalla. El *default framebuffer* contiene una serie de *buffers* que se usan durante el proceso de renderización, a saber: el Color Buffer, el Z-Buffer, el Stencil Buffer, el Multisample Buffer y *buffers* auxiliares (`GL_AUXi`, donde i es el número de identificación del *buffer* auxiliar). Las dimensiones de estos *buffers* son siempre las mismas y se asocian a las del *framebuffer* al que pertenecen y, como ya se ha mencionado, se determinan en el momento de la creación del contexto de OpenGL. A diferencia de las dimensiones, los formatos de píxel de cada uno de los *buffers* de un *framebuffer* pueden ser (y generalmente lo son) diferentes los unos de los otros.

A continuación pasamos a explicar los diferentes *buffers* de los que se compone el *default framebuffer*.

El *Color Buffer* del *default framebuffer* está compuesto por varios *sub-buffers*, a saber: `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT` y `GL_BACK_RIGHT`. Aquéllos con el denominativo `FRONT` corresponden a los *buffers* que se visualizan directamente en la pantalla del dispositivo. Por el contrario, aquéllos con el distintivo `BACK` corresponden a los *buffers* que se renderizan en una memoria interna del dispositivo y que, por lo tanto, no tienen un efecto directo o inmediato sobre el dispositivo de visualización. El motivo de que existan estas dos clases de *buffers* es

[16]Por ejemplo, en EGL se hace mediante la llamada `eglCreateContext`.

[17]Realmente no se asocia directamente, sino a una parte de lo que el programador defina como *viewport* mediante la llamada a `glViewport`, que limita a un rectángulo el rango en el que se va a dibujar el *framebuffer* en el dispositivo de visualización.

para implementar lo que se conoce como *double buffering*. Si dibujásemos directamente todo lo que se va renderizando en la pantalla sobre la marcha, nos daríamos cuenta de ciertos artefactos, pues el contenido del *framebuffer* se visualiza en la pantalla por intervalos de tiempo que el programador no puede controlar. Esto se debe a que, generalmente, la sincronización vertical (también conocida como *vsync*) de la pantalla se lleva a un nivel muy bajo entre el controlador situado físicamente en la pantalla y el controlador de señales de vídeo (MIPI-DSI, LVDS, DP, HDMI, etc.) de la GPU del dispositivo. El problema reside en que esta señal de sincronización se genera o en el controlador de la pantalla o en el controlador de señales de vídeo, puesto que son los encargados de generar y garantizar el *timing* de excitación eléctrica de las filas (*source drivers*) y las columnas (*gate drivers*) del panel de visualización. Esto significa que la GPU es la responsable de generar una interrupción para manejar de forma correcta esta señal[18]. Por lo tanto, si quisiéramos evitar artefactos en la imagen final, el programador tendría que capturar dicha interrupción para dibujar toda la escena en ese momento. Sin embargo, si el tiempo en renderizar la escena es superior al de refresco de la pantalla (sincronización vertical) ésta mostraría la escena a medio renderizar. Generalmente, estos artefactos se traducen en parpadeos en la pantalla repentinos muy desagradables, algo que también se denomina *flickering*. Por lo tanto, una buena solución a este problema pasa por que el proceso de renderización de la imagen y la visualización del *framebuffer* en la pantalla se realicen de manera asíncrona, es decir, por separado y sin comunicación alguna. Para lograrlo, las GPUs modernas incluyen dos *buffers*, y así permitir al programador implementar el *double buffering*. Esta técnica consiste en renderizar toda la escena en un *back buffer* y, una vez finalizado el proceso de renderización, se vuelca todo el contenido de este *back buffer* al *front buffer* mediante una llamada explícita por parte del programador[19]. Esta llamada lo que hace es intercambiar los punteros (en vez de copiar todo el contenido) de los *buffers* en cuestión, y sincronizar de manera explícita el proceso de renderizado con los intervalos de refresco de la pantalla (el ya mencionado *vsync*). Es importante destacar que el *double buffering* no previene artefactos en la imagen de manera automática, pues si a mitad

[18]Esto se hace generalmente a nivel de *kernel*.

[19]Por ejemplo, en EGL (un sistema de gestión de contextos de OpenGL) se usa la llamada explícita `eglSwapBuffers`.

de visualizar la imagen en la pantalla se hace el intercambio de *buffers*, la mitad (o un porcentaje) de la pantalla mostrará el contenido de un *buffer* mientras que la otra parte mostrará el otro *buffer*, creando un efecto extraño. A este artefacto se le conoce como *tearing effect* o *screen tearing*, y no está limitado a sólo dos *buffers*. Para solucionarlo, el *driver* tiene que saber el momento en el que la pantalla accede al *buffer* para leerlo, de modo que evite hacer el intercambio de punteros en ese momento[20]. Por otro lado, los *buffers* con el denominativo `LEFT` o `RIGHT` se utilizan para lo que se conoce como *stereo rendering*, que generalmente se utiliza en el ámbito de la realidad virtual, en la que se necesitan dos imágenes para que cada ojo del sujeto visualice una escena con un pequeño desplazamiento, y así, dar la sensación real de profundidad de la escena. En *rendering* normal, es decir, no *stereo rendering*, se usan únicamente los dos *buffers* con el denominativo `LEFT`, ignorando por completo aquéllos con el denominativo `RIGHT`.

Por otro lado, el Z-Buffer contiene las profundidades de los fragmentos que han superado el Z-Test hasta el momento durante la ejecución del *pipeline* gráfico (véase la Sección 2.1.7).

El *Stencil Test* se utiliza como un test adicional de visibilidad (al Z-Test) que se aplica a los fragmentos usando un *buffer* alternativo denominado *Stencil Buffer*. Este test tiene lugar justo después de la etapa *Fragment Processing*, en la etapa *Late Z-Test*, y consiste en descartar fragmentos en función del valor de *stencil* almacenado en la posición correspondiente del *Stencil Buffer*. Generalmente, se construye primero este *Stencil Buffer* mediante *draw calls* que no alteran el estado final del *Color Buffer*, porque se fuerza (mediante la función `GL_NEVER`) a que nunca pasen el *Stencil Test*, pero sí a que modifiquen el *Stencil Buffer*, y luego se emiten *draw calls* con el *Stencil Test* activado para hacer el descarte de fragmentos con el *Stencil Buffer* ya construido.

Hay que tener en cuenta que mientras que el Z-Test se evalúa mediante el valor de profundidad de un fragmento concreto y el correspondiente valor en el Z-Buffer, en el *Stencil Test* se evalúa únicamente el valor de *stencil* que haya en la posición

[20] Si el controlador de la pantalla contiene internamente un *framebuffer* (generalmente denominado GRAM), entonces éste tiene que proveer con una señal el momento en el que es seguro transferir el contenido de la imagen. A esta señal generalmente se le denomina bajo el nombre Tearing Effect Line, TEE o VSYNC.

correspondiente del *Stencil Buffer*, sin mirar ningún atributo del fragmento. Esto se debe a que el valor de referencia con el que se realiza la comparación siempre es el mismo durante la *draw call*, pues se ha establecido previamente mediante la función `glStencilFunc`.

2.2. Cómo se implementa el *pipeline* gráfico en *hardware*

Ya hemos visto el modelo conceptual y funcional del *pipeline* gráfico, pero no hemos visto los aspectos *hardware* relacionados con su implementación. En este sentido, existen varias maneras de diseñar y configurar el *pipeline* gráfico, de las cuales las más conocidas son *Immediate Mode Rendering* (IMR) y *Tile-based Rendering* (TBR) que, como veremos, cada una tiene sus ventajas e inconvenientes.

2.2.1. Arquitecturas *Immediate Mode Rendering* (IMR)

La forma más directa que existe de implementar el *pipeline* gráfico, atendiendo al modelo funcional del mismo, es *Immediate Mode Rendering* (IMR). Como su nombre indica, es un modo de renderizado directo, lo que significa que tal cual van llegando los comandos emitidos por el programa principal en la CPU, se van procesando en la GPU y renderizando en el *render target* correspondiente. Si bien es cierto que esta implementación obtiene grandes beneficios en el rendimiento, requiere de una gran cantidad de recursos y memoria durante el proceso de renderizado, lo cual se traduce en un alto consumo energético. Es por esto que este tipo de arquitectura se implementa en GPUs de alto rendimiento, generalmente de equipos de escritorio, y no en GPUs para dispositivos móviles.

La Figura 2.15 muestra una implementación de la arquitectura IMR, con todas sus etapas y memorias necesarias para llevar a cabo el proceso de renderizado. En las siguientes subsecciones se verá la implementación interna de cada una de sus etapas.

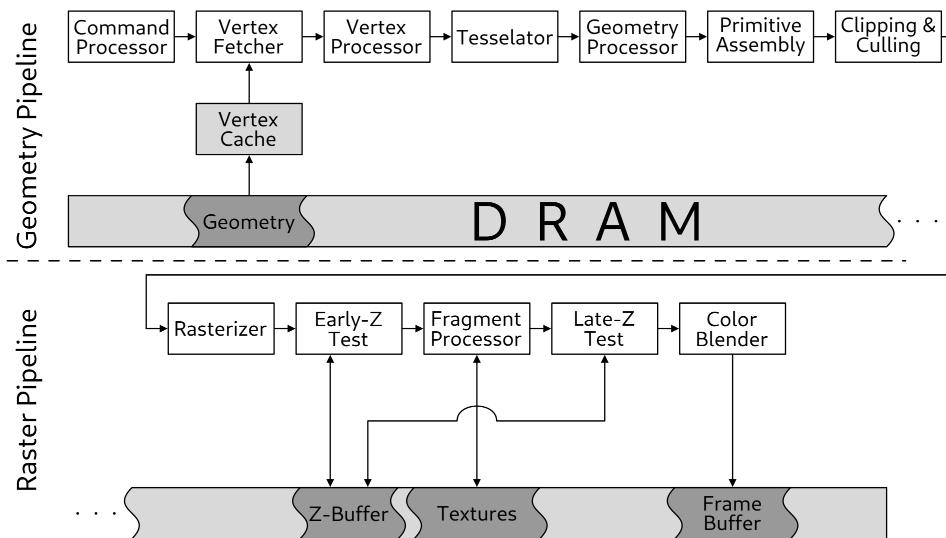


Figura 2.15: Implementación del *pipeline* gráfico de una arquitectura basada en IMR.

Etapa *Command Processor*

El *pipeline* gráfico empieza, como no podía ser de otra manera, con el *Command Processor*. Como ya se ha visto en la Sección 2.1, el estado de la GPU para renderizar se prepara con comandos, en concreto, comandos de estado. Por otro lado, los modelos que se van a renderizar finalmente en la pantalla también son comandos, en concreto, *draw calls*. Las GPUs normalmente implementan internamente su protocolo o lenguaje interno con el que comunicarse con el *pipeline* gráfico. Esto quiere decir que el *pipeline* gráfico y todo el procesamiento que se requiere durante su ejecución no tiene por qué estar acoplado a una API concreta como puede ser OpenGL, Vulkan o DirectX. De ser así, el *pipeline* gráfico tendría que entender cada una de estas APIs, algo que complicaría mucho la implementación de cada etapa y, por supuesto, la complejidad del *hardware*. Es por esto que existe esta unidad, que hace de interfaz entre la API que el programador conoce y el lenguaje interno que el *pipeline* gráfico entiende[21].

Anteriormente, se ha visto que para renderizar una escena es necesario un contexto de renderización. Sin embargo, es muy común que en un sistema existan va-

[21] Si bien es cierto que esta interfaz ya la ofrece el *dirver* del fabricante concreto, hay muchos aspectos *hardware* que tienen que llevarse a cabo en esta etapa.

rios contextos donde se dibujan cosas muy dispares. Por ejemplo, en un PC con un sistema operativo se puede usar un contexto OpenGL para renderizar el sistema de ventanas y la interfaz gráfica mientras que, dentro de una ventana, se renderiza la escena de un videojuego. Como se puede apreciar, en esta situación tenemos dos contextos corriendo simultáneamente. Así pues, esta etapa no sólo tiene que ser capaz de mantener y controlar el estado de los contextos existentes, sino también de planificarlos. Por lo tanto, es en esta etapa donde se llevan a cabo los cambios de contexto de OpenGL[22]. Este proceso de gestión y planificación está asistido por el *driver*.

Una vez preparado el contexto, se emite una *draw call*, con la que se activa el *pipeline* gráfico para renderizar el modelo concreto.

Etapa *Vertex Fetcher*

La *draw call* que emite el *Command Processor* de por sí no contiene la información del modelo, sino que contiene un índice a una estructura de datos donde se almacenan todos los modelos del contexto. Dicha estructura de datos se encuentra en la memoria principal de la GPU dada la alta demanda que un modelo puede requerir. Para obtener los datos del modelo a renderizar es, por lo tanto, necesario acceder a memoria principal. Para este propósito está el *Vertex Fetcher*, una etapa que dado el índice a la estructura de datos que contiene el modelo, accede a ella buscando atributos de vértices y los pasa a la cola que conecta con la etapa de *Vertex Processing*.

Para acelerar el proceso de búsqueda se hace uso de una caché denominada *Vertex Cache*, que a su vez está conectada a una caché de segundo nivel (L2), tal y como se puede apreciar en la Figura 2.15. Aparte de usar una caché, esta etapa no emite vértices por separado, sino que éstos los agrupa en grupos de cuatro vértices denominados *quad vertices*. Como veremos en la siguiente etapa, el motivo de esta agrupación tiene que ver con el rendimiento y el aprovechamiento de las unidades vectoriales.

[22] Aparte del *pipeline* gráfico, existe lo que se conoce como *pipeline* de computación o *compute pipeline*. Para hacer uso del mismo se crea otro tipo de contexto que esta etapa es capaz de manejar.

Etapa *Vertex Processing*

Esta etapa está formada por varios procesadores que se encargan de manejar los *quad vertices* de entrada. A estos procesadores se les conoce como *Vertex Processors*, y actúan de manera completamente independiente los unos de los otros, lo que significa que un *quad vertex* es procesado únicamente por un *Vertex Processor*. Un *Vertex Processor* está compuesto por varias unidades aritmético-lógicas (o ALUs) y unidades de punto flotante (o FPUs) organizadas generalmente como unidades vectoriales SIMD (*Single Instruction Multiple Data*), lo que significa que con la misma instrucción se pueden manejar varios vértices al mismo tiempo. Este es el motivo por el que los vértices se agrupan en *quad vertices*. Obviamente, el número de vértices de los grupos dependerá del ancho de estas unidades vectoriales. Por ejemplo, si nuestras unidades vectoriales SIMD son de 8 unidades funcionales, entonces los vértices los agruparíamos en grupos de 8.

Cada *Vertex Processor* es capaz de ejecutar paralelamente un número determinado de *threads*, que a su vez, están agrupados en *warps* (bloques de *threads*). Dado que en esta etapa no es necesario compartir datos entre *threads* y, de hecho no existe ninguna memoria compartida para tal propósito debido a la simplicidad, todos los *threads* del *Vertex Processor* se agrupan en el mismo *warp*.

Internamente, los *threads* se gestionan mediante un planificador que implementa el *Vertex Processor*.

Etapa de teselación

Como se introdujo más arriba, las teselaciones son una forma de aumentar la cantidad de geometría de un objeto sin necesidad de aumentar las necesidades de almacenamiento. De este modo, con muy poca memoria y un buen algoritmo de teselación podemos obtener objetos con mucho detalle, obviamente, haciendo uso conjuntamente del los *geometry shaders*. Esta etapa la lleva a cabo el *Tessellator*.

A nivel *hardware*, esta unidad requiere de unidades de división en punto flotante para poder dividir líneas y superficies. Aquí básicamente el *Tessellator* hace lo mismo que el *Primitive Assembly*, que es generar primitivas pero, esta vez, a partir

de vértices generados sobre la marcha.

Por otro lado, se necesitan procesadores para ejecutar tanto los *Tessellation Control Shader* como los *Tessellation Evaluation Shader*. Éstos pueden ser los mismos, pues se ejecutan en etapas diferentes de la teselación.

Etapa *Geometry Processing*

En esta etapa nos encontramos nuevamente con procesadores, pero esta vez procesadores de geometría, conocidos como *Geometry Processors*. Este tipo de procesadores, aparte de tener un conjunto de instrucciones para modificar los atributos de los vértices de las primitivas entrantes, también dispone de directivas para generar primitivas sobre la marcha. Hay que tener en cuenta que lo que llega a esta unidad son primitivas, no vértices ni *quad vertices* independientes, por lo que los vértices vienen implícitamente en paquetes de vértices que definen la primitiva. Como una primitiva puede tener un número indefinido de vértices, es necesario acordar el número máximo de vértices que va a tener una primitiva que entre en el *Geometry Processor*. Esto se define en el *geometry shader* de antemano con el atributo `max_vertices`, de modo que cualquier paquete que exceda ese número provocará un comportamiento indeterminado en la ejecución del *geometry shader*. Del mismo modo, para las primitivas de salida también se tiene que definir este límite superior. Así pues, el *hardware* implicado puede asegurar el almacenamiento para estas estructuras, de modo que si se excediera este límite, saltaría un error durante la ejecución, y se abortaría la ejecución del programa.

Otro aspecto a tener en cuenta de esta etapa es que los *quad vértices* dejan de existir, es decir, el simple hecho de formar primitivas ya agrupa vértices, por lo que agruparlos a su vez en *quad vertices* carece de sentido, aparte de que podría llegar a ser incluso ineficiente. Imaginemos por ejemplo, primitivas que sólo estuviesen compuestas por triángulos. En tal caso, los *quad vertices* contendrían tres vértices y el último se desecharía, dando lugar así a un desperdicio de recursos. Es por esto que, a pesar de que los *Geometry Processors* sean capaces de modificar los atributos de los vértices de las primitivas que manejan, es recomendable hacer dichas transformaciones en los *vertex shaders*, que se benefician mucho más de las unidades

vectoriales SIMD.

Etapa *Primitive Assembly*

Como se ha visto en la Sección 2.1.4, esta etapa se encarga de triangular la geometría entrante, es decir, convertir en triángulos las primitivas que le llegan a la entrada. El *hardware* de esta etapa es relativamente simple, pues lo único que tiene que hacer es utilizar una función de ensamblado cuyos datos de entrada son, por un lado, el tipo de primitiva que viene dado por el comando emitido (la *draw call* emitida por el *Command Processor*) y, por otro lado, los vértices de la primitiva de entrada. En un registro interno lleva la cuenta del vértice del triángulo que está generando. Por otro lado, tiene un registro capaz de almacenar tres vértices, que son los que compondrán el triángulo. Aquí hay que tener en cuenta que las necesidades de almacenamiento de un vértice vienen dadas por el número de atributos, por lo que es importante que esta estructura contenga un máximo de atributos que garantice que se pueden almacenar los tres vértices sin problemas. Para poder garantizar esto, el programador tiene la responsabilidad de consultar de antemano el número máximo de atributos soportados por el formato de los vértices. En OpenGL esto se hace con una llamada a `glGetIntegerv` con el atributo *pname* establecido a `GL_MAX_VERTEX_ATTRIBS`. OpenGL garantiza un mínimo de 8 atributos por vértice, por lo que si el formato de los vértices que ha definido el programador está por debajo de este umbral, no es necesario consultar dicha información.

Cuando el registro interno de esta etapa llega a tres, entonces significa que la estructura interna del triángulo está lista, por lo que el triángulo se emite a la siguiente etapa.

Por supuesto, no hay que despreciar tampoco las primitivas basadas en líneas y puntos, pues el *hardware* necesita también manejar este tipo de primitivas. Dado que éstas son más simples, el *hardware* que se ha visto antes se reutiliza para este propósito.

Etapa *Clipping & Culling*

Esta etapa requiere un poco más de *hardware* que la anterior para realizar los cortes de los triángulos entrantes. Realmente, esta etapa hace algo parecido a la etapa *Geometry Processing*, pero con triángulos ya formados. De este modo, dado un triángulo esta etapa puede dejarlo como estaba, generar más de un triángulo o eliminarlo, dependiendo de si éste se encuentra completamente dentro de la pantalla, parcialmente dentro o completamente fuera de la pantalla, respectivamente. La complejidad de esta etapa viene dada por el algoritmo de recorte que ajusta los triángulos dentro de la pantalla (véase la Sección 2.1.5).

Etapa *Rasterizer*

A la hora de implementar el *Rasterizer* en *hardware* hay que tener en cuenta cómo se van a generar los fragmentos del triángulo. Como se ha visto en la Sección 2.1.6, existen muchas formas de rasterizar un triángulo, de las cuales una de las más conocidas es generar únicamente los fragmentos que forman parte exclusiva del triángulo haciendo uso del algoritmo de Bresenham [14]. Este algoritmo se considera liviano a nivel *hardware* porque no hace uso ni de unidades de multiplicación ni de división y, además, no requiere de operaciones de punto flotante. Sin embargo, este algoritmo es para dibujar líneas, no para rellenar triángulos, por lo que se usa una variante de éste adaptada para triángulos, como el método del *scanline* [2].

En este aspecto, lo primero que hay que hacer es ordenar los vértices en orden descendente, de modo que se pueda aplicar correctamente el escaneo. Esto conlleva almacenar, al menos, tres punteros para los vértices, que sabiendo que son tres se pueden representar con dos *bits*. Por lo tanto, es necesario un registro de 6 *bits* que determine este orden. Por otro lado, es necesario un algoritmo que ordene estos triángulos. Una forma de hacer este proceso sin que ello implique mucho coste computacional ni complejidad de diseño es hacer tres recorridos sobre las coordenadas de los vértices, de modo que primero se busca el más alto y se coloca al principio del *buffer* de punteros a vértice. Luego se hace lo mismo pero con el del

centro y el más bajo. Existen más casuísticas que el *hardware* tiene que controlar, como por ejemplo, qué pasa si dos vértices del triángulo están a la misma altura y son los más altos. Respecto al recorrido del *scanline*, lo primero que hay que hacer es colocar la *scanline* (también llamada línea de rastreo) arriba del todo, en la cima del triángulo. Esto lo sabemos accediendo al *buffer* de punteros a vértice del que hemos hablado. El que esté en la primera posición es el más alto, por lo que colocamos nuestra línea de rastreo a esa altura. Ahora, lo que se hace es computar los valores incrementales de Bresenham de cada línea que queda a la izquierda y a la derecha, de modo que a medida que el *scanline* baje se actualicen acordemente los extremos del segmento a rasterizar.

Por otro lado, hay que computar las coordenadas baricéntricas. Esto tiene como coste computacional hacer tres productos y un par de sumas. Esto habría que hacerlo por cada fragmento que se genera, sin embargo, esto supondría un coste computacional muy alto. Para solucionar este problema lo que se hace es aprovechar los incrementos de Bresenham, de modo que se calculan las coordenadas baricéntircas una vez, y el resto se van incrementando a medida que se mueve la *scanline*.

Etapa *Early Z-Test*

La implementación *hardware* de esta etapa es relativamente sencilla, pues a nivel funcional lo que tiene que hacer es comparar los valores de profundidad de los fragmentos entrantes con aquéllos que se encuentran en un *buffer* de profundidades. Como ya se ha mencionado en la Sección 2.1.7, a este *buffer* se le denomina Z-Buffer y almacena la profundidad del último fragmento que pasó el Z-Test. A su vez, el Z-Test es una función fija que determina si el fragmento pasa (y actualiza el Z-Buffer en consecuencia) a la siguiente etapa en función de su profundidad y la almacenada en la misma posición del Z-Buffer.

Por lo tanto, el *hardware* sólo necesita acceder para leer la posición concreta del Z-Buffer, hacer una comparación aplicando la función de Z-Test y, en función del resultado de ésta, escribir el valor de profundidad del fragmento entrante en la misma posición del Z-Buffer y pasar el fragmento a la siguiente etapa.

Con el fin de ocultar latencias de acceso a la memoria subyacente, lo que se hace es usar una tabla que por cada fragmento entrante almacene el estado en el que está su petición a memoria, de modo que pueda haber varios fragmentos evaluándose en vuelo sin provocar detenciones considerables en el *pipeline*. Obviamente, esta tabla tiene un tamaño limitado, y depende de la productividad del *pipeline*, por lo que tiene que ser dimensionada acordemente.

Por su parte, la etapa *Late Z-Test* tiene la misma complejidad que ésta, por lo que no la vamos a explicar.

Etapa *Fragment Processing*

En esta etapa es en la que más recursos *hardware* se emplean, pues es la encargada de realizar todo el grueso de cómputo del renderizado de la escena, y es muy importante que sea productiva de cara a no provocar muchas detenciones en el *pipeline* gráfico. En este sentido, esta etapa suele estar formada por un *array* de unidades de procesamiento que se encargan de procesar los fragmentos que llegan de la etapa *Early Z-Test*, los denominados *Fragment Processors*. Como ya se ha mencionado en la Sección 2.1.8, estas unidades se encargan de invocar al *fragment shader* y ejecutarlo con el fin de obtener un color a la salida a partir de un fragmento (formado inicialmente por atributos). Para ejecutar tal programa de manera eficiente, es necesario hacer uso de unas cachés, tanto de datos como de instrucciones. Dado que los únicos datos a los que acceden estos procesadores son texturas[23], las cachés que guardan los datos son denominadas cachés de texturas (o *Texture Caches*). Los *Fragment Processors* implementan un conjunto de instrucciones específicas para procesar fragmentos, entre las que se encuentran las operaciones de texturas. Éstas acceden a la memoria de texturas y procesan los colores obtenidos mediante lo que se denomina unidades de texturas (o *Texture Units*). Las *Texture Units* generalmente hacen de interfaz entre las instrucciones de acceso a los datos y los datos en sí.

Para aumentar el rendimiento y el paralelismo SIMD, un *Fragment Processor*

[23] Esto no es del todo cierto puesto que un *Fragment Processor* puede ejecutar también código de propósito general, pues las GPUs también cuentan con su parte GPGPU (*General Purpose GPU*), la cual aprovecha el mismo *pipeline* gráfico para implementar el *pipeline* de cómputo de propósito general. Sin embargo, por motivos históricos y en este contexto, puede decirse que sólo acceden a texturas.

puede lanzar varios *warps* al mismo tiempo trabajando sobre el mismo conjunto de datos. Estos *warps* a su vez lanzan varios hilos (*threads*) para aprovechar la instrucción en vuelo de la *Instruction Unit* (de ahí el término SIMD, *Single Instruction Multiple Data*). Es importante que los *threads* de un mismo *warp* no diverjan durante la ejecución del código, pues de hacerlo, aquéllos que estén ejecutando una instrucción diferente tendrían que esperarse a que la *Instruction Unit* busque la siguiente instrucción, bloqueando a su vez a los *threads* que no habían divergido.

Dado que los *threads* se organizan en bloques formando los *warps*, si un *warp* se queda bloqueado durante una operación de memoria, un planificador de *warps* se encarga de apartarlo del flujo de ejecución para meter a otro *warp* que esté listo para ejecutarse. De este modo se ocultan latencias de acceso a memoria.

Etapa *Color Blending*

A la salida de los *Fragment Processors* obtenemos colores en un espacio acotado (o bien RGB o RGBA) que se tienen que escribir en la imagen final. Esta unidad es muy parecida a nivel *hardware* al *Early Z-Test*, pues realiza operaciones RMW (*Read, Modify and Write*), es decir, tiene que leer del *Color Buffer* un valor de color, aplicarle una función fija con el color entrante y escribirlo de vuelta al *Color Buffer*.

2.2.2. Arquitecturas *Tile-based Rendering* (TBR)

Como se ha visto anteriormente, una arquitectura IMR procesa la escena de entrada sobre la marcha, es decir, conforme le llegan objetos, éste los triangula y renderiza directamente (de ahí su nombre). A pesar de la simplicidad de este esquema, éste presenta una serie de problemas que lo convierten en una arquitectura prohibitiva para sistemas móviles, en los que la eficiencia energética y la autonomía de la batería determinan las decisiones de diseño. En primer lugar, sus necesidades de almacenamiento son considerablemente altas, por lo que las estructuras internas que usa no se pueden almacenar en memorias *on-chip* rápidas. Otro problema que presenta es el alto tráfico a memoria que genera debido a estas necesidades de almacenamiento.

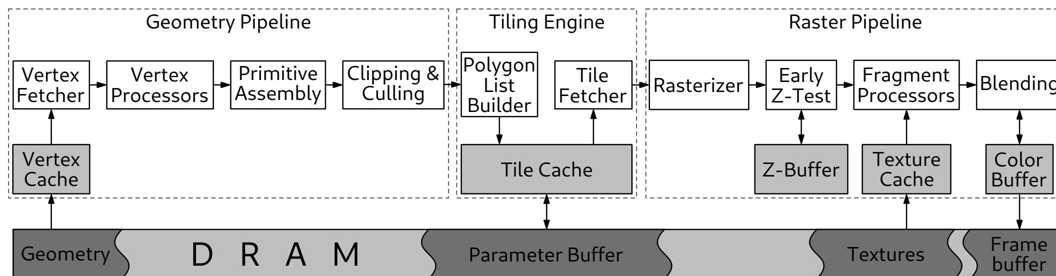


Figura 2.16: Implementación del *pipeline* gráfico de una arquitectura TBR.

La arquitectura *Tile-based Rendering* (TBR) trata de solucionar este problema de tráfico y necesidades de almacenamiento dividiendo la pantalla en pequeñas porciones más sencillas de renderizar. Así pues, esta arquitectura se ajusta mucho mejor a arquitecturas móviles dadas sus bajas necesidades de almacenamiento y su reducido tráfico a memoria que, a su vez, reducen el consumo energético. Una de las arquitecturas más antiguas con este tipo de *pipeline* es Pixel-planes 5 [31].

La Figura 2.16 muestra la implementación del *pipeline* de una arquitectura TBR. Como puede observarse, hasta la etapa de *Clipping & Culling*, esta arquitectura es exactamente igual que IMR. Sin embargo, a partir de esta etapa ambas arquitecturas difieren en cuanto a diseño.

Como puede observarse, TBR divide el renderizado de una escena en tres partes esenciales, a saber: *Geometry Pipeline*, *Tiling Engine* y *Raster Pipeline*. El *Geometry Pipeline*, como su nombre indica, es la fase en la que se procesa toda la geometría de la escena, y su funcionalidad se implementa igual que en IMR. El *Tiling Engine* es la parte que más caracteriza a TBR, pues es la que se encarga de dividir la pantalla en pequeñas porciones cuadradas del mismo tamaño denominadas *tiles*. El tamaño de estos *tiles* se determina durante el diseño arquitectónico de la GPU, y suele ser de 32x32 o 64x64 píxeles. Los triángulos de la etapa *Clipping & Culling* llegan al *Polygon List Builder*, la primera etapa del *Tiling Engine*. Esta etapa se encarga de hacer el particionado de la pantalla en *tiles* y almacenar la escena en el *Parameter Buffer*, una estructura de datos almacenada en memoria que, para acelerar su acceso, hace uso de una caché intermedia, la *Tile Cache*.

La Figura 2.17 muestra la estructura y organización interna del *Parameter Buf-*

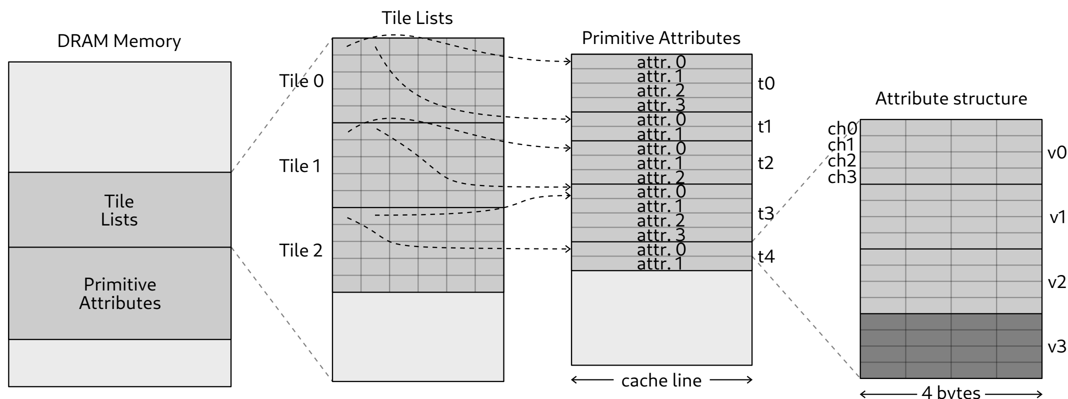


Figura 2.17: Estructura interna del Parameter Buffer de una GPU basada en TBR.

fer. Como puede apreciarse, por cada *tile* de la pantalla se mantiene una lista de triángulos que lo solapan. Esto quiere decir que cada vez que llega un triángulo al *Polygon List Builder*, éste calcula en qué *tiles* cae (total o parcialmente) y lo almacena en las listas correspondientes. Como se ha visto en la Sección 2.1, un triángulo tiene asociados unos atributos que lo definen, por lo que esta lista tiene que ser capaz de almacenarlos. Sin embargo, si almacenásemos todos los atributos de un triángulo en estas listas podría darse el caso (bastante probable) de que un triángulo solape más de un *tile*, por lo que los atributos estarían repetidos y las listas serían enormes. Para abordar este problema de atributos duplicados lo que se hace es definir otra región donde almacenarlos, y hacer referencia (definir punteros) a ellos desde las listas de triángulos. De este modo, tenemos dos regiones, una donde se almacenan los punteros de los triángulos que solapan en cada *tile*, y otra donde se almacenan los datos de los atributos de cada triángulo.

Una vez procesada toda la geometría de la escena y almacenada en el *Parameter Buffer*, se dispara el renderizado de la misma por parte del *Raster Pipeline*. Esta fase comienza también en el *Tiling Engine*, pues lo primero es obtener los *tiles* para procesarlos. De esto se encarga el *Tile Fetcher*. Esta etapa se encarga de hacer dos tareas esenciales. La primera es planificar el orden en el que se van a renderizar los *tiles*, es decir, en qué orden se van a procesar los *tiles* y a qué *Raster Pipeline* se van a mandar. La segunda es buscar la geometría de cada *tile* en forma de triángulos de cara a pasarlos al resto del *pipeline* y renderizarlos. Los triángulos de salida del *Tile*

Fetcher pasan al *Raster Pipeline*. Aquí es importante destacar que puede haber más de un *Raster Pipeline*, por lo que el *Tile Fetcher* podría mandar varios *tiles* a la vez, uno a cada *Raster Pipeline*, y no habría problemas al procesarlos en paralelo puesto que otra idea clave de una arquitectura TBR es que cada *tile* se procesa de manera *independiente*.

La primera etapa del *Raster Pipeline* es el *Rasterizer*, cuya implementación es similar a la que podríamos encontrar en IMR, con la salvedad de que en TBR sólo genera los fragmentos del triángulo que se encuentran estrictamente dentro del *tile*, es decir, si hay partes del triángulo que quedan fuera del *tile*, el *Rasterizer* no generará fragmentos en dichas áreas.

La siguiente etapa es el *Early Z-Test*, y su implementación también es similar a la de IMR. Aquí la diferencia está en que en vez de almacenar el Z-Buffer en memoria DRAM, éste se almacena en una pequeña memoria *on-chip* del tamaño del *tile*, es decir, por cada píxel del *tile* almacena un valor de profundidad. Aquí podemos apreciar una de las grandes virtudes de TBR frente a IMR. Mientras que en IMR se tiene que acceder a memoria DRAM por cada fragmento del cual se evalúa su profundidad, en TBR no se accede ni una vez puesto que toda la información de profundidad de un *tile* se puede almacenar en una memoria local *on-chip*.

La siguiente etapa del *Raster Pipeline* de TBR es el *Fragment Processing* y ésta, también se implementa igual que en IMR. Sin embargo, aquí también encontramos beneficios respecto a IMR debido a que el *working set* con el que trabaja (texturas en esencia) es mucho más reducido que en IMR, por lo que se mejora la localidad de las cachés de texturas y se reduce el tráfico a memoria en consecuencia.

La última etapa del *Raster Pipeline* de TBR es el *Color Blending*, y al igual que con otras etapas anteriormente vistas, la lógica que implementa es idéntica a la de una arquitectura IMR. Aquí, TBR vuelve a beneficiarse del reducido y acotado[24] *working set*, pues en vez de acceder directamente al *framebuffer* que está almacenado en memoria DRAM, accede a otra pequeña memoria *on-chip* denominada *Color Buffer*[25], reduciendo nuevamente el tráfico a memoria. Aquí, a diferencia de lo que

[24]Es importante notar también que el *working set* de TBR está *acotado*.
[25]No confundir con el *Color Buffer* de OpenGL, el cual contiene la información de color de todo el *framebuffer*

sucedía en la etapa *Early Z-Test*, la cual no necesita la memoria principal para nada, TBR necesita acceder al *framebuffer* final para que se visualicen los cambios en la pantalla del dispositivo (o en un *render target*). Por lo tanto, necesita acceder al menos una vez a memoria para volcar todos los cambios en el *framebuffer*. De este modo, cuando un *tile* termina de renderizarse por completo, es decir, cuando todos los fragmentos de todos los triángulos han pasado la etapa *Color Blending*, el contenido del *Color Buffer* (ya en estado definitivo) se vuelca en la porción correspondiente del *framebuffer*. A esta operación también se le denomina *color flushing*.

A pesar de que TBR reduce el tráfico a memoria respecto a IMR, también presenta una serie de desventajas. La primera, y quizá la más evidente, es que hay una fuerte barrera entre el *Geometry Pipeline* y el *Raster Pipeline*, pues para que el *Raster Pipeline* pueda trabajar, es necesario que previamente el *Geometry Pipeline* haya almacenado toda la escena en forma de *tiles* en el *Parameter Buffer*. Es por esto que si no se aprovechan todas las bondades de TBR, esto puede suponer una degradación considerable del rendimiento. Por otro lado, el hecho de tener que almacenar el *Parameter Buffer* en memoria hace que se produzca un tráfico a memoria que no se presentaba en IMR. A su vez, ciertas operaciones como las teselaciones (y demás operaciones que generen geometría nueva) se desaconsejan completamente en arquitecturas TBR, pues dado que se almacena toda la geometría resultante en memoria DRAM, se genera aún más tráfico. Sin embargo, este tipo de operaciones sí son apropiadas en IMR dado que esta nueva geometría no llega a almacenarse nunca en memoria, sino que se transfiere de etapa a etapa a través de colas *on-chip*.

2.3. Ejemplos de GPUs móviles comerciales

En esta Sección se verán dos de las arquitecturas de referencia que se han seguido a lo largo de esta Tesis Doctoral. Se trata de dos arquitecturas de GPU enfocadas a móviles que se han implementado en GPUs comerciales, a saber PowerVR (de Imagination Technologies) y Mali (de ARM).

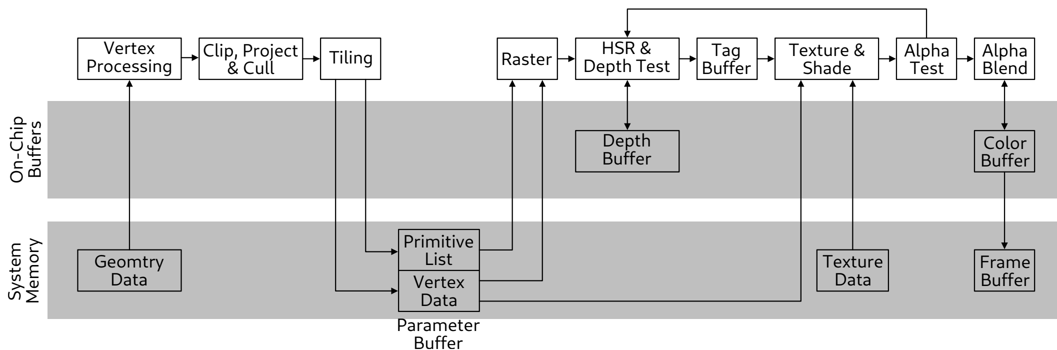


Figura 2.18: Esquema de la arquitectura TBDR de PowerVR.

2.3.1. PowerVR (arquitectura TBDR)

La familia de arquitecturas de PowerVR para GPUs de Imagination Technologies [70] van un paso más allá de TBR, e implementan un mecanismo de procesado previo de la escena denominado *deferred rendering* que permite eliminar por completo el *overdraw*. Como se ha visto en la Sección 1.1.1, el *overdraw* es un problema que sufren las GPUs cuando los objetos se renderizan de detrás hacia delante, lo cual resulta en un desperdicio de recursos. La idea que subyace detrás de *deferred rendering* es la de eliminar por completo el *overdraw* procesando la escena previamente para obtener los valores de profundidad del Z-Buffer final, pero sin renderizarla por completo (de ahí el término *deferred*). Esto permite, en una fase posterior, renderizar la escena con un Z-Buffer más maduro y capaz de eliminar por completo el *overdraw*. Este método de eliminación de trabajo de superficies que quedan ocultas también recibe el nombre de *Hidden Surface Removal* (HSR) [25], y hace referencia, no a una técnica en concreto, sino a una familia de técnicas que persiguen este objetivo. Dado que este esquema de *deferred rendering* se implementa sobre una arquitectura TBR (aunque no de manera exclusiva), generalmente también se le conoce como Tile-based Deferred Rendering (TBDR).

La Figura 2.18 muestra el diseño de la arquitectura TBDR implementada en las GPUs de PowerVR [80]. La primera etapa es la de *Vertex Processing* que, al igual que en la arquitectura TBR descrita en la Sección 2.2.1, se encarga de procesar la geometría entrante para convertirla de coordenadas de modelo a coordenadas NDC por medio de la ejecución de los conocidos *vertex shaders*. Sin embargo, tal y como

puede observarse, ésta también incluye la etapa de búsqueda de geometría desde memoria. La siguiente etapa, *Clip, Project & Cull*, como su propio nombre indica, se encarga de hacer el recorte de las primitivas para que encajen en el cubo unitario NDC, de convertir los puntos de NDC a coordenadas de pantalla y de hacer el descarte de las primitivas que correspondan.

Por su parte, la etapa *Tiling* se encarga de organizar la pantalla en *tiles* para almacenarlos en el *Parameter Buffer* que, como puede observarse, está compuesto por las listas de primitivas como punteros a atributos (estructura *Primitive List*) y por los atributos de las primitivas almacenadas (estructura *Vertex Data*). Esta etapa puede considerarse el *Polygon List Builder* de esta arquitectura TBDR.

Una vez almacenada toda la geometría de la escena en el *Parameter Buffer* en forma de *tiles*, el *Raster* se encarga de leer las primitivas de cada *tile* para rasterizarlas. Se puede apreciar que esta unidad funcional desempeña las funciones de *Tile Fetcher* y *Rasterizer* al mismo tiempo, tal y como se ha visto en la Sección 2.2.2, con la salvedad de que el *Rasterizer* no interpola los atributos, sino que sólo obtiene las coordenadas baricéntricas (la interpolación la llevará a cabo la etapa de *shading*). La etapa HSR & *Depth Test* también desempeña una doble función, a saber: el Z-Test y la actualización del *Tag Buffer* para eliminar fragmentos ocultos. El *Tag Buffer* es otra estructura del tamaño del *tile*, que por cada posición almacena un puntero de primitiva (de la estructura *Primitive List*) y las coordenadas baricéntricas del fragmento en cuestión. Así pues, cada fragmento que llega a esta etapa se evalúa para hacer el Z-Test, y si lo pasa, además de actualizar el Z-Buffer en consecuencia, también se actualiza el *Tag Buffer* con el puntero de la primitiva a la que pertenece el fragmento y sus coordenadas baricéntricas. De este modo, el *Tag Buffer* lleva un seguimiento de los fragmentos visibles hasta el momento y cómo recuperarlos de cara a renderizarlos más tarde.

Una vez que se han procesado todas las primitivas del *tile* por parte de la etapa *HSR & Depth Test*, entonces se comienza a renderizar la escena (se hace el *deferred rendering*). La etapa *Texture & Shade* recorre el *Tag Buffer* y por cada elemento (identificador de puntero y coordenadas baricéntricas) accede a los atributos de la primitiva correspondiente consultando la estructura *Vertex Data* por medio del

puntero almacenado y, con la ayuda de las coordenadas baricéntricas, interpola los atributos que a su vez, se pasan a los *fragment shaders*.

Los fragmentos resultantes pasan a la etapa *Alpha Test*, donde se pueden dar tres situaciones, a saber:

- Que el valor de *alpha* del fragmento sea 0, es decir, el fragmento es completamente transparente, en cuyo caso es descartado de inmediato dado que no va a tener ningún impacto sobre la imagen final.
- Si el fragmento ha sufrido un cambio de profundidad durante la ejecución del *fragment shader*, podría no ser visible, así que pasa de vuelta a la etapa *HSR & Depth Test* para evaluar de nuevo su profundidad y volver al ciclo inicial. Nótese que esto haría las veces de un Late Z-Test convencional.
- Si el fragmento no ha sufrido ninguna modificación en su profundidad y, además, su valor de *alpha* es mayor que cero entonces el fragmento es visible y pasa a la siguiente etapa.

Finalmente, la etapa *Alpha Blend* se encarga de mezclar los colores del *Color Buffer on-chip* con los colores entrantes, de manera análoga a cómo lo hace la etapa *Color Blending* explicada en la Sección 2.2.1.

Una vez renderizado el *tile* por completo, se escribe en el *framebuffer* global, que está almacenado en memoria DRAM.

2.3.2. Mali

La serie de procesadores de Mali es una familia de GPUs diseñada por ARM. Dentro de esta serie, la arquitectura más reciente es Valhall, que comparte gran parte de su diseño con su predecesora, la arquitectura Bifrost.

La Figura 2.19 muestra una vista general de la arquitectura Valhall de Mali. Respecto al número de *Shader Cores*, es un parámetro que decide el fabricante del SoC que integra esta GPU, y éste determina una gran parte del área ocupada. Como se puede observar en la Figura, esta arquitectura usa *Shader Cores* unificados, lo que

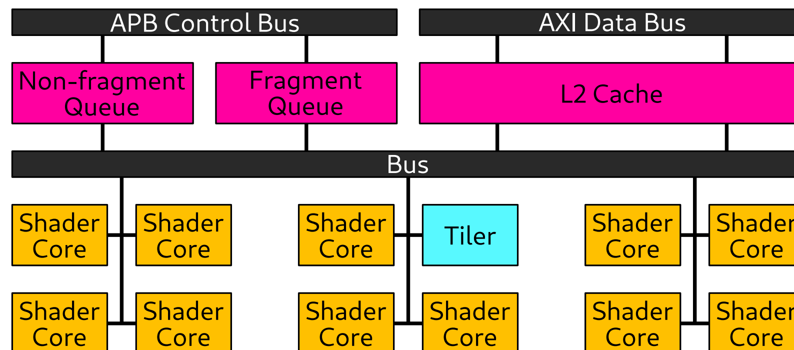


Figura 2.19: Esquema general de la arquitectura Valhall de Mali.

significa que se pueden usar tanto para procesar geometría, como fragmentos o cargas de cómputo de propósito general (como por ejemplo, *kernels* de OpenCL). Este tipo de arquitectura permite aprovechar al máximo los *Shader Cores*. A la hora de planificar y gestionar las cargas de trabajo que van a procesar los *Shader Cores*, éstas se dividen en dos colas separadas, a saber: *Fragment Queue*, *Non-fragment Queue*. Como sus propios nombres indican, la primera se encarga de almacenar y redirigir las cargas de trabajo relativas al procesamiento de fragmentos, mientras que la otra se encarga del resto de cargas de trabajo como, por ejemplo, la geometría. Este diseño permite el procesamiento de diferentes tipos de cargas de trabajo en paralelo, siempre y cuando sean de fotogramas independientes[26]. Así, por ejemplo, es posible procesar el renderizado de un fotograma mientras se procesa la geometría del siguiente[27]. Tanto los *Shader Cores* como el *Tiler* están conectados mediante un bus interno que los interconecta con la caché L2 y las colas anteriormente mencionadas. Cada *Shader Core* puede escribir en el bus hasta dos píxeles de 32 bits (4 canales) por ciclo, por lo que si, por ejemplo, tenemos una arquitectura con 4 *Shader Cores*, este bus podría alcanzar un ancho de banda de hasta 256 *bits* por ciclo, tanto para lecturas como para escrituras, sin que eso suponga un cuello de botella para el rendimiento de la arquitectura. Por su parte, la caché L2 está conectada al resto de la

[26]Como se ha visto en la Sección 2.2.2, existe una barreara entre el *Geometry Pipeline* y el *Raster Pipeline*, por lo que sólo podrían ejecutarse en paralelo si ambos procesaran datos de fotogramas diferentes.

[27]Nótese que en tal caso sería necesario mantener en memoria dos *Parameter Buffers*, pues durante la fase de geometría de un fotograma se necesita escribir en el *Parameter Buffer* la escena de dicho fotograma, mientras que durante la fase de renderizado del otro fotograma se necesita leer otra escena diferente de otro *Parameter Buffer*.

jerarquía de memoria mediante un bus de datos AXI, El tamaño de la L2 también lo decide el fabricante del SoC, y suele estar entre los 64 KiB y los 128 KiB por *Shader Core*. Este parámetro también determina en gran medida el área de silicio disponible para fabricar el SoC, por lo que es importante que los fabricantes de SoCs equilibren este parámetro de manera acorde al resto de unidades de la arquitectura. Además, los fabricantes de SoCs pueden elegir tanto el ancho de banda como el número de puertos de la L2 con la memoria externa, nuevamente, teniendo en cuenta el coste en área que supone aumentar cada uno de estos parámetros.

El *Shader Core* de Valhall

La Figura 2.20 muestra la arquitectura interna de un *Shader Core* de la arquitectura Valhall.

A simple vista, se puede observar que el *pipeline* de renderizado está desacoplado del *pipeline* del resto de cargas de trabajo (denominadas *Non-fragment*), es decir, corren en paralelo. Como se puede observar, el *pipeline* de renderizado empieza igual que en el resto de arquitecturas, pues empieza con la búsqueda de geometría (*Tile List Reader*), continúa con el rasterizado de las primitivas, le sigue una etapa de *Early-Z Test* y finalmente se generan los *threads* que van a procesar los fragmentos resultantes. Respecto al *pipeline* del resto de cargas de trabajo, simplemente se asume que los datos llegan en crudo de alguna zona de la memoria gráfica. Éstos pueden ser vértices, datos de cómputo (para ejecutar *kernels* de OpenCL, por ejemplo) o simplemente atributos definidos por el usuario.

Los *threads* resultantes de cada *pipeline* entran en lo que se denomina el *Execution Core*. Esta unidad se encarga de desmenuzar los *threads* en *warps* para mejorar la eficiencia en la ejecución, evitando bloqueos por altas latencias a memoria o por contención de recursos. Para llevar a cabo la ejecución de estos *threads*, el *Execution Core* cuenta con dos unidades de procesamiento (*Processing Unit* 0 y 1) que se encargan de repartir el trabajo entre las diferentes unidades funcionales en función del tipo de carga de trabajo. Para ello, hace uso de un bus de datos (*Message Fabric*). Respecto a las unidades funcionales disponibles, el *Execution Core* cuenta con una unidad de carga y almacenamiento (*Load/Store Unit*) para gestionar los acce-

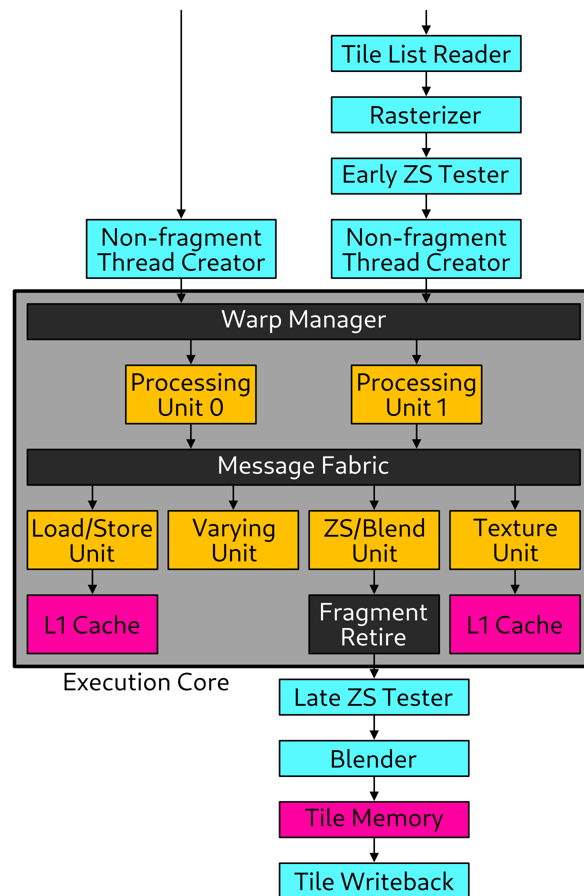


Figura 2.20: Diagrama de un *Shader Core* de la arquitectura Valhall de Mali.

sos a memoria tanto de escritura como de lectura; una *Varying Unit* que se encarga de interpolar atributos; una unidad de ejecución de *shader* (*ZS/Blend Unit*), que se encarga de manejar todos los accesos al *tile buffer* (el *Color Buffer* en esencia) tanto intrínsecos de OpenGL ES, como programáticos definidos explícitamente en el código del *shader*; y una unidad de texturas (*Texture Unit*), necesaria para acceder a las texturas de memoria por medio de funciones de filtrado. Tanto la *Load/Store Unit* como la *Texture Unit* están conectadas a la jerarquía de memoria por medio de la caché L1.

Si la carga de trabajo que se está ejecutando es de tipo fragmento, entonces a la salida del *ZS/Blend Unit* se retiran los fragmentos ya generados (*Fragment Retire*), que salen del *Execution Core* para pasar por la etapa *Late Z-Test*, luego por el *blending*, después al *Color Buffer* y finalmente al *framebuffer* en memoria (Tile Writeback).

***Index-Driven Vertex Shading* (IDVS)**

Una de las novedades que introduce Mali desde su arquitectura Bifrost, y que se desmarca del resto, es la forma en la que procesa la geometría. En este aspecto, las nuevas arquitecturas de Mali, como la Valhall, implementan lo que la marca denomina *Index-Driven Vertex Shading* (IDVS) que, como su nombre indica, el procesado de la geometría está dirigido por los índices de las primitivas que definen el objeto, y no por los vértices como se hace convencionalmente. Esto significa que el *Geometry Pipeline* empieza en la etapa de *Primitive Assembly*, que es la primera etapa donde se tienen en cuenta los índices del EBO del objeto que se quiere renderizar; y no en la etapa de búsqueda de vértices como hace el resto de arquitecturas. Por otro lado, IDVS introduce otro concepto novedoso, que es separar la ejecución del *vertex shader* en dos partes, a saber: el procesamiento exclusivo de posiciones (*Position Shading*), y el procesamiento del resto de atributos que no son posición (*Vertex non-position*). La motivación detrás de este concepto es la misma que la que usa PowerVR en *deferred rendering*, pero aplicado al procesamiento de la geometría. De este modo, si hay geometría oculta que por su posición se detecta que no será visible, no se procesa, aliviando la carga de trabajo del *shader core*, reduciendo así el consumo energético.

Como se ha mencionado anteriormente, en IDVS el procesamiento de la geometría comienza en la etapa de *Primitive Assembly*, la cual empieza leyendo índices en paquetes de primitivas. Es decir, si tenemos por ejemplo el tipo de primitiva `GL_TRIANGLES`, entonces leerá tres índices en cada iteración. Estos índices se pasan al *Position Shading*, el cual sólo ejecuta el *vertex shader* para las posiciones (vertex positions). Esto se hace así para efectuar el *clipping* y el *culling* de las primitivas antes de procesar el resto de atributos[28]. Los paquetes de índices que pasan el test (las primitivas en esencia) se pasan al Varying Shading, el cual ejecuta el mismo *vertex shader*, pero esta vez lo hace para el resto de atributos que no se habían procesado antes, es decir, aquellos que no son posiciones (*vertex non-positions*). El resultado de esta ejecución mixta son los atributos de los vértices de las primitivas ya transformados.

Para poder hacer esta ejecución de manera eficiente, es necesario que las posiciones de los vértices y sus atributos correspondientes estén almacenados de manera separada, de modo que se pueda acceder rápidamente a cada conjunto y recorrerlo sin necesidad de dar saltos en la memoria del *Parameter Buffer*. Para lograrlo, es necesario cambiar la estructura del *Parameter Buffer*, dividiendo la región de atributos de vértice en dos partes, a saber: posiciones y resto de atributos.

Aunque *a priori* esto pueda tener ventajas como las ya mencionadas, puede presentar sus dificultades a la hora de implementarla, como por ejemplo, qué pasa con los vértices compartidos en varias primitivas. Según este esquema, como la geometría está dirigida por los índices, cada vez que se procese se tendrá que buscar en memoria, aunque ya se haya buscado anteriormente. Aquí se presume que se usa una caché muy pequeña para almacenar los últimos n vértices de la última primitiva que se ha procesado.

[28]Nótese la fuerte similitud que tiene con *deferred rendering*. El concepto es el mismo, primero procesa las posiciones para determinar la visibilidad de las primitivas de manera muy temprana y, luego, procesa sólo las primitivas que finalmente son visibles.

3

Metodología de trabajo

3.1. Herramientas y entorno de simulación

Para realizar esta Tesis Doctoral se ha usado TEAPOT [8], un potente *framework* de simulación de GPUs móviles modernas que permite, entre muchas otras cosas, simular arquitecturas IMR, TBR y TBDR.

La Figura 3.1 muestra la arquitectura TBDR que se ha usado en los trabajos realizados durante esta Tesis, y que es con la que se ha modelado el simulador ciclo a ciclo.

Para modelar el consumo energético y la potencia de las unidades funcionales que incluye la arquitectura de GPU modelada, se ha usado McPAT [50], una herramienta bastante conocida y usada en el campo de la arquitectura de computadores. Respecto al subsistema de jerarquías de memoria, éste se ha modelado usando

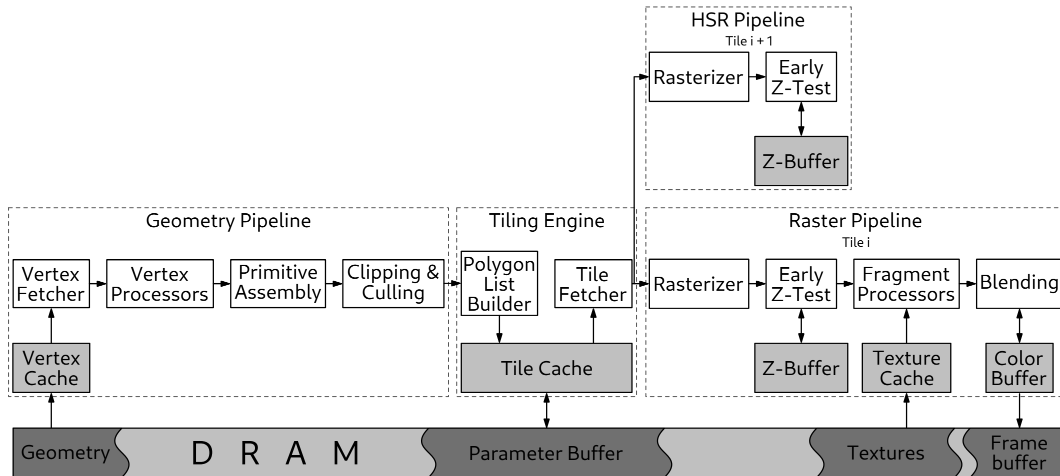


Figura 3.1: Implementación del *pipeline* gráfico de una arquitectura TBDR usado en TEAPOT.

DRAMsim2 [73], el cual permite simular con una precisión muy alta el *timing* de diferentes tipos de memorias DRAM. A su vez, permite configurar el subsistema de memoria para modelarlo acorde a las necesidades de las arquitecturas que queremos evaluar.

Respecto a los datos con los que se alimenta el simulador, se trata de trazas obtenidas mediante GAPID [34], una herramienta de depuración de gráficos para dispositivos Android desarrollada por Google[33] que permite guardar la traza de ejecución de una aplicación (APK) para luego reproducirla tanto en un dispositivo Android como en un ordenador de sobremesa. Dado que el simulador necesita una traza compatible, es decir, una traza de GAPID directamente no sirve, es necesaria una conversión. Para ello se usa el *softpipe renderer* de Gallium [30], un *driver* gráfico proveniente del proyecto Mesa 3D (de código abierto) que permite ejecutar el *pipeline* gráfico mediante *software*, es decir, sin usar una GPU física.

La Figura 3.2 muestra el flujo de ejecución del simulador, que se explica a continuación. Todo comienza con un videojuego para Android (descargado de la Play Store [35]). Éste se juega sobre el depurador GAPID para obtener la traza. Una vez obtenida la traza, ésta se reproduce con `gapir`, una herramienta de GAPID para tal fin. Como el simulador ciclo a ciclo se alimenta de otros datos, `gapir` se ejecuta sobre el *softpipe* de Gallium. El motivo de que sea un renderizador *software* es que

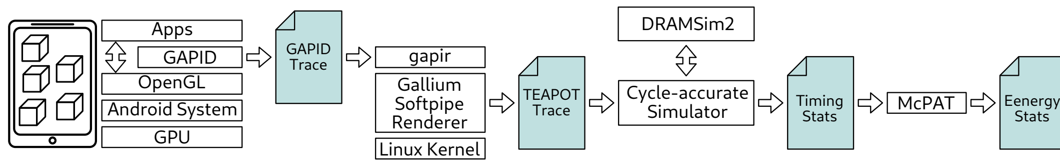


Figura 3.2: Flujo de ejecución del entorno de simulación TEAPOT.

se puede instrumentar fácilmente, y obtener los datos intermedios del proceso de renderizado (vértices, colores, primitivas, fragmentos...). Esta versión de Gallium instrumentada genera un fichero legible por el simulador ciclo a ciclo. Lo siguiente es ejecutar este simulador. Al finalizar la ejecución de éste, se genera un fichero con los datos de interés (contadores de acceso, cargas de trabajo por unidad funcional, *throughput*...). El modelo de McPAT se alimenta de estos contadores para reportar el consumo energético final, desglosado por unidades funcionales.

Un problema que tiene GAPID a la hora de generar trazas, como cualquier otro depurador, es la degradación en el rendimiento que se produce por la introducción de sobrecargas derivadas de este proceso de depuración (barreras, cerrojos, funciones trampolín para generar los datos de depuración...). Esto provoca que la aplicación vaya más lenta. La gran mayoría de videojuegos, cuando detectan que el rendimiento del dispositivo baja, éstos hacen lo que se denomina *frame skipping* para no perjudicar *mucho* la experiencia de usuario. Como su propio nombre indica, está técnica consiste en saltarse fotogramas de renderizado con el fin de que el motor físico del juego avance a un ritmo *real*, creando así la sensación de que todo avanza en tiempo real. Esto tiene como consecuencia directa que las trazas obtenidas tienen una coherencia de *frame* muy pobre, haciendo que técnicas como Triangle Dropping u Ω-Test se vean severamente perjudicadas. Para solucionar este problema, lo que se ha hecho ha sido instrumentar también GAPID, de modo que *engaña* al reloj interno de Android haciéndole creer que ha pasado menos tiempo del que realmente ha pasado. Esto se consigue interceptando las llamadas `gettimeofday` (y sus derivadas) de Linux. De este modo, el motor gráfico se piensa que ha pasado tan poco tiempo que considera que no es necesario hacer *frame skipping*, por lo que la coherencia entre fotogramas vuelve a la normalidad.

La Tabla 3.1 muestra los parámetros de simulación generales usados a la hora

Tabla 3.1: Parámetros generales de simulación usados en el simulador ciclo a ciclo.

Baseline GPU Parameters	
Frequency	600 MHz
Voltage	1.0 V
Technology node	22 nm
Screen resolution	2160x1080
Tile size	16x16 pixels
Main Memory	
Frequency	400 MHz
Voltage	1.5 V
Technology node	32 nm
Latency	50-100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Line Size	64 bytes
Size	1 GiB, 8 banks
Queues	
Vertex (Input & Output)	16 entries, 136 bytes/entry
Triangle & Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Color	64 entries, 24 bytes/entry
Caches	
Block size	64 bytes (all caches)
Vertex Caches (× # VPs)	4 KiB, 1-cycle lat, 2-way assoc
Texture Caches (× # FPs)	16 KiB, 2-cycle lat, 4-way assoc
Tile Cache	32 KiB, 2-cycle lat, 2-way assoc
L2 Cache	256 KiB, 8 banks, 18-cycle lat, 8-way assoc
Color Buffer	1 KiB, 1 bank, 1-cycle latency
Depth Buffer	1 KiB, 1 bank, 1-cycle latency
Non-programmable stages	
Primitive assembly	1 vertex/cycle
Rasterizer	1 attribute/cycle
Early Z-Test	8 in-flight quad-fragments
Programmable stages	
Vertex processing stage	4 Vertex Processors (VPs)
Fragment Processing stage	4 Fragment Processors (FPs)

Tabla 3.2: Caracterización de los *benchmarks* evaluados.

Benchmark	Alias	Description	Downloads (Mill.)	Vertex shader instr. (Mill.)	Fragment shader instr. (Mill.)	Execution Time (Mill. cycles)
Beach Buggy Racing	bbr	Racing	100-500	96	2052	749
Derby Destruction Simulator	dds	Racing & Battle royale	10-50	165	4993	1140
Gravity	gra	Action	1-5	74	355	144
Hellrider	hrd	Racing	1-5	112	3534	868
Hot Wheels	hwl	Racing	50-100	431	2073	950
Maze 3D	maz	Labyrinth	10-50	131	4420	1112
Sniper 3D	s3d	Shooter	100-500	144	1600	684
Sonic Dash	snd	Adventure arcade	100-500	87	4154	1219
Counter Strike	cou	Shooter	10-50	-	-	-
300	300	Hack & slash	10-50	-	-	-
Captain America	cam	Beat'em up	1-5	-	-	-
Vegas Crime Simulator	vcs	Sandbox & Crime	100-500	-	-	-
Temple Run	tru	Adventure arcade	100-500	-	-	-

de evaluar cada una de las técnicas propuestas en esta Tesis, salvo que se indique lo contrario para algún parámetro concreto en alguna técnica.

3.2. Benchmarks

Para evaluar las técnicas propuestas en esta Tesis Doctoral, aparte del simulador, hace falta baterías de prueba (*benchmarks*) que expriman al máximo o, en la medida de lo posible, las características de la GPU. En esta sección se muestran los *benchmarks* evaluados a lo largo de este trabajo.

Los *benchmarks* que se han evaluado están caracterizados por su complejidad gráfica en las escenas renderizadas, teniendo en cuenta así, su número de vértices, comandos, texturas, resolución de éstas, complejidad de los *shaders*, *overdraw*, geometría ocluida, si son tridimensionales, etc. También se tiene en cuenta el tipo de juego que es, es decir, si es de carreras, un *shooter* o un *arcade*, por ejemplo. Este aspecto es importante a la hora de conocer el comportamiento de los objetos de cara a realizar optimizaciones en el *pipeline* gráfico. Así pues, por ejemplo, se puede determinar que en los juegos de *shooter* es más común ver movimientos laterales que en un juego de carreras, en el cual la cámara se mueve continuamente hacia delante. La Tabla 3.2 muestra esta caracterización.

Los fotogramas que se muestran a continuación se han obtenido de las trazas de las secuencias de interés de los videojuegos evaluados, por lo que son orientativas.

Figura 3.3: Fotograma del videojuego Beach Buggy Racing.

La Figura 3.3 muestra un fotograma del videojuego de carreras Beach Buggy Racing. Se puede observar que este videojuego es de carreras y en tres dimensiones, por lo que los movimientos son generalmente hacia delante. El objetivo es, como en casi todos los videojuegos de carreras, acabar el primero. Para hacer más entretenida la carrera, y así no limitarse a conducir, durante el recorrido van apareciendo diferentes objetos que el jugador puede coger y usarlos a su favor o en contra del resto de participantes, como por ejemplo, los turbos para correr más durante un intervalo de tiempo. En lo que a gráficos se refiere, este videojuego presenta uno de los sets de texturas con más resolución que se han evaluado. Del mismo modo, la complejidad en cuanto a su geometría es bastante destacable, dada la complejidad de los modelos 3D que usa. Sin embargo, a pesar de toda esta complejidad, es uno de los videojuegos mejor optimizados en cuanto a *overdraw*, por lo que el rendimiento en dispositivos de bajas prestaciones es excepcional. En este aspecto, cabe destacar que, si bien es cierto no es el videojuego con menos *overdraw* de la suit de *benchmarks*, éste presenta un ratio de *overdraw* es muy bajo respecto al resto de juegos con las mismas características gráficas.

La Figura 3.4 muestra un fotograma del videojuego Derby Destruction. *A priori* parece un juego de carreras, pero no lo es realmente, pues la pista es circular

Figura 3.4: Fotograma del videojuego Derby Destruction.

y cerrada. La idea de este videojuego es destruir al resto de contrincantes colisionándoles y ser el último superviviente de la pista. Los movimientos son bastante bruscos, sobre todo cuando el resto de coches choca con el jugador. Respecto a sus texturas, a pesar de lo que pueda parecer por la imagen, tienen bastante resolución. En cuanto a la complejidad de sus modelos, es cierto que los escenarios son algo simples, sin embargo, los modelos de los vehículos tienen bastante detalle, sobre todo a medida que se avanza en el juego.

La Figura 3.5 muestra un fotograma del videojuego Gravity. Se trata de un juego en tres dimensiones en tercera persona. El jugador maneja a un astronauta flotando en el espacio, y tiene que seguir las instrucciones que le aparecen en la pantalla a lo largo de las misiones. A simple vista, los gráficos parecen simples en tanto que todo el fondo está vacío, es decir, no se renderiza nada. Sin embargo, los modelos tienen un nivel de detalle bastante decente, y sus texturas son de una resolución bastante alta para lo que se puede apreciar.

La Figura 3.6 muestra un fotograma del videojuego Hell Rider. El videojuego trata de perseguir en una moto a un sujeto que ha secuestrado a alguien. A lo largo del camino aparecen obstáculos que el jugador debe esquivar para que el enemigo no se escape. En este caso, se puede ver que la escena es muy simple, así como

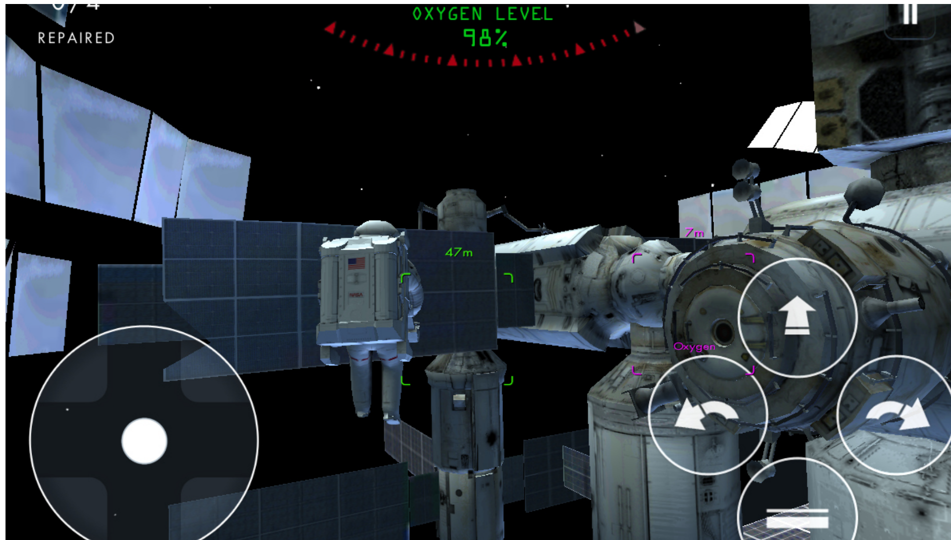


Figura 3.5: Fotograma del videojuego Gravity.

sus modelos y texturas. Sin embargo, los movimientos son bastante rápidos, lo que mantiene la atención del jugador, haciéndolo entretenido.

La Figura 3.7 muestra un fotograma del conocido videojuego Hot Wheels. Se trata de un juego de carreras, aunque no al uso, pues éste se visualiza de manera lateral. A este tipo de videojuegos se les denomina 2.5D, porque los gráficos que tienen son en tres dimensiones, pero la forma en la que se coloca la cámara hace que la componente de profundidad se diluya, haciendo parecer que es en dos dimensiones. En este videojuego, el objetivo es llegar a la meta antes que el contrincante y, además, ganar puntos haciendo acrobacias de por medio. Cada jugador conduce en su propia pista, por lo que no pueden colisionar entre sí. Las pistas también presentan una forma poco usual en comparación con otros videojuegos de carreras, pues son pistas muy finas en las que sólo entra un coche, son casi flotantes, no existen curvas y presentan loopings, tirabuzones y rampas con saltos muy pronunciados. Respecto a su complejidad gráfica, las pistas son simples, mientras que los coches tienen un nivel de detalle bastante elevado. El fondo son unos edificios que, a pesar de ser simples, son muchos y gastan muchos recursos. Una peculiaridad de este videojuego es que los edificios se renderizan siempre, suponiendo un malgasto de recursos, puesto que la mayoría de ellos no están dentro del plano de la cámara.

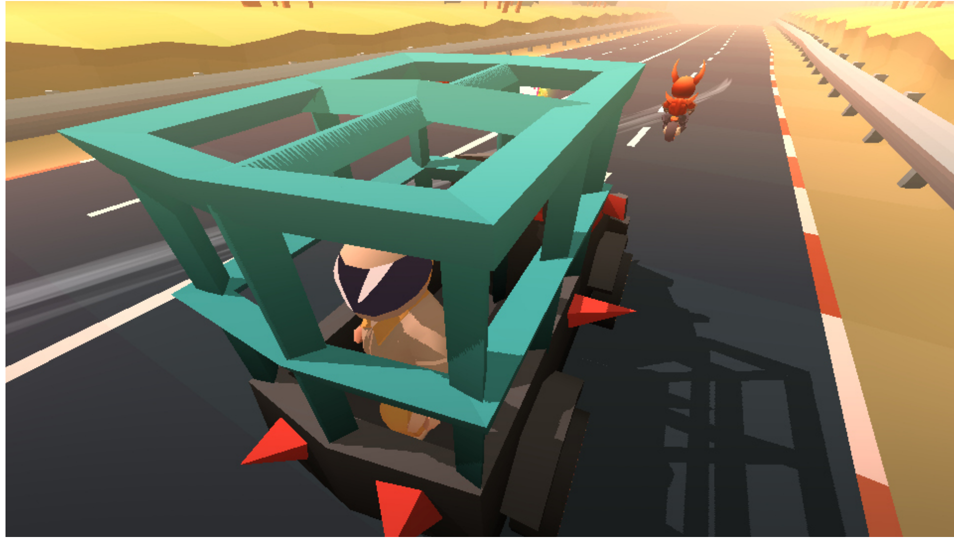


Figura 3.6: Fotograma del videojuego Hell Rider.

Figura 3.7: Fotograma del videojuego Hot Wheels.

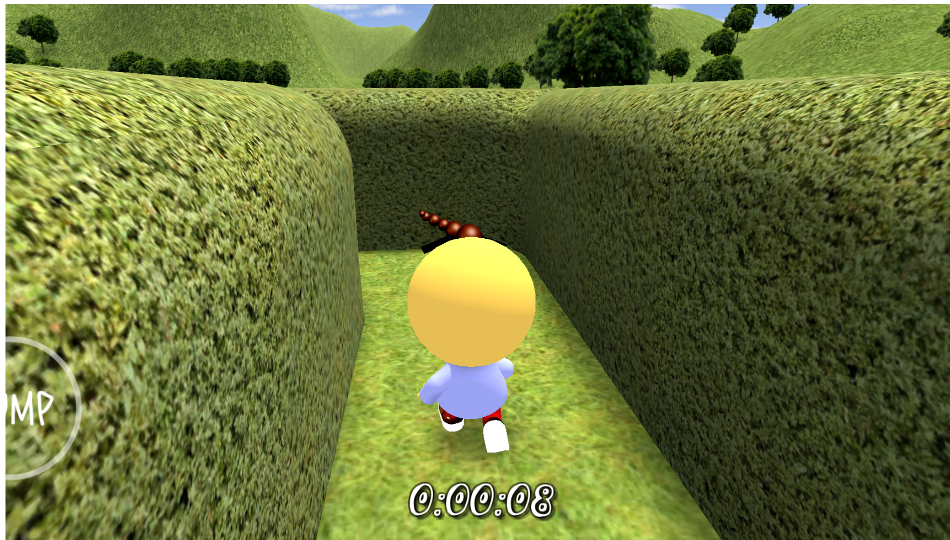


Figura 3.8: Fotograma del videojuego Hot Wheels.

La Figura 3.8 muestra un fotograma del videojuego Maze 3D. Se trata de un muñeco que entra en un laberinto para llegar a la meta. Es el típico videojuego de laberinto, pero en tres dimensiones, lo que lo hace especialmente complicado dado que no se puede ver el mapa directamente. Sin embargo, el jugador puede ayudarse de pequeños saltos para observar más allá de los setos. Respecto a sus gráficos, el número de texturas es muy bajo, pero las pocas que hay son de gran resolución. Los modelos son muy simples, sin embargo, el muñeco se dibuja de manera muy ineficiente, pues cada forma geométrica es un comando por separado y, además, tienen un número muy alto de caras, lo cual es innecesario, puesto que el modelo que se trata de representar es simple de base.

La Figura 3.9 muestra un fotograma del videojuego Snipper 3D. Se trata de un francotirador que tiene que abatir a sus enemigos desde las azoteas de las ciudades. Aunque no sea un *shooter* como tal, puesto que en cada partida, el jugador permanece siempre estático en la azotea, se puede considerar como tal. La complicación de este videojuego es que el jugador tiene que abatir al enemigo sin fallar, puesto que podría ser visto y fracasar la misión. Además, alrededor del enemigo va apareciendo gente inocente que el jugador tiene que tratar de no disparar por razones obvias. Para aumentar más la dificultad, cuando el jugador apunta para enfocar

Figura 3.9: Fotograma del videojuego Snipper 3D.

al enemigo, el pulso le va temblando cada vez más, lo que hace más impreciso el disparo. En lo que a gráficos respecta, las texturas tienen una resolución moderada. Los modelos que dibuja tienen también un nivel medio de detalle. Los movimientos son muy lentos, salvo cuando se hace un disparo acertado, entonces la cámara va detrás de la bala hasta impactar con el enemigo.

La Figura 3.10 muestra un fotograma del videojuego Sonic Dash. En este videojuego, el jugador maneja al clásico puercoespín azul, Sonic. Éste va avanzando continuamente sin parar, y tiene que ir esquivando obstáculos hasta llegar a la meta. Para esquivar estos obstáculos, el jugador puede saltar o moverse lateralmente. Opcionalmente, puede derribar a sus enemigos si, al saltar, cae justo encima sus cabezas, aunque esto depende del enemigo. Para conseguir puntos, aparte de abatir enemigos, el jugador tiene que ir recolectando los anillos dorados que aparecen a lo largo de la pista. Adicionalmente, para llegar más rápido a la meta y para saltar los precipicios, eventualmente van apareciendo trozos de suelo que aceleran al personaje. Los movimientos no son muy bruscos, salvo cuando se pisa uno de estos trozos de suelo. Respecto a los gráficos, presenta una complejidad alta para lo que parece en la imagen, pues las texturas tienen una resolución alta y los modelos son variados y con muchos detalles.

Figura 3.10: Fotograma del videojuego Sonic Dash.

Figura 3.11: Fotograma del videojuego Counter Strike.

La Figura 3.11 muestra un fotograma del videojuego Counter Strike. Se trata de un *shooter* en el que el objetivo es abatir a los enemigos a medida que van apareciendo por el camino. A lo largo del juego el jugador va obteniendo nuevas armas y desbloqueando nuevos niveles. Respecto a la calidad de los gráficos, tanto la geometría como la resolución de las texturas son bastante pobres. Sin embargo, el videojuego es bastante fluido, con mucho movimiento y acción.

La Figura 3.12 muestra un fotograma del videojuego 300. Se trata de un videojuego de acción que trata sobre la mítica batalla de las Termópilas y los trescientos soldados del ejército espartano dirigidos por el rey Leónidas. El jugador lleva a un soldado espartano, y su objetivo es abatir soldados persas hasta llegar a un punto concreto del nivel para pasar al siguiente. Si bien es cierto que los modelos usados tienen una geometría muy simple, y las texturas una resolución baja, la cámara (generalmente) apunta desde lejos a la melé de soldados que hay en toda la escena, haciendo que se muestre todo el escenario y, por lo tanto, dando lugar a una escena con una calidad de gráficos decente.

La Figura 3.13 muestra un fotograma del videojuego Captain America. Es un juego de lucha en la que el jugador lleva al conocido personaje Capitán América.

Figura 3.12: Fotograma del videojuego 300.

Se trata de otro videojuego 2.5D, pues la escena es tridimensional, pero la cámara se sitúa lateralmente. El objetivo es llegar al final del nivel saltando los obstáculos y abatiendo a los enemigos que aparecen por el camino. Los modelos usados tienen una calidad moderada, sin llegar a tener muchos detalles. Sin embargo, las texturas usadas presentan una calidad baja.

La Figura 3.14 muestra un fotograma del videojuego Vegas Crime Simulator. Se trata de un juego de mundo abierto, lo que significa que el jugador puede moverse libremente por el mapa. El objetivo de este videojuego es completar las misiones criminales que se van ordenando. Todo esto tiene lugar en Las Vegas. La calidad general de las escenas es bastante pobre, pero es entendible dado que se trata de un mundo abierto en el que se tienen que renderizar muchos objetos, como coches, edificios gigantescos, gente de todo tipo, etc.

La Figura 3.15 muestra un fotograma del videojuego Temple Run. Se trata de un *arcade* en el que el jugador va huyendo de unas criaturas extrañas con forma de mono demoníaco y cabeza de calavera. Al igual que en Sonic Dash, el jugador no para de correr y tiene que esquivar los obstáculos que van apareciendo a lo largo del camino. Cuanto más tiempo aguante el jugador sin que le pillen los monos demoníacos, más puntos gana. Respecto a la calidad de los gráficos, tanto los modelos

Figura 3.13: Fotograma del videojuego Captain America.

Figura 3.14: Fotograma del videojuego Vegas Crime Simulator.

Figura 3.15: Fotograma del videojuego Temple Run.

como las texturas presentan una calidad media. Los movimientos, por su parte, son bastante rápidos, pues hay bastantes giros y saltos.

4

Ω-Test

Como se ha visto en la introducción (Sección 1.1.1), el *overdraw* en las escenas de los videojuegos no es despreciable (véase Figure 1.2).

Para dar una idea más aproximada del problema que supone el *overdraw*, la Figura 4.1 muestra el detalle de los fragmentos que se procesan en una escena. Se puede observar que de media el 37.7 % de los fragmentos están ocluidos, llegando incluso a alcanzar el 56.5 % de *overdraw* (en el caso de `gra`).

La existencia de *overdraw* en las escenas se traduce en un desperdicio en el uso de los recursos, puesto que los fragmentos que no van a ser finalmente visibles en la pantalla podrían evitar todo su procesado sin que ello tenga ningún impacto en la imagen final. Si este problema no se aborda de una forma efectiva, se traducirá en una degradación de la eficiencia energética que penalizará considerablemente tanto la duración de la batería como su vida útil, dando lugar a una mala experiencia de usuario. Por tanto, es importante reducir la cantidad de *overdraw* al máximo. En

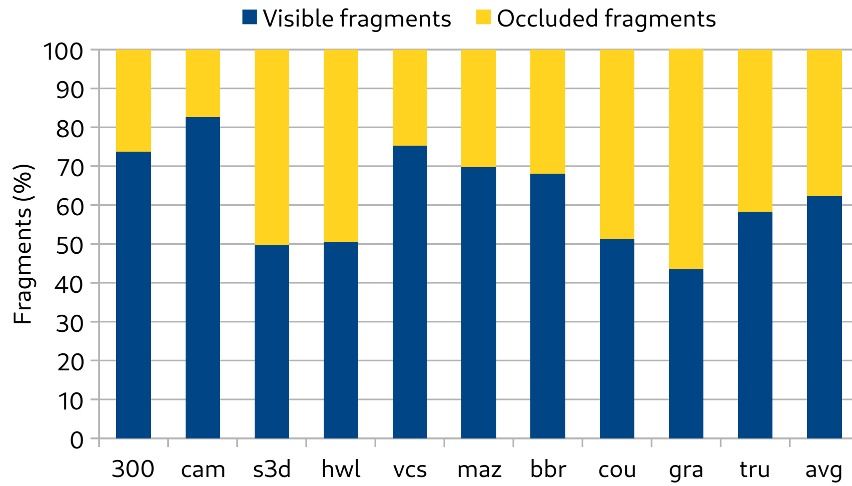


Figura 4.1: Desglose de fragmentos sombreados en una escena, clasificados en visibles y ocluidos.

este aspecto, el papel de la etapa Early Z-Test es clave, pues se trata de la etapa encargada de determinar la visibilidad de los fragmentos del fotograma.

En este Capítulo se presenta una propuesta novedosa, denominada Ω-Test, que modifica el comportamiento de etapa encargada de determinar la visibilidad esta etapa añadiendo un test adicional. Este segundo test, en lugar de usar el Z-Buffer tradicional, se apoya en una estructura denominada Ω-Buffer que almacena las profundidades del Z-Buffer del mismo *tile* dentro del fotograma pero referido al fotograma anterior. Este test actúa de manera análoga a cómo lo hace el Z-Test. Cuando un fragmento también pasa el Ω-Test, puede proceder a la etapa *Fragment Processing*, donde se realizará todo el proceso de sombreado e iluminación. De lo contrario, el fragmento se descarta puesto que se asume que también estará oculto en el fotograma actual, en tanto que ya lo estuvo en el fotograma anterior teniendo en cuenta el contenido del Ω-Buffer, que no es más que una versión "reducida" del Z-Buffer del fotograma anterior. Es importante notar que, a diferencia del Z-Buffer que se actualiza cuando un fragmento pasa el Z-Test, el Ω-Buffer no se ha de actualizar sea cual sea el resultado del Ω-Test. Esto se debe a que no hay ninguna necesidad puesto que la estructura encargada de mantener actualizadas las profundidades de cada uno de los fragmentos sigue siendo el Z-Buffer.

4.1. Implementación y funcionamiento

En la Figura 4.2 se muestran las modificaciones que se han realizado a una arquitectura TBR para implementar nuestro mecanismo de Ω-Test. Esta técnica reduce el *overdraw* de una escena respecto a una arquitectura TBR convencional en tanto que cada fragmento tiene que pasar un segundo test. En este aspecto, se puede decir que Ω-Test es más restrictivo, pues tienen que cumplirse más condiciones para que un fragmento pase a la etapa de sombreado. Dicho de otro modo, el Ω-Test se puede ver como un test de refuerzo para aquellos casos en los que el Early Z-Test tradicional no funciona eficientemente (por ejemplo, cuando las primitivas llegan en un orden de atrás hacia delante), ofreciendo una segunda oportunidad de eliminar fragmentos potencialmente ocluidos usando los valores de profundidad del Z-Buffer del fotograma previo. Debido a la coherencia entre fotogramas, si un fragmento fue descartado en el fotograma anterior, es muy probable que también se descarte en el fotograma actual. Sin embargo, y debido a que la información de profundidad del Z-Buffer de un fotograma a otro varía, esta técnica no garantiza que un fragmento descartado por el Ω-Test no sea visible en la imagen final. Por ejemplo, si un objeto que hay delante de la cámara desaparece repentinamente, los valores de profundidad del Ω-Buffer serán más cercanos que los equivalentes del Z-Buffer, por lo que los fragmentos que hubiese situados entre estos valores no se renderizarían, dando lugar a un potencial error. Este resultado se debe a la naturaleza predictiva de Ω-Test. Para estos pocos casos en los que el Ω-Test no acierta la predicción y pudiendo llegar a generar un error en la pantalla, esta técnica implementa un mecanismo de recuperación de la imagen capaz de corregir los errores generados, tal y como se detalla en la Sección 4.2.

Siendo más específicos, se pueden dar tres situaciones en función de los resultados del Z-Test y el Ω-Test, a saber:

- Caso 1: el fragmento entrante no supera el Early Z-Test y, por lo tanto, se ignora el resultado del Ω-Test. En este caso el fragmento se descarta de manera segura, es decir, no se producirá ningún error potencial dado que la decisión está basada en información actualizada del fotograma actual (almacenada en

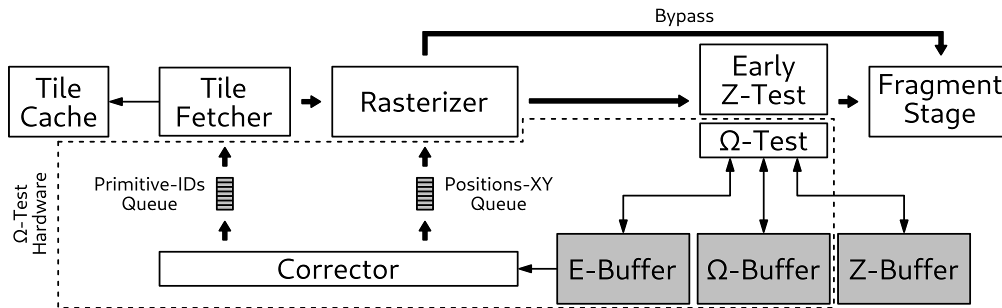


Figura 4.2: Implementación de Ω-Test sobre una arquitectura TBR. Las nuevas estructuras están delimitadas por una línea de puntos para diferenciarlas mejor

el Z-Buffer) y, en consecuencia, no hay especulación.

- Caso 2: el fragmento entrante supera el Early Z-Test, pero no el Ω-Test. Según el Early Z-Test el fragmento debería ser visible en ese momento, pero la información del fotograma previo (proveniente del Ω-Buffer) indica que en el fotograma anterior dicho fragmento habría sido descartado y, por lo tanto, se descarta de manera especulativa. Aquí la decisión está basada en la información del fotograma previo, por lo que se puede producir un error potencial. Éste es el único caso en el que se puede provocar un error potencial.

- Caso 3: el fragmento entrante pasa el Z-Test, y también el Ω-Test. En este caso el fragmento pasa a la etapa de sombreado e iluminación. Esta situación no produce ningún error en tanto que ha pasado ambos tests, pero puede provocar falsos positivos si el fragmento en cuestión es ocluido por otro más reciente.

Uno de los principales aspectos que caracterizan una arquitectura TBR es que el conjunto de datos de trabajo (*working set*) sobre el que se opera es un *tile*, lo que quiere decir que los requerimientos de memoria están acotados al tamaño del *tile*. Sin embargo, una vez que se termina de procesar un *tile*, la valiosa información que contiene el Z-Buffer se descarta completamente. Como lo que queremos es ser capaces de usar esa información relativa a las profundidades de cada fragmento del *tile* en el fotograma siguiente, esta información debe preservarse de algún modo. Para ello, la técnica propuesta emplea una estructura a nivel de fotograma denominada

Ω-Table en la que se almacena la información de todos los valores de profundidad de cada *tile* del fotograma anterior. Si decidiéramos implementar dicha estructura usando memoria *on-chip*, las necesidades de almacenamiento para un fotograma de resolución Full-HD (1920x1080 píxeles) serían de alrededor de 8 MiB (asumiendo 4 bytes por valor de profundidad), lo que contradiría completamente la filosofía TBR, que alienta el uso de pequeñas memorias *on-chip* para mejorar la eficiencia energética. Así pues, esta opción queda descartada.

Otra solución a este problema de almacenamiento sería almacenar esta información en la memoria DRAM del sistema, en lugar de en una memoria *on-chip*. El inconveniente de esta solución es el altísimo tráfico a memoria antes y después de procesar un *tile*, pues antes de procesarlo se necesitaría cargar el Ω-Buffer desde DRAM, y al finalizar, almacenar todos los valores de profundidad del Z-Buffer en DRAM. Esta solución daría lugar a un coste energético prohibitivo, pues hay que tener en cuenta que la memoria DRAM consume más del 50 % de la energía total de la arquitectura base, tal y como se indica en el Capítulo 1. Aun así, esta solución se ha evaluado de manera cuantitativa pudiéndose verificar que el efecto neto en el consumo energético global del sistema resulta ser negativo, puesto que el coste de los accesos adicionales a DRAM sobrepasa los beneficios obtenidos de reducir el *overdraw*. Por lo tanto, esta opción también ha sido descarta.

Para afrontar las necesidades de almacenamiento de la Ω-Table, sin que éstas perjudiquen severamente los costes energéticos, la solución pasa por no almacenar todos los valores de profundidad del *tile* que se está procesando, sino almacenar sólo unos pocos que sean representativos del global. A pesar de que esto provoca inevitablemente una pérdida de información, como se verá más adelante, se ha observado que almacenar sólo unos cuantos valores representativos del *tile* dando resultados casi tan efectivos como almacenarlos todos. Como es obvio, hay una pequeña cantidad de errores inducidos por el uso de información menos precisa, pero esto no supone ningún problema serio gracias a que el mecanismo de detección y corrección de errores (véase Sección 4.2.3 para más detalles) los arregla de manera eficiente. Además, la sobrecarga inducida por la fase de corrección se amortiza por el hecho de no tener que usar una memoria *on-chip* grande, ni acceder a memoria DRAM para almacenar los valores del Z-Buffer del fotograma previo.

Existen varias formas de seleccionar valores representativos para comprimir la Ω-Table. Sin embargo, se ha evitado usar algoritmos tradicionales de compresión debido a su alta complejidad *hardware* y coste energético. Como se ha mencionado anteriormente, el Ω-Test ya genera algunos errores (que se corrigen posteriormente) debido a que reutiliza la información de profundidad del fotograma anterior. Por lo tanto, no es necesario ser 100 % preciso con la información a usar, pues esta técnica es capaz de afrontar algunos errores iniciales que son corregidos convenientemente. Esto significa que para esta técnica, un esquema de compresión con pérdida de información pero rápido prevalece sobre otro sin pérdida de compresión, lento y costoso en energía. Como solución intermedia, se ha decidido hacer uso del *coarsening* y funciones de agregado convencionales (por ejemplo, las funciones máximo, mínimo o la media aritmética) para seleccionar un conjunto de valores de profundidad representativos. La idea que hay detrás del *coarsening* es almacenar un único valor para un conjunto de píxeles vecinos, basándose en la observación de que píxeles vecinos tienden a tener la misma profundidad o a ser muy parecidas. Por otro lado, las funciones de agregado son sencillas y fáciles de implementar en *hardware*, sobre todo las funciones máximo y mínimo.

Definimos un factor de *coarsening* que representa el nivel de granularidad que se usa para construir la Ω-Table. Por ejemplo, asumiendo *tiles* de 16x16 píxeles, un factor de *coarsening* de 8x8 dividiría el *tile* en 4 sub-cuadrados no solapados de 8x8. Después, cada sub-cuadrado de 8x8 pasa por la función de agregado para obtener un único valor de profundidad. El resultado es una matriz de 2x2 elementos que contiene los valores resultantes de aplicar la función de agregado a cada subcuadrado. En resumen, de una matriz de 16x16 pasamos a una matriz de 2x2, lo que reduce las necesidades de almacenamiento en un factor de 64×. Para ilustrar mejor este concepto, la Figura 4.3 muestra este esquema de compresión basado en el uso del *coarsening* junto con una función de agregado.

La Tabla 4.1 resume las necesidades de almacenamiento de la Ω-Table en función del factor de *coarsening* para una resolución de pantalla HD estándar (1280x720 píxeles) y *tiles* de 16x16 píxeles. Un factor de *coarsening* de 1x1 sería equivalente a no aplicar *coarsening*, es decir, almacenar todos los valores de profundidad del *tile* en la Ω-Table. Un factor de *coarsening* del mismo tamaño que el *tile*, en este caso de

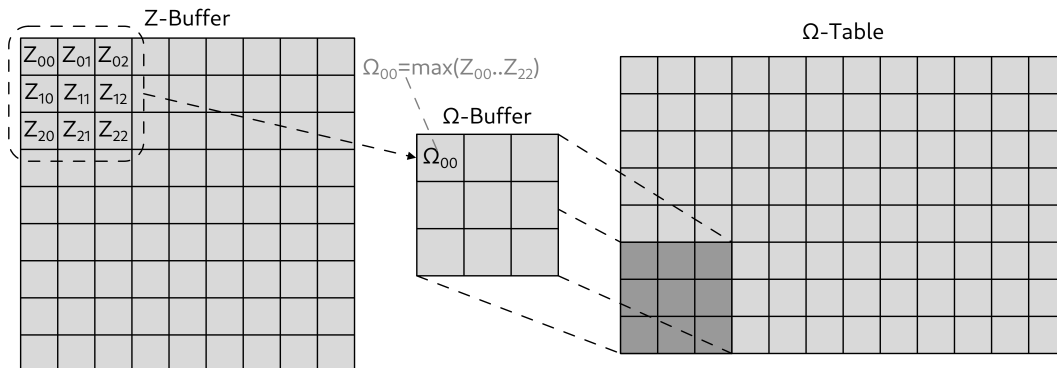


Figura 4.3: Ejemplo del esquema de compresión basado en *coarsening* usando un *tile* de 9x9 píxeles y un factor de *coarsening* de 3x3. El Z-Buffer se procesa en lotes no solapados de 3x3 píxeles, tal y como establece el factor de *coarsening*. Por cada lote se computa el máximo y se almacena en el Ω-Buffer. Una vez procesados todos los lotes de 3x3, el Ω-Buffer se escribe en la posición correspondiente de la Ω-Table.

Tabla 4.1: Necesidades de almacenamiento de la Ω-Table para los diferentes factores de *coarsening* posibles en un *tile* de 16x16, asumiendo una resolución de pantalla de 1280x720 píxeles.

Factor de *coarsening*	Tamaño de la Ω-Table
1x1 (o sin *coarsening*)	3.52 MiB
2x2	900 KiB
4x4	225 KiB
8x8	56.25 KiB
16x16	14.06 KiB

16x16, supone el factor máximo, y significa que el *tile* completo se representa con un único valor. Para entender mejor como se calcula el tamaño de la Ω-Table, consideremos el caso del factor de *coarsening* de 8x8. En este caso, sólo se almacenarán 4 (2x2 píxeles *gruesos* de 8x8) valores. Dado que una pantalla de 1280x720 píxeles tiene 3600 *tiles* (80x45) y proveyendo 32 bits (4 bytes) para cada valor de profundidad, el tamaño final de la Ω-Table será de 56.25 KiB (4 valores $\times$ 3600 *tiles* $\times$ 4 bytes), resultando en una reducción de almacenamiento de 64$\times$ respecto a no usar *coarsening* (el caso 1x1 en la Tabla 4.1).

Una vez diseñado este esquema de compresión, pasamos a medir la cantidad de errores iniciales provocados por el uso del *coarsening* usando diferentes funciones de agregado, a saber: máximo, mínimo y media aritmética. La Figura 4.4 mues-

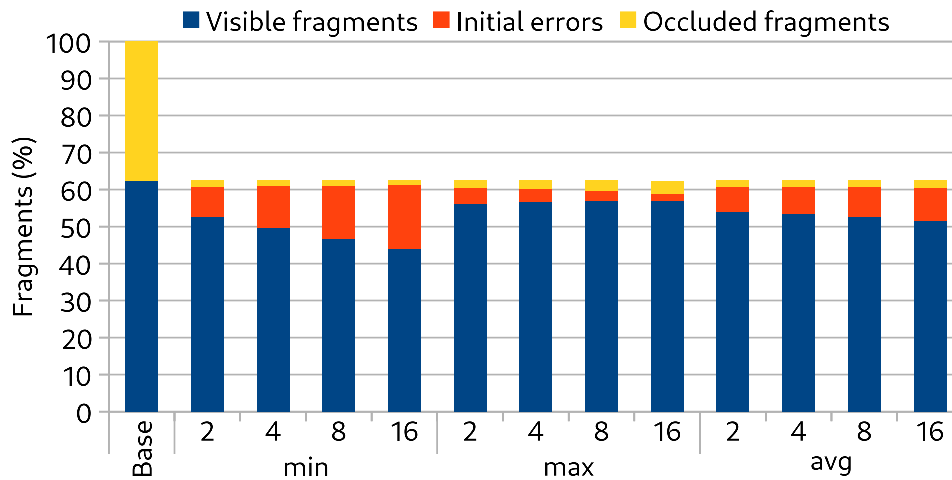


Figura 4.4: Estudio del impacto del *coarsening* y sus funciones de agregado sobre el *overdraw* y los errores iniciales en Ω-Test.

tra la fracción de errores respecto a la cantidad total de fragmentos procesados (incluyendo también el *shading* y el *overdraw* residual), donde cada barra representa la media de todas las escenas evaluadas (refiérase a la Sección 3.1 para más detalles).

Analicemos primeramente cómo se comportan las diferentes funciones de agregado. La función *mínimo* mantiene la profundidad de los fragmentos más cercanos a la cámara, por lo que el Ω-Test es más restrictivo y, por lo tanto, acaba descartando más fragmentos de los necesarios. El *overdraw* se reduce en detrimento de generar muchos errores, tal y como puede observarse en la Figura 4.4. Por otro lado, usar la función *máximo* hace más permisivo al Ω-Test en tanto que las profundidades de los fragmentos entrantes se comparan con los valores de profundidad más lejanos de cada grupo. El *overdraw* en este caso no se reduce tanto como si se usara la función *mínimo*, sin embargo, la cantidad de errores se reduce bastante más. Finalmente, usar la función *media aritmética* como función de agregado resulta en un equilibrio entre errores y *overdraw*, como era de esperar. Como la meta de esta técnica es generar la menor cantidad de errores posible debido al alto sobrecoste que éstos provocan durante la fase de corrección, se ha decidido finalmente optar por la función *máximo* como función de agregado para construir la Ω-Table, es decir, los valores de profundidad de cada grupo se representarán con el más distante a la cámara.

En cuanto al factor de *coarsening*, la Figura 4.4 también muestra que el factor de *coarsening* más grande posible (en este caso 16x16) combinado con la función de agregado *máximo*, no implica una pérdida significativa de potencial, pues supone menos del 3 % de los errores iniciales. Como resultado, para el diseño final de la Ω-Table se ha optado por usar un factor de *coarsening* de 16x16 que, en la práctica, significa que cada *tile* se representará con un único valor de profundidad, en concreto el máximo de todos los valores del grupo. Esto supone una reducción total de las necesidades de almacenamiento en un factor de 256×, sin penalizar de manera significativa el potencial de la técnica. Respecto a las necesidades de almacenamiento, usar un factor de *coarsening* de 16x16 resulta en una Ω-Table de menos de 14 KiB de memoria (Tabla 4.1), la cual se puede almacenar sin problemas en una pequeña memoria *on-chip*.

4.2. Manejo eficiente de los errores

4.2.1. Posibles escenarios que pueden provocar errores iniciales

La ventaja principal de Ω-Test es que reduce considerablemente la cantidad de fragmentos tipo *quad* (*quad fragments*) que pasan a ejecutarse a la etapa de sombreado (donde se ubican los *Fragment Processors*), descartando de manera especulativa aquéllos que están ocultos, y basándose para ello en el contenido de la Ω-Table. Una pequeña contrapartida de este mecanismo es que puede provocar errores en tanto que la información usada no es precisa al 100 %. Si bien es cierto que dado que hay coherencia entre fotogramas, los valores de las profundidades de un fotograma a otro son muy parecidas, pero no son los mismos. Para ilustrar mejor estos escenarios, la Figura 4.5 muestra los casos posibles que pueden dar lugar a errores potenciales y cómo Ω-Test los maneja.

La Figura 4.5-(a) muestra la escena inicial (fotograma i) con dos primitivas que se solapan y que se renderizan en orden de atrás hacia delante. El primer caso analizado se muestra en la Figura 4.5-(b), e ilustra el caso de una primitiva alejándose de la cámara. Es importante notar que cada fragmento de una primitiva tiene su valor de profundidad en el fotograma previo (obtenido de la Ω-Table) denominado

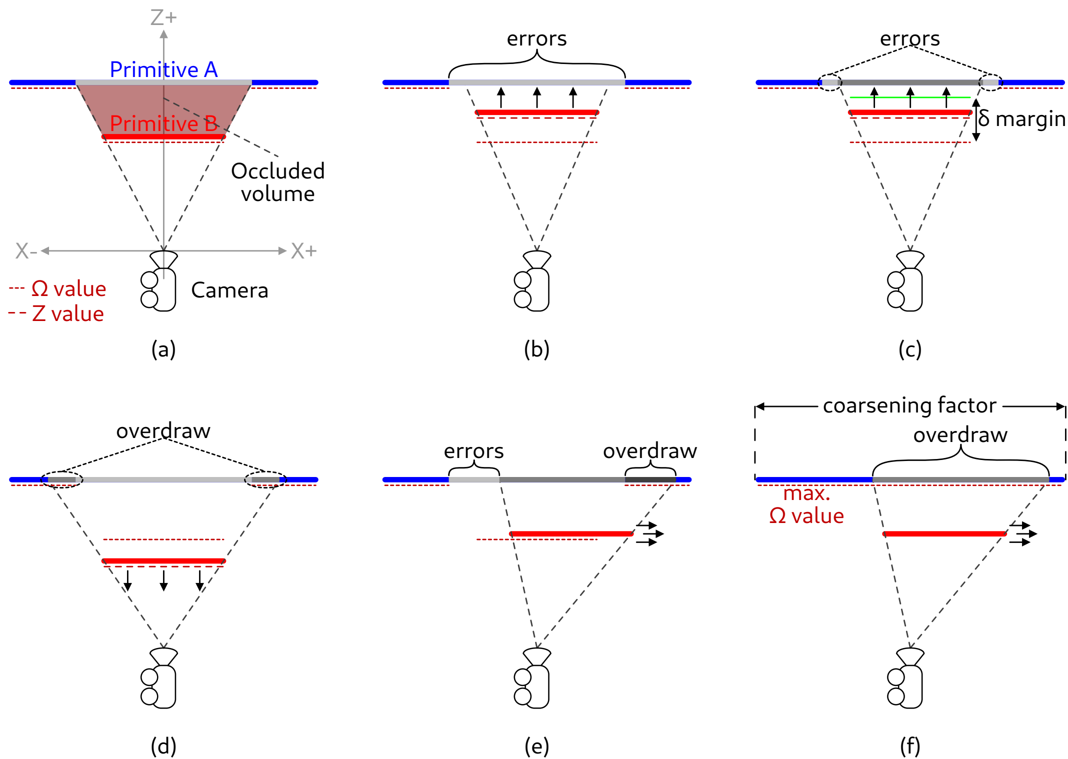


Figura 4.5: Conjunto de escenarios que podrían producir errores potenciales usando la técnica Ω-Test. (a) representa la escena inicial (fotograma i) con dos primitivas que se solapan renderizadas usando el algoritmo del pintor, es decir, de atrás hacia delante. (b)-(f) representan el fotograma $i+1$ con los posibles movimientos que puede realizar la primitiva B. Los casos (b) y (c) muestran la primitiva B alejándose de la cámara, lo que conlleva a errores potenciales en la imagen como se puede observar. En el caso (b) no se utiliza el margen δ, por lo que se producen errores en todo el área que ocupa la primitiva B sobre A. En el caso (c) sí que se utiliza el margen δ, y como puede observarse, los errores relativos a esta área desaparecen, quedando sólo unos pocos errores en el borde de la primitiva A. En el caso (d) se muestra la primitiva B acercándose a la cámara. Este caso no produce ningún error, puesto que los valores actuales de profundidad (los del Z-Buffer) son más cercanos que los del Ω-Buffer, por lo que se pasan ambos tests. En los casos (e) y (f) la primitiva B se mueve lateralmente. En el caso (e) no se aplica *coarsening*, lo que produce errores en la parte de la primitiva A que antes cubría la primitiva B, y produce *overdraw* en la parte hacia la que se mueve. En el caso (f) se utiliza el *coarsening* con la función de agregado *máximo*, lo que evita los errores provocados en el caso (e), puesto que los valores de profundidad del Ω-Buffer tienden a ser más lejanos que los del Z-Buffer, gracias al uso de la función *máximo* como función de agregado.

Z′, además de su propio valor de profundidad del fotograma actual, denominado Z. En este caso particular, como la primitiva B se ha movido hacia atrás, sus fragmentos fallarán el Ω-Test ($Z < Z'$), por lo que se producirá un error al no haber ninguna primitiva que tape el hueco negro que se ha generado.

En la Figura 4.5-(c) se muestra el caso de usar un margen δ, un mecanismo enfocado a mitigar los errores (véase Sección 4.2.2), con el que se pueden reducir una cantidad ingente de errores, siempre y cuando el valor de δ sea lo suficientemente grande como para *enmascarar* los movimientos hacia atrás. En este caso, los fragmentos de la primitiva B pasarán el Ω-Test ($Z < Z' + \delta$) mientras que los de la primitiva A que estaban ocluidos en el fotograma previo seguirán fallando, dando lugar a un ahorro del *overdraw*, sin generar muchos errores. Nótese que si el margen δ es muy grande puede provocar una situación de *overdraw*, pues ambas primitivas se renderizarían completamente, impidiendo que la técnica Ω-Test obtenga beneficio.

La Figura 4.5-(d) muestra el movimiento contrario, es decir, una primitiva que se acerca a la cámara (o a la inversa, la cámara moviéndose hacia delante, haciendo que la primitiva se acerque). Este caso no genera errores en tanto que los valores de profundidad actuales son más cercanos a la cámara que los del fotograma previo ($Z < Z'$) y, por lo tanto, los fragmentos pasarán el Ω-Test. Como mucho, este escenario podría provocar *overdraw*. Como ejemplo, los videojuegos evaluados incluyen juegos de carreras como Beach Buggy Racing o Hot Wheels (refiérase a la Tabla 3.2) en los que es usual que haya movimientos hacia delante de cámara.

Otro caso que puede llevar a potenciales errores son los movimientos laterales de una primitiva, tal y como se muestra en la Figura 4.5-(e). En este ejemplo, dado que la primitiva B se mueve de izquierda a derecha, la parte derecha de la primitiva A que estaba ocluida en el fotograma i se vuelve visible en el fotograma $i+1$, dando lugar a un potencial error en la imagen final.

Como caso final, la Figura 4.5-(f) muestra cómo el uso de una función de agregado *máximo* junto con el esquema de compresión con factor de *coarsening* mitiga considerablemente los errores producidos por movimientos laterales en detrimento del *overdraw*. En este caso, como todos los fragmentos de la primitiva B pasarán el

Ω-Test, no se generarán errores. Sin embargo, como las primitivas se renderizan en orden de atrás hacia delante se producirá *overdraw* como sucede en la arquitectura base.

Finalmente, es importante destacar que cualquier error, tanto si es provocado por un movimiento lateral como si se produce por el uso de un valor de profundidad "agregado" para representar un *tile* entero, éstos son resueltos por el mecanismo de corrección de errores descrito en la Sección 4.2.3. Por lo tanto, la imagen generada por la arquitectura con Ω-Test es *idéntica* a la generada por la arquitectura base.

4.2.2. Mitigando los errores: el margen delta (δ)

Es bastante común tener escenas en las que los valores de profundidad del Z-Buffer se mantienen igual en muchos *tiles* durante muchos fotogramas consecutivos, sobre todo si la escena es muy estática y no hay movimientos de cámara ni de objetos. Este escenario es ideal para Ω-Test porque no produce errores ni *overdraw*. Sin embargo, como se ha descrito en la Sección 4.2.1, existen otros escenarios en los que los objetos o la cámara se mueven en la escena que afectan a varios *tiles* del fotograma. Como se puede observar en los casos mostrados en la Figura 4.5-(b) y en la Figura 4.5-(c), existe un tipo de movimiento particular que lleva a errores cuando los objetos (o la cámara) se alejan.

Para ser capaces de tolerar esos movimientos de modo que no se produzcan errores, Ω-Test incluye un margen de seguridad en la Ω-Table llamado *margen delta (δ)*. Con esto se consigue relajar la condición del Ω-Test, haciéndolo menos restrictivo. El resultado es equivalente a desplazar ligeramente todos sus valores de profundidad un poco más lejos de la cámara. La idea de este margen δ es ser más permisivo haciendo que no se eliminen fragmentos de primitivas que se mueven ligeramente hacia atrás de un fotograma a otro. Usando este margen de seguridad, Ω-Test se vuelve más flexible, provocando así menos errores. Por otro lado, la cantidad de *overdraw* extra inducido por este margen es casi despreciable, por lo que merece la pena.

En esencia, lo que se consigue con este mecanismo es intercambiar errores por

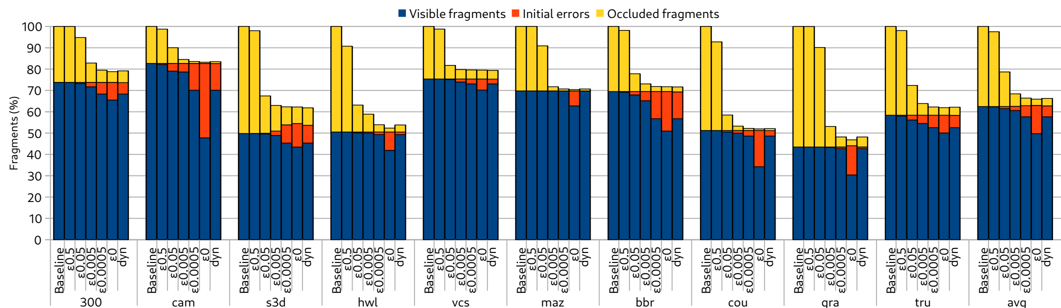


Figura 4.6: Estudio del ratio *overdraw*/errores analizando varios valores para δ: 0.5, 0.05, 0.005, 0.0005 y 0. Los resultados están normalizados al *baseline*. La última barra de cada *benchmark* (dyn) corresponde a la implementación del mecanismo de margen δ dinámico.

overdraw, es decir, el margen δ ayuda a reducir los errores pagando por ello un poco de *overdraw*, por lo que se pierde un poco de potencial. Para entender mejor el efecto provocado por este margen de seguridad, se ha analizado un amplio rango de valores de δ, observando como éstos afectan al *overdraw* y a los errores potenciales. En la Figura 4.6 se puede observar cómo, a medida que se aumenta el valor de δ, la cantidad de fragmentos que pasan el Ω-Test es mayor, como es lógico, dando lugar a más *overdraw*. Sin embargo, el número de errores potenciales se reduce considerablemente. Contrariamente, con valores más pequeños de δ el Ω-Buffer es menos tolerante a cambios de profundidad, provocando más errores, pero siendo más efectivo reduciendo el *overdraw*. La Figura 4.6 muestra que el valor δ más efectivo es 0.0005 para muchos *benchmarks*[1]. Es importante destacar la importancia que tiene la coherencia entre fotogramas en el margen δ, pues cuanta más coherencia, más suaves serán los movimientos en la escena y, por lo tanto, un δ menor será suficiente para capturar dichos movimientos.

A pesar de que la idea del margen δ es prometedora, usar un valor fijo para este parámetro puede no funcionar bien en todos los escenarios, pues hay una gran variedad de movimientos presente en los juegos. Para manejar de manera correcta estas variaciones entre fotogramas, Ω-Test implementa un mecanismo adicional muy simple que adapta dinámicamente el valor δ en función de si los objetos en la

[1] Recordemos que los valores de profundidad del Z-Buffer están normalizados en el rango de $[0, 1]$, siendo 0 el *near plane* y 1 el *far plane*.

escena se mueven mucho o no dentro de un fotograma. Así pues, este mecanismo dinámico elige un buen valor δ para cada fotograma en particular.

Este mecanismo adaptativo cambia el margen δ basándose en el ratio $overdraw/errores$ del fotograma previo. Para ello, se define una función de coste con estos dos parámetros como entrada (Ecuación 4.1). En dicha ecuación, c_o es el coste relativo asociado al *overdraw*, mientras que c_e es el coste relativo asociado a los errores, en términos de energía. La idea es establecer un equilibrio admisible de *overdraw* y errores. Estos dos parámetros de coste son fijos, y se han determinado de manera experimental evaluando el coste tanto de los errores como el del *overdraw* en todos los *benchmarks*. Dicho estudio revela que el coste de corregir un error es $3\times$ superior al de hacer el renderizado de un fragmento, por lo que este esquema adaptativo prioriza reducir la cantidad de errores inducidos estableciendo un peso para $c_o = 0.5$ y $c_e = 0.75$. Finalmente, en esta ecuación los parámetros o y e representan la cantidad de *overdraw* por fotograma y los errores potenciales, respectivamente.

$$cost(o, e) = c_o \times o + c_e \times e \tag{4.1}$$

El mecanismo dinámico usa una tabla de 8 valores de δ, a saber: 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1 y 0.5. También se usa un índice que apunta a una entrada de dicha tabla, que es el que selecciona el valor actual de δ. También precisa de una variable booleana (un bit) que indica en qué dirección se mueve el índice anterior sobre la tabla, y dos contadores que cuentan la cantidad de errores inducidos y el *overdraw* en el fotograma que se está procesando. Inicialmente, el valor de δ se establece en 0.0005 dado que es el valor estático que mejor funciona, en promedio, en todos los *benchmarks*. El esquema adaptativo funciona de la siguiente manera. Cuando se termina de renderizar un fotograma, se evalúa la función de coste definida por la Ecuación 4.1 y el resultado se compara con el obtenido en el fotograma anterior, que ha sido previamente almacenado en un registro global. Si el coste actual es mayor que el del fotograma previo, se cambia la dirección en la que se mueve el índice de la tabla con una simple operación `not` lógica. Esto nos indica que el valor δ no va en la dirección correcta dado que el coste está subiendo, de ahí que se cambie la dirección en la que se mueve el índice de la tabla. En

caso contrario, se mueve el índice una posición hacia la dirección indicada por el bit de dirección, pues la dirección es correcta, pero no la magnitud de δ. Si el índice alcanza el máximo o el mínimo de la tabla, se queda donde está.

Este esquema dinámico que selecciona de manera dinámica el valor δ es capaz de reducir la fracción de errores potenciales como se pretendía. En particular, se puede observar en la Figura 4.6 que la fracción media de errores es tan sólo el 5.08 % del total de fragmentos de la escena gracias al uso de un δ dinámico.

Se ha evaluado otro mecanismo δ dinámico pero calculado a nivel de *tile*, en lugar de a nivel de fotograma. Sin embargo, este esquema más refinado no se ha incluido en la implementación final debido al sobrecoste de las necesidades de almacenamiento y al poco beneficio obtenido respecto a hacerlo a nivel de fotograma. Dado que cada *tile* necesita almacenar su propio valor δ, se requerirían 32 KiB de memoria asumiendo una resolución FullHD, *tiles* de 16x16 píxeles y 4 bytes por cada δ. Los resultados experimentales obtenidos con este mecanismo a nivel de *tile* mostraron una reducción de los errores, pasando de 5.08 % a 5.02 %, llegando a la conclusión de que dicho mecanismo más complejo no merece la pena.

4.2.3. Detección y corrección de errores

Tal y como se ha comentado previamente, dado que la técnica Ω-Test puede descartar fragmentos que deberían ser visibles en la imagen final, es necesario un mecanismo para detectar y corregir los errores generados por esta acción. Los errores se generan en la etapa Early Z-Test, donde se determina si un fragmento debe pasar o no a la etapa *Fragment Processing* para proceder al su renderizado. Un fragmento que pasa el Early Z-Test tradicional pero no el Ω-Test (caso 2 explicado en la Sección 4.1) puede provocar un error potencial. Sin embargo, nótese que otro fragmento visible que lo solape puede ocultar a éste. En este caso, Ω-Test ha evitado una situación de *overdraw*, ahorrando así trabajo innecesario. Sin embargo, si a esa posición no llega ningún fragmento que supere ambos tests, se producirá un *hueco* en el Color Buffer final. Como es lógico, estos *huecos* no deben visualizarse en el Frame Buffer final. Es por ello que, aunque sean pocos casos, se requiere un procedimiento de corrección de estos potenciales errores.

Para llevar un seguimiento de los errores potenciales generados durante el procesamiento de un *tile*, se necesitan algunas estructuras de datos adicionales. En concreto, se necesita un *array* bidimensional llamado E-Buffer (de *Error Buffer*) del mismo tamaño que un *tile*, donde cada elemento almacena el identificador de la primitiva que debería aparecer en la posición correspondiente. Una vez el fragmento evaluado pasa el Early Z-Test, se evalúa el Ω-Test. Si éste falla (caso 2), se produce un *posible* error en tanto que se ha descartado el fragmento de forma especulativa. Si luego resulta que este fragmento era visible, necesitamos anotar la información necesaria para recuperarlo. Esta información es el identificador de primitiva, junto a la posición implícita en el E-Buffer. Así pues, en este caso se escribe el identificado de la primitiva correspondiente al fragmento procesado (esta información viaja junto a los atributos del fragmento durante todo el *pipeline*) en la posición correspondiente del E-Buffer. Por el contrario, si el fragmento supera Ω-Test, entonces el fragmento es visible en ese momento y, por lo tanto, no se ha producido ningún error. En este caso se elimina (si lo hubiera) el identificador de primitiva del E-Buffer de la posición correspondiente. Para indicar esto se usa un identificador de primitiva especial, que es el valor -1. Este valor indica que no hay error en esa posición del E-Buffer.

Una vez todas las primitivas *opacas* han sido rasterizadas y la etapa Early Z-Test no tiene más fragmentos que procesar, el E-Buffer se encuentra en su estado final, indicando dónde se encuentran los errores finales (fragmentos no renderizados o erróneamente descartados) dentro del *tile*. En este punto, la fase de corrección está lista para ser ejecutada. Es importante destacar que esta fase sólo se lleva a cabo si hay algún error en el E-Buffer, por lo que no se ejecuta innecesariamente cuando no hay errores. Para comprobar de manera rápida y eficiente esta condición se hace uso de un contador global que indica el número de errores que se han ido cometiendo en la etapa Early Z-Test. Este contador se encuentra inicialmente a 0 al comenzar el procesamiento del *tile*, se incrementa cuando se almacena un identificador de primitiva en el E-Buffer y se decrementa cuando se almacena un -1 en una posición donde había un identificador de primitiva válido.

Para ilustrar mejor este proceso de corrección, la Figura 4.7 muestra el funcionamiento del Corrector. En primer lugar, el Corrector lee *quad fragments* del E-

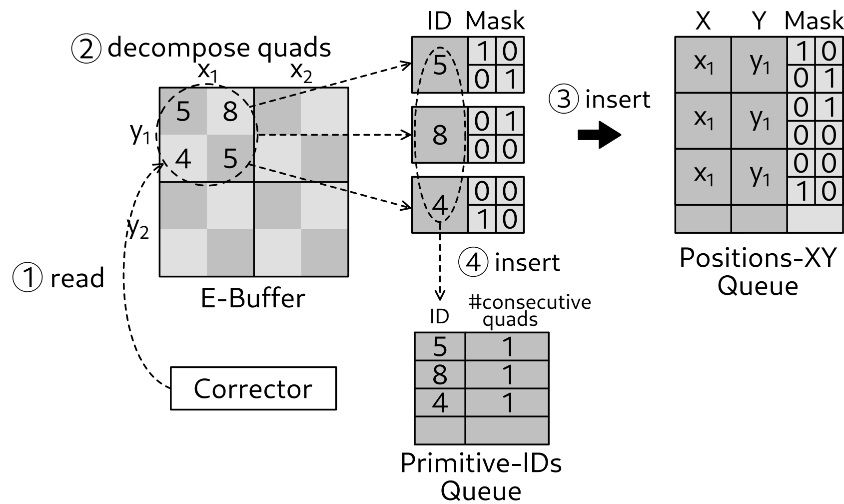


Figura 4.7: Implementación de la fase de corrección. Cualquier número diferente de -1 en cualquier celda del E-Buffer indica un error a corregir en dicha posición. En este ejemplo concreto hay 4 errores en el *quad fragment* situado en la esquina superior izquierda, correspondientes a las primitivas 5, 8 y 4.

Buffer (paso ①). En este ejemplo, el *quad fragment* situado en la esquina superior izquierda contiene 4 errores que corresponden a tres primitivas distintas (identificadores 5, 8 y 4). Por cada primitiva *diferente* se genera una máscara de visibilidad de 4 bits (2x2) cuyos bits activos corresponden a los píxeles que la primitiva ocupa dentro del *quad fragment* (paso ②). Por lo tanto, un *quad fragment* puede generar hasta cuatro máscaras de visibilidad si todos los errores que contiene provienen de primitivas diferentes[2]. Estas máscaras de visibilidad, junto con sus correspondientes posiciones dentro del *tile*, se almacenan en la cola *Positions-XY queue* (paso ③). Si todas las primitivas válidas del *quad fragment* (según la máscara de visibilidad) son iguales (éste es el mejor caso y, afortunadamente, el más común) se incrementa otro contador global, indicando el número de *quad fragments* consecutivos (en orden *scan-line*) de la misma primitiva. Cuando el Corrector encuentra una primitiva diferente, se inserta una nueva entrada en la cola *Primitive-ID queue* (paso ④) que contiene tanto el identificador de la primitiva anterior como un contador que indica que número de *quad fragments* consecutivos de la misma primitiva (éste último se establece a cero por cada nueva primitiva).

[2]Éste es el peor caso. Sin embargo, no es el más usual dado que hay homogeneidad en los errores producidos por una primitiva.

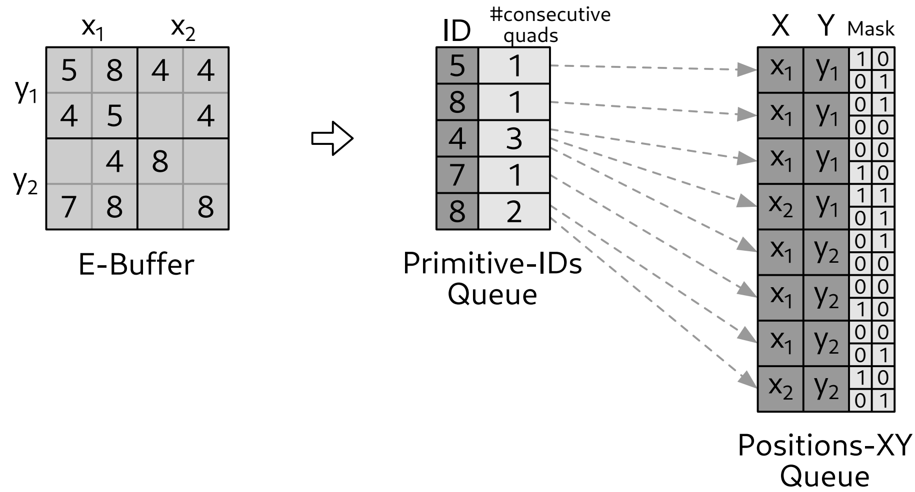


Figura 4.8: Ejemplo del estado final de las colas *Primitive-ID queue* y *Position-XY queue* dado un E-Buffer de 4x4 píxeles (4 *quad fragments*).

Para entender mejor cómo funciona el Corrector, la Figura 4.8 muestra un ejemplo de funcionamiento del Corrector con un E-Buffer que contiene varios errores. Las entradas del E-Buffer con un número (un identificador de primitiva) corresponden a posiciones donde hay un error para esa primitiva concreta, mientras que las celdas vacías representan posiciones del *tile* que contienen información correcta (es decir, un -1). Por motivos de claridad, en este ejemplo se asumen *tiles* de 4x4 píxeles. Asimismo, en la Figura 4.8 también se muestra el estado final de las colas *Primitive-ID queue* y *Positions-XY queue*.

Tan pronto hay una primitiva disponible en la cola *Primitive-ID queue*, el Tile Fetcher empieza a trabajar buscando las primitivas que contiene para corregir los errores generados. Aquí el Tile Fetcher trabaja en modo corrección en lugar de en el modo tradicional. En este modo, en vez de buscar todas las primitivas del Tile que se está procesando sólo busca aquéllas que encuentra en la *Primitive-ID queue*, ahorrándose la búsqueda del resto de primitivas. Además del identificador de primitiva, la *Primitive-ID queue* proporciona el número de fragmentos que se deben corregir de dicha primitiva, información que se le transfiere al Rasterizer, para que éste sólo corrija únicamente los fragmentos que han producido errores, en vez de rasterizar la primitiva completa. Este mecanismo evita, por un lado, que el Tile Fetcher busque la misma primitiva varias veces si ésta ha producido varios errores y,

por otro lado, que se llene la *Primitive-ID queue* de información redundante.

El Rasterizer también tiene un modo de corrección que lo hace comportarse de un modo ligeramente diferente. En este modo, el Rasterizador recibe el número de fragmentos a corregir de la primitiva dada, información que le ha transferido previamente el Tile Fetcher, obtenida a su vez de la *Primitive-ID queue*. Por lo tanto, en vez de rasterizar la primitiva completa, simplemente rasteriza aquellos fragmentos que han producido un error. Por ejemplo, si un triángulo muy grande tiene un píxel erróneo, sólo se generará dicho fragmento, lo que ahorrará bastantes cálculos en la rasterización. Para hacer esto, el Rasterizer tiene que calcular las coordenadas baricéntricas del primer fragmento y los incrementos en X e Y como se hace convencionalmente. Las posiciones (X,Y) donde se producen los errores se obtienen de la cola *Position-XY queue*, junto a la máscara de visibilidad (véase Figura 4.7). Luego, el *quad fragment* se envía a la etapa *Fragment Processing* para realizar el sombreado pertinente. Nótese que los fragmentos corregidos no pasan por la etapa Early Z-Test dado que se sabe de antemano que son visibles, pues no en vano se están corrigiendo.

Una vez finalizada la fase de corrección, el *pipeline* gráfico continúa funcionando de manera convencional. Cuando el *tile* se renderiza por completo, y el Color Buffer se ha computado y volcado en el Frame Buffer, el *pipeline* gráfico está listo para comenzar el procesamiento del siguiente *tile*.

Finalmente, hay una situación difícil de resolver que sucede cuando el Tile Fetcher encuentra una primitiva transparente. Como se ha visto en la Sección 2.1.10, una primitiva se considera transparente si tiene su atributo de *blending* activado, indicando que todos sus fragmentos se deben *mezclar* con los ya existentes en el Color Buffer en ese momento. El problema con estas primitivas es que cuando se procesa uno de sus fragmentos, el Color Buffer debe contener en esa posición el color del fragmento opaco generado previamente, por lo que si se ha producido un error durante el Ω-Test en esa posición habrá un *hueco* (en color negro) y, por lo tanto, se realizará un *blending* incorrecto. Para solventar esta situación, la fase de corrección se dispara tan pronto llega un fragmento transparente a la etapa Early Z-Test, asegurando que cualquier error generado durante el procesamiento de la

geometría opaca se ha corregido debidamente. Cuando termina la corrección de los errores producidos por las primitivas opacas, el *pipeline* puede continuar procesando el fragmento transparente, y el Tile Fetcher continúa funcionando en modo normal (no en modo corrección). Esta situación provoca inevitablemente una detención en el *pipeline* que puede afectar negativamente al rendimiento de Ω-Test. Sin embargo, afortunadamente estas situaciones en las que se intercalan ráfagas de primitivas transparentes y opacas no suelen darse, pues dichos patrones no siguen las recomendaciones establecidas por el manual de programación de OpenGL sobre el orden de renderizado, que establece que las primitivas transparentes deben renderizarse después de las opacas para obtener una imagen correcta [78]. En concreto, el único *benchmark* de nuestra batería de pruebas que presenta este patrón anómalo es Maze 3D (maz), con un número muy reducido de primitivas. Como es de esperar, Ω-Test produce exactamente la misma imagen de salida que la arquitectura base, y debido a la poca ocurrencia de estos patrones, el rendimiento obtenido en Maze 3D (maz) no se ve degradado en absoluto, tal y como se verá en la Figura 4.10.

4.3. Resultados experimentales

En esta Sección se han evaluado una arquitectura TBR como sistema base así como una GPU que implementa el mecanismo Ω-Test, con un factor de *coarsening* de 16x16 (es decir, el *tile* completo queda representado por un único valor), la función *máximo* como función de agregado, así como usando un margen δ dinámico. La Tabla 4.2 muestra en detalle los parámetros de simulación específicos usados para evaluar Ω-Test.

4.3.1. Reducción del *overdraw*

La Figura 4.9 muestra los primeros resultados obtenidos por la técnica Ω-Test. En concreto, esta figura muestra un desglose de todos los fragmentos renderizados de cada *benchmark*, diferenciando la fracción de fragmentos opacos ocluidos (es decir, lo que venimos denominando *overdraw*), los fragmentos transparentes,

Tabla 4.2: Parámetros de simulación usados en Ω-Test.

Baseline GPU Parameters	
Screen Resolution	1280x720
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Ω-Test hardware	
Ω-Table	14 KiB
Positions-XY Queue	64 entries, 13 bytes/entry
Primitive-ID Queue	64 entries, 8 bytes/entry
E-Buffer	1 KiB
Corrector	4 quad-fragments/cycle

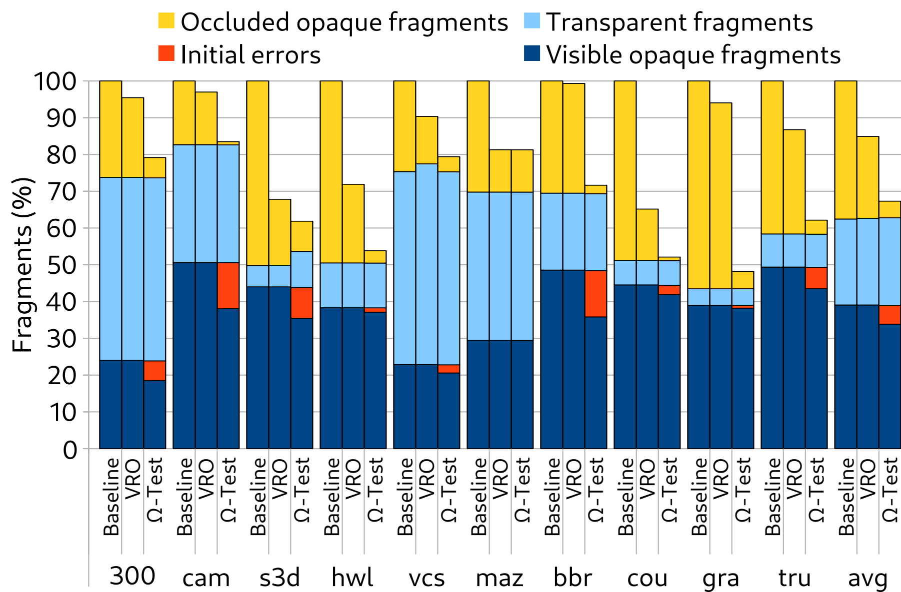


Figura 4.9: Desglose de los fragmentos renderizados para los *benchmarks* evaluados, comparando la arquitectura base, VRO y Ω-Test.

los errores inducidos por el Ω-Test (en rojo) y los fragmentos opacos que son visibles en el *framebuffer* del dispositivo (en azul oscuro). Como Ω-Test está enfocado a optimizar el problema de la visibilidad, sólo puede atacar a la fracción de *overdraw* (el segmento amarillo). Nótese que una implementación ideal capaz de determinar la visibilidad de una forma 100 % precisa eliminaría este segmento por completo. Con el propósito de hacer una comparación de rigor, hemos evaluado también VRO [23], una técnica avanzada de tipo *Hidden Surface Removal* (HSR) que ordena los objetos de la escena en orden de delante hacia atrás basada en la coherencia entre fotogramas (véase Sección 4.4 para más detalles). De media, se puede observar que Ω-Test reduce un 32.7 % el sombreado de fragmentos proveniente del *overdraw* causado en la arquitectura TBR base, dejando tan sólo un *overdraw* residual del 5 % que es incapaz de eliminar. Estos primeros resultados muestran que la propuesta de Ω-Test está muy cerca del caso ideal, a diferencia de VRO que deja un *overdraw* residual del 22.2 %. A pesar de que tanto VRO como Ω-Test aprovechan la coherencia entre fotogramas para reducir el *overdraw*, ambos consiguen este objetivo siguiendo estrategias muy diferentes. VRO trabaja a nivel de objeto, ordenando los comandos de OpenGL de modo que son procesados en orden de delante hacia atrás. Sin embargo, Ω-Test opera a un nivel muchísimo más bajo, a nivel de fragmento, lo que lo hace más efectivo al ser capaz de eliminar *overdraw* de primitivas que pertenecen al mismo objeto (*overdraw* intra-objeto) o de objetos que están parcialmente ocluidos. Ni qué decir tiene que el *overdraw* intra-objeto supone una fracción importante del *overdraw* total en los *benchmarks* evaluados, y en los vídeojuegos para dispositivos móviles en general, en tanto que típicamente incluyen objetos complejos en una única *draw call*. Por ejemplo, en Hot Wheels (hwl), la ciudad de fondo (compuesta por múltiples y complejos edificios - véase Figura 3.7) es un único objeto renderizado con una simple *draw call* de OpenGL. Como es de esperar, en esta escena hay mucho *overdraw* intra-objeto que VRO es incapaz de eliminar, mientras que Ω-Test sí que puede.

Por otro lado, los errores (es decir, los fragmentos que se han descartado de manera errónea por el Ω-Test y que deben ser visibles) pueden perjudicar el rendimiento puesto que tienen que ser corregidos en la fase de corrección, por lo que es deseable que sean el menor número posible. En este aspecto, Ω-Test genera de

media un 5.1 % de errores *iniciales* que son posteriormente corregidos en la fase de corrección[3]. Como apunte adicional, el número de errores de media que se corrigen por *tile* es, en términos absolutos, tan sólo 13 fragmentos de los posibles 256 visibles dentro de un *tile* de 16x16 píxeles.

4.3.2. Rendimiento y análisis del tráfico a memoria

En esta Sección se analiza el impacto neto general en el rendimiento debido tanto a la reducción de *overdraw* como a los errores que tienen que ser corregidos. La Figura 4.10 muestra el rendimiento obtenido en una arquitectura con Ω-Test en el *pipeline* gráfico. Se puede observar que Ω-Test logra una mejora media en el tiempo de ejecución del 16.3 %, llegando a alcanzar su máximo en Gravity (gra), con un 32.7 % de mejora. La Figura 4.10 también muestra los resultados obtenidos por una implementación idealizada de HSR sin *overheads*, con un conocimiento perfecto sobre la visibilidad de los fragmentos. Esta configuración muestra una cota superior del 17.9 %. También se muestra el rendimiento obtenido por VRO, que como se puede observar es del 12.75 %. Como es de esperar, Ω-Test está muy cerca del escenario perfecto debido a su bajo *overdraw* residual (tan sólo un 4.5 %), y además supera a VRO. Centrándonos en Ω-Test, a pesar de que reduce considerablemente el *overdraw* en muchas aplicaciones, en algunos casos el tiempo de ejecución no se reduce en la misma proporción. Esto se debe al *overhead* inducido por la corrección de los errores, y también debido a las detenciones en el *pipeline* gráfico provocadas por la espera a esta fase de corrección. Para tener una visión más clara de esto, es necesario fijarse en los accesos que acaban yendo a DRAM como resultado de los fallos en la Texture Cache (equivalente a una L1 en una CPU tradicional). Como se esperaba, las aplicaciones con texturas pequeñas son más propensas a obtener mejores tasas de acierto en la Texture Cache debido a que éstas pueden encajar perfectamente en esta caché y, por lo tanto, reducir *overdraw* en estas aplicaciones no se traduce en una reducción proporcional de los accesos a DRAM, y tampoco en la latencia. Por el contrario, las aplicaciones que tienen texturas más detalladas

[3]Volvemos a destacar que Ω-Test en ningún caso produce una imagen incorrecta, pues a pesar de que hablamos de errores, éstos son corregidos finalmente en la fase de corrección (de ahí el término *iniciales*), por lo que la imagen generada será siempre igual a la que produce la arquitectura TBR base.

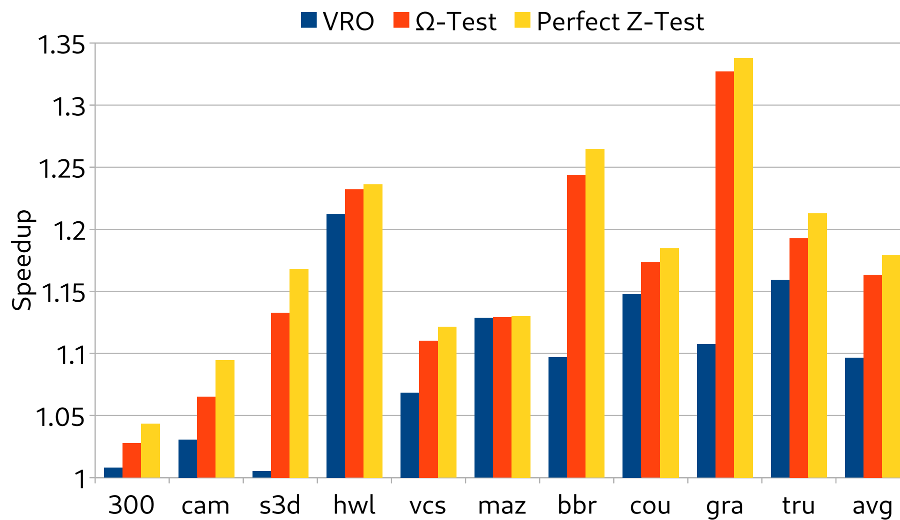


Figura 4.10: Comparación del rendimiento entre VRO, Ω-Test y una implementación HSR perfecta. Los resultados de tiempo de ejecución están normalizados respecto a la arquitectura base.

obtienen una tasa de acierto más baja en la Texture Cache, por lo que generan más tráfico a DRAM y sacan más partido a Ω-Test. Por consecuencia, reducir el factor de *overdraw* en estas aplicaciones tiene un impacto más significativo. Éste es el caso de Gravity (gra), Hot Wheels (hwl) y Beach Buggy Racing (bbr).

Para tener una idea más detallada de este efecto, las Figuras 4.11 y 4.12 muestran los accesos a la Texture Cache, a la caché L2 y a la DRAM. Mientras que en la Figura 4.11 se desglosan los accesos a la Texture Cache en aciertos y fallos (estos acaban yendo a la L2), en la Figura 4.12 se muestra la cantidad de peticiones que sirve la DRAM. Vamos a fijarnos en dos ejemplos representativos: Counter Strike (cou) y Gravity (gra). Antes de nada, hay que destacar que ambos *benchmarks* obtienen reducciones netas de *overdraw* (véase Figura 4.9) similares, de 47.9 % y 51.8 % respectivamente. Sin embargo, estas reducciones se traducen en unas mejoras del rendimiento del 17.4 % y el 32.7 % respectivamente. Si nos fijamos en el comportamiento que estos dos *benchmarks* tienen en la memoria (Figura 4.11), se observa que, gracias al uso de Ω-Test, se reduce el número de demandas al sistema de memoria en un 46.2 % y un 41.2 % respectivamente, lo que correlaciona bastante bien con las reducciones de *overdraw* reportadas en la Figura 4.9. Sin embargo, mientras que en Gravity se reduce la cantidad de accesos a DRAM en un 30.3 %, en Counter

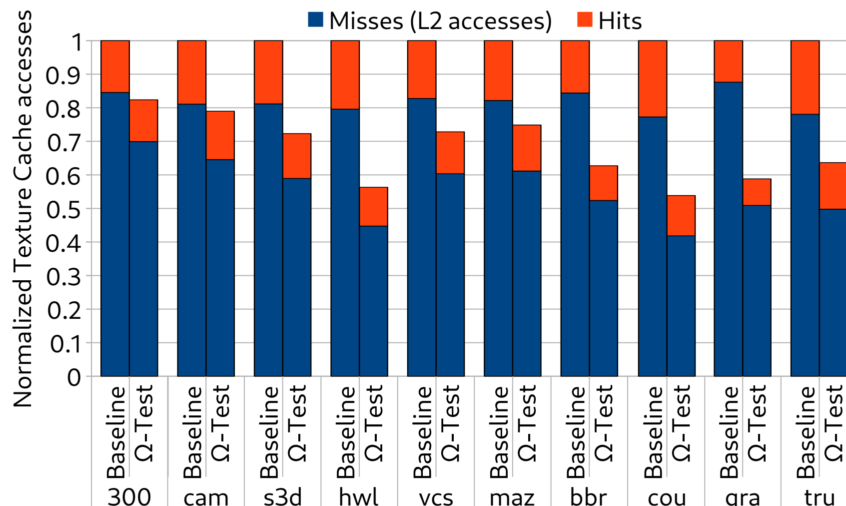


Figura 4.11: Accesos a la Texture Cache desglosados en aciertos y fallos que van a ir a la L2, normalizados respecto a la arquitectura base.

Strike apenas se ve reducida (tan sólo un 3 %), tal y como puede observarse en la Figura 4.11. Esto se debe a que la complejidad de las texturas es diferente en cada *benchmark* (Gravity tiene texturas más grandes, mientras que Counter Strike tiene texturas que pueden caber en la Texture Cache por completo o en la L2, generando así menos tráfico a memoria DRAM).

4.3.3. Ahorro de energía

En esta Sección vamos a centrarnos en las ganancias obtenidas por Ω-Test en términos de energía. La Figura 4.13 muestra el consumo energético obtenido por Ω-Test en cada uno de los *benchmarks* evaluados. Se puede observar que se obtienen unos ahorros de energía medios del 15.17 %, llegando a reducir un máximo del 26.9 % en Gravity (gra), superando con creces a VRO, que tan sólo alcanza un ahorro de energía del 9.6 %. Juegos como Vegas Crime Simulator (vcs), Gravity (gra) o Beach Buggy Racing (bbr) obtienen un mejor rendimiento con Ω-Test debido a la alta complejidad de sus texturas, tal y como se ha explicado anteriormente, dado que gran parte de los costes del *overdraw* provienen de los accesos a memoria durante el proceso de texturizado, provocados por fallos en los niveles superiores. Otros *benchmarks* como Hot Wheels (hwl) o Counter Strike (cou) también obtienen

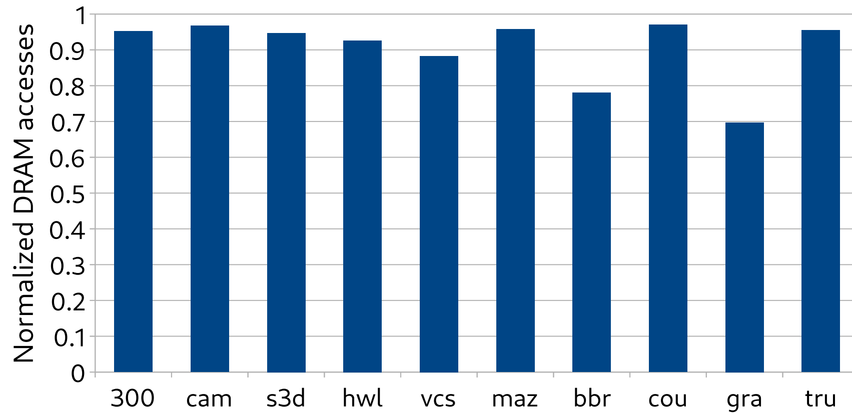


Figura 4.12: Cantidad total de accesos a DRAM (normalizados respecto a la arquitectura base) para los *benchmarks* evaluados.

grandes ahorros de energía (alrededor del 18.7 %), pero no debido al pobre comportamiento de la Texture Cache, sino a la gran cantidad de *overdraw* que presentan y que Ω-Test es capaz de eliminar.

Finalmente, para rematar esta batería de experimentos, la Figura 4.14 muestra la combinación de rendimiento y ahorro de energía en términos de EDP (*Energy-Delay Product*). Se puede observar que Ω-Test logra unos ahorros de EDP del 26.42 % de media, llegando a un máximo del 42.65 % en Gravity (gra), mientras que VRO se queda en menos de la mitad, un 20.34 % de media.

4.4. Trabajos relacionados

En esta Sección veremos los trabajos relacionados con la determinación de la visibilidad, clasificados en tres tipos, a saber: técnicas basadas en *Deferred Rendering*, técnicas basadas en el concepto de *HiZ occlusion culling*, y otros trabajos que aprovechan la coherencia entre fotogramas para mejorar el problema de la visibilidad. El conjunto de técnicas que pretende optimizar el problema de la visibilidad por medio de eliminación de primitivas ocultas se denomina *Hidden Surface Removal* (HSR).

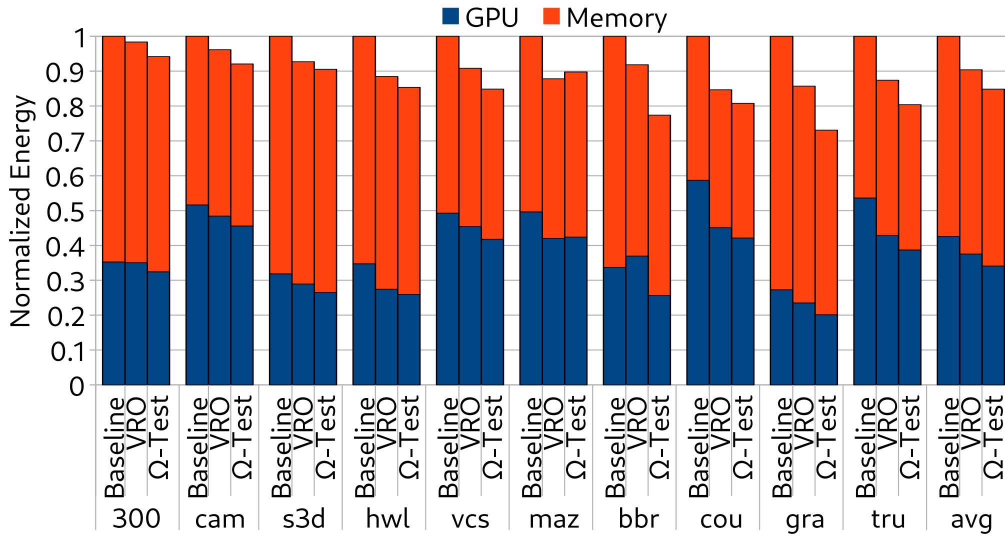


Figura 4.13: Ahorro de energía logrado por Ω-Test en una arquitectura TBR, normalizado respecto a la arquitectura base.

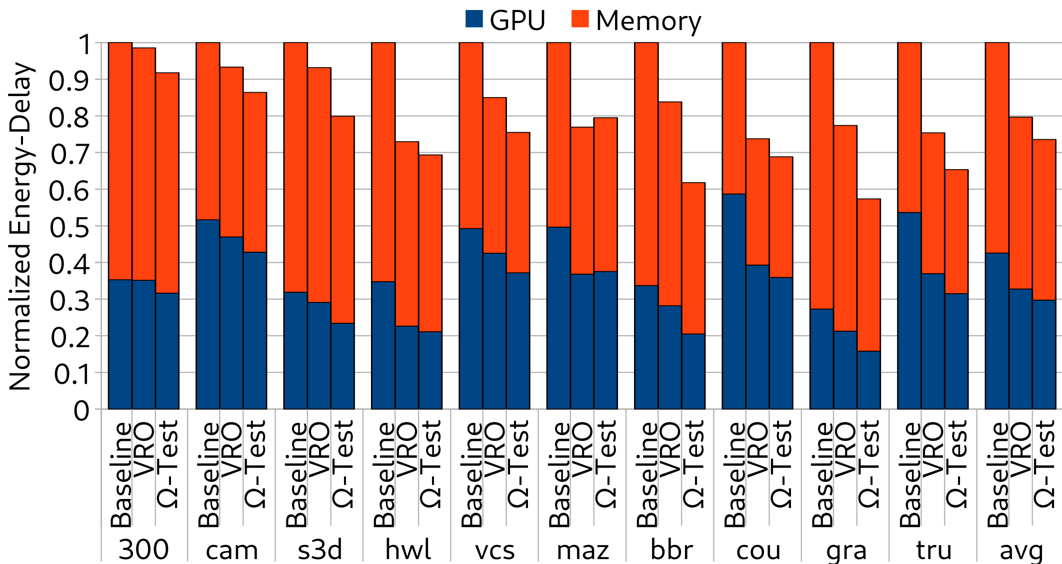


Figura 4.14: Ahorro en términos de EDP obtenidos gracias a Ω-Test respecto a la arquitectura base.

4.4.1. Técnicas basadas en *Deferred Rendering*

Al contrario de lo que sucede con un esquema tradicional de determinación de visibilidad, en el que la visibilidad se va computando a medida que van llegando los fragmentos al *pipeline* gráfico, los esquemas basados en *Deferred Rendering* (también conocidos como *Deferred Shading*) primero procesan toda la escena para determinar la visibilidad, y una vez se conoce ésta, se procede a renderizar la escena completa, de ahí el término *deferred* (diferido).

En [39] se propone un esquema denominado Z-Prepass, que se puede considerar como una técnica básica basada en *Deferred Rendering* dado que desacopla el procesamiento de la geometría del procesamiento del sombreado. En concreto, Z-Prepass es una técnica *software* capaz de eliminar todo el *overdraw* de una escena provocado por objetos ocultos. Para ello efectúa dos pasadas de renderizado en el nivel de aplicación. La primera pasada consiste en procesar toda la geometría de la escena y rasterizarla con un *shader* nulo (*null shader*). Esto es equivalente a anular la etapa *Fragment Processing*, por lo que los fragmentos sólo llegan a procesarse en la etapa Early Z-Test. De este modo, se computa completamente la información final del Z-Buffer. La segunda pasada consiste en rasterizar la escena de manera normal, pero ahora el Z-Buffer no parte de cero, sino que tiene toda la información necesaria para permitirle a la etapa Early Z-Test eliminar todo el *overdraw* provocado por las superficies opacas. Nótese que esta implementación, a pesar de eliminar por completo el *overdraw*, tiene sus inconvenientes que la hacen en ocasiones inviable en escenas complejas, pues es necesario procesar dos veces la escena y, aparte, las dos pasadas no se pueden ejecutar en paralelo dado que son dependientes la una de la otra, incurriendo en una barrera que perjudica gravemente al rendimiento.

El uso de G-Buffers [1, 81] también se ha extendido bastante. Estas estructuras de datos a menudo se usan en aplicaciones en forma de *shader programs* para desacoplar el procesamiento de la geometría del renderizado. Además, estas estructuras dan más flexibilidad a la hora de definir modelos de sombreado y cómputos de materiales en tanto que el programador las puede usar para lo que desee.

Tal y como se ha visto en la Sección 2.3.1, la arquitectura de las GPUs de PowerVR implementan una técnica *hardware* propietaria basada en el esquema *Defe-*

rred Rendering [51]. El problema que tiene esta técnica es el *overhead* que tiene, tanto en tiempo de ejecución como en *hardware* adicional, pues es necesario duplicar etapas del Raster Pipeline.

A diferencia de [51], Ω-Test no introduce una sobrecarga en el tiempo de ejecución del *pipeline* gráfico y además, en lo que a *hardware* se refiere, sus estructuras adicionales (la Ω-Table, el E-Buffer, el Ω-Buffer y las colas de corrección) son pequeñas memorias *on-chip*. En [23] se propone VRO, una técnica capaz de reducir el *overdraw* ordenando los objetos en orden de delante hacia atrás. En dicho trabajo, los autores se comparan cuantitativamente con TBDR, superándolo con creces. En este Capítulo se ha comparado Ω-Test con VRO (véase Sección 4.3), y se ha visto que Ω-Test supera a VRO. Por lo tanto, podemos concluir que Ω-Test supera a TBDR sin necesidad de una comparación directa.

Otro aspecto que diferencia a Ω-Test del esquema *Deferred Rendering* es que mientras que éste último requiere una pasada completa sobre la geometría en la segunda pasada, Ω-Test es más selectivo con su fase de corrección (similar a la segunda pasada de *Deferred Rendering*), en la que únicamente se procesan aquellos fragmentos cuya visibilidad ha sido mal predicha. Es decir, sólo en el peor escenario posible, en el que Ω-Test fallase todas las predicciones de visibilidad, lo que supondría que todos los fragmentos visibles del *tile* tendrían que pasar por la fase de corrección, Ω-Test se comportaría exactamente igual que un *Deferred Rendering*, puesto que en la fase de *rendering* normal (equivalente a la primera pasada de *Deferred Rendering*) se resolvería la visibilidad de todos los fragmentos actualizando el Z-Buffer acordemente, mientras que en la fase de corrección (equivalente a la segunda pasada de *Deferred Rendering*) se realizaría el renderizado de los fragmentos erróneos. Tal y como se menciona en la Sección 4.3, sólo unos pocos fragmentos de media necesitan pasar a la fase de corrección, por lo que los sobrecostes de esta segunda pasada son muy inferiores a los generados por un esquema basado en *Deferred Rendering* como TBDR. Por lo tanto, Ω-Test es mejor que estos esquemas.

4.4.2. Técnicas basadas en *HiZ occlusion culling*

Tal y como se ve en la Sección 2.2.1, las GPUs basadas en IMR generan un tráfico a memoria considerable en tanto que el Z-Buffer y el Color Buffer están almacenados en DRAM. Para paliar este efecto, se han propuesto en la literatura una serie de técnicas denominadas *HiZ occlusion culling* (también conocidas como *Hierarchical Z occlusion culling*) [37, 36, 87, 60, 6] para eliminar primitivas ocluidas en una región del *framebuffer* a un grano más grueso que usando un Z-Buffer convencional. Dicha técnica se implementa introduciendo una etapa de *culling* justo antes de la conocida etapa Early Z-Test, pero ésta en vez de funcionar a nivel de fragmento funciona a nivel de primitiva. Por lo tanto, el *culling* de HiZ evita accesos a memoria provocados por el Z-Buffer, determinando la visibilidad de "trozos" de primitiva en lugar de fragmentos. Para implementar dicha técnica, se necesita una versión de grano grueso del Z-Buffer, que se genera sobre la marcha, a medida que se van procesando fragmentos. Este *buffer* se denomina *Hierarchical Z-Buffer*, y se computa de la siguiente manera. Primero se subdivide el Z-Buffer en regiones del mismo tamaño, y por cada región se almacena un par de valores (Z_{min}, Z_{max}), donde el primero indica la profundidad más cercana de esa región y el segundo la más lejana. Para realizar el *culling* de las primitivas, se tiene que crear una lista de primitivas que solapa cada región. De este modo, cuando se evalúa una primitiva en una región concreta se comprueba si todos sus vértices tienen una profundidad superior a Z_{max}), y si es así, la primitiva se puede eliminar de manera segura puesto que no será visible. Si todos los vértices de la primitiva tienen una profundidad inferior a $Z - Max$, entonces la primitiva pasa directamente a la etapa de renderizado. Si la profundidad de alguno de los vértices de la primitiva a evaluar está en el rango $[Z_{min}, Z_{max}]$, entonces se escala un nivel en el *Hiearchical Z-Buffer* (un nivel de grano más fino) y se vuelven a evaluar las subregiones resultantes, de ahí el sobrenombre de *jerárquico*.

La primera implementación conocida de HiZ se propuso en [37, 36] como una solución puramente *software*. Esta propuesta usaba estructuras complejas tales como *oct-trees* para determinar de manera eficiente la visibilidad de objetos completos. En [87] se propone una implementación HiZ con una jerarquía completa. En

[60] se propone una implementación HiZ mucho más sencilla y eficiente en la que sólo se usa un único nivel de jerarquía. En cualquier caso, es importante destacar que HiZ no reemplaza a la etapa Early Z-Test, pues el objetivo de estas técnicas es reducir el tráfico a memoria DRAM por medio de la eliminación de primitivas ocluidas de una manera más temprana, siendo imprescindible una etapa que resuelva el problema de la visibilidad a *nivel de fragmento*. También es importante destacar que las estructuras de HiZ se construyen por fotograma, por lo que no es capaz de eliminar más *overdraw* que Ω-Test si las primitivas llegan en orden de atrás hacia delante. De hecho, HiZ no elimina más *overdraw* que un Early Z-Test tradicional puesto que no es ése su objetivo. Por el contrario, Ω-Test es capaz de reducir mucho más *overdraw* aprovechando la información del Z-Buffer del fotograma previo para determinar de manera especulativa la visibilidad de los fragmentos del fotograma actual.

4.4.3. Otros trabajos relacionados que aprovechan la coherencia entre fotogramas

Existen trabajos más recientes en la literatura que aprovechan la coherencia entre fotogramas para optimizar el problema de la visibilidad y eliminar *overdraw* en una escena. Visibility Rendering Order (VRO) [23], como ya se ha dicho anteriormente, es una técnica que ordena los objetos de la escena en orden de delante hacia atrás, de modo que el Z-Buffer se actualiza antes, dando la oportunidad a la etapa Early Z-Test de eliminar más fragmentos.

En [7] se presenta Early Visibility Resolution (EVR), una técnica que computa el punto más lejano de cada *tile* en un fotograma para predecir primitivas ocluidas en el fotograma siguiente, con el objetivo de renderizar las primitivas que se suponen ocluidas al final del *pipeline* gráfico, es decir, renderizarlas las últimas, de modo que la etapa Early Z-Test para esas primitivas tenga un Z-Buffer más actualizado y capaz de eliminar más *overdraw*. Tanto VRO como EVR usan información del fotograma previo para reordenar los objetos (en VRO) o las primitivas (en EVR) en el fotograma actual para aumentar la efectividad de la etapa Early Z-Test. Mientras que VRO optimiza el problema de la visibilidad a *nivel de objeto*, EVR lo optimiza

trabajando a *nivel de primitiva*. De manera diferente, Ω-Test optimiza el problema de la visibilidad a *nivel de fragmento*, siendo así capaz no sólo de eliminar *overdraw* intra-objeto, sino también el *overdraw* intra-objeto y, por lo tanto, superando a VRO como se pudo observar en la Figura 4.9.

4.5. Conclusiones

Tal y como se ha podido observar en este capítulo, el rol que juega la etapa Early Z-Test es crucial, pues es el paso previo a la operación computacional y energéticamente más costosa de todo el *pipeline* gráfico, que es la llevada a cabo por los *Fragment Processors*. Por lo tanto, es importante que esta etapa tenga la capacidad de descartar fragmentos que finalmente no van a ser visibles en la pantalla. Es por eso que Ω-Test usa el Z-Buffer del fotograma previo en vez de partir de cero. Como se ha visto, este Z-Buffer (Ω-Buffer) está en un estado mucho más maduro para eliminar fragmentos de manera mucho más temprana. Si bien es cierto esta información es imprecisa y, por lo tanto, puede llevar a errores en la imagen final, Ω-Test precisa de un sofisticado y eficiente mecanismo para corregirlos de modo que no afecte en absoluto a la imagen resultante.

5

Triangle Dropping

Las arquitecturas TBR dividen el proceso de renderizado de una escena en tres fases esenciales, tal y como se explicó previamente (Sección 2.2.2), a saber: procesamiento de la geometría (llevada a cabo por el Geometry Pipeline), el renderizado de fragmentos (llevado a cabo por el Raster Pipeline) y el Tiling Engine como mediador entre estas dos fases para generar los denominados *tiles*. Esto presenta un problema que no aparece en las arquitecturas IMR, y es que al estar completamente acopladas las fases Geometry Pipeline y Raster Pipeline[1], la geometría que se transmite de una fase a otra tiene que almacenarse en el Parameter Buffer (véase Figura 2.17), el cual está alojado en memoria DRAM.

Tal y como se ha visto en la introducción (Sección 1.1.2), el procesamiento de la geometría ocluida en una escena no es despreciable (véase Figura 1.3), lo que

[1]Esto quiere decir que la fase Geometry Pipeline tiene que ejecutarse completamente para que empiece a ejecutarse la fase Raster Pipeline.

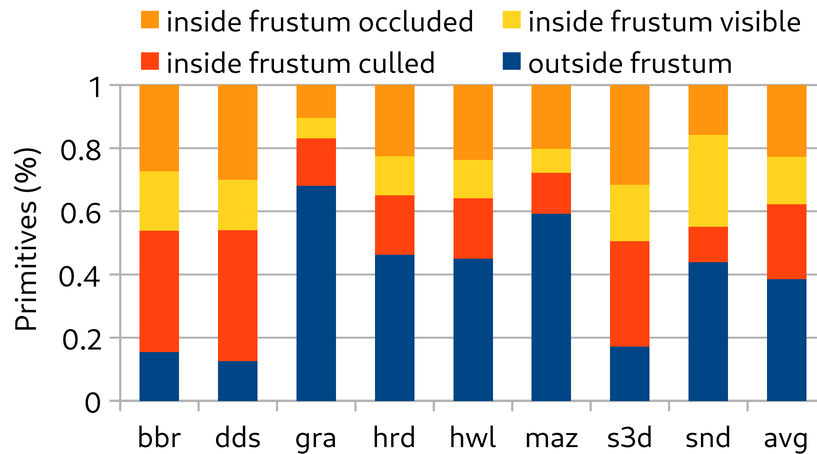


Figura 5.1: Caracterización de la geometría de una escena, desglosada en primitivas que caen dentro del *frustum* de la cámara (tanto visibles como ocluidas) y primitivas que caen fuera de éste.

supone una drástica degradación del rendimiento de la GPU.

Para mostrar de manera más detallada este problema, la Figura 5.1 muestra un desglose de la geometría medida en cantidad de primitivas de cada escena evaluada, y clasificada en geometría que cae dentro del *frustum* de la cámara y la que cae fuera de éste. Las primitivas que caen dentro del *frustum* están a su vez desglosadas en aquéllas que muestran su cara trasera (barra roja, *inside frustum culled*), aquéllas que muestran alguna parte visible en la imagen final (barra amarilla, *inside frustum visible*), y aquéllas que están ocultas por otras, es decir, que no muestran ninguna parte visible en la imagen final (barra naranja, *inside frustum occluded*). Aquí se puede observar que, de media, las primitivas que se procesan para pasar al proceso de sombreado suponen el 37.7 % de la geometría (barras amarilla y naranja), de las cuales el 60.1 % están completamente ocluidas, suponiendo así un malgasto de recursos (tanto de almacenamiento como de cómputo).

Para hacer frente a este problema, en este Capítulo presentamos Triangle Dropping, una técnica hardware capaz de predecir la visibilidad de las primitivas en un fotograma aprovechando la visibilidad de las primitivas en el fotograma previo. Como ya se ha mencionado en el Capítulo 1, una gran parte de las primitivas presentes en una escena se encuentra completamente ocluida, es decir, ninguna de las

partes de dichas primitivas son finalmente visibles en la pantalla, por lo que no tienen impacto alguno en ésta. Esto provoca escrituras innecesarias en el Parameter Buffer, la estructura encargada de almacenar la geometría de la escena organizada en *tiles*, de las cuales muchas acaban yendo a memoria DRAM. Por lo tanto, la eliminación prematura de estas primitivas supondrá una ganancia en rendimiento y una reducción de energía.

Triangle Dropping reduce accesos al Parameter Buffer por medio de la eliminación de la geometría ocluida, basándose en la información de visibilidad calculada en el fotograma previo. Por lo tanto, Triangle Dropping permite descartar de una manera más prematura este tipo de geometría, es decir, directamente en el Geometry Pipeline en vez de hacerlo en mitad del Raster Pipeline como es habitual en técnicas basadas en Z-Test que trabajan a nivel de fragmento en lugar de a nivel de primitiva. Por otro lado, si la arquitectura subyacente de la GPU es de tipo TBR, esta técnica tiene como ventaja adicional la reducción del *overdraw* generado por esas primitivas ocluidas. Sin embargo, en la evaluación realizada en este Capítulo se ha optado por utilizar TBDR como arquitectura base que, como se explicó en la Sección 2.3.1, es capaz de reducir absolutamente todo el *overdraw* presente en una escena. Por lo tanto, los beneficios que obtiene Triangle Dropping y que se presentan en los resultados de este Capítulo se deben a ahorros en el Geometry Pipeline y en el Tiling Engine, y no a la reducción del *overdraw* como tal.

Para lograr el descarte prematuro de primitivas que están completamente ocluidas, Triangle Dropping se apoya en la gran cantidad de coherencia entre fotogramas que presentan las cargas gráficas. El Raster Pipeline genera información muy valiosa acerca de la visibilidad final de una escena después de ser renderizada; información que en lugar de ser desechada se puede usar para predecir las primitivas que serán visibles en el siguiente fotograma. Sin embargo, como Triangle Dropping se basa en predecir la visibilidad a nivel de primitiva, debe ser muy conservadora a la hora de eliminar geometría para no inducir errores perceptibles en la imagen final.

Triangle Dropping opera a nivel de primitiva, haciendo un seguimiento de aquellas que resultan estar ocluidas en el fotograma actual para así descartarlas

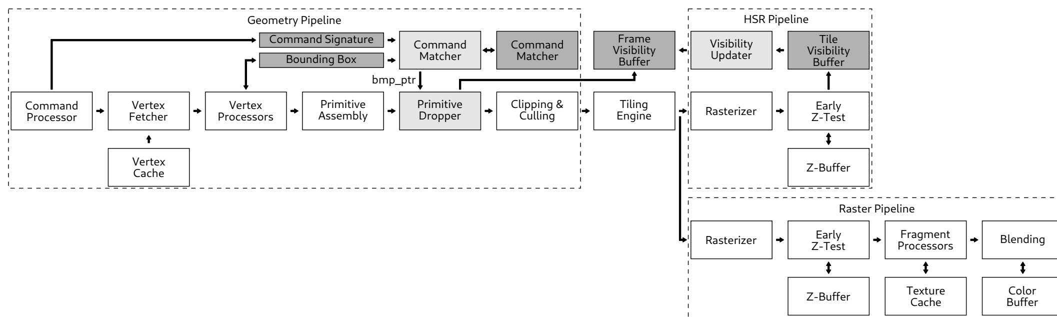


Figura 5.2: Implementación de Triangle Dropping en una arquitectura TBDR. Las unidades adicionales aparecen sombreadas.

en el siguiente fotograma. La Figura 5.2 muestra un esquema de esta técnica implementada en una arquitectura TBDR. A cada comando OpenGL (nótese que un comando OpenGL representa un objeto dentro de la escena) se le asocia un *visibility bitmap* (mapa de visibilidad), que no es más que un vector de *bits*, en el que cada *bit* está asociado a una primitiva de dicho comando y su valor indica si la primitiva es o no visible. Así que antes de procesar una primitiva, primero se consulta su visibilidad en este mapa de bits. Si el *bit* de visibilidad está activo, se asume que la primitiva va a ser visible en el *frame* actual, dado que en el *frame* anterior también lo fue y, por lo tanto, la primitiva se procesa en el Geometry Pipeline como se haría en la arquitectura base. En caso de que el *bit* de visibilidad esté desactivado, la primitiva se descarta y no se procesa en el Geometry Pipeline, evitando así cualquier escritura ocasionada en el Parameter Buffer y, por supuesto, cualquier otra actividad de rasterización que se pudiera derivar de haber procesado dicha primitiva. El identificador de primitiva depende exclusivamente del orden en el que fue emitida dentro del comando OpenGL, es decir, con la *draw call*. Por lo que es importante que este orden no se vea alterado de un *frame* a otro. De lo contrario, la información de visibilidad sería incorrecta en tanto que no se corresponderían los *bits* de visibilidad del *visibility bitmap* con las primitivas. Sin embargo, nótese que no importa el orden en el que los comandos OpenGL son emitidos. Un comando OpenGL que tiene asociado un Geometry Shader o una teselación sufre alteraciones en su geometría en tanto que está alterando su número de primitivas. Tal situación provocaría este desorden. Para solventarlo, Triangle Dropping es capaz de comunicarse con el *driver* de OpenGL para detectar que un comando OpenGL cumple estas características,

y entonces deshabilitar la técnica para evitar resultados incorrectos. Esto supone, obviamente, una pérdida de potencial pero no altera la calidad de la imagen, que es una de las principales prioridades. Recordemos que el descarte de la geometría en una fase tan temprana como el Geometry Pipeline supone no poder saber si se ha producido un error y, por lo tanto, la imposibilidad de recuperarse del mismo. Dicho de otro modo, no existe forma factible de recuperarse ante los errores provocados por primitivas descartadas indebidamente en el Geometry Pipeline, por lo que es crucial ser conservadores a la hora de descartar primitivas.

5.1. Identificación de comandos OpenGL entre fotogramas

Las GPUs convencionales no implementan un método *hardware* para identificar a un mismo comando OpenGL a lo largo de múltiples fotogramas consecutivos en tanto que son tratados como entidades completamente independientes de un fotograma al siguiente. Obviamente, a nivel de aplicación existen estos identificadores, pero no son pasados explícitamente a la GPU. Como Triangle Dropping se apoya en la coherencia entre fotogramas, se necesita un mecanismo que permita identificar comandos entre dos fotogramas consecutivos para poder acceder al mapa de visibilidad correspondiente y así ser capaz de eliminar sus primitivas acordemente. Una simple solución basada en el codiseño *hardware-software* podría ser usar el *driver* de OpenGL para proporcionar estos identificadores de comando que fueron previamente provistos por la aplicación en sí.

Sin embargo, como no podemos asumir que ni las aplicaciones evaluadas ni el *driver* de OpenGL proporcionan dichos identificadores de comando, en esta Tesis hemos implementado un mecanismo transparente para el programador y completamente basado en *hardware* para identificar un mismo comando que se utiliza a lo largo de múltiples fotogramas. Para lograrlo se ha propuesto el uso de dos tipos de información: 1) estática, que se obtiene de la máquina de estados de OpenGL (en forma de firma de comando); y 2) dinámica, basada en la *bounding box*[2] del objeto

[2]Se denomina *bounding box*, más propiamente conocida como *axis-aligned bounding box* (AABB), al tetraedro que define los límites de un modelo en el espacio. O dicho de otro modo, la *bounding box* de un modelo se obtiene calculando el máximo y el mínimo de las tres dimensiones de cada vértice que lo compone.

(es decir, sus coordenadas de pantalla en el fotograma actual).

En el primer caso, se calcula una firma CRC de 64 *bits* usando información estática del comando OpenGL, y se realiza cuando el Command Processor emite el comando. Esta información es estática en tanto que se refiere a propiedades que se mantienen constantes a lo largo de toda la vida del comando en la secuencia de imágenes. En concreto, se han usado los siguientes parámetros constantes:

- Información del *render target* en el que se va a renderizar, en concreto: direcciones de memoria así como las dimensiones del Color Buffer y del Z-Buffer.
- Número de vértices y primitivas del comando. Generalmente, es una información que se mantiene constante a no ser que se ejecute un *geometry shader*, en cuyo caso la técnica se deshabilita para no generar errores perceptibles.
- Información del Z-Test, a saber: si está habilitado o no (*enabled bit*), si modifica el Z-Buffer o no (*write mask bit*) y la función del test de profundidad (*depth function*).
- Información sobre el *blending*, a saber: bit de habilitado (*enabled bit*), función lógica de *blending* (si se aplica), color RGB, canal *alpha* para transparencias y la máscara de color.
- Información sobre los *vertex shaders* y los *fragment shaders*, a saber: punto de entrada, número de *attribute locations*, de entradas y de salidas. Es importante no tener en cuenta el código en sí, puesto que los comandos pueden cambiar de *shader* de un fotograma a otro para dotar de un efecto especial diferente al comando. Es por eso que sólo se tienen en cuenta las entradas y las salidas, que deberían coincidir en tanto que el número de atributos de entrada y de salida no van a variar puesto que se trata del mismo objeto.
- Tipo de ensamblado de primitiva.

Esta firma se almacena entonces en un registro denominado *Command Signature*.

Por otro lado, la información dinámica para identificar un mismo comando (objeto) a lo largo de varios fotogramas consecutivos consiste en utilizar una *bounding box*, que es calculada por los *Vertex Processors* mientras procesan los vértices del comando. Dicha información se almacena en un registro denominado *Bounding Box*. Este registro consta de dos puntos tridimiensionales: uno para el máximo y otro para el mínimo, y cada punto ocupa un total de 3 números de punto flotante de simple precisión, lo que da un total de 24 bytes para el registro Bounding Box.

Toda esta información se entrega al Command Matcher, que se encarga de determinar si un objeto en una escena (un comando en terminología OpenGL) en el fotograma actual se corresponde con él mismo en el fotograma previo, con el fin de usar su información de visibilidad previa. A este proceso se le denomina *matching* de comandos OpenGL, dando un resultado positivo cuando el test es superado. Para que esto suceda se tienen que cumplir dos condiciones. La primera es que las firmas CRC que contienen la información estática de los dos comandos han de ser idénticas (*bit* a *bit*). La segunda condición es que las *bounding boxes* de ambos comandos OpenGL tienen que ser "similares", aunque no necesariamente idénticas. Esto significa que se permite un pequeño margen de movimiento denominado *delta margin*, dado que los objetos (o incluso la cámara) pueden moverse ligeramente por la pantalla de un fotograma al siguiente. Los detalles de implementación del Command Matcher se explican en la Sección 5.5

5.2. Comprobación de la visibilidad de un Triángulo

La Figura 5.2 también muestra el Primitive Dropper justo después de la etapa Primitive Assembly. Esta unidad accede al *visibility bitmap* para comprobar el *bit* de visibilidad de la primitiva y así tomar la decisión de descartar o no la primitiva en cuestión. Cuando un objeto del fotograma actual hace *matching* con otro que se encuentra en el Command Buffer (véase Sección 5.5), se activa el *bit* de recientemente usado (*recently used bit*) del comando dentro del Command Buffer y se envía el puntero *bitmap pointer* a la unidad Primitive Dropper. Un comando OpenGL (objeto de la escena) con el *recently used bit* activado no puede hacer *matching* con otro comando. Esto se hace así para evitar efectos de interferencias. Por otro lado, si

no se ha encontrado ningún comando coincidente, dicho comando se inserta en el Command Buffer (siempre y cuando haya espacio libre) en tanto que es un nuevo comando que no había en el fotograma previo. Además, se activa el *recently used bit*, evitando así que dos comandos diferentes dentro del mismo fotograma hagan *matching* erróneamente, pues es común que haya objetos repetidos dentro de la misma escena. Para este nuevo comando insertado, se reserva entonces un *visibility bitmap* en la estructura Frame Visibility Buffer, con tantos *bits* como primitivas tenga el comando. El *vibility bitmap* se inicializa con todos los elementos a cero, indicando que todas las primitivas del comando están ocluidas, y que la visibilidad actual de dichas primitivas será actualizada justo después de concluir su fase de rasterización.

Calcular la *bounding box* completa de un comando OpenGL requiere que todos sus vértices hayan sido procesados por los *Vertex Processors*. Si el comando tiene muchas primitivas, este cálculo podría incurrir en un retardo considerable en el Geometry Pipeline debido a que el Command Matcher estaría esperando a tener disponible el registro Bounding Box, bloqueando por consecuencia al Primitive Dropper que estaría esperando, a su vez, el *bitmap pointer* para acceder a la información de visibilidad. Además, podría suceder una situación de interbloqueo (*deadlock*) si la cola de salida de los *Vertex Processors* se llena durante este período de espera. Para evitar estos dos inconvenientes, en vez de calcular la *bounding box* completa del objeto, se calcula una *bounding box* parcial considerando sólo los primeros `n` vértices del comando. Esto podría provocar una pérdida de precisión en el mecanismo de *matching* de las *bounding boxes* y, por lo tanto, podría resultar en una pérdida de potencial. Para evitar dicha situación, se ha realizado un estudio que cuantifica los vértices de media necesarios que ha de tener una *bounding box* parcial para que ofrezca resultados similares a los de una *bounding box* completa.

A pesar de que este mecanismo reduce la probabilidad de interbloqueo, no la previene del todo. Para evitar por completo la posibilidad de *deadlock* es necesario que la cola de salida de los *Vertex Processors* pueda almacenar, al menos, los vértices de una *bounding-box* parcial, es decir, 18 quad-vértices.

Por otro lado, también se ha comprobado que esta cantidad de vértices no supone ninguna sobrecarga en cuanto a tiempo. Así pues, este estudio se ha comple-

mentado analizando la cantidad media de quad-vértices que hay entre los *Vertex Processors* y el Primitive Dropper. Si dicha cantidad es inferior a los de una *bounding box* parcial, entonces no se genera ningún bloqueo dado que el Primitive Dropper tendrá la *bounding box* disponible antes de procesar la primera primitiva. De lo contrario, el Primitive Dropper recibiría la primera primitiva sin tener aún la *bounding box*, lo que provocaría un retardo. Este estudio muestra que en promedio hay 33 quad-vértices entre los *Vertex Processors* y el Primitive Dropper, una cantidad muy inferior a la requerida para computar una *bounding box* parcial. Por lo tanto, no se genera ninguna sobrecarga de tiempo debido al cómputo de la *bounding box* parcial.

El Frame Visibility Buffer es una estructura global que almacena la información de la visibilidad de todos los comandos del fotograma previo. Contiene los *visibility bitmaps* de todos los comandos. El Primitive Dropper accede a esta tabla mediante un sencillo cálculo de índice, en el que se usa el *bitmap pointer* como una dirección base más el identificador de la primitiva dentro del comando (recalcar que los identificadores de primitiva dentro del mismo comando no cambian de orden de emisión de un fotograma para otro, a no ser que sean procesados por un Geometry Shader).

Respecto a las primitivas *back-faced*, es decir, las que muestran su cara trasera, y las que se encuentran fuera del *frustum*, nuestra técnica Triangle-Dropping las marca como visibles en el Frame Visibility Buffer, pues ya se descartan de manera natural en una arquitectura TBDR base, y no se escriben ni siquiera en el Parameter Buffer. Lo que se logra con ésto es evitar que este tipo de primitivas provoque errores, puesto que si en el siguiente fotograma aparecen en escena, no son descartadas. Con este mecanismo se pierde un poco de potencial (sólo un poco en los bordes de la pantalla), pero se evitan errores que impacten la calidad de la imagen.

5.3. Cómputo de la visibilidad durante el Raster Pipeline

La visibilidad final de las primitivas de una escena se conoce una vez se ha rasterizado por completo la imagen, es decir, la visibilidad se determina durante el Raster Pipeline. El Tile Fetcher empieza leyendo atributos del Parameter Buffer,

leyendo también el *visibility pointer* de cada primitiva que se manda al Rasterizer. Cuando el Rasterizer genera un *quad-fragment*, le adjunta también el *visibility pointer* como si fuese un atributo más. En la etapa Early Z-Test se encuentra un *buffer* on-chip adicional llamado Tile Visibility Buffer. Este *buffer* tiene las mismas dimensiones que el Z-Buffer, es decir, un *tile* completo. Su función es la misma que la del Z-Buffer, pero en vez de almacenar la profundidad del fragmento más cercano a la cámara, almacena el *visibility pointer* de la primitiva más cercana a la cámara hasta el momento. Este *buffer* se inicializa con punteros nulos, indicando que ninguno de los fragmentos del *tile* son visibles. Dado que este *buffer* está limitado a un único puntero por posición, sólo podemos mantener la información de visibilidad de las primitivas opacas, pues las transparentes podrían dejar visible más de un fragmento en esa posición (debido al *blending*). Por lo tanto, Triangle Dropping no considera las primitivas transparentes, y las deja pasar por el Geometry Pipeline como en la arquitectura base. Así que estas primitivas no escriben en el Tile Visibility Buffer. Esto es análogo a lo que pasa con el Z-Buffer y las primitivas transparentes, pues generalmente estas primitivas tienen deshabilitada la opción de escribir en el Z-Buffer. La información de visibilidad de un *tile* se sabe por completo una vez se procesa el último *quad-fragment* en el Early Z-Test. En ese momento, el Visibility Updater se activa. Esta unidad itera sobre todos los punteros del Tile Visibility Buffer. Si el puntero es nulo, no se hace nada. Si el puntero no es nulo, entonces accede al *bit* de dicho puntero en el Frame Visibility Buffer y lo establece a uno, indicando que esa primitiva ha sido visible finalmente. Nótese la importancia de haber inicializado el *visibility bitmap* con ceros, pues por defecto una primitiva es invisible hasta que se encuentra en, al menos, un *tile*. Una vez el Visibility Updater ha concluido, la visibilidad de las primitivas de cada comando está lista para ser usada en el siguiente fotograma. Es importante destacar que el Visibility Updater funciona en paralelo con el resto del *pipeline* (es decir, los *Fragment Processors* y la Blending Unit) en tanto que está implementado después de las etapas de HSR (Hidden Surface Removal) y, por lo tanto, no se incurre en ninguna penalización de tiempo.

5.4. Intervalo de refresco

Cuando una primitiva se marca como ocluida en el Frame Visibility Buffer, se descarta para los fotogramas siguientes, y no se volvería a procesar nunca más en el Geometry Pipeline. Sin embargo, si dicha primitiva se vuelve visible en un fotograma posterior, debe restaurarse su *visibility bit* a uno. Una manera simple de lograrlo es deshabilitar la técnica durante un único fotograma cada cierto número de fotogramas consecutivos. Este fotograma, durante el cual la técnica queda deshabilitada, se denomina *key frame*, y es el que restablece la información de visibilidad de la escena, evitando así propagar errores en sucesivos fotogramas. Este comportamiento se controla mediante un parámetro denominado intervalo de refresco (*refreshing interval*). Nótese que cuanto más corto es el intervalo de refresco menos errores potenciales se van a producir, pero también se pierde potencial para Triangle Dropping. Para ofrecer el mejor resultado, Triangle Dropping es capaz de ajustar dinámicamente el intervalo de refresco, consiguiendo así un buen equilibrio entre calidad de imagen y potencial exprimido. En la Sección 5.7 se encuentran más detalles sobre este mecanismo dinámico.

5.5. El Command Matcher

El Command Matcher es la unidad encargada de realizar el proceso de *matching* entre comandos OpenGL de un fotograma a otro, descrito en la Sección 5.1. La Figura 5.3 muestra la estructura interna del Command Matcher y su implementación en *hardware*.

Esta unidad funcional usa dos tablas que se almacenan en memorias *on-chip*, a saber: la *Main Table* y el *Overflow Buffer*. Al conjunto de estas dos tablas se le denomina Command Buffer. Cada una de estas tablas está formada por un número determinado de entradas con un formato específico, que denominamos *sets*. Un *set* está compuesto por un número de vías (llamadas *slots*) y un *overflow pointer*, que es un puntero que apunta a una entrada del Overflow Buffer. Cada *slot* almacena información particular de un comando, a saber: Command Signature, Bounding Box,

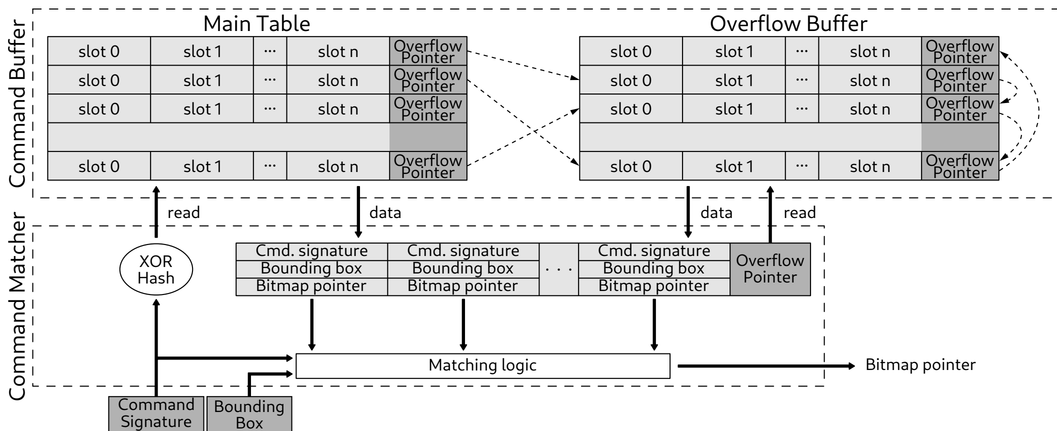


Figura 5.3: Esquema de la implementación *hardware* y la lógica del Command Matcher y el Command Buffer.

valid bit, *recently used bit* y *bitmap pointer* La Main Table se indexa mediante el Command Signature, al que previamente se le ha aplicado una función *hash* XOR para ajustarse al número de entradas de dicha tabla. Una vez se ha accedido a la Main Table, la entrada consultada se almacena en un registro interno. Cada uno de los *slots* de dicho registro es comparado con el Command Signature y la Bounding Box de entrada. La función de comparación empleada es la descrita en la Sección 5.1. Si hace *matching*, entonces la Bounding Box del *slot* correspondiente es actualizada a la de entrada, y entonces se devuelve como resultado el *bitmap pointer* almacenado en dicho *slot*, que lo usará el Primitive Dropper para acceder a la información de visibilidad de las primitivas (el *visibility bitmap*) almacenada en la estructura Frame Visibility Buffer. Si no hace *matching*, se consulta el Overflow Pointer del registro intermedio y, si no es nulo, se usa para acceder a la entrada correspondiente del Overflow Buffer. La entrada consultada del Overflow Buffer es almacenada en el registro intermedio, y se vuelve a realizar el proceso de *matching*, que se detendrá o bien cuando encuentre un *slot* con el que haga *matching*, o bien cuando el Overflow Pointer del registro intermedio sea nulo, en cuyo caso se inserta el comando siempre y cuando haya espacio disponible en el Command Buffer. Nótese que el Overflow Buffer es una estructura que almacena listas simplemente enlazadas, y que la Main Table es accedida como un mapa. Esto nos permite tener una memoria de búsqueda asociativa muy rápida como método principal de búsqueda de un

comando; y una memoria de búsqueda iterativa más lenta como *backup*, pero que no es tan costosa en *hardware* (el Overflow Buffer). Así pues, se logra un equilibrio entre velocidad de acceso y coste de almacenamiento.

La inserción de un comando en el Command Buffer funciona de la siguiente manera. Primero, se comprueba si hay un *slot* libre en el conjunto que le corresponde (computado previamente con la función *hash* XOR). Si es así, entonces se escriben en dicho *slot* tanto el Command Signature como la Bounding Box de entrada, y se activan los *bits valid* y *recently used*. Cuando se inserta un nuevo comando en el Command Buffer, también se reserva un *visibility bitmap* en el Frame Visibility Buffer, siempre y cuando haya espacio suficiente para dicho vector, y se devuelve el puntero *bitmap pointer* al Command Buffer para que se escriba también esa información en el *slot*. Si todos los *slots* del *set* están ocupados, entonces se sigue el *overflow pointer* del conjunto, que apunta a un conjunto del Overflow Buffer. Aquí se vuelve a buscar un *slot* libre. Si el conjunto está lleno, se sigue de nuevo el *overflow pointer* del conjunto, que apunta a otro conjunto en forma de lista enlazada (tal y como puede observarse en la Figura 5.3). La búsqueda continúa hasta encontrar un *slot* libre. De manera alternativa, si un conjunto está lleno pero su *overflow pointer* es nulo, entonces se reserva un nuevo conjunto en el Overflow Buffer y se enlaza al conjunto previo. Finalmente, si el Overflow Buffer está completamente lleno, el comando no se podrá insertar en el Command Buffer y, por lo tanto, se devolverá un puntero nulo al Primitive Dropper. Esta situación indica que el comando no fue encontrado en el Command Buffer y, por lo tanto, se trata como si fuera un comando visible, provocando una pequeña pérdida de potencial. Nótese que puede suceder que el Overflow Buffer se llene, pero no así la Main Table, debido a que es una tabla de búsqueda e inserción asociativa. Cuando se termina de procesar el fotograma, el *bit recently used* de todos los comandos se pone a cero, y aquellos comandos que tuvieran este *bit* ya a cero son eliminados del Command Buffer en tanto que se asume que han desaparecido de la escena.

La reserva de un *visibility bitmap* en el Frame Visibility Buffer se hace de la siguiente forma. El Command Matcher tiene un registro interno inicializado a cero, que se usa para generar los *bitmap pointers*. Este registro se incrementa cada vez que un comando se inserta. El valor con el que se incrementa es el número de primi-

tivas del comando que se ha insertado. Cuando este registro alcanza la capacidad máxima del Frame Visibility Buffer, el comando no se puede insertar porque ya no queda espacio libre. Por lo tanto, se devuelve un puntero nulo al Primitive Dropper.

Para dimensionar el Command Buffer de manera apropiada, se han realizado diferentes experimentos. De todos los *benchmarks* evaluados, el que más comandos OpenGL tiene en un fotograma concreto es *bbr*, con hasta 312 comandos. Esto supone un total de 12.56 KiB por tabla en el Command Buffer. De manera similar, para estimar la capacidad del Frame Visibility Buffer se ha medido el número máximo de primitivas por *benchmark* dentro de un fotograma. Este experimento nos muestra un máximo de 190,449 primitivas por fotograma, lo que se traduce en un Frame Visibility Buffer de 23.25 KiB. Estos tamaños de memoria *on-chip* son bastante razonables en una GPU móvil moderna. Por lo tanto, para esta evaluación se han redondeado estos números a 16+16 KiB para el Command Buffer y 32 KiB para el Frame Visibility Buffer. Sin embargo, si un *benchmark* excede estos tamaños en un fotograma en concreto, lo único que sucede es una pequeña pérdida de potencial para Triangle Dropping, que según nuestros experimentos escogiendo tamaños más pequeños es casi despreciable.

5.6. Manejo de primitivas con visibilidad intermitente

Hay ciertos comandos que, o bien por la naturaleza de sus movimientos o bien por el entorno que lo rodea, tienen primitivas cuya visibilidad cambia continuamente en un rango determinado de fotogramas. Es el caso, por ejemplo, de un objeto rotando que es tapado parcialmente por otro objeto estático. En este caso, las primitivas que estaban ocluidas serán visibles en un momento determinado para luego volver a ser ocluidas por dicho objeto estático, debido al movimiento de rotación. De esta forma, si Triangle Dropping marcase estas primitivas cuya visibilidad es intermitente como invisibles, haría una predicción incorrecta de su visibilidad cuando éstas volviesen a aparecer en la escena, dando lugar a un error en la imagen final y, por tanto, a un decremento en la calidad de imagen. Este artefacto de visibilidad podría ser incluso peor en caso de producirse durante un *key frame*, pues las primitivas descartadas erróneamente se harían visibles de repente, dando como

resultado un impacto visual muy negativo en cuanto a calidad de imagen. En adelante, nos referiremos a este tipo de primitivas que cambian su visibilidad a lo largo del tiempo como *primitivas intermitentes*.

Para solucionar este problema que potencialmente pueden provocar las primitivas intermitentes y no causar errores perceptibles en la imagen final, éstas han de ser detectadas de cara a que Triangle Dropping las ignore, quedando así descartadas. Para lograrlo, Triangle Dropping usa la siguiente heurística. Si se detecta que una primitiva cambia su estado de visibilidad de invisible a visible de un fotograma para otro, entonces esta primitiva se marca como intermitente en tanto que ya ha provocado un error perceptible en la imagen, con lo que ya no se le da otra oportunidad para predecir su visibilidad y continuar generando errores. Como Triangle Dropping no actualiza la visibilidad real hasta llegar a un *key frame*, el Visibility Updater realiza esta tarea durante un *key frame*. Antes de sobrescribir el estado de visibilidad de la primitiva en el Frame Visibility Buffer, se comprueba su última visibilidad. Si esta propiedad ha cambiado de invisible a visible, entonces la primitiva es marcada de manera conservadora como intermitente. Esta propiedad se almacena en el *visibility bitmap* con dos bits extra por entrada: *intermittent bit* y *previous visibility bit*. Por lo tanto, cuando el Primitive Dropper consulta la visibilidad de una primitiva, mira primero el *intermittent bit*. Si está activado, entonces la primitiva es intermitente y, por lo tanto, tiene que ser procesada por el Geometry Pipeline como se haría de forma convencional, independientemente de su último estado de visibilidad.

5.7. Mecanismo de intervalo de refresco dinámico

Para poder detectar adecuadamente las primitivas intermitentes de una escena se necesita un período de calentamiento que puede durar varios fotogramas puesto que éstas solo pueden ser detectadas durante un *key frame* (que ocurre cada n fotogramas). Triangle Dropping incorpora un mecanismo de mitigación cuyo objetivo es minimizar la duración de este período de calentamiento usando un intervalo de refresco dinámico para capturar de una manera más prematura primitivas intermitentes. Este mecanismo establece inicialmente el intervalo a un valor mínimo, que

es 2 (es decir, uno de cada dos fotogramas es un *key frame*). En cada *key frame* se determina si han entrado en escena comandos nuevos desde el último *key frame*. Si no hay comandos nuevos, se incrementa en una unidad el intervalo de refresco. En el caso de que se haya introducido al menos un comando en la escena, entonces el intervalo de refresco vuelve a su valor inicial mínimo de 2. El valor máximo que puede alcanzar el intervalo de refresco se ha determinado de manera experimental. En concreto, se ha observado que un valor máximo de 5 fotogramas para el intervalo de refresco resulta en una pérdida despreciable de potencial, mientras que la calidad de la imagen se mantiene casi perfecta.

A pesar de que esta técnica dinámica podría provocar una pérdida de potencial, a largo plazo este efecto es despreciable puesto que los objetos tienden a permanecer en la escena durante un período largo de fotogramas, por lo que esta información de visibilidad intermitente generada se aprovecha durante mucho tiempo.

5.8. Resultados

La Tabla 5.1 muestra los parámetros de simulación usados a la hora de evaluar Triangle Dropping.

Tabla 5.1: Parámetros de simulación específicos de Triangle Dropping usados para su evaluación.

Triangle Dropping hardware	
Main Table	16 KiB (32 lines, 16-way associative)
Overflow Buffer	16 KiB (32 lines, 16-way associative)
Frame Visibility Buffer	32 KiB
Tile Visibility Buffer	1 KiB, 1 bank, 1 cycle
Primitive Dropper	1 primitive/cycle
Visibility Updater	2 pointers/cycle, 8 in-flight pointers

5.8.1. Impacto en la calidad de la imagen

Dado que Triangle Dropping determina la visibilidad de manera especulativa, a pesar de los mecanismos de mitigación como el comentado arriba, pueden

darse situaciones en las que aparezca un error en la imagen final. Por eso, hemos evaluado la calidad de la imagen resultante tras aplicar Triangle Dropping. El procedimiento consiste en comparar la secuencia de imágenes generada por una GPU que implementa Triangle Dropping con la generada por la arquitectura base, que se toma como referencia. Para obtener la calidad de la imagen se ha empleado la métrica MSSIM (por sus siglas, *Mean Structural Similarity*) [83], una métrica ampliamente usada en la literatura, basada en la calidad perceptible que calcula el grado de similitud de dos imágenes. La métrica MSSIM funciona mejor que otras métricas de similitud que solamente miden diferencia en el color de los píxeles (como por ejemplo PSNR o MSE) en tanto que correlaciona mucho mejor con la percepción del sistema visual humano (también conocido como HVS por sus siglas, *Human Visual System*) [32]. El índice MSSIM es un número en el rango de [0, 1], donde un 1 significa que la imagen de entrada es idéntica a la de referencia, y el umbral perceptible por el sistema visual humano (o HVS por sus siglas en inglés) es 0.95 [29]. En un trabajo posterior, los mismos autores que publicaron la métrica MSSIM extendieron esta métrica a VSSIM (por sus siglas, *Video Structural Similarity*) [84], otro índice de similitud estructural pero enfocado a medir la calidad de una secuencia de vídeo.

Para realizar un estudio más esclarecedor de la calidad de imagen resultante tras aplicar Triangle Dropping, hemos utilizado tanto MSSIM como VSSIM. La Tabla 5.2 muestra los resultados obtenidos para cada *benchmark*, con el valor mínimo y la media de MSSIM, excluyendo los *key frames* (en tanto que obtienen un MSSIM de 1, haciendo que la media sea optimista), así como el valor mínimo y el VSSIM final de todos los fotogramas. Como se puede observar, todos los valores de MSSIM obtenidos por Triangle Dropping no sólo no están por encima del error perceptible por el ojo humano (0.95) sino que el valor mínimo de MSSIM de cada *benchmark* está por encima de 0.99, con un índice MSSIM medio por fotograma de 1 en cada juego evaluado. Incluso más aún, si consideramos la secuencia completa de fotogramas, la calidad mínima de vídeo obtenida según la métrica VSSIM está siempre por encima del 0.99 en todos los juegos evaluados. Por lo tanto, concluimos que Triangle Dropping no incurre en ningún error perceptible por el HVS.

Tabla 5.2: Calidad de imagen y vídeo de los *benchmarks* evaluados usando las métricas MSSIM y VSSIM.

Benchmark	min. MSSIM	avg. MSSIM	min. VSSIM	VSSIM
bbr	0.99925	0.999944	0.999829	0.999849
dds	0.999447	0.999957	0.999869	0.999919
gra	0.99981	0.999951	0.999593	0.999782
hrd	0.999161	0.999852	0.999618	0.99969
hwl	0.995626	0.999253	0.99863	0.998913
maz	0.999371	0.999979	0.99989	0.999951
s3d	0.992689	0.99894	0.991632	0.995344
snd	0.999849	0.999991	0.999919	0.999937

5.8.2. Reducción de la geometría

Dado que Triangle Dropping ataca a la geometría ocluida de una escena, una primera aproximación de su potencial se podría obtener cuantificando la cantidad de geometría que es capaz de eliminar. La Figura 5.4 muestra un desglose de las primitivas que finalmente son escritas en el Parameter Buffer, es decir, aquellas que están dentro del *frustum* y no son *back-faced*. En este experimento se comparan la arquitectura base ("base" en el eje x) con Triangle Dropping ("td" en el eje x), donde puede observarse que Triangle Dropping elimina una media del 31.38 % de las primitivas que acaban escribiéndose en el Parameter Buffer, llegando a alcanzar una reducción máxima del 41.31 % en el caso de *hrd*.

En este mismo experimento también se observa una mayoría de primitivas opacas frente a transparentes, pues de media tan sólo el 8.51 % de las primitivas son transparentes, por lo que el potencial de Triangle Dropping no se ve amenazado. Incluso si quisiéramos aprovechar un poco más de potencial con las primitivas transparentes, de este 8.51 % sólo podríamos aprovechar realmente la fracción de primitivas ocluidas, que tan sólo representa el 3.95 % de media.

5.8.3. Reducción del tráfico a memoria DRAM

La meta principal de Triangle Dropping es reducir los accesos al Parameter Buffer, por lo que un parámetro interesante a analizar es el tráfico a la memoria

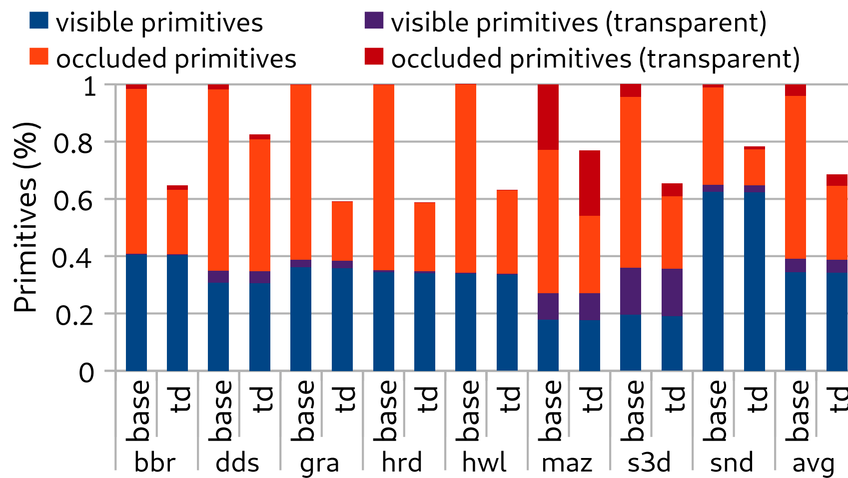


Figura 5.4: Reducción de geometría usando Triangle Dropping, desglosado en primitivas transparentes y opacas.

DRAM causado por el acceso a esta estructura. La Figura 5.5 muestra la reducción del tráfico conseguida. En esta gráfica se muestra un desglose de este tráfico, distinguiendo entre tráfico a DRAM provocado por accesos al Parameter Buffer (barra roja), y tráfico provocado por accesos a otros tipos de datos (barra azul, etiquetada como “other”). Como era de esperar, la fracción del tráfico que se reduce es la causada por accesos al Parameter Buffer, excepto en el caso de *hrd* y *snd*, en el que observamos que la fracción “other” también se reduce. Esto se debe a que la presión sobre la L2 se ha reducido, pues ya no está *contaminada* con geometría ocluida, lo que se traduce en más espacio para almacenar otros tipos de datos, que principalmente son texturas, lo que a su vez, se traduce directamente en una reducción del tráfico a memoria provocado por accesos a texturas.

De media, Triangle Dropping es capaz de reducir el 16.92 % del tráfico a memoria DRAM, llegando a alcanzar reducciones de hasta el 32.28 % en el caso de *hrd*.

Para complementar este estudio, se ha realizado un experimento poniendo el foco en los accesos al Parameter Buffer que acaban yendo a memoria DRAM, es decir, la barra roja de la Figura 5.5. La Figura 5.6 muestra esta fracción de accesos a DRAM desglosada en lecturas y escrituras. Se puede observar que en promedio, Triangle Dropping es capaz de reducir un 28.78 % de esta fracción de accesos, de los

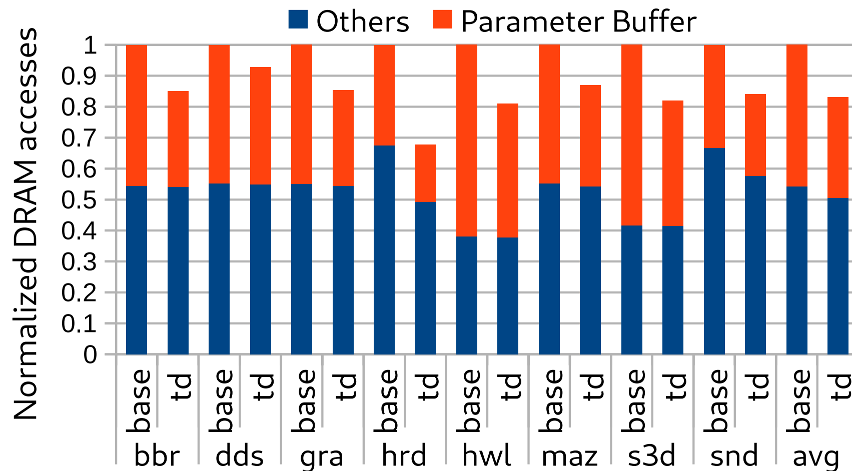


Figura 5.5: Reducción del tráfico a memoria DRAM obtenida por Triangle Dropping, desglosada en tráfico provocado por el Parameter Buffer y tráfico causado por otros tipos de datos.

cuales el 10.92 % vienen de lecturas, es decir, del Tile Fetcher, y el 17.86 % vienen de escrituras, es decir, del Polygon List Builder. Nótese que este estudio sólo muestra los accesos a memoria DRAM provocados por accesos al Parameter Buffer, es decir, accesos que corresponden a primitivas que están dentro del *frustum volume* de la cámara y no son *back-faced*.

5.8.4. Rendimiento y eficiencia energética

Debido a que Triangle Dropping elimina primitivas del *pipeline*, se ahorran muchos cálculos y sobre todo accesos a memoria, por lo que se obtiene una ganancia en rendimiento. La Figura 5.7 muestra el *speedup* obtenido por Triangle Dropping sobre una arquitectura TBDR, obteniendo de media un *speedup* del 20.2 %. En el caso particular de *hrd*, que logra un *speedup* del 43.17 %. La mejora no sólo viene por eliminar accesos al Parameter Buffer, sino también de accesos a texturas (como se explica en la Sección 5.8.3).

La Figura 5.8 muestra el ahorro global de energía logrado aplicando Triangle Dropping sobre una arquitectura TBDR. Dado que el tráfico a memoria DRAM se ha reducido (véase Sección 5.8.3), también se reduce de manera acorde la energía gastada por los accesos a memoria. De media, Triangle Dropping logra una reduc-

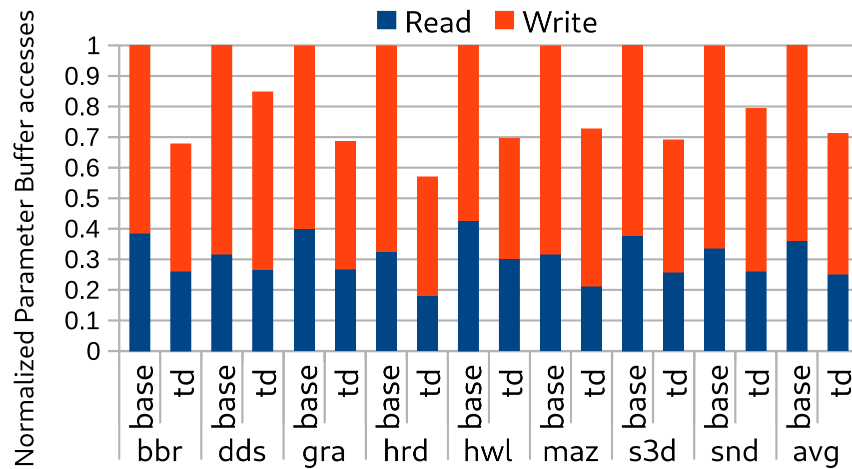


Figura 5.6: Desglose de la fracción de accesos al Parameter Buffer que acaban yendo a memoria DRAM.

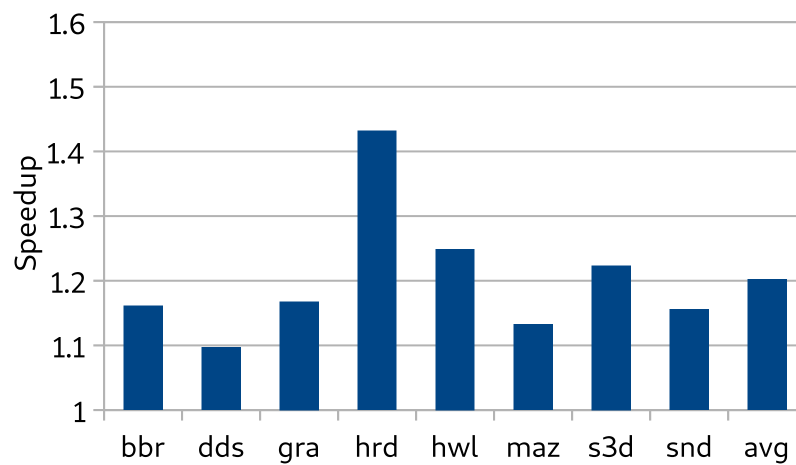


Figura 5.7: *Speedup* obtenido por Triangle Dropping sobre una arquitectura TBDR.

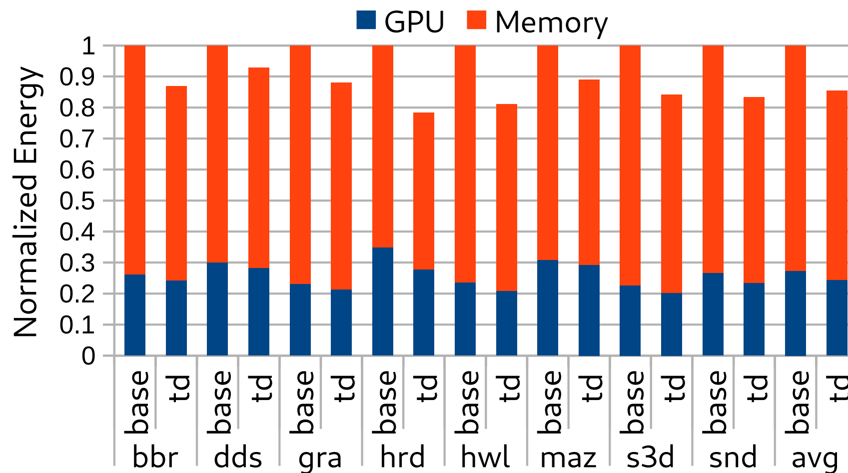


Figura 5.8: Ahorro del consumo energético logrado por Triangle Dropping sobre una arquitectura TBDR.

ción de la energía del 14.5 %, de la cual el 11.7 % viene del sistema de memoria y el otro 2.8 % restante de la actividad de la GPU. La reducción energía en la GPU viene causada principalmente por la reducción de la actividad de la etapa Clipping&Culling, del Polygon List Builder y del Tile Fetcher. Nótese que el Tile Fetcher multiplica estos ahorros por dos dado que en la fase HSR (*Hidden Surface Removal*) también se accede a la geometría almacenada en el Parameter Buffer. En el caso particular de *hrd*, el ahorro de energía puede alcanzar hasta el 21.6 %. Como era de esperar, los ahorros de energía logrados por Triangle Dropping correlacionan bien con la reducción del tráfico a memoria DRAM reportado en la Figura 5.5.

Respecto a la eficiencia combinada de energía-rendimiento de Triangle Dropping, la Figura 5.9 reporta el EDP (*Energy-Delay Product*) obtenido. Se puede observar que de media Triangle Dropping obtiene un ahorro del EDP del 28.2 %, y hasta un 45.2 % en el caso de *hrd*.

Cabe mencionar que Triangle Dropping no afecta a la actividad extra que el *overdraw* provoca en los *Fragment Processors*, en tanto que se ha implementado sobre una arquitectura TBDR, que gracias a la fase HSR es capaz de eliminar por completo el *overdraw* trabajando a nivel de fragmento. Sin embargo, nuestra técnica sí permite eliminar el trabajo que dichas primitivas sobrescritas ocasionarían, y lo consigue hacer en una etapa mucho más temprana del *pipeline* gráfico, en concreto, durante

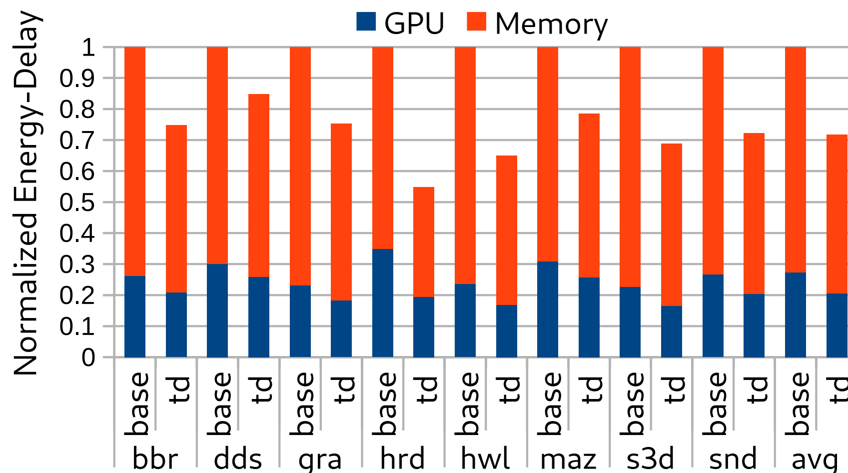


Figura 5.9: Ahorro del EDP logrado por Triangle Dropping sobre una arquitectura TBDR.

la fase de geometría.

5.9. Trabajos relacionados

Una forma diferente de reducir la geometría (no necesariamente ocluida) de una escena es mediante algoritmos de simplificación de modelos, un conjunto de técnicas comúnmente denominado *mesh simplification* o *mesh decimation*. La idea principal de estos algoritmos es, partiendo de un modelo 3D, generar versiones simplificadas del mismo con diferente nivel de detalle o *level of detail* (LOD) siguiendo las restricciones establecidas por un problema específico (topología, perspectiva, ubicación...). Sin embargo, debido a su complejidad, estas técnicas se aplican a nivel de aplicación (o directamente ya están pre-calculadas) y no se suelen realizar a nivel *hardware* de manera transparente.

Una de las primeras propuestas en este campo se encuentra en [76]. En este trabajo se propone un mecanismo simple de reducción de geometría en un modelo 3D. La idea consiste en hacer varias pasadas por los vértices de una malla, eliminando en cada una de ellas un vértice que cumple un cierto criterio. La condición de parada depende del número de vértices que se desea eliminar, que generalmente es un porcentaje del total de la malla. Cada vértice en una pasada se clasifica en cinco

tipos diferentes: simple, complejo, frontera, arista interior o esquina. Los vértices denominados complejos no se pueden eliminar de la malla, dado que si se hiciera, la topología de la malla se vería muy afectada. Puesto que un vértice es compartido por varios triángulos, eliminar dicho vértice provocaría un agujero en la malla, por lo que esta propuesta lo soluciona mediante un mecanismo que denomina "triangulación", mediante el cual se determina un plano equivalente formado por uno o dos triángulos para rellenar ese hueco. El resultado de aplicar este algoritmo es una malla que respeta la topología original pero con una cantidad de vértices muy inferior.

En [56] se propone un algoritmo de simplificación de líneas curvas (también conocido como *line simplification* o *line generalization*) basado en un proceso de subdivisión recursiva. Dada una curva, se calcula la desviación máxima respecto a una recta que pasa por los dos extremos de la curva. Si esta desviación supera un umbral definido por el usuario, entonces la curva se divide en dos partes, por el punto de desviación máxima, y se repite el proceso en cada una de esas dos curvas. Si la desviación es inferior a dicho umbral (caso base), entonces la curva puede sustituirse directamente por una línea recta. Este algoritmo se usa sobre todo en cartografía para computar mapas dependiendo del nivel de zoom para inspeccionar el mismo.

En [75] se propone un algoritmo de reducción de malla basado en triangulación y agrupamiento de vértices (*vertex-clustering*). En este trabajo se propone por primera vez el concepto de *vertex-clustering*, que consiste en agrupar vértices próximos en el espacio que reúnen una serie de características similares y que, por lo tanto, se pueden simplificar en uno sólo que represente al grupo de vértices. A este grupo de vértices se le denomina *cluster*. Este método logra mantener la topología de la malla desde cualquier punto de vista arbitrario de la cámara. Para evitar dañar la topología de la malla, se computa un peso para cada vértice en función de su importancia para la forma del modelo. Los vértices con más peso son los representantes de cada *cluster*. Por supuesto, este algoritmo admite un parámetro para determinar el nivel de simplificación, que depende del criterio a la hora de formar los *clusters*. En [40] se propone un algoritmo similar de reducción de mallas, pero en este trabajo el método propuesto prioriza las curvaturas de las mallas. En [5] se presenta un mecanismo de reducción basado en el concepto *edge-collapse*

(unión de aristas), que consiste en etiquetar vértices como candidatos a ser eliminados. Luego se agrupan en parejas que forman aristas para después realizar la unión de los vértices de cada arista en un punto medio, reduciendo así cada arista candidata a tan sólo un vértice. Nótese que este mecanismo es un caso particular de *vertex clustering*, en el que cada cluster está compuesto únicamente por dos vértices que forman una arista [74]. En [52] se presenta un algoritmo *out-of-core* para GPUs basado también en *vertex-clustering* y enfocado a reducir grandes conjuntos de datos de geometría poliédrica. Una mejora de estos algoritmos de simplificación se presenta en [24], donde para mejorar la calidad de los modelos emplean métodos probabilísticos y estructuras basadas en *octa-tress* (árboles octales). Otro mecanismo *out-of-core* para GPUs se presenta en [67]. En dicho trabajo se propone un algoritmo basado en *edge collapse* para simplificación de mallas que aprovecha la coherencia entre fotogramas para ahorrar cómputos en dicha operación.

En [43] se presenta una implementación híbrida GPU-CPU para la simplificación de mallas mediante la eliminación de pequeños detalles que no aportan mucha información a la figura. En [54] se usan *geometry shaders* que implementan algoritmos de simplificación pre-computada para aplicarlos dinámicamente a modelos y hacer así una simplificación a medida en función de las características de la topología del objeto. Este algoritmo, además, es dependiente del punto de vista, lo que significa que la simplificación se hace en función del punto de vista de la cámara hacia el objeto. Esto le da al algoritmo una información extra, que consiste en saber qué partes del objeto forman parte del actual contorno de la malla proyectada, por lo que se le da más importancia a estos vértices. Esto tiene como consecuencia el poder eliminar más vértices preservando mejor la topología que se percibe, respecto a un algoritmo no *view-dependent*. En [64] se presenta una implementación en OpenCL de un algoritmo de simplificación basado en *edge collapse*. En [44] se propone otro algoritmo de simplificación de mallas densas. La idea es usar una *función de energía* que represente el grado de compactación de la malla, con el fin de minimizarla y así obtener un modelo simplificado que mantenga la topología original de la misma. Este mecanismo es tan pesado, computacionalmente hablando, que no se puede realizar en tiempo real, ya que requiere realizar múltiples iteraciones que hacen uso intensivo de la memoria. En [17] se hace un extenso análisis com-

parativo entre múltiples algoritmos de simplificación de mallas. En [55] también se realizan varias comparaciones entre algoritmos de simplificación de mallas. Sin embargo, todos estos métodos propuestos en la literatura son técnicas implementadas en *software* para reducir la cantidad total de triángulos de una malla (sean estos visibles o no dentro de la escena) por lo que no se pueden comparar directamente con Triangle Dropping, que está implementada dentro de la propia GPU, a nivel *hardware*, con la ventaja de ser completamente transparente al programador.

Por otro lado, existen lo que se denominan *occlusion queries* [3] (o consultas de oclusión), que son llamadas que hace el programador de manera explícita a la GPU para saber de antemano si una *draw call* de OpenGL va a ser visible o no en la escena final. Esto se logra pre-rasterizando las primitivas del comando y comprobando que todos los fragmentos fallan el Z-Test. Esta pre-rasterización tiene una sobrecarga muy considerable y se ha de realizar a nivel de aplicación, por lo que no es viable hacer este tipo de llamadas para todos los comandos de la escena, simplemente para aquellos que sean más complejos y propensos a ser ocluidos. Dado que las *occlusion queries* suelen tardar tiempo en responder, éstas se hacen comúnmente de manera asíncrona, es decir, en vez de esperar al resultado, se puede realizar otro trabajo mientras tanto (como por ejemplo, dibujar otros comandos que no requieran de *occlusion queries*). Y cada cierto tiempo, el programador comprueba (también de manera explícita) si el resultado está disponible. Como se observa, el uso de este tipo de técnicas no sólo no es transparente al programador sino que además conlleva un sobrecoste de desarrollo *software*. En [46] se presenta una implementación *software* de *occlusion queries*. La idea es obtener una lista de lo que denominan *occluders*, es decir, objetos que son propensos a tapar a otros objetos. Para cada uno de estos *occluders* se computa un *shadow frustum*, que no es más que un *frustum* que se construye como resultado de proyectar el objeto en la pantalla usando como punto de referencia la cámara. Si otros objetos se encuentran dentro de algún *frustum*, entonces son completamente ocluidos y, por lo tanto, se puede evitar su procesamiento. Otra técnica de oclusión similar es presentada en [27].

Un aspecto importante a tener en cuenta es que Triangle Dropping se aplica encima de todas estas técnicas *software*, lo que significa que son completamente ortogonales. Triangle Dropping está completamente implementada en el *hardware* del

procesador gráfico y, por lo tanto, es transparente al programador. No es necesario el código fuente de los *vertex* y/o los *fragment shaders*, y se aplica sin intervención alguna del programador.

Hasta donde tenemos entendido, no hay ninguna técnica puramente *hardware* como Triangle Dropping que elimina geometría ocluida de una escena aprovechando la coherencia entre fotogramas de una manera completamente agnóstica al programador, ni para GPUs móviles ni para GPUs de escritorio. La propuesta que más se asemeja es el uso de las *occlusion queries* que, si bien es cierto pueden implementarse con un cierto soporte *hardware* [86], sigue siendo responsabilidad del programador usarlas apropiadamente en cada escenario a nivel de la aplicación.

Sin embargo, dado que todos los *benchmarks* evaluados son juegos comerciales (cuyo código fuente no está disponible de manera pública), asumimos que ya explotan todas las capacidades de las GPUs, incluyendo una potencial implementación de *occlusion queries* a nivel de aplicación para obtener el mejor rendimiento. En este sentido, todos los resultados reportados en esta Tesis se obtienen considerando cualquier posible optimización a nivel de aplicación que pudiesen incorporar dichos *benchmarks*. Además, una recomendación de OpenGL es emitir la menor cantidad de *draw calls* posible debido a su coste de procesamiento. Un ejemplo de este escenario puede observarse en el *benchmark* Hot Wheels (*hwl*), en el que todos los edificios del fondo están agrupados en una única *draw call*, formando un objeto de grandes dimensiones. Por lo tanto, es muy probable que al menos una pequeña parte de este comando sea visible dentro de la pantalla, invalidando así cualquier beneficio de una *occlusión query*, al contrario que Triangle Dropping. Esto se debe a la fina granularidad a la que opera Triangle Dropping (a nivel de primitiva), mientras que las *occlusión queries* se realizan a un grano más grueso a nivel del comando (objeto) completo.

Finalmente, en la literatura se pueden encontrar otras técnicas enfocadas a reducir el efecto negativo del *overdraw*. En [23] se presenta Visibility Rendering Order (VRO), una técnica capaz de reducir el *overdraw* de una escena mediante la reordenación de los comandos en la misma, de modo que son dibujados de delante hacia atrás, permitiendo así al Early Z-Test eliminar más fragmentos que la arquitectura

base. Para lograrlo, se genera un grafo que establece las dependencias de orden de cada comando, y se van alterando para llegar a una solución óptima. Dado que la construcción de este grafo es costosa en términos computacionales se aprovecha la coherencia de fotogramas para ir actualizando el grafo, en vez de construirlo desde cero. Early Visibility Resolution (EVR) [7] aborda el problema del *overdraw* de forma diferente, evitando renderizar *tiles* que no cambian su geometría de un fotograma a otro. Para lograrlo, se genera una firma por *tile* basada en los atributos de la geometría que contiene. Un mecanismo detecta qué primitivas están ocluidas para evitar incluirlas en la firma, y así evitar perder potencial provocado por *tiles* que tienen la misma información de color pero sus primitivas ocluidas cambian. Por otro lado, nuestro mecanismo Ω-Test (presentado en el Capítulo 4) también reduce el *overdraw* a un grano aún más fino. Sin embargo, cabe destacar que la reducción del *overdraw* no es un objetivo clave de Triangle Dropping y, por lo tanto, la hemos evaluado sobre una arquitectura TBDR que elimina por completo el *overdraw*, por lo que los beneficios de Triangle Dropping reportados en esta Tesis provienen exclusivamente de la reducción de la geometría. En cualquier caso, ni VRO, ni EVR ni Ω-Test están enfocados a reducir la actividad derivada de la geometría. En este sentido, Triangle Dropping sí es capaz de eliminar trabajo innecesario en una etapa muchísimo más temprana del *pipeline*, a diferencia de las técnicas mencionadas, ya que explota su capacidad para reducir tráfico a DRAM provocado por accesos innecesarios al Parameter Buffer, una característica que está completamente fuera del alcance de las técnicas mencionadas puesto que funcionan a nivel de fragmento, ya en la fase de Rasterización.

5.10. Conclusiones

En este Capítulo hemos visto el impacto que tiene la geometría ocluida de una escena en el rendimiento y, sobre todo, en el tráfico a memoria que esto supone en una arquitectura TBR debido a los accesos extra al Parameter Buffer y, en consecuencia, el malgasto de energía que conlleva. Es por eso que es importante detectar y eliminar esta geometría de manera prematura. En concreto, Triangle Dropping lo hace después de ensamblar las primitivas (triángulos en esencia), etapa en la que

éstas aún no han llegado a escribirse en el Parameter Buffer. Para lograrlo Triangle Dropping hace uso de la visibilidad de las primitivas de la escena en el fotograma anterior, de modo que puede eliminar aquéllas que no han sido visibles finalmente en la pantalla. También se ha visto la importancia de ser especialmente conservativos a la hora de eliminar este tipo de geometría, puesto que, a pesar de que la información de visibilidad sea muy parecida entre fotogramas consecutivos, no es exactamente la misma, lo que puede llevar a errores perceptibles en la imagen final.

6

DTM-NUCA

Como se ha mencionado en el Capítulo 1, actualmente es bastante común ver 8 o más procesadores incorporados en los dispositivos móviles. Esta tendencia va en aumento conforme va avanzando la tecnología, por lo que no sería de extrañar ver en un futuro 128 o más procesadores incorporados en este tipo de dispositivos de bajo consumo. A medida que el número de procesadores aumenta en un sistema, aumenta también la complejidad de comunicarlos de manera eficiente, convirtiéndose este aspecto en todo un desafío en el ámbito de los dispositivos móviles. En lo que respecta al ámbito específico de las GPUs, a medida que aumenta el número de procesadores, aumenta la replicación entre sus cachés de texturas (Texture Caches), que son privadas, dada su naturaleza de localidad espacial y el grado de compartición entre éstas, provocando en consecuencia una disminución de su capacidad efectiva total. En la introducción (Sección 1.1.3) se ha visto que para una GPU con 4 *Fragment Processors*, el 49.19 % de los bloques de texturas están presentes en los 4

Fragment Processors (véase Figura 1.4).

En este capítulo proponemos DTM-NUCA (*Dynamic Texture Mapping NUCA*), una organización de cachés basada en el esquema NUCA (*Non-Uniform Cache Access*[1]) pero teniendo en cuenta la particularidad del patrón de accesos a memoria que tienen los *Fragment Processors* respecto a la Texture Cache. De este modo, DTM-NUCA persigue dos objetivos, a saber:

- Aumentar la capacidad efectiva total de la Texture Cache por medio de la reducción del grado de replicación de bloques dentro de éstas.
- Como consecuencia de lo anterior, reducir la latencia media de acceso a dichas texturas.

En este esquema propuesto, un bloque de texturas que no se encuentra en una Texture Cache local se puede servir por una caché perteneciente a un nodo remoto, a un coste inferior, en energía y latencia, que acceder a la caché L2. Esto se consigue usando un esquema de mapeo dinámico y liviano que asocia conjuntos de bloques de texturas a *Fragment Processors*, e implementando una red de interconexión en forma de malla 2D para comunicar todas las Texture Caches.

Un aspecto importante sobre las Texture Caches es que, a diferencia de las cachés convencionales de propósito general, éstas no necesitan mantener coherencia entre niveles de caché en tanto que son de sólo lectura, por lo que no presentan patrones de *lectura-modificación-escritura*[2]. Sin embargo, DTM-NUCA utiliza una tabla de afinidad, denominada Affinity Table, para hacer un seguimiento de la propiedad de cada bloque de texturas. De este modo, dada una dirección de un bloque de texturas, la Affinity Table devuelve el *Fragment Processor* que, para ese momento particular, posee ese bloque. Esta relación de propiedad cambia dinámicamente a lo largo del tiempo para adaptarse al patrón de accesos a texturas con el objetivo de maximizar los accesos locales sobre los remotos. De manera adicional, esta propuesta incorpora un mecanismo capaz de permitir replicación en aquellos casos en los que es necesaria para favorecer los accesos locales, con un impacto mínimo en

[1] En la literatura también es común llamarlo *Non-Uniform Cache Architecture.*

[2] También conocidos como accesos RMW (por sus siglas en inglés, *Read, Modify and Write*).

el rendimiento a pesar de la pequeña pérdida de capacidad efectiva debida a esta replicación. DTM-NUCA se presenta en dos variantes. Una primera opción de diseño usa una Affinity Table *centralizada*, enfocada a maximizar la capacidad efectiva total, con el consiguiente coste de aumentar los accesos remotos. Una segunda opción de diseño usa una Affinity Table *distribuida*, que también persigue reducir la latencia media de accesos a bloques de texturas, permitiendo un poco más de replicación.

Así pues, las contribuciones principales de esta propuesta son las siguientes:

- Se propone una nueva organización NUCA pensada específicamente para las Texture Caches privadas de una GPU, capaz de interconectar muchos nodos con una tabla de mapeo de propiedad muy pequeña para reducir sobrecostes.
- Se logra reducir el grado de replicación entre Texture Caches, lo que resulta en un aumento medio de 8 veces de la capacidad efectiva (para el caso de 32 *Fragment Processors*).
- Se reduce la presión sobre la caché L2 compartida eliminando en media el 41.8 % de sus accesos, reduciendo así significativamente la latencia de acceso a texturas.
- En términos generales, esto resulta en un aumento del rendimiento de 16.9 %, y en una reducción de 7.6 % de la energía consumida por el sistema completo.

6.1. Arquitecturas NUCA convencionales

Una caché NUCA convencional se caracteriza por cuatro políticas destacadas que determinan su comportamiento, a saber [53]:

- Política de alojamiento (también conocida como *bank placement policy*). Esta política determina en qué bloque/banco de un conjunto de caché se va a alojar el bloque entrante.
- Política de búsqueda (también *bank search policy*). Esta política es la encargada de buscar el bloque pedido dentro de un conjunto de bloques de caché o, en

caso de no encontrarlo, lanzar un fallo de caché al subsistema de jerarquía de memoria.

- Política de remplazo (también *bank replacement policy*). Esta política determina qué hacer con un bloque que es expulsado de la caché.
- Política de migración (también *bank migration policy*). Esta política se encarga de mover los bloques de localización en función del uso de los mismos (frecuencia, distancia de reuso, distancia al nodo que más, importancia, etc.).

De todas estas políticas, la más crítica es la de alojamiento en tanto que determina el comportamiento operacional de una arquitectura NUCA. En este aspecto, la implementación más simple que existe es S-NUCA [48] (static NUCA), que usa un mapeo estático de direcciones de memoria, para lo cual hace uso de una función fija que asigna bloques del mapa de memoria a un banco concreto (por ejemplo, usando los *bits* menos significativos de la dirección de memoria). Una ventaja que tiene una función de mapeo estática es que no requiere de memoria para su cómputo, sin embargo, los bloques del mapa de memoria se asignan siempre a los mismos bancos, independientemente de si los datos son más frecuentemente accedidos por otros nodos de la red. Esto perjudica drásticamente al rendimiento en los cambios de afinidad que pueda haber en una aplicación y, por lo tanto, obliga a hacer más compleja la política de migración, con el coste que ello conlleva (funciones más complejas, tablas, transferencias *cache-to-cache*...). De manera alternativa, D-NUCA [48] (dynamic NUCA) permite una mayor flexibilidad de alojamiento puesto que cualquier bloque del mapa de memoria se puede colocar en cualquier banco, con el consecuente coste de almacenar estas asociaciones (bloque/banco) en una tabla y consultarla (o bien accediendo a una estructura centralizada, o bien distribuyendo esta tabla entre los nodos de la red de interconexión).

La Figura 6.1 muestra la organización de una arquitectura NUCA convencional, en la que el nivel compartido se reparte en trozos (también conocidos como *slices*), y cada nodo tiene acceso directo a un único *slice*. Como se puede observar, la red de interconexión, que consiste en una matriz de *routers*, comunica todos los *slices* entre sí, por lo que cada nodo puede acceder a todo el nivel compartido a costa de un retardo en función de la distancia que hay hasta el *slice* de destino.

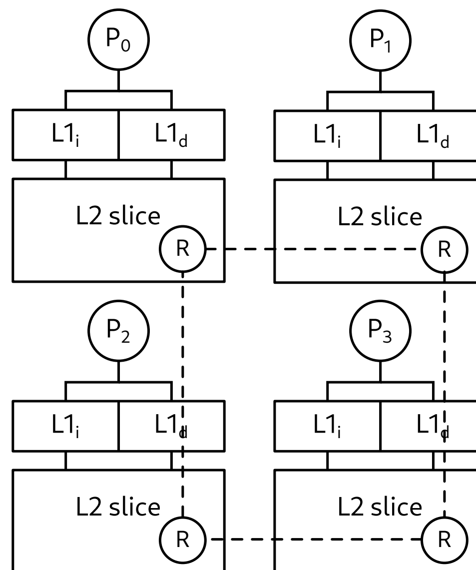


Figura 6.1: Arquitectura NUCA convencional con nodos en una red de interconexión con topología de matriz de 2x2.

6.2. Una organización NUCA para las Cachés de Texturas

Como se ha mencionado anteriormente, el objetivo principal de la propuesta DTM-NUCA es aumentar la capacidad efectiva total de las cachés de texturas privadas asociadas a cada *Fragment Processor* por medio de reducir el grado de replicación de los bloques, de modo que tanto el número de accesos a la caché L2 y la latencia global de acceso se reducen. Para lograr esto, la Texture Cache se organiza como una NUCA en la que cada *Fragment Processor* tiene acceso temporal exclusivo a un rango específico de bloques de texturas. Otra característica importante de DTM-NUCA es que usa un esquema de mapeo dinámico de bloques de texturas basado en el comportamiento de acceso a texturas que presentan los *Fragment Processors*.

6.2.1. Relación de propiedad de bloques de texturas basada en *buckets*

DTM-NUCA usa un esquema de mapeo dinámico muy similar al de una D-NUCA convencional. Sin embargo, en vez de usar un gran directorio con el tamaño

del nivel subyacente compartido L2/L3, ésta usa un pequeño *buffer* (la ya mencionada Affinity Table) para llevar un seguimiento de la propiedad de los bloques almacenados. Hacer un seguimiento del propietario de cada bloque resultaría en una Affinity Table con un tamaño prohibitivo que no se podría almacenar en una memoria *on-chip*. Para reducir esta sobrecarga de almacenamiento, en vez de almacenar el propietario para cada bloque de memoria, DTM-NUCA es más flexible en ese aspecto y agrupa bloques de memoria de manera entrelazada (no contiguos), de modo que se reducen de manera considerable las necesidades de almacenamiento, hasta el punto de que con este esquema, el tamaño de la Affinity Table es independiente del tamaño del espacio de direcciones. A estos grupos se les denomina *buckets*. Esta agrupación de bloques en n *buckets* podría hacerse con una simple función módulo de la dirección del bloque (a) tal que la dirección a pertenece al *bucket* b, donde $b = a \ mod \ n$. Para entender mejor este concepto de *buckets*, imaginemos que tenemos un espacio de direcciones infinito, y queremos dividirlo en dos *buckets*. Entonces podríamos decir que los bloques con dirección par pertenecerían al *bucket* 0, mientras que el resto de bloques (cuya dirección es impar) pertenecerían al *bucket* 1. Sin embargo, se ha probado que agrupar más de un bloque en el mismo "trozo" de *bucket* favorece más la localidad, al mismo tiempo que se reduce el grado de replicación. Por lo tanto, en este esquema de agrupamiento se consideran, además, páginas de m bloques consecutivos, en vez de bloques individuales (el número de página p se puede derivar simplemente de la dirección de bloque a como: $p = a/m$). De este modo, el agrupamiento de todas las páginas en n *buckets* sigue una función módulo similar, de modo que la página p pertenece al *bucket* b, donde $b = p \ mod \ n$.

Para seguir con el mismo ejemplo de antes, imaginemos que tenemos un espacio de direcciones finito y lo queremos dividir en 2 *buckets*. Entonces, lo que hacemos es dividir este espacio en páginas de m bloques consecutivos. Así pues, podríamos considerar que las páginas con índice par pertenecen al *bucket* 0, mientras que aquéllas con índice impar pertenecen al *bucket* 1. De manera experimental, se ha podido comprobar que con páginas de 8 bloques se obtienen los mejores resultados (véase Sección 6.4.1 para más detalles). La Figura 6.2 ilustra este esquema de agrupamiento de bloques de memoria en *buckets*.

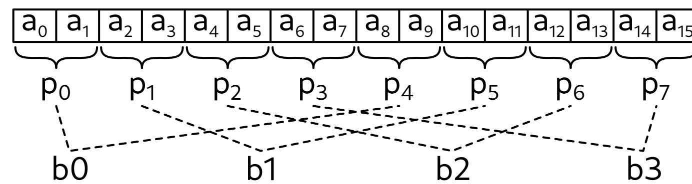


Figura 6.2: Esquema de agrupamiento de bloques de texturas en buckets. Este ejemplo muestra un espacio de direcciones formado por 16 bloques (a_0-a_{15}) las cuales están agrupadas (módulo 4) en 4 buckets (b_0-b_3).

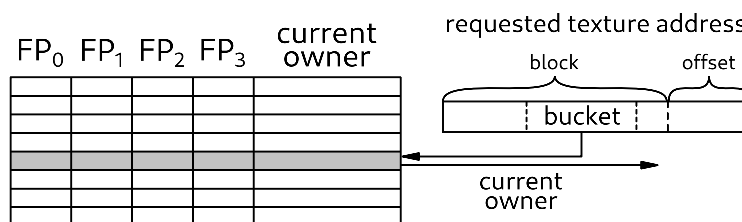


Figura 6.3: Diagrama de implementación de la Affinity Table considerando en caso de 4 *Fragment Processors* y 8 *buckets*.

Tal y como se muestra en la Figura 6.3, la Affinity Table se implementa como una pequeña memoria que almacena una entrada por *bucket*, que contiene un contador saturado por *Fragment Processor*, más un campo para identificar el propietario actual del *bucket*. En la evaluación de esta propuesta, se han usado contadores saturados de 4 *bits* y, para los identificadores de propietario, se han usado 5 *bits*, en tanto que hay 32 *Fragment Processors* en la configuración base. De manera experimental, se ha comprobado que con 32 *buckets* es suficiente para lograr un buen compromiso entre almacenamiento y rendimiento (véase Sección 6.4.2 para más detalles). Esta Affinity Table de 32 entradas ocupa un total de 4256 *bits* (ó 0.51 KiB), y funciona de la siguiente manera. Cuando una petición de un *Fragment Processor* llega a la Affinity Table, se computa el identificador de *bucket*, que se usa para indexar la tabla. Entonces, se obtiene el identificador del *Fragment Processor* propietario y se incrementa el contador saturado del *Fragment Processor* que acaba de acceder a ese *bucket*, anotándose así un acceso más al mismo.

6.2.2. Seguimiento dinámico de la propiedad

Para realizar el proceso de texturizado, los *Fragment Processors* acceden a su Texture Cache local. Sin embargo, debido a que la escena experimenta cambios de un fotograma al siguiente (por ejemplo, cuando se mueven objetos o la cámara), cada *Fragment Processor* accede a bloques de texturas diferentes. Por lo tanto, usar un esquema de mapeo estático resultaría en una degradación significativa del rendimiento debido al alto número de accesos remotos. Para intentar paliar este problema, DTM-NUCA permite que la propiedad de los bloques de texturas cambie dinámicamente. Como se ha mencionado anteriormente, para lograrlo sin que ello conlleve un sobrecoste prohibitivo en memoria, se define una política de asignación de propiedad de grano grueso, basado en el uso de *buckets*. Para determinar el propietario de un *bucket* (esto es, a qué *Fragment Processor* pertenece un *bucket*), la Affinity Table lleva la cuenta del número de accesos de cada *Fragment Processor* a cada *bucket*. Cada vez que un *Fragment Processor* accede a un bloque de texturas, se computa el *bucket* al que pertenece dicho bloque dentro de la Affinity Table, se accede a la entrada correspondiente de la Affinity Table y se incrementa el contador correspondiente. Estos contadores indican cuántas veces ha accedido cada *Fragment Processor* a cada *bucket* hasta el momento y, por lo tanto, aquél que tenga más accesos es el mejor candidato a ser propietario de dicho *bucket*. Inicialmente, todos los contadores están establecidos a cero, por lo que no hay un mejor candidato. Así pues, el primero en acceder al *bucket* se hace propietario de manera inmediata[3]. Cuando un contador satura, es decir, llega a su valor máximo, se produce un evento de *reset*, que consiste en dividir entre dos el valor de todos los contadores de esa entrada de la Affinity Table (esto se logra mediante una operación de desplazamiento de 1 bit a la derecha). Esto también incluye al contador que ha saturado. Entonces, la propiedad se asigna al *Fragment Processor* cuyo contador ha saturado (el propietario podría ser el mismo). Para evitar efectos *ping-pong*[4] durante la asignación del propietario, debido a que dos *Fragment Processors* accedan de manera similar al mismo *bucket*, el contador saturado debe ser mayor que el del actual pro-

[3] A esto se le conoce también como una política de asignación *first-touch*.

[4] Se denomina así porque se estarían robando la propiedad del *bucket* continuamente en períodos muy cortos de tiempo.

pietario más un pequeño porcentaje, indicando así que no se trata de un empate y que, por lo tanto, el propietario está claro.

A pesar de que DTM-NUCA persigue asignar un único propietario para cada *bucket*, debido al alto grado de compartición que un bloque de texturas en particular pueda tener con el resto de *Fragment Processors*, podría ser beneficioso que dichos bloques se sirviesen desde más de un propietario. Para esos casos, DTM-NUCA permite una cierta cantidad de replicación en aras de promover los accesos locales y, en consecuencia, de mejorar el rendimiento. Debido a los cambios de afinidad mencionados, hay un punto en el que un bloque de texturas puede estar replicado tanto en la caché del propietario antiguo como en la del nuevo. Durante este tiempo, el bloque del antiguo propietario podría seguir usándose en tanto que las Texture Caches locales se consultan primero, por lo que el bloque se sirve de manera local en caso de acierto, independientemente del cambio reciente de propietario. Sin embargo, con el fin de consolidar al nuevo propietario y así evitar replicación durante largos períodos de tiempo, los contadores LRU no se siguen actualizando cuando una caché local de un propietario recién cambiado sirve el bloque. De este modo, acabará expulsando el bloque de manera natural de su Texture Cache, si es que ya no lo usa.

Otro problema que puede surgir es el relacionado con la inercia que un cambio de afinidad pueda tener debido al uso de contadores saturados. Si un *Fragment Processor* accede frecuentemente a un *bucket* determinado y, más tarde, otro *Fragment Processor* empieza a hacer lo mismo, pasa un tiempo hasta que el segundo *Fragment Processor* adquiere la propiedad del *bucket*. Esta inercia se exacerba aún más con contadores más grandes, pues el punto de saturación se encuentra más lejano en el tiempo. Existen dos formas de mitigar este problema. La primera pasa por reducir el tamaño de los contadores saturados, aunque sin ser demasiado excesivos, ya que podría disminuir la precisión de la asignación del propietario. La segunda consiste en forzar este evento (el de asignación de propietario) de manera periódica, es decir, hacer un *reset* de todos los contadores, esto es, dividirlos entre dos, y recalcular la propiedad de los *buckets* cada cierto tiempo, denominado época. Una época se mide en términos de acceso a texturas, de modo que cada vez que el conteo total de accesos a texturas exceda la longitud de época, se produce un evento de *reset*

de los contadores. Determinar el tamaño adecuado de época es importante dado que épocas de larga duración provocarían que el problema de la inercia persistiese, mientras que épocas de corta duración provocarían muchos cambios de afinidad. Esta duración de época se ha determinado de manera experimental, con el objetivo de lograr un buen compromiso entre accesos remotos y cambios de afinidad (véase Sección 6.4.3 para más información). Se han evaluado diferentes duraciones de época (expresados en número de accesos a texturas), a saber, 100, 1K, 10K, 20K, 50K y 100K, y se ha podido observar que el mejor compromiso se obtiene con una duración de época de 20K accesos a texturas.

6.3. Opciones de diseño de DTM-NUCA

Hay dos maneras diferentes de implementar la Affinity Table en una arquitectura DTM-NUCA: centralizada en un puto de la red, o distribuida entre los diferentes nodos (*Fragment Processors*) de la misma.

6.3.1. Affinity Table centralizada

En esta versión centralizada, todos los nodos acceden a la misma tabla. La principal ventaja que tiene esta implementación es que todos los nodos comparten al mismo tiempo la misma información actualizada acerca del propietario de cada *bucket*, por lo que la replicación se reduce considerablemente. Sin embargo, esta decisión de Affinity Table centralizada tiene una serie de inconvenientes. El primero, y más obvio, es que produce mucha contención cuando varios nodos necesitan acceder a la Affinity Table a la vez. En segundo lugar, al tener una única tabla, cuando varios nodos necesitan actualizarla, estas operaciones se serializan. Y finalmente, resulta en un aumento de la latencia debido a la ruta crítica provocada por los nodos más lejanos a la Affinity Table y, por lo tanto, no escalaría bien para muchos nodos. Todas estas desventajas hacen que la versión centralizada de la Affinity Table sea un potencial cuello de botella.

En la arquitectura de referencia, cada *Fragment Processor* puede acceder únicamente a su Texture Caché privada, la cual está conectada a un bus compartido

que comunica con la caché L2. Para conectar los *Fragment Processors* de manera directa y permitir el acceso a una Texture Caché remota, DTM-NUCA usa una red de interconexión en forma de malla 2D compuesta por enrutadores que redirigen las peticiones a su *Fragment Processor* de destino usando un mecanismo de enrutamiento X-Y. La Figura 6.4-(a) muestra la mencionada red de interconexión para una Affinity Table centralizada. La latencia de enrutar un paquete, es de un ciclo.

6.3.2. Affinity Table distribuida

Como se ha mencionado anteriormente, un esquema centralizado en el que la Affinity Table es un nodo más de la red de interconexión deriva en una alta contención y latencia de acceso. Para evitar este problema, se ha propuesto una alternativa en la que la Affinity Table está distribuida, tal y como se puede observar en la Figura 6.4-(b), en la que cada nodo tiene su propia versión que se sincroniza más tarde con el resto, a una granularidad de época (definida en 20K accesos a texturas, tal y como se ha mencionado en la Sección 6.2.2) usando un mecanismo de difusión (o *broadcasting*) que funciona de la siguiente manera.

Inicialmente, todos los *Fragment Processors* poseen todos los *buckets* en tanto que no ha habido ninguna comunicación entre ellos hasta el momento. En cada época, las Affinity Tables se sincronizan mediante la difusión por la red de todos los *buckets* de los que son propietarios junto a sus contadores. Es importante destacar que un evento de sincronización no supone una detención en la operación de los *Fragment Processors*. El proceso de difusión se realiza en segundo plano y cada *Fragment Processor* actualiza su Affinity Table local sobre la marcha, según se vaya encontrando paquetes del resto de *Fragment Processors*, actualizando así la propiedad de cada *bucket* con el propietario más reciente. En el caso especial en el que dos contadores tengan exactamente el mismo valor, la propiedad se asigna al *Fragment Processor* de menor identificador. Por lo tanto, los *Fragment Processors* continúan trabajando sin esperar a que la sincronización se complete. Un efecto colateral que tiene esta sincronización *perezosa*, aunque menor, es que justo al comienzo de una época la información de propiedad no está del todo actualizada entre todos los nodos. Sin embargo, esto tiene un impacto casi despreciable dado que este período

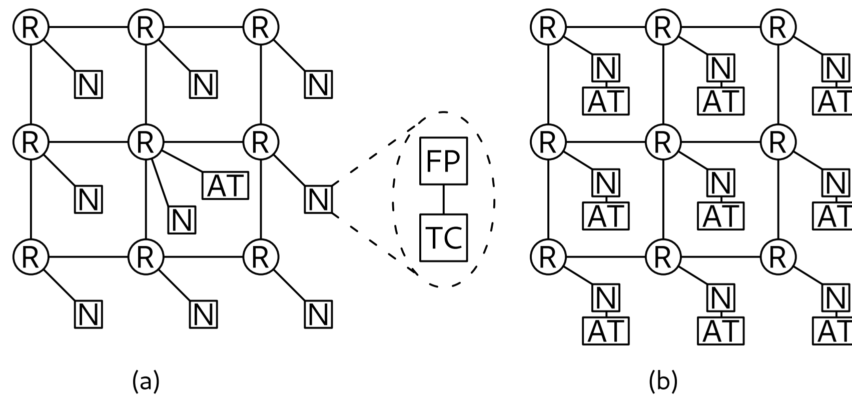


Figura 6.4: Red de interconexión para una implementación DTM-NUCA de 9 nodos, en la que cada nodo consiste en un *Fragment Processor* y su correspondiente Texture Cache local. (a) Esquema centralizado con la Affinity Table en el centro de la red. (b) Esquema distribuido con una Affinity Table por nodo.

de tiempo supone un porcentaje muy bajo del total de la duración de la época[5]. Simplemente, un *Fragment Processor* obtendrá el mismo propietario que tenía almacenado en su Affinity Table justo antes del cambio de época. En cualquier caso, el mecanismo garantiza que la información de propiedad se propagará completamente a todos los *Fragment Processors* de la red después de un pequeño retardo.

6.4. Ajuste de parámetros de DTM-NUCA

En esta sección se verá cómo se han determinado los parámetros arquitectónicos de DTM-NUCA. Éstos han sido obtenidos por medio de experimentos en función de los *benchmarks* evaluados, por lo que no son definitivos y, por lo tanto, tampoco son los más adecuados para todas las situaciones/escenas posibles a las que se puede enfrentar esta arquitectura.

[5]Experimentalmente se ha podido probar que esta sincronización está completamente resuelta durante el primer percentil de la época.

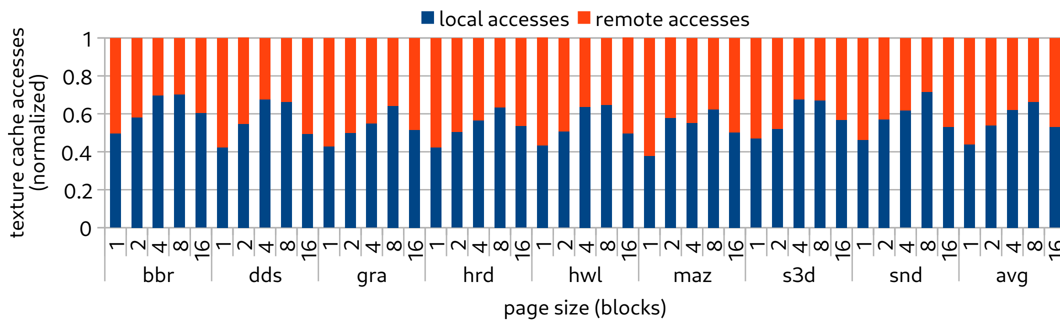


Figura 6.5: Análisis del impacto del tamaño de página sobre los accesos a texturas, desglosados en accesos locales y remotos (normalizados respecto al *baseline*).

6.4.1. Determinando el mejor tamaño de página para el esquema de agrupamiento de DTM-NUCA

Tal y como se ha mencionado en la sección 6.2.1, el esquema de agrupamiento en *buckets* de DTM-NUCA describe páginas de m bloques de texturas consecutivos. Por lo tanto, cada *bucket* está compuesto por una serie de páginas entrelazadas con el fin de favorecer lo máximo posible la localidad de bloques (véase Figura 6.2).

Para determinar el mejor tamaño de página, la Figura 6.5 muestra el impacto que tiene este parámetro en la localidad de acceso a texturas. Aquí se puede observar que a medida que aumenta el tamaño de página, la cantidad total de accesos locales también aumenta. Sin embargo, a partir de 8 bloques de texturas por página, la localidad empieza a disminuir debido a la gran cantidad de cambios de afinidad que se producen, pues cuanto más grandes son las páginas, más probabilidad hay de que dos o más nodos accedan concurrentemente a ella, compitiendo así por la propiedad del *bucket* completo. La Figura 6.6 complementa este análisis mostrando los cambios de afinidad en función del tamaño de página. Así pues, este estudio demuestra que el mejor equilibrio entre accesos locales y cambios de afinidad se encuentra en elegir tamaños de página de 8 bloques consecutivos.

6.4.2. Número de *buckets* de la Affinity Table

El esquema de particionado del rango de direcciones de texturas en *buckets* es crucial para lograr almacenar la Affinity Table en una memoria tan pequeña que

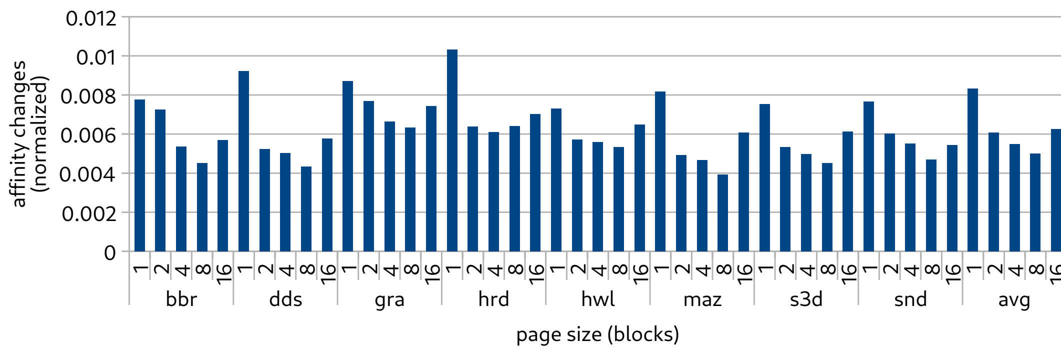


Figura 6.6: Análisis del impacto del tamaño de página sobre los cambios de afinidad.

quepa en memoria *on-chip*. Sin embargo, es importante no ser muy agresivo con el particionado de cara a no perjudicar el rendimiento global. Por un lado, si tenemos un particionado muy fino de direcciones, es decir, un particionado con muchos *buckets*, entonces el tamaño de la Affinity Table será más grande y no se podrá almacenar en memoria *on-chip*. Mientras que si, por el contrario, el particionado es de grano muy grueso, es decir, se escogen pocos *buckets*, entonces la Affinity Table tendrá tan poca información que se provocarán muchas colisiones en el acceso a *buckets* por parte de diferentes procesadores, provocando a su vez que los cambios de afinidad aumenten y, por lo tanto, que el rendimiento disminuya drásticamente. Es por eso que un compromiso entre almacenamiento y localidad de datos es crítico a la hora de seleccionar este parámetro.

La Figura 6.7 muestra el impacto que tiene el tamaño de la Affinity Table sobre el rendimiento en términos de accesos a texturas. Como se puede observar, el mejor compromiso entre almacenamiento y localidad se logra escogiendo 32 *buckets* para la Affinity Table. Por encima de ese valor, las necesidades de almacenamiento aumentan exponencialmente sin suponer una mejora substancial en la localidad de acceso a texturas.

6.4.3. Determinando la longitud de época

Elegir una longitud de época adecuada en una arquitectura DTM-NUCA es también un aspecto crucial para su rendimiento, pues este parámetro se encarga

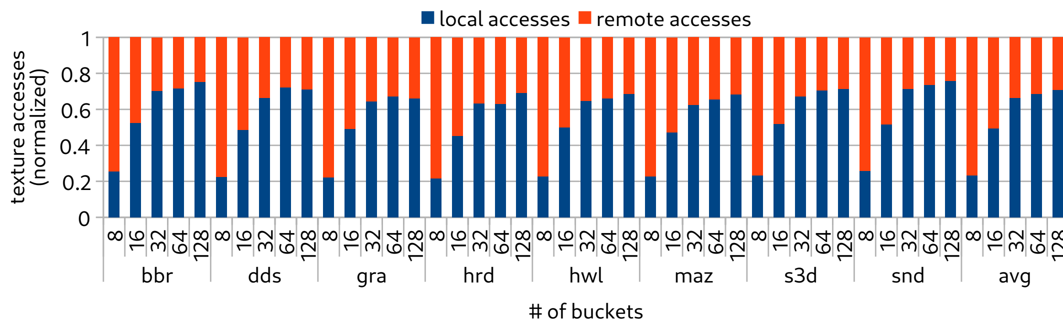


Figura 6.7: Análisis del impacto del tamaño de la Affinity Table sobre el rendmiento en términos de accesos locales a texturas (normalizados respecto al *baseline*).

de forzar los cambios de afinidad y paliar así la inercia que se pueda provocar debido a largos períodos de propiedad de un *bucket* por parte de un procesador. En este aspecto, una época corta supondría muchos cambios de afinidad, pues es más probable que los propietarios de cada *bucket* no se hayan consolidado, degradando así el rendimiento. Por otro lado, si las épocas son muy largas, entonces la inercia de propiedad será muy grande, provocando que los propietarios se acomoden en *buckets* de los que probablemente no sean los mejores candidatos a poseerlos, disminuyendo así la localidad de acceso a texturas.

Para elegir una buena longitud de época, se ha evaluado un amplio rango de longitudes, a saber: 100, 1K, 10K, 20K, 50K y 100K accesos a texturas. La Figura 6.8 muestra el impacto que tiene la longitud de época sobre la localidad de accesos a texturas. Tal y como se puede observar, a medida que la longitud de época aumenta, los accesos locales a texturas disminuyen debido a la mencionada inercia de propiedad. Además, puede notarse un decremento de accesos locales bastante más pronunciado en longitudes de época por encima de los 20K accesos a texturas.

Por otro lado, la Figura 6.9 muestra el impacto de la longitud de época sobre los cambios de afinidad. Tal y como se esperaba, a medida que la longitud de época aumenta, los cambios de afinidad disminuyen debido a que el evento de *reset* se produce menos veces. Aumentar la longitud de época por encima de los 20K accesos a texturas tiene un impacto más bajo sobre los cambios de afinidad, puesto que las épocas son tan largas que los eventos de *reset* se producen debido a la saturación de los contadores de acceso, en lugar de por la época en sí.

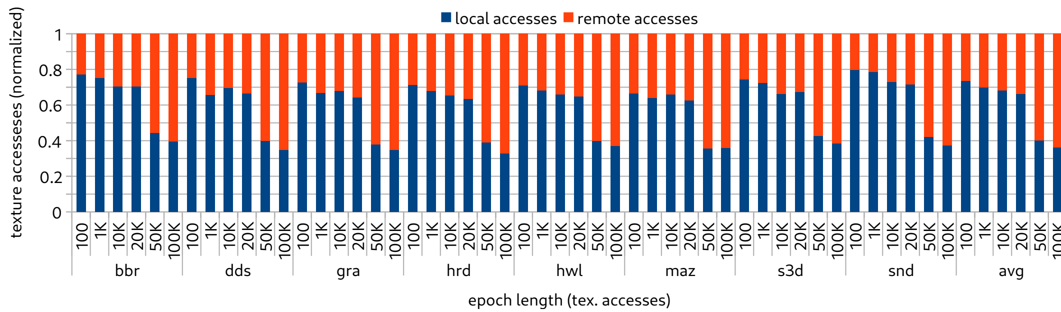


Figura 6.8: Análisis del impacto de la longitud de época sobre el rendimiento en términos de accesos locales a texturas (normalizados respecto al *baseline*).

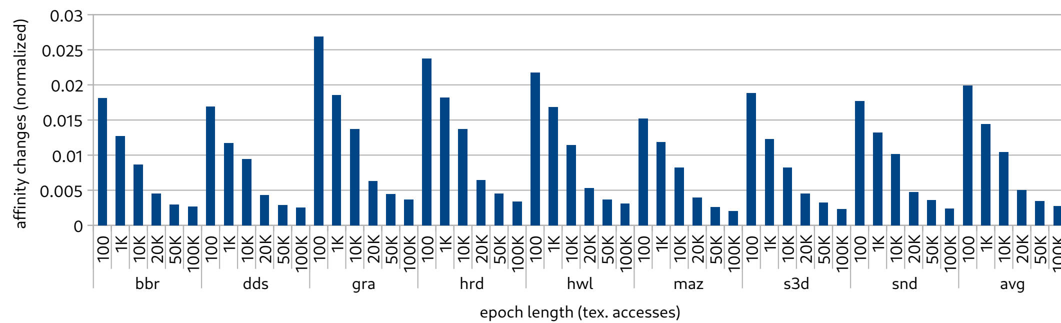


Figura 6.9: Análisis de los cambios de afinidad en función del tamaño de época (normalizados respecto al total de accesos a texturas).

Tabla 6.1: Parámetros de simulación usados en DTM-NUCA.

Caches	
L2 Cache	1 MiB, 8 banks, 18-cycle lat, 8-way assoc
Programmable stages	
Fragment Processing stage	32 Fragment Processors (FPs)
DTM-NUCA parameters	
Number of buckets	32 buckets
Access counters size	4 bits
Page size	8 blocks
Affinity Table size	0.51 KiB
Routing latency	1 cycle/hop
Epoch length	20K texture accesses

Resumiendo, para tamaños de época por encima de los 20K accesos a texturas la cantidad total de cambios de afinidad no se ve reducida considerablemente, mientras que por contra, el rendimiento en términos de accesos locales a texturas se ve drásticamente perjudicado. Por otro lado, se puede observar que para longitudes de época por debajo de los 20K los accesos locales a texturas aumentan muy poco, sin embargo, los cambios de afinidad se incrementan considerablemente. Por lo tanto, y a la vista de estos estudios experimentales, el mejor compromiso entre cambios de afinidad y localidad de accesos a texturas se logra escogiendo longitudes de época de 20K accesos a texturas.

6.5. Resultados experimentales y discusión

La Tabla 6.1 muestra los parámetros usados para la evaluación de DTM-NUCA.

Adicionalmente a la configuración de referencia (denominada *baseline* en este documento), la cual consiste en una Texture Cache privada por cada *Fragment Processor*, se han evaluado tres arquitecturas NUCA diferentes. La primera es una implementación D-NUCA que, en lugar de replicar los datos entre cachés y tener un directorio con una lista de compartidores, evita completamente la replicación de bloques de texturas, permitiendo un único compartidor por entrada de directorio. Nótese que dicho esquema requiere un directorio del tamaño del nivel compartido,

que en este caso es la caché L2. Contra esta implementación comparamos las dos variantes DTM-NUCA propuestas en este Capítulo, es decir, DTM-NUCA con una Affinity Table centralizada y DTM-NUCA con una Affinity Table distribuida (etiquetadas como DTM-NUCA y DTM-NUCA-Dist, respectivamente). En todas las configuraciones consideramos 32 *Fragment Processors* con sus respectivas 32 Texture Caches.

Dado que el propósito de esta propuesta NUCA es aumentar la capacidad efectiva de las Texture Caches, el primer conjunto de experimentos mide la cantidad de accesos al nivel superior que se reducen debido a esta capacidad extra. La Figura 6.10 muestra la cantidad de accesos a la caché L2 normalizados respecto a la arquitectura de referencia para las tres arquitecturas NUCA mencionadas. Como era de esperar, D-NUCA muestra los mejores resultados en cuanto a accesos a L2 (un 42.2 % de reducción de media) debido a que evita por completo la replicación y maximiza la capacidad efectiva del conjunto de Texture Caches, por lo que se puede considerar como una cota superior en este aspecto. Por otro lado, DTM-NUCA alcanza una reducción media del 41.8 %, muy cercano a la capacidad óptima, mientras que DTM-NUCA-Dist obtiene una reducción más modesta del 33.3 % de media. Esto se debe al retardo en la sincronización de las Affinity Tables, el cual conlleva una pérdida de precisión en tanto que, hasta que la sincronización no se complete, algunas Affinity Tables pueden diferir y asignar el mismo *bucket* a diferentes *Fragment Processors*.

La Figura 6.11 ilustra mejor los beneficios de DTM-NUCA comparando la cantidad total de accesos a la caché L2 de ésta con los producidos en una arquitectura base con una Texture Cache de tamaños variados, que van desde 16 KiB hasta 512 KiB. Como se puede observar, una DTM-NUCA de 16 KiB (línea roja) tiene casi la misma cantidad de accesos a L2 que una arquitectura base con 8 veces más capacidad (128 KiB), mientras que una DTM-NUCA-Dist de 16 KiB logra los mismos resultados que un baseline con una Texture Cache 4 veces más grande (64 KiB).

Sin embargo, el objetivo de DTM-NUCA es incrementar también la cantidad total de accesos locales frente a los remotos, como medida de reducir la latencia media. Esto se logra permitiendo un pequeño grado de replicación en las cachés lo-

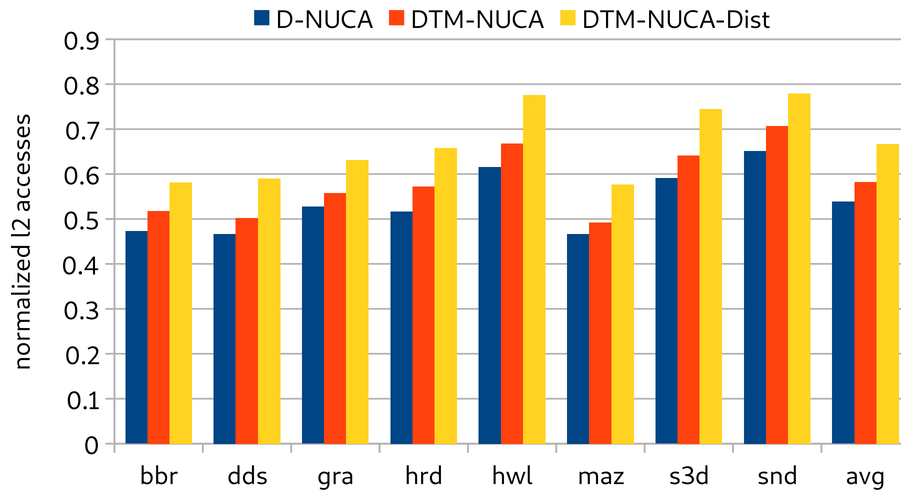


Figura 6.10: Accesos normalizados a la caché L2 para los diferentes diseños de NU-CA con respecto a la arquitectura de referencia (*baseline*).

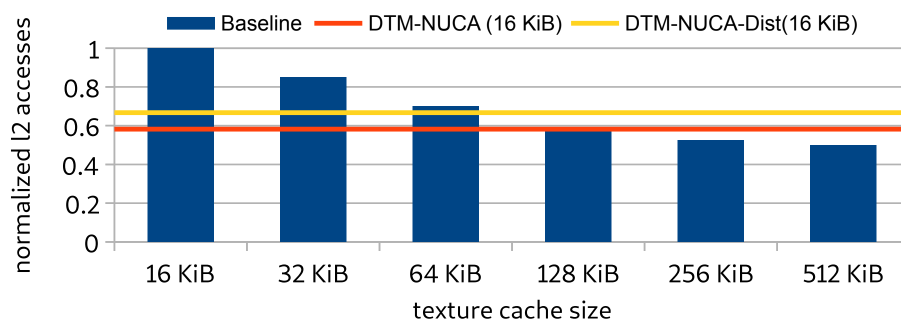


Figura 6.11: Accesos a L2 normalizados de DTM-NUCA respecto al *baseline* con diferentes tamaños de Texture Cache.

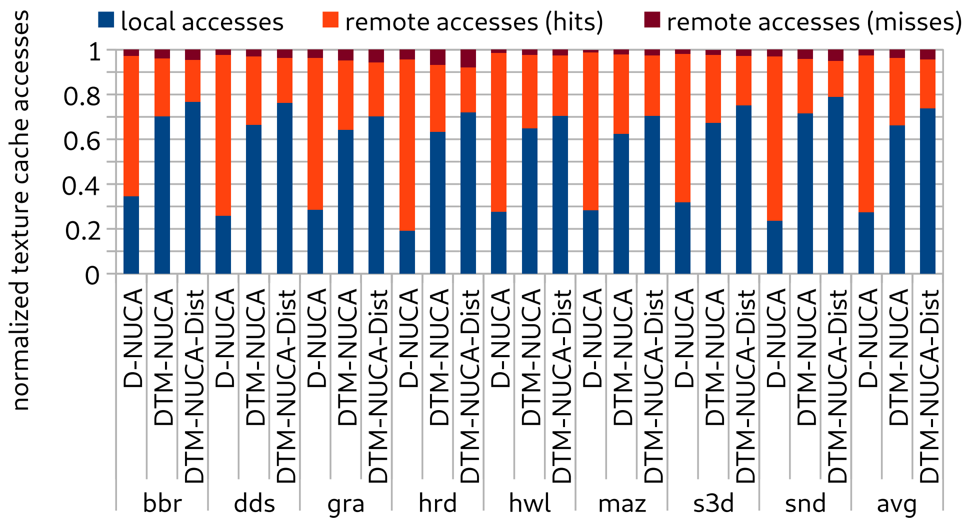


Figura 6.12: Accesos locales y remotos a la Texture Cache para las organizaciones NUCA evaluadas.

cales. La contrapartida es una reducción general de la capacidad efectiva. La Figura 6.12 muestra los accesos a texturas desglosados en accesos locales y remotos que, a su vez, están desglosados en aciertos y fallos de la caché L2. Como se puede observar, DTM-NUCA y DTM-NUCA-Dist obtienen un número de fallos en la caché L2 un poco más elevado que D-NUCA (tal y como se ha discutido anteriormente), sin embargo, obtienen una cifra impresionante de accesos locales que son aciertos (una media de 66.3 % y 73.9 %, respectivamente), muchos más que el 27.5 % que obtiene D-NUCA, el cual muestra una cantidad de aciertos remotos elevada. Esto se debe a que D-NUCA ignora completamente si un acceso se hace a una caché local o remota, ya que su objetivo es eliminar la replicación por completo. Además, D-NUCA carece de un mecanismo de migración de datos para maximizar los accesos locales sobre la marcha, a diferencia de DTM-NUCA y DTM-NUCA-Dist, gracias a la Affinity Table. Dado que DTM-NUCA-Dist permite un mayor grado de replicación debido al retardo en el proceso de sincronización de las Affinity Tables, obtiene más accesos locales que DTM-NUCA, en detrimento de la capacidad efectiva total. Estos resultados demuestran la bondad de la asignación dinámica de *buckets* realizada por la Affinity Table.

El rendimiento de DTM-NUCA puede observarse en la Figura 6.13. Aquí se

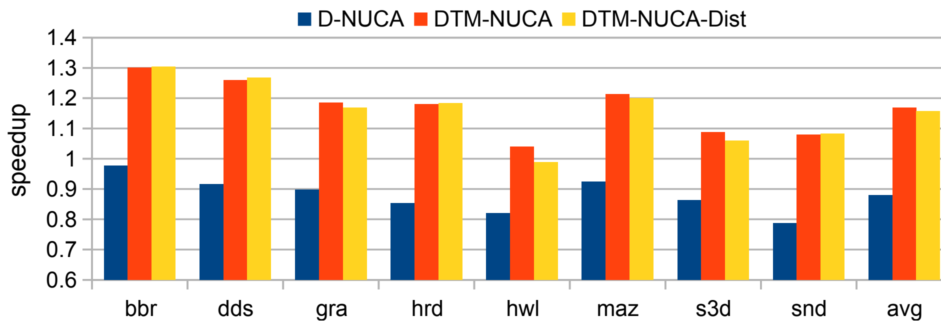


Figura 6.13: Aceleración de las organizaciones NUCA evaluadas respecto al *baseline*.

pueden observar mejoras en el rendimiento de hasta el 30.2 % y el 30.4 % en el caso de `bbr` usando DTM-NUCA y DTM-NUCA-Dist respectivamente. De media, DTM-NUCA y DTM-NUCA-Dist obtienen una mejora del rendimiento del 16.9 % y el 15.7 % respectivamente. Por el contrario, D-NUCA sufre una degradación del rendimiento del 22 % debido a su pobre política de migración de datos, que sólo permite migración cuando un bloque se expulsa de la caché del propietario. Esto deriva en un aumento de los accesos remotos que terminan incrementando la latencia de acceso media.

La Figura 6.14 muestra el consumo de energía de las arquitecturas NUCA evaluadas, desglosado en memoria y GPU. Nótese que las principales fuentes de ahorro energético vienen de la energía dinámica debido al incremento del rendimiento y la reducción de accesos a la caché L2. Los mejores resultados se obtienen en `bbr` y `maz`, para el caso de DTM-NUCA-Dist, logrando reducciones de hasta el 11.1 % y el 10.3 % respectivamente. De media, DTM-NUCA y DTM-NUCA-Dist obtienen ahorros de energía del 6.9 % y el 7.6 % respectivamente. Por el mismo motivo que el rendimiento, D-NUCA es peor incluso que el *baseline*, incrementando el consumo energético en un 4 % de media.

Finalmente, nos centramos en la escalabilidad de DTM-NUCA. La Figura 6.15 muestra la mejora en el rendimiento de cada variante, evaluando desde 4 hasta 32 *Fragment Processors*. Como se puede observar, con tan sólo 4 *Fragment Processors* DTM-NUCA obtiene una ganancia de rendimiento media del 10.2 %, mientras que DTM-NUCA-Dist obtiene un 7.0 %. A medida que el número de nodos crece, el ren-

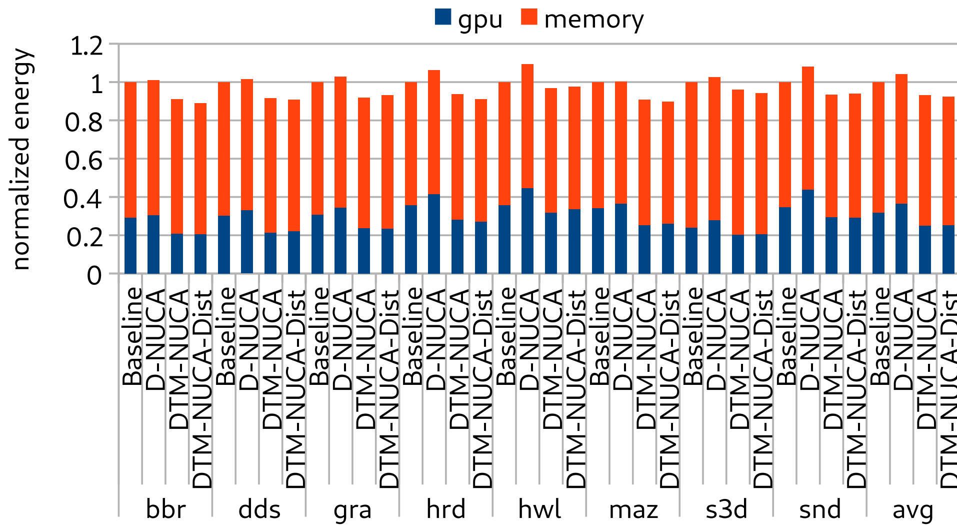


Figura 6.14: Energía consumida por las organizaciones NUCA evaluadas normalizada al *baseline*.

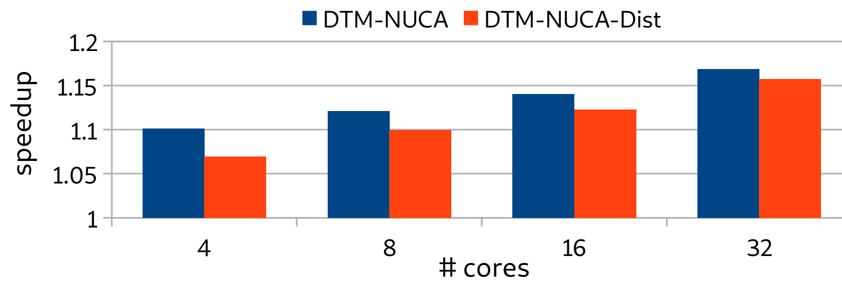


Figura 6.15: Escalabilidad de DTM-NUCA para un número de nodos creciente.

dimiento (normalizado respecto al *baseline* con el mismo número de nodos) aumenta, pero ese incremento se ve reducido en cada paso. A pesar de que DTM-NUCA obtiene mejores resultados, DTM-NUCA-Dist tiende a converger con DTM-NUCA a medida que el número de nodos aumenta.

6.6. Trabajos relacionados

Respecto a arquitecturas con organización NUCA, existen esquemas mucho más complejos como R-NUCA [42] que aplica diferentes políticas de alojamiento en función de la clase de acceso (es decir, si es acceso a la caché privada o acceso al nivel compartido), y realiza un entrelazado rotativo con un conjunto de nodos en

consecuencia. En [53] se propone un novedoso algoritmo de búsqueda para organizaciones D-NUCA cuya meta principal es reducir tanto la latencia de acceso como el tráfico hacia la red de interconexión haciendo peticiones a los nodos más cercanos. En [61] se propone y evalúa una novedosa red de interconexión heterogénea de cachés interbanqueadas, prestando especial atención en la sobrecarga que tiene en el rendimiento el retardo de ésta en una organización basada en NUCA, por lo que la modelan con gran precisión y detalle. En [11] se propone un esquema D-NUCA que se adapta dinámicamente en función de las necesidades de la aplicación para reducir el consumo de energía estática de las cachés. En [12] se demuestra que las políticas de migración de datos basadas en promoción aumentan el consumo de energía dinámica en algunos escenarios, así que proponen un esquema de migración dinámico selectivo, que va en función de estos escenarios. En [10] se propone un nuevo esquema de codiseño entre el último nivel de la jerarquía de caché[6] y la red de interconexión, para lo cual predefinen un conjunto de rutas rápidas fijas (*fast paths*) entre los bancos de la caché que se caracterizan por conectar directamente a nivel de reloj los enrutadores intermedios de éstas. De este modo, las transferencias entre bancos de caché pueden hacerse en un único paso sin necesidad de hacer ni *buffering* ni conmutación de paquetes intermedios que viajan por la red, pues los paquetes que viajan por estas rutas llegan directamente a sus destinatarios y, por lo tanto, son consumidos al instante. En [58] se presenta una arquitectura NUCA diseñada para CMPs[7], que reduce la latencia de acceso usando una función de coste que coloca los datos lo más cerca posible de sus propietarios. En [57] los mismos autores perfeccionan este mecanismo permitiendo algunas víctimas y réplicas de bloques, provocando así que se aprovechen mucho mejor las lecturas de datos compartidos por parte de una multitud de nodos, y maximizando además la utilización de la caché en situaciones en las que la carga sobre los nodos está severamente desbalanceada. En [45] también se aprovechan las víctimas y las réplicas para aumentar la localidad de los datos usando un método adaptativo basado en la presión que tiene cada banco de caché en un período de tiempo determinado. Este esquema se rige por tres conceptos clave, a saber: una política de remplazo pseudo-LRU basada tanto en la actividad como en la réplica de cada bloque de caché; una política de

[6]También conocido como *last-level cache* (o LLC).
[7]Conocidos así por sus siglas: *Chip Multi-Processor*.

alojamiento *first-touch* y la ya mencionada política de replicación. En [26] se presenta una arquitectura NUCA con un particionado dinámico en el nivel compartido, el cual controla dinámicamente la cantidad de capacidad que tiene cada *slice* de caché en función de las necesidades. La idea que subyace en este trabajo es que el particionado de una caché no tiene por qué ser homogéneo. Por otro lado, Nu-RAPID [16] desacopla el alojamiento de los datos del de los *tags* de los bloques de caché, y coloca datos frecuentemente accedidos en los bancos de caché más rápidos para reducir en la medida de lo posible la cantidad neta de migraciones de datos.

A pesar de que existen un sinfín de esquemas relacionados con arquitecturas NUCA, el único que se implementa en procesadores comerciales a día de hoy es S-NUCA y, por supuesto, continúa siendo un campo activo en el ámbito de la investigación [71].

DTM-NUCA difiere de los trabajos anteriormente citados en que es la primera organización NUCA diseñada específicamente para el nivel L1 privado de las cachés de texturas de las GPUs, las cuales tienen la peculiaridad de ser de sólo lectura, por lo que no es necesario un protocolo de coherencia, y en que se centra principalmente en reducir la replicación de bloques con un esquema de mapeo mucho más simple, basado en la temporalidad de la propiedad de los bloques de texturas.

6.7. Conclusiones

A lo largo de este capítulo se ha visto la importancia que tiene la escalabilidad de un sistema paralelo a medida que aumenta el número de unidades de procesamiento. Especialmente, en lo que a GPUs móviles se refiere, los *Fragment Processors* presentan un mayor grado de replicación a medida que aumenta el número de éstos debido a la alta localidad espacial que presenta el patrón de acceso a texturas. Como se ha podido observar, este grado de replicación trae consigo una drástica reducción de la capacidad efectiva del conjunto de cachés de texturas, lo cual supone un desperdicio de recursos. DTM-NUCA trata de aprovechar al máximo esta capacidad por medio de la compartición de bloques de texturas entre *Fragment Processors*, de modo que a un conjunto de bloques de texturas le corresponde un único

Fragment Processor y, sólo cuando esta replicación de bloques es positiva para el rendimiento, se permite.

7

Conclusiones y vías futuras

7.1. Conclusiones

Para cerrar esta Tesis Doctoral, vamos a hablar de las conclusiones que se pueden sacar de los trabajos realizados durante la misma.

Como se ha podido observar, el *overdraw* juega un papel muy importante para la eficiencia del renderiazdo de una escena, pues supone trabajo y esfuerzo inútil llevado a cabo por las unidades de procesamiento de la GPU, lo que conlleva a un malgasto de energía y una degradación de la batería del dispositivo móvil. En este aspecto, se ha propuesto Ω-Test, una técnica microarquitectónica que resuelve el problema de la visibilidad de manera eficiente haciendo uso de la información del Z-Buffer del fotograma previo, a diferencia del clásico Early Z-Test, el cual usa únicamente información del fotograma actual y construye el Z-Buffer desde cero en cada fotograma.

Este esquema se beneficia de la coherencia de *frame*, presente de manera natural en los videojuegos. Sin embargo, una pequeña diferencia de profundidades entre un fotograma y otro podría llevar a errores en la imagen, por lo que Ω-Test cuenta con un sistema de detección de errores que localiza los errores y los corrige de manera selectiva, dando lugar a una imagen *idéntica* a la original. Por otro lado, el ancho de banda y las necesidades de almacenamiento de Ω-Test se han reducido mediante el uso de un esquema de *coarsening*, el cual tiene una pérdida de precisión despreciable.

Con todo esto, Ω-Test consigue reducir el *overdraw* un 32.7 % de media, aumentar el rendimiento medio de la GPU un 16.3 % y reducir el consumo energético una media de 26.4 % en términos de EDP.

Por otro lado, durante la realización de esta Tesis se ha podido observar que en los videojuegos comerciales existe una gran cantidad de geometría que se procesa pero que finalmente no es visible (o lo es pero muy poco). A esto se le denomina geometría ocluida. Esto obviamente supone un malgasto de recursos de la GPU y de memoria, lo que conlleva a la degradación de la duración de la batería del dispositivo móvil.

Para superar este problema, se ha propuesto Triangle Dropping, una microarquitectura para GPUs móviles que reduce drásticamente la cantidad de geometría ocluida en una escena aprovechando la visibilidad de las primitivas en el fotograma previo. Con este esquema se demuestra que la geometría de una escena se puede reducir un 31.3 % de media. Además, para demostrar que las ganancias que se obtienen de Triangle Dropping son integralmente por la reducción de geometría y no de *overdraw*, se ha usado como *baseline* una arquitectura TBDR, que elimina por completo el *overdraw*. Con todo esto, se demuestra que Triangle Dropping aumenta el rendimiento medio de una GPU móvil un 20.2 %, reduce el consumo energético un 14.5 % y reduce el EDP un 28.2 %.

Finalmente, durante esta Tesis se ha podido observar que la replicación de los bloques de texturas puede suponer un desaprovechamiento de la capacidad neta de las Texture Caches. Es por eso que se propone DTM-NUCA, un esquema NUCA de mapeo dinámico diseñado exclusivamente para Texture Caches en una GPU móvil

cuyo objetivo es aumentar la capacidad neta de la Texture Cache por medio de la eliminación de réplicas en diferentes nodos. A diferencia de una NUCA convencional, DTM-NUCA incorpora una pequeña tabla (la Affinity Table) cuyo tamaño se desvincula por completo del tamaño del nivel superior (la caché L2) Para mejorar el acceso local a las Texture Caches sin detrimento del rendimiento, DTM-NUCA permite algunas réplicas de manera deliberada. DTM-NUCA se presenta en dos versiones, a saber: un diseño centralizado y otro distribuido. Esta propuesta consigue reducir la presión de la caché L2 un 41.8 % de media y maximizar los accesos locales a una media del 73.9 %. Todo esto tiene como resultado un aumento medio del rendimiento del 16.9 % y una reducción de energía del 7.6 % de media respecto a una arquitectura tradicional de cachés privadas.

Cabe remarcar que las propuestas anteriores han dado como resultado una serie de publicaciones que se enumeran a continuación (en orden de realización):

- D. Corbalán-Navarro, J.L. Aragón, M. Anglada, E. de Lucas, J.M. Parcerisa y A. González. "Omega-Test: A Predictive Early-Z Culling to Improve the Graphics Pipeline Energy-Efficiency". *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, issue 14, pp. 4375-4388, Diciembre de 2022.

- D. Corbalán-Navarro, J.L. Aragón, M. Anglada, E. de Lucas, J.M. Parcerisa y A. González. "Improving the Energy Efficiency of the Graphics Pipeline by Reducing Overshading". *Actas de las XXXI Jornadas de Paralelismo*, pp. 125-134, Málaga, Septiembre de 2021.

- D. Corbalán-Navarro, J.L. Aragón, M. Anglada, J.M. Parcerisa y A. González. "Triangle Dropping: An Occluded-Geometry Predictor for Energy-efficient Mobile GPUs". *ACM Transactions on Architecture and Code Optimization*, vol. 19, issue 3, article 39, pp. 1-20, Septiembre de 2022.

- D. Corbalán-Navarro, J.L. Aragón, J.M. Parcerisa y A. González. "DTM-NUCA: Dynamic Texture Mapping-NUCA for Energy-Efficient Graphics Rendering". *Proc. of the 30th IEEE Euromicro International Conference on Parallel, Distributed and Network-based Computing* **(PDP)**, Valladolid, Spain, pp. 144-151, Marzo de 2022.

- J. Ortiz-Escribano, D. Corbalán-Navarro, J.L. Aragón y A. González. "MEGsim: A Novel Methodology for Efficient Simulation of Graphics Workloads in GPUs". *Proc. of the 2022 IEEE International Symposium on Performance Analysis of Systems and Software* **(ISPASS)**, Singapore, Singapore, pp. 69-78, Mayo de 2022.

- *En proceso de revisión*: D. Corbalán-Navarro, J.L. Aragón, J.M. Parcerisa y A. González. "A Novel NUCA Organization for Dynamic Texture Mapping in Mobile GPUs". *Microprocessors and Microsystems: Embedded Hardware Design* **(MICPRO)**.

7.2. Vías futuras

Respecto a Ω-Test, como trabajo futuro se pueden considerar otros esquemas de compresión para la Ω-Table, que aumenten la precisión de la misma y reduzcan sus necesidades de almacenamiento. Estos esquemas podrían ser (sin estar limitados a ellos) la cuantización, el *downsampling* o la reducción de la precisión. Otra vía a explorar sería la de tener un delta por *tile* en vez de uno global. Esto permitiría capturar los movimientos de los objetos de manera individual y así aprovechar aún más las bondades del *δ-margin*. Por otro lado, es bastante común que la GPU exponga características especiales a las aplicaciones de modo que éstas puedan hacer uso de ellas en función de sus necesidades. Estas características son, por ejemplo, el uso de *Occlusion Queries*, *Texture Samplers*, *Stencil Tests* o *Variable Shading Rate*. Con este tipo de control, la aplicación podría informar a Ω-Test de las transformaciones matriciales que están sufriendo los objetos de la escena, de modo que Ω-Test podría predecir de manera más precisa las profundidades del siguiente fotograma.

Respecto a Triangle Dropping, y al hilo de lo que se venía comentando anteriormente en Ω-Test, el uso de funciones extendidas por parte de la API gráfica podría ser beneficiosa. Una técnica que se podría implementar para reducir el consumo de memoria *on-chip* por parte de la *Frame Visibility Table*, y sin limitarla a un número fijo de vértices, sería ofrecer una entrada de la API que diera información de visibilidad por comando dibujado. De este modo, el programador, al comienzo

de cada fotograma podría preguntar sobre la visibilidad de un objeto (o varios que estime importantes) en el fotograma anterior, y en función de ésta decidir si dibujar el comando o no. El programador conoce los movimientos que van a hacer los objetos, por lo que puede eliminar de una manera mucho más precisa objetos ocluidos sin afectar a la imagen final. Así pues, el programador también podría decidir qué objetos van a ser eliminados y cuáles no, en función de su criticidad, consumo de recursos y otros criterios que estimase oportunos.

Respecto a DTM-NUCA, se ha visto que usa un tamaño fijo de época en función de un estudio experimental. En este sentido, DTM-NUCA podría determinar la longitud de época de manera dinámica en función de ciertas métricas como, aparte del número de accesos, el número de cambios de afinidad, historia de la afinidad de *buckets* de cada *Fragment Processor*, historia de accesos locales por *bucket*, etc. Por otro lado, se podrían usar otras funciones de mapeo dinámico que no fuese por *buckets*. Por ejemplo, en vez de asignar un *bucket* entero a un *Fragment Processor*, lo cual podría ser muy radical, se podría tener en la *Affinity Table* una lista acotada de bloques de texturas que más afinidad tengan con cada *Fragment Processor*.

Centrándonos de manera general en la coherencia entre fotogramas, como vía futura se podría estudiar un mecanismo para capturar el patrón de accesos a texturas en un fotograma para hacer una planificación más eficiente de los fragmentos en el fotograma siguiente, de modo que la caché de texturas sea más eficiente en términos de tasa de acierto, evitando así accesos a los niveles superiores de la jerarquía de memoria. Otra vía futura en este sentido podría ser el diseño de un algoritmo de reemplazo de la caché de texturas que actuase en función de la frecuencia de acceso de ciertos bloques de texturas en el fotograma anterior. De este modo, se podría diseñar una política de reemplazo que diera más prioridad a los bloques que se sabe que se van a usar más, todo esto tomando como base la historia de los bloques de texturas en el fotograma previo.

Bibliografía

[1] Photo-realistic deferred lighting. https://www.beyond3d.com/content/articles/19/, accedido en marzo de 2021.

[2] Bryan D Agkland and Neil H Weste. The edge flag algorithm—a fill method for raster scan displays. *IEEE Transactions on Computers*, 100(1):41–48, 1981.

[3] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2019.

[4] Andrea Alaimo, V Artale, C Milazzo, and Angela Ricciardello. Comparison between euler and quaternion parametrization in uav dynamics. In *AIP Conference Proceedings*, volume 1558, pages 1228–1231, 2013.

[5] Maria-Elena Algorri and Francis Schmitt. Mesh simplification. In *Computer Graphics Forum*, volume 15, pages 77–86, 1996.

[6] Magnus Andersson, Jon Hasselgren, and Tomas Akenine-Möller. Masked depth culling for graphics hardware. *ACM Transactions on Graphics*, 34(6):1–9, 2015.

[7] Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan L Aragón, and Antonio González. Early visibility resolution for removing ineffectual computations in the graphics pipeline. In *2019 IEEE International Symposium on High Performance Computer Architecture*, pages 635–646, 2019.

[8] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems. In *Proceedings of the 27th International ACM Conference on Supercomputing*, pages 37–46, 2013.

[9] Jose Maria Arnau Montañés. *Energy-efficient mobile GPU systems*. PhD thesis, Universitat Politècnica de Catalunya, 2014.

[10] Anuj Arora, Mayur Harne, Hameedah Sultan, Akriti Bagaria, and Smruti R Sarangi. Fp-nuca: A fast noc layer for implementing large nuca caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(9):2465–2478, 2014.

[11] Alessandro Bardine, Manuel Comparetti, Pierfrancesco Foglia, Giacomo Gabrielli, and Cosimo Prete. Way adaptable d-nuca caches. *International Journal of High Performance Systems Architecture*, 2(3-4):215–228, 2010.

[12] Alessandro Bardine, Manuel Comparetti, Pierfrancesco Foglia, Giacomo Gabrielli, and Cosimo Antonio Prete. A power-efficient migration mechanism for d-nuca caches. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 598–601, 2009.

[13] Jack Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM*, 20(2):100–106, 1977.

[14] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[15] Jack E Bresenham. Incremental line compaction. *The Computer Journal*, 25(1):116–120, 1982.

[16] Zeshan Chishti, Michael D Powell, and TN Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, 2003.

[17] Paolo Cignoni, Claudio Montani, and Roberto Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

[18] David Corbalan-Navarro, Juan L Aragón, Marti Anglada, Enrique De Lucas, Joan-Manuel Parcerisa, and Antonio Gonzalez. Omega-test: A predictive early-z culling to improve the graphics pipeline energy-efficiency. *IEEE transactions on visualization and computer graphics*, 28(12):4375–4388, 2021.

[19] David Corbalán-Navarro, Juan L Aragón, Martí Anglada, Joan-Manuel Parcerisa, and Antonio González. Triangle dropping: an occluded-geometry predictor for energy-efficient mobile gpus. *ACM Transactions on Architecture and Code Optimization*, 19(3):1–20, 2022.

[20] David Corbalán-Navarro, Juan L Aragón, Joan-Manuel Parcerisa, and Antonio González. Dtm-nuca: dynamic texture mapping-nuca for energy-efficient graphics rendering. In *30th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 144–151, 2022.

[21] David Corbalán Navarro, Juan Luis Aragón, Martí Anglada Sánchez, Enrique de Lucas Casamayor, Joan Manuel Parcerisa Bundó, and Antonio María González Colás. Improving the energy efficiency of the graphics pipeline by reducing overshading. In *Avances en arquitectura y tecnología de computadores: Actas de las Jornadas SARTECO 20/21: Málaga, 21 a 24 de Septiembre*, pages 125–134, 2021.

[22] Enrique De Lucas. *Reducing redundancy of real time computer graphics in mobile systems*. PhD thesis, Universitat Politècnica de Catalunya, 2018.

[23] Enrique De Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González. Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):473–485, 2018.

[24] Christopher DeCoro and Natalya Tatarchuk. Real-time mesh simplification using the gpu. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 161–166, 2007.

[25] Susan E Dorward. A survey of object-space hidden surface removal. *International Journal of Computational Geometry & Applications*, 4(03):325–362, 1994.

[26] Haakon Dybdahl and Per Stenstrom. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *13th International Symposium on High Performance Computer Architecture*, pages 2–12, 2007.

[27] Jihad El-Sana, Neta Sokolovsky, and Cláudio T Silva. Integrating occlusion culling with view-dependent rendering. In *Proceedings Visualization, 2001. VIS'01.*, pages 371–575, 2001.

[28] I Ernst, H Rüsseler, H Schulz, and O Wittig. Gouraud bump mapping. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 47–53, 1998.

[29] Jeremy R Flynn, Steve Ward, Julian Abich, and David Poole. Image quality assessment using the ssim and the just noticeable difference paradigm. In *International Conference on Engineering Psychology and Cognitive Ergonomics*, pages 23–30, 2013.

[30] Freedesktop. Gallium3d. https://www.freedesktop.org/wiki/Software/gallium, accedido en julio de 2021.

[31] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *ACM Siggraph Computer Graphics*, 23(3):79–88, 1989.

[32] Xinbo Gao, Wen Lu, Dacheng Tao, and Xuelong Li. Image quality assessment based on multiscale geometric analysis. *IEEE Transactions on Image Processing*, 18(7):1409–1423, 2009.

[33] Google. Android sdk. https://developer.android.com/studio, accedido en julio de 2021.

[34] Google. Gapid. https://developers.google.com/vr/develop/unity/gapid, accedido en julio de 2021.

[35] Google. Google play. https://play.google.com, accedido en julio de 2021.

[36] Ned Greene. Hierarchical polygon tiling with coverage masks. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 65–74, 1996.

[37] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, 1993.

[38] Günther Greiner and Kai Hormann. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics*, 17(2):71–83, 1998.

[39] Eric Haines and Steven Worley. Fast, low memory z-buffering when performing medium-quality rendering. *J. Graph. Tools*, 1(3):1–6, febrero 1996.

[40] Bernd Hamann. A data reduction scheme for triangulated surfaces. *Computer aided geometric design*, 11(2):197–214, 1994.

[41] William Rowan Hamilton. On quaternions; or on a new system of imaginaries in algebra. *Philosophical Magazine*, 25(3):489–495, 1844.

[42] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. R-nuca: Data placement in distributed shared caches. *Computer Architecture Lab at Carnegie Mellon Technical Report*, 2009.

[43] Jon Hjelmervik and Jean-Claude Léon. Gpu-accelerated shape simplification for mechanical-based applications. In *IEEE International Conference on Shape Modeling and Applications*, pages 91–102, 2007.

[44] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26, 1993.

[45] Anwen Huang, Jun Gao, Wei Guo, Wenqiang Shi, Minxuan Zhang, and Jiang Jiang. Psa-nuca: A pressure self-adapting dynamic non-uniform cache architecture. In *7th International Conference on Networking, Architecture, and Storage*, pages 181–188, 2012.

[46] Tom Hudson, Dinesh Manocha, Jonathan Cohen, Ming Lin, Kenneth Hoff, and Hansong Zhang. Accelerated occlusion culling using shadow frusta. In *Proceedings of the 13th annual symposium on Computational geometry*, pages 1–10, 1997.

[47] Mark J Kilgard. A practical and robust bump-mapping technique for today's gpus. In *Game Developers Conference 2000*, pages 1–39, 2000.

[48] Changkyu Kim, Doug Burger, and Stephen W Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 211–222, 2002.

[49] Robert Kooima. Generalized perspective projection. *J. Sch. Electron. Eng. Comput. Sci*, 6, 2009.

[50] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[51] Imagination Technologies Limited. PowerVR Hardware. architecture overview for developers. http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf, accedido en julio de 2021.

[52] Peter Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 259–262, 2000.

[53] Javier Lira, Carlos Molina, Antonio González, et al. Hk-nuca: Boosting data searches in dynamic non-uniform cache architectures for chip multiprocessors. In *IEEE International Parallel & Distributed Processing Symposium*, pages 419–430, 2011.

[54] Haik Lorenz and Jürgen Döllner. Dynamic mesh refinement on gpu using geometry shaders. In *The 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2008.

[55] David P Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, 2001.

[56] Robert Brainerd McMaster. Automated line generalization. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 24(2):74–111, 1987.

[57] Javier Merino, Valentin Puente, and Jose A Gregorio. Esp-nuca: A low-cost adaptive non-uniform cache architecture. In *16th International Symposium on High-Performance Computer Architecture*, pages 1–10, 2010.

[58] Javier Merino, Valentín Puente, Pablo Prieto, and José Ángel Gregorio. Sp-nuca: a cost effective dynamic non-uniform cache architecture. *ACM SIGARCH Computer Architecture News*, 36(2):64–71, 2008.

[59] EEL Mitchell and AE Rogers. Quaternion parameters in the simulation of a spinning rigid body. *Simulation*, 4(6):390–396, 1965.

[60] Steve Morein et al. Ati radeon hyperz technology. http://www.graphicshardware.org/previous/www_2000/presentations/ATIHot3D.pdf, accedido en junio de 2021. Presented at SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware, 2000.

[61] Naveen Muralimanohar and Rajeev Balasubramonian. Interconnect design considerations for large nuca caches. *ACM SIGARCH Computer Architecture News*, 35(2):369–380, 2007.

[62] Kent W Nixon, Xiang Chen, Hucheng Zhou, Yunxin Liu, and Yiran Chen. Mobile GPU power consumption reduction via dynamic resolution and frame rate scaling. In *6th Workshop on Power-Aware Computing and Systems (HotPower)*, 2014.

[63] Jorge Ortiz, Juan L Aragón, and Antonio González. Megsim: A novel methodology for efficient simulation of graphics workloads in gpus. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 69–78, 2022.

[64] Alexandros Papageorgiou and Nikos Platis. Triangular mesh simplification on the gpu. *The Visual Computer*, 31(2):235–244, 2015.

[65] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *51st ACM/EDAC/IEEE Design Automation Conference*, pages 1–6, 2014.

[66] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 303–306, 1997.

[67] Chao Peng and Yong Cao. A gpu-based approach for massive model rendering with frame-to-frame coherence. In *31st Computer Graphics Forum*, number 2pt2, pages 393–402, 2012.

[68] Logah Perumal. Quaternion and its application in rotation using sets of regions. *International Journal of Engineering and Technology Innovation*, 1(1):35, 2011.

[69] Juan Pineda. A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 17–20, 1988.

[70] MBX PowerVR. Imagination technology.

[71] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jorg Henkel. Neural network-based performance prediction for task migration on s-nuca many-cores. *IEEE Transactions on Computers*, 2020.

[72] Boris A Rosenfeld. *A history of non-Euclidean geometry: Evolution of the concept of a geometric space*, volume 12. Springer Science & Business Media, 2012.

[73] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011.

[74] Jarek Rossignac. Geometric simplification and compression, 1997.

[75] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in computer graphics*, pages 455–465. 1993.

[76] William J Schroeder, Jonathan A Zarge, and William E Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 65–70, 1992.

[77] Graham Sellers and John Kessenich. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.

[78] Dave Shreiner. Opengl® programming guide 7th edition, 2010.

[79] Ivan E Sutherland and Gary W Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[80] Imagination Technologies. Introduction to powervr for developers. https://imagination-technologies-cloudfront-assets.s3.eu-west-1.amazonaws.com/website-files/documents/Introduction_to_PowerVR_for_Developers.pdf?dlm-dp-dl-force=1&dlm-dp-dl-nonce=5021498b5e, accedido en julio de 2022.

[81] Nicolas Thibieroz and W Engel. Deferred shading with multiple render targets. *Shader X*, 2:251–251, 2004.

[82] Bala R Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[83] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[84] Zhou Wang, Ligang Lu, and Alan C Bovik. Video quality assessment based on structural distortion measurement. *Signal processing: Image communication*, 19(2):121–132, 2004.

[85] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. *ACM SIGGRAPH computer graphics*, 11(2):214–222, 1977.

[86] Michael Wimmer and Jiří Bittner. Hardware occlusion queries made useful. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. marzo 2005.

[87] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E Hoff III. Visibility culling using hierarchical occlusion maps. In *Proc. of the 24th annual conference on Computer graphics and interactive techniques*, 1997.