Títol:	Caracterización e implementación de algoritmos de compresión en la GPU ATILA.
Volum:	1 de 1
Alumne:	Christian Pérez Llamas
Director/Ponent:	Agustín Fernandez Jiménez
Departament:	Arquitectura de Computadors
Data:	31 de gener de 2008

DADES DEL PROJECTE

Títol del Projecte: Caracterización e implementación de algoritmos de compresión en la GPU ATILA.

Nom de l'estudiant: Christian Pérez Llamas
Titulació: Enginyeria en Informàtica
Crèdits: 37,5
Director/Ponent: Agustín Fernandez Jiménez
Departament: Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Roger Espasa Sans

*Vocal: Francesc P*rats Duaygues

Secretari: Agustín Fernandez Jiménez

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Índice de contenido

1 Introducción 1

1.1 Objetivos 3

1.2 Organización de la memoria 4

2 Conceptos previos 7

2.1 Introducción a la renderización 7

2.1.1 Etapa de geometría 10

Transformaciones del modelo y de la vista 10

Iluminación 12

Proyección 14

Recortado (clipping) 15

Adaptación a las coordenadas de ventana (screen mapping) 16

2.1.2 Etapa de rasterización 17

2.2 ATILA: Simulador de GPU 19

2.2.1 Arquitectura 20

2.2.2 Entorno de trabajo 23

2.2.3 Configuración del simulador 24

3 Antecedentes 27

3.1 Anti-aliasing 27

3.1.1 Aliasing y muestreo 28

Bordes en diente de sierra (jaggies) 30

3.1.2 Reconstrucción 31

3.1.3 Métodos anti-aliasing 34

Prefiltrado Infinitas muestras por pixel 35

Sin filtrado Una muestra por pixel 36

Postfiltrado n muestras uniformes por pixel 36

Postfiltrado muestras estocásticas 39

3.2 Compresión de los buffers de profundidad y color en la GPU 39

3.2.1 Arquitectura básica del buffer de profundidad 40

3.2.2 Compresión del buffer de profundidad 42

Fast z-clear 43

Differential Differential Pulse Code Modulation (DDPCM) 43
Anchor encoding 44
Plane encoding 45
Depth offset compression 45
3.2.3 Arquitectura del buffer de color 46
3.2.4 Compresión del buffer de color 47
Multi-sampling compression 47
Color plane compression 48
Offset compression 48
Entropy coded pixel differences 49
4 Desarrollo y resultados 51
4.1 Entorno de trabajo 52
4.1.1 ATILA 52
4.1.2 Cluster 54
4.1.3 Procesado y visualización de resultados. 54
4.2 Trazas y configuración utilizadas 55
Trazas de ejecución 55
Parámetros de configuración del simulador 56
4.3 Analisis de tráfico de memoria 57
4.4 Requisitos de los algoritmos de compresión 59
4.5 Algoritmos de compresión evaluados 60
4.5.1 Offset Compression 62
hilo 62
hilore 68
hilorebi 70
hilore4ref 71
comp 73
4.5.2 Multi-sampling compression 74
msaa 74
4.6 Análisis de los algoritmos de compresión 78
4.7 Resultados comparativos 87
4.8 Implementación en el simulador 90
5 Planificación temporal 93

6 Valoración económica **95**
6.1 Análisis del tiempo de realización del proyecto 95
6.2 Coste económico de proyecto 96
7 Conclusiones **99**
7.1 Futuras lineas de trabajo 100
8 Bibliografía **101**
9 Apéndice **103**
9.1 Organización del CD-ROM 103
9.2 Archivo de configuración común para todos los experimentos 104

Índice de ilustraciones

Ilustración 2.1 9
Ilustración 2.2 10
Ilustración 2.3 11
Ilustración 2.4 12
Ilustración 2.5 13
Ilustración 2.6 14
Ilustración 2.7 15
Ilustración 2.8 16
Ilustración 2.9 16
Ilustración 2.10 17
Ilustración 2.11 19
Ilustración 2.12 21
Ilustración 2.13 23
Ilustración 3.1 28
Ilustración 3.2 29
Ilustración 3.3 30
Ilustración 3.4 31
Ilustración 3.5 32
Ilustración 3.6 32
Ilustración 3.7 33
Ilustración 3.8 34
Ilustración 3.9 34
Ilustración 3.10 36
Ilustración 3.11 38
Ilustración 3.12 39
Ilustración 3.13 41
Ilustración 3.14 44
Ilustración 3.15 45
Ilustración 3.16 46
Ilustración 3.17 48
Ilustración 3.18 49

Ilustración 3.19 49
Ilustración 4.1 53
Ilustración 4.2 59
Ilustración 4.3 61
Ilustración 4.4 62
Ilustración 4.5 63
Ilustración 4.6 64
Ilustración 4.7 65
Ilustración 4.8 66
Ilustración 4.9 67
Ilustración 4.10 68
Ilustración 4.11 68
Ilustración 4.12 70
Ilustración 4.13 72
Ilustración 4.14 73
Ilustración 4.15 74
Ilustración 4.16 75
Ilustración 4.17 76
Ilustración 4.18 77
Ilustración 4.19 78
Ilustración 4.20 88
Ilustración 4.21 92
Ilustración 5.1 93

Índice de tablas

Tabla 4.1 56
Tabla 4.2 57
Tabla 4.3 77
Tabla 4.4 79
Tabla 4.5 79
Tabla 4.6 88
Tabla 4.7 90
Tabla 6.1 95

1 Introducción

En 1996 salió al mercado de consumo la que se considera la primera tarjeta gráfica con aceleración 3D real. Desde entonces y hasta hoy las tarjetas gráficas no han dejado de evolucionar y de incorporar nuevas características.

Así como los procesadores de propósito general, siguiendo la Ley de Moore, han visto incrementado su rendimiento del orden de dos veces cada año y medio, en el caso de los procesadores gráficos (GPU) el incremento es mayor, del orden de dos veces cada año.

Por otro lado, no hemos visto lo mismo para el caso de la memoria. El incremento de rendimiento de la misma no sigue la misma progresión que las GPUs. La GPU necesita acceder a memoria, pero debido a la brecha de rendimiento existente entre ambas, la memoria acaba siendo el cuello de botella de la GPU.

Una de las mejoras más importantes que han incorporado las GPUs en los últimos tiempos tienen que ver con la mejora de la calidad de la imagen. Para ello se han incorporado técnicas de anti-aliasing que van dirigidas a paliar los artefactos que aparecen en la imagen como resultado de representar de forma discreta una escena que está modelada en un espacio continuo.

El multi-sampling es una de esas técnicas, y básicamente, consiste en utilizar varias muestras por cada fragmento de un objeto representable. El hecho de utilizar más muestras implica manejar mayor cantidad de memoria en ciertas etapas del proceso de trabajo de la GPU.

Por tanto nos encontramos con el problema de que, no solo la memoria no se equipara en rendimiento a la GPU, sino que además la GPU incrementa su necesidad de acceder a memoria.

Para disminuir la cantidad de información que circula entre la GPU y la memoria se pueden emplear técnicas de compresión de datos.

El acceso a la información de profundidad (necesario para gestionar la visibilidad de los objetos de una escena) y a la información de color de la imagen que se está generando, representan entre un 20 y un 40 % del tráfico total.

En las GPUs actuales ya hace algún tiempo que se han incorporado técnicas de compresión de la información de profundidad y más recientemente de la información de color.

En este proyecto se va a constatar dicho incremento en la necesidad de acceder a memoria para las diferentes configuraciones de multi-sampling, y se van a evaluar un conjunto de algoritmos de compresión basados en la literatura existente y en modificaciones de los mismos diseñadas por nosotros.

Para ello se utilizará un simulador de GPU llamado ATILA que ha sido desarrollado por un grupo de investigación del Departamento de Arquitectura de Computadores.

Además se implementará un algoritmo de compresión en el simulador, que será escogido en base a los resultados obtenidos tras la evaluación.

1.1 Objetivos

A continuación se enumeran los objetivos que se pretenden alcanzar con la realización de este proyecto:

- Comprender la organización y el funcionamiento global de los sistemas software y hardware de generación de gráficos 3D.

- Conocer la arquitectura hardware de la GPU ATILA.

- Entender los problemas relacionados con la calidad de la imagen y los métodos disponibles para mejorarla en los sistemas gráficos actuales.

- Identificar y comprender las implicaciones que tiene usar técnicas de mejora de la imagen en cuanto a la cantidad de recursos necesarios en los sistemas hardware de generación de gráficos.

- Confirmar la necesidad de emplear compresión de datos para la comunicación de la GPU con la memoria externa.

- Buscar y evaluar algoritmos de compresión y/o proponer otros nuevos susceptibles de ser utilizados con los datos de profundidad y color.

- Implementar una solución de compresión en el simulador de GPU ATILA.

1.2 Organización de la memoria

A continuación se resumirá brevemente cual es el contenido de este documento y como se organiza por apartados.

En el apartado 2 se introducirán los conceptos básicos necesarios para entender los temas que se tratarán en el proyecto. Se introducirá el proceso de renderización y se describirá el simulador de GPU ATILA.

En el apartado 3 se expone el estudio realizado para comprender el problema tratado en el proyecto y los antecedentes encontrados para su solución. Se hablará de los métodos anti-aliasing para mejorar la calidad de la imagen y de las técnicas de compresión que pueden ser utilizados en las GPUs para disminuir el ancho de banda.

En el apartado 4 se describirá el trabajo llevado a cabo y los resultados obtenidos. Se explicará el entorno de trabajo, los algoritmos que serán evaluados, los resultados de los análisis de tráfico de memoria, y los resultados del análisis de los algoritmos de compresión. También se justificará la elección e implementación de uno de ellos en el simulador.

En el apartado 5 se muestra el diagrama de Gantt con las tareas planificadas y su distribución en el tiempo previsto para el proyecto.

En el apartado 6 se presentará la valoración económica del proyecto que incluye el análisis del tiempo de realización del proyecto y el coste económico.

Finalmente en el apartado 7 se presentan las conclusiones del trabajo realizado y se exponen las posibles lineas de trabajo futuro para este proyecto.

En los apartados 8 y 9 encontraremos la bibliografía y el apéndice respectivamente.

2 Conceptos previos

En este apartado se introducirán los conceptos básicos necesarios para entender los temas que se tratarán en los siguientes apartados.

Se empieza introduciendo el proceso de dibujado de las escenas (renderización) que se lleva a cabo, en términos generales, en los procesadores gráficos y que se puede dividir en dos etapas, la etapa de geometría y la de rasterización.

A continuación se describe el simulador de GPU ATILA, utilizado como base para comprender la arquitectura interna de una GPU real, y para entender los procesos de trabajo que se han llevado a cabo en este proyecto.

2.1 Introducción a la renderización

En el campo de los gráficos 3D históricamente han evolucionado dos aproximaciones distintas de visualización de escenas 3D con dos objetivos básicos fundamentales: conseguir una visualización lo más realista posible en el menor tiempo de cálculo.

La primera aproximación es la *visualización fotorealista* (Photo realistic rendering), basada principalmente en simular un modelo complejo de iluminación aproximando las interacciones de luz reales. En el modelo de iluminación fotorealista, para calcular el color de cada punto de la superficie de la escena, se tiene en cuenta no solamente las iluminación producida por las fuentes de luz, sino también las contribuciones de luz u oclusiones de luz (sombras) de otros objetos de

la escena. A este modelo se le llama modelo de iluminación global, y se caracteriza porque el cálculo es recursivo y bastante costoso, ya que cada punto de la superficie necesita conocer las contribuciones de luz de otros puntos de la superficie previamente calculados. Existen diferentes algoritmos que aproximan a una solución razonable en coste, entre los más conocidos: Ray Tracing, Radiosity o Photon Map.

La alternativa rápida a esta visualización la *visualización interactiva directa* (Direct rendering) que utiliza un modelo simplificado de iluminación en el que solamente se tiene en cuenta para cada punto de la superficie de la escena la iluminación directa desde las fuentes de luz. A este modelo se le llama modelo de iluminación local. Además de simplificar enormemente los cálculos, se introduce una ventaja muy importante, y es que como el color de cada punto de una superficie puede ser calculado independientemente del color del resto de puntos de la escena, desaparece la recursividad y la dependencia entre los puntos. De esta forma el cálculo de la iluminación se convierte en 100% paralelizable, lo que da la posibilidad de ser implementado eficientemente en hardware paralelo. Con iluminación local perdemos realismo en la visualización (perdemos las sombras entre objetos y realismo en la iluminación) pero ganamos mucha velocidad, y hacemos que la visualización 3D pueda ser interactiva y en tiempo real.

Llamamos visualización en tiempo real a aquella en que podemos generar un número aceptable de imágenes por segundo. Se considera un número aceptable entre 30 y 60 imágenes por segundo para poder interactuar con la escena 3D de una forma cómoda. Actualmente sólo disponemos de hardware gráfico para

conseguir visualización interactiva con Direct Rendering mientras que Photorealistic Rendering se utiliza en otras aplicaciones que no requieren interactividad pero sí una imagen final más realista, como por ejemplo, el desarrollo de fotogramas para películas de animación 3D o la visualización final de prototipos visuales de vehículos.

Direct rendering es la aproximación que utilizan las tarjetas gráficas 3D. A partir de ahora nos referiremos a él simplemente como renderización.

El proceso de renderización se puede dividir en dos grandes etapas tal y como se muestra en la Ilustración 2.1: la etapa de geometría y la etapa de rasterización. A su vez cada una de estas etapas abarcan una serie de procesos que serán explicados con más detalle a continuación.

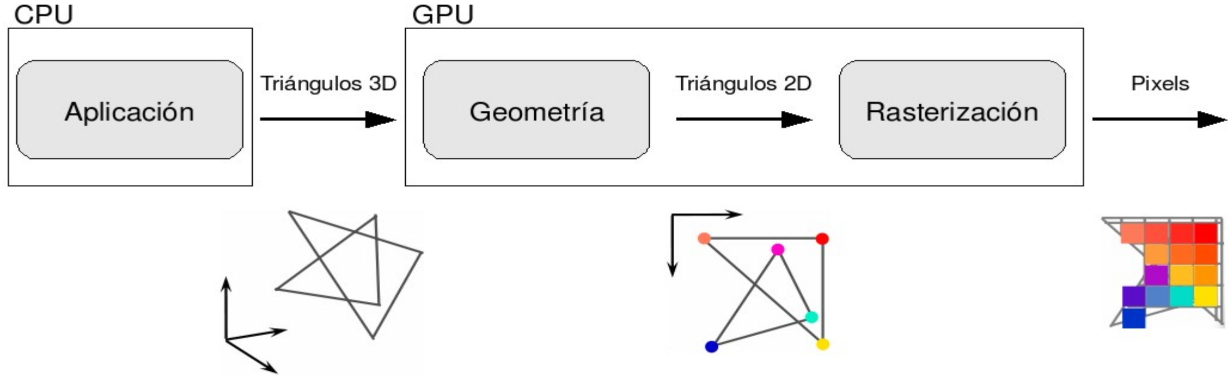


Ilustración 2.1: Etapas del proceso de renderización.

2.1.1 Etapa de geometría

La etapa de geometría es responsable de las operaciones realizadas sobre los vértices de los polígonos renderizados. Esta etapa se puede dividir en subetapas funcionales tal y como se muestra en la Ilustración 2.2.

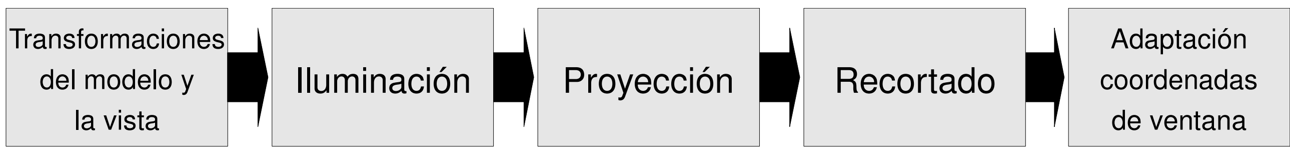


Ilustración 2.2: Subdivisión de la etapa de geometría.

Transformaciones del modelo y de la vista

De camino hacia la pantalla, un modelo es transformado hacia diferentes *espacios de coordenadas* o *sistemas de coordenadas*. Originalmente, un modelo reside en su propio espacio de coordenadas (llamado *espacio de modelo*), que significa que no ha sido transformado todavía. Cada modelo puede tener una transformación propia asociada de tal forma que pueda ser situado y orientado. Es posible asociar diferentes transformaciones a un mismo modelo. Esto permite tener varias copias (llamadas instancias) del mismo modelo con diferentes posiciones, orientaciones, y tamaños en la misma escena sin necesitar replicar la información de geometría para cada una.

Los vértices y las normales del modelo son transformados usando *transformaciones del modelo*. Las coordenadas de un objeto se llaman *coordenadas del modelo*, y después de haberles aplicado dichas transformaciones se dice que el modelo esta situado en *coordenadas de mundo* o *espacio de mundo* (véase la Ilustración 2.3). El espacio de mundo es único, y después de que los modelos hayan

sido transformados con sus respectivas transformaciones todos ellos residen en el mismo espacio.

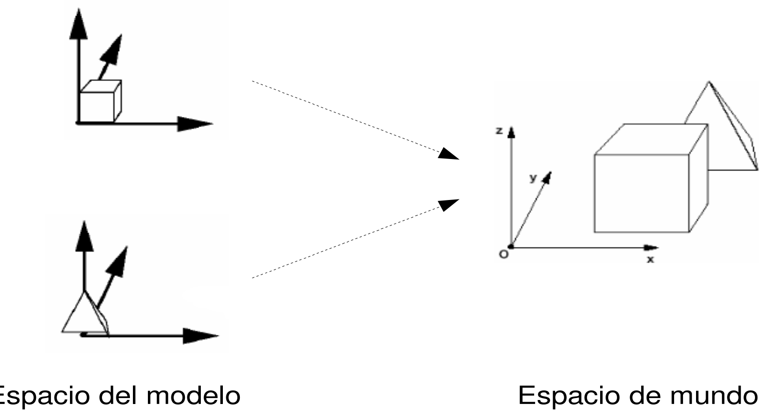


Ilustración 2.3: Transformación de los modelos a espacio de mundo.

Tan solo los modelos que la cámara (observador) puede ver son renderizados. La cámara tiene su posición en el espacio de mundo y una dirección, ambos permiten situar y orientar la cámara con el fin de definir que parte de la escena será observada.

La cámara y los modelos son transformados con la *transformación de vista*, cuyo objetivo es situar la cámara en el origen de coordenadas y orientarla para que quede en la dirección del eje negativo de la z, con el eje y apuntando hacia arriba y el eje x hacia la derecha.

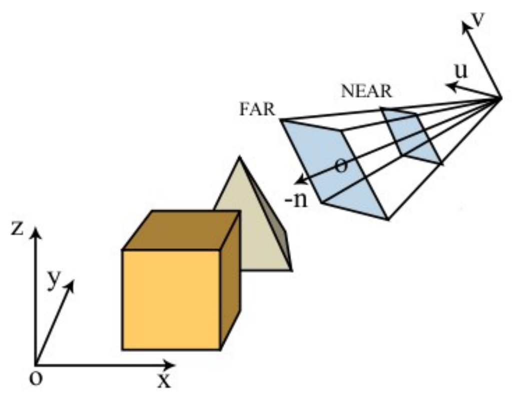


Ilustración 2.4: Transformación de vista.

El nuevo espacio de coordenadas en el que residirá, no solo la cámara sino también los modelos transformados, se llama *espacio de cámara* o *espacio de ojo*.

Las transformaciones se implementan como matrices de 4x4, y varias transformaciones se pueden concatenar en una sola mediante el producto de sus matrices resultando en una nueva matriz.

Iluminación

Con el fin de que los modelos tengan una apariencia más realista, se pueden poner luces en la escena. Los modelos geométricos pueden tener también un color asociado con cada vértice o una parte de una imagen (*textura*) asociada a él.

Para los modelos afectados por fuentes de iluminación, se usa una ecuación de iluminación para calcular el color de cada vértice del modelo. Esta ecuación se encarga de aproximar la interacción en el mundo real entre los fotones y las superficies. En el mundo real los fotones son emitidos por las fuentes de luz y son reflejados o absorbidos por las superficies. En gráficos en tiempo real, no se puede gastar mucho tiempo simulando dicho fenómeno (reflexiones verdaderas y

sombras, por ejemplo, no son contempladas por la ecuación de iluminación comentada).

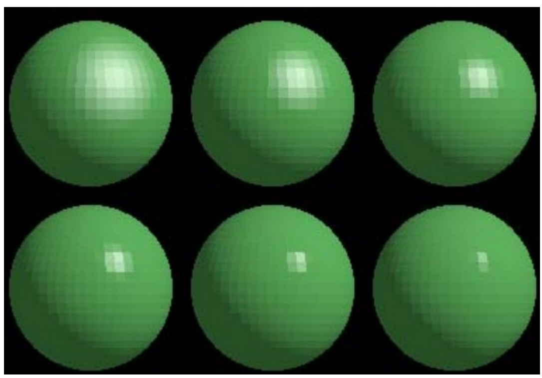


Ilustración 2.5: Esferas renderizadas utilizando diferentes parámetros de iluminación.

Los modelos son representados normalmente con triángulos, ya que son las primitivas gráficas que la mayor parte del hardware gráfico soporta de forma nativa.

El color en cada vértice de la superficie se calcula a partir de la posición de las fuentes de la luz y sus propiedades, la posición y la normal del vértice, y las propiedades del material al que pertenece el vértice (véase la Ilustración 2.5 para un ejemplo).

Los colores del interior del triángulo son interpolados a partir de los que se calcularon para sus vértices, es lo que se conoce como interpolación de *Gourand*.

Proyección

Después del proceso de iluminación viene la transformación de proyección, que transforma el volumen de visión a un cubo unitario con los extremos situados en (-1, -1, -1) y (1, 1, 1). Dicho cubo unitario es conocido como el *volumen de visión canónico*.

La proyección se puede realizar de dos maneras básicamente, con el método *ortogonal* o con la proyección de *perspectiva* (véase Ilustración 2.6).

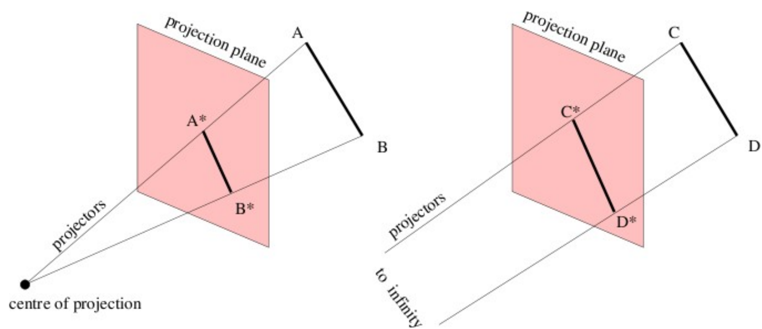


Ilustración 2.6: En la izquierda proyección de perspectiva y en la derecha proyección ortogonal.

El volumen de visión ortogonal es una caja rectangular normalmente. La proyección ortogonal transforma dicho volumen de visión al cubo unitario. La característica principal de la proyección ortogonal es que las lineas paralelas siguen siendo paralelas después de la transformación, que es en realidad una combinación de una translación y un escalado.

La proyección de perspectiva es más compleja, en ella los objetos que se muestran a mayor distancia de la cámara aparecen más pequeños cuanto mayor es esta distancia. Además las lineas paralelas pueden converger en el horizonte.

La transformación de perspectiva simula la manera en que los humanos percibimos el tamaño de los objetos.

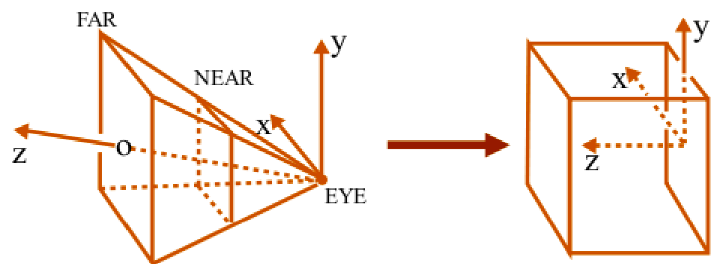


Ilustración 2.7: Transformación a coordenadas normalizadas de dispositivo.

Geométricamente hablando el volumen de visión se llama *frustrum* y es una pirámide recortada con base rectangular. El frustrum acaba siendo transformado a un cubo unitario después de la proyección (véase la Ilustración 2.7). Tanto la transformación de perspectiva ortogonal como la de perspectiva se pueden implementar mediante matrices de 4x4 y después de aplicar dicha transformación se dice que los modelos están en *coordenadas normalizadas del dispositivo*.

Recortado (clipping)

Tan solo las primitivas que están dentro del volumen de visión completa o parcialmente necesitan pasar a la etapa de rasterización, donde será dibujada en pantalla. Las primitivas que quedan totalmente dentro del volumen son pasadas a la siguiente etapa tal cual, y las que están totalmente fuera son descartadas. En el caso de las primitivas que queden parcialmente dentro del volumen de visión será necesario recortar las porciones de dicha primitiva que quedan fuera.

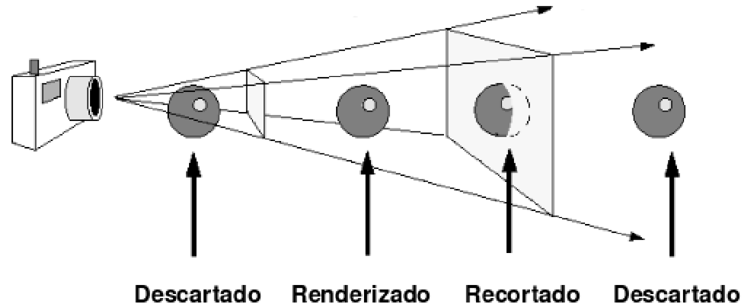


Ilustración 2.8: Ejemplos de objetos descartados, renderizados y recortados.

Adaptación a las coordenadas de ventana (screen mapping)

Cuando las primitivas (convenientemente recortadas) son pasadas a esta etapa las coordenadas todavía son tridimensionales. Las coordenadas x e y de cada primitiva son transformadas a *coordenadas de pantalla*. Las coordenadas de pantalla junto a la coordenada z se llaman *coordenadas de ventana*.

Por ejemplo, si la escena debe ser renderizada en una ventana con las esquinas entre (x_1, y_1) y (x_2, y_2), donde $x_1 < x_2$ y $y_1 < y_2$, entonces la adaptación de coordenadas de ventana es una transformación de translación seguido de un escalado. La coordenada z no se ve afectada por dicha transformación. Las nuevas coordenadas x e y son llamadas coordenadas de pantalla y son pasadas a la etapa de rasterización junto a la coordenada z ($-1 \leqslant z \leqslant 1$).

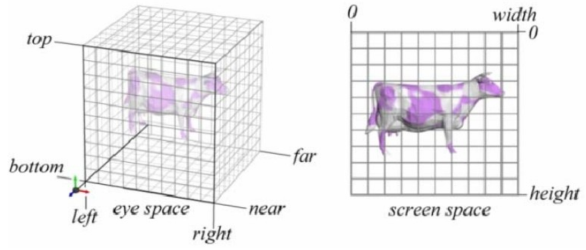
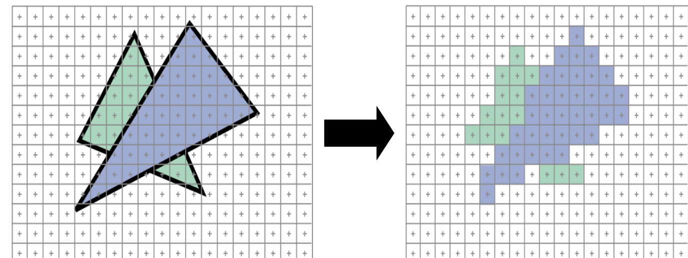


Ilustración 2.9: Adaptación de las coordenadas de pantalla.

2.1.2 Etapa de rasterización

A partir de los vértices transformados y proyectados, los colores y las coordenadas de textura obtenidas de la etapa de geometría el rasterizador deberá asignar el color adecuado a cada uno de los fragmentos que forman la primitiva. Este proceso se llama rasterización y consiste en la conversión de vértices bidimensionales en espacio de pantalla - cada uno con un valor de profundidad (obtenido a partir de la coordenada *z*), uno o dos colores, y posiblemente uno o más conjuntos de coordenadas de textura asociadas con cada vértice en pixeles en la pantalla. En la imagen Ilustración 2.10 se muestra un ejemplo de dos triángulos rasterizados.

Ilustración 2.10: Ejemplo de rasterización de dos polígonos sobre el buffer de color.



A diferencia de la etapa de geometría, que realiza operaciones a nivel de polígonos, la etapa de rasterización realiza operaciones a nivel de fragmentos. La información de cada fragmento es almacenada en el buffer de color, que consiste en una matriz rectangular de colores (con componentes rojo, verde y azul cada uno).

En esta etapa se debe resolver la visibilidad de fragmentos. Esto significa que cuando la escena ha sido renderizada, el buffer de color debe contener el color de las primitivas en la escena que son visibles desde el punto de vista de la cá-

mara. La mayor parte del hardware gráfico lo hace ayudándose de un buffer de profundidad (usando el algoritmo de Z-buffer). El buffer de profundidad es del mismo tamaño y forma que el buffer de color, y para cada fragmento se guarda el valor de la coordenada z, que representa la profundidad desde la cámara hasta la primitiva más cercana.

Esto significa que cuando una primitiva es renderizada en una posición del buffer de color, el valor de la coordenada z del fragmento que está siendo renderizado, es comparado con el contenido del buffer de profundidad correspondiente para dicha posición. Si el nuevo valor de z es menor que el valor del buffer de profundidad entonces la primitiva que está siendo renderizada está más próxima a la cámara que la primitiva que fue renderizada previamente en esa posición del buffer de color. En tal caso el nuevo valor de z sustituye al que había en el buffer de profundidad y el color del fragmento se copia al buffer de color. En caso contrario, si el nuevo valor resultara ser mayor que el valor que había en el buffer de profundidad, el buffer de color y el de profundidad no se actualizarán con los valores del fragmento.

El algoritmo de Z-buffer es muy simple y permite que las primitivas sean renderizadas en cualquier orden. Sin embargo, las primitivas con fragmentos parcialmente transparentes no pueden ser dibujadas en cualquier orden, sino que tienen que hacerlo después de que se hayan rasterizado las primitivas con fragmentos opacos y en orden de mayor a menor distancia con respecto a la cámara.

La *texturación* es una técnica que se usa para incrementar el nivel de rea-

lismo de la escena. Básicamente consiste en pegar una imagen a un objeto. Las imágenes pueden ser de una, dos o tres dimensiones, aunque lo más común es que sea de dos dimensiones. En la Ilustración 2.11 se muestra un ejemplo de texturación de una malla de polígonos que forman una esfera.

Ilustración 2.11: Ejemplo de texturación de una malla de polígonos.

Cuando las primitivas han pasado la etapa de rasterización, aquellas que son visibles desde el punto de vista de la cámara son mostradas en la pantalla.

2.2 ATILA: Simulador de GPU

ATILA es un simulador de la microarquitectura de una *GPU* (Graphics Processor Unit) desarrollado por un grupo de investigación del Departamento de Arquitectura de Computadores de la UPC. La microarquitectura simulada presenta las características más importantes que se pueden encontrar en cualquier GPU actual. El simulador sigue el paradigma de simulación basado en eventos discretos a nivel de ciclos, está implementado en C++ y es lo suficientemente flexible como para incorporar nuevas características de forma modular. Además es altamente configurable lo cual permite evaluar diferentes parámetros de la microarquitectura y permite obtener múltiples estadísticos que pueden ser analizados después de

una simulación.

2.2.1 Arquitectura

ATILA implementa el modelo unificado de *pipeline* gráfica. En dicho modelo las unidades de proceso (*shaders*) pueden ser utilizadas tanto para el procesamiento de vértices (*vertex shaders*) como de fragmentos (*fragment shaders*). Además también puede ser configurado para representar el modelo tradicional de *pipeline* fija en la que existía una separación entre las unidades de proceso de vértices y de fragmentos. Puede consultarse [1] para más información.

La GPU está compuesta por unidades funcionales (véase la Ilustración 2.12) que se describen a continuación:

- **Streamer**: Se encarga de obtener datos de geometría del controlador de memoria, convertirlos al formato interno y pasarlos a los shaders para que se lleve a cabo el procesamiento de los vértices (*vertex shading*)

- **Primitive Assembly**: Recoge los vértices procesados y forma primitivas (de momento tan solo se soportan triángulos).

- **Clipper**: Se encarga de determinar que triángulos quedan fuera o dentro del frustrum de visión y eliminar los que estén completamente fuera.

- **Triangle setup**: Se calculan los planos formados por los bordes del triángulo y los parámetros necesarios para interpolar la profundidad de los fragmentos que forman el triángulo.

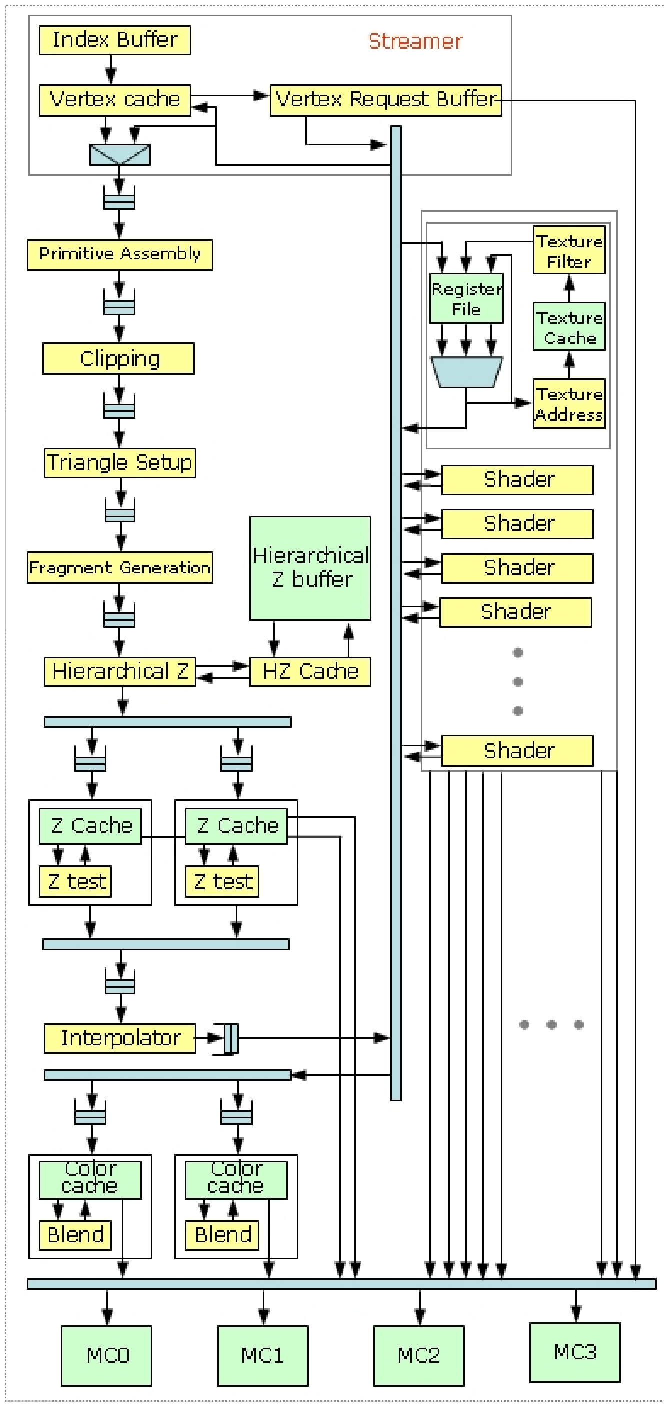


Ilustración 2.12: Esquema de bloques del pipeline gráfico unificado implementado por ATILA.

–**Fragment Generator**: Recorre el área del triángulo y genera fragmentos agrupados en tiles.

–**Hierarchical Z**: Permite detectar los fragmentos que han sido cubiertos por otros ya renderizados antes de que lleguen al test de profundidad y descartarlos de forma temprana. También descarta los fragmentos que fueron marcados como fuera de la ventana de renderizado.

–**Z & Stencil Test**: Recibe los fragmentos en grupos de 2x2 llamados quads (unidad de transmisión utilizado a partir de esta etapa en adelante) y determina si pasan los tests de profundidad y stencil. Emplea una cache con el fin de explotar la localidad en los accesos al buffer de profundidad y stencil.

– **Interpolator**: Se encarga de interpolar los atributos de los fragmentos a

partir de los valores de los vértices del triángulo. Utiliza interpolación lineal con corrección de la perspectiva. A continuación transfiere los quads a los shaders para que sean procesados (*fragment shading*).

- **Blend**: Se encarga de actualizar el buffer de color a partir de los fragmentos procesados. Al igual que la unidad Z & Stencil Test implementa una cache y soporta el borrado rápido.

- **Memory Controller**: Es la unidad encargada de acceder a la memoria de la GPU. Todas las unidades de la GPU que necesitan acceder a memoria lo hacen a través de este controlador. La unidad mínima de acceso a memoria tiene un tamaño de 64 bytes e implementa la especificación GDDR3.

- **Shader**: Son procesadores SIMD (Single Instruction Multiple Data) con registros de 4 elementos. También pueden ejecutar instrucciones escalares. Se encargan de procesar los vértices y los fragmentos dependiendo del programa (llamado kernel) que tenga asignado.

- **Command Processor**: No se muestra en la ilustración pero se encuentra presente en todo el pipeline ya que es la unidad que lo controla todo y recibe y procesa los comandos enviados por la CPU del sistema.

- **Texture Unit**: Hay una por cada shader y se encarga de acceder y filtrar la información de las texturas. Implementa una cache con el fin de disminuir los accesos a memoria a través del Memory Controller.

2.2.2 Entorno de trabajo

El entorno de trabajo con ATILA se divide en cuatro partes, la parte de captura, la de verificación, la de simulación y la de análisis. En la Ilustración 2.13 se muestra un esquema que ayudará a entender los procesos que se explican a continuación.

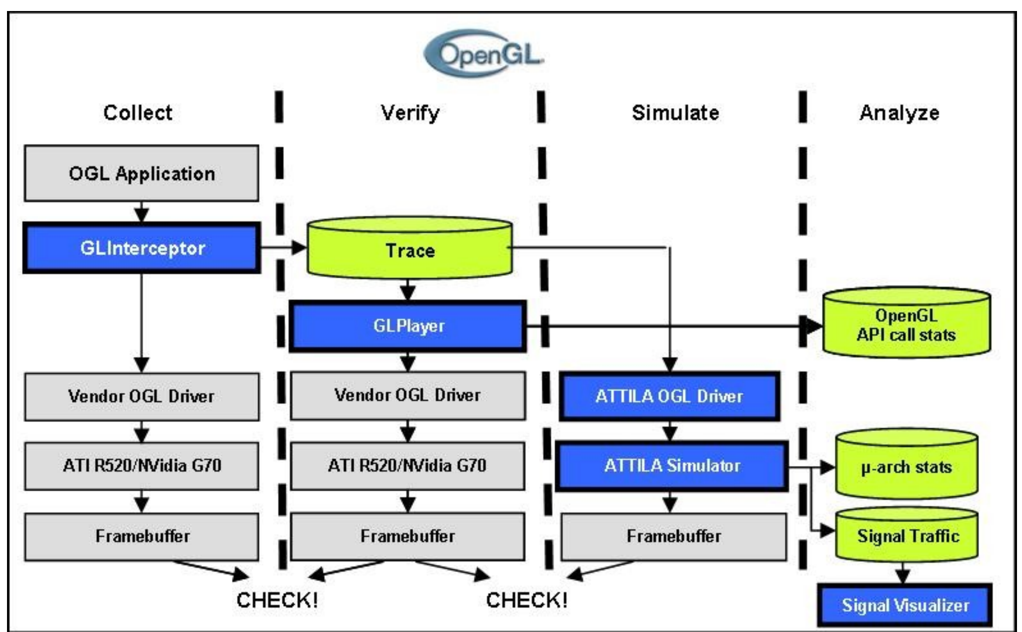


Ilustración 2.13: Escenario de trabajo típico para la captura, verificación, simulación y análisis de trazas con el simulador para el caso de OpenGL.

En la parte de captura partimos de aplicaciones reales que se ejecutan en un sistema real. El objetivo es registrar los datos que circulen entre la aplicación y la API de renderización (*OpenGL* o *DirectX*). Para ello se utiliza un programa (*GLInterceptor* o *DXInterceptor*) que intercepta las operaciones que la aplicación realiza sobre dicha API y que se encarga de registrar en un fichero especial (llamado traza) información sobre que que operaciones se han invocado, con que parámetros y que resultados han generado.

La siguiente parte tiene como finalidad verificar que las trazas obtenidas son correctas y se pueden reproducir de nuevo. Para ello se utiliza una aplicación específica (*GLPlayer* o *DXPlayer*) que es capaz de volver a invocar las operaciones sobre la API de renderización a partir de la información obtenida de la traza. De esta manera podemos comprobar si las imágenes obtenidas coinciden con las originales.

En la siguiente parte está la simulación con la GPU ATILA. En esta ocasión es el simulador quien lee las trazas a través de un driver propio que procesa las operaciones de la traza y las transforma a comandos de bajo nivel que la GPU simulada es capaz de entender. Estos comandos son el equivalente a los comando AGP o PCI Express que generaría un driver real en un sistema real. Cabe destacar que en esta parte se han sustituido la API de renderización y el driver real por uno propio de ATILA, y se ha sustituido la GPU real por el simulador de GPU de ATILA. De la misma manera que hacíamos en la etapa de verificación, podemos comparar las imágenes obtenidas con las que se generaron en la etapa de captura con el fin de verificar que el simulador ha renderizado correctamente.

Además el simulador genera toda una serie de tablas con información sobre su funcionamiento interno que puede ser utilizado para su posterior análisis. Esto se corresponde con la parte de análisis.

2.2.3 Configuración del simulador

Existen multitud de parámetros configurables en el simulador (entorno a 100 parámetros). Para especificar al simulador que valores deben tomar cada uno

de ellos se utiliza un fichero de configuración. Dicho fichero está dividido en secciones y cada sección contiene asignaciones para un conjunto de parámetros relacionados entre sí, por ejemplo la sección de memoria permitirá configurar aspectos como cuanta memoria se debe utilizar, que ancho de bus se utilizará, etc. Se puede consultar la especificación del formato con más detalle en la wiki del proyecto ATILA [2].

A continuación, y a modo de ejemplo, se muestra un pequeño fragmento del mismo:

```
[GPU]

NumVertexShaders = 8
NumFragmentShaders = 4
NumStampPipes = 4

[COMMANDPROCESSOR]

PipelinedBatchRendering = FALSE
```

También se puede utilizar la linea de comandos para especificar algunos de los siguientes parámetros:

- La traza que se utilizará,
- el frame de inicio donde se empezara a renderizar,
- el número de frames que se renderizaran.

3 Antecedentes

Ha sido necesario un exhaustivo trabajo de documentación para poder trabajar en este proyecto. Al tratarse de temas relacionados con la investigación la mayor parte de la información es bastante técnica y procede de libros muy especializados, artículos e incluso patentes.

A continuación se empieza introduciendo un problema muy común relacionado con la calidad de las imágenes renderizadas, el aliasing, y se explica porqué aparece. También se habla de los procedimientos que se pueden llevar a cabo para minimizar su efecto, es lo que se conoce como métodos anti-aliasing. Para cada uno de estos métodos se explicará que técnicas se están usando en sistemas reales.

Se termina el apartado con una revisión de la arquitectura empleada en los sistemas actuales para los buffers de profundidad y color. También se introducen los métodos de compresión más importantes que serán tomados como base para el desarrollo del proyecto.

3.1 Anti-aliasing

En la calidad de la imagen final, resultante de renderizar una escena tridimensional, intervienen varios factores. Uno de ellos está relacionado con el proceso de muestreo de la imagen (véase la Ilustración 3.1). El aliasing es un efecto no deseado que aparece como consecuencia de un muestreo a una frecuencia más baja de la necesaria. Se engloban dentro del anti-aliasing todos aquellos métodos

que van dirigidos a reducir dicho efecto.

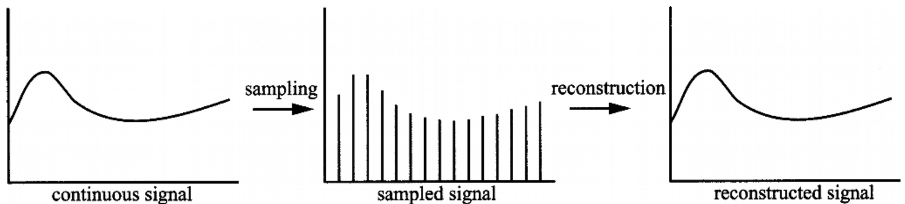


Ilustración 3.1: Procesos de muestreo de una señal y reconstrucción a partir de las muestras.

En este apartado se explica que es el aliasing con más detalle, a que es debido y la teoría relacionada. A continuación se habla sobre el proceso de reconstrucción de la imagen y algunos filtros que se pueden emplear para ello. Finalmente se explican los métodos generales que se pueden usar con el fin de reducir los efectos no deseados del aliasing complementado con ejemplos de algoritmos actuales que los implementan.

La mayor parte de la información se puede complementar con las referencias [3] y [4].

3.1.1 Aliasing y muestreo

El término aliasing se refiere al efecto que se obtiene cuando una señal es muestreada a una frecuencia demasiado baja. El resultado es que la señal obtenida es de menor frecuencia que la original y no permite su reconstrucción de forma unívoca.

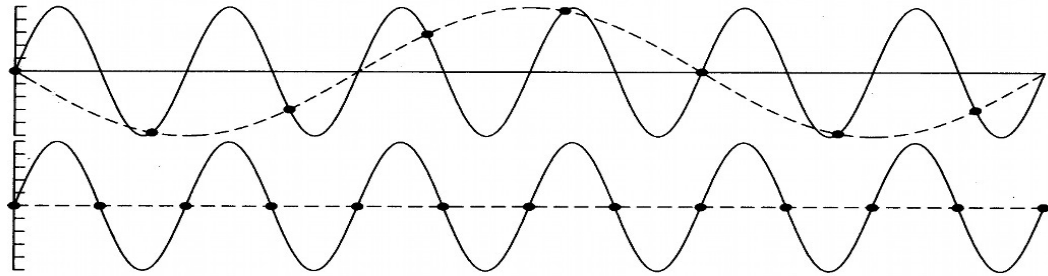


Ilustración 3.2: La linea continua representa la señal original, los puntos las muestras tomadas y la linea discontinua la señal reconstruida.

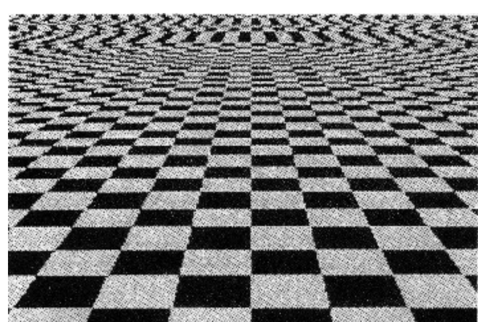
Se pueden ver un par de ejemplos en la Ilustración 3.2 en el que se utiliza una onda sinusoidal como representación de una señal.

Si tomamos muestras de dicha señal con una frecuencia demasiado baja con respecto a la frecuencia de la señal original y reconstruimos una señal a partir de las muestras obtenidas, el resultado es que se trata de una señal distinta a la original.

Para que una señal pueda ser muestreada de forma correcta, y por tanto que la señal original pueda ser reconstruida a partir de las muestras, la frecuencia de muestreo tiene que ser estrictamente mayor que dos veces el ancho de banda de dicha señal. Esto es conocido como *teorema de muestreo* o *criterio de Nyquist*. Por tanto la frecuencia de muestreo debe ser mayor que $2 \cdot fmax$, donde $fmax$ es la frecuencia máxima de la señal compleja original.

El proceso de renderizar una imagen es fundamentalmente un proceso de muestreo. Consiste en tomar muestras de una escena tridimensional con el fin de obtener el valor de los colores para cada uno de los pixeles de la imagen (conjunto discreto de pixeles).

Las frecuencias espaciales en una imagen son ilimitadas, los bordes de los polígonos, límites de las sombras y otros fenómenos producen una señal que cambia de forma discontinua y que por tanto produce frecuencias que son básicamente infinitas. Por tanto es imposible eliminar por completo los problemas de aliasing. Si incrementamos la resolución de los pixeles, los artefactos debidos al aliasing simplemente ocurren en frecuencias espaciales más altas. Aunque generalmente esto supondrá que son menos apreciables. Se puede ver un ejemplo de esto en la Ilustración 3.3. En ella se representa un tablero de ajedrez infinito. En la imagen de la derecha se ha utilizado una frecuencia de muestreo superior a la de la imagen de la izquierda en la cual se pueden apreciar las perturbaciones debidas al aliasing. En la derecha siguen existiendo dichas perturbaciones solo que a una frecuencia mayor, o lo que es lo mismo en este caso, a mayor distancia.



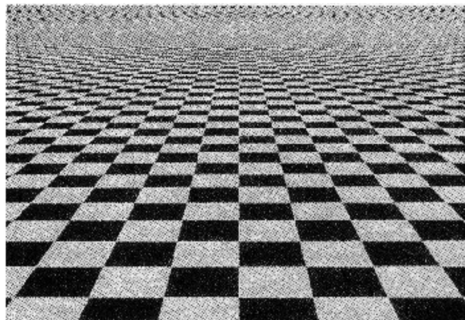


Ilustración 3.3: Tablero de ajedrez. La imagen de la izquierda ha sido muestreada con una frecuencia menor. En la imagen de la derecha los efectos de aliasing siguen apareciendo pero a mayor distancia.

Bordes en diente de sierra (jaggies)

Son producidos como consecuencia del tamaño finito de los pixeles, que generalmente son cuadrados, cuando aparecen bordes cuyo color contrasta con el entorno en la imagen. Cuando se da este efecto en imágenes animadas da la sensación de que haya pequeños objetos animados (llamados *crawlies*). En la Ilustra-

ción 3.4 se puede ver un ejemplo en el que la imagen de la izquierda no tiene jaggies mientras que la de la derecha sí.

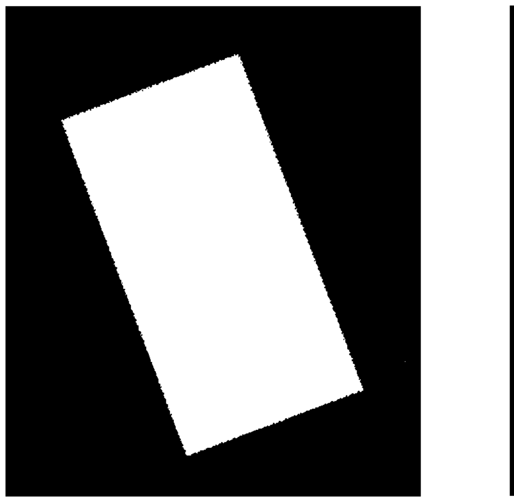
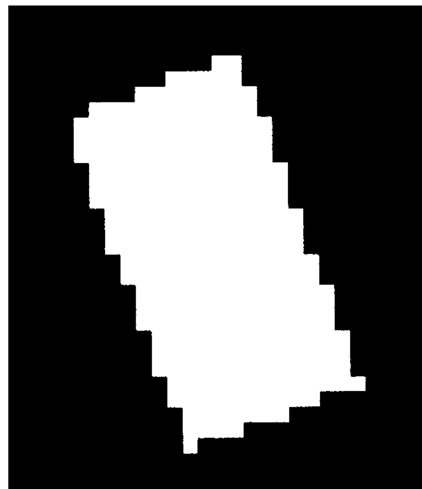



Ilustración 3.4: Jaggies.

No son defectos debidos al aliasing en el sentido clásico de lo que se ha explicado sino que son debidos a las limitaciones de resolución del dispositivo de visualización final. Sin embargo coincide con el aliasing en que aumentando la resolución disminuye su efecto.

3.1.2 Reconstrucción

Para reconstruir la imagen a partir de las muestras obtenidas se pueden usar diferentes tipos de filtros. Los filtros que comúnmente más se utilizan son tres: el rectangular (*box filter*), el triangular (*tent filter*) y el de paso bajo (*lowpass filter*). Se muestran sus funciones en la Ilustración 3.5.

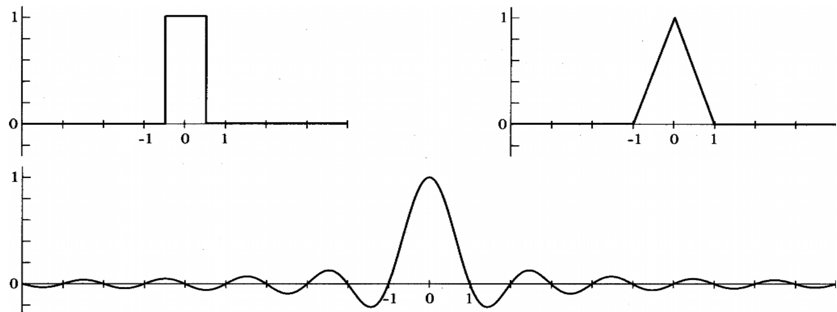


Ilustración 3.5: Filtros de reconstrucción más comunes. En la parte superior izquierda el filtro rectangular (box filter), en la parte superior derecha el filtro triangular (tent filter) y en la parte inferior el filtro de paso bajo de tipo sinc (sinc filter).

– Filtro rectangular (**box filter**): es el más simple y el que peores resultados da ya que la señal resultante queda escalonada de forma no continua. Como se aprecia en la Ilustración 3.6, para cada muestra se aplica una función en forma de caja rectangular trasladada al punto de la muestra y escalado según el valor de dicha muestra. Al final la suma de todas las funciones conforma la señal reconstruida.

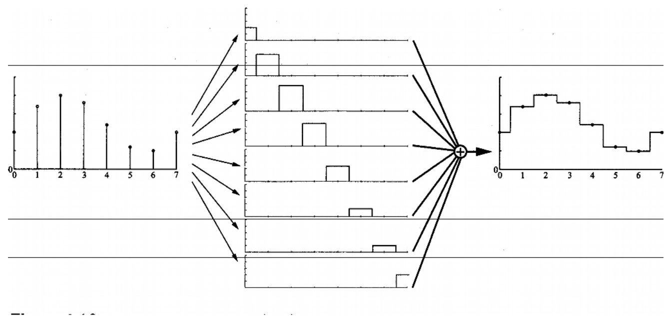


Ilustración 3.6: Filtro rectangular. Ejemplo de aplicación.

– Filtro triangular (**tent filter**): El procedimiento es el mismo que para el

caso del filtro rectangular solo que en lugar de tener forma de caja rectangular tiene forma triangular. Este tipo de filtro implementa en realidad una interpolación lineal entre muestras vecinas. Es mejor que el filtro rectangular ya que la señal resultante es continua. Sin embargo hay cambios repentinos de pendiente en los puntos de las muestras que hacen que la señal no sea muy suave. Puede verse un ejemplo de aplicación en la Ilustración 3.7.

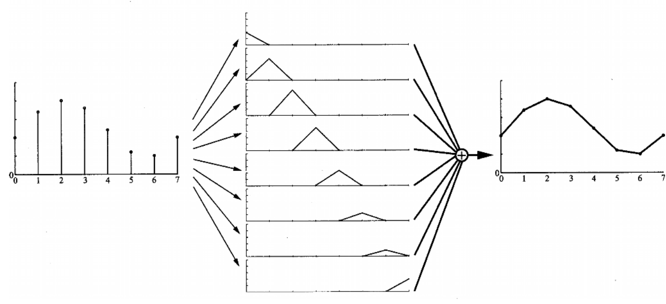


Ilustración 3.7: Filtro triangular. Ejemplo de aplicación.

– Filtro de paso bajo (**low pass filter**): Es el filtro ideal. La idea básica de este filtro es que elimina los componentes de frecuencia que están por encima de un cierto umbral de frecuencia definido por el filtro. El filtro de paso bajo ideal es el llamado *sinc*:

$$\frac{\sin(\pi \cdot x)}{\pi \cdot x}$$

Puede verse un ejemplo de aplicación en la Ilustración 3.8. El resultado es una señal mucho más suave. El motivo por el que debe utilizarse un filtro de paso bajo ideal y porqué el filtro *sinc* es ideal se explica en la *teoría del análisis de Fou-*

rier cuyo estudio está fuera del alcance de este proyecto.

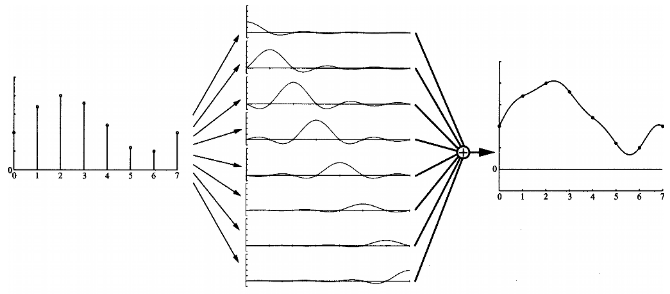


Ilustración 3.8: Filtro de paso bajo. Ejemplo de aplicación.

3.1.3 Métodos anti-aliasing

A continuación se explicarán brevemente las diferentes opciones disponibles con el fin de mejorar la calidad de la imagen final en el proceso de renderización (véase la Ilustración 3.9) y se comentarán los algoritmos más utilizados.

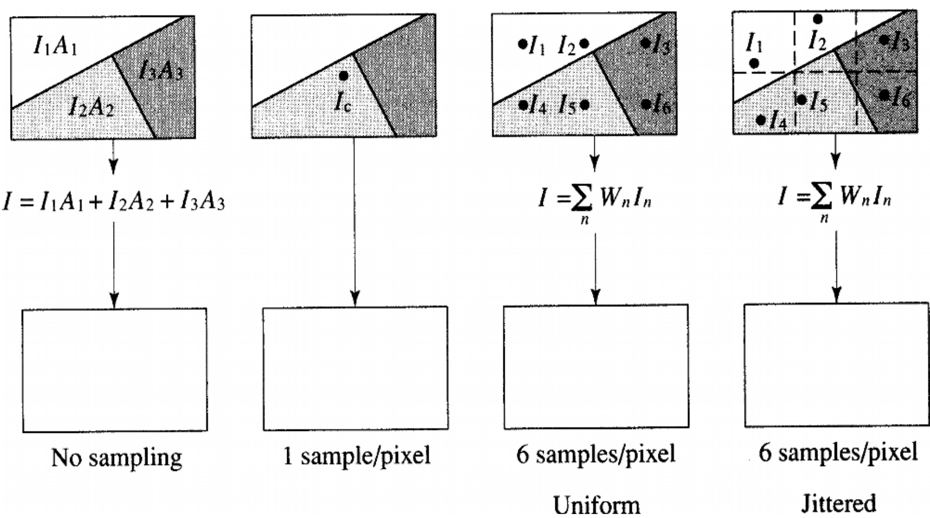


Ilustración 3.9: Comparación de cuatro aproximaciones para calcular un único color para un pixel.

Prefiltrado Infinitas muestras por pixel

Consiste en el cálculo de la contribución de los fragmentos de un objeto proyectado sobre un pixel cuyo valor será tomado como el color del pixel. En la práctica se trata de una reducción de la resolución infinita del modelo a una resolución finita de los pixeles del dispositivo. Se asume que la intensidad de la luz es constante para todo el fragmento de un mismo pixel.

Un algoritmo que implementa este método es el llamado *Carpenter's A-buffer*, también conocido como *multi-sampling*. Este algoritmo calcula la cobertura aproximada de un polígono de cada una de las celdas de una rejilla (asociada a un pixel) véase la Ilustración 3.10. En general comparte la mayor parte de los cálculos relativos a un pixel (iluminación, profundidad, etc) entre todas las celdas de dicho pixel, lo cual permite reducir el coste de computación. Generalmente se utiliza el punto medio para dichos cálculos, aunque para el caso de la profundidad existen otras aproximaciones como tomar dos valores (el mínimo y el máximo). Así pues, la máscara de cobertura, y los valores de iluminación y profundidad conforman un fragmento que acabará contribuyendo al color final del pixel (siempre y cuando no sea cubierto por otros fragmentos).

Sin embargo, este algoritmo no evita los efectos de aliasing debido a texturas y sombras. El algoritmo de A-buffer se centra sobre todo en anti-aliasing en los bordes de polígonos y renderizado correcto de transparencias.

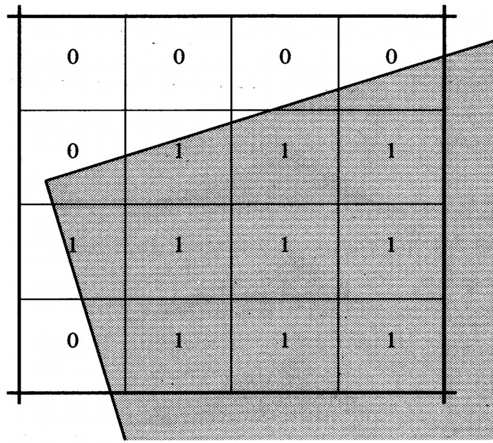


Ilustración 3.10: Cobertura de las celdas de un pixel por un polígono. Se puede representar mediante una máscara de bits según si una celda está o no cubierta.

Sin filtrado Una muestra por pixel

Es la más sencilla y barata en términos de computación, aunque también es en la que peor calidad se obtiene. Es la más extendida entre los sistemas de animación en tiempo real o como previsualización en sistemas de producción que no son en tiempo real.

Postfiltrado n muestras uniformes por pixel

Este es el método más utilizado en anti-aliasing e implica renderizar una imagen virtual de n veces la resolución de la imagen final en pantalla. La imagen final es entonces generada muestreando la imagen virtual y reconstruyéndola con una operación de convolución.

El algoritmo que implementa de forma directa este método se llama *Full*

Scene Anti-Aliasing (*FSAA*). Es un algoritmo comúnmente implementado en hardware de consumo dada su simplicidad, aunque representa un coste computacional importante ya que se deben realizar todos los cálculos de iluminación y demás para cada muestra, así como reservar espacio en memoria para cada muestra también.

Un algoritmo relacionado es el *acumulation buffer*. En lugar de reservar un buffer con el tamaño necesario para cada una de las muestras, simplemente se reserva espacio suficiente para la imagen final utilizando más bits para cada color. Consiste en generar varias imágenes, cada una con la cámara desplazada medio pixel en x e y. Las imágenes son acumuladas en el acumulation buffer. Una vez se han renderizado todas las imágenes se calcula la media de cada color (mediante una operación de división por el número de imágenes renderizadas) y se envía a la pantalla. El inconveniente es que se necesitan varios pasos, es decir, que la información de geometría y todos los cálculos derivados deben ser repetidos varias veces.

Otra variante es el *T-buffer*. Consiste en un conjunto de buffers de imagen y profundidad, cada uno de los cuales puede ser utilizado para renderizar. Existe una máscara que determina hacia donde será enviado un triángulo renderizado. Al final de la pipeline existe una lógica encargada de calcular la media de dichos buffers y enviar la información a la pantalla. Al igual que en el acumulation buffer se pueden realizar efectos de movimiento (motion blur), aunque la ventaja evidente es que se puede aplicar anti-aliasing en un solo paso de renderizado.

Las ventajas del acumulation buffer y del T-buffer frente a FSAA es que las muestras de un pixel no tienen porqué estar dispuestas siguiendo un patrón ortogonal uniforme. Por ejemplo pueden estar rotadas un cierto ángulo con respecto al centro del pixel. Los algoritmos que explotan esta posibilidad se llaman Rotated Grid Super-Sampling (RGSS).

En la Ilustración 3.11 se muestran diferentes patrones posibles de disposición de las muestras en un pixel.

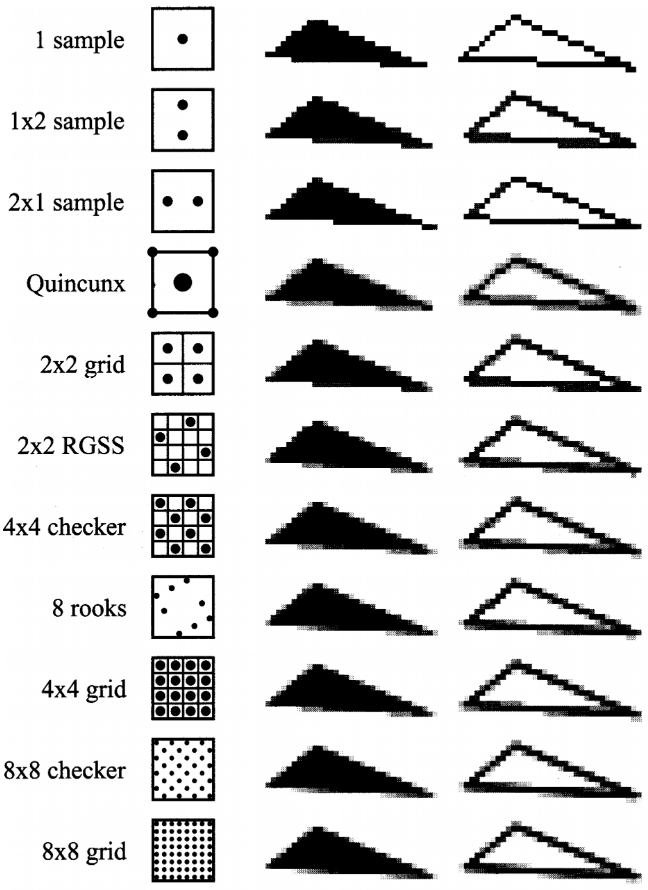


Ilustración 3.11: Diferentes patrones para la disposición de las muestras dentro de un pixel.

Postfiltrado muestras estocásticas

Éste método es una modificación del anterior en la que en lugar de tomar las muestras de un pixel siguiendo siempre un mismo patrón, la disposición de las mismas va cambiando siguiendo un patrón estocástico. La idea subyacente básica es que se acaban convirtiendo los aliases debidos a la uniformidad de las muestras en ruido.

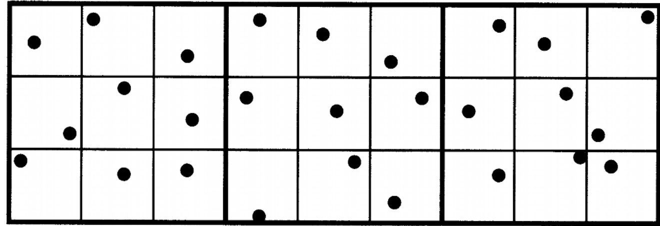


Ilustración 3.12: Ejemplo de jittering. Se muestran tres pixels, cada uno dividido en 3x3 regiones. Cada muestra se dispone en cada región de forma aleatoria.

El algoritmo más común es el jittering que funciona de la siguiente manera. Se divide el pixel en n regiones de igual área y se toma cada muestra en una posición aleatoria de cada una de esas regiones. El color final del pixel se obtiene como la media de las muestras. Véase la Ilustración 3.12 como ejemplo.

3.2 Compresión de los buffers de profundidad y color en la GPU

En este apartado se describen los detalles básicos de la arquitectura del buffer de profundidad (Z-Buffer) a modo de orientación de cómo se implementa en las GPUs actuales y se describen los principales algoritmos de compresión que se emplean con el fin de disminuir el ancho de banda entre la GPU y la memoria externa. A continuación se describe también la arquitectura y los algoritmos de

compresión para el buffer de color.

La mayor parte de la información se ha obtenido de los artículos de revisión [5] y [6], además para los temas en los que se ha necesitado comprender mejor el contenido se han consultado las patentes [7] y [8].

3.2.1 Arquitectura básica del buffer de profundidad

Como se ha explicado en el apartado el rasterizador es el encargado de identificar los pixeles que están dentro del triángulo que se está renderizando. Generalmente se trabaja en grupos de *n* x *m* pixels llamados **tile** con el fin de maximizar la coherencia de memoria entre los diferentes componentes de la arquitectura.

Cuando el rasterizador encuentra un tile que cubre parcial o totalmente el triangulo, distribuye los pixeles que comprenden dicho tile entre los *pipelines* encargados de procesarlos (veáse la Ilustración 3.13). Cada pipeline procesa básicamente la profundidad y el color del pixel, además contiene una unidad encargada de descartar los pixeles que son cubiertos por geometría previamente renderizada (test de profundidad). Para ello la unidad del test de profundidad debe consultar el buffer de profundidad donde se encuentran las profundidades calculadas para los pixeles que ya han sido renderizados. El buffer de profundidad se encuentra almacenado en memoria externa.

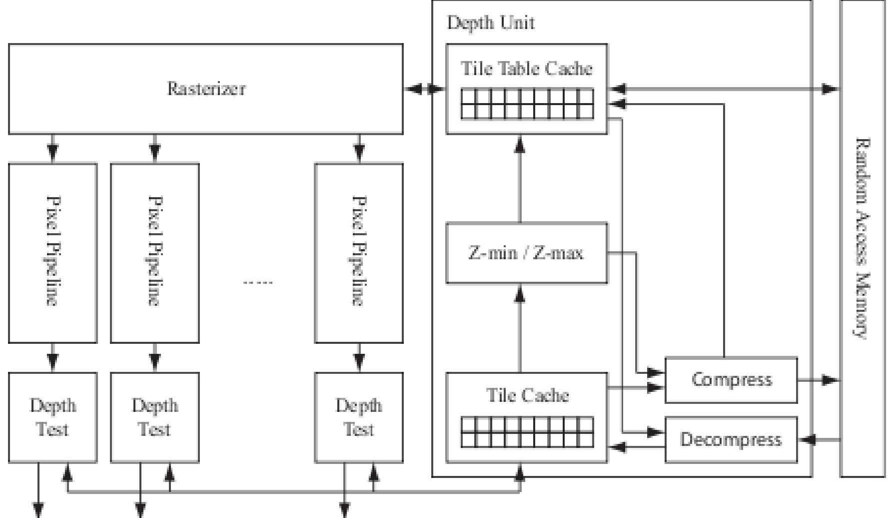


Ilustración 3.13: Arquitectura del buffer de profundidad.

En su forma más básica se consulta y almacena la información de profundidad correspondiente al tile que se está procesando. Sin embargo es conveniente poder almacenar localmente información de profundidad para más de un tile, no sólo el que se está renderizando. Para ello se puede utilizar una memoria cache residente en el propio chip. En el momento en que se necesite consultar información de profundidad que ya se encuentre en la cache se ahorrarán accesos a memoria externa, evitando así la penalización que ello supone.

Existen diferentes técnicas para mejorar aun más el rendimiento de la arquitectura basada en tiles. Una muy común consiste en guardar una tabla con información sobre cada tile. Dicha tabla se guarda a parte del buffer de profundidad e idealmente localmente en el chip, o sino en memoria externa pero utilizando una memoria cache específica residente localmente en el chip.

Cada entrada de la tabla puede contener información sobre las profundidades mínima y máxima del tile correspondiente. Dicha información puede ser

utilizada o bien para detectar cuando pueden ser descartados de golpe todos los pixeles del tile renderizado o bien saber con seguridad que todos pasarán el test de profundidad, evitándose así accesos de lectura al buffer de profundidad.

También se puede utilizar compresión de la información de profundidad cada vez que sea necesario el movimiento de la misma entre la cache (que se encuentra en el chip) y la memoria externa. De esta manera se disminuye el ancho de banda necesario. Se pueden implementar uno o varios algoritmos de compresión, o compresores. Un compresor intentará, en general, comprimir la información de profundidad de un tile a un bloque de tamaño fijo. El ratio de compresión mide la relación entre el tamaño original y el tamaño comprimido. Si la compresión no es posible la información se transmitirá sin comprimir ocupando el espacio original. Por ello se debe reservar suficiente memoria en el buffer de profundidad para dar cabida al peor de los casos, es decir que ningún tile pueda ser comprimido. Es importante tener en cuenta que el objetivo no es ahorrar espacio de memoria, sino ancho de banda en la transmisión de la información.

En la tabla de tiles se puede guardar también información sobre el algoritmo de compresión utilizado con el fin de conocer de antemano el tamaño de un bloque comprimido correspondiente a un determinado tile que se encuentre en memoria externa.

3.2.2 Compresión del buffer de profundidad

A continuación se enumerarán y se explicarán brevemente los fundamentos de los diferentes tipos de compresión del buffer de profundidad que se

pueden encontrar más comúnmente en las arquitecturas actuales.

Se asume que los valores de profundidad se guardan como elementos de 24 bits en coma flotante en el rango [0.0, 1.0] después de que se haya aplicado la matriz de proyección. En el caso de utilizar aritmética entera el 0.0 se representa como el entero de 24 bits 0 y el 1.0 como $2^{24}- 1$.

Con el fin de evitar confusiones y dado que prácticamente toda la literatura referente se encuentra en inglés, se utilizarán los nombres de los algoritmos en ingles.

Fast z-clear

Se puede considerar como la forma más simple de compresión. Cuando la información para uno o varios tiles debe ser borrada (generalmente cuando se inicializa el renderizado de un frame) en lugar de escribir el valor de borrado en memoria externa, se marca en la tabla de tiles dicha situación. De esta manera se ahorran accesos a memoria externa, no solo por el borrado en si mismo, sino porque los siguientes accesos a la información de dichos tiles pueden ser omitidos si se sabe de antemano que están borrados.

Differential Differential Pulse Code Modulation (DDPCM)

Este algoritmo explota el hecho de que los valores de profundidad son interpolados linealmente en el espacio de coordenadas de pantalla. Se basa en calcular los diferenciales de segundo orden de los valores de profundidad de un tile tal y como se muestra en la Ilustración 3.14.

Si un tile está completamente cubierto por un triángulo, los diferenciales de segundo orden valdrán siempre cero debido a la interpolación. Aunque en la práctica, debido a que la precisión utilizada en la interpolación suele ser mayor que la que se emplea para el almacenamiento los valores son {-1, 0, +1}.

Existen una serie de esquemas de compresión que aprovechan este hecho para codificar un valor de referencia con todos los bits necesarios y el resto de diferenciales con un numero fijo de bits que es menor que el de referencia.

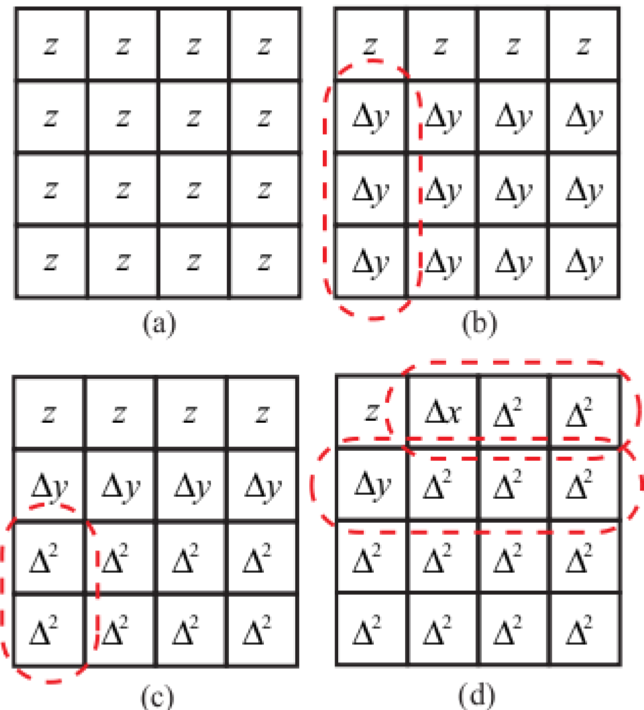


Ilustración 3.14: Cálculo de los diferenciales de segundo orden. a) Tile original, b) Diferenciales de primer orden de la columna, c) Diferenciales de segundo orden de la columna, d) Diferenciales de segundo orden de la fila.

Anchor encoding

Es muy parecido al DDPCM. Consiste en escoger el valor de profundidad de un pixel del tile como valor ancla, que será almacenado con todos los bits ne-

cesarios. A continuación se escogen 2 valores más de profundidad, diferenciales de x e y (véase la Ilustración 3.15), que se almacenan con menos bits. Estos tres puntos conforman un plano que puede ser utilizado para predecir el resto de valores de profundidad del tile. El resto de valores se guardan como la diferencia entre el plano que se predice y el valor real, utilizándose para ello un número fijo de bits todavía menor que el utilizado para los diferenciales *x* e *y*.

d	d	d	d
d	$z \rightarrow$	Δx	d
d	Δy	d	d
d	d	d	d

Ilustración 3.15: Anchor encoding para un tile de 4x4.

Plane encoding

Se trata de una aproximación similar a la anterior, solo que no se guardan los factores de corrección y tan solo se guardan los planos de predicción parametrizados. Obliga a utilizar la misma resolución en la interpolación que en el almacenamiento.

Depth offset compression

Este esquema de compresión asume que los valores de profundidad en un tile se encuentran a menudo en un intervalo cercano a uno de los valores de profundidad mínimo o máximo del tile tal y como se muestra en la Ilustración 3.16.

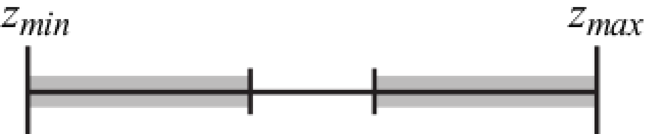


Ilustración 3.16: Depth Offset Compression. Los rangos sombreados son los intérvalos cercanos al mínimo o al máximo que pueden ser representados.

Se puede obtener la compresión guardando para cada valor de profundidad la diferencia entre el valor de profundidad y uno de los valores mínimo o máximo utilizando n bits, donde n es un valor prefijado. Se utiliza además otro bit para indicar si el valor de referencia es el mínimo o el máximo. La compresión falla si alguna de las diferencias no puede ser representada usando n bits.

Una de las ventajas de este esquema de compresión es que su implementación es sencilla y consume pocos recursos. Aunque no es de los mejores esquemas de compresión si que representa un complemento adecuado para los algoritmos explicados anteriormente para los casos en que fallara la compresión con el algoritmo principal.

3.2.3 Arquitectura del buffer de color

La mayor parte de los aspectos explicados en el apartado 3.2.1 sobre la arquitectura del buffer de profundidad se pueden aplicar al buffer de color. En especial la organización de los pixels en tiles y en el uso de tablas con información relativa a cada tile.

3.2.4 Compresión del buffer de color

A continuación se enumerarán y se explicarán brevemente los fundamentos de los diferentes tipos de compresión del buffer de color que se pueden encontrar más comúnmente en las arquitecturas actuales.

Tal y como se ha hecho con el apartado de compresión del buffer de profundidad se conservarán los nombres en inglés de los algoritmos.

Multi-sampling compression

Este algoritmo de compresión va dirigido a buffers de color que implementan multi-sampling (véase el apartado Prefiltrado de la sección 3.1.3 para más información sobre el multi-sampling). Se asume que se usan n muestras por cada pixel.

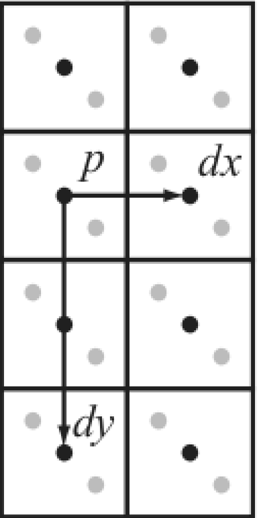
Debido al multi-sampling las muestras que pertenecen a un mismo pixel frecuentemente comparten el mismo color. Esta característica puede ser aprovechada para la compresión, ya que en lugar de guardar n colores idénticos para cada pixel bastaría con guardar un solo color y algún tipo de marca que identificara esta situación.

Otra situación que se puede dar es que el borde de un triángulo atraviese un pixel por en medio. En tal caso encontraríamos que el pixel estaría formado por dos colores, el del triángulo y el del fondo del pixel. En tal caso se guardarían los dos colores y un identificador de un bit por cada muestra que hiciera referencia a uno de los dos colores.

Color plane compression

Se basa en calcular un plano de predicción a partir de tres pixels contiguos, tal y como se muestra en la Ilustración 3.17. Se guarda la información del plano de predicción y el resto de pixels como diferencias entre el valor real del pixel y el valor predicho por el plano para dicho pixel.

Ilustración 3.17: Color plane compression



Offset compression

Se basa en comprimir un tile a partir de la identificación de de unos valores de referencia. Todos los pixels del tile son codificados como un índice al valor de referencia correspondiente y una diferencia del valor original del pixel y el valor de referencia. Una implementación típica toma los valores mínimo y máximo como valores de referencia. En la Ilustración 3.18 se pueden ver los rangos representables para la implementación comentada.

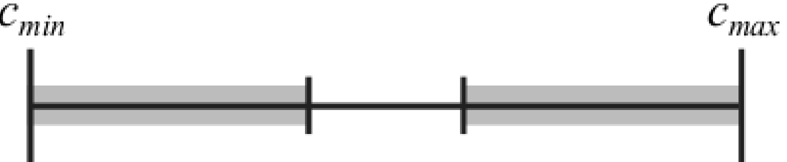


Ilustración 3.18: Color offset compression, cuando se utilizan el mínimo y el máximo como colores de referencia.

Entropy coded pixel differences

De forma muy resumida consiste en generar una secuencia de pixels para la cual se van calculando las diferencias entre el pixel actual y el predecesor. Las diferencias entre pixels contiguos con frecuencia son de pequeña magnitud debido a la naturaleza continua de las imágenes. Al final se utiliza algún tipo de codificación de dichas diferencias, como por ejemplo la que se muestra en la Ilustración 3.19.

Code	Representable value
0_b	0
$10s_b$	± 1
$110s_b$	± 2
$1110sx_b$	$\pm[3,4]$
$11110sxx_b$	$\pm[5,8]$
$111110sxxx_b$	$\pm[9,16]$
$1111110sxxxx_b$	$\pm[17,32]$
$11111110xxxxxxxx_b$	8-bit absolute value

Ilustración 3.19: Codificación de las diferencias usando códigos de longitud variable.

4 Desarrollo y resultados

En este apartado se describe el trabajo llevado a cabo para este proyecto y los resultados obtenidos.

Se empieza describiendo las características del entorno de trabajo, las trazas y la configuración utilizadas para realizar las simulaciones.

A continuación se analiza el uso del bus de memoria para las diferentes configuraciones simuladas y se justifica la necesidad de emplear técnicas de compresión para disminuirlo.

También se describen los algoritmos que han sido evaluados. La elección de los cuales ha estado basada en unos requisitos que también serán comentados. Para cada uno de los algoritmos se propone un posible diagrama de bloques de lo que sería su implementación en hardware.

Después de la descripción se presentan los resultados obtenidos del análisis de dichos algoritmos y se estudia la viabilidad de implementación de cada uno de ellos.

Finalmente se explican los detalles del algoritmo implementado en el simulador y se justifica su elección.

4.1 Entorno de trabajo

A continuación se describen las herramientas y los entornos más significativos utilizados para la realización del proyecto. Se ha intentado utilizar, siempre que ha sido posible, herramientas multiplataforma y que no fuera propietaria. En general se ha trabajado con plataformas *GNU/Linux* al estar más familiarizados con ella.

4.1.1 ATILA

Para el desarrollo del proyecto se ha hecho uso intensivo del simulador ATILA que ha sido explicado con más profundidad en el apartado 2.2.

Con el fin de facilitar la tarea de compilar el simulador en diferentes plataformas *GNU/Linux* y con diferentes versiones del compilador *gcc* se ha reescrito el sistema de construcción utilizando *Makefiles*. No ha sido una tarea inmediata, ya que ATILA es un proyecto muy grande, pero ha permitido trabajar de forma mucho más cómoda con él durante el resto del proyecto.

Se han realizado también pequeñas modificaciones en el simulador con el fin de poder interceptar y capturar el tráfico de datos entre las memorias cache (de profundidad y de color) y el controlador de memoria. Dicho punto se corresponde con el lugar donde se ubica el compresor. En la Ilustración 4.1 puede verse enmarcado en un rectángulo más oscuro los puntos interceptados.

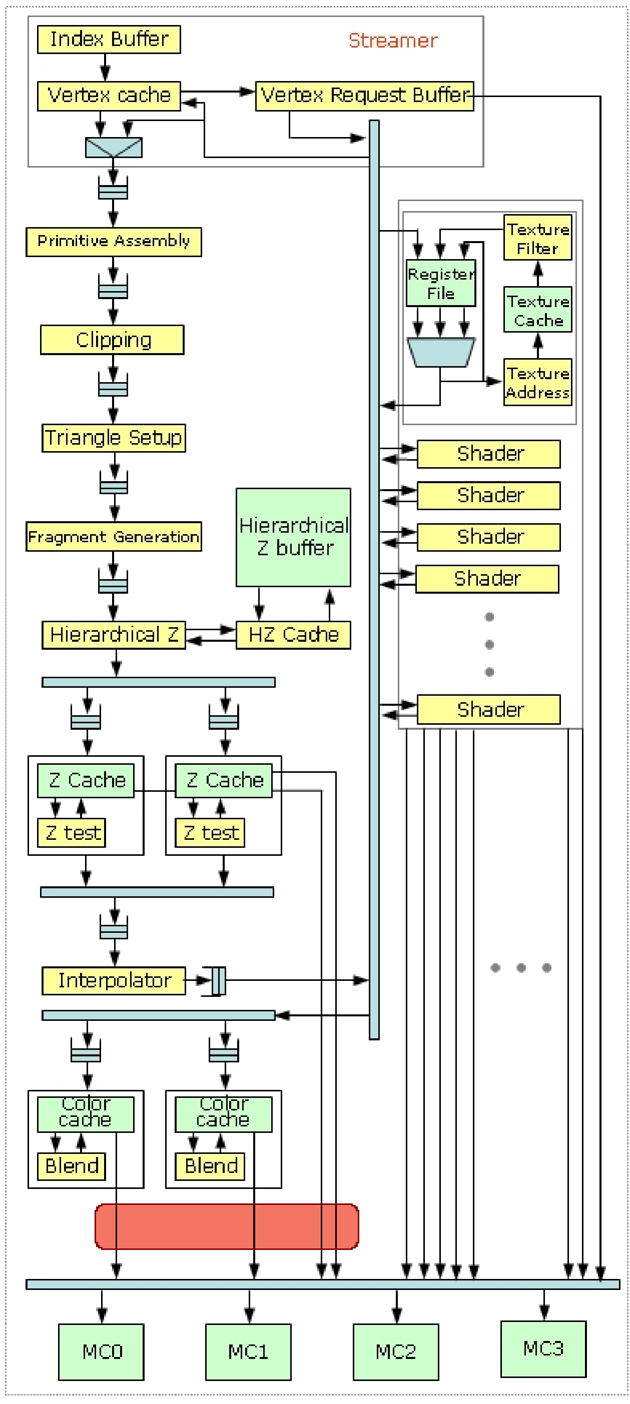


Ilustración 4.1: Puntos interceptados para la captura de datos para la evaluación de algoritmos de compresión en rojo (caja más oscura).

Cada vez que se ejecuta una simulación se obtiene una captura que puede ser almacenada y utilizada a posteriori para su análisis. Una ventaja de esta aproximación es que permite realizar cambios en los algoritmos analizados o añadir nuevos algoritmos a posteriori sin necesidad de volver a ejecutar la simulación (lo cual implicaría un coste de tiempo importante). El inconveniente es que las capturas ocupan bastante espacio de almacenamiento (alrededor de 400 Gb para las trazas analizadas).

4.1.2 Cluster

Las trazas que se han utilizado para ejecutar las simulaciones tienen más de 2000 frames. Cada *frame* tarda entre 20 y 180 minutos en ejecutarse dependiendo de la configuración empleada para la simulación. Si tenemos en cuenta que se han ejecutado 4 trazas diferentes con 4 configuraciones diferentes por cada traza resulta evidente que solo podía realizarse en un tiempo razonable utilizando procesamiento en paralelo. Para ello se ha utilizado el cluster *Salvat* del *Departamento de Arquitectura de Computadores* que está formado por 64 nodos cada uno de los cuales tiene 2 procesadores con 2 núcleos cada uno.

El cluster tiene un sistema de ejecución por colas (*N1 Grid Engine*[1] de *Sun Microsistems*[2]) que permite gestionar la ejecución de varios procesos entre los diferentes nodos. Se ha desarrollado un sistema de scripts ad-hoc para la ejecución y coordinación de las simulaciones de los diferentes frames de las trazas así como el procesado de los resultados con el fin de automatizar lo máximo posible todo el trabajo. Se han utilizado para ello *bash*, *python* y *makefiles*.

4.1.3 Procesado y visualización de resultados.

Los resultados intermedios resultantes de la ejecución de simulaciones y del procesado de los resultados son básicamente tablas en formato *CSV* (Comma Separated Values) que permiten ser fácilmente tratados tanto con herramientas de scripting o simples editores de texto, como con herramientas más sofisticadas

1 Para más información sobre estas herramientas puede consultarse la referencia [9]
2 Sun Microsystems es una marca registrada de Sun Microsystems, Inc. en Estados Unidos y otros países.

como hojas de cálculo gráficas o paquetes estadísticos. Para cada frame acaban generándose varias de estas tablas con diferentes tipos de información. Generalmente las tablas están en formato comprimido con el fin de ahorrar espacio de almacenamiento (se utiliza *gzip*).

Además de las herramientas de scripting utilizadas en el apartado 4.1.2 se han utilizado herramientas de procesamiento estadístico (*GNU R*[1]) y de generación de gráficos (*gnuplot*[2]). Ambas totalmente automatizables mediante el uso de scripts propios de cada herramienta.

4.2 Trazas y configuración utilizadas

Para que el simulador pueda funcionar necesita dos cosas, por un lado una traza de datos con el fin de proveer de comandos a la GPU, y por otro una configuración de simulación que determine el comportamiento del mismo. A continuación se describen con más detalle cada uno de estos aspectos y los valores concretos que se han utilizado.

Trazas de ejecución

Las trazas utilizadas en este proyecto (véase la Tabla 4.1) ya estaban generadas y disponibles para el público en general en la wiki de ATILA [2]. Fueron obtenidas utilizando herramientas especiales de captura desarrolladas por el equipo ATILA a partir de diferentes juegos de ordenador.

1 Para más información sobre esta herramienta puede consultarse la referencia [11]
2 Para más información sobre esta herramienta puede consultarse la referencia [10]

	Juego	Resolución	Nº de frames
	Doom 3	1280x1024	2398
	Quake 4	1280x1024	2270
	Unreal Tournament 2004	1280x960	2759
	Prey guru5	1280x1024	2802

Tabla 4.1: Trazas utilizadas para las simulaciones con ATILA.

Parámetros de configuración del simulador

De la gran cantidad de parámetros que pueden ser configurados en el simulador ATILA, la gran mayoría han sido configurados con un valor constante para todos los experimentos llevados a cabo. Se puede consultar el contenido del archivo de configuración utilizado en el apartado 9.2 del apéndice.

Para poder evaluar diferentes configuraciones de multi-sampling se han tenido que ajustar dinámicamente para cada experimento los siguientes parámetros:

- **ForceMSAA**: Permite activar/desactivar la funcionalidad de multi-sampling en el simulador.

- **MSAASamples**: En caso de que esté activado el multi-sampling permite especificar cuantas muestras por pixel son necesarias.

Para cada traza se han realizado simulaciones con cada una de las combinaciones mostradas en la Tabla 4.2.

Nombre de la configuración	ForceMSAA	MSAASamples
x0	FALSE	0
x2	TRUE	2
x4	TRUE	4
x8	TRUE	8

Tabla 4.2: Configuraciones utilizadas para cada una de las trazas simuladas.

4.3 Analisis de tráfico de memoria

En [12] se caracterizan diversos juegos y se estudian multitud de parámetros para una configuración del simulador equivalente a una tarjeta *ATI R520*. Sin embargo no encontramos información sobre como influyen las diferentes configuraciones de multi-sampling que el simulador soporta sobre el tráfico de memoria generado.

Por tanto se consideró importante contrastar la hipótesis de que configuraciones con más muestras de multi-sampling acabarían generando más tráfico de memoria y conocer en que proporción aproximadamente.

Dicha información se ha obtenido a partir de los estadísticos que el simulador genera, en concreto de los estadísticos por frame.

Se han utilizado los siguientes estadísticos:

- **ReadBytes_Sched[0-8]** que permite medir el trafico total de lectura que se ha generado para cada uno de los 8 planificadores del controlador de memoria configurados.

- **WriteBytes_Sched[0-8]** que permite medir el trafico total de escritura que se ha generado para cada uno de los 8 planificadores del controlador de memoria configurados.

En el gráfico de la Ilustración 4.2 se muestra el tráfico generado, tanto de lectura como de escritura, para una parte significativa de la traza quake4. En él se puede ver cómo las configuraciones de multi-sampling de mayor número de muestras presentan mayor cantidad de tráfico. Sin embargo, se observa que dicho incremento no es proporcional al número de muestras. Esto es debido a que es tráfico total y que incluye, no solo el tráfico sensible al multi-sampling, sino también el de texturas, vértices, etc.

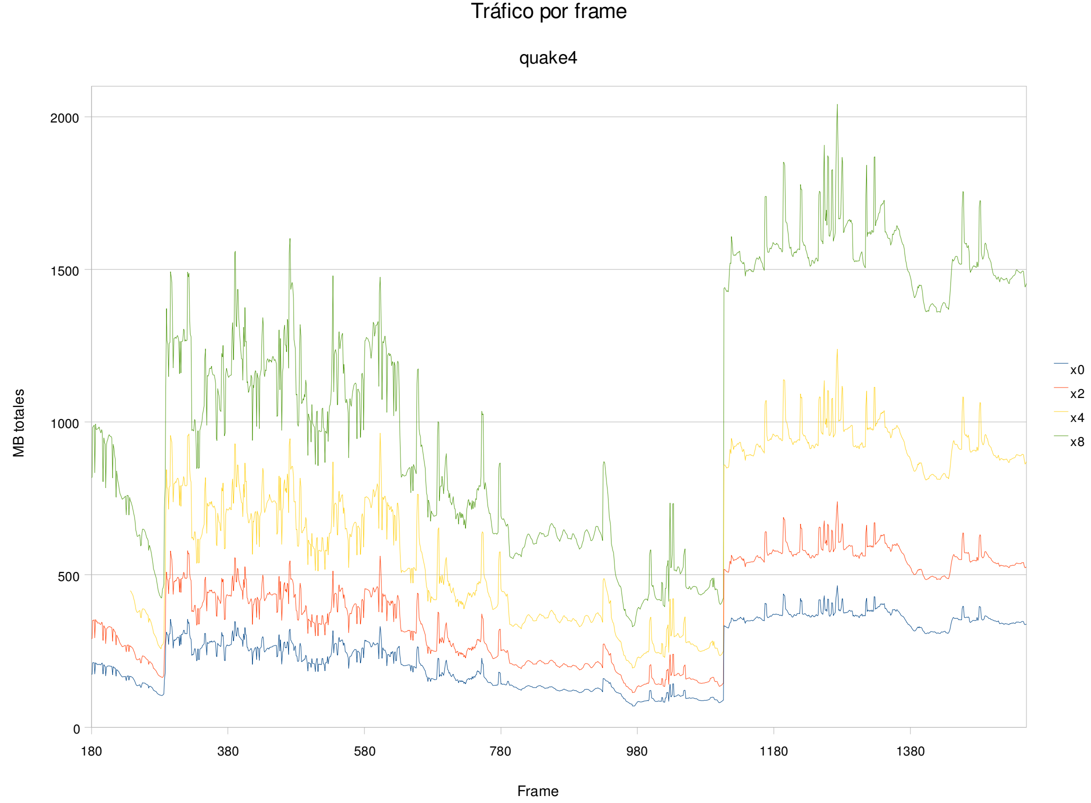


Ilustración 4.2: Tráfico por frame para la traza quake4.

4.4 Requisitos de los algoritmos de compresión

Uno de los objetivos más importantes de este proyecto consiste en evaluar diferentes algoritmos de compresión con el fin de acabar implementándolo en el simulador. El simulador es una aplicación de software que representa un modelo de hardware. Por tanto hay que tener en cuenta que los algoritmos de compresión que se evalúen deben ser susceptibles de ser implementados en hardware con unos costes asequibles. Ello impone una serie de requisitos que se enumeran a continuación:

- Debe ser altamente paralelizable.

- Debe poder ser dividido en etapas.

- La latencia de ejecución debe ser lo más baja posible.

- Los bloques resultantes de la compresión deben tener un tamaño múltiplo del ancho del bus de datos.

Estos requisitos han condicionado totalmente la elección de los algoritmos utilizados. En el apartado 4.5 se explicarán con más detalle tanto los algoritmos como su posible implementación en hardware.

4.5 Algoritmos de compresión evaluados

En base a los conocimientos adquiridos sobre métodos de compresión de los buffers de profundidad y color (véase el apartado 3.2), y los requisitos enumerados en el apartado 4.4 se han diseñado una serie de algoritmos con el fin de evaluarlos y poder contrastar el rendimiento de cada uno.

Los algoritmos diseñados se basan en los métodos de *offset compression* y *msaa compression* debido a que su implementación es sencilla y además permiten paralelizarlos.

El esquema general del proceso de compresión se puede ver en la Ilustración 4.3. Un determinado compresor será visto como una caja negra que admite una entrada (un bloque de datos de tamaño fijo) y como resultado da un bloque

de datos que puede ser del mismo tamaño que el bloque de entrada o bien menor (recordemos que se debe cumplir el requisito de que sea múltiplo del ancho del bus de datos). Así mismo el bloque de datos de salida debe poder ser descomprimido con el fin de obtener el bloque de datos original.

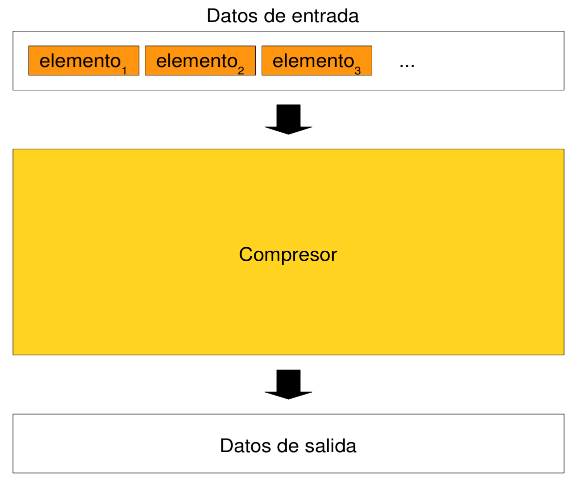


Ilustración 4.3: Esquema del proceso de compresión.

El tamaño de un bloque de datos de la memoria cache tanto para profundidad como para color en el simulador ATILA es de 256 bytes. En estos 256 bytes hay 64 elementos contiguos (que codifican o bien la profundidad de una muestra o bien el color) donde cada elementos es de 4 bytes. La disposición de dichos elementos en la imagen final viene determinada por la arquitectura interna del simulador y no se ha considerado en la evaluación. Lo que si se sabe es que los elementos contiguos en un bloque de datos se corresponden con partes contiguas de la imagen.

A continuación se explican con más detalle cada uno de los algoritmos clasificados según el método de compresión en el que se basan.

4.5.1 Offset Compression

hilo

Este algoritmo está basado en la patente [7]. En realidad el algoritmo fue diseñado para compresión de buffers de profundidad y no de color. Sin embargo se consideró la posibilidad de utilizar los 32 bits que componen un color (8 bits para cada una de las componentes *RGBA*) como un número natural con el que poder realizar las operaciones que se describen en el algoritmo (véase la Ilustración 4.4).

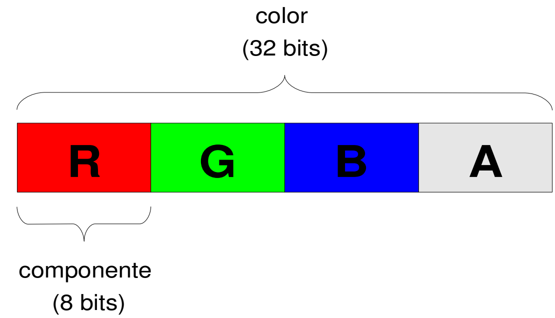


Ilustración 4.4: Representación de un color en componentes RGBA de 8 bits cada una utilizando un entero de 32 bits.

El procedimiento es el siguiente, se obtienen cuatro elementos de referencia del bloque de datos de entrada. Las dos primeras referencias se corresponden con los valores mínimo y máximo de dicho bloque, y las otras dos referencias restantes se obtienen sumando un desplazamiento al mínimo y restando el mismo desplazamiento al máximo (véase Ilustración 4.5).

Si llamamos a esas referencias min, max, a y b tenemos que podemos definir cuatro rangos de valores que son $[min, a)$, $[a, a+desplazamiento)$, $(b-desplaza$-

miento, *b*], (*b*, *max*]. Todos los elementos que estén dentro de estos rangos pueden ser codificados como una diferencia del valor del elemento con respecto a una de las referencias y un índice que indique cual de esas referencias se ha utilizado.

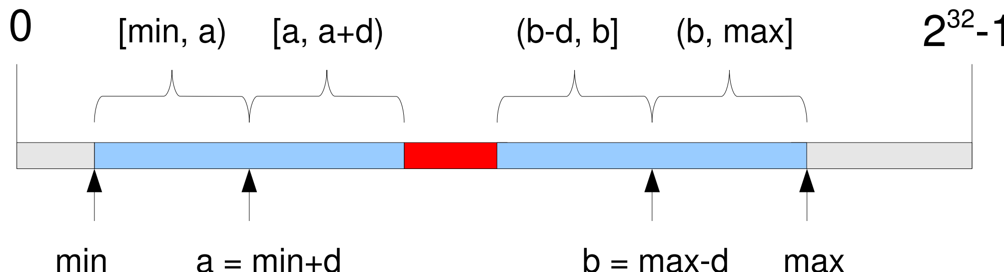


Ilustración 4.5: Eje de valores representables para un entero de 32 bits en el que se delimitan los rangos definidos por las referencias min, max, a y b. Los segmentos en gris se corresponden con rangos en los que no entrará ningún elemento del bloque de entrada. En azul los rangos que si que pueden ser codificados y en rojo el rango no codificable.

El número de bits necesarios para guardar las diferencias del valor del elemento con respecto al valor de referencia es menor que el número de bits necesarios para el elemento original, y para codificar el índice bastarán 2 bits.

Si se utiliza un desplazamiento múltiplo de 2 para la obtención de los valores de referencia *a* y *b*, entonces los cálculos y las comparaciones se simplifican bastante y permite distinguir dos partes para cada elemento: la parte alta y la parte baja (véase la Ilustración 4.6). La parte alta será la que se utilizará para las comparaciones <u>con las partes altas de los valores de referencia</u> y la parte baja será la que se utilizará para codificar las diferencias del valor del elemento con respecto al valor de referencia. A partir de ahora para el caso de los algoritmos basados en offset cada vez que se haga mención a comparaciones con las referencias se estará refiriendo a la comparación de las partes altas.

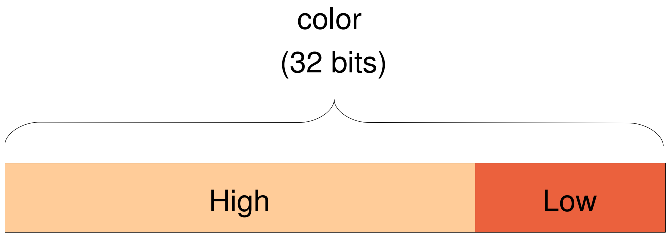


Ilustración 4.6: Representación de la parte alta (high) y la parte baja (low) en que se puede dividir un elemento si lo tomamos como un entero natural de 32 bits.

Si alguno de los elementos del bloque de entrada no puede ser codificado siguiendo el criterio explicado anteriormente debido a que no está dentro de ninguno de los rangos de referencia, entonces la compresión falla y el bloque de datos de salida del compresor será idéntico al de entrada y no se habrá podido reducir su tamaño.

Según cuantos bits se utilicen para codificar el desplazamiento utilizado para calcular las referencias a y b, o lo que es lo mismo si tenemos en cuenta que el desplazamiento es múltiplo de 2, según cuantos bits se utilicen para la parte alta y baja, el tamaño del bloque de datos de salida del compresor será diferente (véase la Ilustración 4.7). Cuanto mayor sea el tamaño de la parte baja, mayor será el tamaño del bloque de salida. Por otro lado un tamaño mayor de la parte baja implica que la parte alta será menor. Puesto que la parte alta es la que se utiliza para comparar con las referencias, cuanto menor sea, mayor será la probabilidad de que coincida con alguna referencia, y por tanto de que el bloque de entrada pueda ser comprimido. De aquí se obtiene que pueden utilizarse diferentes tamaños de parte alta y baja según el equilibrio que se quiera tener entre el tamaño del bloque de datos de salida y la probabilidad de éxito de compresión del bloque de

entrada.

Si se fijan los tamaños posibles del bloque de datos de salida (cosa que viene motivado por el hecho de que los bloques de salida del compresor deben tener un tamaño múltiplo del ancho del bus de datos), nos interesará utilizar el tamaño de la parte alta más pequeño posible que permita que el tamaño del bloque de salida sea menor o igual que el tamaño prefijado, o lo que es equivalente que la parte baja tenga el número máximo de bits. De esta manera se obtienen diferentes niveles de compresión que pueden ser implementados en paralelo. En el caso de que varios niveles de compresión tengan éxito se escoge el bloque de datos de salida más pequeño.

Low bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bytes	24	32	40	48	55	63	71	79	86	94	102	110	117	125	133	141

Low bits	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Bytes	148	156	164	172	179	187	195	203	210	218	226	234	241	249	257	265

Ilustración 4.7: Tamaños del bloque de salida en función del número de bits de la parte baja. Si se fijan los tamaños de salida de 64, 128 y 192 bytes nos interesará trabajar con partes bajas de tamaño 5, 13 y 21 bits respectivamente.

En la Ilustración 4.8 se muestra un esquema de bloques de una posible implementación en hardware del algoritmo explicado. Los bloques de color naranja (más oscuros) representan bloques de datos y el número indica el tamaño en bytes (cabe recordar que un color tiene un tamaño de 4 bytes, por tanto en un bloque de 256 bytes caben 64 colores). El bloque min, max references se encarga de encontrar las referencias del mínimo y el máximo a partir del bloque de entrada. A partir de éstas referencias el bloque a, b references calcula las otras dos (a y

b) utilizando una unidad de suma y otra de resta respectivamente. Dicha información es pasada a los diferentes bloques de *check & encode* de cada uno de los niveles de compresión implementados para que codifique, si es posible, el bloque de entrada. Si no es posible la compresión, entonces la salida queda indefinida. Cada codificador transfiere un bloque comprimido de salida a un selector (*best level selector*) que escoge el de menor tamaño en función de la información que le llega de los codificadores de cada nivel.

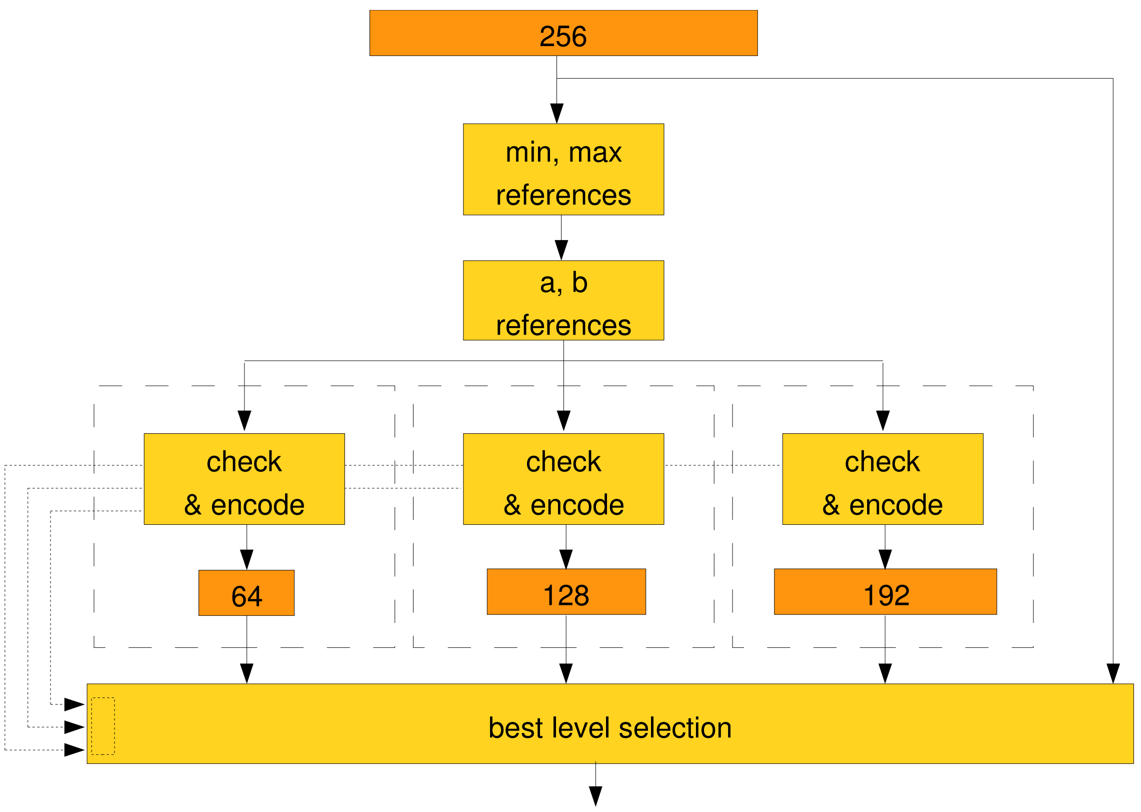


*Ilustración 4.8: Arquitectura del compresor **hilo** para tres niveles de compresión (bloques de 64, 128 y 192 bytes).*

En la Ilustración 4.9 se muestra el proceso de búsqueda de las referencias mínimo y máximo a partir del bloque de entrada. Para ello se utilizan comparadores en cascada que acaban formando una pirámide invertida en la que se distin-

guen diferentes niveles, cada uno de los cuales dependiente del nivel superior. Los comparadores de un mismo nivel trabajan en paralelo y el número de niveles será $\log_2$(nº elementos). En realidad son necesarias dos cascadas diferentes que trabajan en paralelo, una para la búsqueda del mínimo y otra para el máximo.

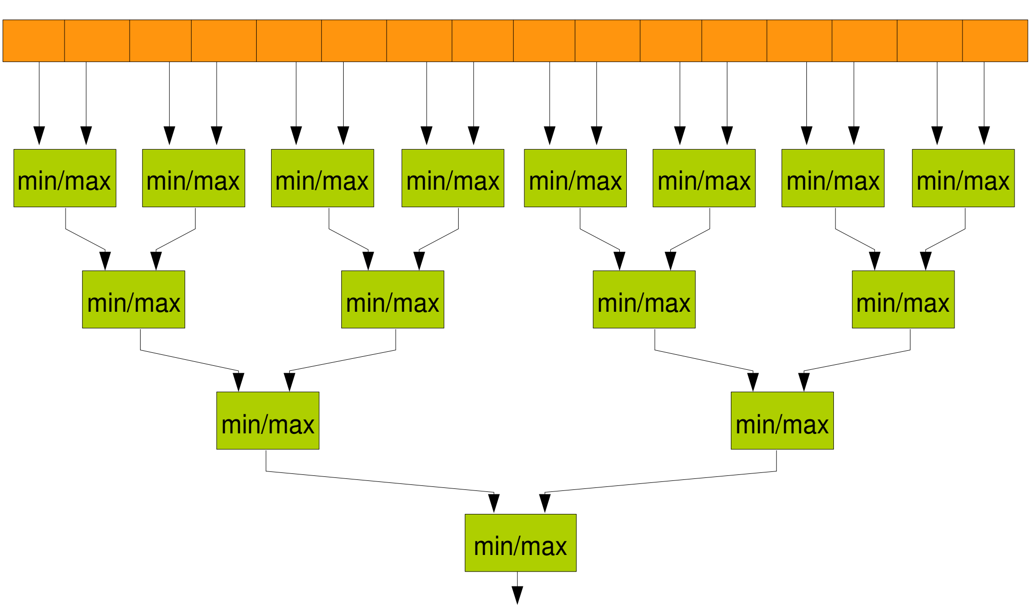
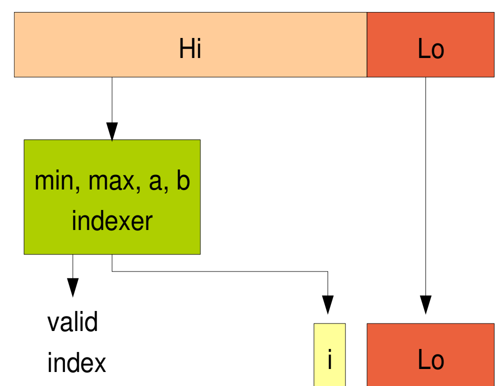


Ilustración 4.9: Búsqueda en cascada del mínimo/máximo de un bloque de datos de entrada. Se distinguen diferentes niveles, cada uno de los cuales depende del superior. Los comparadores de un mismo nivel trabajan en paralelo. Se a tomado un bloque de 16 elementos a modo de ejemplo, aunque se puede extender fácilmente a bloques de mayor tamaño teniendo en cuenta que el número de niveles será $\log_2$(nº elementos).

En la Ilustración 4.10 se muestra el proceso de comprobación y codificación que se lleva a cabo en el bloque check & encode para uno sólo de los colores del bloque de entrada. Si los indicadores de valid index para todos los elementos son verdaderos, entonces ese bloque puede ser codificado utilizando ese nivel de compresión.

Ilustración 4.10: Comprobación y codificación de un elemento del bloque de datos de entrada. El indexador comparará la parte alta (Hi) con cada una de las referencias, si coincide con alguna valid index vale 1 y i representa un índice a la referencia.



hilore

Si recordamos la división que se hacía entre parte alta y baja y tenemos en cuenta los componentes que forman un color y cómo están dispuestas, se ve que dependiendo del número de bits que se tomen para la parte alta y baja, predominarán bits de las componentes R y G en la parte alta y B y A en la parte baja. Si por ejemplo la componente R varía mucho de un color a otro dentro de un mismo bloque entonces el algoritmo de compresión no será tan efectivo, aunque el resto de componentes a penas varíen. Una forma de intentar evitar que una sola componente pueda tener tanta influencia por si misma, consiste en partir a su vez cada componente en parte alta y baja y reordenar los bits de cada componente de tal forma que los bits altos de las componentes queden en la parte alta y los bits bajos de las componentes queden en las partes bajas. En la Ilustración 4.11 se muestra claramente esta reordenación.

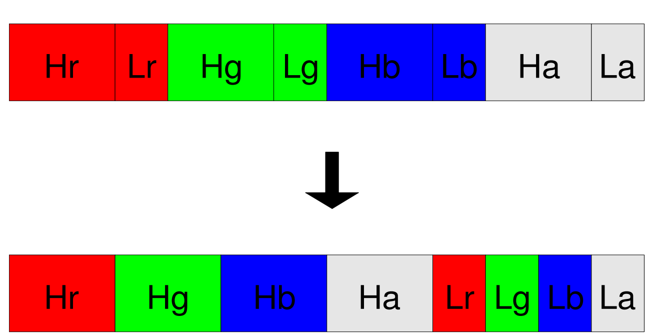


Ilustración 4.11: Reordenación de bits por componentes. Las letras mayúsculas H y L representan High y Low respectivamente y las minúsculas r, g, b, a se refieren a las componentes del color.

Se debe intentar dividir cada componente en partes alta y baja teniendo en

cuenta que las partes altas de las componentes deben encajar lo mejor posible en la parte alta del color entero y lo mismo para la parte baja. Por ejemplo, si inicialmente se había decidido utilizar una parte baja para el color entero de 8 bits, lo ideal sería tomar 2 bits de la parte baja por cada componente de forma que sumen un total de 8. Aunque no siempre se puede hacer de forma exacta, pues si se escogen 5 bits para la parte baja del color y 1 bit para la parte baja del componente vemos que quedará un bit de la parte baja del color que será rellenado por un bit al que le correspondería la parte alta del color.

Tal y como se hacía en hilo dependiendo del número de bits que se tomaran para las partes alta y baja del color podían definirse diferentes niveles de compresión.

Como la búsqueda de referencias viene a continuación de la reordenación y para cada nivel de compresión tenemos una reordenación diferente de los bits, entonces necesitaremos bloques de búsqueda de referencias independientes para cada nivel de compresión. Esto implicará un coste mayor en términos de área del chip al tener que replicar estructuras. Para simplificar un poco el proceso, en este algoritmo tan solo se emplean dos referencias, el mínimo y el máximo, ahorrándonos el tener que calcular a y b. Entonces se necesitará tan solo 1 bit para los índices de referencia. El resto del proceso de compresión sería equivalente que para el caso de hilo.

En la Ilustración 4.12 se muestra un diagrama de bloques de una posible implementación en hardware de este algoritmo. El bloque reorder se encarga de

reordenar los bits de cada elemento tal y como se ha explicado. Dicho proceso que en software requeriría varias operaciones, en hardware es trivial.

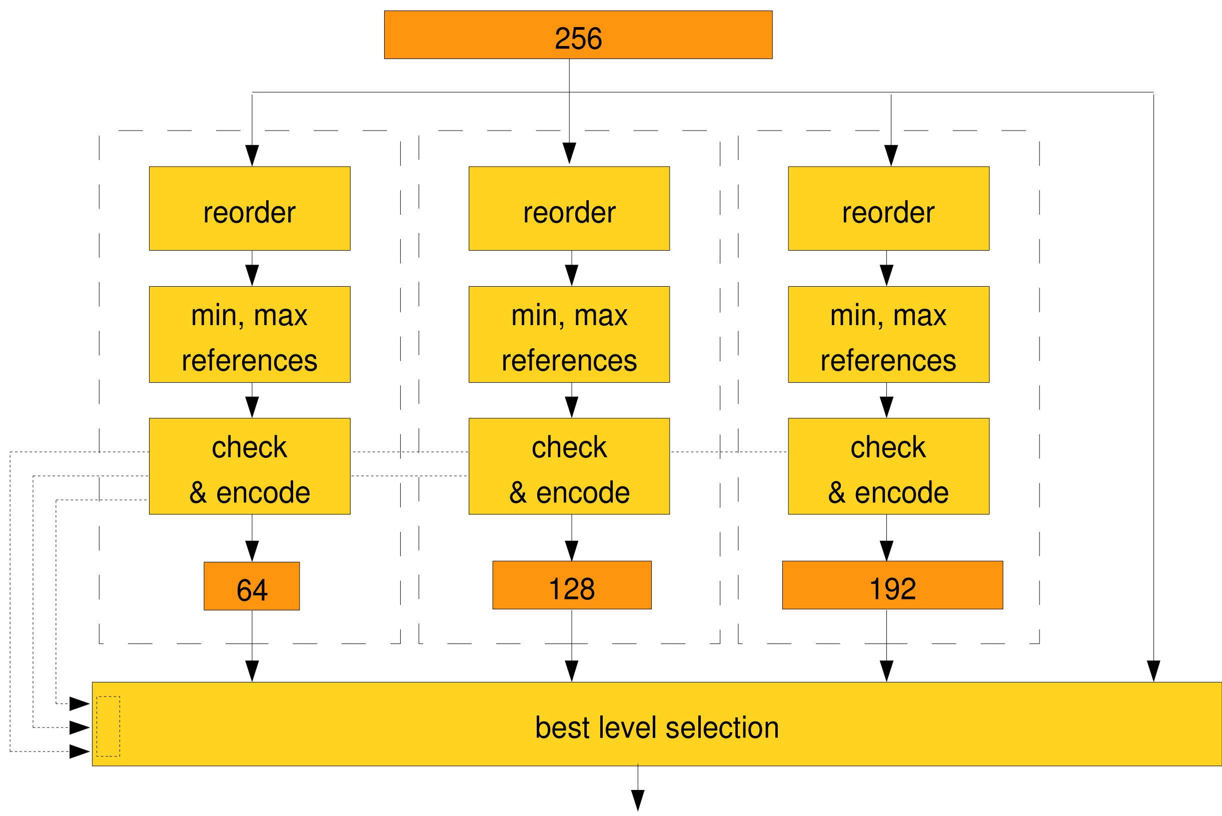


*Ilustración 4.12: Arquitectura del compresor **hilore** con tres niveles de compresión (bloques de 64, 128 y 192 bytes).*

hilorebi

Como se ha visto en la Ilustración 4.9 la búsqueda del mínimo o el máximo implica tener que atravesar de forma secuencial una cascada de comparadores. Según el número de elementos del bloque de entrada el número de niveles que hay que atravesar secuencialmente se calcula como $\log_2$(nº elementos). En el caso de ATILA los bloques son de 64 elementos, por lo que el número de niveles que hay que atravesar es 6. La latencia de cálculo de dicho proceso depende del número de niveles, por tanto interesará minimizar el número de niveles que atrave-

sar. Una forma de conseguirlo es dividir el bloque de entrada en dos particiones y buscar el mínimo y el máximo para cada una de estas particiones de forma independiente (en paralelo). De esta forma acabamos obteniendo cuatro referencias, el mínimo y el máximo de una partición y el mínimo y el máximo de la otra.

Este algoritmo será igual que el hilore solo que en lugar de usar sólo dos referencias usará cuatro y además se habrá reducido la latencia de búsqueda de los mínimos y máximos en un nivel de comparadores. Se puede consultar la Ilustración 4.12, pero en lugar de bloques min, max references habrá bloques min1, max1, min2, max2 references . El número de bits por índice será de 2.

En este algoritmo además de intentar mejorar la latencia total también se están usando más referencias, lo cual implica mejorar las posibilidades de éxito del compresor, aunque dependerá del carácter de los datos de entrada. Si ambas particiones tienen colores similares y las referencias acaban siendo iguales no habremos ganado mucho, sino más bien todo lo contrario, pues cuantas más referencias tengamos mayor tamaño tendrá el bloque de salida. Sin embargo si coincide que con las cuatro referencias podemos cubrir todo el rango de valores posibles (véase la Ilustración 4.5) entonces tendremos garantizado el éxito de compresión de dicho bloque.

hilore4ref

En este algoritmo la estrategia es similar al anterior (hilorebi), también se buscan cuatro referencias. La diferencia es que se intenta que sean lo más significativas posible, es decir que permitan cubrir el máximo rango de valores posibles.

Las dos primeras referencias las obtiene, como el resto de algoritmos, de buscar el mínimo y el máximo del bloque de datos de entrada. Las otras dos referencias se obtienen como se explica a continuación.

Se comparan todos los elementos del bloque de entrada con la referencia de valor mínimo y se escoge el primer elemento empezando por el principio de valor diferente a dicha referencia. Se realiza lo mismo pero con la referencia de valor máximo y se escoge el primer elemento empezando por el final que tenga valor diferente. En caso de no encontrar valores diferentes se escogerá el mínimo o el máximo de forma indiferente. De esta manera se intenta tener el máximo número de referencias diferentes. En la Ilustración 4.13 se puede ver un ejemplo.

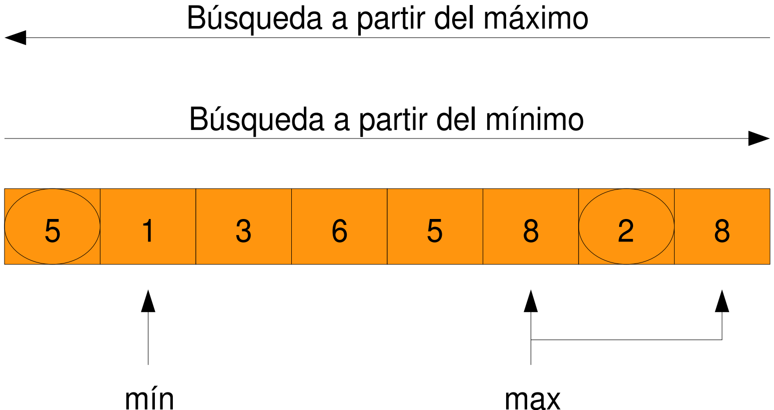


Ilustración 4.13: Búsqueda de referencias. Una vez encontradas el mínimo y el máximo se obtienen las otras dos referencias (marcadas con un círculo). El 5 se obtiene a partir del mínimo y el 2 a partir del máximo siguiendo el sentido que marcan las flechas de la parte superior.

El resto del proceso es igual que para el algoritmo hilorebi pero utilizando las cuatro referencias obtenidas tal y como se ha explicado.

comp

Este algoritmo trata de hacer lo más independientes posibles las variaciones de valor de las componentes RGBA. Para ello se fragmenta el bloque de datos de entrada en 4 bloques más pequeños cada uno de los cuales contiene información de color de una sola componente (véase la Ilustración 4.3).

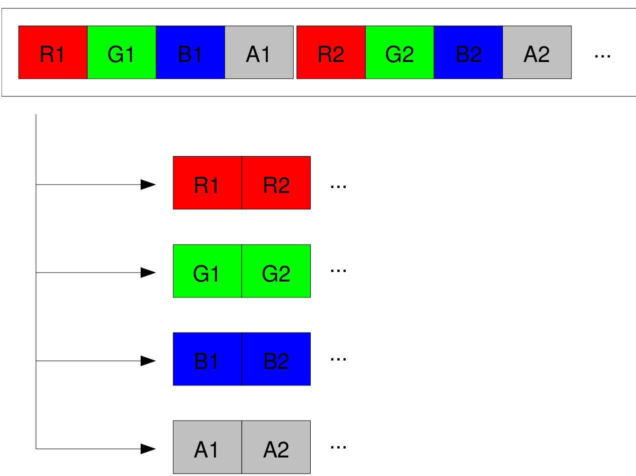


Ilustración 4.14: Tratamiento de las componentes de forma independiente.

Para cada componente se buscan las referencias mínimo y máximo y se lleva a cabo el resto del proceso tal y como se ha explicado para los demás algoritmos. Los bloques de salida se acaban juntando y se obtiene así el bloque de salida comprimido.

El inconveniente es que de cada elemento de entrada acabaremos obteniendo 4 índices de 1 bit cada uno, uno por cada componente. Es decir, acabaremos teniendo que almacenar en el bloque de salida 4 bits por cada color de entrada. Por otro lado la búsqueda de referencias es mucho más rápida ya que se puede realizar en paralelo para cada componente y además los números que se ma-

nejan son de sólo 8 bits.

4.5.2 Multi-sampling compression

msaa

Este algoritmo se ha basado en la patente [8]. El objetivo principal de este algoritmo es explotar el hecho de que cuando se utiliza multi-sampling es más probable que se dé el caso de que todas las muestras que pertenecen a un mismo fragmento sean iguales. Incluso se puede dar el caso todavía más favorable de que varios fragmentos contiguos sean idénticos. En estos casos bastará con guardar un solo color.

Otro caso que se contempla es el hecho de que cuando el borde de un triángulo atraviesa un fragmento quedan delimitadas dos zonas. La que pertenece al interior del triángulo y la del fondo del fragmento. Esto implica que en muchas ocasiones bastará con guardar los colores del fondo y del triángulo y un índice por cada muestra que indique cual de esos dos colores debe utilizarse. En la Ilustración 4.15 se representan estos casos.

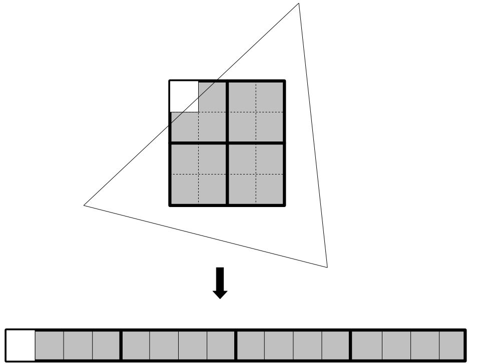


Ilustración 4.15: Ejemplo de cobertura parcial y total de los fragmentos y su representación en un bloque de 16 elementos. Se muestra un tile de 2x2 fragmentos (también llamado quad). Cada fragmento a su vez esta formado por 2x2 muestras. Se asume que las muestras sombreadas acabarán teniendo el mismo color al estar cubiertos por el triángulo mientras que la muestra que está fuera tomará el color del fondo de la imagen. Los fragmentos totalmente sombreados pueden ser comprimidos usando un solo color de referencia mientras que el fragmento superior izquierdo necesitará 2 colores de referencia.

El algoritmo consiste en buscar subbloques de colores contiguos que cumplan alguno de los casos citados anteriormente. Se empezará explicando el primer caso que es el más sencillo y a continuación se explicará el segundo caso.

En el primer caso se tratará de buscar subbloques de elementos contiguos que cumplan que son idénticos. Cuanto mayor sea el tamaño de dichos subbloques mejores resultados se obtendrán, aunque el número de elementos por subbloque debe ser el mismo para todos los subbloques de un mismo bloque de entrada. Teniendo en cuenta que los bloques de salida comprimidos deben ser de tamaño 64, 128 o 192 bytes y que los bloques de entrada son de 256 bytes, tan solo interesará comprobar subbloques de 2 o 4 elementos, pues con más elementos se obtienen bloques de salida mas pequeños de 64 bytes, que no pueden ser tratados por el controlador de memoria. En la imagen Ilustración 4.16 se muestra un ejemplo.

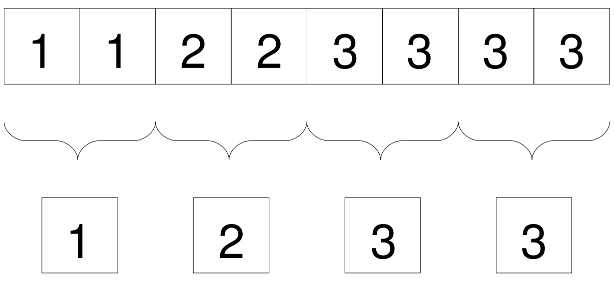


Ilustración 4.16: Ejemplo de compresión con subbloques de 2 elementos usando una sola referencia por subbloque.

En el segundo caso en el que se utilizaban dos colores de referencia por cada subbloque se seguirá el mismo procedimiento pero con la restricción de que en un subbloque no pueden haber más de dos colores diferentes. Igualmente to-

dos los subbloques de un bloque de entrada deben tener el mismo número de elementos. En este caso se guardarán dos colores por cada subbloque, y no uno como se hacía en el caso anterior. Además se tendrá que guardar una máscara de bits que indiquen para cada elemento del subbloque que color se tomó como referencia. En la Ilustración 4.17 se muestra un ejemplo.

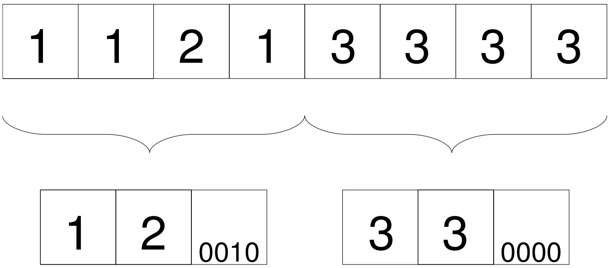


Ilustración 4.17: Ejemplo de compresión con subbloques de 4 elementos con 2 referencias por subbloque.

Los diferentes niveles de compresión que se pueden utilizar vienen determinados por dos parámetros:

- El número de elementos que se consideran por subbloque.
- El número de referencias que se toman por subbloque.

En la Tabla 4.3 se muestran los niveles utilizados para evaluar este algoritmo en función de dichos parámetros y el tamaño resultante del bloque comprimido que se obtiene a la salida.

		Nº de elementos por subbloque			
		2	4	8	16
Nº de referencias por subbloque	1	128	64		
	2		192	128	64

Tabla 4.3: Niveles de compresión utilizados. En cada celda se especifica el tamaño en bytes del bloque comprimido resultante. Las celdas vacías se corresponden con niveles no evaluados o que no tiene sentido evaluar.

En la Ilustración 4.20 se muestra un diagrama de bloques de una posible implementación en hardware de este algoritmo.

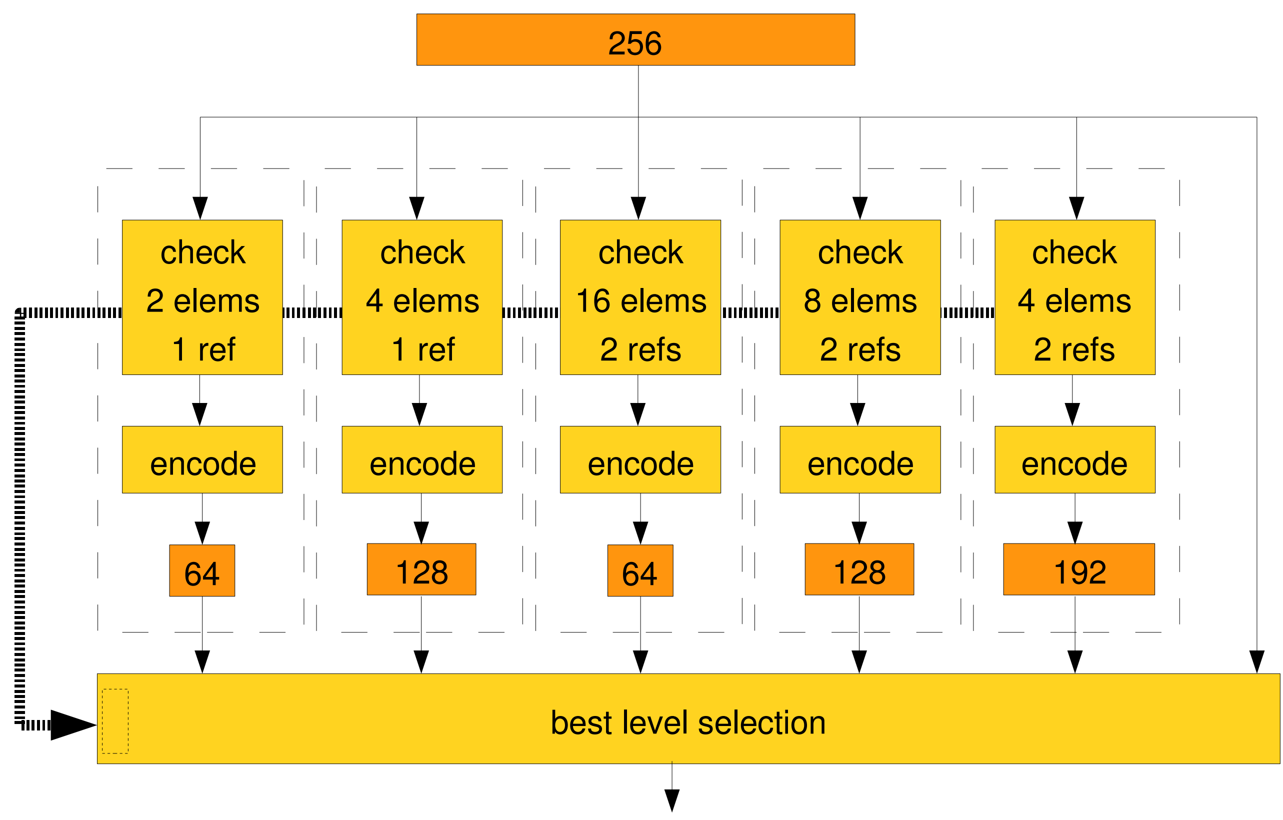


Ilustración 4.18: Arquitectura del compresor msaa.

4.6 Análisis de los algoritmos de compresión

En la Ilustración 4.19 se muestra el proceso llevado a cabo para analizar los algoritmos de compresión explicados en el apartado 4.5. Se ha utilizado el sistema de tuberías de *UNIX* para crear una cadena de analizadores que reciben los bloques de datos por la entrada estándar, los procesan y los redireccionan hacia la salida estándar. De esta manera todos los analizadores procesan la misma información, y puesto que el sistema operativo lanza un proceso por cada analizador el trabajo se acaba repartiendo en paralelo entre los procesadores disponibles en la máquina.

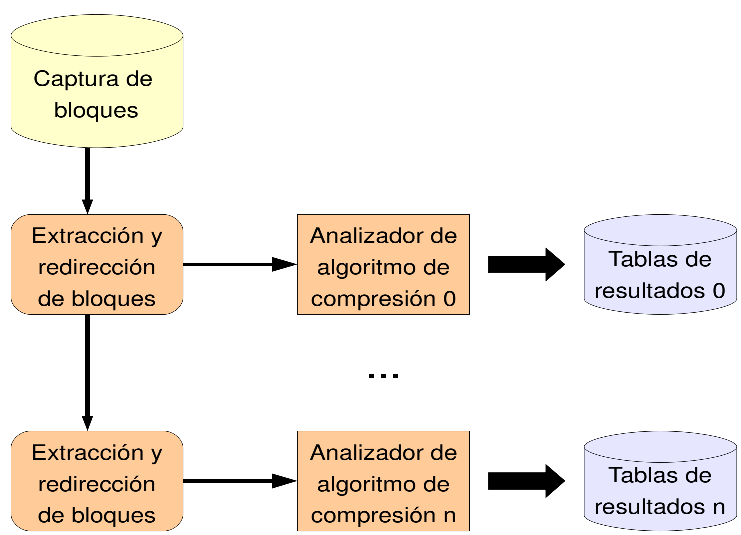


Ilustración 4.19: Proceso general de análisis de los algoritmos de compresión.

Los bloques de datos se obtienen de las capturas de tráfico entre las memorias cache y el controlador de memoria tal y como se explicó en el apartado 4.1.1. Puesto que existe una captura por cada traza y configuración simuladas, el análisis de las diferentes capturas se puede repartir entre los diferentes nodos disponibles del cluster.

Existe una implementación software por cada algoritmo de compresión evaluado. Dicha implementación se encarga de recibir bloques de datos, aplicar el algoritmo de compresión y generar estadísticos con los resultados obtenidos. Se ha utilizado el lenguaje *Java 1.4* ya que presenta un balance adecuado entre velocidad de ejecución y sencillez de utilización.

Por cada combinación posible de traza, configuración y algoritmo se obtiene una tabla en la que se cuentan cuantos bloques se han generado de cada tamaño posible. En la Tabla 4.4 se muestra un ejemplo.

level	count
64	412869
128	165944
192	215347
256	67438

Tabla 4.4: Ejemplo de tabla obtenida tras analizar la traza prey para la configuración de multi-sampling x4 para el algoritmo comp.

A partir de estas tablas se pueden generar tablas derivadas que contienen el tráfico de datos total procesado y el tráfico de salida generado tal y como se muestra en la Tabla 4.5 de ejemplo.

level	none	hilo	hilore	hilorebi	hilore4ref	comp	msaa
64	0	11,71	12,53	12,48	13,46	12,16	11,78
128	0	2,35	7,09	7,49	8,71	6,66	19,74
192	0	28,49	14,21	15,48	13,46	16,5	3,67
256	91,8	2,28	8,54	6,25	2,6	7,86	0,32
total	91,8	44,82	42,37	41,7	38,23	43,17	35,51

Tabla 4.5: Ejemplo de tabla con tráfico de salida (medido en GB totales) de los diferentes algoritmos analizados para la traza prey con multi-sampling x4.

El tráfico total generado se calcula como:

$$\text{Tráfico de salida} = \sum_{s \in \{64, 128, 192, 256\}} s * \text{Bloques de tamaño s generados}$$

Debido a que el número de tablas generadas es muy grande no se mostrarán aquí, aunque podrán ser consultadas en el CD-ROM adjunto (véase el apartado 9.1 del apéndice para consultar cómo está organizado).

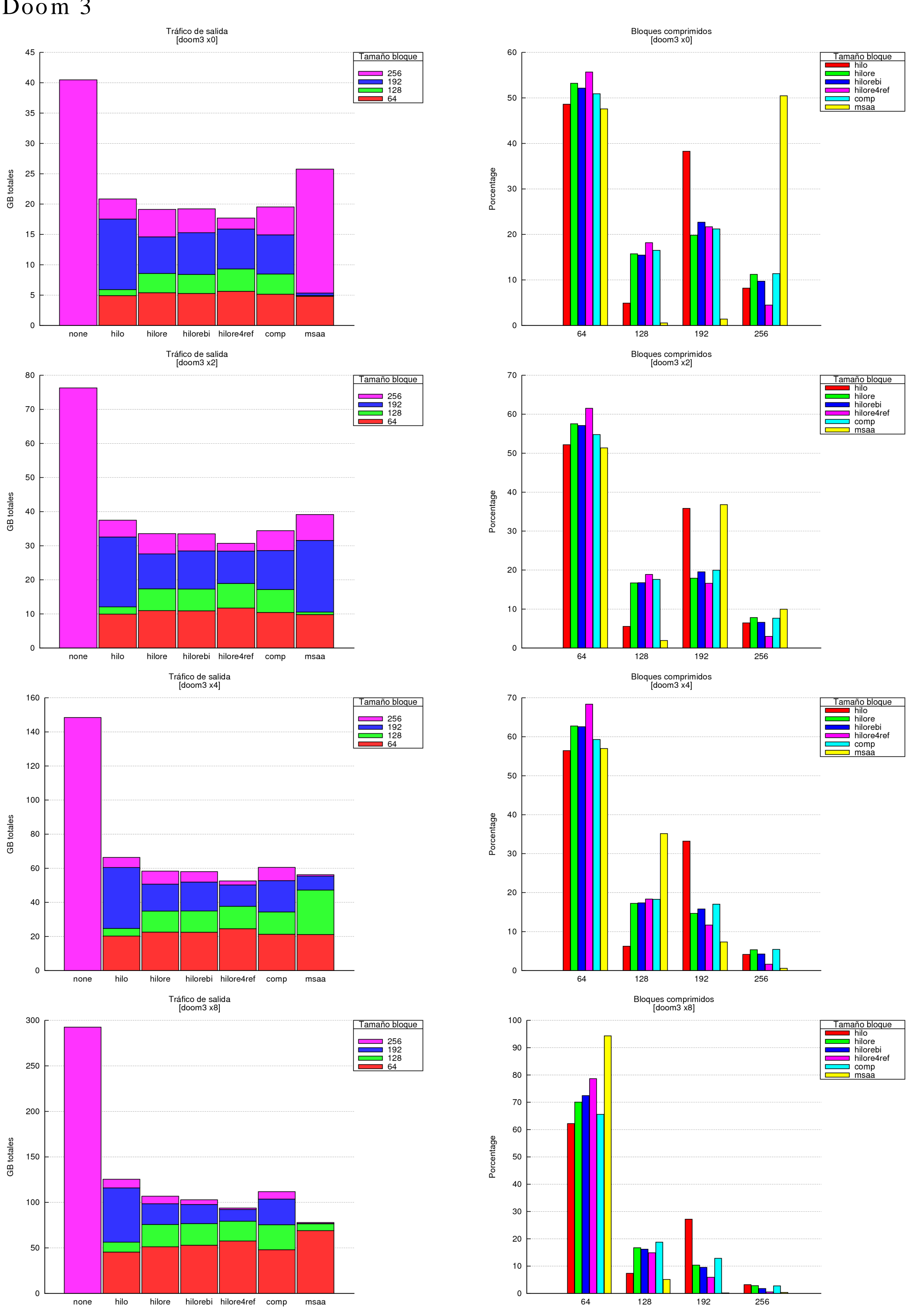
A partir de los datos obtenidos se han generado dos tipos de gráficos:

- Tráfico de salida generado por cada algoritmo separado por tamaños de bloque posibles.

- Tanto por ciento de bloques comprimidos para cada tamaño de bloque posible.

Los gráficos se muestran a continuación y están agrupados por trazas. La columna de la izquierda se corresponde con los gráficos de tráfico generado y la de la derecha con el tanto por ciento de bloques comprimidos. Cada fila se corresponde con una configuración de multi-sampling diferente en el siguiente orden: *x0*, *x2*, *x4*, *x8*.

Doom 3



Quake 4

Tráfico de salida [quake4 x0]

Bloques comprimidos [quake4 x0]

Tráfico de salida [quake4 x2]

Bloques comprimidos [quake4 x2]

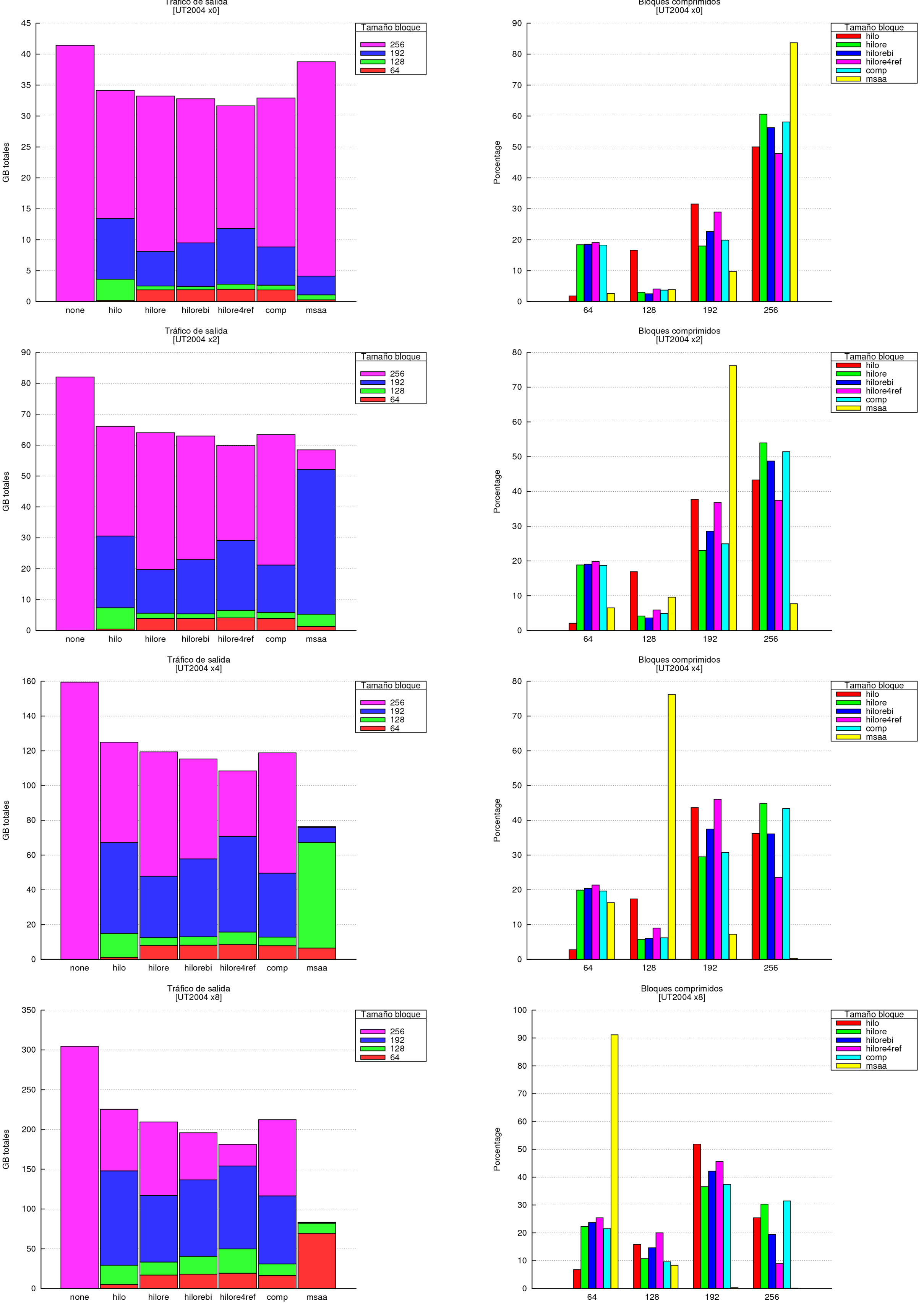
Tráfico de salida [quake4 x4]

Bloques comprimidos [quake4 x4]

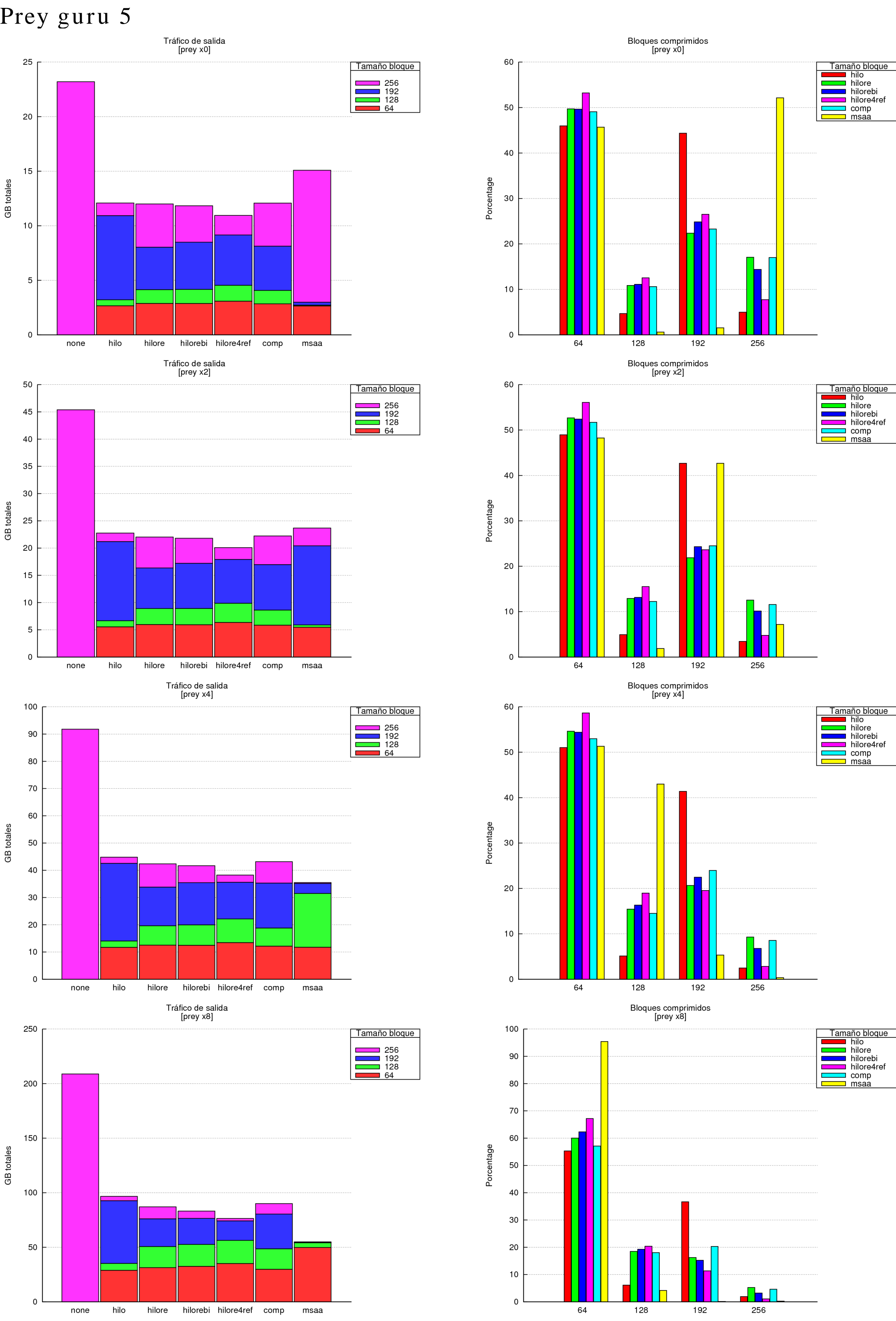
Tráfico de salida [quake4 x8]

Bloques comprimidos [quake4 x8]

Unreal Tournament 2004



Prey guru 5



En los gráficos de tráfico de salida, cada columna representa el tráfico generado a la salida de cada compresor. La excepción es la columna none que representa que no se ha usado compresor y que por tanto todos los bloques generados han sido del máximo tamaño (256 bytes   color cían), o lo que es lo mismo equivale al tráfico total a la entrada de los compresores.

Lo deseable es que el tráfico a la salida de un compresor sea lo más bajo posible, que es un indicador de que el algoritmo se comporta mejor.

En esta ocasión, a diferencia del gráfico de la Ilustración 4.2, el tráfico de salida para diferentes configuraciones de multi-sampling si que es proporcional en función del número de muestras. Esto es debido a que el tráfico capturado es totalmente sensible a dicho factor y ya no incluye el tráfico generado por unidades ajenas.

Además cada barra está dividida en secciones de diferente color que se corresponden con el tráfico generado a partir de los bloques que pertenecen a un determinado tamaño de entre los posibles (64, 128, 192 y 256 bytes). Lo deseable en este sentido es que la mayor parte del tráfico pertenezca a las regiones de los tamaños de bloque más pequeños. El compresor que sea capaz de generar más bloques de tamaños más pequeños dará mejores resultados que los demás.

Lo explicado también se puede observar desde el punto de vista de la proporción de bloques de un determinado tamaño que se han generado. Esto es lo que se mide con los gráficos que están en la derecha.

Como ejemplo podemos mirar el gráfico de la traza UT2004 para la configuración x8. En él el tráfico a la salida del compresor msaa es en su mayor parte debido a bloques de tamaño 64 bytes (rojo) y se comporta mucho mejor que cualquier otro.

Otra forma de mirarlo es que los compresores que fallen más bloques (y que generarán mayor cantidad de bloques de tamaño 256 bytes) presentarán resultados peores. Es lo que se observa en las gráficas de UT2004. En este juego hay escenas con mucha variación de color y los compresores en general han fallado muchos más bloques, obteniéndose así peores resultados que en el resto de trazas en las que generalmente las escenas tenían menos iluminación y menos variación de los colores.

Pese a que todos los resultados se muestran teniendo en cuenta un ancho de bus de 64 bytes (de ahí que los posibles tamaños de los bloques sean 64, 128, 192 y 256) también se han realizado análisis teniendo en cuenta un ancho de bus de 32 bytes. Pero no se muestran para abreviar. El resultado en general es que se consigue ahorrar un poco más de tráfico de memoria debido a la relajación de la restricción de alineación de los tamaños de bloque a la salida posibles que pasan a ser 32, 64, 96, 128, 160, 192, 224, 256 bytes.

También se debe tener en cuenta que la escala de los gráficos para las diferentes configuraciones es diferente, y que por tanto, pequeñas variaciones de las barras del gráfico en la configuración x8 son más significativas que en el gráfico de x0. En este sentido, cabe observar que el algoritmo msaa se comporta peor

que los demás en las configuraciones de multi-sampling de menos muestras, sin embargo, para los casos de mayor número de muestras es muchísimo mejor, suponiendo un ahorro de tráfico mucho más importante.

4.7 Resultados comparativos

Con el fin de facilitar la decisión sobre que algoritmo escoger se han diseñado nuevas tablas y gráficos utilizando los ratios de compresión medios de todas las trazas.

A partir del tráfico de entrada y el tráfico generado a la salida se puede calcular el ratio de compresión medio obtenido para una traza, configuración y algoritmo determinado. Permite caracterizar en cuantas veces se ha reducido el tráfico de entrada y se calcula como:

$$\text{Ratio de compresión} = \frac{\text{Tráfico de entrada}}{\text{Tráfico de salida}}$$

Para una misma configuración de multi-sampling se puede calcular la media de los ratios de compresión de las diferentes trazas. En la Tabla 4.6 se muestran los ratios medios obtenidos para los diferentes algoritmos de compresión evaluados, y en la Ilustración 4.20 la representación gráfica de dicha tabla.

	hilo	hilore	hilorebi	hilore4ref	comp	msaa
x0	1,77	1,85	1,87	2,00	1,83	1,46
x2	1,83	1,94	1,96	2,12	1,92	1,81
x4	1,92	2,08	2,13	2,32	2,04	2,49
x8	2,03	2,28	2,41	2,62	2,21	3,73

Tabla 4.6: Ratio de compresión medio por configuración de multi-sampling para cada algoritmo evaluado.

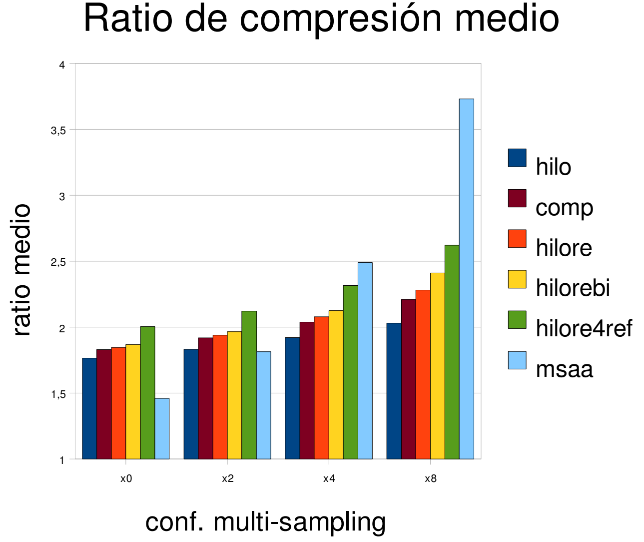


Ilustración 4.20: Gráfico comparativo de ratios de compresión medio por configuración de multi-sampling para cada algoritmo evaluado.

A simple vista se puede ver cómo el algoritmo msaa destaca sobre todos los demás en las configuraciones de multi-sampling x4 y x8, tal y como habíamos observado en el apartado de análisis, pese a su menor rendimiento en las configuraciones x0 y x2. No es de extrañar teniendo en cuenta que se trata de un algoritmo especialmente diseñado para explotar las características del multi-sampling.

Entre los algoritmos basados en offset compression se ve cómo a pesar de que el rendimiento mejora con las configuraciones de multi-sampling con mayor número de muestras, el rendimiento de un algoritmo en relación a los demás puede ser clasificado independientemente de dichas configuraciones.

Por tanto se pueden clasificar teniendo en cuenta tan solo el ratio de compresión de la siguiente manera:

- msaa
- hilore4ref
- hilorebi
- hilore
- comp
- hilo

Los resultados obtenidos hasta este punto todavía no son concluyentes, pues todavía hay otro factor que se debe tener en consideración.

Los algoritmos que se han evaluado (véase apartado 4.5) han sido seleccionados teniendo en cuenta los requisitos comentados en el apartado 4.4 con el fin de que fueran aptos para una implementación en hardware. Sin embargo no se pueden implementar todos, y por tanto hay que seleccionar el algoritmo o algoritmos más adecuados. Para ello no solo se tendrá en consideración los resultados de rendimiento obtenidos en el análisis (véase apartado 4.7) sino que también se

deberá tener en consideración el coste de ejecución en hardware.

Para cuantificar el coste de ejecución se utilizará la latencia de cálculo del algoritmo de compresión. Dicha latencia se medirá en ciclos de reloj y será estimada a partir de los diagramas de bloques propuestos en el apartado 4.5. Se asumirá que la latencia de cálculo de una unidad básica de comparación equivale a 1 ciclo de reloj.

En la Tabla 4.7 se muestran las latencias estimadas para cada algoritmo.

Algoritmo	Latencia
hilo	10
hilore	9
hilorebi	8
hilore4ref	12
comp	7
msaa	6

Tabla 4.7: Latencias estimadas para los algoritmos evaluados.

Como se puede observar el algoritmo que mejor ratio de compresión tiene también es el más económico en términos de latencia. Otro detalle observable es que el siguiente algoritmo mejor, hilore4ref, es el que tiene un mayor coste de ejecución lo que lo hace menos apropiado.

4.8 Implementación en el simulador

Después de estudiar los resultados de los algoritmos la decisión sobre que algoritmo implementar en el simulador se decanta hacia el algoritmo msaa. Pese a que no tiene el mejor rendimiento para configuraciones de multi-sampling bajas

si que presenta la mejor relación coste/rendimiento con diferencia sobre los demás presentando los mejores resultados tanto a nivel de ratio de compresión como de coste de ejecución·

A continuación se describe la arquitectura software de la implementación del compresor y sus relaciones con otros componentes del simulador (véase también la Ilustración 4.21).

Las unidades que necesitan acceder a los buffers de profundidad y color son la unidad de test de profundidad (*Z Test*), la unidad de mezcla del color (*blending*), el *DAC* y el *Blitter*.

La unidades DAC y Blitter tan solo necesitan tener acceso de lectura y por tanto tan solo necesitarán descomprimir.

Las unidades de Z test y blending accederán a través de una memoria cache. Existe una clase que implementa las funcionalidades comunes de la memoria cache que se llama *ROPCache*. A partir de ella derivan las implementaciones específicas para profundidad y color llamadas *ZCacheV2* y *ColorCacheV2* respectivamente.

Se han tenido que hacer pequeñas modificaciones en *ROPCache*, *ZcacheV2*, *ColorCacheV2*, *DAC* y *Blitter* para adaptarse a la infraestructura de compresión implementada.

La infraestructura de compresión ha sido diseñada de tal forma que per-

mita implementar nuevos compresores en el futuro de forma sencilla sin tener que modificar a penas nada del simulador.

Para ello se ha diseñado una interfaz abstracta que deberá cumplir todo nuevo compresor (*CompressorEmulator*) y dos clases singleton (*DepthCompressorEmulator* y *ColorCompressorEmulator*) encargadas de encapsular la implementación concreta del compresor (*MsaaCompressorEmulator*).

Además se han diseñado clases auxiliares para facilitar el trabajo del compresor. Estas son *BitStreamWriter* y *BitStreamReader* que permiten escribir y leer cadenas de bits hacia y desde un buffer de memoria.

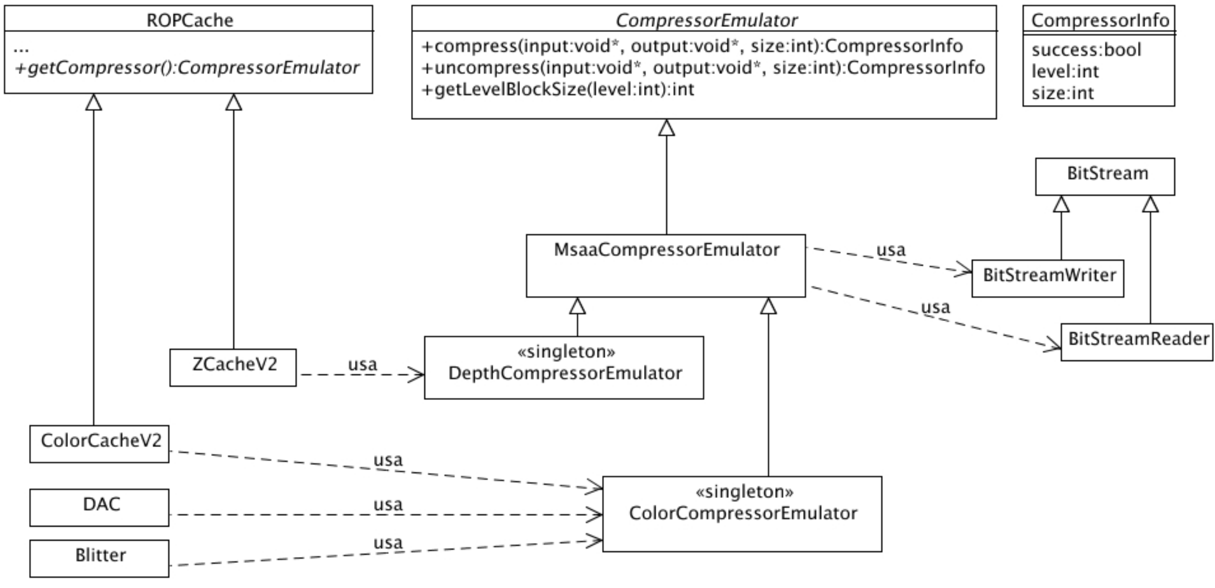


Ilustración 4.21: Diagrama UML de la implementación del compresor en el simulador.

5 Planificación temporal

En el diagrama de Gantt de la Ilustración 5.1 se muestran las tareas planificadas y su distribución en el tiempo de duración previsto del proyecto.

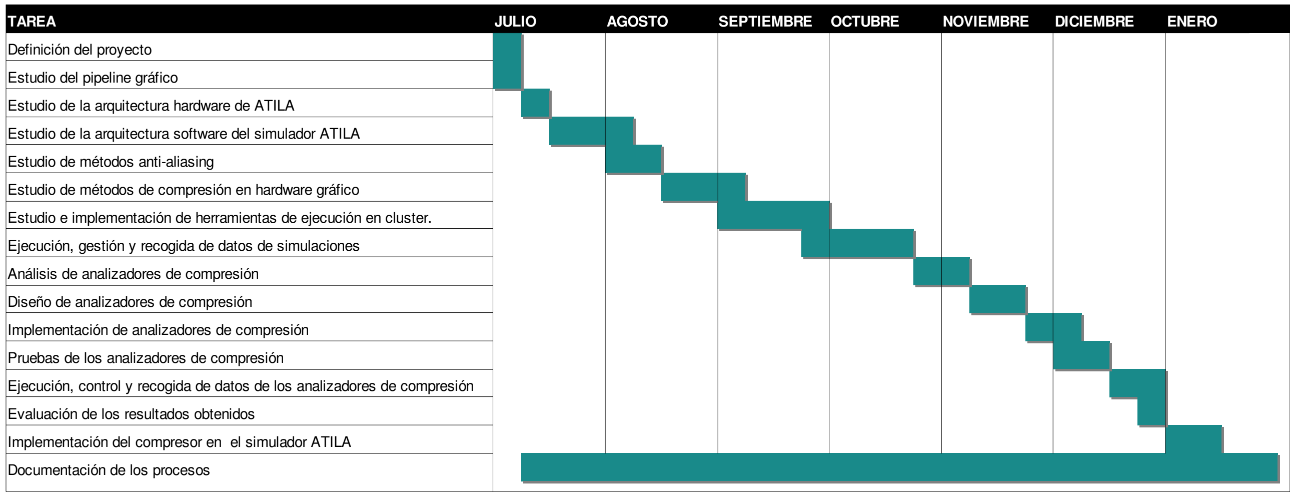


Ilustración 5.1: Planificación temporal de las tareas.

6 Valoración económica

6.1 Análisis del tiempo de realización del proyecto

En este apartado se presenta una estimación de las horas dedicadas a la realización de este proyecto. Los datos se intentan acercar lo máximo posible a la realidad, sin embargo se asumen posibles variaciones en las estimaciones.

La Tabla 6.1 muestra el tiempo de trabajo necesario para el analista, el programador y el grupo de pruebas y ejecución.

TAREA	Analista	Program.	Pruebas
Definición del proyecto	15 h.		
Estudio del pipeline gráfico	20 h.		
Estudio de la arquitectura hardware de ATILA	30 h.		
Estudio de la arquitectura software del simulador ATILA	50 h.	50 h.	
Estudio de métodos anti-aliasing	50 h.		
Estudio de métodos de compresión en hardware gráfico	50 h.	40 h.	
Estudio e implementación de herramientas de ejecución en cluster.		80 h.	
Ejecución, gestión y recogida de datos de simulaciones			100 h.
Análisis de analizadores de compresión	35 h.		
Diseño de analizadores de compresión	30 h.		
Implementación de analizadores de compresión		60 h.	
Pruebas de los analizadores de compresión		40 h.	60 h.
Ejecución, control y recogida de datos de los analizadores de compresión			55 h.
Evaluación de los resultados obtenidos	20 h.		
Implementación del compresor en el simulador ATILA		60 h.	
Documentación de los procesos	160 h.		

Tabla 6.1: Tiempo de trabajo de las tareas.

Tanto el analista como el programador deben estudiar la arquitectura y los métodos de compresión. El analista para realizar el análisis y diseño de los algoritmos de compresión y el programador para su implementación.

El programador además deberá dedicar tiempo a estudiar el entorno de simulación en el cluster ya que deberá programar los scripts de ejecución.

El equipo de pruebas y ejecución se encargará tanto de ejecutar las pruebas del software desarrollado, como de ejecutar, gestionar y recoger los resultados de las simulaciones.

Total horas analista: 460 horas

Total horas programador: 330 horas

Total probador: 215 horas

Total horas: 1005 horas

6.2 Coste económico de proyecto

En este apartado se realiza una estimación del coste económico del proyecto, teniendo en cuenta el número de horas dedicadas por trabajador del apartado anterior. Para calcularlo no se han tenido en cuenta las horas dedicadas a los estudios.

Para hacer los cálculos se usarán los siguientes costes por hora de analista, programador y grupo de pruebas y ejecución.

- Analista: 40 euros / hora
- Programador: 30 euros / hora
- Grupo de pruebas y ejecución: 25 euros / hora

En los costes se incluyen costes directos e indirectos (energía eléctrica, mantenimiento infraestructura de computación, etc).

El coste de un trabajador se calcula como el producto de horas dedicadas al proyecto por el coste de cada hora para dicho trabajador:

- Coste analista = 40 euros / hora * 460 horas = 18400 euros
- Coste programador = 30 euros / hora * 330 horas = 9900 euros
- Coste probador = 25 euros / hora * 215 horas = 5375 euros

El coste total se calcula como la suma del coste del analista, el coste del programador y el coste del grupo de pruebas y ejecución (compuesto por 2 personas):

$$\text{Coste total} = \text{Coste analista} + \text{Coste programador} + 2 \cdot \text{Coste probador}$$

$$\text{Coste total} = 18400 + 9900 + 2 \cdot 5375 = 39050\,\text{euros}$$

7 Conclusiones

El aumento de la calidad de las imágenes en los gráficos 3D implica el uso de técnicas cada vez más sofisticadas que conllevan un aumento significativo del los recursos de hardware necesarios.

Una de estas técnicas, dirigida a disminuir los efectos no deseados del aliasing y por tanto mejorar la calidad de la imagen, es el multi-sampling.

Un problema directamente asociado con el uso de esta técnica es el incremento significativo en las necesidades de ancho de banda de acceso a memoria.

Para tratar de paliar dicho problema nos hemos centrado en la parte de la GPU encargada de acceder a los buffers de profundidad y color ya que representan una parte importante del ancho de banda total necesario (entre un 20 y un 40%).

En la literatura especializada se proponen técnicas de compresión para reducir dicho ancho de banda.

Teniendo en cuenta los requisitos de implementación en hardware y la arquitectura de la GPU ATILA se ha diseñado un conjunto de algoritmos de compresión a partir de algoritmos ya documentados y de modificaciones de los mismos.

A partir de las simulaciones realizadas con trazas de juegos de ordenador reales y el análisis de los algoritmos se ha podido confirmar por un lado el aumento en necesidades de ancho de banda y por otro la efectividad de dicha solu-

ción para disminuirlo, y ha permitido caracterizar el comportamiento de los algoritmos de compresión evaluados teniendo en cuenta tanto el ratio de compresión como el coste de ejecución.

Para las configuraciones de multi-sampling de 4 y 8 muestras por fragmento destaca sobre todos los demás el algoritmo msaa, tanto por alcanzar mejores ratios de compresión como por su menor coste de ejecución. Por tanto dicho algoritmo ha sido implementado en el simulador ATILA.

7.1 Futuras lineas de trabajo

Los algoritmos evaluados han sido analizados de forma aislada del resto de la GPU mediante el uso de capturas de bloques de cache. Para evaluar mejor el impacto global habría que realizar simulaciones con los algoritmos implementados en el simulador. De esta manera, se podría estudiar, no solo el tráfico que se ha conseguido reducir, sino también cómo afecta a los tiempos de ejecución de la GPU.

Otra posible linea de trabajo recomendable sería continuar evaluando el resto de algoritmos de compresión, que pese haber sido explicados, no se han tenido en cuenta en la evaluación para este proyecto. De esta manera podríamos tener una visión más amplia. De la misma manera, habría que estudiar también los algoritmos de compresión del buffer de profundidad, ya que no era viable su estudio junto a los de color, todo en un sólo proyecto.

8 Bibliografía

[1] Moya V. et all: ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. (2006)

[2] Wiki del proyecto ATILA, https://attila.ac.upc.edu

[3] Akenine-moller T, Haines E: Real-Time Rendering. A K Peters, Segunda ed., 2002.

[4] Watt A: 3D Computer Graphics. Pearson, Tercera ed., 2000.

[5] Hasselgren J., Akenine-Moller T.: Efficient Depth Buffer Compression. Graphics Hardware (2006)

[6] Rasmuson, J et all: Exact and Error-bounded Approximate Color Buffer Compression and Decompression. Graphics Hardware (2007)

[7] Morein S., Natale M.: System. Method, and Apparatus for Compression of Video Data using Offset Values. US Patent 6,762,758 (2004)

[8] Elder G. M.: Method and Aparatus for Anti-Aliasing using Floating Point Subpixel Color Values and Compression of Same. US Patent Application 2006/0188161 A1 (2006)

[9] N1 Grid Engine Web, http://gridengine.sunsource.net/

[10] gnuplot, http://www.gnuplot.info/

[11] The R Project for Statistical Computing, http://www.r-project.org/

[12] Roca J. et all: Workload Characterization of 3D Games. (2006)

9 Apéndice

9.1 Organización del CD-ROM

En la carpeta raíz se encuentra este mismo documento en formato pdf para que pueda ser consultado en pantalla.

En la carpeta `cluster-tools` se encuentran los scripts utilizados para las ejecuciones en el cluster.

En la carpeta `gpu3d` se encuentran los ficheros de código fuente implementados en este proyecto para el simulador ATILA.

En la carpeta `resultados` se encuentran organizadas por carpetas los resultados y los gráficos generados para cada una de las trazas utilizadas y los algoritmos evaluados.

Por cada traza se distinguen las siguientes carpetas: `x0`, `x2`, `x4` y `x8`. Se corresponden con los resultados de simular cada una de las configuraciones de multi-sampling citadas en el apartado 4.2. Dentro de cada una de ellas encontraremos:

- `tables32` y `tables64`: Contienen las tablas resultantes de los análisis de los algoritmos de compresión considerando anchos de bus de 32 y 64 bytes respectivamente.

- `graphs32` y `graphs64`: Contienen los gráficos, en pdf, png y ps, para los

análisis de los algoritmos considerando anchos de bus de 32 y 64 bytes respectivamente.

9.2 Archivo de configuración común para todos los experimentos

```
Statistics = TRUE
StatisticsRate = 100000
PerFrameStatistics = TRUE
PerBatchStatistics = FALSE
StatsFile = "stats.cycles.csv.gz"
StatsFilePerFrame = "stats.frames.csv.gz"
StatsFilePerBatch      =      "stats.bat-
ches.csv.gz"

DumpSignalTrace = FALSE
StartSignalDump = 0
SignalDumpFile = "signaltrace.txt"
SignalDumpCycles = 10000

GenerateFragmentMap = FALSE
FragmentMapMode = 3

DoubleBuffer = FALSE

ObjectSize0 = 512
BucketSize0 = 131072
ObjectSize1 = 4096
BucketSize1 = 32768
ObjectSize2 = 64
BucketSize2 = 65536
UseACD = FALSE

[GPU]

NumVertexShaders = 8
NumFragmentShaders = 4
NumStampPipes = 4

[COMMANDPROCESSOR]
PipelinedBatchRendering = FALSE


[MEMORYCONTROLLER]

MemorySize = 201326592
MemoryClockMultiplier = 1
MemoryFrequency = 1
MemoryBusWidth = 64
MemoryBuses = 4
SharedBanks = FALSE
BankGranurality = 1024
BurstLength = 16
ReadLatency = 10
WriteLatency = 5
WriteToReadLatency = 5
MemoryPageSize = 4096
OpenPages = 8
PageOpenLatency = 13
MaxConsecutiveReads = 16
MaxConsecutiveWrites = 16
CommandProcessorBusWidth = 8
StreamerFetchBusWidth = 64
StreamerLoaderBusWidth = 64
ZStencilBusWidth = 64
ColorWriteBusWidth = 64
DACBusWidth = 64
TextureUnitBusWidth = 64
MappedMemorySize = 16777216
ReadBufferLines = 32
WriteBufferLines = 64
RequestQueueSize = 128
ServiceQueueSize = 32

MemoryControllerV2 = TRUE

V2MemoryChannels = 8
V2BanksPerMemoryChannel = 8
V2MemoryRowSize = 2048
V2BurstElementsPerCycle = 2

V2MaxChannelTransactions = 32

V2ChannelInterleaving = 256

V2BankInterleaving = 256

V2ChannelScheduler = 3

V2PagePolicy = 1

V2PerfectMemory = FALSE

[STREAMER]

IndicesCycle = 2
IndexBufferSize = 2048
InputRequestQueueSize = 128
AttributesCycle = 8
InputCacheLines = 32
InputCacheLineSize = 256
InputCachePortWidth = 16
InputCacheRequestQueueSize = 8
InputCacheInputQueueSize = 8
OutputFIFOSize = 512
OutputMemorySize = 512
VerticesCycle = 2
AttributesSentCycle = 4


[VERTEXSHADER]

ExecutableThreads = 12
InputBuffers = 4
ThreadResources = 128
ThreadRate = 1
FetchRate = 1
ThreadGroup = 1
```

```
LockedExecutionMode = FALSE
ScalarALU = FALSE
ThreadWindow = TRUE
FetchDelay = 0
SwapOnBlock = FALSE
InputsPerCycle = 1
OutputsPerCycle = 1
OutputLatency = 11


[PRIMITIVEASSEMBLY]

VerticesCycle = 2
TrianglesCycle = 2
InputBusLatency = 10
AssemblyQueueSize = 32


[CLIPPER]

TrianglesCycle = 2
ClipperUnits = 2
StartLatency = 1
ExecLatency = 6
ClipBufferSize = 32


[RASTERIZER]

TrianglesCycle = 2
SetupFIFOSize = 32
SetupUnits = 2
SetupLatency = 10
SetupStartLatency = 4
TriangleInputLatency = 2
TriangleOutputLatency = 2
TriangleSetupOnShader = FALSE
TriangleShaderQueueSize = 8
StampsPerCycle = 4
MSAASamplesCycle = 2
OverScanWidth = 4
OverScanHeight = 4
ScanWidth = 16
ScanHeight = 16
GenWidth = 8
GenHeight = 8
RasterizationBatchSize = 4
BatchQueueSize = 16
RecursiveMode = TRUE
DisableHZ = FALSE
StampsPerHZBlock = 16
HierarchicalZBufferSize = 262144
HZCacheLines = 8
HZCacheLineSize = 16
EarlyZQueueSize = 256
HZAccessLatency =  5
HZUpdateLatency =  4
HZBlocksClearedPerCycle = 256
NumInterpolators = 4
ShaderInputQueueSize = 128
ShaderOutputQueueSize = 128
ShaderInputBatchSize = 64
TiledShaderDistribution = TRUE
VertexInputQueueSize = 32
ShadedVertexQueueSize = 512
TriangleInputQueueSize = 32
TriangleOutputQueueSize = 32
GeneratedStampQueueSize = 256
EarlyZTestedStampQueueSize = 32
InterpolatedStampQueueSize = 16
ShadedStampQueueSize = 640
EmulatorStoredTriangles = 64

[FRAGMENTSHADER]

ExecutableThreads = 2048
InputBuffers = 16
ThreadResources = 8192
ThreadRate = 4
FetchRate = 2
ThreadGroup = 16
LockedExecutionMode = TRUE
ScalarALU = TRUE
ThreadWindow = TRUE
FetchDelay = 4
SwapOnBlock = FALSE
InputsPerCycle = 4
OutputsPerCycle = 4
OutputLatency = 11
TextureUnits = 1
TextureRequestRate = 1
TextureRequestGroup = 64
AddressALULatency = 6
FilterALULatency = 4

TextureBlockDimension = 2
TextureSuperBlockDimension = 4
TextureRequestQueueSize = 512
TextureAccessQueue = 256
TextureResultQueue = 4
TextureWaitReadWindow = 32
TwoLevelTextureCache = TRUE
TextureCacheLineSize = 64
TextureCacheWays = 8
TextureCacheLines = 8
TextureCachePortWidth = 4
TextureCacheRequestQueueSize = 32
TextureCacheInputQueue = 32
TextureCacheMissesPerCycle = 8
TextureCacheDecompressLatency = 1
TextureCacheLineSizeL1 = 64
TextureCacheWaysL1 = 16
TextureCacheLinesL1 = 16
TextureCacheInputQueueL1 = 32


[ZSTENCILTEST]

StampsPerCycle = 1
BytesPerPixel = 4
DisableCompression = FALSE
ZCacheWays = 4
ZCacheLines = 16
ZCacheStampsPerLine = 16
ZCachePortWidth = 32
ZCacheExtraReadPort = TRUE
ZCacheExtraWritePort = TRUE
ZCacheRequestQueueSize = 8
ZCacheInputQueueSize = 8
ZCacheOutputQueueSize = 8
BlockStateMemorySize = 262144
BlocksClearedPerCycle = 1024
CompressionUnitLatency = 8
DecompressionUnitLatency = 8
#ZQueueSize = 64
```

```
InputQueueSize = 8
FetchQueueSize = 64
ReadQueueSize = 16
OpQueueSize = 4
WriteQueueSize = 8
ZALUTestRate = 1
ZALULatency = 2

[COLORWRITE]

StampsPerCycle = 1
BytesPerPixel = 4
DisableCompression = FALSE
ColorCacheWays = 4
ColorCacheLines = 16
ColorCacheStampsPerLine = 16
ColorCachePortWidth = 32
ColorCacheExtraReadPort = TRUE
ColorCacheExtraWritePort = TRUE
ColorCacheRequestQueueSize = 8
ColorCacheInputQueueSize = 8
ColorCacheOutputQueueSize = 8
BlockStateMemorySize = 262144
BlocksClearedPerCycle = 1024
CompressionUnitLatency = 8
DecompressionUnitLatency = 8
#ColorQueueSize = 64
InputQueueSize = 8
FetchQueueSize = 64
ReadQueueSize = 16
OpQueueSize = 4
WriteQueueSize = 8
BlendALURate = 1
BlendALULatency = 2

[DAC]

BytesPerPixel = 4
BlockSize = 256
BlockUpdateLatency = 1
BlocksUpdatedPerCycle = 1024
BlockRequestQueueSize = 32
DecompressionUnitLatency = 1
RefreshRate = 5000000
SynchedRefresh = TRUE
RefreshFrame = TRUE
```