Sistemas Informáticos
Curso 2004-2005

Aplicaciones de gráficos 3D sobre plataformas reconfigurables

Alfonso de Torres Pérez
José Manuel Sánchez Díaz
Francisco Javier Dávila Foncuverta

Dirigido por:
Prof. Marcos Sánchez-Élez Martín
Dpto. Arquitectura de Computadores Y Automática

Facultad de Informática
Universidad Complutense de Madrid

ÍNDICE

1 RESUMEN

En esta memoria analizaremos un algoritmo de renderización de imágenes 3D para su implementación sobre una arquitectura reconfigurable. La renderización de imágenes 3D es un proceso que conlleva una gran carga computacional con una importante restricción en el tiempo de ejecución para conseguir resultados interactivos. Esto hace que sea un algoritmo perfecto para ejecutarlo en arquitecturas reconfigurables, las cuales podrán aprovecharse del paralelismo inherente en este tipo de aplicaciones. En concreto, usaremos la arquitectura reconfigurable de grano grueso MorphoSys. En esta memoria demostramos que es posible implementar el algoritmo de renderización en dicha arquitectura reconfigurable consiguiendo valores de 6480 fps.

En el contenido de la memoria, explicaremos el algoritmo de renderización, y para qué se utiliza. Describiremos la arquitectura reconfigurable Morphosys, sobre la que implementamos el algoritmo de renderización. Se plantearán y explicarán las distintas estrategias que hemos estudiado para implementar el algoritmo en esta plataforma. Por último, mostraremos los resultados experimentales obtenidos para las estrategias elegidas, y las conclusiones sobre la realización del proyecto.

En inglés:

In this report we analyze a 3D image renderization algorithm and its implementation in a reconfigurable architecture. The 3D image renderization is a highly computational process with an important restriction in execution time due to the interactive results. Therefore, the algorithm is perfect for executing in reconfigurable architectures, which can take advantage to the paralelism of this kind of applications. Specifically, we use the coarse grain reconfigurable architecture MorphoSys. In this report, we probe that it is possible to implement the renderization algorithm in this reconfigurable architecture, obtaining values of 6480 fps.

In the content of this report, we explain the renderization algorithm, and its uses. We describe the reconfigurable architecture MorphoSys, in which we implement the renderization algorithm.We discuss the different strategies studied to implement the algorithm in this platform. Finally, we show the experimental results obtained for the strategies chosen, and several conclusions about the realization of the project.

2 INTRODUCCIÓN AL PROYECTO

Como ya hemos dicho, este proyecto va a consistir en como implementar un algoritmo de renderización en una arquitectura reconfigurable, Morphosys en nuestro caso. Para llevar a cabo este proyecto, lo primero ha sido un trabajo de investigación dirigido sobre tres puntos.

Primero la investigación se centro en la búsqueda de información sobre rendering, buscando posibles modos de implementar nuestro algoritmo, así como distintas técnicas para posteriormente aplicarlas sobre el algoritmo y obtener imágenes de buena calidad. La elección final fue un algoritmo de scan-line programado en C++, que se explicará a lo largo de este documento, y sobre el que fuimos añadiendo distintas mejoras, como la inserción del Z-Buffer, sombreado e iluminación.

Lo siguiente fue buscar información sobre hardware reconfigurable de grano grueso, muy importante, por un lado porque el proyecto consistía en implementar el algoritmo sobre una arquitectura de este tipo, y por otro porque teníamos un total desconocimiento de estas arquitecturas de grano grueso. No solo centramos la investigación en Morphosys, que es la arquitectura sobre la que trabajaremos en el proyecto, sino que también buscamos información de otras arquitecturas, su funcionamiento y de métodos para programar sobre ellas, que nos ayuden a la hora de planificar y diseñar nuestro algoritmo sobre Morphosys.

Por último, una vez que hemos tenido diseñado el algoritmo, y que ya sabíamos más sobre arquitecturas reconfigurables, y en particular, habiendo ya estudiado Morphosys con detalle, pasamos a la implementación. Para esto tuvimos que planificar y dividir nuestro código en C++ de la mejor manera posible para plasmarlo sobre la arquitectura y buscar la mejor estrategia posible para sacarle el máximo rendimiento a Morphosys. Una vez llegados a este punto, utilizamos en simulador de Morphosys, “mulate”, para hacer una pequeña simulación de lo que sería la parte más crítica del proyecto, que es el renderizado de los píxeles, y ver que era posible ejecutarlo sobre Morphosys.

Los resultados de este proyecto están reflejados en el artículo: José Manuel Sánchez, Alfonso de Torres, Javier Dávila y Marcos Sánchez-Elez *“Diseño e Implementación de un Algoritmo de Renderización sobre Plataforma Reconfigurable”* publicado en las V Jornadas de Computación Reconfigurable y Aplicaciones (JCRA’2005) que se celebran dentro del I Congreso Español de Informática (CEDI’2005) del 13 al 16 de Septiembre de 2005 en el Palacio de Exposiciones y Congresos de Granada.

2.1 Introducción

La informática gráfica es el campo de la computación que se encarga de todas las facetas de la creación de imágenes por ordenador, y es uno de los campos de la informática que ha experimentado un mayor desarrollo, gracias a los avances tecnológicos que se han producido en este campo. Esto es debido a que la informática gráfica tiene muchos promotores, como son las empresas de hardware, empresas de software gráfico o centros de investigación, dado que podemos aplicarla a muchos campos distintos del conocimiento. Dichos campos abarcan el diseño (de cualquier tipo de estructura), la infografía (para el desarrollo de animaciones), interpretación y representación de datos meteorológicos, cartográficos, o de cualquier otro tipo, simulación de aparatos ya diseñados, y sobre todo las compañías dedicadas al cine y videojuegos, que mueven billones de dólares al año.

A lo largo del tiempo, se han ido desarrollando diversos estándares para aplicaciones gráficas, mediante librerías como GKS, PHIGS u OpenGL, sin embargo, hay tanta demanda en el apartado gráfico, que no da tiempo a que dichas librerías lleguen a formar un estándar, por lo que son los propios desarrolladores los que tratan de imponer sus propios estándares.

El nacimiento de la informática gráfica tiene lugar en la universidad de Boston, en los años sesenta, gracias a un dispositivo que realizaba representaciones gráficas del cálculo numérico. Este prototipo de llamó SketchPad, y fue el precursor de las pantallas gráficas, las cuales se empezaron a desarrollar en el año 1965. El fabricante fue Tektronix, y desarrolló pantallas basadas en tubos de almacenamiento **[1]**. A principios de los años setenta, se incorporó a las pantallas gráficas la tecnología de tubos de rastreo de los televisores convencionales, lo cual abarató mucho el coste de fabricación. De hecho, aunque a día de hoy se siguen utilizando algunos monitores de tubos de almacenamiento, la mayoría de monitores se basan en tubos de rastreo. Así mismo, comenzaron a fabricarse los primeros plotters, que tuvieron una gran demanda entre los diseñadores industriales. Con el comienzo de la década de los noventa, se empezó a comercializar las primeras impresoras láser, las cuales predominan en el mercado actual.

Una de las partes más importantes a la hora de desarrollar aplicaciones gráficas son las librerías gráficas. Estas son bibliotecas que hacen de enganche entre el hardware gráfico y los programas de aplicaciones. Para ello, cuentan con un API de funciones que facilitan la programación gráfica. La primera librería gráfica salió a mediados de los ochenta, con el nombre de GKS. Dicha librería se amplió con GKS3D, que daba cobertura a los gráficos 3D, ya que su predecesor carecía de esta función, y quedó obsoleto en seguida. Para competir con este estándar europeo, desde Estados Unidos se desarrolló PHIGS, mejorándose más tarde con PHIGS+.

A principio de los años noventa surgió OpenGL, librería basada en GL, una librería de Silicon Graphics. OpenGL está desarrollada por algunas de las empresas más importantes, como SGI, IBM o Microsoft **[1]**, siendo actualmente la librería gráfica más extendida.

Dentro de la informática gráfica, los renderizadores juegan un papel fundamental. Un renderizador es la aplicación que transforma un escenario modelado y diseñado casi siempre en 3D, y calcula una representación lo más realista posible en las dos dimensiones de la pantalla.

En los siguientes apartados, se explicará con más detalle en qué consisten los distintos tipos de renderizadores que existen actualmente, así como los componentes que conforman un renderizador. En concreto, mostraremos cómo se crean las primitivas gráficas, base de toda imagen gráfica, así como el proceso de creación de imágenes gráficas, en el que se explicarán cada una de las partes que forman el renderizador, cómo partiendo de una imagen hecha a base de polígonos, calculamos el color exacto de cada píxel de la imagen. Dada su importancia para la obtención de renderizadores potentes, que creen imágenes realistas, explicaremos en las dos últimas secciones, especialmente, el cálculo de la proyección de un objeto, y el de la iluminación de sus vértices; dos técnicas de la renderización que mejorarán significativamente el aspecto de cualquier imagen, y que son imprescindibles en los renderizadores actuales.

2.2 Renderizadores

La renderización es un término general que incluye los procesos anteriormente explicados. Dichos procesos tienen como objetivo proporcionar una visión en dos dimensiones de un objeto en tres dimensiones.

En el proceso de creación de la imagen, las bibliotecas gráficas usan algoritmos que convierten la proyección de los objetos en píxeles concretos del dispositivo gráfico. Ese proceso, más la aplicación de proyecciones e iluminación se suele agrupar en un algoritmo independiente que es lo que se conoce propiamente como renderizador.

Los renderizadores pueden ser implementados a través de diferentes técnicas. Las técnicas van desde el rénder de alambre (wireframe rendering), pasando por el rénder basado en polígonos, hasta las técnicas más modernas como: *Scanline Rendering, raytracing o radiosity*

2.2.1 Wireframe rendering

Wireframe rendering [2] es un algoritmo de renderización del que resulta una imagen semitransparente, de la cual sólo se dibujan las aristas de la malla que constituye al objeto. De ahí su nombre. Es el algoritmo de rendering más básico.

Casi nunca se emplea en la representación final de una imagen, pero sí en su edición, debido a la escasa potencia de cálculo necesaria (comparada con otros métodos).
Para conseguir una imagen en wireframe sólo tenemos que tener en cuenta las posiciones de los puntos en el espacio tridimensional y las uniones entre ellos para formar los polígonos.

Habitualmente estas imágenes no tienen en cuenta la presencia de luces en la escena.

Figura 1. Escena renderizada con wireframe

2.2.2 Scanline rendering

Para seleccionar el píxel del polígono que vamos a renderizar en cada momento, se recorre sistemáticamente dicho polígono por líneas en un determinado orden. Scanline [2] renderiza una línea de pixels del polígono en cada momento.

Estos métodos tienen en cuenta sólo la iluminación de los objetos, y no las propiedades físicas de estos. Son métodos muy rápidos Scanline es quizás el mejor compromiso entre calidad y velocidad en el rendering. Esta característica puede permitir obtener imágenes de manera interactiva. La naturaleza del algoritmo permite paralelizar sus cómputos. Es el método elegido para implementar nuestro algoritmo de renderización en el proyecto.

Scanline es por lo tanto el método preferido para generar la mayoría de los gráficos por ordenador para las películas de animación. También es el método usado en videojuegos y en la mayoría del software gráfico para ingeniería (vía OpenGL normalmente) Algoritmos de scanline están también implementados en hardware en muchos casos.

El proceso que siguen los algoritmos de scanline para renderizar polígonos es explorarlos de vértice a vértice y convertir en series de líneas horizontales o verticales(normalmente horizontales) que serán convertidas a coordenadas de píxeles. Dependiendo de los atributos del polígono, tales como su color, a cada píxel se le da su color correspondiente.

Además se usa un algoritmo de profundidad, llamado z-buffer, que consiste en solo considerar los píxels con coordenadas más cercanas a la cámara respecto a los más alejados. Así solo se renderizan los objetos visibles de la escena, y no se renderizan píxels innecesarios.

Los algoritmos de scanline son buenos soportando diferentes implementaciones en cuanto a las primitivas que usan. Pueden actuar sobre polígonos de diferente número de lados, sin variar apenas su estructura. En nuestro caso usaremos triángulos como primitiva.

Características fotorrealistas como reflexiones, transparencias, múltiples focos de luz, sombras, áreas de luz... pueden ser añadidas a los algoritmos de scanline sin producir un descenso brusco en la productividad del mismo.

Programas que implementan este tipo de algoritmos son 3D Studio Max **[3]**, RenderMan **[4]** o Electric Image **[5]**.

Figura 2. Escena renderizada con scanline

2.2.3 Raytracing

Ray Tracing [2] o trazado de rayos es una familia de algoritmos síntesis de imágenes tridimensionales. Propuesto inicialmente por Turner Whitted en 1980, está basado en el algoritmo de determinación de superficies visibles de Arthur Appel denominado Ray Casting (1968).

En el algoritmo Ray Casting se determinan las superficies visibles en la escena que se quiere sintetizar trazando rayos desde el observador (cámara) hasta la escena a través del plano de la imagen. Se calculan las intersecciones del rayo con los diferentes objetos de la escena y aquella intersección que esté más cerca del observador determina cuál es el objeto visible.

El algoritmo de trazado de rayos extiende la idea de trazar los rayos para determinar las superficies visibles con un proceso de sombreado (cálculo de la intensidad del píxel) que tiene en cuenta efectos globales de iluminación como pueden ser reflexiones, refracciones o sombras arrojadas.

Para simular los efectos de reflexión y refracción se trazan rayos recursivamente desde el punto de intersección que se está sombreando dependiendo de las características del material del objeto intersectado.

Para simular las sombras arrojadas se lanzan rayos desde el punto de intersección hasta las fuentes de luz. Estos rayos se conocen con el nombre de rayos de sombra (shadow rays).

El algoritmo básico de trazado de rayos fue mejorado por Robert Cook (1985) para simular otros efectos en las imágenes mediante el muestreo estocástico; entre estos

efectos podemos citar el desenfoque por movimiento (blur motion), la profundidad de campo o el submuestreo para eliminar efectos de aliasing en la imagen resultante.

En la actualidad, el algoritmo de trazado de rayos es la base de otros algoritmos más complejos para síntesis de imágenes (Photon Mapping, Metropolis, ...) que son capaces de simular efectos de iluminación global complejos como la mezcla de colores (color bleeding) o las cáusticas.

Los métodos de ray tracing son muy lentos (los que más), por lo que no se usa para animación en 3D; pero consiguen, en general, los mejores efectos fotorrealistas ya que son capaces de calcular reflexiones, refracciones, transparencias, sombras, etc.

El fallo más importante de estos métodos es precisamente en el sombreado, que, si bien, es más realista que con métodos scanline, lo es menos que con radiosity. Dentro del raytracing, se encuentran básicamente dos submétodos:

- **photom tracing:** se calculan las trayectorias directas, entre las fuentes de iluminación y el observador. Método poco usado.

- **el visible tracing:** se calculan las trayectorias inversas, entre el observador y las fuentes de iluminación. Es el método más usado, y es el que implementan programas como: POV-Ray **[6]**, Real3D **[7]**, Animation Master **[8]**.

Figura 3. Escena renderizada con raytracing

2.2.4 Radiosity

La técnica de radiosity [13] o radiosidad engloba un conjunto de algoritmos de renderización que tratan de resolver el problema básico de la renderización de la forma más realista posible.

El problema es que el transporte de la luz sólo se puede modelar de forma óptima considerando que cada fuente luminosa emite un número enorme de fotones, que

rebotan al chocar contra una superficie describiendo una cantidad de trayectorias imposibles de simular en un computador.

La radiosidad emplea métodos como el uso de algoritmos de Monte Carlo para resolver este problema de forma estadística. Calculan las ecuaciones de radiosidad para cada objeto, en función de la energía que reciben, emiten y propagan.

Son métodos independientes del punto de vista del observador, muy lentos, pero que producen los mejores efectos de sombreado/iluminación. Tienen en cuenta las propiedades físicas relacionadas con la reflexión, pero no la refracción.

El auge de la radiosidad y otros métodos eficientes de renderización han posibilitado un auge en la infografía, siendo muy habitual encontrar por ejemplo películas que aprovechan estas técnicas para realizar efectos especiales.

Programas que implementan radiosity son por ejemplo LightWave 3D, y que combinan radiosity con raytracing, Lightscape

Figura 4. Escena renderizada con radiosity

2.3 Creación de imágenes gráficas

En esta sección se explicarán las primitivas gráficas que se usan como base a la hora de diseñar escenarios con gráficos complejos. Además, se mostrarán los actuales formatos de archivos gráficos más usados, tratando brevemente las técnicas de cada uno de los formatos para guardar la información gráfica.

2.3.1 Creación de imágenes

La creación de imágenes supone la representación 2D de una imagen formada por distintos elementos, generalmente descritos en un espacio 3D. Por lo tanto, la creación de imágenes supone el cálculo de la visibilidad de los objetos, así como la iluminación,

reflejos y sombreado para obtener una imagen 2D lo más realista posible de nuestro escenario 3D desde el punto de vista elegido.

Para tener una visión completa de la creación de imágenes, trataremos en esta sección tres apartados importantes:

- Los estándares de colores, con los cuales representamos la información de color de cada píxel para mostrarlo por pantalla.

- El proceso de creación de imágenes, en el que mostraremos todos los pasos que lleva a cabo la renderización, desde que el escenario 3D es creado con una aplicación de diseño, hasta que la imagen renderizada se muestra por pantalla. En este apartado, trataremos brevemente las librerías gráficas, los estándares gráficos de funciones con los que crear fácilmente imágenes. Usaremos también un ejemplo de librería gráfica, OpenGL, para ilustrar cada una de las operaciones que un renderizador debe seguir.

- La pantalla gráfica y su funcionamiento, cómo interpreta la imagen renderizada para mostrarla por pantalla. También describiremos brevemente los tipos actuales de pantallas que existen.

2.3.2 Estándares de colores

Un estándar de color es un conjunto de parámetros que se adoptan como representación de los colores que puede tener una imagen. En la actualidad coexisten diversos estándares, de los cuales el predominante es el RGB.

El modelo RGB tiene tres componentes reales, uno para el rojo, otro para el verde, y otro para el azul, cuyos valores varían entre 0 y 1. De esta manera, sumando la cantidad de cada componente, se obtiene el color real. Prácticamente todas las librerías gráficas usan este estándar de color. También existen otros estándares, como CMY, que es el complementario del RGB, o el estándar YIQ, el cual se usa para la transmisión de las imágenes de la televisión. Una extensión a RGB es RGBA, con una cuarta componente que indica el grado de transparencia que tendrá el color. De esta forma, el color variará desde la opacidad a la transparencia total.

Un último estándar de color, más complejo, es HSV. Dicho estándar, además de especificar el color, da valores también para el grado de transparencia y el sombreado. El estándar RGB no incluye este tipo de información, aunque se puede calcular con algoritmos de iluminación y sombreado.

2.3.3 Proceso de creación de imágenes

El proceso de creación de una imagen tiene tres partes bien diferenciadas: la entrada, el proceso gráfico y la salida. Nos centraremos en los dos últimos apartados, ya que la entrada de datos de la imagen (teclado, ratón, escáner) no es importante para la elaboración del proyecto.

Figura 5. Proceso de creación de una imagen

Para crear una imagen, usaremos una aplicación de diseño de modelos, como simulaciones, las cuales dejarán el modelo totalmente definido. Este modelo es usado por la biblioteca gráfica que se esté usando, de forma que ésta traducirá dicho modelo a una imagen sobre un dispositivo virtual (como una pantalla, por ejemplo), que le sirve de referencia para crear la imagen. Una vez creada la imagen, se traduce sobre un dispositivo real, como la pantalla del ordenador, para que pueda ser visualizada. Entraremos en más detalle en la funcionalidad de los dos últimos apartados del proceso de creación de imágenes:

2.4 Ejemplo de renderizadores comerciales (librerías gráficas)

Como hemos visto en la introducción a la informática gráfica, definimos las librerías gráficas como un API de funciones que nos ayuden a construir los objetos e imágenes que mostraremos por la pantalla. Hasta la actualidad, se han usado distintas librerías gráficas, como GKS o PHIGS (comentadas anteriormente), pero la mayoría han quedado ya obsoletas, dejando paso a la librería gráfica adoptada como "estándar" por la mayoría de compañías, OpenGL. A continuación, explicaremos brevemente sus características principales.

Introducción

OpenGL se desarrolló en base a la librería gráfica GL (Graphics Library), considerada su antecesora. Dicha librería hacía un renderizador hardware, el cual consiste en usar los dispositivos hardware cuya finalidad es puramente gráfica (tarjetas gráficas) para realizar casi todo el trabajo. De esta forma, se descarga al procesador principal de mucha parte del trabajo de renderización. El problema de GL es que no era portable, y dependía de la plataforma hardware donde se implantó originalmente. Por ello, diversos fabricantes de software y hardware (SGI, Microsoft, Intel, IBM…) usaron GL para crear OpenGL, una librería de código abierto portable, que se podía usar en cualquier plataforma. La primera versión se creó en 1992, y a partir de ahí, estas empresas son las que se dedican a diseñar mejoras para OpenGL y lanzar nuevas versiones.

Características generales

OpenGL es portable, por lo que no importa el hardware o sistema operativo donde se aplique. La librería consta de más de un centenar de funciones sencillas de utilizar, con las que crear aplicaciones gráficas utilizando las técnicas ya comentadas de

renderización: OpenGL está preparado para crear aplicaciones gráficas con iluminación, uso de texturas, transformaciones de objetos y sombreado, por destacar algunas de las técnicas más interesantes. Por ello, con OpenGL podremos diseñar cualquier tipo de aplicación gráfica avanzada.

Además, entre las funciones OpenGL dispone de varias primitivas para dibujar figuras geométricas simples, como pueden ser puntos, rectas, polígonos, cubos o esferas. La sintaxis típica de una función en OpenGL consiste en escribir "gl" seguido de la acción a realizar, más el número de parámetros que tendrá la función y el tipo de dichos parámetros. Por ejemplo, la siguiente función:

glVertex2f(GLfloat x, GLfloat y)

Dibuja un punto de dos coordenadas en punto flotante en la posición que especifiquen los parámetros *x* e *y*. Igualmente ocurre con las constantes, las cuales se escriben en mayúscula, empezando por "GL_", más el nombre de la constante. OpenGL también dispone de sus propios tipos, los cuales son equivalentes a los tipos de la mayoría de los lenguajes. Su sintaxis consiste en anteponer al nombre del tipo el texto "GL". Por ejemplo:

GLint, GLfloat

Funcionamiento

OpenGL renderiza las imágenes pasando vértice a vértice de cada objeto de la escena por su tubería gráfica. Dicha tubería gráfica se implementa multiplicando las coordenadas del vértice por una serie de matrices que delimitan su proyección, posición y vista en la pantalla. A continuación explicamos el funcionamiento de esta tubería gráfica.

Dentro del proceso de las imágenes por parte de las librerías, el cual es distinto dentro de cada una, mostraremos el caso de OpenGL, por ser la librería más extendida en la actualidad. OpenGL tiene una tubería gráfica que usa matrices, de manera que cada uno de los vértices de los polígonos de los que se compone la imagen debe pasar por dicha tubería para llegar a dibujarse en la pantalla.

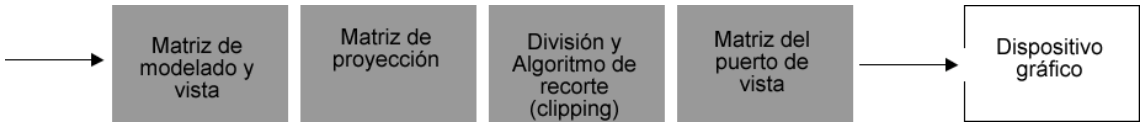

Figura 6. Tubería gráfica para vértices de OpenGL

Cada vértice está descrito como una matriz de cuatro filas y una columna, donde están colocadas tres coordenadas del vértice, y un factor de escalado, que nos indica las modificaciones de escalado del vértice (por defecto vale 1). Los pasos que debemos dar por la tubería gráfica para representar cada vértice son los siguientes:

A) Matriz de modelado y vista: Multiplicamos la matriz del vértice por esta matriz, que es la que tiene la representación de las coordenadas oculares (las de la cámara). De esta manera, conseguimos las coordenadas de ese vértice en el sistema de

coordenadas de la cámara. Esta matriz es la que nos sirve para aplicar transformaciones a los vértices, de escalado, rotación o traslación, simplemente realizando estas transformaciones sobre la matriz de modelado y vista, y multiplicándola por la matriz del vértice. Intuitivamente, sería más lógico aplicar la transformación en el vértice que en la matriz de modelado, pero de esta manera nos ahorramos cálculos si tenemos una serie de vértices a los que aplicar la misma transformación, ya que sólo tenemos que aplicarlos una vez a la matriz de modelado.

B) Matriz de proyección: En esta matriz guardamos una representación del tipo de proyección que tendrá la imagen, obteniendo las coordenadas del vértice proyectado al multiplicar la matriz del vértice con esta matriz de proyección. Este vértice está ya definido en 2D, aunque todavía incluye la información sobre la profundidad en la escena, que nos será útil más adelante, en el cálculo del z-buffer. Más adelante, explicaremos los tipos de proyecciones más comunes en el diseño gráfico.

Una vez obtenemos la matriz con las coordenadas del vértice transformadas, calculamos la iluminación sobre el mismo. Con esto, obtendremos el color real de ese vértice. Las técnicas de cálculo de iluminación sobre vértices las explicaremos posteriormente. Con el color transformado, calcularemos ahora la división y el recorte.

C) División y recorte: El siguiente paso consiste en dividir las coordenadas del vértice entre su factor de escalado, para obtener las coordenadas del vértice normalizadas. Con esto, eliminamos el factor de escala, quedándonos únicamente con los valores del vértice. Además, en este paso, hacemos también el recorte, o clipping, que consiste en eliminar las partes de la escena que quedan fuera de la ventana, para no seguir procesándolas. Para ello, se diseñan algoritmos que tienen en cuenta el tamaño de la ventana para, usando determinadas estrategias, ir eliminando las líneas o polígonos que sobran, o partir cada uno de ellos justo donde corte con la ventana.

D) Matriz del puerto de vista: Esta matriz representa la ventana a través de la cual se va a reproducir la imagen. Multiplicando la matriz del vértice por esta matriz, obtenemos la ubicación exacta en 2D de su píxel correspondiente en la ventana.

Cuando el vértice ha llegado a este punto, ya se ha transformado en un píxel de la ventana de vista, con su color definido. Sin embargo, el proceso del cálculo del píxel no acaba aquí, ya que debe procesar distintas técnicas avanzadas que den un aspecto más realista, como son la eliminación de partes ocultas, sombreado e iluminación, o aplicación de texturas sobre los objetos:

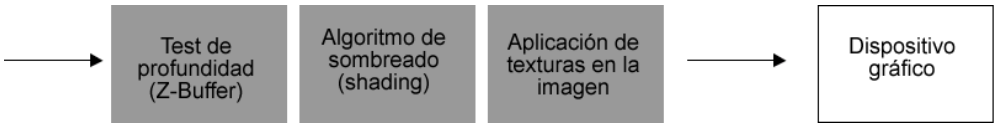

Figura 7. Tubería gráfica para píxels

E) Test de profundidad: Mediante este test, determinaremos las superficies visibles en coordenadas de la pantalla, para desechar aquellas partes que quedan ocultas en la imagen. Para ello, se cuenta con una memoria llamada z-buffer, la cual es un array de dos dimensiones que contiene tantas posiciones como píxels habrá en la ventana de la imagen. El proceso consiste en comparar la profundidad de los píxels que tengan las

mismas coordenadas (x,y), para quedarse con el píxel más cercano a la pantalla. Como inicialización, todas las celdas deberán tener la profundidad máxima, para guardar siempre el primer píxel que se compare en una posición determinada. En la siguiente figura, se muestra un algoritmo básico de su implementación en pseudocódigo:

```
P = (Conjunto de polígonos de la imagen)

//Se mira cada pixel de la proyección de cada polígono
for each polígono p of P do

        for each pixel pi of p do

                /* Se guarda el color del pixel si es más cercano
                   a la pantalla que el que hay guardado:
                */
                if (pi.z < z-buffer[pi.x,pi.y])

                        z-buffer[pi.x,pi.y] = pi.z;
                        color-buffer[pi.x,pi.y] = pi.color;

                fi

        od

od
```

Figura 8. Algoritmo de Z-buffer

Para seleccionar el píxel del polígono que vamos a renderizar en cada momento, se usa un algoritmo de 'scan-line', el cual consiste en recorrer los píxels del polígono en líneas, generalmente de izquierda a derecha y de abajo a arriba, aunque no es imprescindible recorrer el polígono en este sentido.

Como se puede ver en el algoritmo, nos hace falta saber la profundidad de cada píxel para comprobar si dicho píxel se verá o no en la pantalla. Los renderizadores no tienen este valor, ya que el volumen de datos a procesar sería enorme. Sin embargo, contamos con la profundidad de cada uno de los vértices que compone cada polígono de un objeto, por lo que se puede obtener, a partir de ellos, los valores de profundidad de cada píxel. La forma de calcular estos valores en un algoritmo de 'scan-line', es mediante una interpolación lineal de píxels, la cual consiste en hallar un dato que está dentro de un intervalo de valores, si conocemos sus dos extremos.

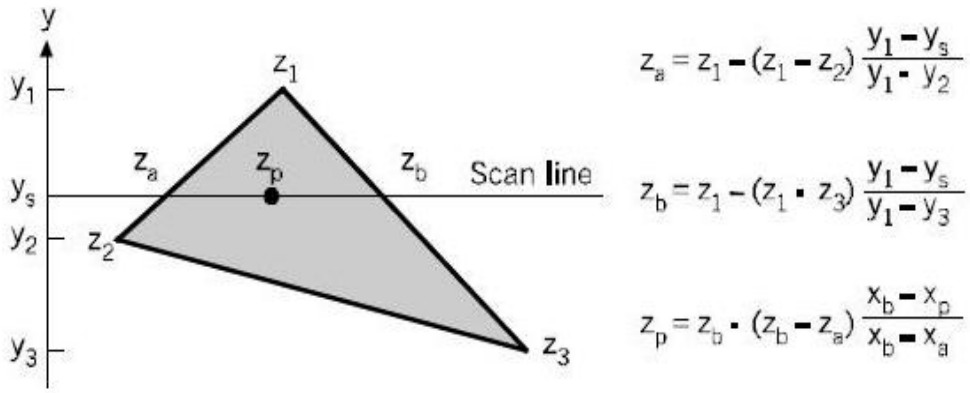

Figura 9. Ejemplo de scan-line de un triángulo, e interpolación de sus píxels

Como se puede ver en la figura 9, calculamos por interpolación lineal la componente Z del primero y del último píxel de la fila actual que vayamos a procesar (píxels a y b). Para ello, necesitamos los extremos de un intervalo de valores de Z en los que estén los valores del primero y el último píxel. Estas cotas las encontramos en los valores Z de los vértices 1,2 y 3, que están en las mismas rectas que los píxels a y b, por lo que lo calculamos en función de dichos vértices. Una vez hecho, calculamos la componente Z de cada Zp interpolando con los píxels a y b.

Debido al procesamiento de cada polígono, el z-buffer no requiere ninguna ordenación previa de los mismos. Además, su funcionamiento es muy simple, así como su implementación hardware en tarjetas gráficas. Sin embargo, el z-buffer tiene ciertas desventajas, como pérdida de precisión según aumente la distancia, o un coste elevado de memoria, dado que tenemos que construir una matriz con las dimensiones de la ventana de imagen. Por ejemplo, con una resolución normal de 640x480 píxels, puede requerir un z-buffer de hasta 1MB, dependiendo de la cantidad de bytes con la que representemos cada píxel.

Otro problema producido por el z-buffer es el *aliasing*, que consiste en mostrar discontinuidades en forma de bordes dentados en el contorno de los objetos. Actualmente los renderizadores modernos incluyen técnicas de *anti-aliasing*, aunque su explicación se escapa del objetivo de esta memoria. Por último, el coste de calcular el color de cada píxel es bastante elevado, por lo que conviene calcularlo cuando haya pasado con éxito el test de profundidad, para evitar cálculos innecesarios.

F) Algoritmo de sombreado: Una vez que hemos superado el test de profundidad, podemos calcular el color del píxel. Para ello, debemos tener en cuenta la iluminación del mismo, la cual depende de diversos factores. Entraremos con más detalle en este parte en el apartado posterior sobre la iluminación.

G) Aplicación de texturas: Este método consiste en aplicar imágenes gráficas, tales como BMP o JPG a las caras de cada objeto de la escena. De esta forma, el objeto queda “envuelto” por las texturas. Este método es imprescindible en los renderizadores modernos, ya que la aplicación de texturas permite simplificar en gran medida los cálculos de formas geométricas complejas, dando un resultado muy realista. Sin embargo, a poca distancia del objeto sí se puede observar que la figura no es tan real como aparenta, y que lo que vemos es una textura plana.

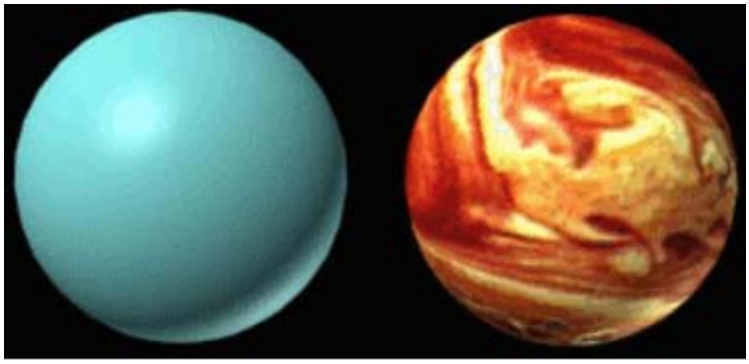

Figura 10. A la izquierda, esfera sin texturas. Ala derecha, la misma esfera envuelta por una textura

Las texturas pueden ser 1D, 2D ó 3D, siendo las 2D las más comunes, ya que las 1D son poco útiles, y las 3D requieren mucho cálculo. Sin embargo, normalmente no coincidirá la forma y tamaño del objeto con la textura, por lo que debemos ajustarla y corregirla de forma que encaje bien en el objeto sobre el que la vamos a aplicar.

Para cada vértice de un objeto, debemos calcular sus coordenadas de textura, que indican qué parte de la textura va a superponerse sobre él. Con estas coordenadas calculadas, el renderizador debe mezclar el color de ese vértice de la textura con el valor RGB calculado en el proceso de iluminación. La textura del resto de píxels que no son vértices, al igual que como veremos en la iluminación, se calcula por interpolación lineal (vista en el cálculo del z-buffer).

Cuando la imagen ha pasado por la tubería gráfica se guarda en el color buffer, el cual es una memoria de imagen. Desde aquí, la imagen pasará al monitor para ser visualizado en la pantalla. El color buffer, es uno de los buffers que conforman el Frame Buffer, donde se ejecutan los últimos cálculos sobre la imagen, y que enlaza con la entrada de datos de la imagen al monitor. Parte de los cálculos descritos anteriormente se hacen aquí, como el cálculo del z-buffer, u otros cálculos con buffers secundarios que no entran en el objetivo de esta memoria.

2.5 Dispositivo gráfico

Se encarga de tomar la imagen creada en la CPU y visualizarla. Para ello, sigue una secuencia de pasos:

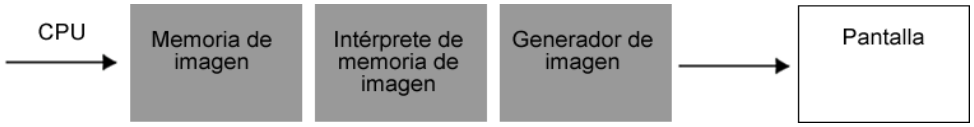

Figura 11. Proceso de visualización de una imagen en el dispositivo gráfico

La memoria de imagen es generalmente un frame buffer, pero en realidad la memoria puede ser de dos tipos: una memoria de listas, donde se guarda la información sobre las primitivas de dibujo de los objetos, así como datos acerca de su color y posición; o un frame buffer, donde lo que se guarda es una representación de la pantalla en puntos o píxels, con información del color que tendrá cada uno de ellos. La memoria de listas se usa en monitores de refresco vectorial, los cuales consiguen el color por penetración (explicado en el apartado de pantallas CRT), y su unidad básica de dibujo es una línea entera. El frame buffer es la memoria usada es los monitores de vídeo de rastreo, los monitores más extendidos en el mercado, donde la unidad de dibujo es el píxel. Para visualizar una escena entera, se van dibujando todos los píxels de una línea horizontal entera de la pantalla, hasta completar los píxels de todas las líneas horizontales que forman la pantalla.

La pantalla

El último paso una vez que la imagen ya ha sido guardada en el frame buffer, es que el dispositivo de salida visualice dicha imagen. Por ello, describiremos brevemente los dispositivos de salida actuales.

A) Tubo de rayos catódicos (CRT): Es una de las tecnologías más antiguas diseñadas para dispositivos de salida, pero aun hoy en día sigue siendo la más usada. El dispositivo está formado por un tubo de vidrio, en el que está hecho el vacío, con un cátodo en un extremo. Al calentarse el cátodo, lanza un haz de electrones hacia la pantalla, el otro extremo del tubo, el cual es más ancho y tiene una capa de fósforo que emite luz al incidir el haz de electrones sobre ella. Para controlar la intensidad y la posición del haz, se colocan en el cuello del tubo una serie de bobinas que gobiernan dicho haz.

Para que la imagen pueda verse, se emite un haz de electrones hacia cada píxel, con una frecuencia determinada, dado que la iluminación de un píxel en la capa de fósforo no tiene una duración ilimitada, y necesita que el cátodo refresque la iluminación cada cierto tiempo.

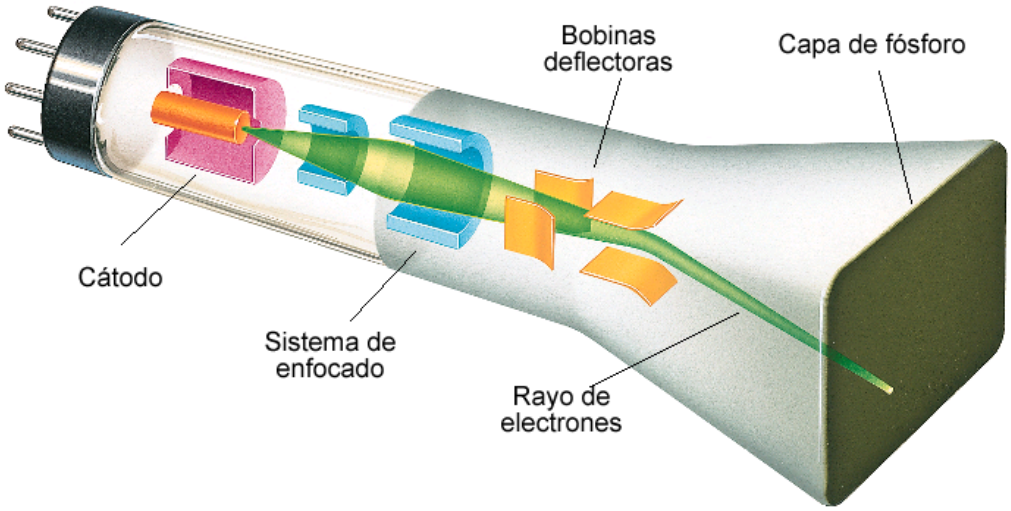

Figura 12. Esquema de un monitor CRT

Existen dos formas de conseguir el color de cada píxel, por penetración y por máscara. En el método por penetración, la capa de fósforo está formada en realidad por varias capas de fósforo de distintos colores. Dependiendo de la intensidad del haz de electrones, se iluminarán un cierto número de estas capas, consiguiendo el color final. En la actualidad, el método comúnmente usado es el método por máscara, el cual tiene píxels formados cada uno por tres puntos de fósforo: uno rojo, otro verde y otro azul. El haz de electrones está formado en realidad por tres haces, cada uno con un cañón de electrones distinto. El tubo contiene, además, una placa reja de metal, de manera que por cada píxel hay un hueco en la reja, para que cada haz triple de electrones pase solamente por el agujero que corresponda. De esta manera, aunque el píxel está formado por tres puntos de color, es tan pequeño que el ojo humano lo percibe como un punto con la suma de los tres colores.

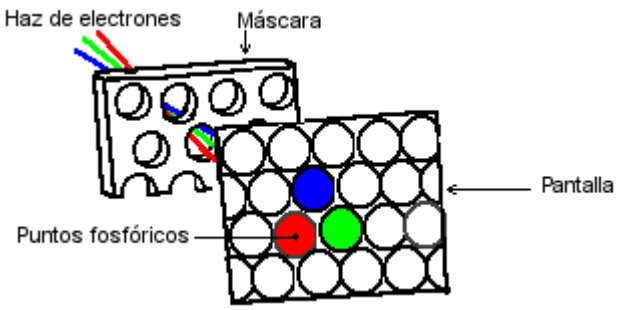

Figura 13. Paso de los haces de electrones hasta la pantalla

B) Cristal líquido: Las pantallas de cristal líquido, o LCD están formadas por dos capas polarizantes, y una capa con una solución de moléculas cristalinas en medio de las dos capas anteriores. Por la capa interior de cristal líquido puede pasar corriente eléctrica, de manera que al pasar la corriente, los cristales se alinean no dejando pasar la luz; al no haber corriente, los cristales dejan pasar la luz. Las pantallas LDC pueden ser monocromas o de color. Dentro de este último grupo, existen dos técnicas distintas de visualización de imágenes: la técnica de matriz activa y la de matriz pasiva. Las segundas son mucho más baratas de fabricar, pero las primeras reproducen imágenes de mucha más calidad. Un ejemplo de este tipo de implementación son las pantallas TFT (Thin Film Transistor).

C) Plasma: Estas pantallas se construyen de forma muy parecida a las pantallas LCD. La pantalla cuenta con una serie de electrodos dispuestos horizontalmente, que acotan cada uno de los píxeles. Para iluminar un píxel, se aplica tensión entre los dos electrodos que lo acoten. Para darle color a ese píxel, la pantalla cuenta con una capa de fósforo con puntos RGB por la que debe pasar la luz.

2.6 Proyecciones

La proyección es la representación con que mostramos en la pantalla cada objeto de una escena. Dependiendo del tipo de proyección, cada punto de un objeto tridimensional tendrá unas coordenadas bidimensionales u otras en el plano de vista. Las proyecciones las definimos a partir del marco de coordenadas de la cámara, y del volumen de vista. El marco de coordenadas de la cámara es ortogonal, y está definido por el sistema de coordenadas, el punto donde está ubicada la cámara, y la dirección hacia la que mira. El volumen de vista está definido por dos planos, el cercano (más cercano a la cámara) y el lejano, y el plano de vista, donde se proyecta la imagen, situado entre los dos planos anteriores, e identificado generalmente con el plano cercano. Los tres planos son, además, perpendiculares al vector de dirección del marco de la cámara.

Básicamente, existen dos tipos de proyecciones: paralelas y perspectivas. A continuación explicaremos con más detalle cada una de ellas.

2.6.1 **Proyecciones paralelas**

Estas proyecciones se caracterizan por tener un volumen de vista en forma de paralelepípedo. Su rasgo fundamental consiste en que la distancia de la cámara con respecto al plano de vista no afecta a la proyección, por lo que los objetos tienen el mismo tamaño independientemente de lo lejos o cerca que se encuentren con respecto a dicha cámara.

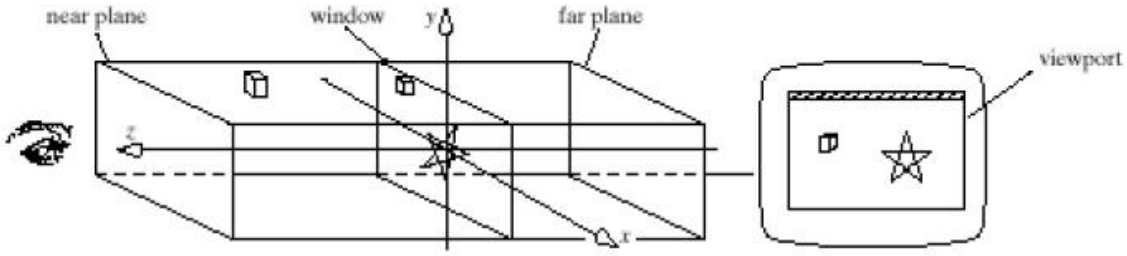

Figura 14. Volumen de vista en proyecciones paralelas

Para proyectar los puntos sobre el plano de vista, se trazan líneas paralelas entre sí que intersequen con el plano, definidas con una dirección dada por el vector de proyección. Ese punto de intersección será la nueva coordenada del punto proyectado. Las proyecciones paralelas nos muestran imágenes exactas sobre cada lado de un objeto; por ello, no son muy realistas. Sin embargo, son muy útiles a la hora de realizar diseños de arquitecturas.

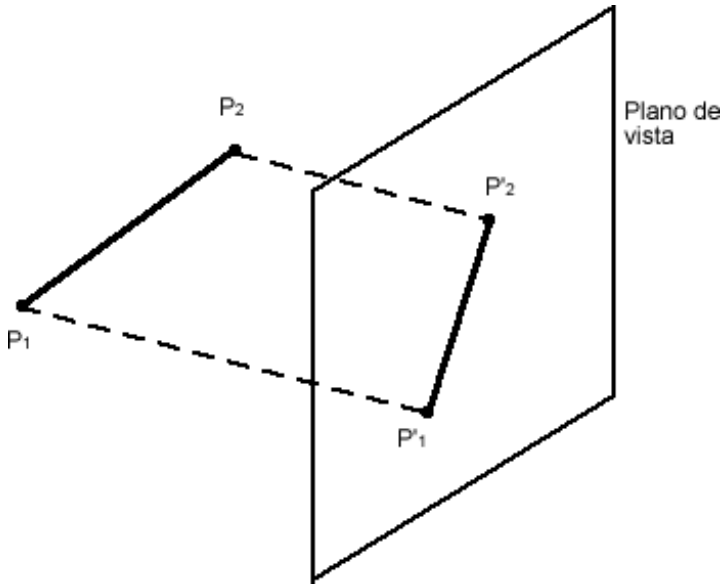

Figura 15: Proyección paralela de una recta

2.6.2 Proyecciones ortogonales

Son aquellas proyecciones paralelas en las que el vector de proyección es perpendicular al plano de vista. Este tipo de proyección representa cada cara de un objeto con total exactitud, motivo por el cual es usado en diseños arquitectónicos.

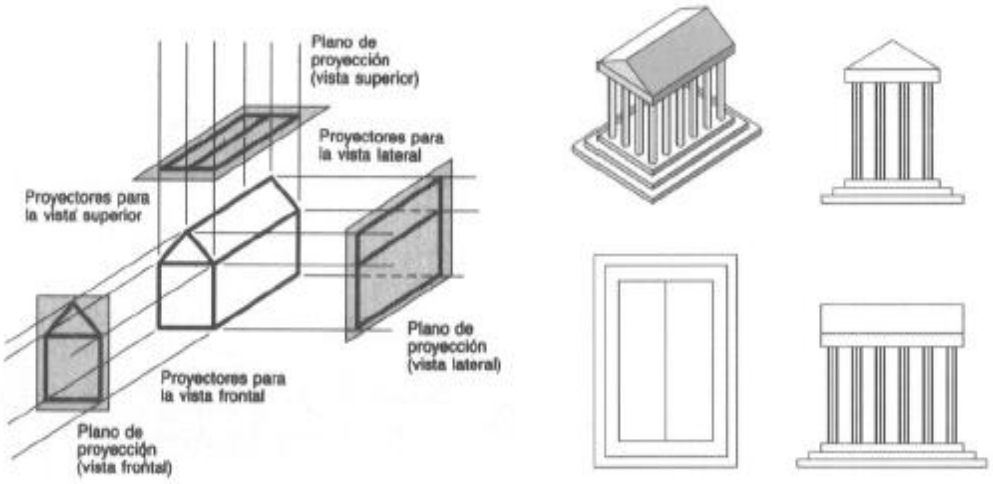

Figura 16. Tipos de vistas en una proyección ortogonal

Las vistas laterales y frontal del objeto se llaman elevaciones (lateral y frontal, respectivamente), mientras que la vista superior se llama vista de planta. Sin embargo, las proyecciones ortogonales no solo son capaces de mostrar una sola cara del objeto, sino que pueden mostrar varias. En estos casos, se habla de proyecciones ortogonales axonométricas.

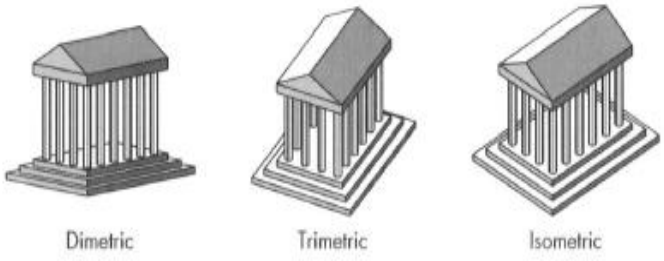

Figura 17. Proyecciones axonométricas más comunes

Las proyecciones axonométricas que se suelen usar con más frecuencia son las dimétricas (dos de los ejes del objeto forman el mismo ángulo con la normal del plano de vista), trimétricas (cada eje forma un ángulo distinto con la normal al plano de vista) e isométricas (los tres ejes del objeto forman el mismo ángulo con la normal del plano de vista).

2.6.3 Proyecciones oblicuas

Se caracterizan por tener un vector de proyección no perpendicular al plano de vista. Este tipo de proyecciones muestra una de las caras con su proyección exacta, pero da una forma en 3D, mostrando otras dos caras del objeto deformadas. Para transformar las coordenadas 3D de un punto del objeto en las coordenadas 2D (x_p, y_p) proyectadas sobre el plano de vista, hace falta conocer dos ángulos:

α , es el ángulo formado por el vector de proyección y el plano de vista.
ϕ, es el ángulo formado por la horizontal del plano de proyección con la recta L. Dicha recta está formada por las coordenadas (x_p, y_p) de la proyección oblicua, y por las coordenadas (x, y)del punto, que equivalen a las coordenadas de la proyección ortogonal.

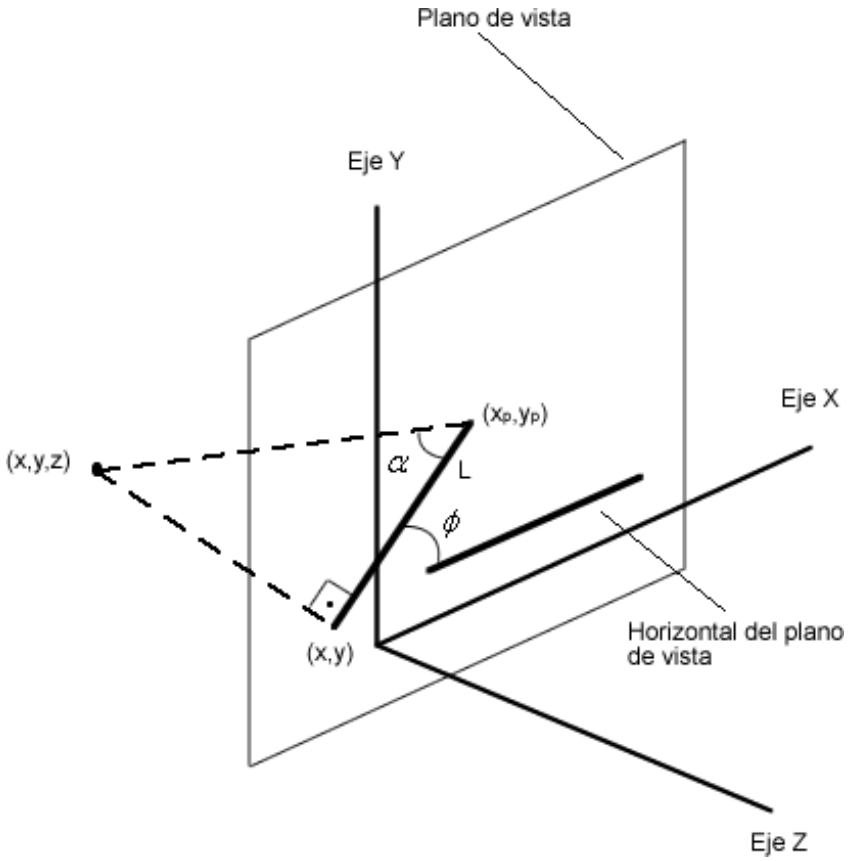

Figura 18. Proyección oblicua de un punto

Con esto, podemos definir las coordenadas de la proyección oblicua como:

$$x_p = x - z(L_1 \cdot \cos\phi)$$
$$y_p = y + z(L_1 \cdot \cos\phi)$$

donde $L_1 = \tan(\alpha)^{-1}$

De esta definición podemos deducir que la proyección ortogonal es un caso particular de proyección oblicua, donde $L_1 = 0$.

Los dos tipos más importantes de proyección oblicua son la proyección caballera y la proyección de gabinete. La primera se produce cuando $\tan(\alpha)$ vale 1. La proyección caballera no altera la longitud de las líneas perpendiculares al plano de vista. En el caso de la proyección de gabinete, $\tan(\alpha)$ vale 2, y produce una mayor sensación de realismo con respecto a la proyección caballera, ya que sí altera la longitud de las rectas perpendiculares al plano de vista; exactamente las reduce a la mitad.

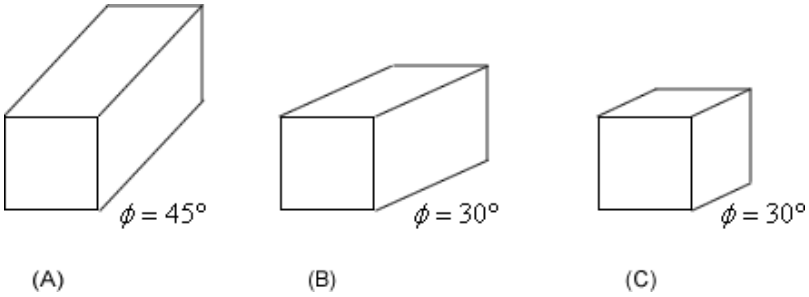

Figura 19. A y B: Proyecciones caballeras con distinto ángulo con respecto a la horizontal. C: Proyección de gabinete

2.6.4 Proyecciones perspectivas

En este tipo de proyección los objetos no conservan sus dimensiones originales, sino que se transforman con las líneas que convergen en el punto de proyección, o punto de fuga. Dichas líneas de convergencia unen el punto que queremos proyectar con el punto de fuga. Por ello, para calcular la posición de un punto en el plano de vista con proyección perspectiva, hay que calcular la intersección de la línea de convergencia que pasa por ese punto con el plano de vista.

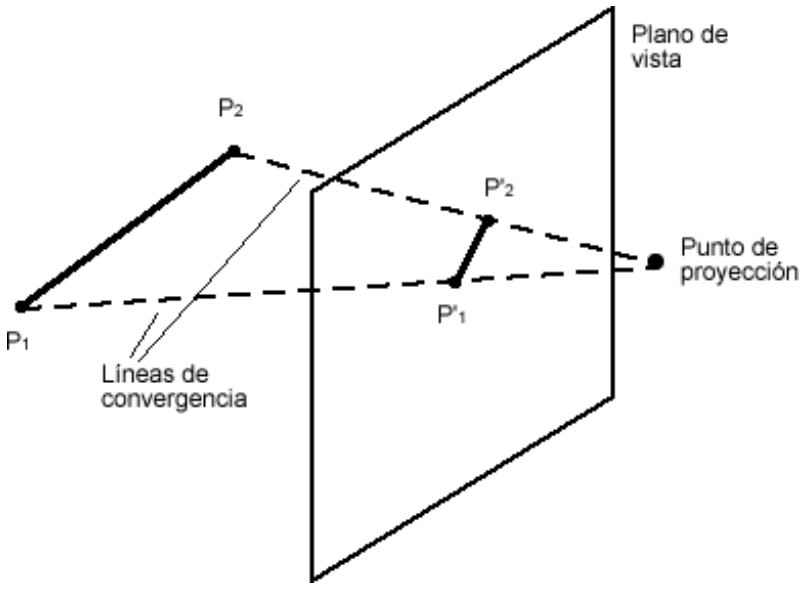

Figura 20. Proyección perspectiva de una recta

Una característica de este tipo de proyección es el hecho de no mostrar como paralelas conjuntos de líneas paralelas, siempre y cuando no sean paralelas al plano de vista. La proyección de este conjunto de líneas será en forma de líneas convergentes al punto de proyección. En caso de que el conjunto de líneas paralelas sea paralelo al plano de vista, la proyección sí mostrará las líneas paralelas entre sí.

Cada conjunto de líneas paralelas tiene su propio punto de fuga. Si un conjunto de líneas paralelas son paralelas a uno de los ejes de coordenadas, entonces el punto de fuga es principal. Las proyecciones en perspectiva las clasificamos en función del número de puntos de fuga principales que contengan (perspectivas de un punto, de dos puntos o de tres puntos), que es el mismo número que el de ejes cartesianos que atraviesan el plano de vista.

La principal ventaja de esta proyección es que da lugar a imágenes muy realistas, aunque no conservan las proporciones reales. Esto quiere decir que el mismo objeto aparecerá más grande en la imagen cuanto más cerca esté de la cámara, con lo que su funcionamiento se asemeja al de una cámara real.

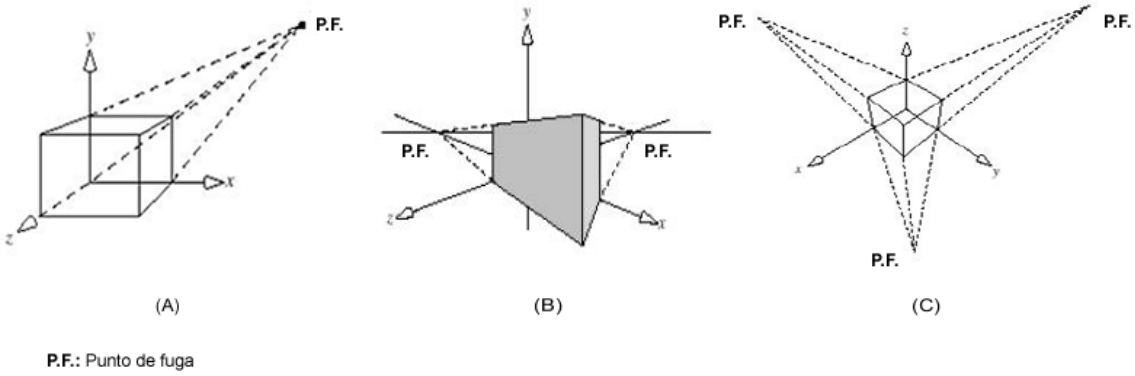

Figura 21. Tipos de proyección perspectiva. A: de un punto. B: de dos puntos. C: de tres puntos

El volumen de vista es aquí distinto al de las proyecciones paralelas; está formado por una pirámide con su parte superior cortada por un plano paralelo a la base, cuyo vértice sería el punto de fuga.

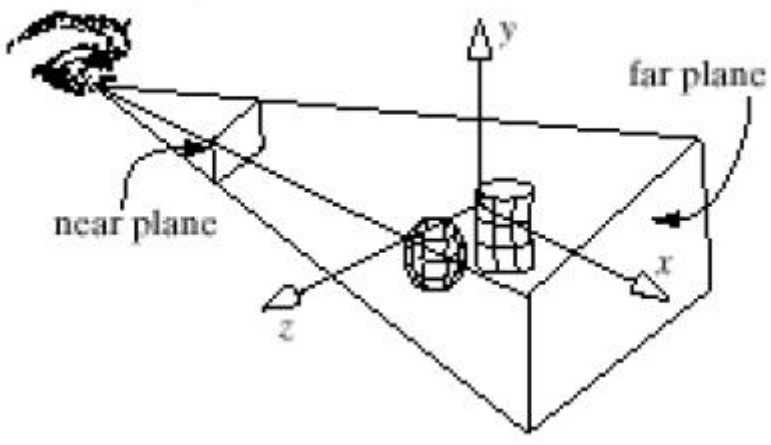

Figura 22. Volumen de vista en proyecciones perspectivas

2.7 Iluminación

2.7.1 Introducción

Cualquier renderizador moderno debe incluir procesamiento de la luz para dar imágenes de cierta calidad, ya que la iluminación dota a la imagen de realismo. Por ello, a partir de ahora, definiremos el color de un punto de la superficie de un objeto según varios factores, como la cantidad de luz que incida sobre él, el material de que está hecho el objeto al que pertenece el punto, el color de la luz, el ángulo con el que incide la luz en el punto, o la orientación de la cámara.

La luz proviene de dos fuentes distintas: por un lado, las fuentes de luz creada (método directo), y por otro, la luz reflejada por los objetos del entorno (método indirecto). Precisamente la composición de este segundo tipo de luces son las que forman y modelan la luz ambiente de la imagen. La cual es constante en toda la escena, independientemente de los focos de luz.

La luz directa proviene de las fuentes de luz, las cuales pueden ser de dos tipos: de puntos de luz, cuando la luz emana de un solo punto original, y de áreas de luz (distribuidas), que son áreas finitas que emiten la misma cantidad de luz por toda su superficie, y que son más difíciles de modelar. Dividiremos la luz directa en componentes:

- Componente difusa: es la luz absorbida por el objeto, y que se refleja en todas direcciones con la misma intensidad.
- Componente especular: es la componente de la luz que al incidir sobre un punto, se refleja con dirección especular a la inicial. Como veremos posteriormente, habrá que hacer una composición de vectores para calcular esta dirección especular.
- Componente ambiente: componente que forma la luz ambiente global.

Estas componentes definen, cada una, un valor RGBA para el punto, con la información sobre el color y la intensidad. Sin embargo, la distancia o el cono de emisión influirán como atenuantes de la luz.

Figura 23. Ejemplo de composición de luces

Dependiendo del material con que esté hecho un objeto, éste tendrá unos coeficientes de reflexión u otros. Estos coeficientes son valores RGBA que nos muestran la cantidad de luz que será reflejada. Existe un coeficiente para cada componente de la luz: el difuso y el ambiente determinan el color del objeto; el especular determina su brillo.

2.7.2 Reflexión de la luz

Para calcular la intensidad y el color de la luz reflejada por un punto P al que le llega luz, debemos tener en cuenta tres vectores: el vector $\vec{m}$, normal a la superficie del objeto donde está P, y que pase por P, el vector $\vec{S}$ que va de P a la fuente de luz, y el vector **V** que va de P a la cámara. Para comprobar que la cámara pueda ver la superficie, el ángulo que forman $\vec{m}$ y $\vec{v}$ debe ser menor de 90º, si no, la cámara no tendrá visión sobre dicha superficie.

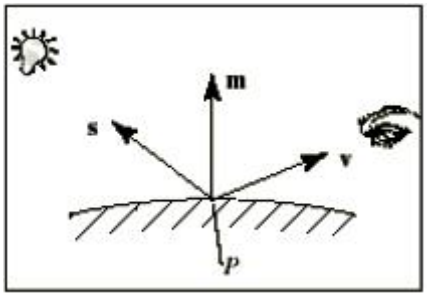

Figura 24. Vectores que influyen al calcular la intensidad de luz

El modelo más extendido a la hora de implementar la reflexión de la luz sobre cada punto de la superficie de los objetos es el modelo de reflexión de Phong, dado que es el que produce los mejores resultados en cuanto a calidad de imagen. Dicho modelos se basa en la interacción de la luz con la superficie del objeto, calculando la interpolación lineal de las tres componentes de la luz antes mencionadas. A continuación, explicaremos cada una de las componentes de la luz, así como la forma que tenemos de calcular su intensidad de reflexión, cuando cada una de estas componentes incide en la superficie de un objeto. Además, veremos la diferencia que existe en la reflexión de la luz dependiendo del material del objeto.

Componente difusa

Esta componente indica la intensidad de reflexión difusa de un punto, al incidir la luz sobre él. Esta reflexión se produce uniformemente en todas direcciones, por ello, nos es irrelevante aquí la posición de la cámara.

Si suponemos como única luz la luz ambiente, la intensidad de la luz difusa se reduce a la ecuación:

$$I = R_d \cdot I_a$$

donde Rd es el coeficiente de reflexión de la componente difusa del material, el cual varía entre 0 y 1, y Ia es la intensidad de luz ambiente. Sin embargo, generalmente no va a ser así, y nos encontraremos con focos de luz en nuestras imágenes, además de la luz ambiente. Para calcular en estos casos la reflexión de la componente difusa, usamos la **Ley de Lambert**, la cual dice que la intensidad de la reflexión difusa depende del ángulo de iluminación, y cuanto mayor es ese ángulo, menor cantidad de luz incidirá sobre la superficie. Dicho ángulo de iluminación, también llamado ángulo de incidencia, es el ángulo formado por los dos vectores que sí entran en juego en el cálculo de esta reflexión: los vectores $\vec{m}$ y $\vec{S}$.

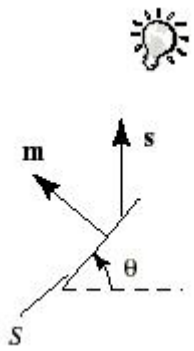

Figura 25. Ángulo formado por los vectores m y s

Según Lambert, la cantidad de luz reflejada es proporcional al coseno de este ángulo de incidencia, el cual se calcula con la ecuación:

$$\cos(\theta) = \frac{\vec{s} \cdot \vec{m}}{|\vec{s}| \cdot |\vec{m}|}$$

pero en realidad, para simplificar cálculos, en los renderizadores modernos se usa simplemente la ecuación:

$$\cos(\theta) = \vec{s} \cdot \vec{m}$$

Teniendo esto en cuenta, escribimos la fórmula del cálculo de la componente difusa de la siguiente manera:

$$I = (\vec{s} \cdot \vec{m}) \cdot I_s \cdot R_d$$

donde

I es la intensidad de reflexión de la componente difusa
I_s es la intensidad de la componente difusa de la luz
R_d es el coeficiente de reflexión difuso del material del objeto en el punto P.

Esta fórmula sirve en el caso de tener una luz monocromática; sin embargo, en la realidad podemos tener luces de diferentes colores. La ley de Lambert también contempla ese caso, por lo que I_s y R_d no son valores en coma flotante, sino que son vectores con tres componentes RGB, para distinguir el tipo de color:

$I_s = (I_{sR}, I_{sG}, I_{sB})$, $R_d = (R_{dR}, R_{dG}, R_{dB})$

Componente especular

Como habíamos avanzado antes, la componente especular es la que se refleja en el objeto con una dirección dependiente del ángulo que forme con el rayo de luz incidente sobre dicho objeto. Esta componente es la culpable de la sensación de brillo en el objeto, calculado para cada punto en función del exponente especular del material, que es el parámetro que determina el aumento o disminución de intensidad de luz reflejada. Además, esta componente depende de más factores para poder ser visto, como la posición de la cámara respecto a la fuente de luz, y con respecto al ángulo de reflexión. Este ángulo es con el que la luz incidente es reflejada, y es bastante similar al ángulo de incidencia. Lo definiremos como el ángulo φ formado por los vectores $\vec{r}$ y $\vec{v}$, donde $\vec{r}$ es el vector de dirección de la reflexión especular, y $\vec{v}$ es el vector ya definido, de dirección a la cámara. Cuando φ es 0, y los dos vectores coinciden, la reflexión especular es máxima; sin embargo, esta disminuye a medida que φ aumenta. Además, cuando la reflexión especular es máxima, la superficie es más brillante, pero se ve desde un número menor de ángulos. En el caso contrario, con un ángulo grande, la superficie será menos brillante, pero visible desde más ángulos distintos.

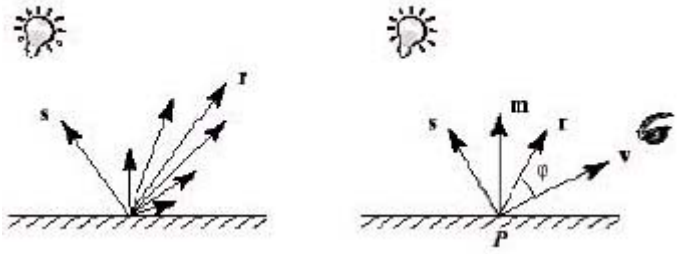

Figura 26. Vectores que influyen en el cálculo de la reflexión especular

Necesitamos conocer el valor de φ para cuantificar la reflexión especular, y para ello nos servimos de los vectores $\vec{r}$ y $\vec{v}$. Sin embargo, tenemos que calcular la dirección de $\vec{r}$, y esto lo hacemos mediante una composición de vectores:

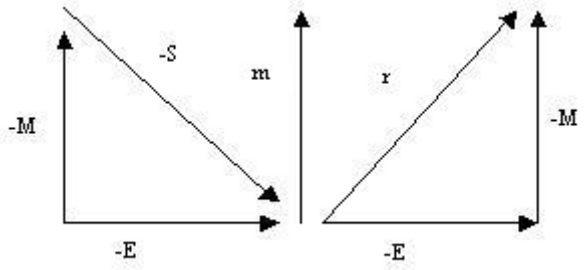

Figura 27. Composición de vectores para calcular r

de la que resulta la solución para $\vec{r}$:

$$\vec{r} = \vec{S} - 2\left(-\vec{S} + \left(\vec{S} \cdot \vec{m}\right) \cdot \vec{m}\right)$$

Antes habíamos establecido la proporción entre el ángulo φ y la intensidad de la reflexión especular, además de la influencia del exponente especular del material del objeto en dicha intensidad. Según aumente el exponente, el cual toma valores entre 1 y 200, aumentará el brillo en ese punto en concreto del objeto. Phong halló la relación entre todas estas proporciones, calculando el $\cos^{f}(\varphi)$, donde f es el exponente especular. Dicha relación se refleja en la siguiente gráfica **[9]**:

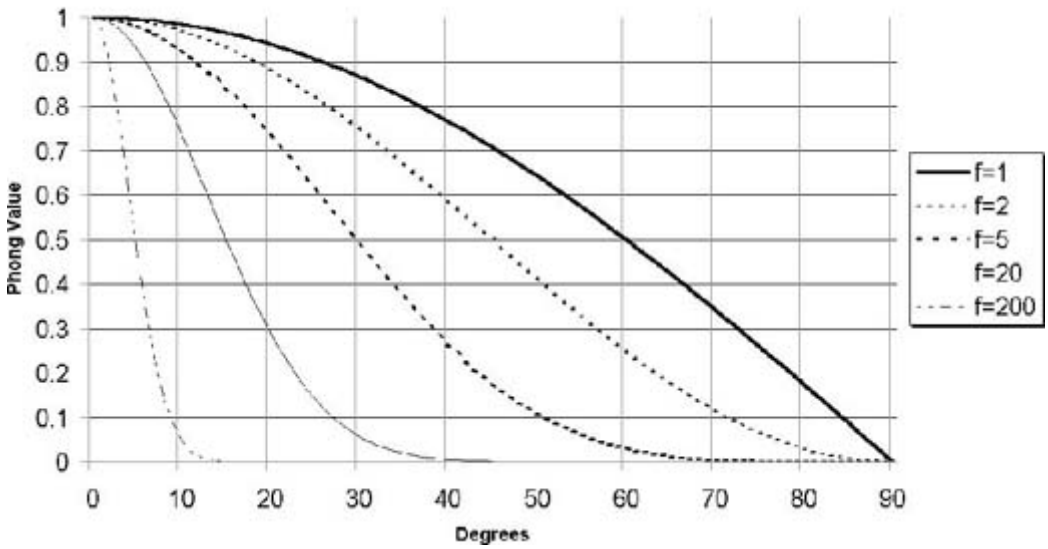

Figura 28. Variación del modelo de Phong al variar el grado especular

En ella, podemos observar como para valores altos de f, el valor del modelo de Phong ($\cos^{f}(\varphi)$) llega a 0 más rápidamente que para valores bajos, con menos intensidad especular. Además, cuanto más brillo tiene la imagen (cuando φ tiene valores más bajos), más rápido se alcanza el 0 en el modelo de Phong. Con todo ya definido, podemos establecer la fórmula de cálculo de intensidad especular como:

$$I = I_{sp} \cdot R_{sp} \cdot \max\left(0, \cos^{f}(\varphi)\right)$$

donde:

I_{sp} es la intensidad de la componente especular de la luz incidente

R_{sp} es el coeficiente de reflexión especular del objeto. Generalmente, este coeficiente tiene el mismo valor que el coeficiente de reflexión difusa.

Sin embargo, al igual que la componente difusa, se puede simplificar cálculos, y dado que:

$$\cos(\varphi) = \frac{\vec{r} \cdot \vec{v}}{\left|\vec{r} \cdot \vec{v}\right|}$$

podemos escribir la fórmula del cálculo de la intensidad de reflexión especular como:

$$I = I_{sp} \cdot R_{sp} \cdot \left(\vec{r} \cdot \vec{v}\right)^f$$

Todavía podemos hacer una última simplificación. El vector $\vec{r}$ requiere demasiados cálculos complejos, como hemos visto anteriormente. Y aunque no hay otra forma de calcularlo, sí que existe un método fiable para calcular una buena aproximación a ese vector. En general, se cumple la propiedad:

$$\vec{r} \cdot \vec{v} \cong \vec{m} \cdot \vec{H}$$

siendo $\vec{H}$ un vector equidistante entre $\vec{s}$ y $\vec{v}$:

$$\vec{H} = \frac{\vec{s} + \vec{v}}{\left|\vec{s} + \vec{v}\right|}$$

Esto se produce cuando suponemos que la fuente de luz y la cámara están ubicados en un punto infinito, dado que entonces, sus respectivos vectores que pasan por P, $\vec{s}$ y $\vec{v}$, sería constantes a lo largo de toda la superficie. Así que $\vec{H}$ es la normal a una superficie que hipotéticamente tendría una orientación de dirección que estaría justo a mitad entre $\vec{s}$ y $\vec{v}$.

Después de todas las simplificaciones, la intensidad final se calcula como:

$$I = I_{sp} \cdot R_{sp} \cdot \left(\vec{m} \cdot \vec{H}\right)^f$$

Componente ambiente

Cada fuente de luz tiene una componente ambiente que se refleja en cada píxel de la pantalla. Componiendo el reflejo de cada una de las componentes ambiente de los focos de luz de la escena sobre cada uno de los píxeles de los objetos sobre los que dichas componentes se reflejan, se forma la luz ambiente global $\left(I_a\right)$. Un ejemplo de este tipo de luz ambiente es la que se ve en un día nublado o lluvioso. En principio, la luz ambiente global no produce sombras; sin embargo, las aplicaciones gráficas pueden llegar a producir algunas sombras suaves si dicha luz ambiente proviene desde arriba, cuando tiene más fuerza. Al contrario de lo que se podría pensar, no hay que calcular esta luz ambiente global, es el propio programador el que la fija.

Lo que sí debemos calcular es la intensidad de reflexión de la luz ambiente en el punto del objeto que procesemos en ese momento. Para ello, simplemente tenemos en cuenta el coeficiente de reflexión ambiente del material del objeto:

$$I = K_a \cdot I_a$$

Los factores de atenuación

Existen dos principales factores de atenuación, que afectan a la intensidad con que la luz procedente de algún foco llega a incidir en los puntos de los objetos. Dichos factores son:

A) La distancia: Para que exista atenuación por distancia, el foco de luz debe ser local, esto es, que se conozca su ubicación. De esta forma, podrá calcularse el vector $\vec{S}$ con respecto al punto sobre el que vayamos a calcular la reflexión, y también podrá calcularse la atenuación producida por la distancia entre ambos puntos. Esta atenuación es:

$$AtDis\tan cia = \frac{1}{\left(k_c + k_l \cdot D + k_q \cdot D^2\right)}$$

donde:

k_c, k_l, k_q = son coeficientes de atenuación de la fuente de luz

D = Distancia entre el punto P y el foco de luz

Sin embargo, la luz puede considerar se remota, es decir, que se conoce su vector de dirección $\vec{S}$, pero no la ubicación del foco de luz. Por ello, se supone $\vec{S}$ constante para todos los puntos de la escena, por lo que no puede calcularse la distancia entre los puntos y el foco de luz, con lo que no hay atenuación por distancia.

B) El cono de emisión: Se supone que el foco de luz emite por un cono, de manera que la intensidad de la luz es máxima en la dirección del eje del cono, pero en las direcciones que parten del foco de luz pero se van alejando de la dirección del eje, tienen menos intensidad. Los puntos de la escena que quedan fuera del campo de acción del cono de emisión, no tienen atenuación por este factor.

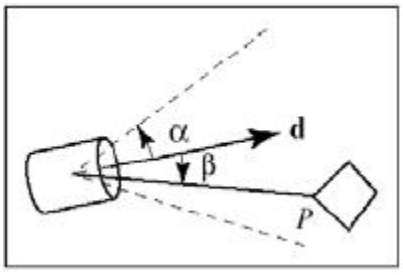

Figura 29. Variación de intensidad en el cono de emisión

Definimos la atenuación por cono de emisión como:

$$AtConoEmisión = \cos^{\varepsilon}(\beta)$$

donde:

ε = es un exponente del foco de luz, que nos indica el grado de intensidad de L.

También podemos tener una fuente de luz omnidireccional, que es la que la luz tiene intensidad constante en todas direcciones, por lo que no hay atenuación por cono de emisión.

Reflexión final

Una vez que han quedado explicadas las reflexiones de cada una de las componentes de la luz sobre un punto de la superficie de un objeto, podemos formular la ecuación total del modelo de reflexión de Phong. Dicha ecuación calcula la reflexión total de n focos de luz L_k sobre un punto P de la superficie de un objeto con coeficientes de reflexión ambiente (R_a), difusa (R_d) y especular (R_e):

$$I = Em + I_a \cdot R_a + \sum_{k=1}^{n} At(P, L_k) \cdot \left(I_{ka} \cdot R_a + I_{kd} \cdot R_d \cdot (\vec{s} \cdot \vec{m}) + I_{ke} \cdot R_e \cdot \left(\vec{m} \cdot \vec{H}\right)^f \right)$$

donde:

Em = luz emitida por ese objeto (0 si no es un foco de luz, como una lámpara, por ejemplo)

I_a = luz ambiente global

$At(P, L_k)$ = atenuación producida en P para la luz L_k

$I_{ka} \cdot R_a$ = reflexión ambiente de la luz L_k sobre el punto P

$I_{kd} \cdot R_d \cdot (\vec{s} \cdot \vec{m})$ = reflexión difusa de la luz L_k sobre el punto P

$I_{ke} \cdot R_e \cdot \left(\vec{m} \cdot \vec{H}\right)^f$ = reflexión especular de a luz L_k sobre el punto P

2.8 Técnicas de sombreado

Las técnicas de sombreado son aquellas que aplican las fórmulas de reflexión de la luz a todos los polígonos de la escena, y a cada punto de cada polígono. Dado que el estándar en modelos de reflexión es el modelo de Phong antes visto, se usará este modelo para cada una de las técnicas que veamos. Las distintas técnicas tratarán, en la medida de lo posible, realizar los menores cálculos para renderizar una imagen con cierta calidad.

Básicamente, hay dos técnicas de sombreado (o *shading*): *flat shading* (sombreado por planos) y *smooth shading* (sombreado suave). La técnica de flat shading consiste en aplicar el modelo de reflexión a un punto de cada polígono, y una vez obtenido el resultado, aplicarlo a todos los puntos de dicho algoritmo. Esta técnica exige muy pocos cálculos, pero será útil en el caso de que no necesitemos mostrar superficies suaves. Esto es debido a que al tener todos los puntos de un polígono la misma intensidad, se

exageran demasiado las diferencias de intensidad entre polígonos adyacentes, resultando una imagen poco realista. Además, aunque es capaz de reproducir la reflexión especular, el resultado no es muy bueno. Con esto, podemos concluir que *flat shading* no es una buena técnica de sombreado.

Figura 30. Sombreado usando flat shading

2.8.1 Smooth shading

Esta técnica de sombreado se conoce también como técnica incremental, debido a que el cálculo de la intensidad de un punto se hace en base a un incremento de la intensidad de un punto anterior a él. Es una técnica mucho más precisa que flat shading, aunque conllevan bastantes más cálculos. La principal ventaja de smooth shading es que resuelve el problema de los saltos de intensidad entre polígonos. A continuación explicaremos los dos algoritmos más importantes de este tipo de técnica.

2.8.2 Gouraud shading

Este algoritmo[10] calcula el vector normal a la superficie por cada vértice, calculando el vector normal promedio de todos los vectores normales a los polígonos que comparten ese vértice. Con estos vectores, calculamos la intensidad de reflexión de cada vértice mediante el modelo de reflexión de Phong. Con estas intensidades, hacemos un algoritmo scan-line de los píxeles de cada polígono, como hacemos para calcular la profundidad de cada píxel, por ejemplo, y para cada píxel, hacemos una interpolación lineal de intensidades que nos den como resultado la intensidad de dicho píxel.

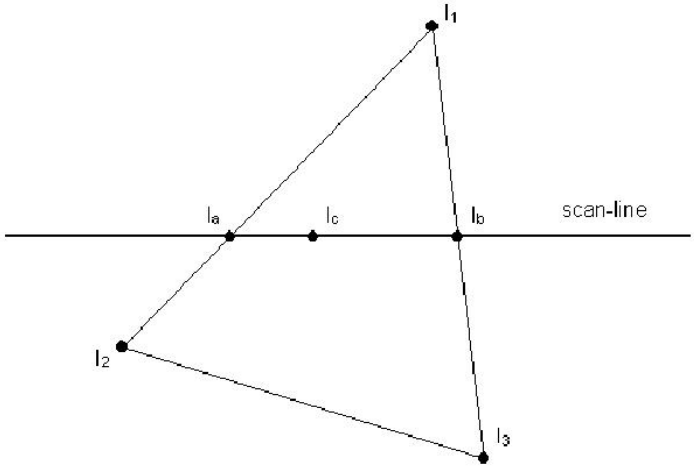

Figura 31. Cálculo de intensidades de reflexión por interpolación lineal

Como se ve en la figura 31, para cada línea de píxeles, calculamos la intensidad del primer y último píxel, I_a y I_b por interpolación lineal de los vértices del polígono, resultando:

$$I_a = I_1 + (I_2 - I_1) \cdot \frac{y_a - y_1}{y_2 - y_1}$$

$$I_b = I_1 + (I_3 - I_1) \cdot \frac{y_b - y_1}{y_3 - y_1}$$

Una vez tenemos estos valores, calculamos la intensidad del píxel actual por interpolación lineal de las dos intensidades anteriores:

$$I_c = I_a + (I_b - I_a) \cdot \frac{x_c - x_a}{x_b - x_a}$$

Sin embargo, existen técnicas para hacer los cálculos mucho más eficientes. Dado que los algoritmos *Smooth shading* son incrementales, podemos calcular la intensidad de un píxel en base a le anterior. Para cada línea del algoritmo de escaneado para calcular la intensidad de los píxeles, podemos calcular el incremento de intensidad como:

$$\Delta I = \frac{\Delta x \cdot (I_b - I_a)}{x_b - x_a}$$

y por lo tanto, podemos calcular cada píxel en base a este incremento:

$$I_c = I_{c-1} + \Delta I$$

El principal problema de este algoritmo es que no representa bien la reflexión especular, ya que la intensidad de reflexión depende más de los vértices que de la superficie del polígono. Esto es debido a que la interpolación de los píxeles de la superficie del polígono la realizamos con las intensidades ya calculadas de los vértices, por lo que si hay una gran intensidad especular sobre la superficie del polígono, y no sobre los vértices, la reflexión especular no será intensa.

2.8.3 Phong shading

El algoritmo de Phong es muy similar al de Gouraud, pero plantea una solución a la mala renderización de la reflexión especular que se calcula con el anterior algoritmo. En Phong, se calculan las normales de cada vértice, pero en vez de interpolar las intensidades, se interpolan dichas normales. Una vez que contamos con las normales a la superficie por cada píxel, calculamos la intensidad de reflexión de cada uno con el modelo de reflexión de Phong. Este método calcula la intensidad especular, en cualquier caso, de manera muy eficiente, a costa de hacer muchos más cálculos que en el algoritmo de Gouraud.

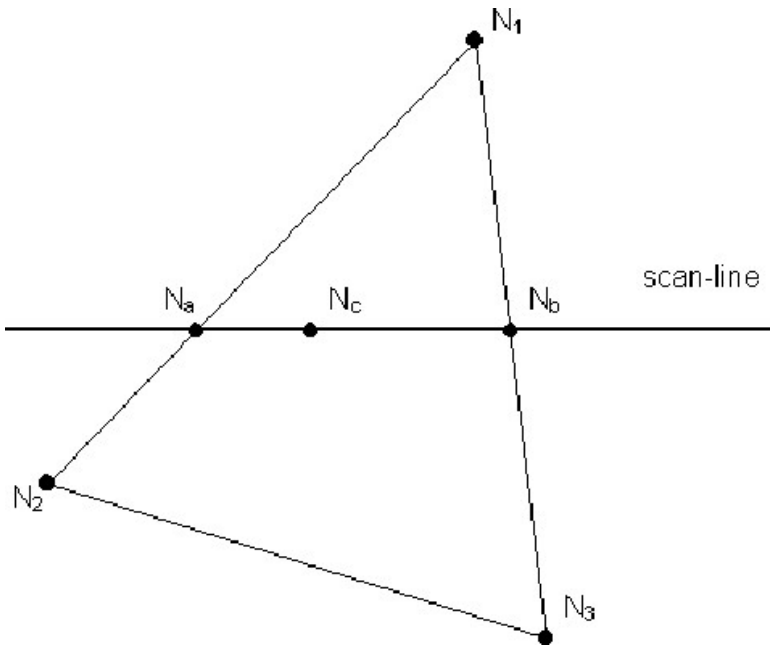

Figura 32. Cálculo de las normales por interpolación lineal

$$N_a = N_1 + (N_2 - N_1) \cdot \frac{y_a - y_1}{y_2 - y_1}$$

$$N_b = N_1 + (N_3 - N_1) \cdot \frac{y_b - y_1}{y_3 - y_1}$$

$$N_c = N_a + (N_b - N_a) \cdot \frac{x_c - x_a}{x_b - x_a}$$

Igual que en el anterior algoritmo, al ser incremental, podemos calcular las normales a partir del anterior píxel y del incremento de normales:

$$N_c = N_{c-1} + \Delta N$$

Una vez vistos los dos algoritmos, podemos concluir que el algoritmo de Gouraud nos es útil en escenas en las que hay una reflexión de la componente difusa principalmente. Para buenos resultados en reflexión especular, es más conveniente usar el algoritmo de Phong, ya que es menos dependiente de la intensidad especular de los vértices del polígono, y se basa más en el propio píxel. Además, la elección de uno u otro algoritmo dependerá también de la importancia que le demos al coste en cuanto a operaciones realizadas, dado que el algoritmo de Phong es mucho más costoso que el de Gouraud.

2.9 **Tarjetas gráficas**

Como inciso especial, al margen del hardware reconfigurable que veremos a continuación, vamos a ver de manera resumida el funcionamiento de una tarjeta gráfica [11], para comprender cómo trabajan estas y poder comparar con lo que haremos nosotros con la arquitectura Morphosys.

2.9.1 Introducción

Las tarjetas gráficas han experimentado un gran crecimiento en los últimos años. Hasta la aparición de las tarjetas SVGA no se le daba demasiada importancia, pero desde entonces han evolucionado de manera muy rápida, pasando de tarjetas gráficas en 2D a tarjetas con aceleración por hardware en 3D con mucha capacidad de procesamiento. Incluyen chips especializados para realizar el proceso de la información que reciben, de manera que el procesador principal no tenga que intervenir en estas operaciones, liberando a este de mucho trabajo, con lo que conseguimos imágenes de mejor calidad además de no colapsar el ordenador y sus buses.

Podríamos considerar en la actualidad la tarjeta gráfica como una pequeña computadora independiente, pues consta de su propio circuito de reloj para controlar las operaciones que ejecuta, su propia BIOS, que puede ser actualizada por software, consta también del "chipset gráfico", y de su propio esquema de memoria, por último incluye la RAMDAC, una circuitería especifica para convertir la señal de forma digital a analógica. Veamos los componentes más importantes:

2.9.2 Procesador (chipset) gráfico

El elemento más importante, encargado de realizar y controlar todas las operaciones gráficas, las entradas, las salidas y accesos a memoria, por lo que se ve claramente que su velocidad será muy importante para un buen rendimiento de la tarjeta, aunque no solo dependerá de él sino también del resto de los componentes.

Se puede distinguir dentro de los chipset, entre los de tipo 2D y los de tipo 3D, aunque en la actualidad tanto para juegos, como los nuevos programas de CAD y tratamiento grafico son las tarjetas 3D las que tienen mayor importancia y sobre las que se trabaja y mejoran constantemente.

2.9.3 La RAMDAC

Es la circuitería que tiene como función mandar los datos almacenados en la memoria después de su procesamiento al monitor, de manera que convierte la señal digital de la tarjeta gráfica en una señal analógica que es lo que puede entender el monitor. De la velocidad a la que transmita (medida en MHz) dependerá tanto la calidad de la imagen como la frecuencia de refresco. Esta frecuencia medida y expresada en MHz significa las operaciones que puede ejecutar la RAMDAC por segundo.

Cuando una nueva imagen ha sido procesada por la tarjeta gráfica, la RAMDAC tiene que transformarla y enviarla al monitor, incluso cuando la imagen no cambia, ésta se tiene que enviar igualmente cada cierto tiempo a la pantalla, pues necesita ser refrescada.

Podemos ver de la siguiente manera, cual debe ser la frecuencia mínima de la RAMDAC según la resolución de la imagen que queramos mostrar. Para hacer este calculo necesitamos saber la resolución de las imágenes que queremos mostrar, la tasa de refresco y el factor "1.32", que es el tiempo que los cañones de rayos catódicos están

refrescando fuera del área visible de la pantalla del monitor. Pondremos un ejemplo: si tenemos una resolución de pantalla de 1600 x 1200 con una tasa de refresco de 85Hz, tenemos que si multiplicamos todo 1600 x 1200 x 85 x 1.32 = 215,4 MHz, por lo que la RAMDAC tendrá que trabajar como mínimo con esa frecuencia para procesar bien esa resolución a esa tasa de refresco. En las tarjetas de gama media-alta, en la actualidad pueden llegar a tener una velocidad de 400 MHz de la RAMDAC.

2.9.4 Memoria de video

En un principio, la aparición de tarjetas 3D hace que éstas requieran mucha más memoria que las tarjetas en 2D, pues su arquitectura es muy diferente. Por su parte las antiguas tarjetas en 2D solo necesitaban un área de memoria para almacenar los datos, mientras que las tarjetas en 3D necesitan 3 bancos específicos de memoria de video: El buffer frontal, donde se almacena la imagen que está siendo mostrada en ese momento; el buffer trasero, necesario para almacenar la siguiente imagen que se está procesando; y el Z-Buffer, donde se almacena la profundidad o información de la tercera dimensión. Del tamaño de esta memoria dependerá la resolución de pantalla y la profundidad de color que se pueda mostrar. Veamos un ejemplo, si queremos usar el modo de color de 24 bits (16,7 millones de colores), vemos que para almacenar la información de cada píxel necesitaremos 3 bytes (1 byte = 8 bits) en cada buffer de memoria, es decir, un total de 9 bytes (3 por cada memoria). Si la resolución que queremos usar es 1024 x 768, multiplicando esto por 9 obtenemos un resultado de 7077888 bytes, o lo que es lo mismo, unos 7.08 Mb, por lo que para poder tener una imagen de resolución 1024 x 768 y 16.7 millones de colores, necesitaremos al menos una memoria de 8Mb.

Sin embargo en la actualidad se ha mejorado más aun, con tarjetas con mucha más memoria, por lo que ya no preocupa la memoria respecto a la resolución de pantalla y de la cantidad de colores. Ahora se usa principalmente para almacenar texturas, que se aplican sobre objetos y polígonos, muy importante para las tarjetas en 3D, ya que deben tratar una cantidad elevada de texturas, que pueden llegar a ser muy complejas, todo esto lleva a que si la tarjeta tiene poca memoria sea más fácil que se sature, y por lo tanto el rendimiento de la tarjeta sea peor.

También podemos destacar la mejora de tener un puerto doble, por lo que se pueden llevar a cabo dos operaciones independientes de lectura o escritura, lo que posibilita que se pueda acceder a memoria simultáneamente por el procesador gráfico y la RAMDAC.

Es importante no solo el tamaño de la memoria sino también la velocidad del bus de la misma (dado en MHz también), que nos indica a que velocidad se transmiten los datos. Para esta velocidad hay que tener en cuenta el tipo de la memoria de la tarjeta, porque si por ejemplo es DDR, la velocidad efectiva es el doble, ya que realiza dos cálculos por ciclo de reloj (por MHz). En la actualidad se puede llegar a una velocidad del bus de 850 MHz.

2.9.5 La tasa de refresco

La hemos nombrado anteriormente a la hora de calcular la frecuencia que debería tener la RAMDAC. Es el numero de veces por segundo que se refresca la pantalla (85Hz significa que se refresca 85 veces por segundo la pantalla). Las tasas tienen que ser de un valor elevado, para así evitar el “Parpadeo” que causa mucho cansancio a los ojos.

3 ARQUITECTURAS RECONFIGURABLES

En esta sección vamos a tratar de manera general que es un Hardware Reconfigurable. Una arquitectura reconfigurable de manera general, es una arquitectura que normalmente contará de un procesador RISC y de una parte reconfigurable (un array o matriz reconfigurable por puntos físicos de control). La idea es usar este tipo de hardware para aplicaciones que realizan volúmenes de cálculos muy grandes, pero para los que se trabaja de forma muy parecida, por lo que es muy probable que se puedan obtener mejoras sustanciales aplicándolas en hardware que puede ejecutar muchos cálculos en paralelo. Esta sección se va a distribuir de la siguiente manera: primero hablaremos de modo general de Hardware Reconfigurable, sus principales características, para que puede venir bien usarlo y los métodos teóricos para pasar código a este tipo de arquitecturas, nombrando algunos de los sistemas más conocidos. Segundo, daremos unos breves ejemplos de algunas de estas arquitecturas, para tener una mayor idea de cómo son en realidad. Tercero, como nuestro proyecto es hacer un algoritmo de renderización, a parte de otras arquitecturas reconfigurables, vamos a hablar del hardware de las tarjetas gráficas, que son el mecanismo más usado y potente en la actualidad a la hora de renderizar. Por último, vamos a hablar de manera mucho más extendida de Morphosys, la arquitectura que usamos en nuestro proyecto. Sus características, su primera versión y una posterior mejorada, los cambios que haremos en ella para adaptarla a nuestro proyecto, como se programa y el uso del simulador de la arquitectura.

3.1 Introducción a las arquitecturas reconfigurables

Las arquitecturas reconfigurables han experimentado un importante crecimiento en los últimos años tanto a nivel académico como comercial, particularmente las arquitecturas reconfigurables de grano grueso. De modo general, éstas consisten en una matriz con un cierto número de unidades funcionales (FUs), las cuales son capaces de trabajar a nivel de palabra en lugar de al nivel de bit como encontramos comúnmente en las FPGAs [12]. Esta granularidad reduce ostensiblemente el área de la arquitectura, la potencia que se consume, el tiempo de retardos y el tiempo de configuración comparado con las FPGAs. Otras posibles mejoras de estas arquitecturas son que tengan topologías flexibles, un pequeño espacio de memoria para guardar configuraciones, etc.

El objetivo principal de estas arquitecturas en la actualidad va dirigido a telecomunicaciones y multimedia. Lo que se hace es gastar más tiempo en ejecutar pequeñas partes de código con características bien definidas (kernels), que son partes críticas en lo que se refiere a su tiempo de ejecución. Las partes criticas más comunes son los bucles, difíciles de llevarlos (de mapearlos) a arquitecturas reconfigurables o grandes partes de código muy irregular, es decir con partes muy distintas las unas de las otras, por lo que es difícil plasmarlo en la arquitectura.

El cómputo reconfigurable es un concepto bastante reciente hoy en día, sin embargo, muchas de sus ideas ya las usábamos en procesadores de propósito general, como puede ser usar distintos componentes simultáneamente para procesar partes independientes de código, etc. Sin embargo, cuando hablamos de un sistema reconfigurable, vamos más

allá y nos referimos a sistemas que incorporan algún tipo de hardware programable, normalmente por distintos puntos físicos de control. Estos puntos pueden ir cambiando periódicamente para poder ejecutar distintas aplicaciones en el mismo hardware. De esta manera, y con este tipo de hardware podremos acelerar la ejecución de distintos algoritmos mapeando aquellas partes que conllevan un cómputo intensivo de cálculos.

Para comprender un determinado sistema reconfigurable, debemos atender a varios puntos. Lo primero que hay que examinar es la tecnología sobre la que se va a trabajar, pues los distintos sistemas reconfigurables usan hardware muy distinto, en los que sus bloques y unidades de computación varían mucho de unos a otros. Tan importante como el hardware, es cómo se efectúa la comunicación entre las partes de éste, ya que estos buses contribuyen de manera directa al tamaño global del hardware reconfigurable y a su coste. Una vez se tiene conocimiento de esta parte física, hay que mirar el software requerido para poder compilar algoritmos en estos sistemas y ver si es posible el mapeado automático o si hay que hacerlo manualmente.

Muchas veces, el hardware reconfigurable no es suficiente o no es capaz para llevar acabo la ejecución de aplicaciones enteras, por lo que la mayoría de arquitecturas de grano grueso están compuestas por una matriz reconfigurable y por un procesador, normalmente RISC. Lo que se hace es que algunas partes importantes del programa son llevadas a la matriz para que se ejecuten, mientras que el resto del código, que no puede ser eficientemente acelerado por la parte reconfigurable, es ejecutado por el procesador. Sin embargo aun no se le presta suficiente apoyo y atención a la integración de estas dos partes (matriz reconfigurable y procesador RISC), debido a que la unión entre el procesador y la matriz reconfigurable es a menudo muy débil y difícil, pues consiste esencialmente en las dos partes separadas conectadas por un canal de comunicación. El resultado de esto es una difícil programación de aplicaciones en estas arquitecturas y el exceso de comunicaciones que se puede dar teniendo solo un canal de comunicación.

No solo cambia el diseño del hardware de unos sistemas a otros, sino también el modelo de ejecución de los distintos entornos reconfigurables, por ejemplo el sistema NAPA **[13]** por defecto para la ejecución del procesador principal mientras se hace la ejecución en el hardware reconfigurable, pero permite que se den computaciones simultaneas usando primitivas “fork” y “join”, como en sistemas multiprocesador. Podemos hacer un pequeño resumen de los modelos principales de cómo en los sistemas compuestos por un procesador de propósito general y lógica reconfigurable se comportan cada una de estas partes.

- El hardware reconfigurable puede ser usado únicamente para aportar unidades funcionales reconfigurables al procesador principal [14]. Esto es permitir a la programación tradicional añadir instrucciones que pueden cambiar cada ciertos periodos de tiempo. Para el correcto funcionamiento de las unidades funcionales el procesador principal tiene que estar cada cortos periodos de tiempo indicando que instrucción será ejecutada.
- La unidad reconfigurable se usa como coprocesador [15], de este modo sería más grande que simplemente una unidad funcional reconfigurable en si, pero permite realizar computaciones sin la constante supervisión del procesador principal. Lo que hace éste es proveer los datos o la información necesaria para que el hardware reconfigurable encuentre los datos, así el sistema reconfigurable llevaría acabo sus operaciones al margen del procesador principal, devolviendo

los datos cuando finalice su ejecución. Esto hace que el procesador principal y el hardware reconfigurable puedan estar trabajando simultáneamente.
- La unidad de proceso reconfigurable se comporta como si fuese un procesador adicional en un sistema multiprocesador **[16]**, sin embargo la caché de datos del procesador principal no es visible para la unidad reconfigurable, lo que penaliza la comunicación entre ésta y el procesador principal.
- Por último, nos encontramos con sistemas en los que el hardware reconfigurable es una única unidad de procesamiento externa **[17]**. En este modelo, el hardware reconfigurable se comunica muy infrecuentemente con el procesador principal, cuando éste existe en el sistema.

Para poder pasar una aplicación a una arquitectura hay que seguir unos procedimientos más o menos comunes a todas las arquitecturas reconfigurables. En general, lo primero que necesitaremos es tener la aplicación en un determinado lenguaje de programación (en nuestro caso, y para el proyecto, desarrollaremos nuestro algoritmo de renderización en C++). Una vez teniendo el código, hay hacer una partición del programa o aplicación con la que estemos trabajando en partes, para ser implementadas, unas en el hardware reconfigurable correspondiente y otras que serán implementadas en el procesador principal, de tal manera que la o las secciones que se llevará al hardware reconfigurable debe ser sintetizada y descrita por un circuito de diseño a nivel de transferencia de registros. En este punto, dependiendo del hardware se seguirán distintas estrategias, dependiendo si se disponen herramientas especializadas para compilar, si hay que hacerlo todo manualmente, o una mezcla de ambas. Aquí es cuando nos encontramos con uno de los problemas más comunes que se dan para el desarrollo de esto tipo de arquitecturas, y es que aunque como venimos diciendo, se pueden conseguir buenos beneficios para cierto tipo de aplicaciones, las arquitecturas reconfigurables son a menudo ignoradas por los programadores, a no ser que puedan ser incorporadas de manera sencilla a sus sistemas. Para esto se requieren herramientas de diseño de software para la creación de configuraciones sobre el hardware reconfigurable. Podemos incluir desde un software que simplemente ayude a la creación manual de circuitos o hasta sistemas de diseño que realizan de manera totalmente automática el circuito. El método más potente es el diseño manual, pero en contra requiere un gran conocimiento de la arquitectura particular que se quiera usar. Por otro lado un sistema de compilación automática puede crear rápidamente un circuito para el sistema reconfigurable y lo hacen más accesible a los programadores, aunque tiene la desventaja que los circuitos que crea son menos eficientes que aquellos hechos manualmente. Es muy normal que se de una mezcla de ambas soluciones, por un lado para que no sea tan complicado mapear programas a un hardware reconfigurable y por otro para que no se produzca una caída de potencia como ocurre con el diseño totalmente automático. Para realizar el paso de un programa a una arquitectura hay que seguir un flujo de diseño, que será diferente para cada arquitectura, pero que en general tienen características comunes de unos sistemas a otros. Mostramos a continuación (Fig. 33) como serían tres tipos de flujo de diseño [18] según si se hace de manera automática, manual o con una mezcla de las dos.

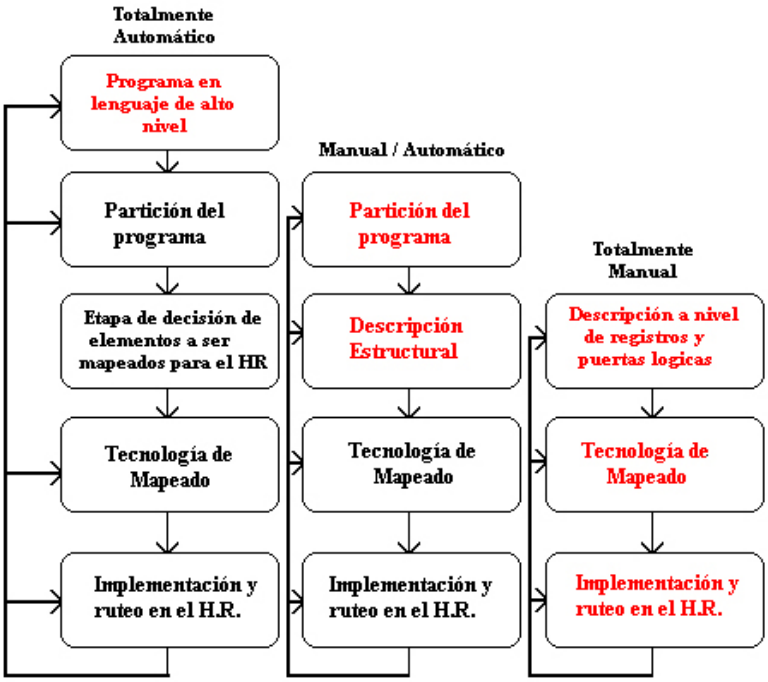

Figura 33. Flujos de diseño según tipo de mapeado. En rojo fases manuales.

En todo este tiempo se han propuesto diversas arquitecturas, sin embargo para muy pocas se han diseñado herramientas y metodologías con las cuales se pueda explotar el alto paralelismo. Hay una gran falta de herramientas eficientes para llevar estas partes críticas de código (para *mapear*) a matrices reconfigurables, mientras que algunas no tienen un buen soporte para poder ser diseñadas.

Podemos hablar de distintas arquitecturas reconfigurables. Morphosys (que es la que usamos en nuestro proyecto y que describiremos con más detalle posteriormente) que usa ensamblador para *mapear* manualmente estas partes críticas de código (*kernels*). Aquí el diseñador tiene que identificar y traducir las transferencias de datos entre el RISC y el array de celdas reconfigurables, lo que implica mucho movimiento de datos. RAPID, que soporta C como lenguaje de programación de los *kernels*. RAPID-C está especializado en pipeline, pero requiere muchos conocimientos de diseño acerca de la arquitectura y no es fácil para integrar con ANSI-C para completar le diseño de la aplicación. PACT, que usa un lenguaje NML, que es esencialmente un lenguaje ensamblador, para hacer los *kernels*. Existen herramientas para colocar y enrutar de manera automática a la hora de *mapear los kernels* a PACT XXP. Recientemente PACT ha comenzado a construir sistemas incluyendo un procesador ARM7 y usando un bus AMBA como canal de comunicación, pero aun no tienen establecido un flujo de diseño para una aplicación completa.

Todo esto se completa con otra arquitectura programable con procesadores VLIW (*very long instruction word*), que ya cuentan con muchos mas soportes y tienen muchas herramientas y metodologías de diseño desarrolladas, y dan buenas mejoras. Dentro de estas arquitecturas (procesador VLIW en lugar de un procesador RISC, más una Matriz reconfigurable,) entra ADRES, que veremos a continuación, nombrando sus ventajas sobre sistemas reconfigurables más comunes, como algunos de los antes mencionados.

Para concluir, podemos decir que estas arquitecturas reconfigurables de grano grueso han aventajado en los últimos tiempos a las tradicionales FPGAs. Una de las principales aplicaciones de estas arquitecturas reconfigurables no es solo poder *mapear kernels*, sino también aplicaciones enteras.

3.2 Tipos de arquitecturas reconfigurables

En esta sección encontraremos las arquitecturas reconfigurables más conocidas, explicando brevemente su funcionamiento y uso.

3.2.1 Arquitectura ADRES

El sistema reconfigurable ADRES (Architecture for Dynamically Reconfigurable Embedded System) [13] ha sido diseñado por miembros de distintas universidades Belgas (Katholieke Universiteit y Vrije Universiteit), junto con algunas herramientas y metodologías de diseño para el mismo.

Aquí vemos el sistema ADRES en general (Fig. 34 A), y el núcleo de este sistema (Fig. 34 B) compuesto por componentes básicos como son Unidades Funcionales (FUs) que pueden almacenar datos intermedios y que con capaces de trabajar a nivel de palabra (en lugar de a nivel de bit) seleccionada por una señal de control y Archivos de Registro (RF) conectados en la topología que nos muestra la imagen.

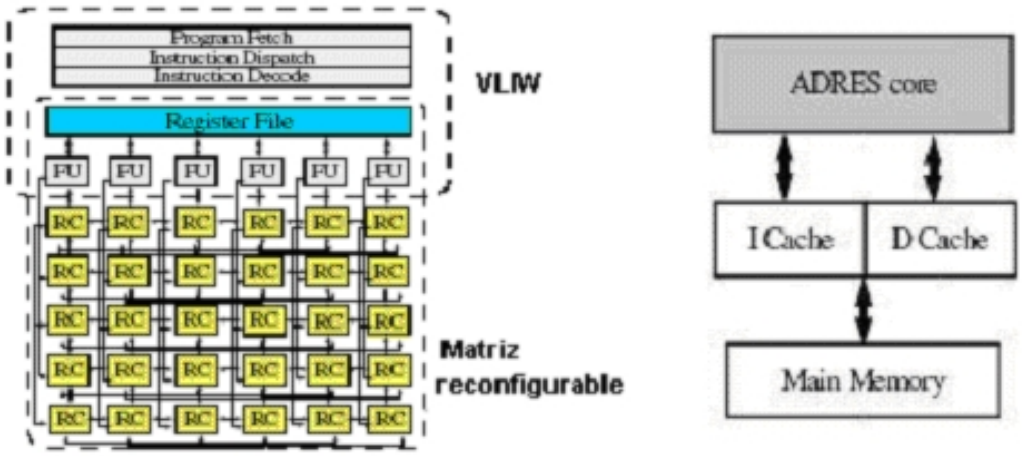

Figura 34.

A: Sistema Adres **B: Núcleo del sistema**

Como podemos observar, tenemos dos partes funcionales claramente diferenciadas, la matriz reconfigurable y el procesador VLIW. La primera se usa para acelerar el flujo de datos del *kernel* (código crítico) con un alto paralelismo, mientras que el procesador VLIW ejecuta el resto del código tratando de explotar el paralelismo a nivel de instrucción (ILP, *instruction level parallelism*). Estas dos partes funcionales ahorran

algunos recursos, ya que comparten algunas Fus y un RF, sin embargo esto no crea ningún conflicto entre ellas gracias al modelo procesador/co-procesador.

Por la parte del procesador VLIW, encontramos una file de FUs, conectadas todas juntas con un único registro (RF) multipuesto, típico de las arquitecturas VLIW. Estas FUs estarán también conectadas a la memoria. Las FUs del procesador VLIW, comparadas con las partes de la matriz reconfigurables, son mejores en término de velocidad y funcionalidad. Para la matriz reconfigurables, a parte de las FUs y el RF que tienen ambas partes en común, cuenta con muchas celdas reconfigurables.

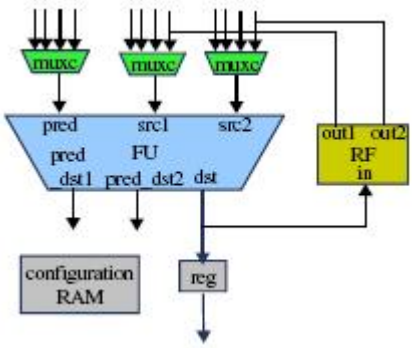

Figura 35. Celdas reconfigurables

Básicamente están compuestas por FUs y FRs también. Las FUs pueden soportar distintos conjuntos de operaciones, incluyendo algunas específicas para admitir bucles. Cada celda contara un unos multiplexores, encargados de dirigir los datos desde las diferentes fuentes que le lleguen. Hay una memoria RAM, para guardar contextos locales. La comunicación entre las dos partes funcionales de ADRES se realiza a través del RF que tienen en común ambas.

Gracias a todo esto, ADRES tiene muchas ventajas. Por un lado al tener un sistema VLIW en vez de un sistema RISC, como otros sistemas reconfigurables, con el que acelera el código no crítico. También reduce la complejidad a la hora de programar por el RF compartido y por los accesos a memoria entre VLIW y la matriz reconfigurable, por último, los recursos compartidos reducen notablemente el coste.

Diseño de flujo C-Based (basado en C)

Para guiar esta breve explicación, veamos como sería el flujo de una arquitectura ADRES.

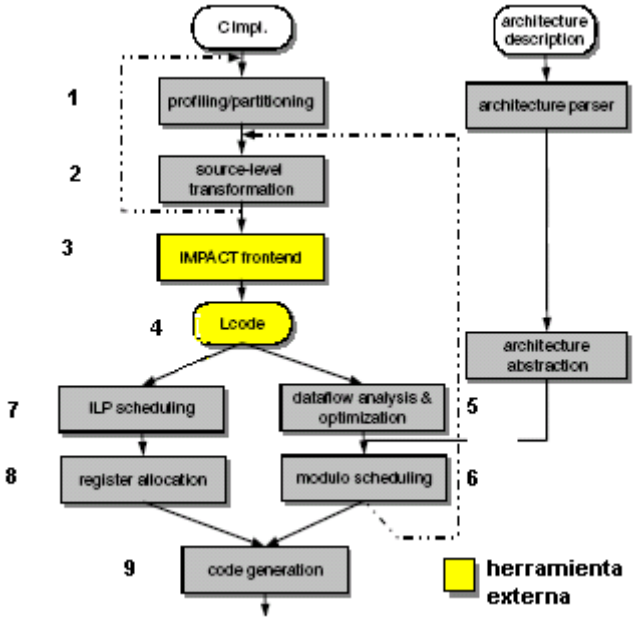

Figura 36. Etapas del diseño de flujo

El diseño lo realizamos a partir de una implementación de la aplicación en un lenguaje de alto nivel, en este caso el lenguaje usado es C.

1. Aquí se identifican aquellos bucles candidatos a ser hechos en la matriz reconfigurable, basándose en el tiempo de ejecución y en la posible ganancia que se podría llegar a obtener.
2. En esta etapa se intenta rescribir el código crítico (kernel) para poder aplicar pipeline y maximizar su rendimiento.
3. Este compilador (IMPACT), fue diseñado por los mismos miembros de las universidades belgas antes citadas. Lo hicieron antes que esta arquitectura y posteriormente lo adaptador a ella para poder explotarla mejor. IMPACT se encarga de parsear el código C, analizándolo y optimizándolo, creando un código intermedio (Lcode), que se usa como entrada en la siguiente etapa.
4. Lcode
5. En esta etapa, lo que se hace es describir la arquitectura objetivo (ADRES) mediante un lenguaje XML, y nos queda transformada internamente a un gráfico.
6. En la siguiente fase se toma el código intermedio (Lcode) y el gráfico creado anteriormente como entradas del planificador, y se aplica el algoritmo de planificación que haya para conseguir un gran paralelismo para los kernels.
7. Por este otro lado lo que se hace es aplicar técnicas cotidianas de planificación ILP para obtener un pequeño paralelismo para el código no crítico.
8. Aquí simplemente se asignan que registros se usaran en VLIW para realizar el código no crítico.
9. Y por último se crea un único código planificado, tanto para la matriz reconfigurable como para el procesador VLIW que podrá ser ejecutado en un simulador de nuestra arquitectura.

La mayoría de las etapas se realizan de manera automática con las herramientas que se han creado para el diseño del flujo de ADRES, pero aun hay etapas que

necesitan que el propio diseñador introduzca datos para una mejora más considerable. En particular hablamos de la primera fase, a la hora de realizar el particionamiento. Las decisiones acerca de esto tienen que tomarse en la fase más temprana, principalmente porque todos los requerimientos de optimización son muy diferentes de la matriz reconfigurable al procesador VLIW.

Con la visión de esta arquitectura, y sobre todo del diseño del flujo para una aplicación, y a pesar de las diferencias entre ADRES y Morphosys, al menos conseguimos tener una idea de cómo encaminar nuestro trabajo, distinguiendo distintas fases, y entendiendo mejor a que nos estamos enfrentando. La idea a la hora de plantearnos el desarrollo de nuestro renderizador, hecho en C++, a Morphosys, no será intentar escribir todo el algoritmo con lo que eso conlleva sobre Morphosys, sino estudiarlo, ver distintas estrategias sobre el código escrito en C++ (como podría dividirse, partes comunes, bucles, etc.) y encontrar partes críticas en el código que nos puedan causar problemas y eso llevarlo a Morphosys, planteando las distintas estrategias, y buscando la mejor que encontramos para este hardware reconfigurable.

3.2.2 Arquitectura XXP (Extreme Processing Platform)

Esta nueva arquitectura [19] nos presenta una nueva tecnología de procesamiento reconfigurable de datos en tiempo de ejecución, reemplazando el concepto de instrucción secuencial por configuración secuencial y dando estrategias de reconfiguración muy eficientes.

Lo que hay que pensar para hacer buen uso de dispositivos de computación reconfigurable es la manera de separar la "computación" y el proceso de configuración. La estrategia de configuración de XXP nos da un modo revolucionario para solventar este problema. La idea principal del desarrollo de XXP se basa en tres puntos:

Procesamiento de data stream

Un data stream es una secuencia de paquetes de datos individuales que viajan a través del grafo de flujo de datos que define el algoritmo. Un paquete de datos en XXP es una palabra máquina de 24 bits, de manera que solo numero y orden de los paquetes que van a través del grafo es importante. Estos streams pueden venir de fuentes naturales como transformadores A/D. Una vez que los datos están en memoria RAM, XXP puede generar direcciones, para producir un data stream. Igualmente, datos calculados pueden ser enviados a distintos destinos como a convertidores D/C o memorias.

Configuraciones

Las configuraciones en XXP son módulos de cálculo básico en paralelo, los cuales derivan del flujo de datos del grafo del algoritmo. Los nodos del grafo son mapeados a operaciones máquina básicas, como sumas, multiplicaciones, etc. Los arcos del grafo son las conexiones entre los nodos. Como dijimos antes, se debe separar el procesamiento de datos con la configuración. Para esto, el concepto básico es en

reemplazar los streams de instrucciones Von-Neumann por un stream de configuración y el proceso de streams de los datos.

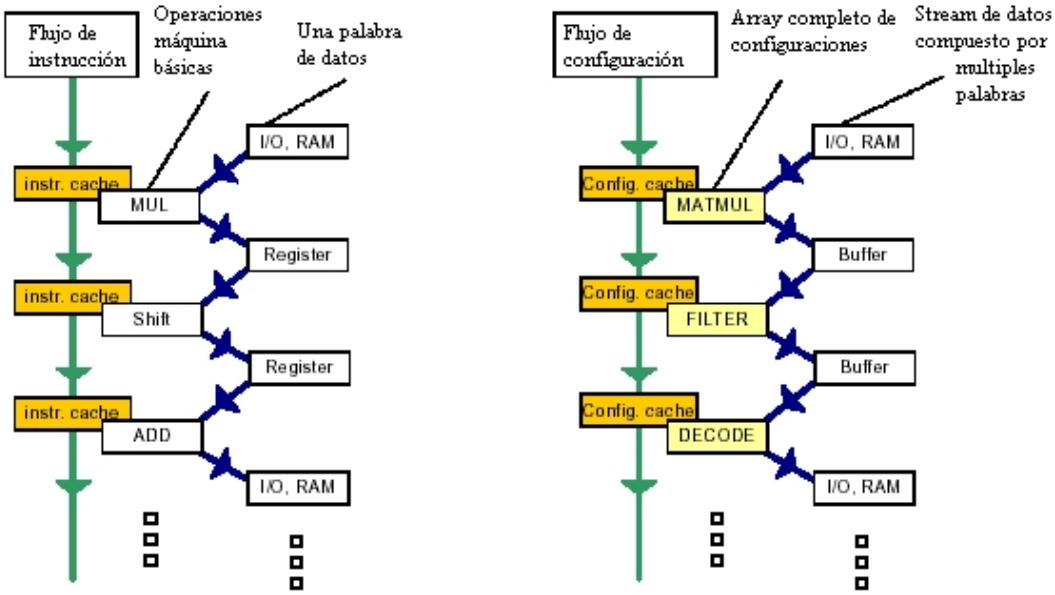

Figura 37. Paso del flujo de instrucción al flujo de configuración

Con este modelo de programación, la aplicación se separa en pequeñas secuencias las cuales pueden ser procesadas en paralelo. Conseguimos que una sola configuración, procese un stream de datos.

Desarrollo de aplicaciones

A parte de la arquitectura, también existe software específico para ella, "XXP development suite" es una herramienta que nos ayuda al desarrollo de programas así como a su corrección. Gracias a su regularidad y simplicidad, un compilador de alto nivel puede extraer instrucciones a nivel de paralelismo y pipelining que esta implícito en los algoritmos.

Todo esto hace que XXP sea una buena elección para el campo de aplicaciones donde grandes streams de datos deben ser procesados. XXP esta bien situado en muchos campos, como procesamiento de imagen y video, visualización en tiempo real…

Componentes físicos y estructuras XXP

Una arquitectura XXP (Fig. 38), está compuesta básicamente de cuatro componentes. Array de proceso de elementos, organizamos como arrays de procesos (Processing arrays, PAs). Un paquete dirigido a la comunicación interna. Un árbol Manager de Configuración (CM). Y un conjunto de módulos de entrada salida (I/O). Con esto se soporta la ejecución de múltiples flujos de datos funcionando en paralelo.

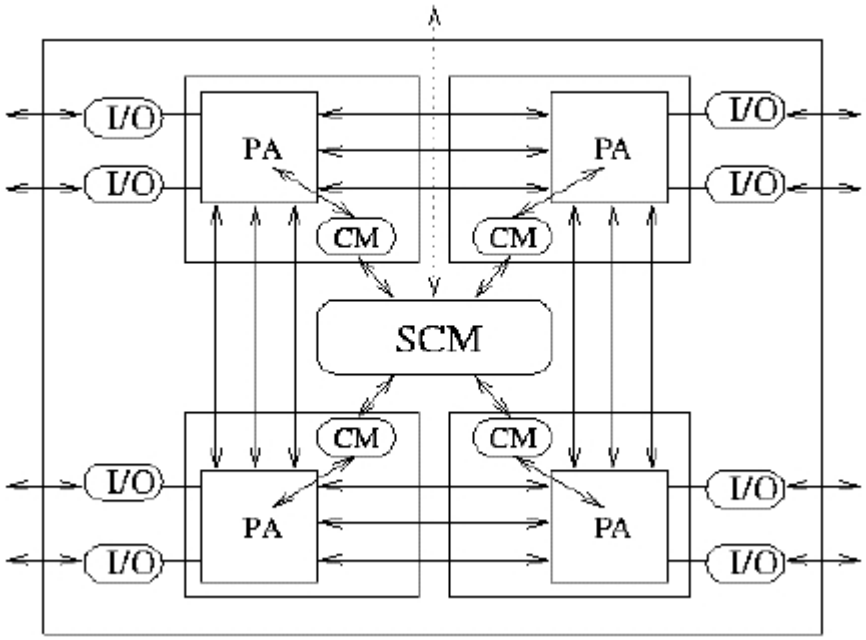

Figura 38. Arquitectura XXP con cuatro PAs.

Un array de procesos (PA) y un CM de bajo nivel se agrupa en una estructura llamada PAC (Cluster de array de procesos). El CM de bajo nivel es el encargado de escribir la configuración de datos en los objetos configurables del PA. Lo normal es que haya varios PACs para la construcción de un dispositivo XXP, de manera que según aumenta su número en el diseño, más CMs son añadidos, te manera que forman una estructura en árbol. Al CM que sea el root de dicho árbol se le llama CM Supervisor (SCM), y normalmente va conectado a una memoria RAM externa. En algunos casos el SCM puede actuar como un simple CM, lo que hace que se puedan soportar arquitecturas con varios dispositivos XXP (Fig. 39).

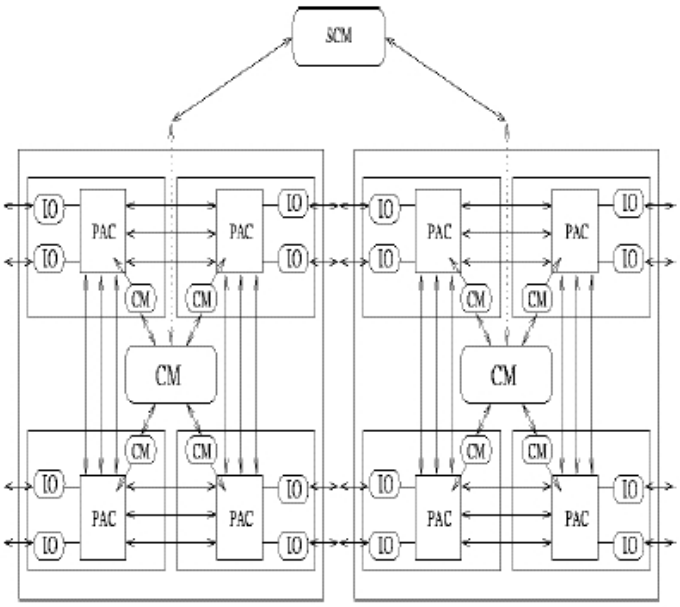

Figura 39. Múltiples dispositivos XXP unidos por un CM

Para estos casos de aplicaciones de múltiples dispositivos, el SCM de cada dispositivo XXP actúa como un simple CM, y se añade un SCM adicional que se usa como el root del árbol CM.

Por último presentamos con más detalle la arquitectura y los objetos de un dispositivo XXP (Fig. 40).

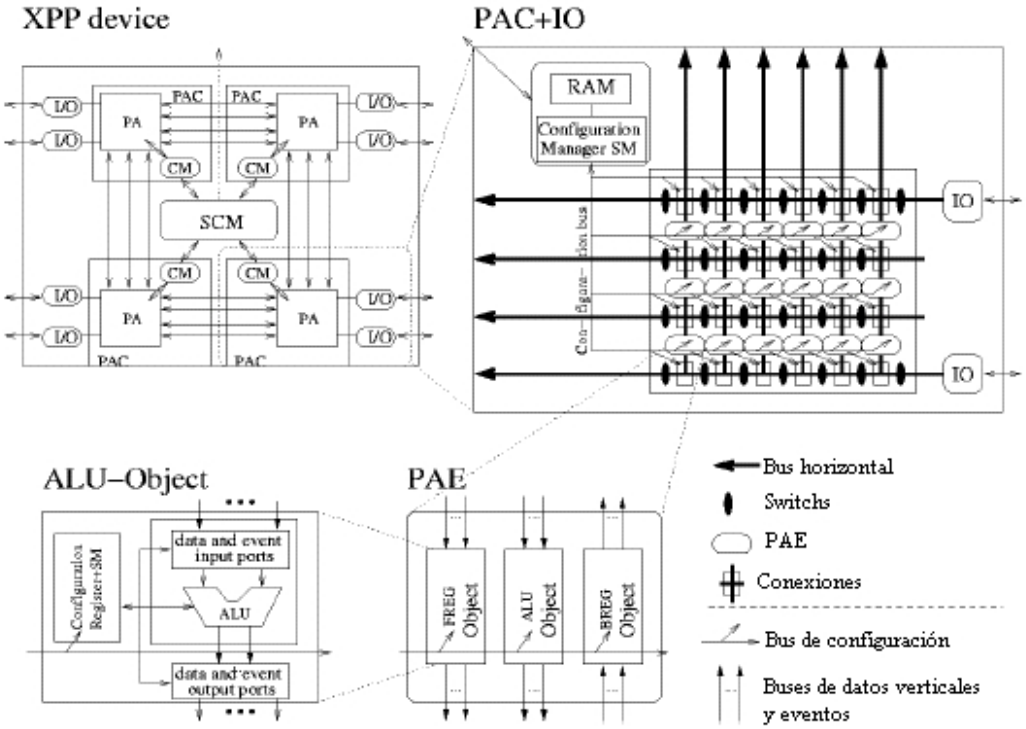

Figura 40. Estructura y objetos de un dispositivo XXP

Los buses horizontales son usados para conectar objetos sin un PAE en una fila. Verticalmente, cada objeto puede conectarse a los buses horizontales. Los switches configurables son usados para segmentar las líneas de comunicación horizontalmente. El ruteo vertical se lleva a cabo mediante Registros integrados en el PAE. Los objetos de entrada salida (I/O) son usados para interactuar con variados dispositivos externos, usando un protocolo handshake. Todos estos módulos están conectados por los switches al bus horizontal. El PAE contiene distintos objetos, el registro BREG (backward register), que da facilidades de ruteo de abajo a arriba y funciones aritméticas, el registro FREG (forward register), que proporciona recursos de ruteo en la dirección contraria que el anterior, y funciones para el control del flujo de datos, y normalmente también cuenta con un objeto ALU, que se basa en la ALU en si misma, con sus puertos de entrada y salida, una máquina de estado (SM) para el control de la ejecución y un CM para controlar la conexión.

Los puertos de entrada y salida pueden tanto recibir como transmitir paquetes de datos y eventos. Los paquetes de datos suelen ser procesados por las ALUs, mientras que los paquetes de evento son procesados por la máquina de estados que controla la ejecución.

Como dijimos en la introducción de hardware reconfigurable, para especificar algoritmos para XXP, usamos un lenguaje específico, NML (Native Mapping

Language). La herramienta "XXP development suite" (XDS) diseña el flujo para la programación sobre XXP según muestra la siguiente figura (Fig. 41)

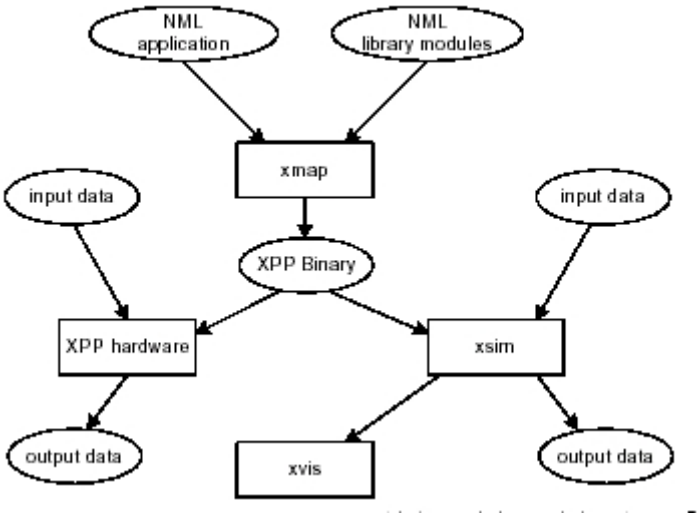

Figura 41. Diseño de Flujo XDS

Esta herramienta hace el diseño tanto para la arquitectura, como para el simulador que existe de la misma (XSIM). El algoritmo que obtenemos en XXP binario, puede ser llevado directamente a su ejecución sobre el hardware o sobre el simulador.

3.2.3 Procesador SMeXXP Media

Como dijimos en la introducción, últimamente PACT ha desarrollado sistemas XXP incluyendo además un procesador ARM7 [20], memorias RAMS, y distintos periféricos. Está desarrollado con orientación a aplicaciones multimedia (como por ejemplo un encoder MPEG4 [20]), Veamos su estructura con la siguiente figura (Fig. 42)

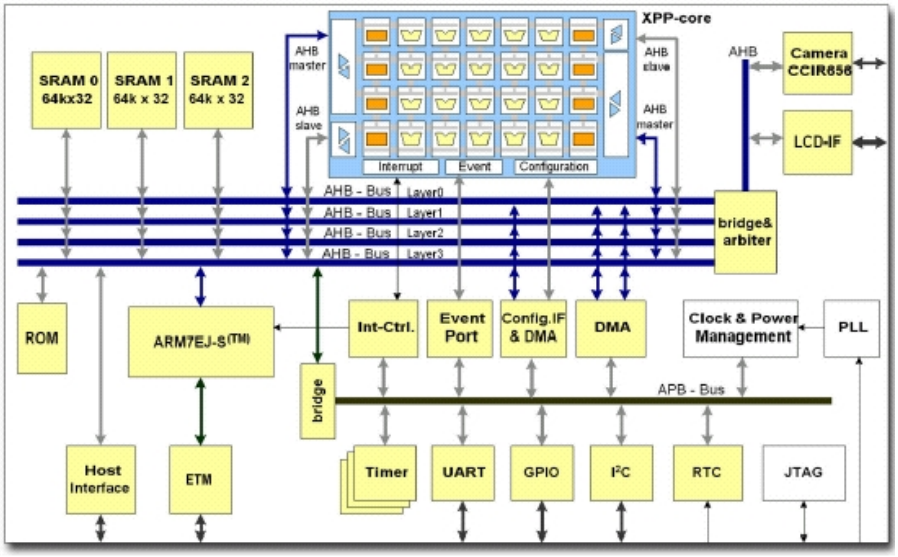

Figura 42. Procesador SMeXXP

El corazón de esta estructura son los buses AMBA, que conforman un bus AHB que conecta todos los componentes como se muestra en la figura anterior. Veamos de manera resumida los componentes.

ARM7

Es un poderoso procesador RISC de 32 bits. Este procesador funciona de manera excelente para todas las tareas y control de flujos orientados a tareas de aplicaciones de video. Su rango de reloj de frecuencia oscila entre los 13 y los 104 MHz.

Memoria

Hay tres bancos de memorias RAMS disponibles tanto para el procesador ARM7 como para el núcleo XPP. Los bancos proveen accesos alineados e independientes de 32 bits desde los buses AHB.

Controlador DMA

Hay dos canales DMA para modos de transferencias a través de los buses AHB. El DMA genera solo secuencias de direcciones lineares. Para casos de direcciones más complejas, la mejor opción es usar el núcleo XXP.

Periféricos

Los periféricos típicos requeridos para ayudar al tipo de aplicaciones hacia los que va dirigida esta arquitectura están integrados en el chip. El bus-APB (bus de periféricos) está conectado por un puente al bus-AHB. Algunos de estos periféricos son el Timer (temporizadores), RTC (tiempo real de reloj), puertos de configuración XXP, etc.

Núcleo XXP

Es un núcleo XXP parecido al descrito anteriormente, de 16 bits. Está formado por un array de 5 x 4 PAEs, 2 x 4 RAM PAEs, dos interfaces maestros de entrada salida AHB, y otros dos interfaces esclavos. (Fig 43)

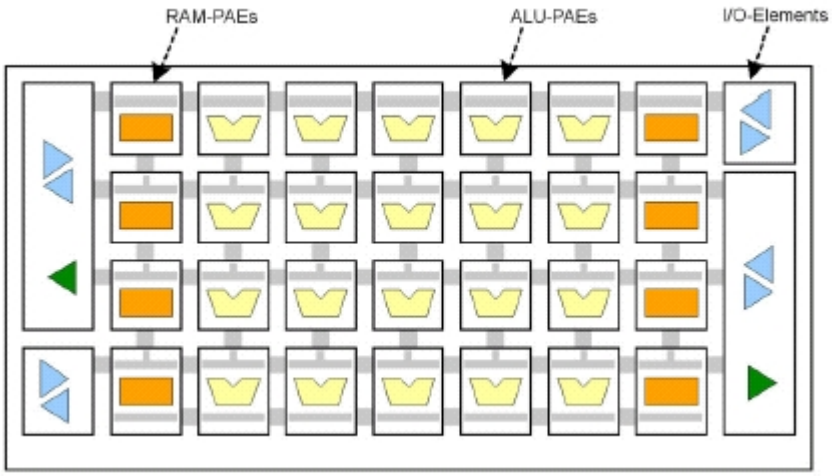

Figura 43. Núcleo XPP del procesador SMeXXP

Donde los módulos PAEs ya los vimos descritos anteriormente y los RAM- PAEs, son iguales, pero en vez de tener una ALU como elemento principal, tienen una memoria de 512 x 16 bits, aparte del registro FREG y BREG.

3.2.4 Arquitectura RaPiD

El proyecto RaPiD [21] ha estado explorando una arquitectura de computo configurable denominada RaPiD (Reconfigurable Pipelined Datapath), que esta optimizado para el dominio de procesamiento de señales e imágenes. El objetivo de esta arquitectura es juntar una alta mejora para las características de estos dominios unido a un bajo consumo tanto económico como energético.

RaPiD está optimizado para operaciones aritméticas sobre valores de datos de muchos bits. Hay un ahorro muy grande si se utilizan unidades específicas de computación optimizadas para palabras de la longitud usada en estos cómputos. El camino de datos de RaPiD soporta un alto nivel de paralelismo. En particular los algoritmos "sistólicos" están muy bien situados en esta arquitectura. El camino de datos de RaPiD implementa la entrada/salida usando múltiples stream de datos independientes. Estos streams van conectados directamente a dispositivos de streams externos o a un sistema stream de memoria. RaPiD coge ventaja de la computación tan repetitiva y regular que se da en estos dominios para general las señales de control que controlen el camino de datos por medio de una ruta de datos muy eficiente.

Según los estudios realizados por miembros de la universidad de Washington, RaPiD puede conseguir hasta un rendimiento de 1.5 billones de operaciones de acumulación o multiplicación por segundo para un buen número de importantes aplicaciones usando un array de un tamaño moderado ejecutándose a 100 MHz.

La arquitectura RaPiD normalmente esta formada por cientos de unidades funcionales, desde simples registros, hasta multiplexores, desplazadores, ALUs y memorias. Estas unidades suelen estar colocadas de manera lineal (Fig. 44) y conectadas usando interconexiones configurables basadas en buses segmentados. Tanto los buses, como las unidades funcionales de esta arquitectura configurable estás preparadas para trabajar a nivel de palabras, en lugar de bits. Esta alineación de las unidades funcionales de la arquitectura puede parecer algo limitada. La clave está en como mapear algoritmos multidimensionales a arrays lineares [22].

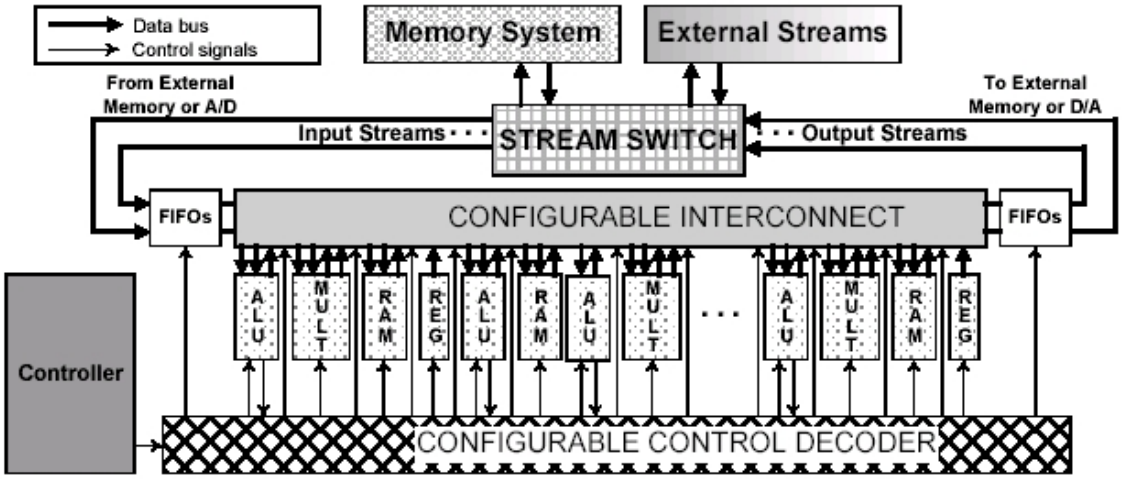

Figura 44. Estructura de RaPiD

En RaPiD, cada unidad funcional recibe los valores de los datos de los buses seleccionados en la interconexión configurable, obteniendo una computación específica por un conjunto de señales de control y resultados de salida en la forma de valores de datos.

A parte de las unidades funcionales antes mencionadas, otras más específicas pueden ser añadidas a la arquitectura RaPiD. Lo más normal, para dominios específicos es añadir una unidad funcional de propósito general, que ayude a mejorar funciones individuales que no tienen entradas de control.

La interconexión configurable consiste en un conjunto de buses segmentados colocados al lo largo de toda la arquitectura. Algunos de estos buses segmentados están unidos por conectores que pueden ser configurados para conectarse con segmentos adyacentes creando segmentos más largos, con la posibilidad de llevar a cabo pipelining. Todos los buses tienen el mismo ancho.

Cada entrada a una unidad funcional selecciona uno de los buses de la interconexión mediante un multiplexor, que opera según un conjunto de señales de control. Los datos de salida son conducidos por buses seleccionados también por señales de control.

La operación que realizará RaPiD en cada momento vendrá dado por las señales de control que determinarán que operaciones hace cada unidad funcional y como los datos serán conducidos por la interconexión configurable a través de las distintas unidades funcionales. Un conjunto entero de señales de control dentro de un ciclo de reloj es llamado "datapath intruction" y un programa en RaPiD es una secuencia de estas instrucciones que hacen la computación deseada. Una arquitectura RaPiD típica, puede contener de orden de unas 100 unidades funcionales, requiriendo para poder controlar todas más de 5000 señales de control.

Seguramente, el problema más común en estas arquitecturas reconfigurables es la generación de las señales de control que controlen la ejecución sobre la arquitectura, ya que el alto nivel de paralelismo unido a la reconfigurabilidad genera un gran número de potenciales señales de control. Generar y conducir estas señales según los ciclos de reloj

por la ruta de datos es muy costoso tanto en superficie para la arquitectura como en consumo. RaPiD solventa este problema dividiendo las señales de control en dos tipos, duras (traducido literalmente, hard) y suaves (soft). Las primeras, comúnmente son señales de configuración que distribuye la memoria estática que cambia muy pocas veces cuando la arquitectura es reconfigurada para diferentes aplicaciones. Las segundas, cambian cada ciclo de reloj dirigido por un programa de control.

RaPiD es programado usando un lenguaje llamado RaPiD-C que está especializado en algoritmos en paralelo. Los programas en RaPiD-C son compilados en archivos de configuración y programas que los controlan por el proceso descrito a continuación (Fig. 45).

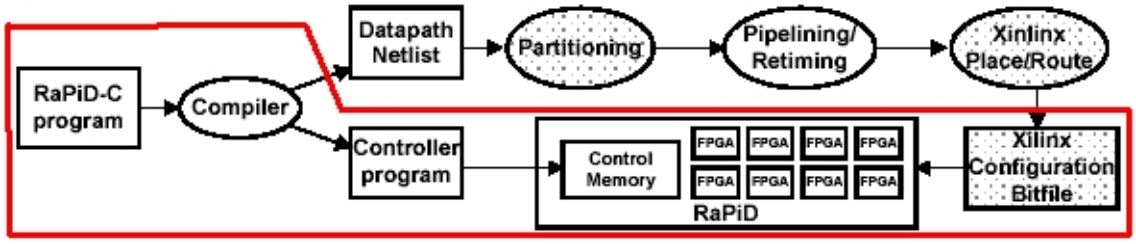

Figura 45. Herramienta de flujo CAD para compilación

Aquí, el compilador analiza en programa en RaPiD-C, una vez analizado lleva a cabo una partición de programa, para cada una de las unidades funcionales, para realizar de manera eficiente las operaciones de datos y el programa de control, el cual genera las señales de control que operarán la arquitectura. Todo esto se hace según la parte señalada dentro del cuadro de la imagen anterior (Fig. 45). El otro flujo que se muestra, es el flujo aumentado para soportar también la compilación de RaPiD-C para el simulador de esta arquitectura, donde será el programador en este caso, el que realice la partición, indicando que operaciones se harán en cada chip.

4 MORPHOSYS

4.1 Introducción

Morphosys es una arquitectura reconfigurable desarrollada por investigadores de la Universidad de California, Irvine. La idea del desarrollo de esta arquitectura proviene de la demanda de mayor velocidad y eficiencia en la ejecución de operaciones multimedia para aplicaciones gráficas. Es una arquitectura de grano grueso. Presentamos a continuación una descripción de M1, la primera versión de Morphosys. Está formado por un procesador principal llamado TinyRISC, consistente en un procesador RISC, con una versión simplificada del conjunto de instrucciones MIPS, una matriz de 8x8 celdas reconfigurables (RC), un Frame Buffer, una memoria de contexto, y un controlador DMA.

4.2 Descripción M1

Esta versión de MorphoSys es más adecuada para el tratamiento de imágenes y audio, debido a que el acceso a memoria que hacen es siempre el mismo, al margen de los datos que tengan de entrada, además que a la hora de hacer estos procesamientos, no se dan casi nunca saltos condicionales. Vamos a describir brevemente cada una de las partes por las que está compuesta M1. En la siguiente figura podemos observar su diagrama de bloques.

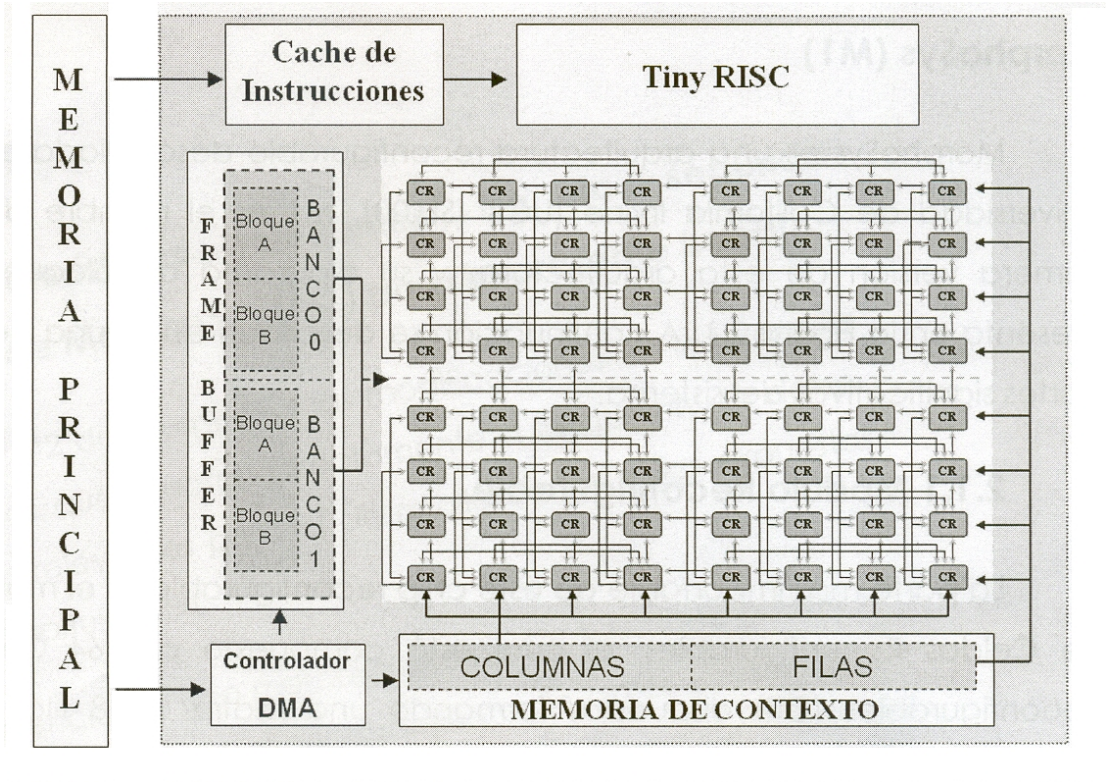

Figura 46. Diseño de Morphosys M1

4.2.1 Matriz Reconfigurable

Ésta es sin duda la parte más importante del módulo reconfigurable. El array está formado por 64 celdas reconfigurables, dispuestas en una matriz de 8 columnas y 8 filas. El tamaño de esta matriz está diseñado a propósito, dado que en el procesamiento de imágenes, con frecuencia se procesan datos de 8x8 elementos, por lo que es posible aprovechar el paralelismo en la ejecución de las celdas para obtener un mejor rendimiento de las aplicaciones.

4.2.2 **Red de interconexión de las celdas**

Las celdas tienen un conjunto de interconexiones, que permite que estén comunicadas. Por ello, se estructura la matriz de celdas en cuatro cuadrantes 4x4. Para cada cuadrante, cada celda está conectada con las 4 más cercanas, además de estar conectada con todas las celdas de su fila y su columna. El conjunto de interconexiones también incluye conexiones entre cuadrantes, de manera que se puede conectar la salida de una celda con hasta otras 4 celdas de otro cuadrante, siempre que estén en la misma fila o columna.

4.2.3 **Celda Reconfigurable**

Cada celda reconfigurable se asemeja a la ruta de datos de un procesador. Está formada por una ALU-Multiplicador, desplazador de 32 bits en la salida, multiplexores para seleccionar la entrada a la ALU, banco de registros, y registro de contexto. Además, también se incluye una pequeña CR-RAM para guardar datos en la misma celda. En la figura 47 podemos ver un diseño de una de las celdas.

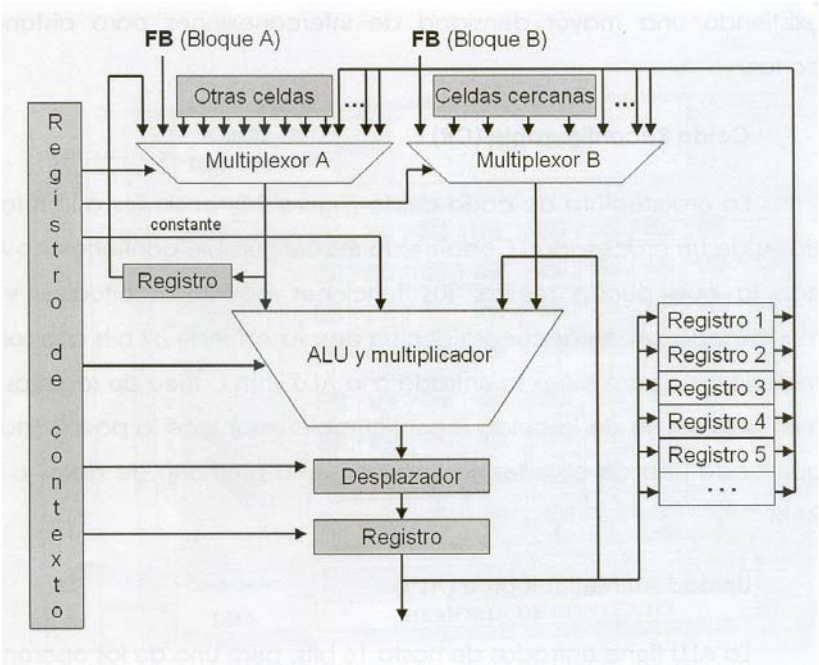

Figura 47. Diseño de las celdas

- ALU: Tiene entradas de 16 bits, pero solo puede generar salidas de 28 bits, debido a que una de las entradas del multiplicador debe ser de 12 bits. Además de tomar los datos de la salida de los multiplexores que contiene la celda, también los puede tomar de una constante que se guarda en el registro de contexto, o del registro de salida. La operación de multiplicación se ha diseñado cuidadosamente para que ocupe un solo ciclo, pues esta operación es muy común en aplicaciones multimedia.

- Multiplexores: Seleccionan los datos para cada entrada de la ALU. El multiplexor A puede tomar la entrada de la salida de las cuatro celdas más cercanas, de las celdas de su cuadrante o intercuadrante que están en la misma fila o columna, de un registro interno de realimentación, del banco de registros de la celda, o de la línea de datos que la conecta con el Frame Buffer. El multiplexor B recibe como entradas la salida de las cuatro celdas más cercanas, el banco de registros interno de la celda, o la línea de datos del Frame Buffer.

- Registro de contexto: Este registro realiza las funciones de unidad de control de la celda reconfigurable, cargándose en él la información (contexto) que reprogramará las funciones de cada uno de los dispositivos de la celda, como la carga de los registros de salida o del banco, la operación a realizar por la ALU, la selección de datos de los multiplexores, etc. El contexto también puede incluir una constante de 12 bits que se usaría como entrada de la ALU.

4.2.4 **Memoria de contexto**

La idea de memoria de contexto (MC) surge para aprovechar la cantidad de cálculos idénticos que se realizan en las aplicaciones de tipo multimedia, sobre datos distintos, pero para los que existe solo un flujo de control. Para poder realizar dichos cálculos de manera paralela, se quiso que las celdas compartieran simultáneamente la misma configuración y contexto, como en un sistema SIMD, pero permitiendo también la ejecución de configuraciones distintas en el módulo reconfigurable simultáneamente. Para ello, las celdas que se encuentran en la misma fila o columna compartirán la misma configuración, permitiendo configuraciones distintas en filas o columnas distintas. Este sistema permite reducir el tamaño de la memoria de contexto, ya que la misma configuración servirá para todas las celdas de la misma fila o columna, reduciendo también el tiempo de carga de contextos, así como la complejidad de la red de conexión de celdas.

Es obvio que no se puede configurar a la vez contextos para filas y para columnas, por ello la memoria de contexto está formada por dos bancos, uno para filas y otro para columnas, cada uno de ellos de 256 bits, 32 por cada celda. Gracias a este sistema, se puede hacer una reconfiguración dinámica, ya que dependiendo de los contextos que se carguen en cada momento, la funcionalidad de cada celda será distinta.

Con respecto al tiempo de penalización de esta memoria, al ser interna, carga los registros en las celdas en un único ciclo. Además, se puede solapar la carga de contextos en la memoria de contexto con la ejecución de las celdas, de manera que no se pierda tiempo en la ejecución.

4.2.5 **Frame Buffer (memoria de Datos)**

El Frame Buffer (FB) es la memoria interna de datos del módulo reconfigurable, la encargada de suministrar los datos con los que operarán las celdas. Está compuesta por dos bancos de memoria, cada uno de los cuales está dividido en dos bloques. De esta manera, para que el FB mande simultáneamente los datos de las dos entradas a cada celda, dichos datos deben estar en el mismo banco, pero en bloques distintos, ya que únicamente hay una conexión por bloque con las celdas reconfigurables, por cada banco de memoria.

La idea de dividir el FB en dos bancos nos sirve para solapar la carga de datos del exterior al FB en uno de los bancos, con la carga de datos desde el otro banco del FB a las celdas reconfigurables, sin obtener un incremento en el tiempo de ejecución.

Los buses que comunican el FB con las celdas de la matriz tienen un ancho de 8 bytes, permitiendo la transferencia de dos datos de 8 bits a cada celda de una fila o columna. Los modos de transferencia permitidos para M1 son:

- Transmitir el mismo dato a las 64 celdas
- Transmitir el mismo dato a cada fila o columna (se cogen 8 datos consecutivos de 8 bits del FB)
- Transmitir un dato diferente para cada celda (se cogen 64 datos consecutivos de 8 bits del FB)

4.2.6 **Controlador DMA**

Es el encargado de realizar las transferencias entre la memoria externa, por un lado, y el Frame Buffer y la Memoria de Contexto, por otro. El controlador DMA recibe las órdenes de transferencia del procesador TinyRISC, teniendo en cuenta que no puede realizar transferencias donde estén involucradas el FB y la MC simultáneamente.

4.2.7 **Procesador principal TinyRISC**

El TinyRISC es un procesador RISC de 32 bits que se encarga de controlar el sistema. Tiene un repertorio de instrucciones similar a MIPS, ampliado con instrucciones de control propias de Morphosys, y un pipeline de cuatro estados: fetch, decodificación, ejecución y escritura. Además, cuenta con un banco de 16 registros, de los cuales el registro 0 siempre tiene el valor 0. Entre sus funciones, el TinyRISC se ocupa de cargar la memoria de contextos, cargar los datos de la memoria externa al FB, almacenar los resultados de Morphosys en la memoria externa, y ciertas funciones de control del array de celdas y del controlador DMA.

4.2.8 Resumen y problemas de M1

Como resumen de Morphosys, podemos indicar que, arquitectónicamente, es un sistema reconfigurable dinámicamente, de grano grueso, que incluye una memoria de contexto organizada en dos bancos, y una memoria interna de datos (FB) con dos bancos totalmente independientes.

Desde el punto de vista de las ventajas de la arquitectura, podemos resaltar la ejecución de la misma configuración sobre toda una fila o columna, y la ejecución de configuraciones distintas en distintas columnas o filas, procesamiento de datos en el módulo reconfigurable realizado en paralelo con las cargas de contextos en la MC o con la transferencia de datos con el FB. Aunque hay que tener en cuenta que las cargas de los contextos y las transferencias desde la memoria externa no se pueden realizar en paralelo.

En M1 se dan algunos problemas, y que en particular para nuestro proyecto resultan importantes, como es que al ser un sistema SIMD, las celdas reconfigurables tienen problemas por falta de autonomía para ejecutar instrucciones condicionales if-then-else de manera local. A parte de esto, también existen otras pegas para esta arquitectura, como puede ser que el ciclo de reloj es el del camino más largo, que corresponde con el acceso al FB, pero hay operaciones de las celdas que no acceden a éste, por lo que el ciclo de reloj será superior al tiempo que tardan en ejecutar esas instrucciones las celdas, lo que implica una perdida de rendimiento. Otro problema es que la aplicación que se ejecute en M1 acceda a datos de manera aleatoria, ya que cuando se acceden a los datos, se presupone que estos están almacenados de manera consecutiva, con lo que tendríamos una penalización muy grande.

Estos problemas se intentan apaliar con la segunda versión de Morphosys, M2, que vamos a ver a continuación de manera resumida, indicando las mejoras que tiene respecto a M1.

4.3 Descripción M2

En la actualidad, han salido nuevas aplicaciones multimedia que se han hecho muy importantes, como es el tratamiento de imágenes 3D, como puede ser ray-tracing (que ya ha sido estudiado para ser implementado con Morphosys [21]), rendering (caso que vamos a estudiar nosotros), etc. En estas aplicaciones hay un problema fundamental con M1, que era el ya mencionado de los saltos condicionales, además las aplicaciones de imágenes 3D no tienen por que seguir un patrón definido en el acceso a datos. En la siguiente figura vemos el nuevo diagrama de bloques para M2 y a continuación describimos rápidamente las mejoras introducidas (Fig. 48).

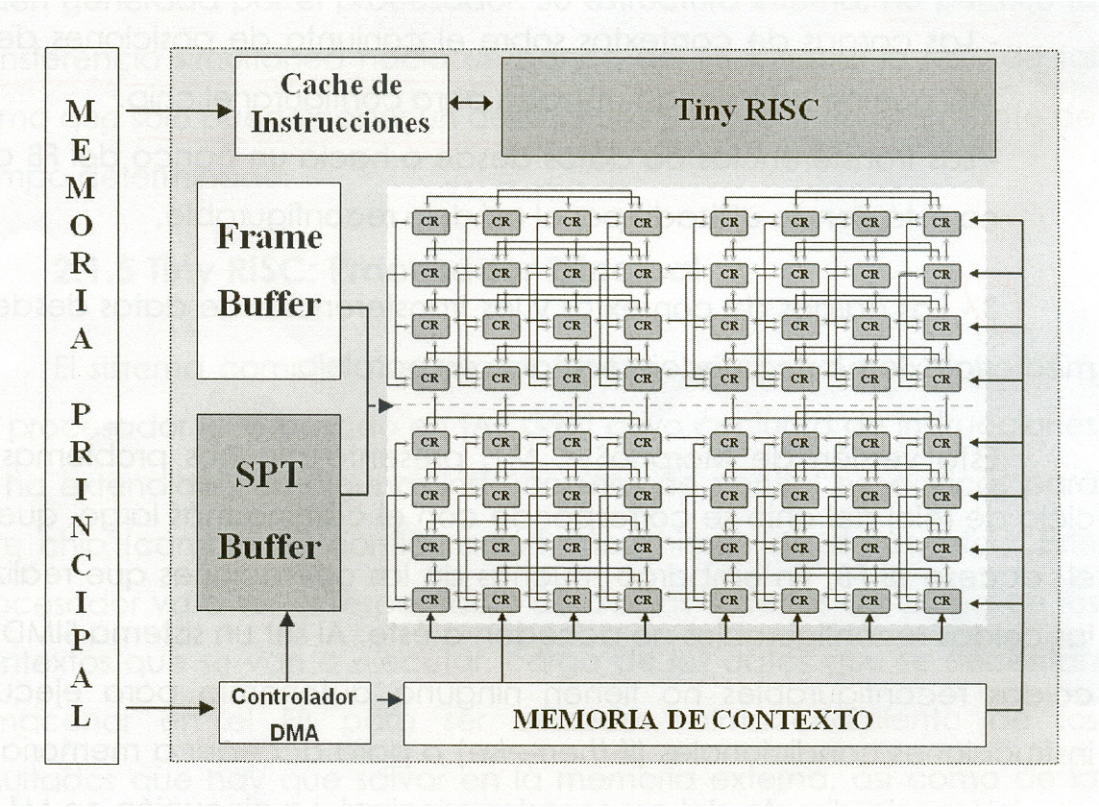

Figura 48. Diseño Morphosys M2

4.3.1 **Ejecución a diferentes velocidades**

Como ya se dijo antes, el ciclo de reloj de M1 venía dado por la transferencia desde el FB hasta las celdas, más el tiempo máximo de ejecución. Sin embargo, el estudio las aplicaciones que queremos ejecutar en Morphosys nos revela que hay más de un 50% de configuraciones que solo referencia a datos de las propias celdas, lo que implicaría una gran perdida de tiempo con el ciclo de reloj que tiene el chip de M1, podemos poner como ejemplo nuestro propio proyecto, ya que los únicos accesos a memoria que se hacen son pedir los datos de los vértices de los triángulos que vamos a renderizar, y el resto de las instrucciones no vuelven a acceder a memoria de datos (frame buffer) por lo que perderíamos muchísimo tiempo teniendo un ciclo de reloj tan grande . Lo que se hace en la nueva versión es aprovechar que el compilador nativo de Morphosys codifica de forma diferente las instrucciones dependiendo si acceden a datos internos o al FB, por lo que el ciclo de reloj que usa M2 va a ser simplemente el tiempo máximo de ejecución de la configuración. Cuando haya una configuración que necesite acceder al FB, su tiempo se escala según el nuevo ciclo de reloj, en particular, si el Tiny Risc detecta una instrucción con acceso al FB en M2 deja pasar 2 ciclos de reloj antes de seguir con la siguiente instrucción.

4.3.2 Mejora de transferencias entre FB y las memorias internas de RCs

Las aplicaciones multimedia trabajan con muchos datos, por lo que las transferencias desde memoria externa a internan llevan un consumo de hasta el 80% del consumo total, por lo que disminuyendo estas trasferencias, bajará mucho el consumo del chip.

El problema real en M1 es que la memoria interna de las celdas (CR-RAM) no puede guardar datos propios, entendiendo esto como "datos intermedios o resultados" que solo son producidos y utilizados por cada celda de manera independiente respecto a las otras, lo que hace que el consumo sea muy elevado, por lo que si podemos almacenar datos propios en la CR-RAM reduciremos el consumo al no tener que acceder al FB para usar estos datos, puesto que las memorias internas tienen un menor tiempo de acceso y menor consumo de potencia al ser éstas más pequeñas que el FB.

Para poder realiza esto, M2 se implementa de manera que las transferencias desde el FB a las CR-RAM se realicen también por vía DMA, añadiendo hardware nuevo como vemos en la figura 56. Se añade un contador carga paralelo y un multiplexor. Lo que se hace es que la dirección de carga, que está en uno de los registros de la celda, se guarde en el contador y que se incremente en uno según se lo comunique el controlador DMA. Las operaciones de lectura o escritura también las controla el DMA que selecciona la columna o fila sobre la que se efectuará la operación. Esto no hace solo que se reduzca el tiempo de ejecución, sino que también se reduzcan el número de configuraciones y de instrucciones del procesador RISC.

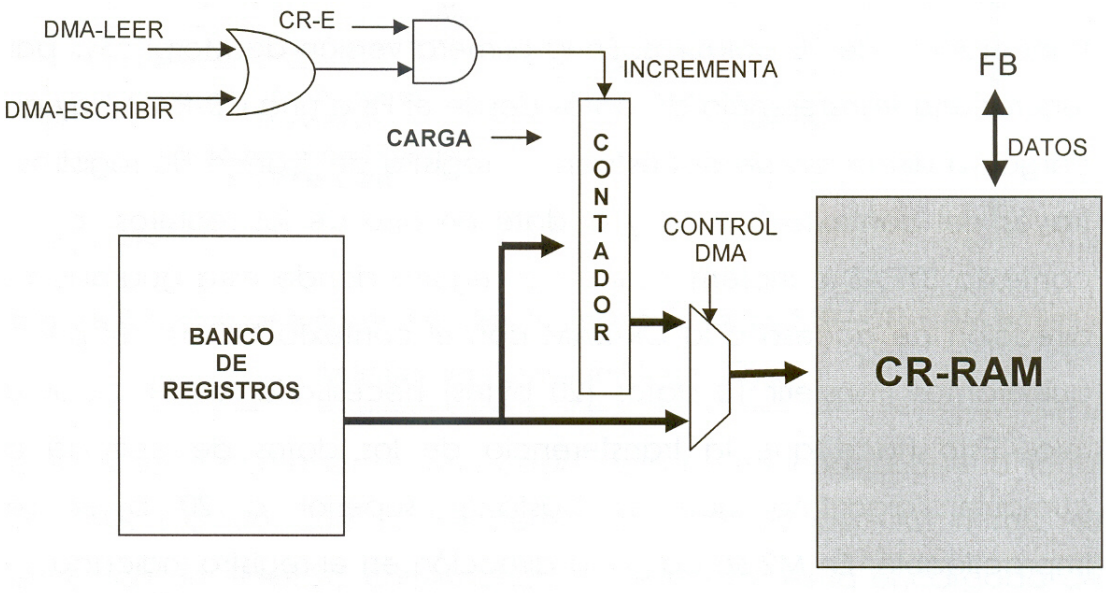

Figura 49. Nuevo hardware añadido a las arquitecura M1

4.3.3 Mejora de la autonomía de las RC

En la versión M1 se podían realizar saltos globales, en los que era el procesador RISC el que tomaba la decisión de que salto hacer, sin embargo, no se admitían los saltos locales, en los que es cada celda la que decide que instrucción ejecutar.

En M2 se han creado nuevas instrucciones y una mascara especial para que cada celda pueda tomar la decisión a la hora de realizar un salto. Se han diseñado otros contextos que se ejecutan dependiendo de un flag de la RC, para ejecutar de manera eficiente los saltos condicionales if-then-else de reducido tamaño. Lo que se hace es que dependiendo del flag y del sufijo del contexto, la RC lo ejecutará o se quedará en estado de no operación.

Para los saltos condicionales if-then-else más complicados, se usan los pseudo-branches (nuevos contextos creados) para emular la ejecución de estos saltos sin necesidad de que el procesador RISC intervenga, además se ha añadido un registro de un bit a las celdas, que indica si la celda está operando o no (despierta o dormida). Lo que se hace, de manera resumida, es realizar una instrucción pseudo-branch (*PBRAN Condicion Etiqueta*) con su condición correspondiente, y en función del resultado de esto las celdas que tomen el salto ponen el bit del registro nombrado anteriormente a 1 (despierto) mientras que las que no tomen el salto, pondrán ese bit a 0 (dormido).

4.3.4 Cambio en la jerarquía de memoria

El problema que se daba en M1 era cuando una aplicación no accedía a sus datos de manera consecutiva, sino que lo hacía de manera aleatoria, lo que se hace en M2 es añadir una memoria adicional al FB que se encargue de los accesos a datos de manera aleatoria. Para su diseño se tiene en cuenta el tipo de aplicación que provoca este problema y en función de esa aplicación esta memoria adicional tendrá una forma u otra, en el caso de las aplicaciones que imágenes 3D, esta nueva memoria es el STP Buffer. En este caso el STP buffer estará formado por 8 bancos, que almacena los datos para cada una de las 8 filas o las 8 columnas. Se puede acceder a todos los bancos o a cada uno de manera individual, decidiendo las RC que datos necesitan, de esta manera se consigue un acceso a datos que están en direcciones no consecutivas.

4.3.5 Conclusión

Hemos visto hasta aquí las características más relevantes de Morphosys, de cada una de sus partes y de su funcionamiento, necesario para a continuación poder realizar nuestro proyecto en esta arquitectura reconfigurable. Como no disponemos de chip, usaremos un simulador del mismo. Este simulador usa el modelo de Morphosys M1, con algunas mejoras, como la mejora de la autonomía de las celdas. Sin embargo para nuestra aplicación en particular, tenemos que añadir una memoria adicional (como el mencionado anteriormente STP Buffer). Ésta es el Z-Buffer que describimos a continuación.

5 Algoritmo de renderización desarrollado

En esta sección explicaremos los pasos que hemos seguido para conseguir renderizar imágenes, creando un renderizador que posteriormente traduciremos a código de MorphoSys. Para ello, describiremos los programas utilizados en detalle, la forma de obtener las imágenes, y las técnicas de renderización utilizadas para obtener el resultado. En la sección posterior, dedicada a MorphoSys, se explicará el enlace entre este renderizador y el código generado para MorphoSys.

5.1 Introducción

Para conseguir imágenes renderizadas a partir de nuestro propio código, se tuvo que diseñar distintas aplicaciones que realizaban los pasos necesarios para generar dichas aplicaciones. En primer lugar se encuentra la aplicación “Renderizador”, la cual genera una matriz de píxels con una imagen renderizada, a partir de una imagen sin renderizar definida según nuestro propio formato, que se introduce como entrada del programa. Para obtener esta imagen sin renderizar codificada en un formato propio, utilizamos dos aplicaciones:

Por un lado, desarrollamos un editor de triángulos con el que dibujábamos triángulos simples en la pantalla del ordenador para renderizarlos después e ir depurando nuestro renderizador.

Por otra parte, también creamos un conversor de imágenes de un formato ya creado (PLY) a nuestro formato. Además, dado que algunas de estas imágenes estaban codificadas en binario, tuvimos que crear una aplicación que interpretara los datos en binario, y los pasara a una imagen PLY estándar.

Este apartado tiene como objetivo explicar el funcionamiento y las características generales de cada una de estas aplicaciones, para mostrar la secuencia de pasos desde una imagen estándar sin renderizar a la matriz de píxels con la imagen renderizada.

5.2 Renderizador

Esta aplicación nos permite obtener una matriz de píxels con la imagen, renderizando los archivos de entrada con nuestro propio formato que introduzcamos. Esta parte es la que emularemos en MorphoSys, objetivo de nuestro proyecto, para disminuir el tiempo de renderización de cada imagen.

Para desarrollar el renderizador, hemos escogido entre las distintas formas de implementación existentes, para hacer el test de profundidad, para añadir iluminación a la imagen, o para procesar cada píxel de la pantalla. A continuación describimos el funcionamiento y las características más importantes de la aplicación.

5.2.1 Funcionamiento de la aplicación

Para renderizar una imagen, tan solo es necesario introducir dicha imagen, en nuestro formato, y en la misma carpeta que el ejecutable del programa. Debemos renombrar la imagen con un nombre aceptado por el renderizador, para que éste comience a extraer los datos de la imagen. El renderizador creará una lista de triángulos, donde cada uno tendrá la información, por cada vértice, del vector normal al mismo, y del color RGB, con las constantes de reflexión de la luz.

La aplicación renderizará los triángulos uno a uno, mostrando en pantalla la imagen cuando se haya acabado con todos los triángulos. La renderización es un método de producción de imágenes que requiere mucho proceso de cálculo, además de gran cantidad de memoria, tanto para almacenar los triángulos y sus propiedades, como la matriz de píxels, o el Z-Buffer. Por ello, existe el riesgo de ralentización en el computador debido a la falta de memoria RAM. En concreto, las pruebas realizadas con un Pentium III a 600 Mhz de velocidad y con 256 MB de RAM muestran una ralentización en la ejecución del renderizador debido al uso del 100% de memoria RAM, llegando en ocasiones a lanzar excepciones debido a que no disponía de más memoria, con imágenes muy grandes que requerían el procesamiento de varios millones de triángulos.

Para solventar este problema, decidimos realizar optimizaciones en el renderizador, almacenando cada vez sólo un conjunto de triángulos en lugar de procesarlos todos del archivo de entrada. Al ser los triángulos totalmente independientes unos de otros, con respecto al procesamiento que requiere cada uno, esta optimización resulta viable. De esta forma, al renderizar un conjunto de triángulos, se eliminan de la memoria y se lee otro conjunto nuevo, hasta renderizar todos los triángulos de la imagen.

En la siguiente figura se muestra en pseudocódigo esta optimización:

```
Lista_triangulos = Archivo_entrada.leer_triangulos(ALL);
renderizar (Lista_triangulos);
```

(A)

```
Lista_triangulos = Archivo_entrada.leer_triangulos(NUM_TRIANGULOS);

mientras ( not ( Archivo_entrada.vacio() ) ) hacer

        renderizar (Lista_triangulos);
        eliminar (Lista_triangulos);
        Lista_triangulos = Archivo_entrada.leer_triangulos(NUM_TRIANGULOS);

fmientras
```

(B)

Figura 50. (A) Pseudocódigo antes de la optimización. (B) Después de la optimización

Como se ve en la figura 50 (A), el programa lee del archivo de entrada todos los triángulos (ALL) de una vez, llamando después a la función de renderizado, a la cual se

le pasa como parámetro la lista de triángulos. En la figura 50 (B), una vez hecha la optimización, se lee solo un cierto número de triángulos (NUM_TRIANGULOS). Mediante un bucle, renderizamos esos triángulos, los eliminamos al terminar de renderizarlos, y leemos de nuevo otro conjunto de triángulos hasta que llegamos al final del archivo de imagen.

El resultado a raíz de esta optimización fue muy satisfactorio, dado que aunque sigue ocupando cierta cantidad de memoria RAM del computador, permite que éste no se colapse, y renderice correctamente cualquier imagen.

5.2.2 Algoritmo de procesamiento de triángulos

Para simplificar los cálculos de la proyección del píxel, la iluminación, test de profundidad, usamos un algoritmo de scan-line de píxels. Este algoritmo procesa los píxels de cada triángulo en líneas horizontales de píxels, que cubren todo el triángulo, recorriéndolas de izquierda a derecha. Para ello, primero realiza una acotación de las alturas mínima y máxima del triángulo, que coinciden con la primera y la última línea de escaneado. Así mismo, para cada línea, acota su anchura para saber cuál es el primer y último píxel del triángulo en dicha línea. Una vez sabemos las coordenadas de cada píxel, podemos tratarlo. Sin embargo, este proceso no podremos realizarlo si no tenemos ordenados los vértices del triángulo por altura, ya que no podríamos establecer las cotas, por lo que el paso anterior a este algoritmo consiste en ordenar los vértices del triángulo a renderizar. Podemos ver un ejemplo de este algoritmo en la siguiente figura:

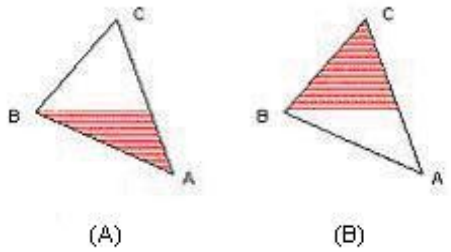

Figura 51. Ejemplo de scan-line sobre un triángulo. (A): Parte inferior. (B): Parte superior

Para poder acotar las alturas y anchuras, el algoritmo se divide en dos partes: primero se renderizan los píxels de la parte inferior del triángulo (desde el vértice con menor altura hasta el intermedio), para renderizar después la parte superior (desde el vértice intermedio hasta el de mayor altura). Sin dividir el algoritmo en dos partes, no sería posible acotar correctamente, dado que la acotación en la parte superior del triángulo depende de los dos vértices superiores, y viceversa.

La razón principal para escoger este algoritmo de scan-line es que tanto el test de profundidad, como la proyección o la iluminación de vértices se pueden desarrollar con algoritmos scan-line, por lo que podemos integrar todas estas técnicas en el algoritmo scan-line original. De esta manera, cada iteración del algoritmo calculará totalmente el píxel, por lo que lo único que debemos hacer para finalizar su procesamiento es almacenarlo en la matriz de píxels que conforma la imagen.

```
para cada Triangulo hacer
      para cada Linea_de_triangulo hacer
            para cada Pixel_de_Linea hacer
                  proyeccion(Pixel);
                  si ( Test_Profundidad(Pixel) ) entonces
                        Iluminación(Pixel);
                  fsi
            fpara
      fpara
fpara
```

Figura 52. Pseudocódigo del algoritmo scan-line usado en el renderizador

5.2.3 Algoritmo de proyección

Mediante este algoritmo, determinamos la forma en que proyectaremos la imagen en la pantalla. En este campo, nuestro objetivo inicial era desarrollar un algoritmo de proyección paralela ortogonal, aunque el renderizador está preparado para procesar, como nueva funcionalidad, nuevos tipos de proyecciones, como son la proyección oblicua y perspectiva.

Figura 53. Triángulo creado con nuestro renderizador con proyección ortogonal

Para crear la proyección paralela ortogonal, elegimos la implementación mediante el algoritmo scan-line, el cual podemos implantar en el algoritmo scan-line base de procesamiento de triángulos del renderizador. La elección de este tipo de proyección se hizo por su simplicidad, dado que en principio nos bastaba con una aplicación sencilla que renderizase imágenes básicas, para añadir nuevos tipos de proyección posteriormente. El desarrollo del algoritmo de proyección se basa en calcular las coordenadas (x,y) en la pantalla, mediante scan-line, del punto actual (x,y,z) que estemos renderizando.

5.2.4 Eliminación de partes ocultas

Siguiendo el mismo desarrollo que el resto de la aplicación, decidimos realizar un test de profundidad para comprobar qué puntos quedaban fuera de la imagen, y cuáles no. Dicho test de profundidad lo implementamos mediante un Z_Buffer, del tamaño del puerto de vista, el cual se rellena al recorrer el algoritmo de scan-line de procesamiento de los triángulos. Para cada uno de ellos, calculamos su profundidad mediante interpolación lineal. Con ella, el algoritmo de scan-line la compara con su píxel correspondiente en el Z-Buffer, decidiendo si debemos renderizar o no dicho píxel. En nuestra aplicación, hemos decidido tomar como convenio, que los píxels tengan una profundidad mayor cuanto más cerca estén con respecto a la cámara.

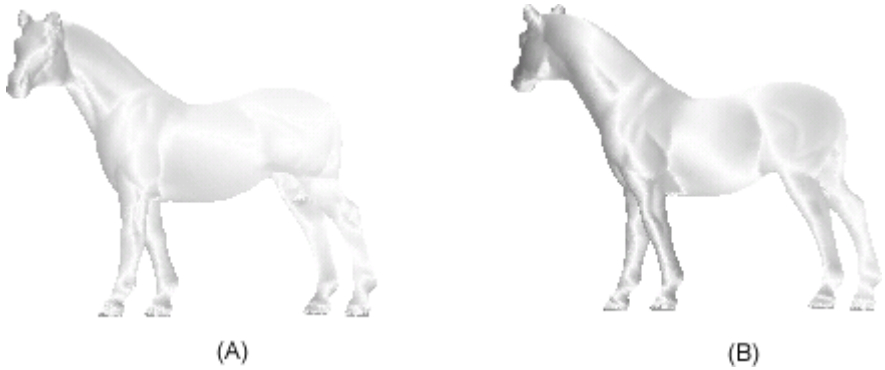

Figura 54. Ejemplo de eliminación de partes ocultas en una imagen renderizada con nuestra aplicación.

En la figura 54 (A), mostramos únicamente la cara izquierda del caballo que no queda oculta por otras partes, desechando los que quedan ocultos, y en consecuencia, no renderizándolos, lo que supone una gran cantidad de cálculos ahorrados. En la figura 54 (B), realizamos el mismo proceso pero con la cara derecha del caballo. En el estado del arte sobre la informática gráfica, en la sección de Producción de Imágenes se encuentra más información acerca de los algoritmos usados aquí, la interpolación lineal y scan-line.

5.2.5 Algoritmo de iluminación

Para renderizar imágenes de cierta calidad, era imprescindible diseñar un módulo que iluminara los objetos, calculando la intensidad de color de cada uno de sus puntos. Para aprovechar las iteraciones realizadas con el algoritmo scan-line base, optamos por escoger un algoritmo de sombreado que tuviera este mismo desarrollo. Los dos algoritmos más efectivos de sombreado que pueden realizarse sobre scan-line son los de Gouraud y Phong, siendo además dos algoritmos que también usaban la técnica de interpolación lineal para calcular la intensidad de luz en cada punto, por lo que podríamos reutilizar dicha técnica, ya usada anteriormente en el cálculo del Z-Buffer.

El algoritmo que usamos finalmente fue el de Gouraud, ya que si bien no produce imágenes de tanta calidad como el algoritmo de Phong, realiza muchos menos cálculos por píxel, y produce imágenes de suficiente calidad cuando éstas contienen muchos polígonos, lo cual consideramos suficiente, teniendo en cuenta que el principal objetivo

del proyecto no es el de diseñar un renderizador muy potente, sino implantarlo en la arquitectura reconfigurable MorphoSys.

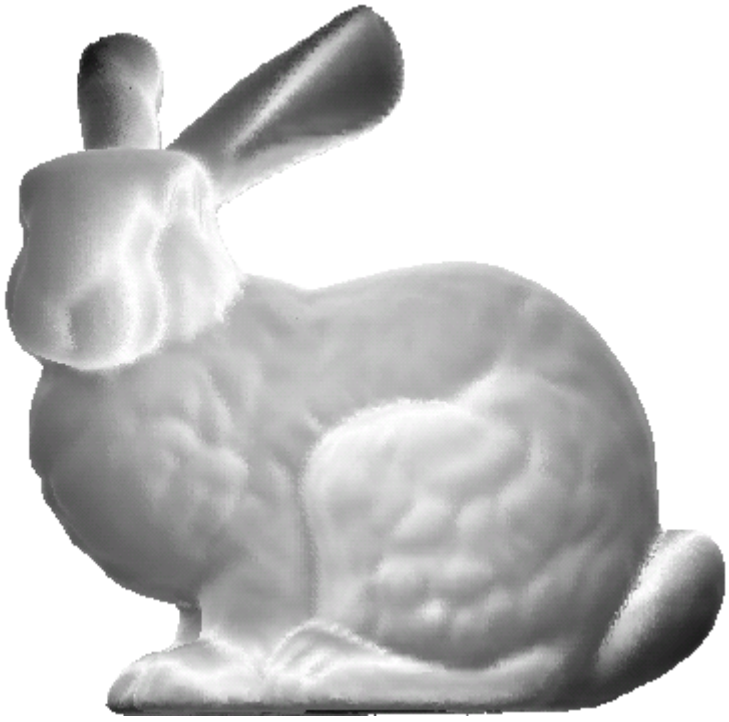

Figura 55. Imagen renderizada con nuestra aplicación usando el sombreado de Gouraud

Como ya comentamos anteriormente, el archivo de entrada con la imagen contiene información sobre las constantes de reflexión de la luz en cada vértice. Para realizar los cálculos de la manera más sencilla, supondremos una luz ambiente global estándar, y un único foco de luz sobre el objeto, además de suponer que las constantes de reflexión tienen el mismo valor. Dependiendo de la imagen, cambiamos las coordenadas del foco de luz, así como la intensidad de cada una de las componentes de la misma, con el objetivo de obtener la imagen lo mejor iluminada posible. Para más información sobre la implementación del algoritmo de Gouraud, consultar la sección de Algoritmos de Sombreado en el Estado del Arte.

6 Z-buffer

Como hemos visto en el punto anterior, para el correcto funcionamiento de la renderización necesitamos tener un Z-Buffer para controlar que píxeles habrá que dibujar, dependiendo de la profundidad de cada uno. Por lo que añadimos esta memoria a la arquitectura de Morphosys en lugar de uno de los bancos del Frame-Buffer según M1, o sustituyendo al STP-Buffer en M2, para ejecutar nuestra aplicación gráfica. En dicha memoria guardamos la información sobre la profundidad de los píxeles en la pantalla. Esta información se almacena en forma de matriz, con tantas filas y columnas como píxeles tenga la pantalla de ancho y alto, respectivamente. Cada píxel procesado se introduce en la posición correspondiente en el z-buffer si su profundidad es menor que los píxeles ya introducidos, es decir, que está más cerca de la pantalla. De esta forma, solo procesaremos los píxeles de los objetos que realmente se vean en la escena, dejando sin procesar las partes ocultas de la misma. Hemos barajados distintas posibilidades a la hora de coger esta memoria. En un principio la idea era enviar los píxeles de 8 en 8, por lo que necesitaríamos que tuviera dos entradas, sin embargo, con la estrategia final elegida (que veremos a continuación) pudimos simplificar esto, de manera que la memoria constará solamente de dos entradas (dos píxeles), que tendremos que asegurar que no sean el mismo antes de enviarlos a estas entradas.

Con esto el diagrama de bloques de la arquitectura Morphosys que utilizamos nosotros es el siguiente:

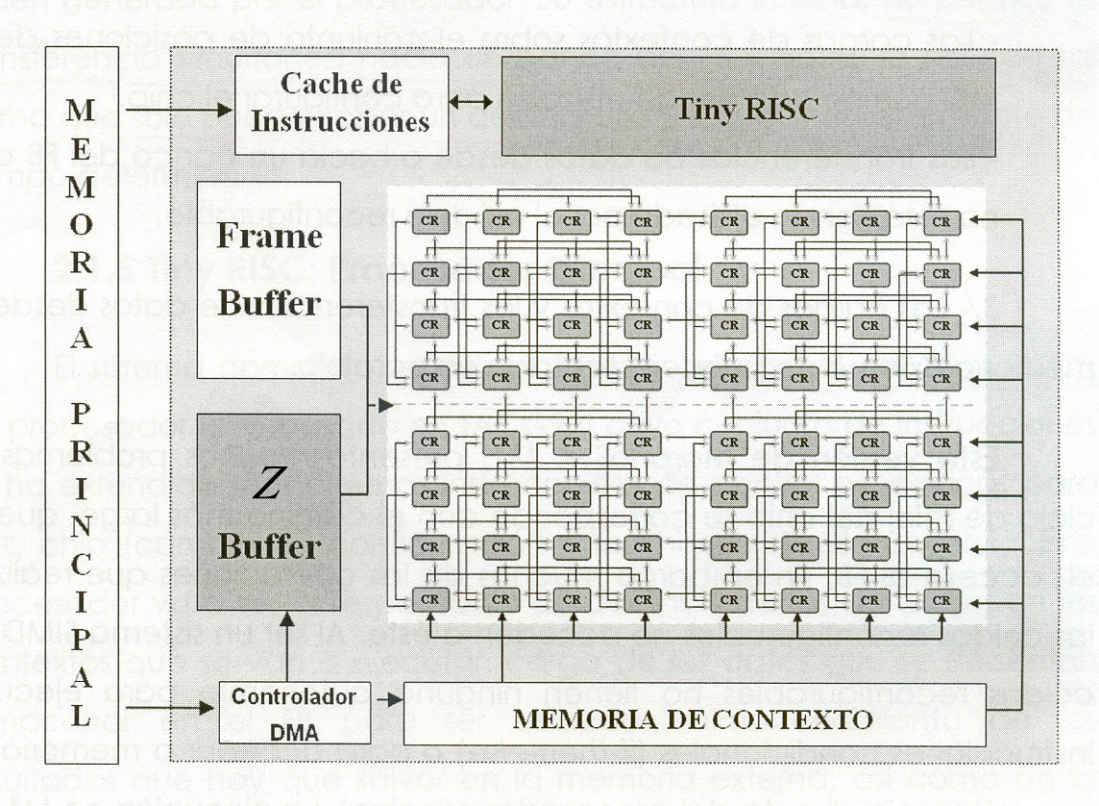

Figura 56. Diseño Morphosys para nuestro proyecto (con Z-Buffer)

7 Programación en Morphosys

Vamos a ver de manera resumida como se programa sobre Morphosys. Como ya se ha mencionado anteriormente, para configurar las celdas de la matriz reconfigurable, es necesario cargar los contextos, en la memoria de contexto, que especifiquen estas configuraciones. En cada contexto se especifica la instrucción que va a ejecutar cada fila o cada columna, veamos un ejemplo de un contexto:

```
set 39,0 BYPAS r0 r1;      configuración para la primera fila
set 39,1 INCMT r1 r1;      segunda fila
set 39,2 INCMT r1 r1;      tercera fila
set 39,3 INCMT r1 r1;      .
set 39,4 INCMT r1 r1;      .
set 39,5 KEEPP;            .
set 39,6 KEEPP;            .
set 39,7 KEEPP;            .
```

en este caso, tenemos el contexto numero 39 en el que todas las celdas de la fila o columna 0 ejecutarán la instrucción BYPAS r0 r1 (pasa el valor de r1 a r0 que son registros de la celda). Las filas 5 y 6 tienen la instrucción KEEPP, que significa que no hacen nada, mientras que las filas 1,2,3 y 4 ejecutan la misma instrucción, que es un incremento de una unidad sobre el registro r1.

Una vez tengamos todos los contextos, habrá que hacer el programa que ejecute el procesador. En nuestro caso los programas deberán comenzar de la siguiente manera:

```
andi $15,$15,0
addi $15,$15,100      #Fijamos la dir. de inicio de los contextos:
ldctx $15,0,0,0,384
addi $14,$14,3000     #Data address=500
andi $5,$5,0          #Frame Buffer address=0
ldfb $14,0,0,$5,28
ldfb $14,1,1,$5,1
```

este código sirve para inicializar tanto los contextos como el frame-buffer.

Los contextos se cargan con la instrucción “ldctxt”, que tiene 5 parámetros. El primero, que indica la dirección de inicio de memoria a partir de la cual cargamos los contextos; el segundo, que indica a partir de que contexto de fila empezaremos; el tercero, que indica si los contextos irán por filas o por columnas, 0 para filas y 1 para columnas; el cuarto, que indica a partir de que contexto vamos a empezar; y quito, que es el número de contextos que vamos a cargar en total contando un contexto por fila o columna, por lo que el total, serán el número de contextos que tengamos multiplicado por las 8 filas o columnas que tenemos.

Los datos se cargan con la instrucción “ldfb”, que tiene 5 parámetros. El primero, que indica la dirección de inicio a partir de la que vamos a cargar los datos en nuestro caso en el simulador que vamos a usar; el segundo y el tercero, que nos indican el bloque y el banco respectivamente sobre el que vamos a cargar los contextos, en nuestra versión de Morphosys como vimos anteriormente solo vamos a tener un banco con dos bloques; el

cuarto, que es un registro con la dirección de memoria del frame buffer de inicio a partir de la cual cargaremos los datos; y el quinto que se refiere al tamaño de los datos medido en palabras (words).

Una vez cargado los contextos y el FB, lo que hay que hacer es ir ejecutando los distintos contextos, para esto tenemos las dos siguientes instrucciones:

cbcast 0,0,0,**2**
sbcb 0,0,0,**1**,0,0,**16**

La primera ejecutaría una vez el contexto 2, en este caso lo ejecutaría para una fila, porque el tercer parámetro es un 0, para hacerlo por columnas sería un 1. La segunda instrucción ejecutaría una vez el contexto 1, también para una fila (los 4 primeros parámetros son iguales que para “cbcast”), los tres últimos parámetros son para indicar un dato del FB, indicando el bloque del banco del FB, el banco del FB y la dirección donde está el dato respectivamente.

8 Implementación en Morphosys

Una vez creado el algoritmo de renderización de aplicaciones gráficas, y estudiada la arquitectura de MorphoSys, debemos implementar este algoritmo en dicho módulo reconfigurable. Para ello, deberemos definir una estrategia óptima para distribuir las tareas de la aplicación en las celdas reconfigurables. El objetivo es diseñar una estrategia que mantenga ocupadas todas las celdas el mayor tiempo posible. Lo ideal sería que todos los triángulos fuesen iguales, no solo en el número de píxeles, sino también en la cantidad de píxeles por línea, sin embargo esto es totalmente imposible, aunque según hemos estudiado los distintos ejemplos, hemos podido observar que todos los triángulos son pequeños y de tamaño similar, aunque que sean iguales totalmente solo sería pura coincidencia.

A la hora de buscar estrategias, es más claro y sencillo partir el código en dos partes: por un lado, la ordenación de los vértices del triángulo que estemos ejecutando en ese momento, y por otro, el desarrollo de los dos bucles que calcularán la ubicación de los píxel que corresponden a ese triángulo, y los pintarán en pantalla. Una vez terminada esta parte el algoritmo se envían los píxeles de cada triángulo al z-buffer.

Hay que saber también que problemas nos vamos a encontrar para cuando queramos hacer la implementación. Por un lado, las celdas de cada fila siempre tendrán que estar siempre ejecutando el mismo contexto, de manera que mientras al menos una siga haciendo cálculos, no se podrá cargar otro contexto. Esto implica que la mejor estrategia lo tendrá que tener en cuenta y buscará que las celdas estén el menor tiempo posible inactivas.

8.1 Ordenación de vértices

La ordenación de vértices se hace en torno a la componente Y de cada uno (pantalla en el plano XY). Esta ordenación se realiza de esta manera debido a que los bucles calculan píxeles línea por línea, en orden ascendente, por lo que hay que hallar el vértice inferior y empezar a pintar a partir de él. Al tener tres vértices por triángulo, el código formado se compone, en su estructura básica, de una sentencia *if* con dos *else*.

```
if (a = = b) then
        c = 5;
else
        d = 5;
endif
```

A

```
if (a = = b) then
        c = 5;
endif
if (a ! = b) then
        d = 5;
endif
```

B

Figura 57 A: Estructura 'if' modificada. B: Estructura modificada para MorphoSys.

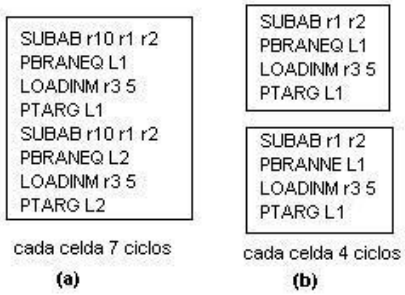

Figura 58 A: Implementación en Morphosys de la estructura 'if' de la figura 57 A
Figura 58 B: Implementación en Morphosys de la estructura modificada, figura 57 B.

La mejor opción, por eficiencia y simplicidad, consiste en ejecutar la ordenación de un triángulo en cada celda. Lo primero, porque no necesitamos partir el código, con lo que no tiene que realizarse ningún paso de parámetros entre celdas, al ser triángulos independientes. Además de está manera conseguimos que todas las celdas hagan lo mismo a la vez simplificando el control de la configuración. Cada celda reconfigurable ejecuta la rama del *if* que satisfaga su condición.

Hemos intentado buscar una manera más eficaz de disminuir el tiempo de espera en la ejecución de una estructura if-then-else. Podemos intentar ejecutar dicha estructura de manera distribuida. Modificamos la estructura del *if* del algoritmo, de manera que cada rama del *if* sea un *if* independiente (habrá tres ramas por triángulo), y se ejecute cada uno en una celda distinta (Fig. 59). Consiguiendo que cada una de las ramas tenga un tiempo de ejecución similar. Para cada triángulo reduciríamos la cantidad de ciclos en torno a 1/3. Sin embargo, esto no ocurre así, ya que en la versión sencilla estaríamos procesando hasta 64 triángulos en paralelo mientras que en está sólo podríamos ejecutar 16 a la vez.

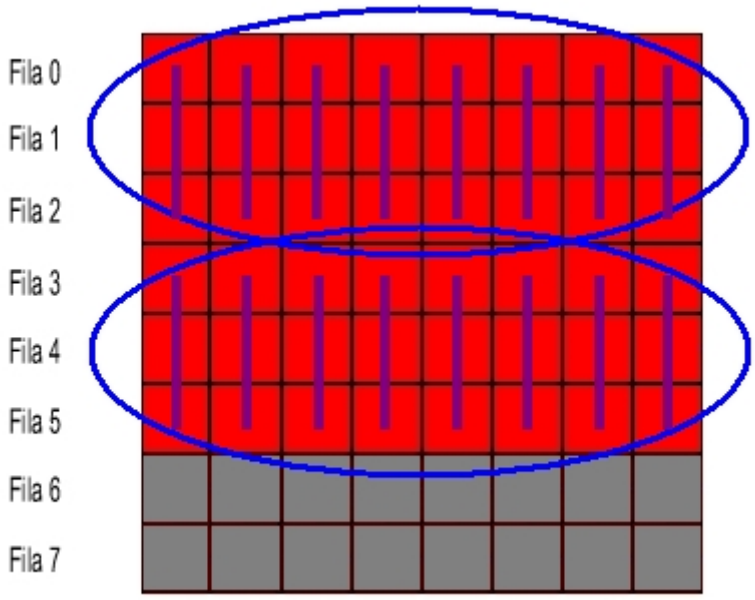

Figura 59. Filas en ejecución ordenando con if's separados

Quedarían 16 celdas libres, ya que por la disposición de contextos en filas o columnas, no pueden tener a la vez una función distinta cada una de ellas, por lo que en el caso de la figura 59, las filas 6 y 7 tienen que estar paradas. Además, una vez hecha la instrucción if en cada celda, éstas deberán comunicarse entre sí, para hacer más comparaciones, ya que cada celda solo compara dos coordenadas y faltaría hacer una última comparación, lo que hace que los ciclos por cada triángulo se reduzcan en menos de *1/3*. En resumen, vemos que de esta manera solo podemos hacer 16 triángulos de una vez, por lo cual para hacer 64, tendrá que repetirse esta operación 4 veces en total, por lo que incluso en el mejor de los casos, donde para cada triángulo tardásemos *1/3* de los ciclos que teníamos con la estrategia de ordenación de un triángulo por celda, en vez de uno entre cada tres (que como ya hemos visto, la mejora es inferior a *1/3*), al tener que realizar la operación 4 veces, el tiempo final sería superior a la estrategia inicial.

De aquí podemos deducir que si no se puede encontrar una manera eficiente de ejecutar las ramas de una sentencia if en paralelo, deberemos usar la mejor estrategia para ejecutar un if secuencialmente. Teniendo en cuenta que la primera estrategia permite que todas las celdas acaben la ordenación a la vez, no habrá celdas que pierdan su tiempo esperando, por lo que es la que elegimos para su implementación.

8.2 Desarrollo de bucles

La primera estrategia examinada consiste en ejecutar todo el desarrollo de los bucles correspondiente a un triángulo en cada celda. Esta opción, además de ser eficiente, hace que la planificación sea muy simple, se ahorra muchos ciclos dedicados a realizar los cálculos necesarios para dividir el código interno de los bucles.

Una manera en principio lógica de dividir el código sin tener que realizar cálculos sobre la disposición de éste sobre el módulo reconfigurable, consiste en repartir los dos bucles de un triángulo en dos celdas. De esta forma, planificaríamos las columnas por bucles, asignando la mitad de las columnas a realizar el primer bucle, y la otra mitad a realizar el segundo. Además, los dos bucles son totalmente independientes, por lo que una vez más, no se pierde tiempo en el paso de parámetros entre celdas a través de la red de interconexión del módulo reconfigurable.

La división del código implica la ejecución de 32 triángulos en vez de 64, ya que dividimos los bucles entre las celdas, aunque se ejecutará en un tiempo ostensiblemente menor que en la primera estrategia. Al tiempo de ejecución de esta segunda estrategia, hay que sumarle un incremento de tiempo derivado de pasar los vértices ordenados a una de las dos celdas que ejecutan sus bucles, suponiendo que la celda que ejecuta la ordenación de vértices va a hacer uno de dichos bucles.

Esta segunda estrategia consume menos tiempo que la primera. Esto se debe a que las celdas no deben esperar a que se termine de procesar el triangulo más grande, sólo al bucle que tarde más. Así, cada fila que haya acabado una de las partes de un triángulo, puede seguir recibiendo otra, ya que el bucle implicado es independiente de la ejecución del bucle que se ejecuta en la columna vecina.

Una vez vistas las opciones de implementación, y la complejidad de cada una, optamos por implementar la que nos ha dado mejores resultados (ver sección resultados), que fue la segunda opción, en la que procesábamos un triángulo por cada dos celdas. Hay que

tener en cuenta que se tendrá que realizar primero la ordenación de vértices, guardar en memoria la información y traerla de nuevo a las celdas cuando ejecutemos los bucles.

Para otro tipo de estrategias que fueron consideradas para intentar reducir el número de ciclos pensamos en ordenaciones anteriores de los triángulos, según su área, y la inclinación de sus lados, para tratar por ejemplo, de ejecutar en cada caso, 64 triángulos lo más parecidos posible, que por lo tanto tarden un número de ciclos bastante parecido, para evitar en la medida de lo posible esperas en determinadas celdas. Este preproceso sería costoso, pero a pesar de todo, si se pudiera hacer en tiempo de compilación, a la hora de ejecutar sobre Morphosys, tendríamos asegurada una mejora en el rendimiento, sin embargo, lo que nos interesa a nosotros es hacer rendering en tiempo real, y cualquiera de estas operaciones son demasiado costosas y penalizarán de tal manera que será imposible obtener rendering en tiempo real. Por todo esto, se han descartado dichas opciones, independientemente de la estrategia que vayamos a utilizar.

8.3 Renderizado de triángulos

Como ya hemos dicho, para hacer la ordenación no tenemos ningún problema, ordenamos los vértices de 64 triángulos a la vez (uno por celda) y los pasamos a memoria. Los problemas a la hora de implementar llegan cuando queremos ejecutar el renderizado de los triángulos en la arquitectura. La distribución va a ser de la siguiente manera, en cada columna vamos a implementar una mitad del triángulo (un bucle), de tal manera, que las dos mitades de un mismo triángulo se ejecutarán en la misma fila y en columnas adyacentes, con lo que en las dos primeras celdas de una fila se realiza un triángulo, en las dos siguientes otro, y así sucesivamente sobre todas las filas de celdas.

Lo primero que hacemos es cargar los vértices de los 32 primeros triángulos desde el Frame Buffer a cada una de las celdas de la siguiente manera:

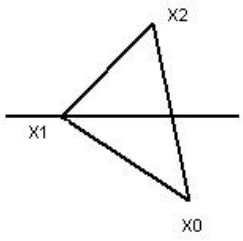

Figura 60. Orden de los píxeles una vez ordenados

A la celda que ejecute la primera mitad del triángulo (primer bucle) se le pasarán los píxeles X0 y X1, mientras que a la celda siguiente, que ejecute la segunda mitad del triángulo (segundo bucle) se le pasarán los píxeles X1 y X2. Cada uno de sus píxeles con sus datos necesarios de profundidad y color.

Una vez que tengamos los datos en cada celda, cargaremos el siguiente contexto, comenzando a realizar la renderización de los triángulos. Para implementarlo de la mejor manera posible, hemos hecho estimaciones, pasando el código de los bucles de nuestro algoritmo en C++ a instrucciones de Morphosys. Los datos obtenidos para renderizar cada fila de píxeles de los triángulos nos muestran que para calcular los píxeles de los extremos de cada fila son necesario 40 ciclos, y que para cada uno de los

píxeles intermedios necesitaremos 18 ciclos. Hicimos este cálculo porque nos dimos cuenta de que si ejecutamos a la vez todos los triángulos en cada fila, mientras todas las celdas estén trabajando, éstas obtendrán al mismo tiempo sus píxeles correspondientes, por lo que se nos crearía un problema a la hora de controlar el Z-Buffer y ver si el píxel tendrá que dibujarse o no.

La complicación está en que tendríamos 64 píxeles obtenidos en el mismo ciclo, por lo tanto, 64 píxeles para enviar al Z-Buffer, y el Z-Buffer, en el mejor de los casos podría contar con 8 entradas de píxeles, por lo que tendríamos que enviar todos los datos en 8 ciclos (cada ciclo los datos de una fila), perdiendo eficiencia cada vez que calculamos 64 píxeles en las celdas. Sin embargo no solo es ese el problema, ya que 8 ciclos tampoco sería excesiva penalización, sino que los píxeles que se le envían al Z-Buffer en cada ciclo hay que asegurarse que no sean el mismo punto, por lo que antes de enviar los píxeles habría que comprobar entre cada 8 si alguno de ellos es el mismo, y elegir solo el adecuado. Para ejecutar esta parte de enviar los píxeles de 8 en 8, también hemos pensado varias opciones.

-Pensamos en tener una fila de celdas que solo se dedique a enviar píxeles al Z-Buffer (Fig. 61), la idea es que según se calculen los píxeles en el resto de las filas, estos sean pasados a la fila que hemos reservado, para que ésta se encargue de enviarlos al Z-Buffer, mientras que las filas se ponen a calcular sus siguientes píxeles.

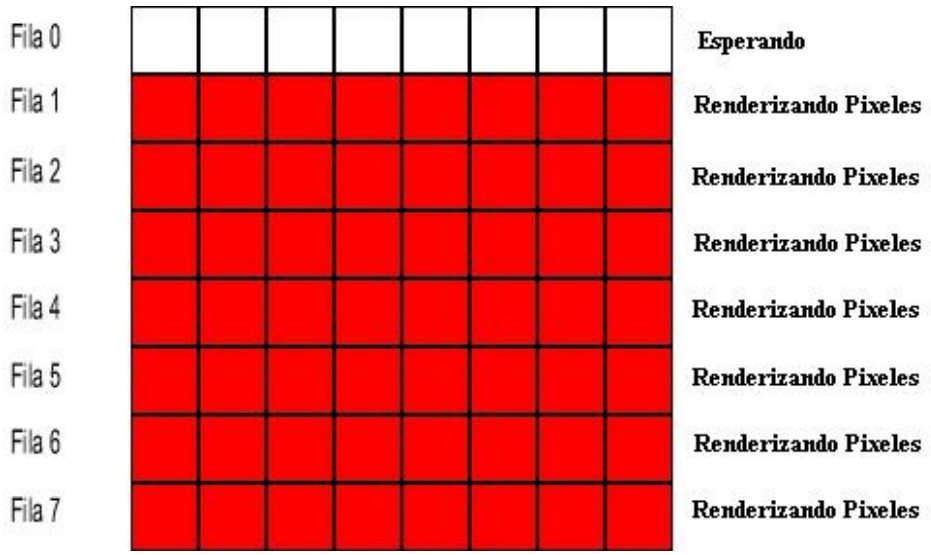

Figura 61. Primera fila esperando píxeles de las siguientes para enviar al Z-Buffer

Esta estrategia resulta impracticable, pues en un principio obtendríamos al mismo tiempo 56 píxeles (7 filas x 8 píxeles cada una). A lo sumo, teniendo 8 celdas en la fila, podríamos pasarle cada vez 8 píxeles de filas inferiores, por lo que mientras el resto de las filas estarían a la espera de que se envíen los píxeles a la primera fila y a su vez, que ésta los envíe al Z-Buffer.

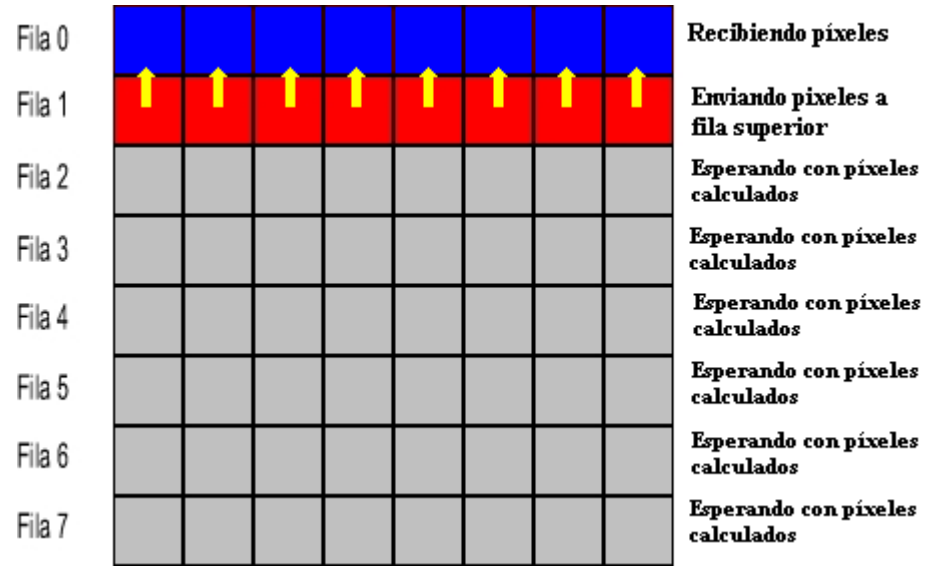

Figura 62. Segunda fila envía píxeles a la primera mientras el resto esperan

Sin embargo los píxeles de la primera fila no se pueden llevar directamente al Z-Buffer, sino que antes habría que comprobar si son iguales entre ellos, haciendo que las 7 filas que calculan píxeles estén inactivas más tiempo aún, esto, junto con que llevaría un cálculo enorme comprobar que píxeles hay que mandar y cuales no haría que no mereciera la pena intentar implementar un algoritmo de este tipo en Morphosys, pues como hemos visto, solo se tarda 18 ciclos en calcular el siguiente píxel mientras que hacer tantas comparaciones antes de enviarlo al Z-Buffer ocupará una cantidad de ciclos muy superior a 18, por lo que las celdas estarán paradas mucho más tiempo.

-Viendo que la primera idea no nos daba los resultados deseados, intentamos cambiar la forma de realización de esta tarea. Ahora nos basamos en la idea de ejecutar cada bucle de un triángulo en una columna distinta, de manera que cuando enviamos los píxeles por filas sabemos que en una misma fila el píxel de la primera y de la segunda columna son siempre distintos (son del mismo triángulo), al igual que los de la tercera y cuarta, quinta y sexta, y por último séptima y octava.

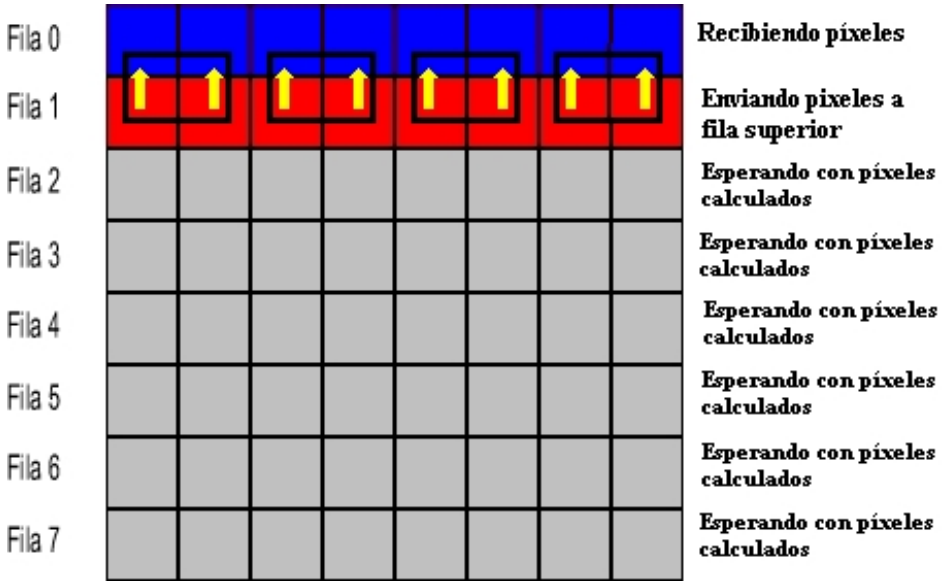

Figura 63. Igual que antes, pero recuadrados los píxeles del mismo triángulo

Esto haría que no habría que comparar todos los píxeles con todos, pero el resultado sigue siendo prácticamente tan ineficiente como antes, porque aún así con muchas comparaciones, aunque nos ahorremos algunas, ya que por cada píxel hay que comparar el menos una de sus coordenadas, con las del resto y en el peor de los casos, habría que comparar las 3 coordenadas, esto para todos los píxeles hace que no podamos implementarlo así tampoco.

- Otra idea pensada fue algún tipo de ordenación una vez calculados los píxeles y enviados a la fila que se encarga de enviarlos al Z-Buffer por columnas, descartando si hubiera alguno igual, pero todas las opciones en las que hay que hacer alguna operación entre píxeles, aunque solo sean de unas pocas celdas, las hemos descartado.

- Hasta aquí hemos visto opciones que a la hora de llevarlas a la práctica resultarían inviables. Una posible mejora, sería que todas las celdas trabajen renderizando píxeles, y una vez que estén todos calculados, se envíen de uno en uno (ver Fig. 64). Para esto el Z-Buffer que necesitaríamos sería mucho más sencillo, pues sólo tendría una puerta de entrada. Esto nos provocaría 64 ciclos de retardo, desde que empieza la primera hasta que termina la última celda de enviar su píxel, sin embargo así nos ahorraríamos tener que hacer comparaciones que hacen que perdamos mucho más de 64 ciclos.

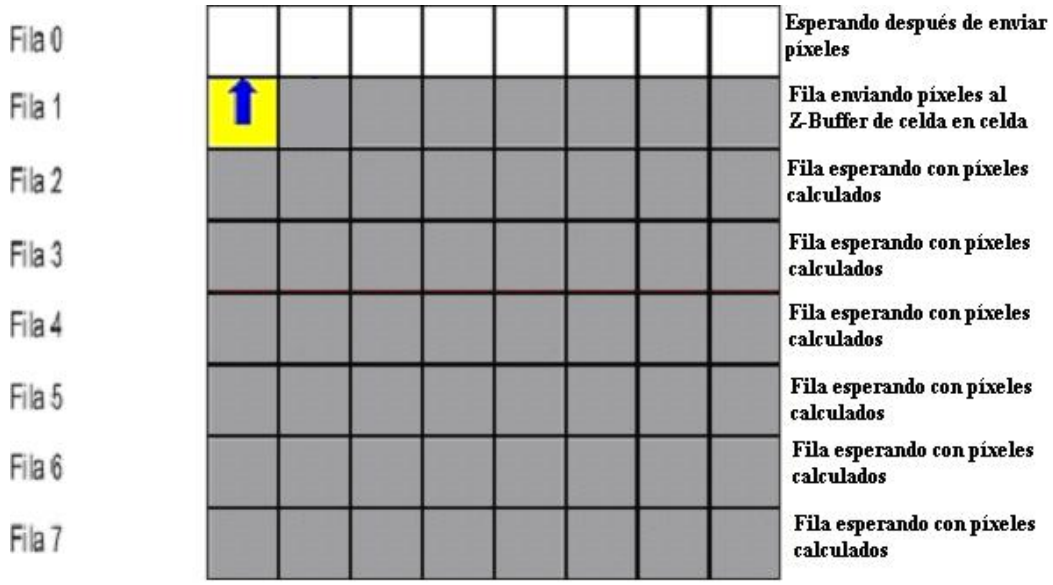

Figura 64. Enviando 64 píxeles de 1 en 1

- Una solución que mejoraría lo anterior de manera considerable es usar un Z-Buffer con solo 2 entradas, y una vez calculados los 64 píxeles, enviarlos de dos en dos sin usar ninguna fila específicamente para ello, sino según se calculan (ver Fig. 65), de manera que cada vez que enviemos dos píxeles, estos serán del mismo triángulo, uno de cada bucle (es decir, uno de cada parte del mismo triángulo), así nos aseguramos que no son el mismo píxel y no habría que hacer ninguna comprobación más, por lo que el retardo real sería de 32 ciclos cada vez que calculamos los 64 píxeles, reduciéndose mucho en comparación con las ideas anteriores y haciendo ahora posible e interesante la implementación en Morphosys, sin embargo, seguiríamos perdiendo muchos ciclos, ya que casi todos los píxeles tardarán sólo 18 ciclos en ejecutarse (menos los extremos, 40).

Figura 65. Enviando los 64 píxeles de 2 en 2

- La única forma que hemos encontrado mejor que la anterior es pensando más en las características de la arquitectura Morphosys. La idea es similar a la anterior, en la que el retardo era de 32 ciclos. Veámosla más detenidamente. Pensamos enviar los píxeles por filas, de dos en dos (como en el caso anterior), ya que así estamos seguros de que esos dos píxeles que enviamos en cada momento son siempre distintos. Esto nos llevaría a gastar 4 ciclos para enviar los 8 píxeles de cada fila. Hasta aquí estamos como en la opción anterior.

Mientras se están enviando los 8 píxeles, están todas las celdas paradas, menos las de la primera fila, y así sucesivamente con cada fila, hasta enviar todos, en total los 32 ciclos mencionados antes. Para solucionar esto e intentar mantener a todas las celdas el mayor tiempo posible en funcionamiento hemos adoptado la medida de retrasar 4 ciclos el comienzo de la ejecución de cada fila respecto a la anterior. Esto es, la segunda fila empezará 4 ciclos después que la primera, la tercera empezará 4 ciclos después que la segunda y por lo tanto 8 ciclos después que la primera y así hasta la última fila.

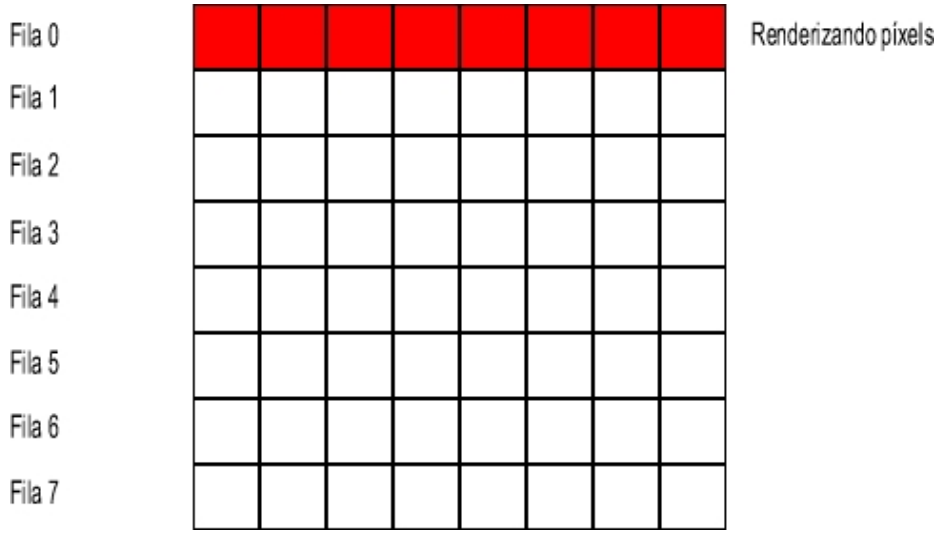

Figura 66. Matriz donde solo ha comenzado a ejecutar la primera fila

En la figura 66 observamos que al empezar, es la primera fila la única que esta renderizando en los primeros 4 ciclos mientras las demás aun no han empezado a trabajar. Una vez consumidos los primeros 4 ciclos, empezará a trabajar la siguiente fila (ver Fig. 67).

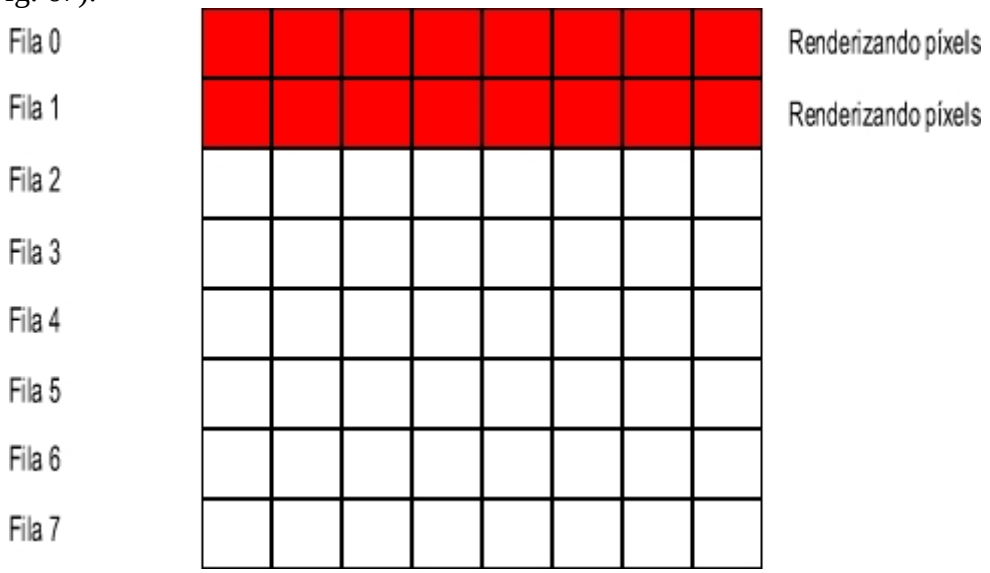

Figura 67. Matriz donde han comenzado ya las dos primeras filas, las demás paradas.

Lo que conseguimos con esto es que cuando la primera fila calcule sus 8 píxeles, a la segunda le falten 4 ciclos para obtenerlos, por lo que mientras la primera fila gasta otros 4 ciclos en enviar los píxeles al Z-Buffer, la segunda fila sigue trabajando y obtiene sus píxeles justo en el momento en el que ya puede enviarlos al Z-Buffer, por lo que no está parada en ningún momento, y así sucesivamente con las siguientes filas. De esta manera, las celdas están paradas por filas, la segunda fila está parada 4 ciclos, la tercera 8 y así hasta la séptima que estará parada 28, pero solo cada vez que se empieza a renderizar una fila de píxeles de un triángulo y no cada vez que calculo 64 píxeles. A partir de aquí, para los píxeles extremos que tardan 40 ciclos en calcularse no habrá ningún problema, todas las filas acabarán renderizando a la vez y será la primera la que obtenga sus píxeles en primer lugar, por lo que los irá enviando mientras el resto siguen con sus cálculos de renderizado (ver Fig. 68)

Figura 68. Matriz con primera fila enviando al Z-Buffer y resto renderizando

El resto de los píxeles tardan en calcularse 18 ciclos, pero como entre la primera fila y la última hay ahora un retardo de 28 ciclos, tendremos que dejar cada fila 10 ciclos inactiva (ejecutando la instrucción del contexto KEEPP), porque sino coincidirían operaciones de envío al Z-Buffer y nos provocaría nuevos errores. De esta manera, el mayor retardo que tendríamos sería de 38 ciclos (28 de retardo + 10 inactiva) para la última fila, pero este retardo sólo se dará al empezar a renderizar el primer píxel de una fila de píxeles de un triángulo, ya que para el resto de la fila, solo estaría parada 10 ciclos. En el caso del primer píxel de las filas, cuando la primera fila acabe de procesar sus píxeles (18 ciclos) las dos últimas aun no habrán empezado (ver Fig. 69)

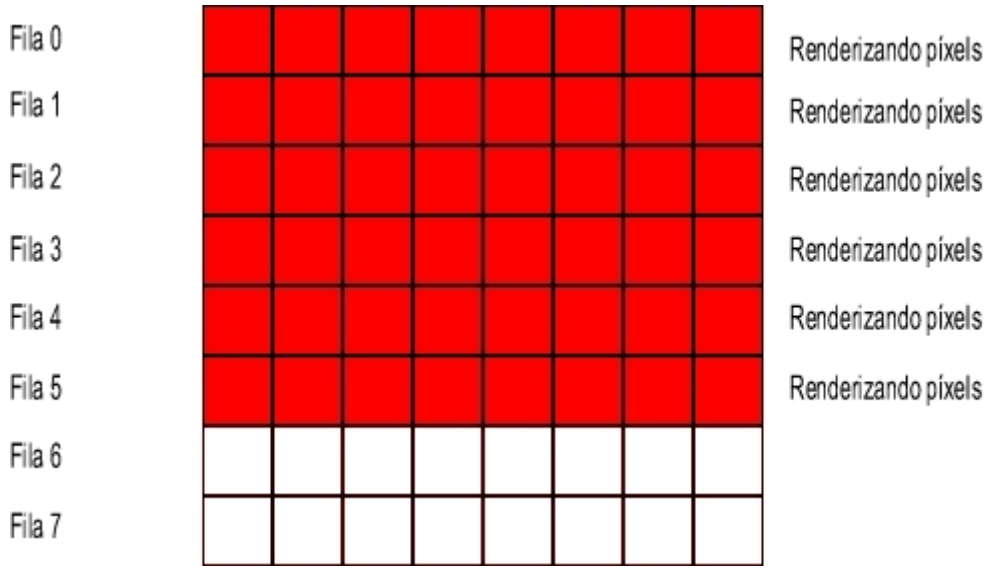

Figura 69. Matriz con las dos últimas filas sin comenzar aún

A partir del primer píxel de cada fila del triángulo, las celdas solo tendrán el retardo mencionado de 10 ciclos, por lo que habrá momentos en los que lo máximo será que tres filas estén inactivas (ver Fig. 70)

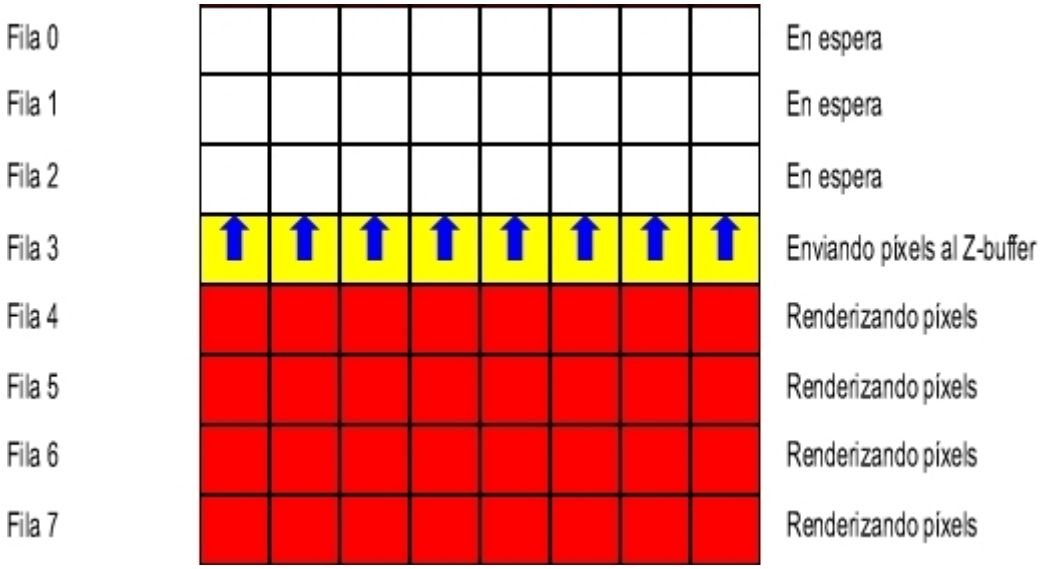

Figura 70. Matriz con las primeras filas en espera

Como se puede observar, está última estrategia es la mejor de las que hemos buscado, pues la anterior tenía a todas las celdas paradas 32 ciclos cada vez que calculaban un píxel cada una. En este último caso, las únicas celdas que superarán esos 32 ciclos serán las dos últimas, que estarán paradas 34 y 38 ciclos respectivamente, mientras que las demás estarán siempre por debajo de 32 ciclos, y como hemos mencionado antes, este retardo máximo solo se da una vez por cada fila de píxeles de los triángulos.

Esta estrategia hace que tengamos más contextos que con las anteriores para poder ejecutarla, sin embargo tampoco es un número muy elevado (hemos obtenido 48 contextos distintos) y hace que tengamos mejores resultados que de cualquier otra manera.

Por último nos surgió una complicación más, ya que cada celda ejecuta un bucle, y cada bucle va fila a fila de píxeles de un triángulo renderizándolos, sin embargo no todas las filas tienen el mismo número de píxeles. Como ya sabemos, en Morphosys todas las celdas de una fila tienen que hacer lo mismo (o todas las de una columna, según se desee trabajar), por lo que si una celda de una misma fila acaba mientras las otras siguen, ésta no puede empezar a calcular el siguiente píxel, y tiene que esperar a que acaben todas, por lo que los ciclos finales para el cálculo de los píxeles de cada una de las filas de los triángulos serán los mismos ciclos que aquella fila que contenga más píxeles.

Lo que hay que hacer es que cada vez que acabemos de hacer un píxel en las celdas y lo llevemos al Z-Buffer, comprobar si ese píxel era el último de la fila (ver Fig. 71).

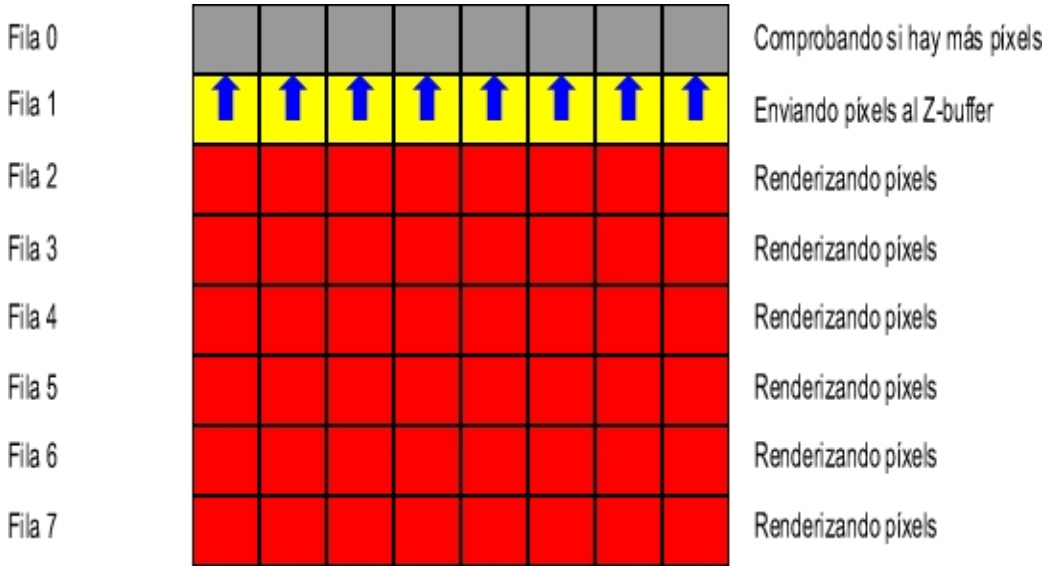

Figura 71. Matriz con todas las filas ejecutando, las 2 primeras con contextos distintos

Esto es simple, pues los píxeles extremos están guardados en dos registros de cada celda. En este caso solo necesitaremos el píxel del extremo derecho (guardado en el registro R6 de cada celda), ya que renderizamos de izquierda a derecha las filas. Con tres instrucciones de Morphosys comparamos si es o no es el último píxel de la fila, y en caso de que lo sea “dormimos” la celda. Con esto lo que se hace es que se sigue

cargando el mismo contexto para la fila determinada, y esa celda lo recibe igual, como si siguiera activa, pero mientras las demás celdas siguen ejecutando normalmente los contextos, esta está parada hasta que no se la despierte (ver Fig. 72).

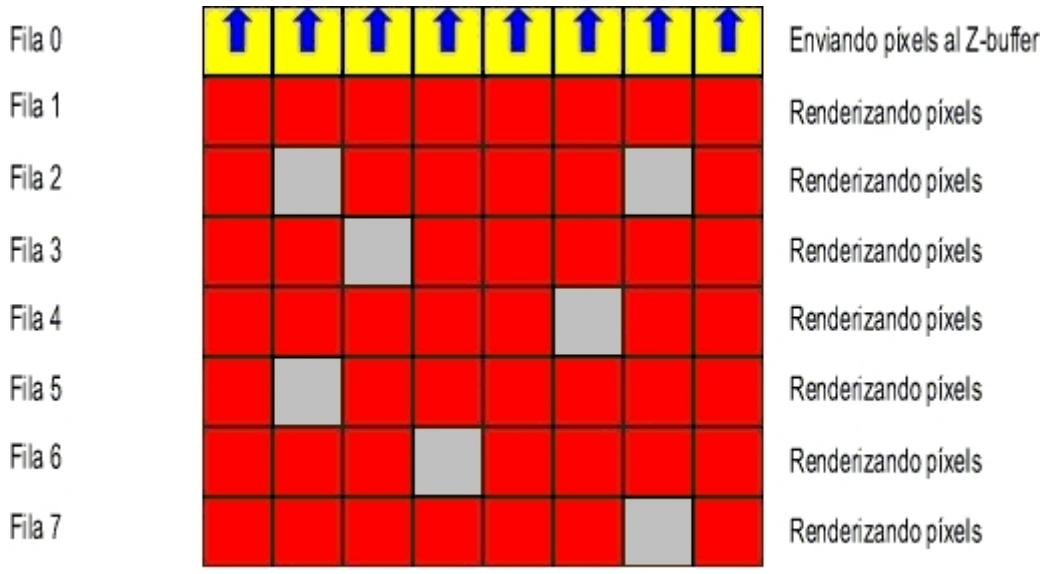

Figura 72. Matriz con todas las filas ejecutando, pero algunas celdas ya dormidas.

Como hemos dicho, para hacer esta última comparación son necesarios otros 3 ciclos (3 instrucciones de un ciclo cada una, ver Fig. 73)

1. **INCMT r11 r11;**
2. **SUBAB r10 r6 r11;**
3. **PBRANEQ L1;**

Figura. 73 Comparación para comprobar si terminamos fila de píxeles

En primer lugar aumentamos la coordenada X del píxel actual, guardado en el registro R11, en una unidad (la Y no la tocamos porque como renderizamos por filas es la misma para todos los píxeles), esto se hace con la instrucción 1. Luego restamos sobre el registro R10, el píxel extremo con el píxel actual (instrucción 2) y por ultimo, comparando el resultado obtenido, si la resta ha dado cero, significa que son el mismo, y dormimos la celda (instrucción 3) sino sigue ejecutándose normalmente. Las celdas estarán dormidas hasta que no se las despierte con una instrucción PTARG Etq, donde Etq es una etiqueta que despertará a aquellas celdas que fueron dormidas con dicha etiqueta.

Como ya mencionamos antes, cada celda tenía que estar 10 ciclos parada después de enviar sus píxeles al Z-Buffer para evitar que se solape con el envío por parte de celdas de otras filas. Como la comparación se hace después del envío al Z-Buffer, hace que tengamos una mejora más, ya que ocupamos 3 de los 10 ciclos que teníamos inactiva la

celda, por lo que ahora solo estará sin hacer nada 7 ciclos, lo que implica que ahora, como mucho, solo 2 filas estén inactivas a la vez (ver Fig. 74).

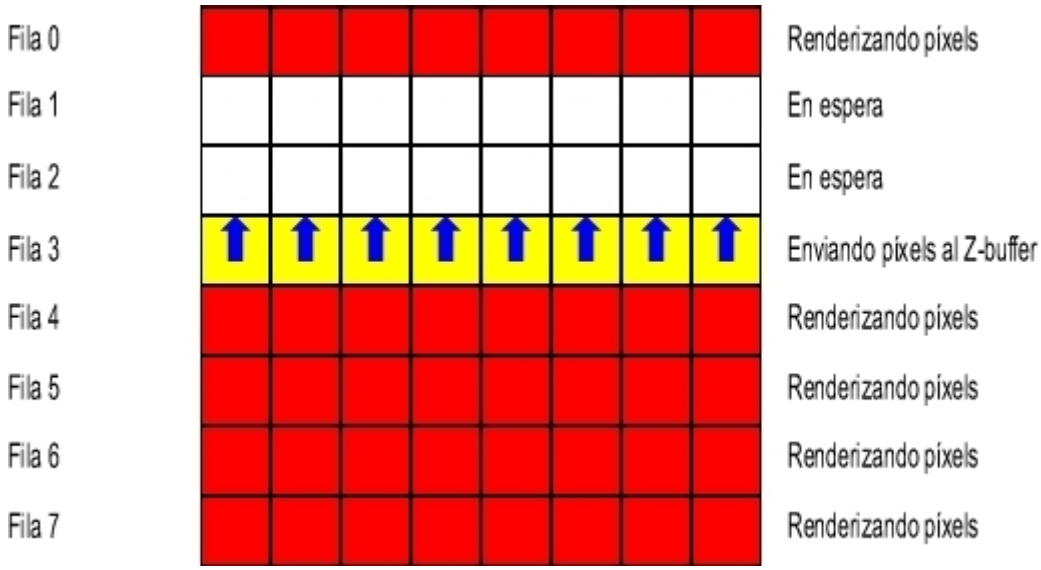

Figura 74. Matriz con dos filas inactivas

Todo esto implica que, hasta que todas las celdas no hayan terminado de calcular sus filas de píxeles, no empezaremos con la siguiente fila, por lo que podemos tener casos en los que muy pocas celdas estén ejecutándose todavía, aunque habiendo estudiado los triángulos de muchas de las imágenes que hemos renderizado observamos que son todos bastante similares, por lo que estos casos peores casi nunca no se darán (ver Fig. 75)

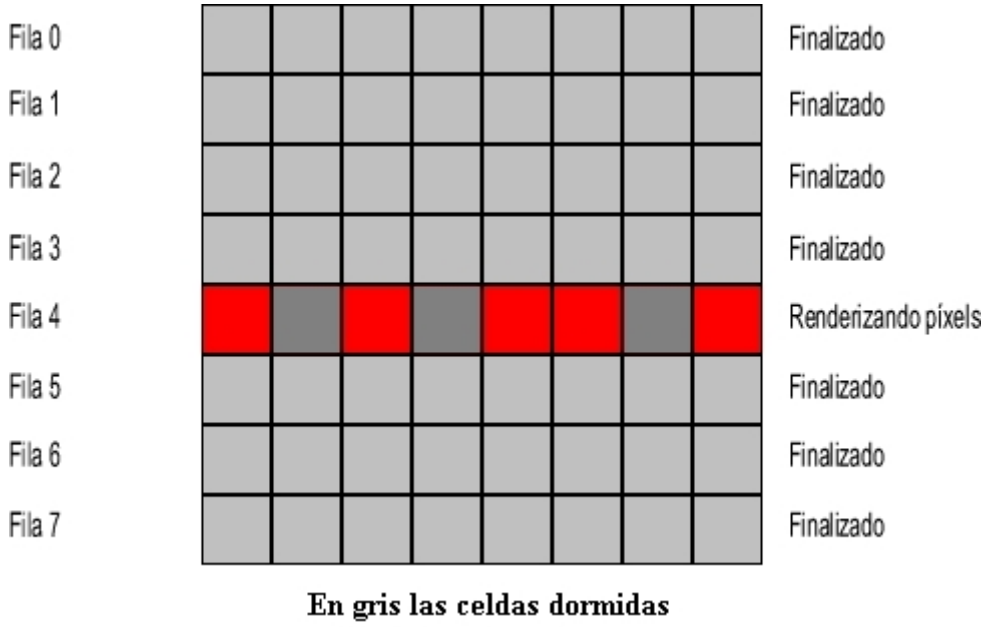

Figura 75. Matriz con algunas celdas de una fila aun trabajando

Aquí tenemos en gris algo más claro todas las celdas que por un lado están dormidas, pero además en sus respectivas filas todas las celdas han terminado de

renderizar. En gris más oscuro son celdas dormidas, pero en cuya fila hay aun celdas renderizando.

Para finalizar, cuando todas las celdas estén dormidas (ver Fig. 76), se detectará y volveremos a comenzar las siguientes filas de los triángulos.

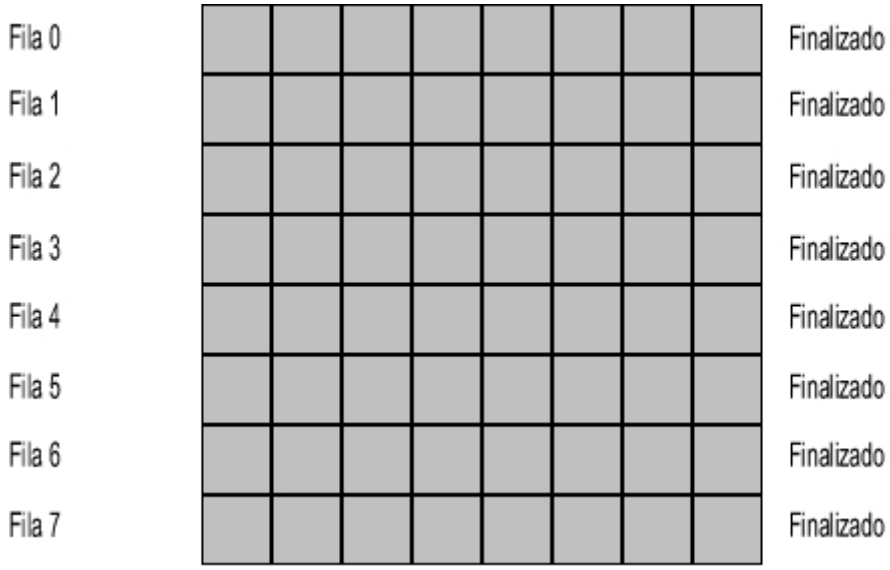

Figura 76. Matriz con todas las celdas finalizadas

9 Simulador de Morphosys

Vamos a describir el simulador de Morphosys utilizado, “MuLate”, sobre el que implementaremos parte de nuestro código. Habrá que ver como generar los programas, como cargar los datos y como ejecutar el programa que cargue nuestros contextos. La siguiente figura muestra el entorno:

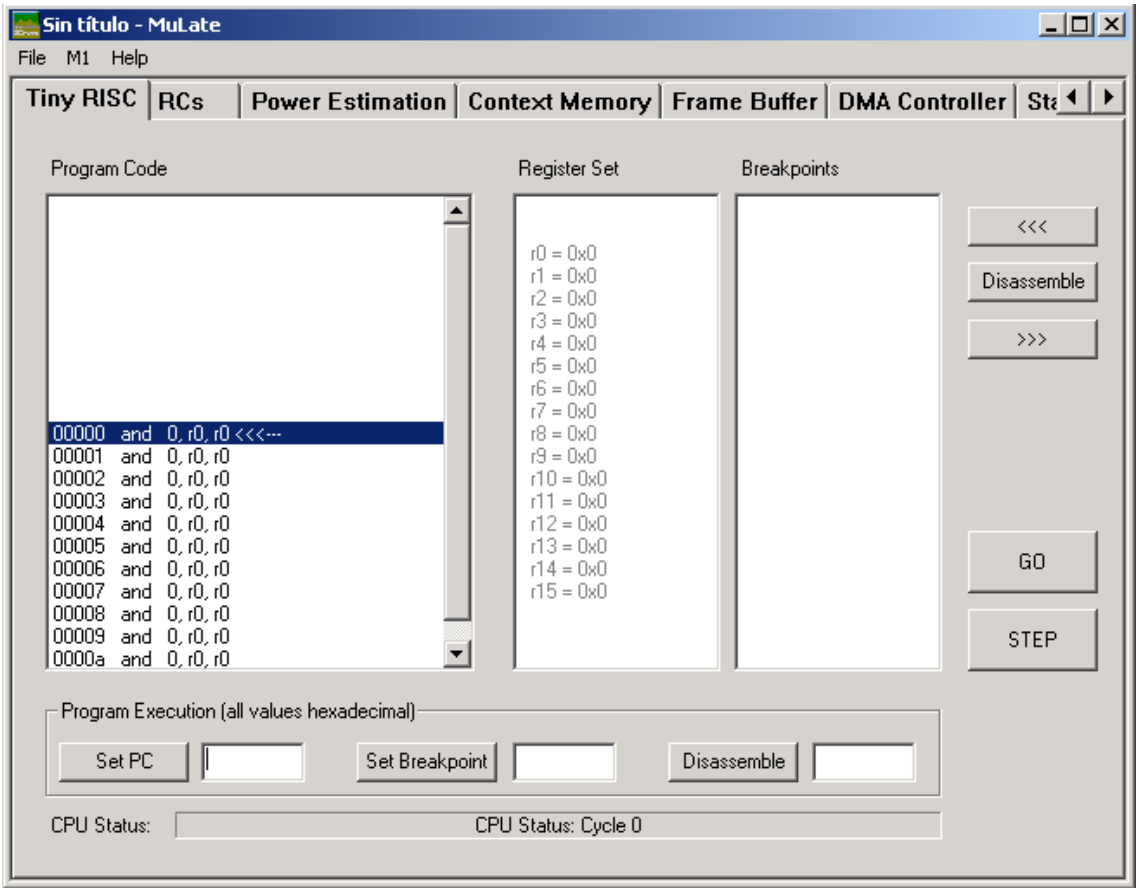

Figura 77. Entorno de simulación de Morphosys

Antes de simular, es necesario tener los archivos de programa y contextos, escritos en formato texto (“txt”), en hexadecimal, para eso el simulador cuenta con distintos programas que hay que ejecutar desde la consola para pasar nuestros archivos de texto con nuestro programa y con nuestros contextos a archivos hexadecimal.

Para los contextos usamos Perl. Ejecutaremos la siguiente línea de comando, en este caso en la carpeta donde tengamos nuestros programas, sino habría que indicar la extensión completa de donde se encuentra el archivo:

C:\CarpetaPrograma>Perl ML.pl ctx_jfa.txt

ML es un programa en perl que se encarga de pasar el archivo indicado (“ctx_jfa.**txt**” en nuestro caso) a un archivo perl, que tendrá el nombre que nosotros indicamos en la última línea del archivo de los contextos mediante la instrucción *writeFile "ctx_jfa.ctx";*. Una vez que hemos conseguido el archivo “ctx_jfa.**ctx**”, hay

que pasarlo a binario. Para eso el simulador cuenta con otro programa, “bin2hex, que usamos de siguiente modo:

C:\CarpetaPrograma> bit2hex ctx_jfa.txt ctx_jfa.hex

con esto ya tenemos nuestros contextos en hexadecimal, listos para cargarlos en el simulador. Ahora nos queda pasar nuestro archivo “prg_jfa.txt” que contiene el programa encargado de ejecutar los contextos, para lo cual utilizamos el programa trasm2002 del simulador de la siguiente manera:

C:\CarpetaPrograma> trasm2002 prg_jfa.txt prg_jfa.hex

Una vez que tenemos todo del modo adecuado para ejecutarlo en el simulador, lo primero que hay que hacer es cargar el programa, los contextos y los datos, en ese orden. Para realizarlo nos vamos a la pestaña “M1”, vamos a la opción “Memory IO” (la primera) y pinchamos en “Memory Load...”. Nos sale, como podemos observar en la siguiente figura (Fig. 77) otra ventana donde se nos pide que introduzcamos el archivo que queremos cargar (ya sean, datos, contextos...) y la dirección de comienzo en hexadecimal (Start Address) sobre la que volcaremos el contenido del archivo.

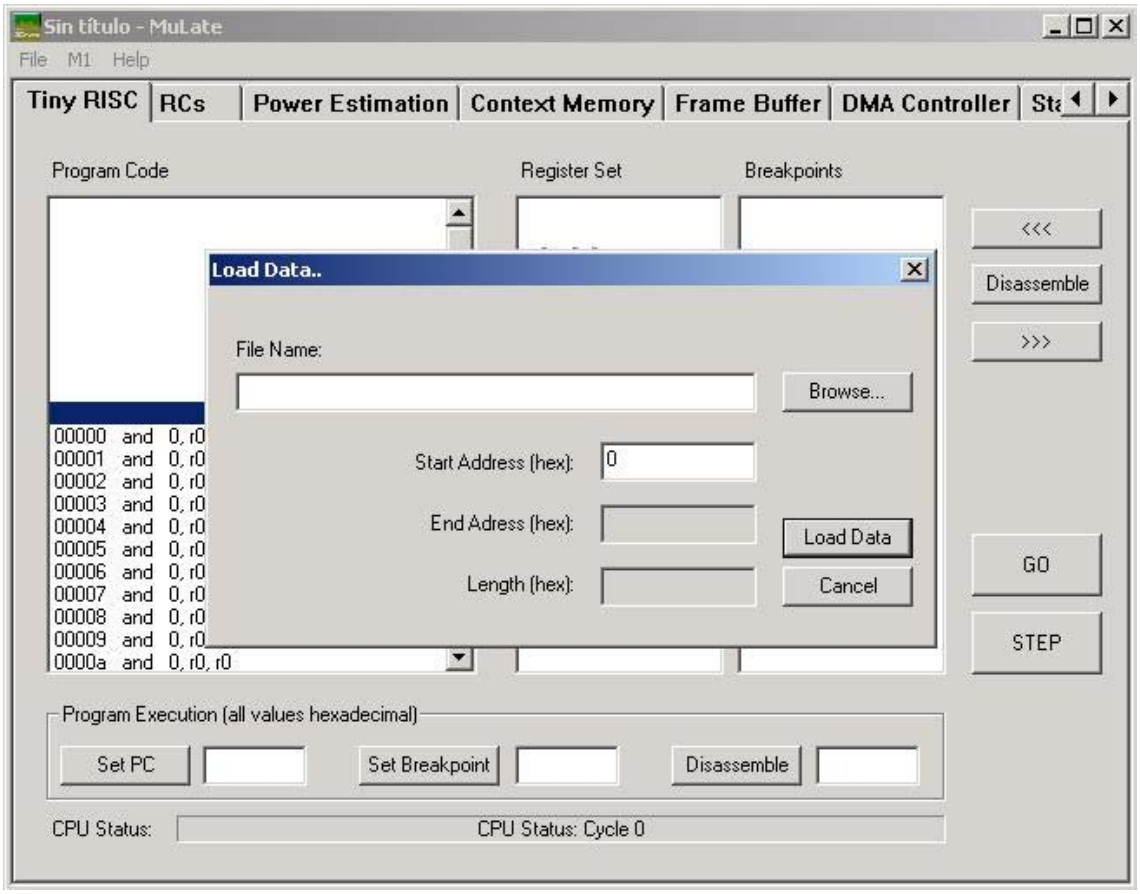

Figura 78. Simulador pidiendo archivos (programa, contextos y datos) y su dirección.

Empezamos con el programa, cargaríamos el archivo “prg_jfa.hex” con la dirección de comienzo 0. Como vemos en Fig. 78, ya tenemos el programa cargado.

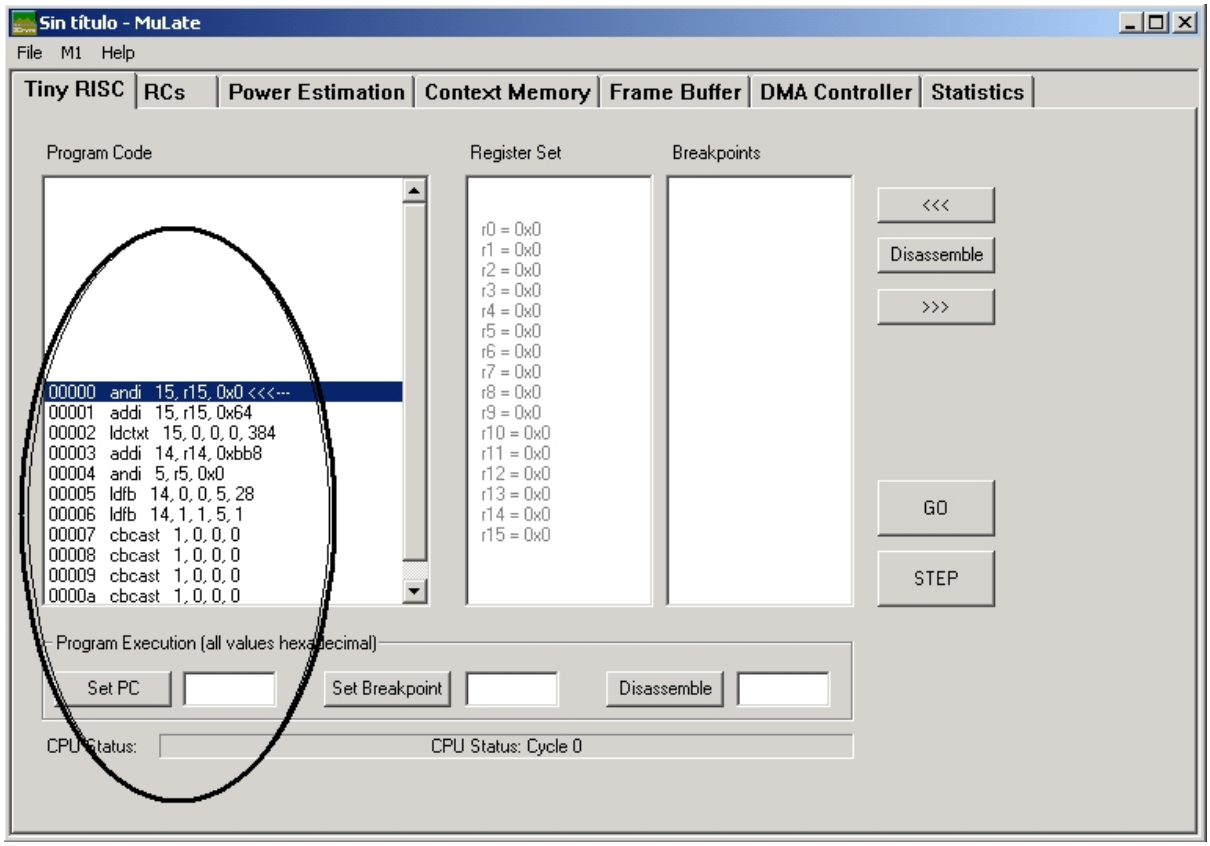

Figura 79. Simulador con el programa cargado

Lo siguiente serían los contextos, volvemos a realizar la misma operación y esta vez cargamos en archivo “ctx_jfa.hex” a partir de la dirección “0x64”. Como se observa en Fig. 80, tenemos todos los contextos a 0.

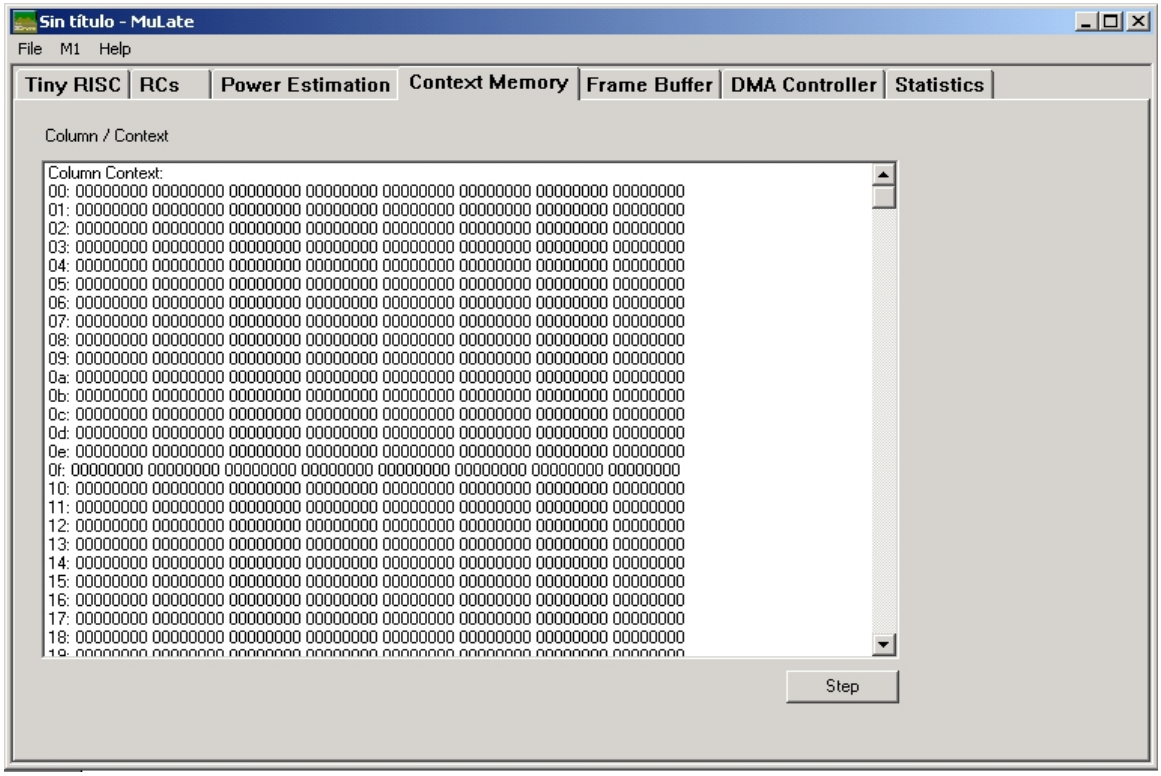

Figura 80. Simulador con los contextos a 0

Y una vez cargado los contextos, vemos lo siguiente (Fig. 81):

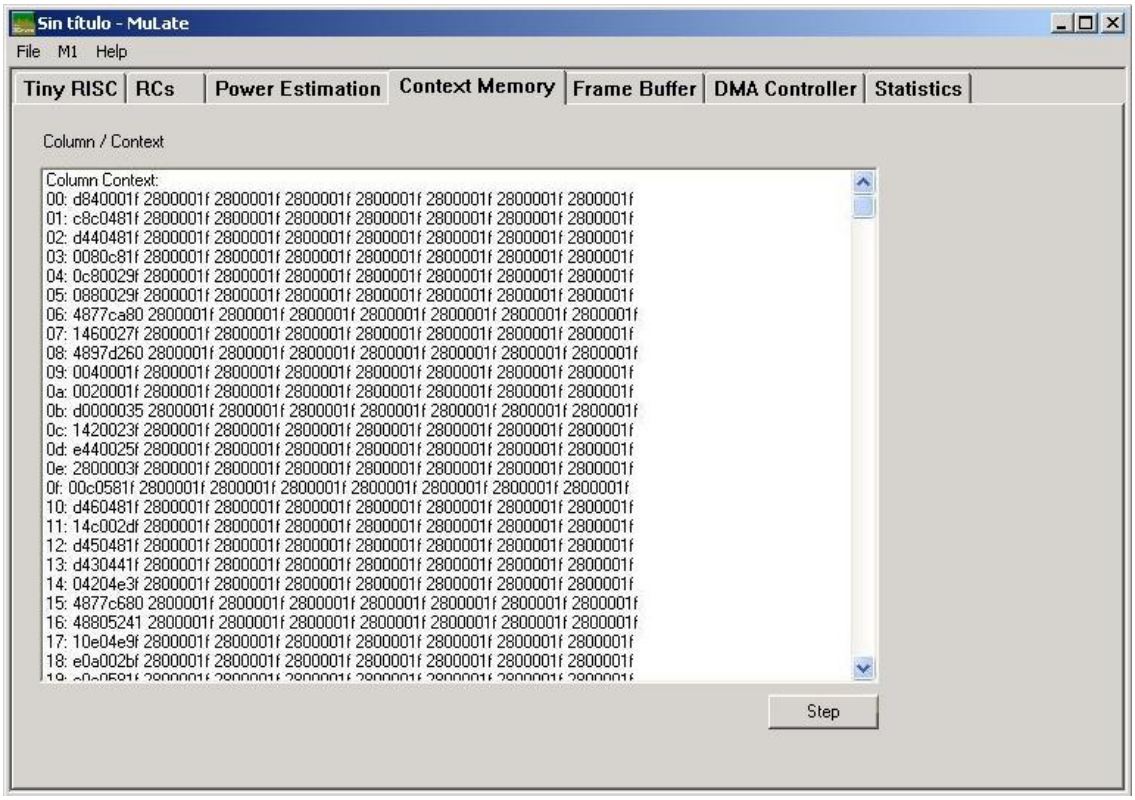

Figura 81. Contextos ya cargados

Lo último será el archivo de datos “datos_jfa.hex” sobre la dirección “0x1f4”. Que de estar todo a cero, ahora tenemos en el banco A lo siguiente (ver Fig. 82):

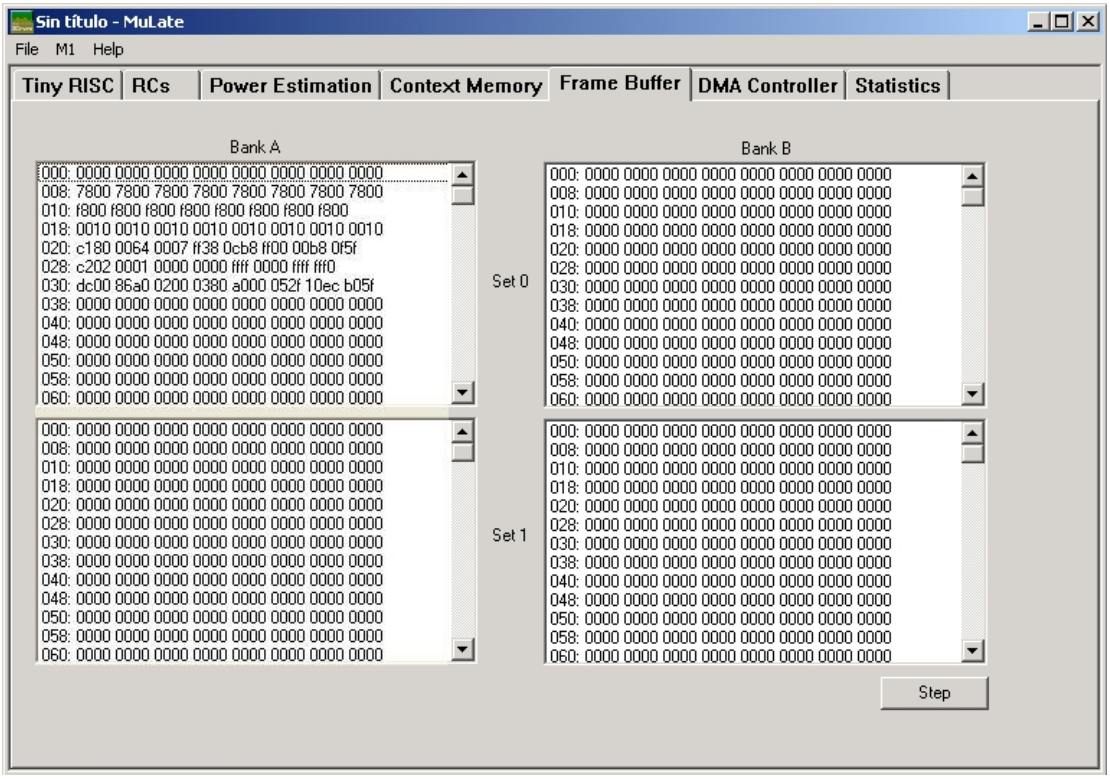

Figura 82. Datos cargados en el Frame-Buffer

Con esto ya tenemos listo el simulador para ejecutar nuestro programa. Podemos pulsar directamente sobre GO para que se ejecute, sin embargo, para verlo paso a paso podemos pulsar STEP, para ir instrucción a instrucción, o poner un Breakpoint con Set BreakPoint donde queramos y ejecutar hasta ahí pulsando GO, observemos la Fig. 83

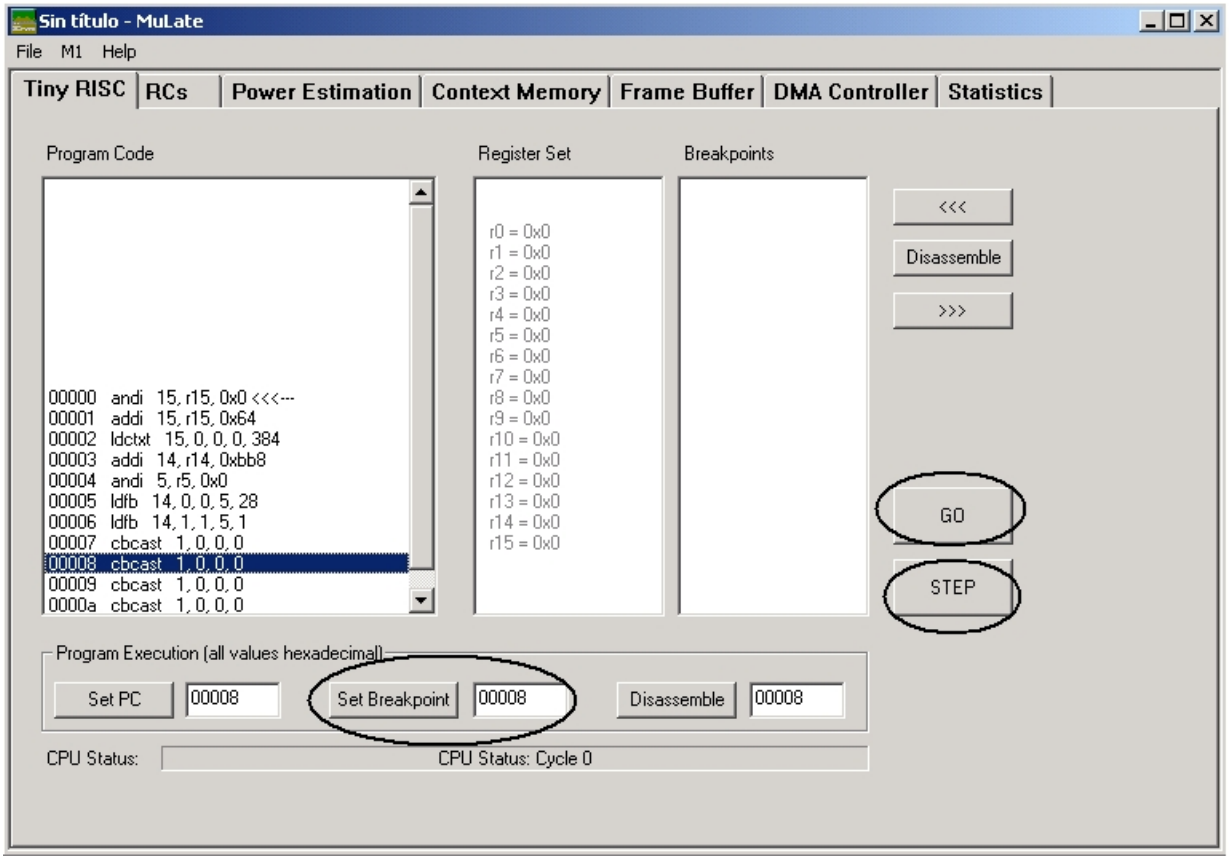

Figura 83. Simulador listo para ejecutar el programa

Una vez estemos ejecutando, podemos pinchar en la pestaña RCs, donde veremos el array de 8x8 celdas reconfigurables, donde pulsando en cada una de manera individual podremos ver que es lo que hace cada una en cada momento, en la Fig. 83 hemos tomado la primera celda que hay, y podemos ver el valor en distintas partes, la instrucción que está ejecutando, las entradas a los multiplexores de la celda o los datos que hay en los distintos registros que estamos usando.

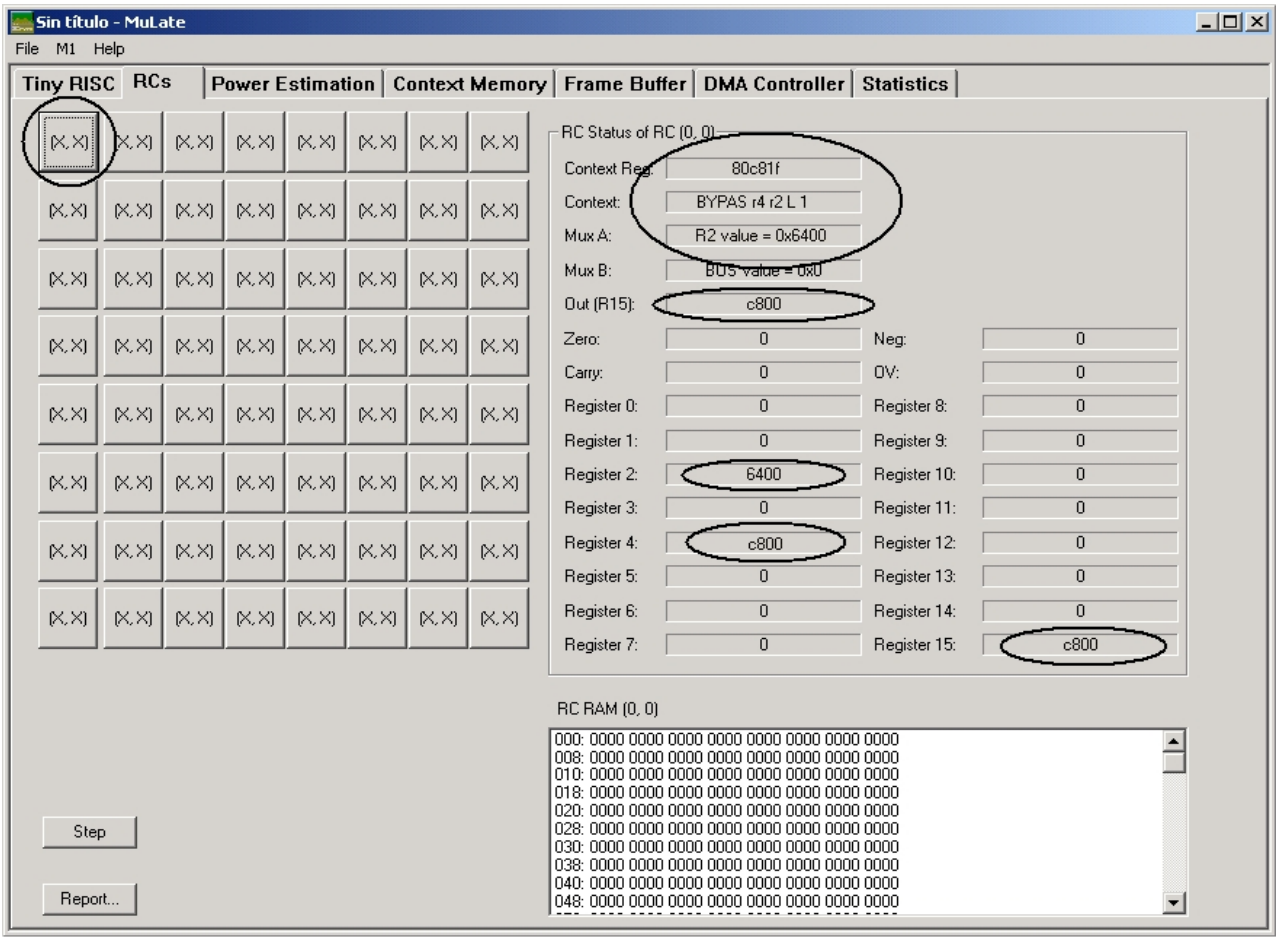

Figura 84. Matriz de las 64 celdas en el simulador, con sus valores actuales en un determinado ciclo

10 RESULTADOS

En este apartado vamos a ver por un lado, aplicaciones que hemos diseñado para ayudarnos a la hora del procesamiento y renderizado de imágenes, así como al cálculo de los resultados experimentales obtenidos.

Las aplicaciones diseñadas que se describen a continuación son un conversor que hemos creado para poder transformar los archivos de formato *.ply a nuestro formato. Tenemos otro conversor binario, con la misma funcionalidad, pero lo usamos cuando los archivos PLY vienen codificados en binario. Y por último tenemos un programa que nos permite crear triángulos de manera aleatoria, pinchando con el ratón sobre el cuadro que nos abre. Una vez explicadas estas aplicaciones, lo último que mostramos son los datos experimentales obtenidos según las dos estrategias principales ya descritas en apartados anteriores para distintas imágenes.

10.1 Conversor

El objetivo de esta aplicación desarrollada en C++ es el de convertir imágenes sin renderizar codificadas en un formato ya creado a nuestro propio formato de imagen, de forma que rendericemos estas imágenes ya creadas en lugar de desarrollar nosotros mismos una aplicación de diseño y creación de imágenes, lo cual es muy costoso y se escapa al objetivo de este proyecto.

Dichas imágenes están codificadas en el formato PLY, desarrollado en el *Stanford Computer Graphics Laboratory*, de la universidad *Stanford University* en Stanford (California) **[23]**.

10.1.1 PLY

El formato PLY es muy sencillo, se compone de una pequeña cabecera, con la información acerca de las propiedades de los vértices y polígonos que componen los objetos de la imagen. Una vez pasada la cabecera, la imagen contiene una lista de cada vértice con sus propiedades, y una lista de polígonos con los vértices que lo forman. A continuación se muestra una cabecera de ejemplo:

```
ply
format ascii 1.0
comment made by Greg Turk
comment this file is a cube
element vertex 8
property float x
property float y
property float z
element face 6
property list uint8 int32 vertex_index
end_header
```

Figura 85. Cabecera de un archivo PLY

La cabecera de un archivo PLY siempre comienza con la palabra ‘ply’, y acaba con ‘end_header’. El resto de comandos que tiene son:

- format: indica el tipo de formato con el que está codificado el archivo. Las posibilidades son: ‘ascii’, ‘binary_big_endian’ y ‘binary_little_endian’; donde la primera está codificada en ASCII, y las dos últimas en binario.
- comment: indica la inclusión de un comentario al archivo.
- element: indica la definición de un elemento, el cual puede ser un vértice (vertex) o un polígono (face) principalmente, con el número de ellos que están presentes en la imagen, y que se listarán al final de la cabecera. Existen otros elementos, como las curvas (edges), pero no los usaremos en nuestra aplicación.
- property: siempre va a continuación de alguna definición de elemento, y definen una propiedad del mismo. Para ello, muestran información sobre el tipo de dato que define esta propiedad, y el nombre de la misma. Las propiedades más comunes para vértices son: ‘x’,’y’,’z’ (coordenadas del vértice); ‘r’,’g’,’b’ (color del vértice); y ‘nx’,’ny’,’nz’ (coordenadas del vector normal a ese vértice); aunque hay más propiedades, como la intensidad de la luz sobre cada vértice (intensity), por ejemplo. Para los polígonos, solo se suele usar la propiedad list, la cual viene con dos tipos definidos, y el método de listado de los polígonos. Este método es habitualmente ‘vertex_indices’, que consiste en especificar, con el primer tipo de la propiedad, un entero sin signo que denota el número de vértices que tendrá ese polígono, y con el segundo tipo, un entero de 32 bits, el tipo del índice a cada uno de los vértices listados en el archivo justo antes que los polígonos.

```
0 0 0          { inicio de la lista de vértices }
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3      { inicio de la lista de polígonos }
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

Figura 86. Resto del archivo PLY

A continuación de la cabecera, como se puede ver en la figura 41, empieza la lista de vértices con los valores de sus propiedades, que en este caso son las coordenadas (x,y,z). Por último, se muestra la lista de polígonos. Su primer valor, 4, nos indica que el polígono tiene cuatro vértices. Los otros cuatro valores son los índices a dichos vértices, resultado de recorrer la anterior lista de vértices de arriba abajo, y empezando por el índice 0.

10.1.2 Nuestro formato

El objetivo del conversor PLY es traducir este tipo de archivos a nuestro formato de imagen, que es el que renderiza nuestra aplicación. Este formato es muy sencillo, dado que solo acepta triángulos, por simplicidad a la hora de renderizar, y porque es el tipo de polígono más utilizado en las renderizaciones. Además, cada triángulo siempre contiene los mismos datos, como explicaremos a continuación.

```
Coordenadas -512.732 512.732 -350 350 1024 699
numTriangulos 96966
Triangulo 123.629 235.436 194.386 -0.97 -0.14 -0.15 0.8 0.8 0.8 ...
Triangulo 123.629 235.436 194.386 -0.97 -0.14 -0.15 0.8 0.8 0.8 ...
Triangulo 123.606 235.222 196.546 -0.99 -0.03 0.00 0.8 0.8 0.8 ...
...
```

Figura 87. Ejemplo de imagen en nuestro formato

El archivo contiene tres tipos de etiquetas:

- 'Coordenadas': Incluye las dimensiones y coordenadas del puerto de vista donde visualizaremos la imagen.
- 'numTriangulos': Esta etiqueta se coloca en primer lugar en el archivo, y nos indica el número de triángulos que contiene la imagen.
- 'Triangulo': Precede a los datos relativos a cada triángulo, los cuales se definen en el siguiente orden:

Triangulo x0 y0 z0 nx0 ny0 nz0 Rr Rg Rb x1 …

donde x0 y0 z0 son las coordenadas del primer vértice, nx0 ny0 nz0 son las coordenadas del vector normal al primer vértice, y Rr Rg Rb son los coeficientes de reflexión del objeto en ese vértice, para cada componente de color RGB, suponiendo que los tres tipos de reflexión (ambiente, difusa y especular) tienen el mismo coeficiente.

10.1.3 Funcionamiento de la aplicación

Para ejecutar la conversión, primero hay que introducir la ruta donde se encuentra el archivo PLY, y la ruta donde queremos dejar el archivo convertido.

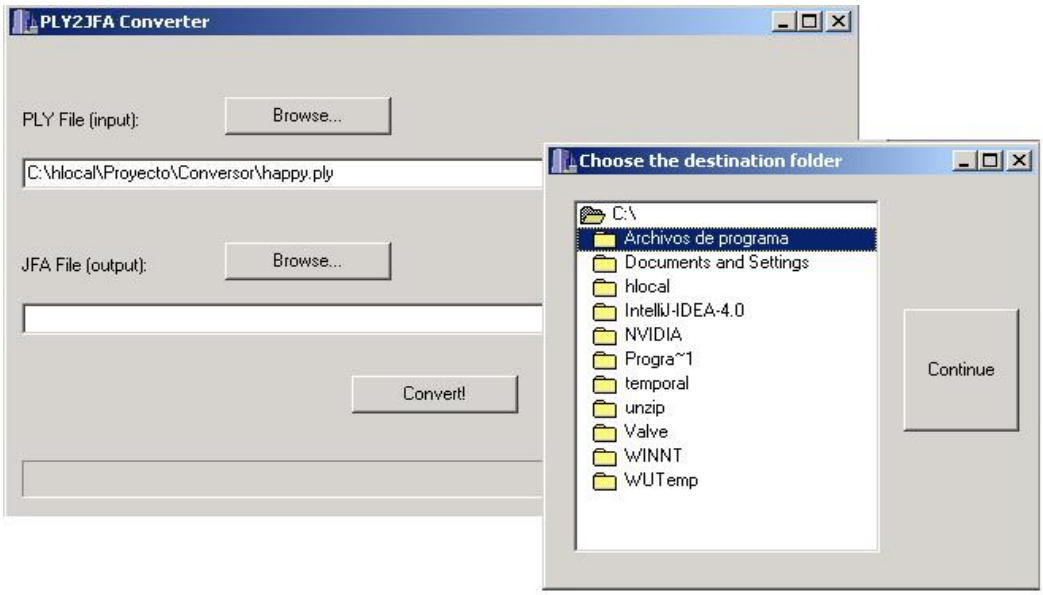

Figura 88. Introducción de las rutas de los archivos en el conversor

Con las rutas fijadas, podemos iniciar el procesamiento de la conversión pulsando el botón *Convert!*. El proceso de conversión es el siguiente:

- Se abre el archivo PLY y se procesa su cabecera. El programa comprueba cuáles de las propiedades que necesitamos están en el archivo (estas propiedades son las coordenadas de los vértices, las coordenadas de los vectores normales a los vértices, y el color RGB de los mismos, además de la lista de polígonos) y sus posiciones, para no procesar las propiedades ni los elementos que no nos interesan.
- Se crean listas dinámicas para almacenar los datos de vértices y polígonos, comprobando primero que dichos polígonos sean triángulos, ya que de otro modo nuestro renderizador no los aceptará.
- La aplicación comprueba si la imagen PLY incluía los vectores normales a cada vértice. Si no, deberá calcularlos. Para ello, primero calcula los vectores normales a cada triángulo de la escena, haciendo el producto vectorial de los vectores correspondientes a dos de sus lados. Con estos vectores normales, calcula el vector normal de cada vértice mediante una aproximación de los vectores normales de los triángulos que comparten ese vértice. Este vector final deberemos normalizarlo para que la imagen sea correcta.

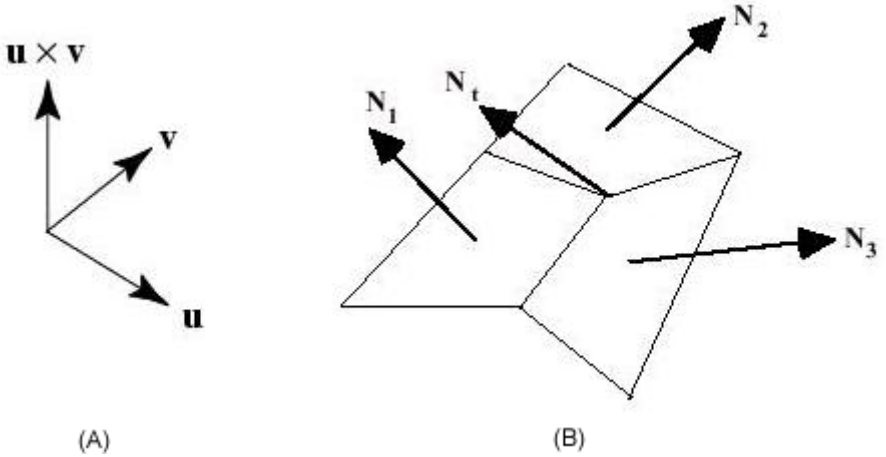

Figura 89. (A) Producto vectorial de dos vectores. (B) Cálculo del vector normal de un vértice mediante aproximación

- Por último, escribimos estos datos en el archivo convertido con el formato antes comentado, escribiendo los datos calculados en el caso de que no vinieran en el archivo original PLY. Estos datos pueden ser los vectores normales a los vértices o el color de cada uno. Si la figura no tuviera los valores RGB de color, se colocarían unos por defecto que cree un tono grisáceo en la figura, y que con la iluminación final de la sensación de que dicha figura, ya renderizada, esté en blanco y negro.

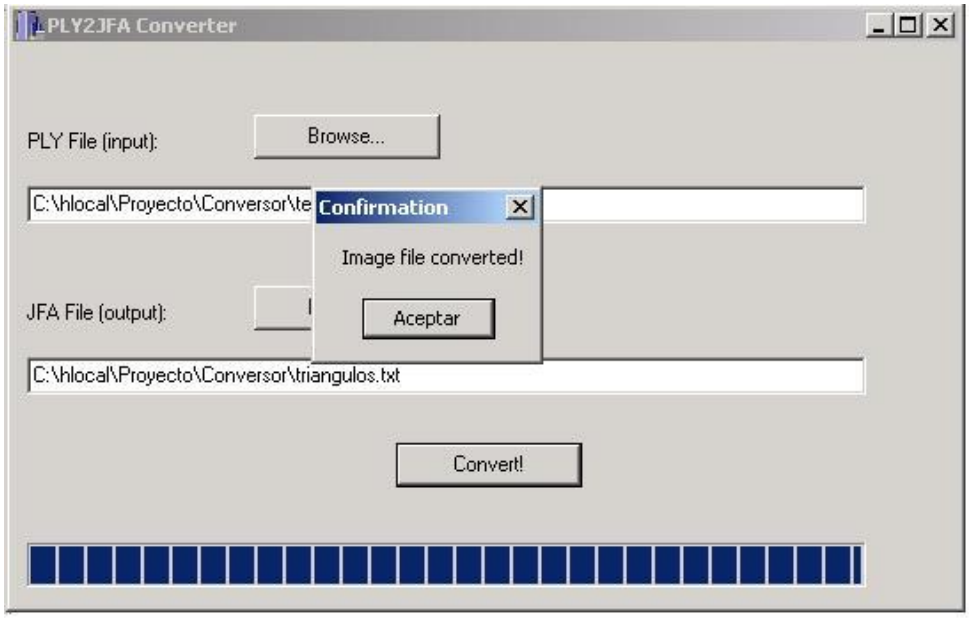

Figura 90. Finalización de la conversión de archivos

10.2 Conversor binario

Esta aplicación solventa el problema de leer los archivos PLY cuando están codificados en binario. La mayoría de estos archivos no se han codificado en ASCII porque el volumen de datos que guardan es demasiado grande, y codificándolos en binario conseguimos reducir el tamaño del archivo. Sin embargo, necesitamos distinguir cuándo un archivo PLY es binario, para traducirlo a código ASCII primero, y poder convertirlo después en nuestro formato.

10.2.1 Funcionamiento de la aplicación

La aplicación está desarrollada en lenguaje C, para aprovechar las librerías de interpretación de archivos PLY binarios que se encuentran en el sitio web del *Stanford Computer Graphics Laboratory*. Mediante estas librerías de funciones, podemos crear un sencillo programa que lee un archivo PLY binario y lo traduce a un archivo PLY ASCII. Para ello, debemos especificar el archivo PLY binario, y el nombre del archivo PLY donde pondremos los datos en formato ASCII. La aplicación interpretará la

cabecera y leerá los datos de los vértices y los polígonos, pasándolos directamente al archivo PLY de salida ya en formato ASCII.

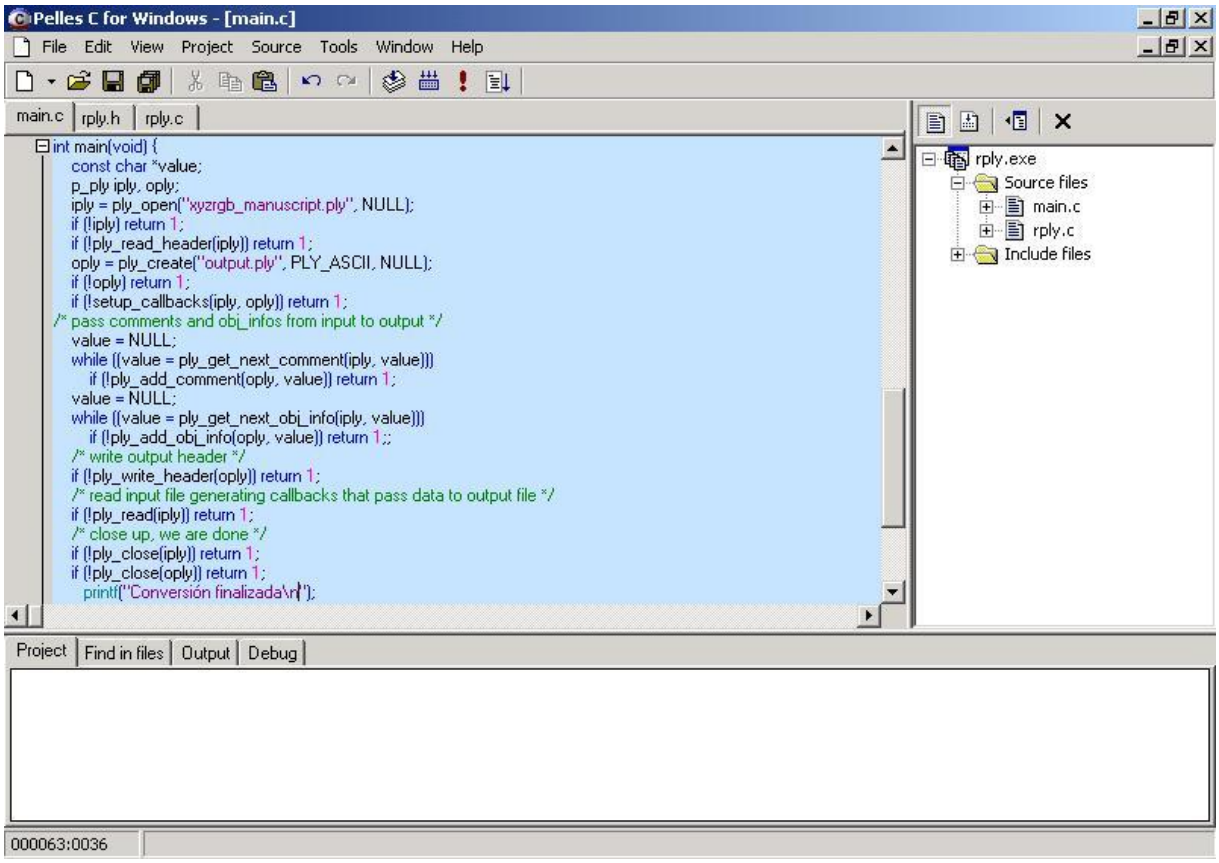

Figura 91. Módulo principal de la librería de manipulación de archivos PLY

10.3 Editor de triángulos

Esta aplicación, desarrollada en C++ tiene como objetivo generar archivos gráficos con nuestro formato, para poder probarlos en renderizador. Esta aplicación se uso en principio para depurar nuestro renderizador, ya que las imágenes contenían cantidades pequeñas de triángulos. Una vez depurado el renderizador, utilizamos las imágenes del conversor para renderizar objetos con mayor cantidad de polígonos.

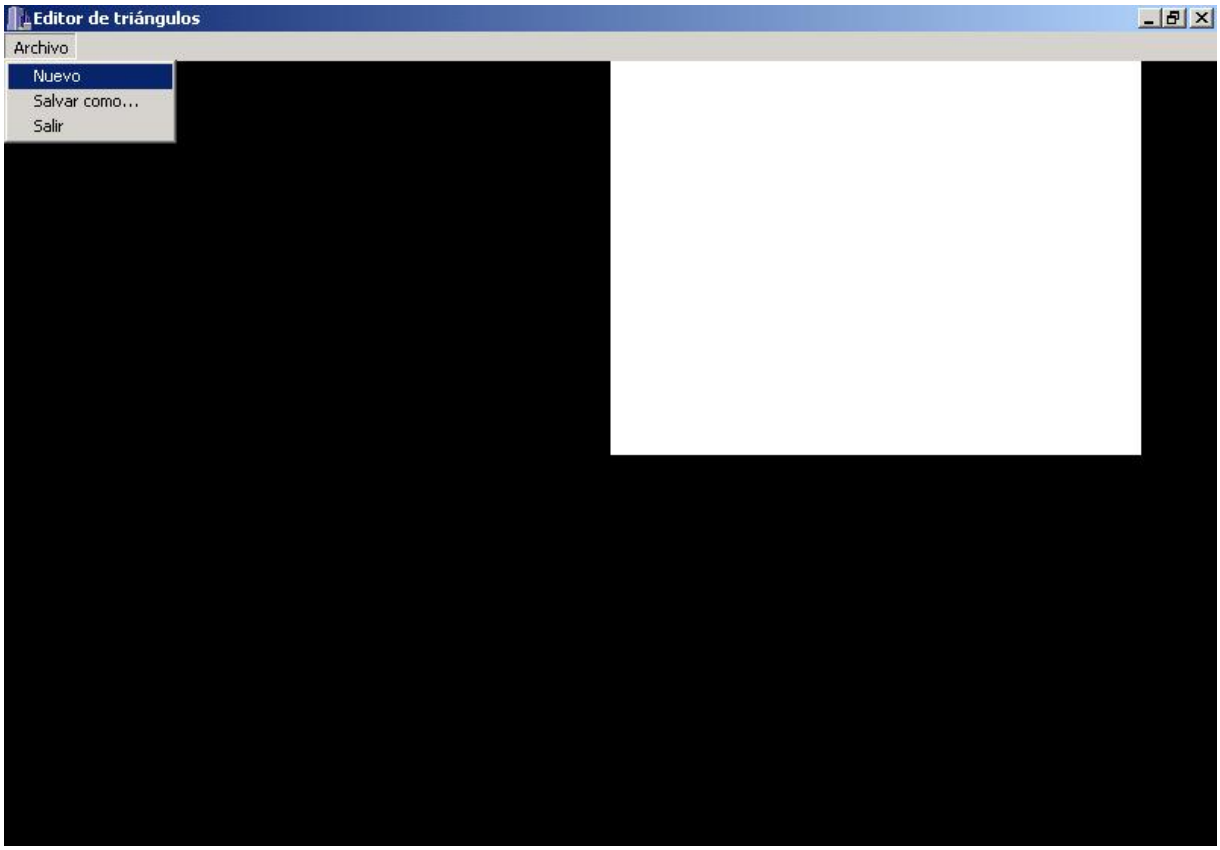

Figura 92. Interfaz gráfica del Editor de Triángulos

10.3.1 Funcionamiento de la aplicación

El programa nos permite dibujar triángulos que posteriormente se grabarán en un archivo de imagen de nuestro formato. Para ello, nos hemos servido de las librerías de OpenGL, creando una interfaz muy sencilla en la que se dibuja cada triángulo pinchando sobre tres puntos de la pantalla. Las opciones de la aplicación nos permiten limpiar la pantalla de triángulos, o guardar la imagen en un archivo en la ruta que queramos especificar. Una vez guardado el archivo, se puede utilizar como entrada de nuestro renderizador para mostrar la imagen resultado.

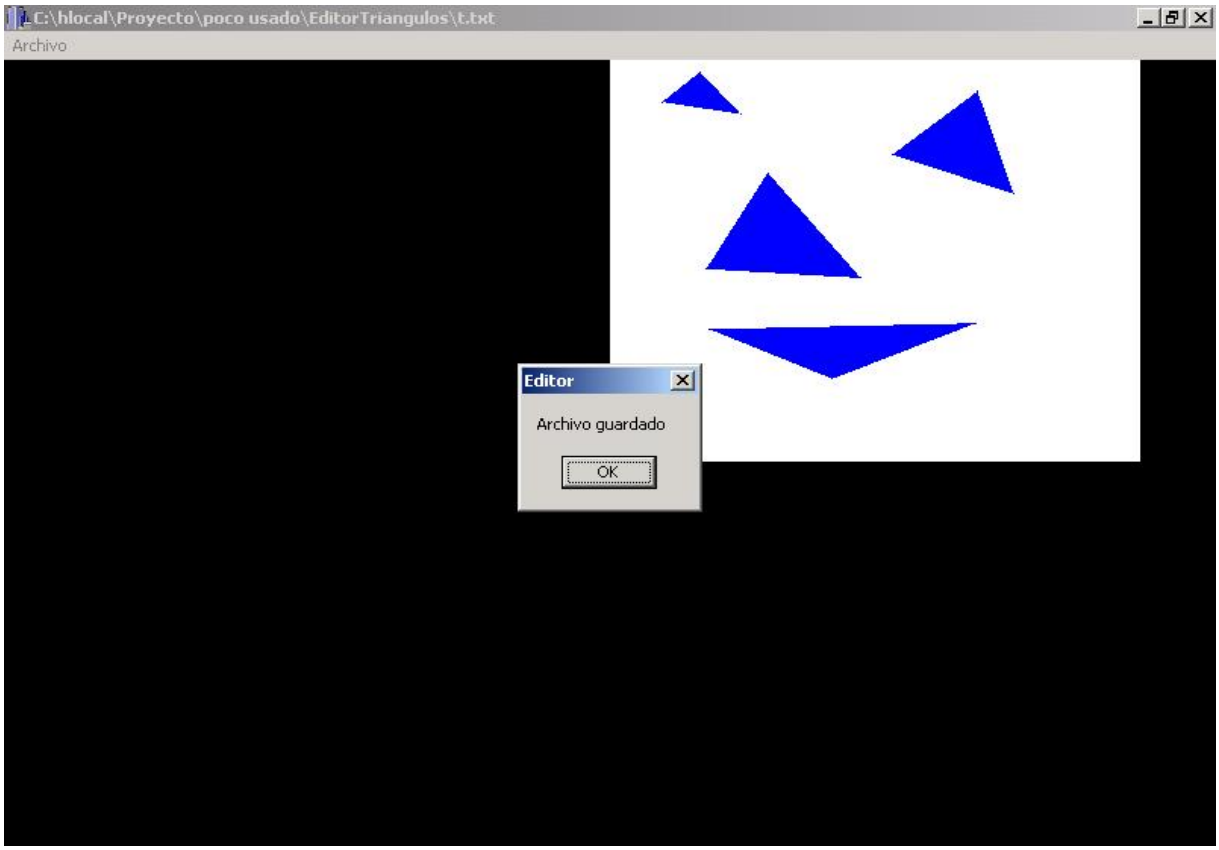

Figura 93. Grabado de la imagen dibujada en la pantalla

Para que la aplicación pueda ejecutar estas acciones, se controlan los eventos de pulsado del ratón sobre la pantalla para obtener las coordenadas de la posición del ratón, transformarlas a coordenadas de OpenGL, y guardarlas como un punto. Esta transformación de sistema de coordenadas debe realizarse, ya que en caso contrario, la renderización mostrará los triángulos en coordenadas de OpenGL, cuando éstos están definidos en coordenadas de la pantalla, por lo que se mostrarán en ubicaciones distintas a las originales. Cuando obtenemos el tercer punto, se crea un objeto de tipo 'Triángulo' y se almacena en una lista, que es la que posteriormente se graba en el archivo de salida, con nuestro formato antes explicado. Las coordenadas y dimensiones de la pantalla del editor se guardan para no alterar el puerto de vista en la renderización.

10.4 Resultados del proyecto

A continuación, se muestran las imágenes renderizadas con nuestra aplicación. Dichas imágenes estaban originalmente en formato PLY, y se han obtenido de la página web del repositorio de imágenes escaneadas del *Stanford Computer Graphics Laboratory*, de la universidad *Stanford University* en Stanford (California) **[4]**. Además, mostraremos una estadística con el número de fps (frames por segundo), y el número de triángulos por segundo, con los que MorphoSys es capaz de ejecutar cada imagen.

10.4.1 Imágenes

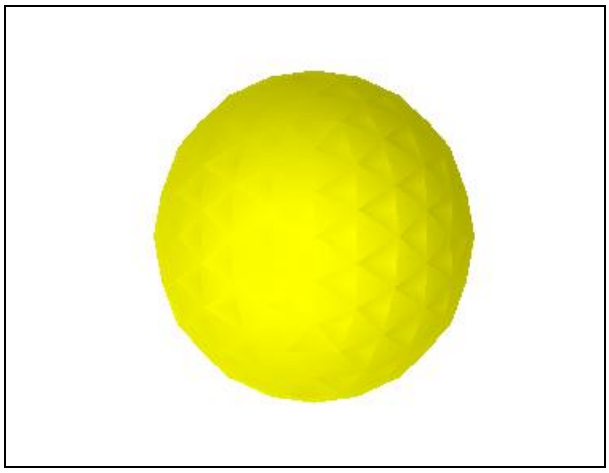

Imagen 1. Bola

Imagen 2. Dinosaurio

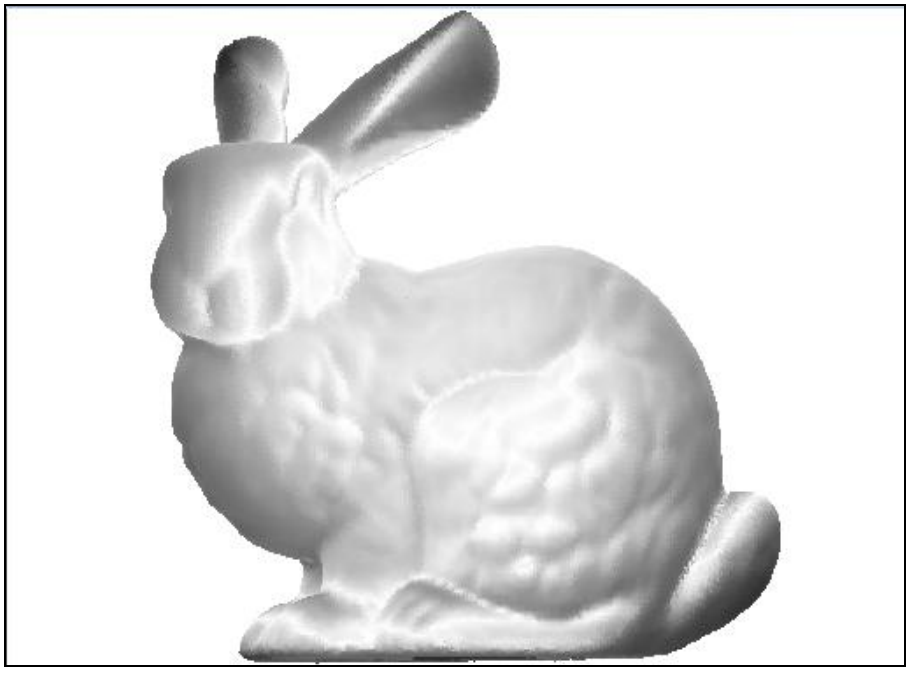

Imagen 3. Conejo

Imagen 4. Tetera (vista de planta)

Imagen 5. Tetera (elevación lateral)

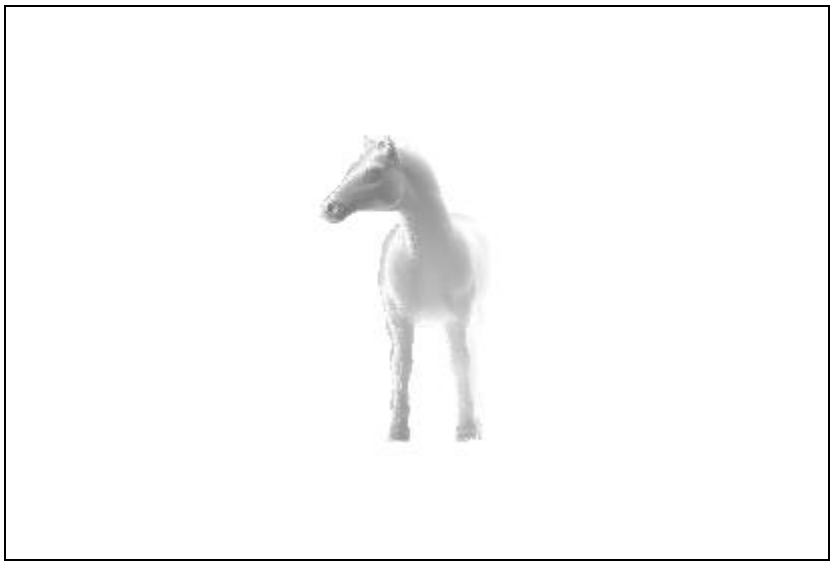

Imagen 6. Caballo (elevación frontal)

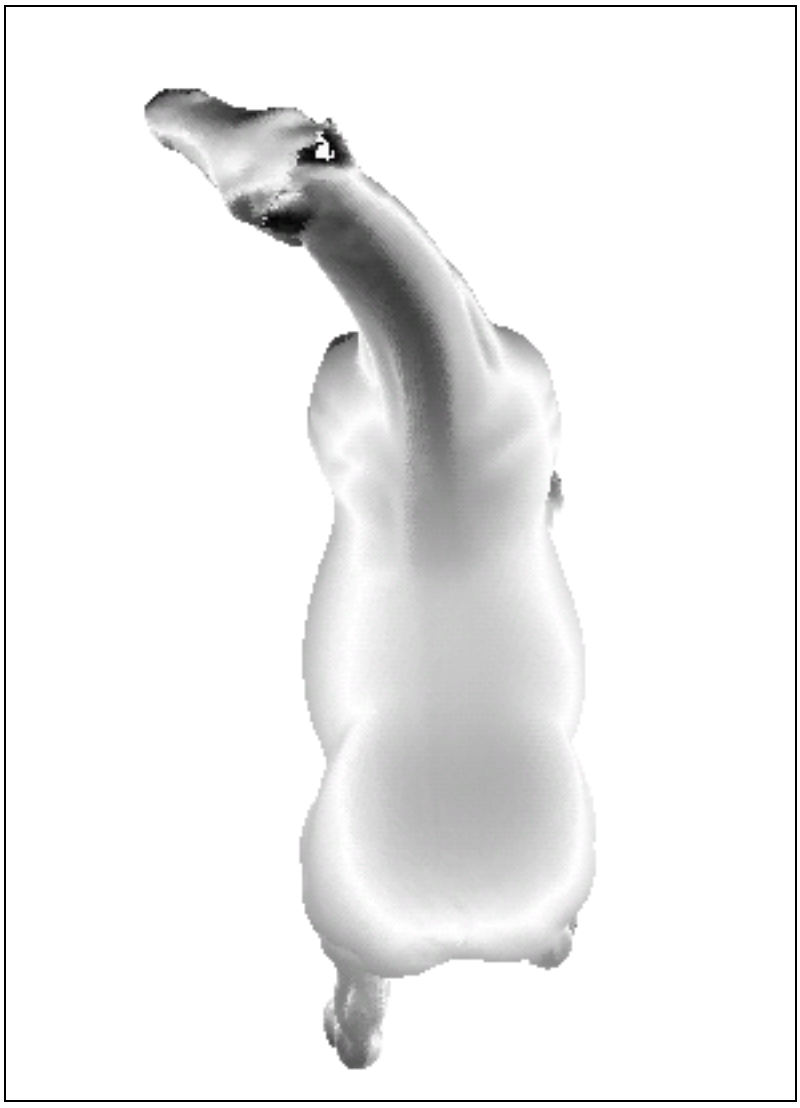

Imagen 7. Caballo (vista de planta)

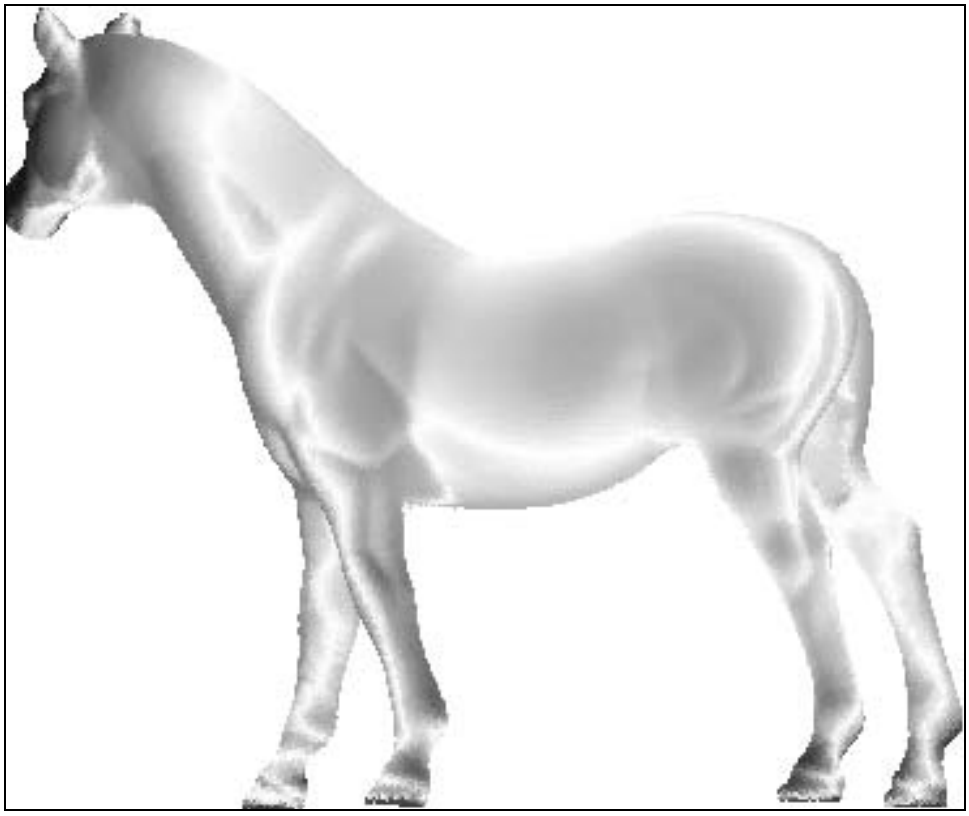

Imagen 8. Caballo (elevación lateral)

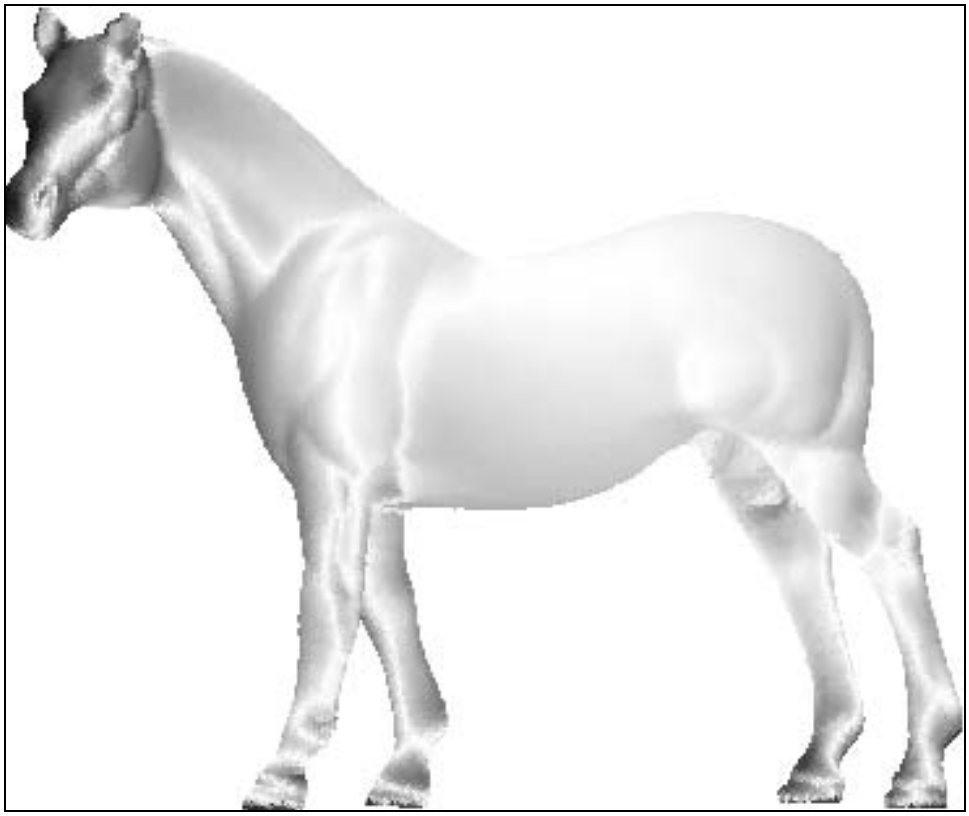

Imagen 9. Caballo (elevación lateral)

Imagen 10. Buda Feliz (elevación frontal)

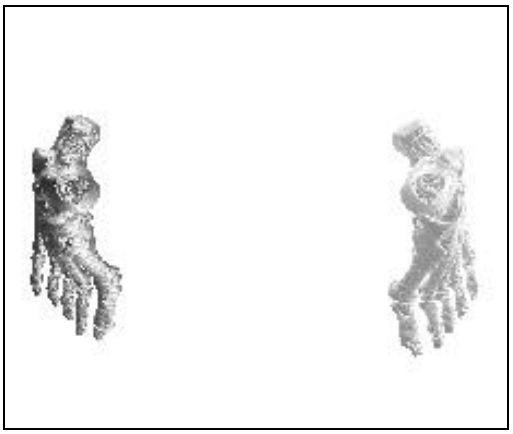

Imagen 11. Huesos de los pies (vista de planta)

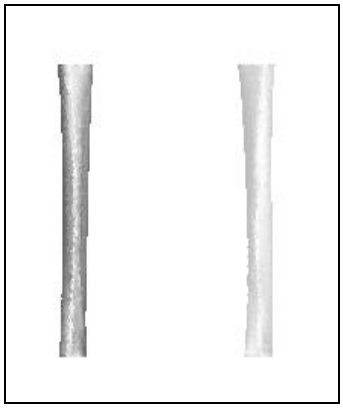

Imagen 12. Fémur (elevación frontal)

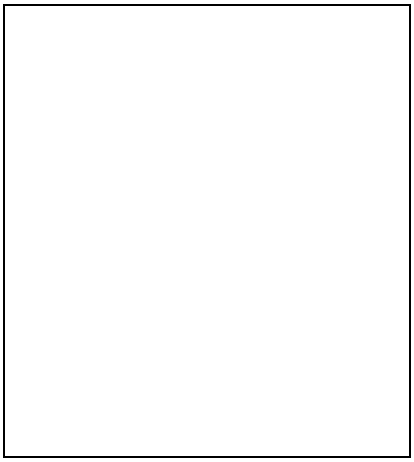

Imagen 13. Tórax y cadera (elevación frontal)

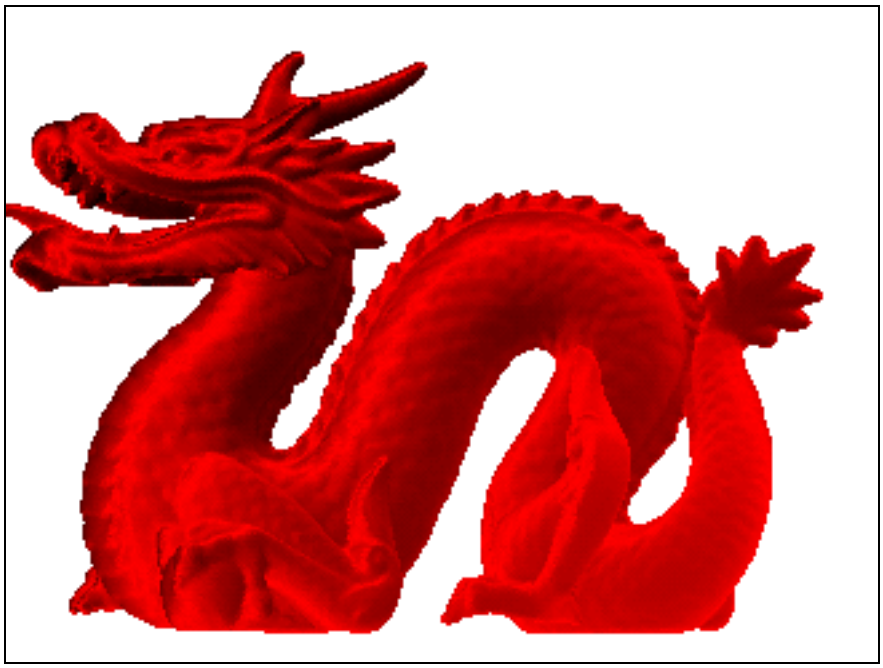

Imagen 14. Dragón (elevación lateral)

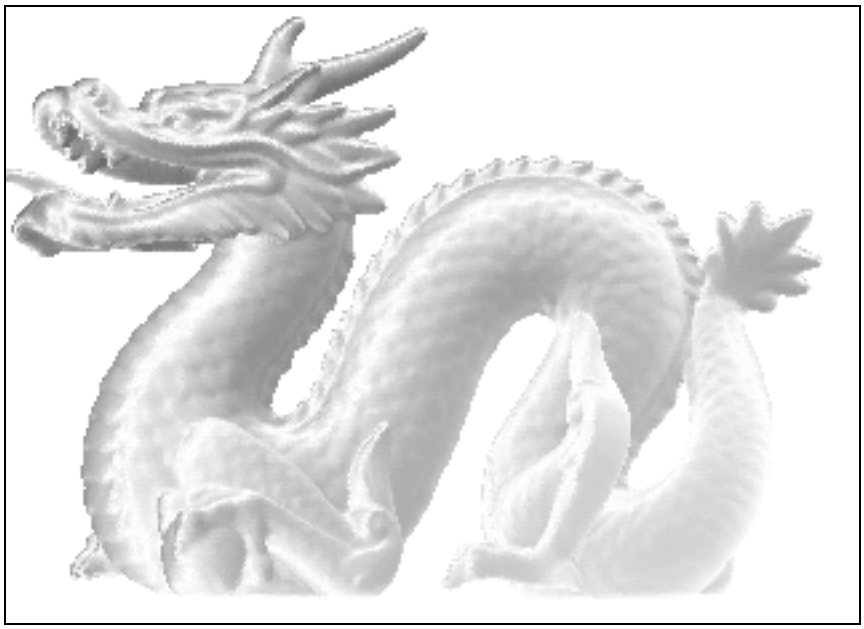

Imagen 15. Dragón (elevación lateral)

Imagen 16. Huesos de la mano (elevación lateral)

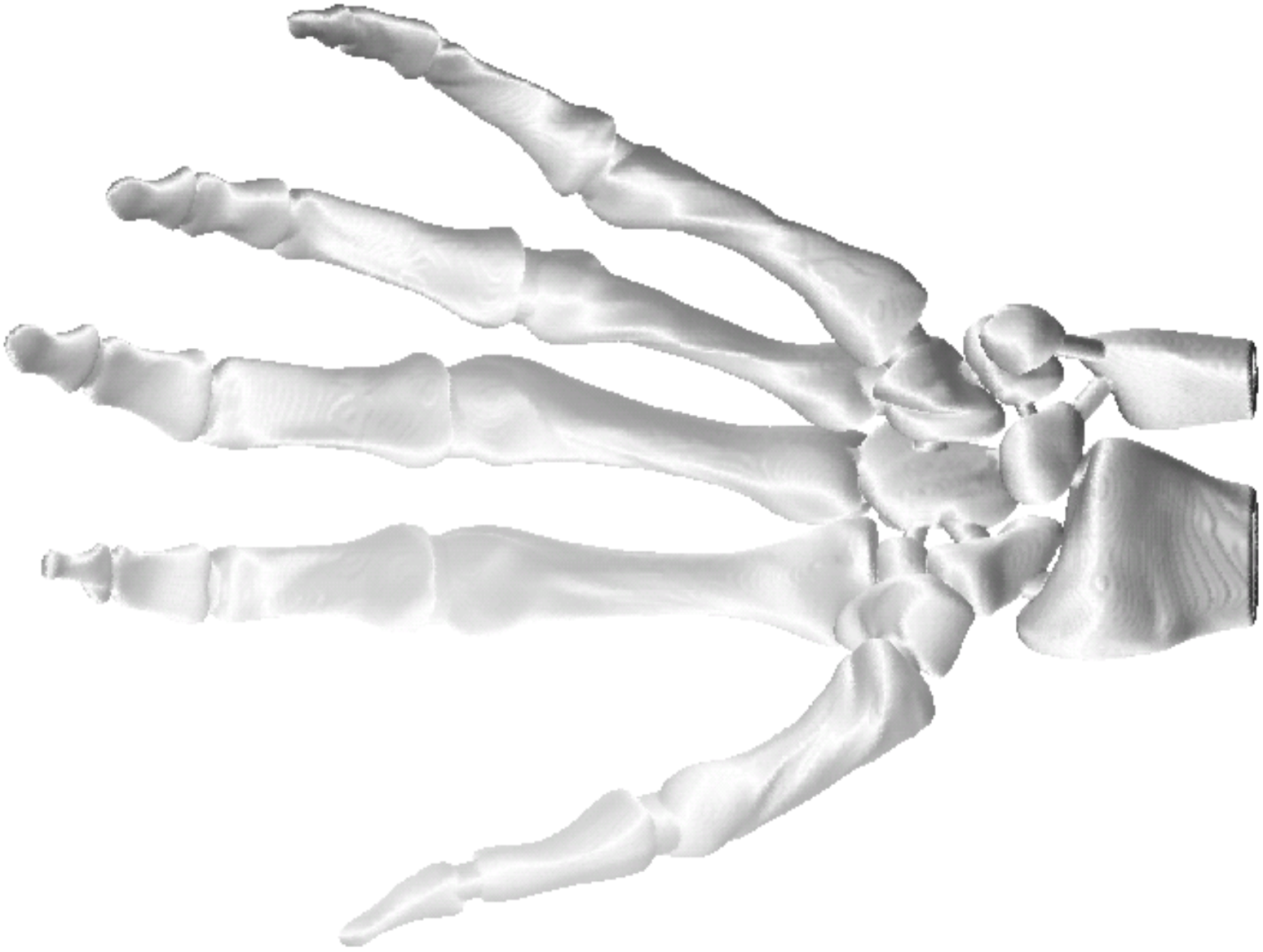

Imagen 17. Huesos de la mano (vista de planta)

10.4.2 Estadísticas

En la siguiente tabla mostraremos, para cada imagen, una estadística con los valores del número de triángulos que tiene, y para cada estrategia de MorphoSys implementada, el número de Frames por segundo (fps) y el número de triángulos por segundo (T/s). Suponemos una resolución de 2300 x 2300 píxels, y una frecuencia de ejecución en MorphoSys de 450 Mhz.

La estrategia 1 es la desarrollada finalmente sobre MorphoSys, consistente en ejecutar cada mitad de un triángulo en una celda.

La estrategia 2 es la estrategia elemental que correspondería a la ejecución de un triángulo por celda.

Imágenes	**Número de triángulos**	**Estrategia 1**		**Estrategia 2**	
		fps	**T/s**	**fps**	**T/s**
Imagen 1. Bola	320	5251	1.680.320	4918	1.573.760
Imagen 3. Conejo	69.451	516	35.836.716	447	31.044.597
Imagen 5. Tetera	2.256	2156	4.863.936	1855	4.184.880
Imagen 8. Caballo	96.966	657	63.706.662	596	57.791.736
Imagen 10. Buda feliz	1.087.716	133	144.666.228	124	134.876.784
Imagen 11. Huesos pies	124.018	1013	125.630.234	939	116.452.902
Imagen 12. Fémur	29.331	1802	52.854.462	1585	46.489.635
Imagen 13. Tórax y cadera	1.136.745	73	82.982.385	67	76.161.915
Imagen 17. Mano	654.666	128	83.797.248	111	72.797.776

Como podemos observar, obtenemos muy buenos resultados, saliendo solamente un número de fps más bajos, para cantidades sumamente grandes de triángulos. Esto hace que podamos implementar nuevas técnicas en el algoritmo de renderización, para mejorarlo obteniendo todavía resultados interactivos para el rendering.

Otros resultados de rendering para diferentes arquitecturas

Arquitecturas	**Unidades de proceso**	**Número de Triángulos**	**Triángulos por segundo**
CM-200[24]	16.384	228.000	681.456
Cluster “Chromium”[25]	32 * 2 Pentium III	4.011.299	71.000.000
Nvidia GeForce FX 5200[26]	1	máximo	81.000.000

11 Bibliografía

[1] Luis Alonso Romero, Juan Corchado Rodríguez. Informática gráfica. Departamento de Informática y Automática, Universidad de Salamanca.

[2] "3D Computer Graphics" Allan Watt. Addisson Wesley, 2003. Pags 50-100.

[3] http://www.ktx.com/3dsmax/html/rendering.html

[4] http://www.pixar.com/renderman/renderdoor.html

[5] http://www.electricimg.com/

[6] http://www.geocities.com/SiliconValley/Way/2419/POV.html

[7] http://www.realsoft.fi.com

[8] http://www.hash.com/

[9] Illumination and shading models.
http://glasnost.itcarlow.ie/~powerk/Graphics/Notes/node10.html

[10] X.J. Guo, B. Land. Phong shading and Gouraud shading.
http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-95to96/guo/report.html

[11] http://www.fullhardware.com/catalog/guiatarjetasgraficas.php

[12] Katholic Universiteit y Vrije Universiteit de Bélgica, "ADRES: An Architecture with tightly coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix". Pag: 1-3.

[13] C.R. Landguth, M. Garverick, T. Gomersall, E.Holt, H. Arnold, J.M. and Gokhale, M. 1998. The NAPA adaptative processing architecture. *IEEE symposium on Field-Programable Custom Computing Machines*, 28-37.

[14] Razdan, R. and Smith, M.D. 1994. A high-performance microarchitecture with hardware-programmable funtional units. *International symposium on Microarchitecture*, 172-180.

[15] Miyamori, T. and Olukotun, K. 1998. A quantitative analysis of reconfigurable coprocessors for multimedia applications. *IEEE Symposium on Field-Programable Custom Computing Machines,* 2-11.

[16] Vullemin, J., Bertin, P., Roncin, D., Shand, D., M., Touati, H. and Boucard, P. 1996. Programable active memories: Reconfigurable system come of age. *IEEE Trans. VLSI Syst.* 4,1, 56-69.

[17] Quickturn, A Cadence Company 1999a. System Realizer™. On line at http://www.quickturn.com/products/systemrealizer.htm. Quickturn, A Cadence Company, San Jose, CA.

[18] M.S.Martín, H. Du, N. Tabrizi, Y. Long, N. Bagherzadeh, M. Fernández. Algorithm optimizations and mapping scheme for interactive ray tracing on a reconfigurable architecture.

[19] http://www.pactcorp.com, Articulo "*The XXP White: A Technical Perspective"*, Pag: 1-6.

[20] http://www.pactcorp.com, Articulo *"Smart Media Processing with XXP: The SMeXXP device in low-cast wireless multimedia terminals"*, Pag: 5-7, 10-16.

[21] Chris Fisher, Kevin Rennie, Guanbin Xing, Stefan G. Berg, Kevin Bolding, John Naegle, Daniel Parshall, Dmitriy Portnov, Adnan Sulejmanpasic y Carl Ebeling de la Universidad de Washington. Articulo *"An Emulator for Exploring RaPiD Configurable Computing Architectures"*

[22] Moldovan, D. I. and Fortes, J. A. B., *"Partitioning and mapping algorithms into fixed sizesystolic arrays"*, IEEE Transactions on Computers, 1986, Pag: 1-12.

[23] Página web del repositorio de archivos PLY del laboratorio de Stanford: http://graphics.stanford.edu/data/3Dscanrep

[24] Thinking Machines Corporation. Connection Machine CM-200 Series Technical Summary, 1991.

[25] Greg Humphreys, Matthew Eldridge, Ian Buck_ Gordon Stoll, Matthew Everett y Pat Hanrahan, Stanford University Intel Corporation. Artículo *WireGL: A Scalable Graphics System for Clusters*

[26] http://www.nvidia.com

12 Lista de palabras claves

- Hardware Reconfigurable
- Algoritmo de renderización JFA
- Gráficos 3D
- Morphosys
- Simulador Morphosys
- Programación Morphosys

Mediante el texto siguiente, se autoriza a la Universidad Complutense de Madrid a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

José Manuel Sánchez Díaz

Alfonso de Torres Pérez

Francisco Javier Dávila Foncuverta