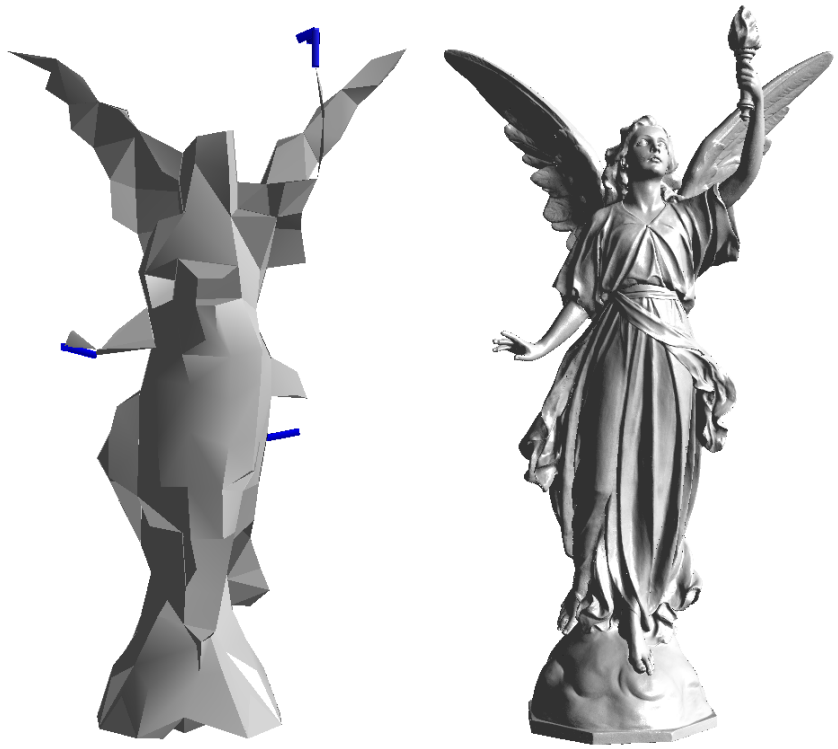
Simplificación de mallas de triángulos

M. Pasenau
C. Andújar



Publicación CIMNE Nº-361, Junio 2011

Simplificación de mallas de triángulos

M. Pasenau
C. Andújar

Publicación CIMNE Nº-361, Junio 2011

Centro Internacional de Métodos Numéricos en Ingeniería
Gran Capitán s/n, 08034 Barcelona, España

Índice

1. Introducción

1.1. Motivación

Con la mayor potencia de cálculo los ordenadores como de los superordenadores, el nivel de detalle de los modelos 3d para simulaciones numéricas también se incrementa proporcionalmente llegando actualmente a los cientos de millones de tetraedros y triángulos. Hablándose de miles de millones en un futuro muy cercano *[Yilmaz2011]*.

También la resolución de los escáneres 3d se va incrementando y se generan mallas de miles de millones de triángulos. Por no hablar de la información extraída de los escáneres médicos y tomografías que después se usan en simulaciones biomédicas tanto de resistencia ósea como de circulación de fluidos dentro de arterias y venas.

Por otro lado actualmente en la sociedad de la comunicación, y más con el boom de la visualización 3d, se quiere tener acceso a toda esta información desde cualquier lugar. Las plataformas que actualmente proliferan son ordenadores con gráficos modestos: smartphones, tablets, portátiles ligeros con gráficos integrados.

Como se puede ver en la **figura 1***[Shilov2010]*, la mayor parte de los ordenadores que actualmente se comercializan disponen de gráficos integrados, con un rendimiento 3D más bien modesto que limita la complejidad de los modelos a visualizar para poder permitir la interactividad con el usuario.

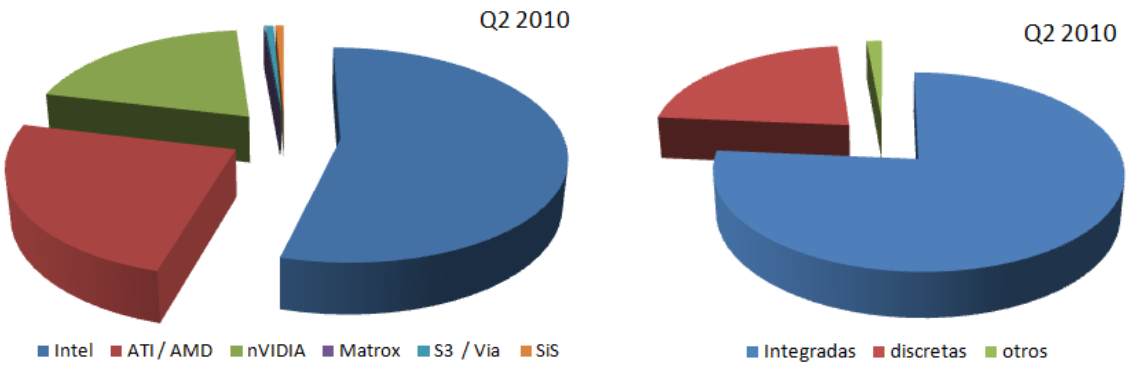


Figura 1: distribución del mercado de tarjetas gráficas en el segundo cuatrimestre de 2.010 [I.2] según marca, izquierda, y según el tipo de tarjeta, derecha.

Según una práctica de laboratorio que se hizo para la asignatura de Visualización avanzada en Octubre del 2.009 *[Pasenau2009],* los tamaños prácticos para conseguir unos 25fps de los modelos según las diferentes tarjetas gráficas son los mostrados en la siguiente tabla:

Plataforma	Número de triángulos
Intel Dual Core E8400 + ATI Radeon X1300 (256MB)	2.000.000
Intel Dual Core E8500 + ATI Radeon HD 3450 (256MB)	2.000.000
Intel QuadCore Q9550 + Software Mesa GL	200.000
Intel QuadCore Q9550 + Intel G45 (shared memory)	2.000.000
Intel QuadCore Q9550 + ATI Radeon HD 3650 (256 MB)	4.000.000
Intel QuadCore Q9550 + NVIDIA GeForce 9800 GT (512 MB)	7.000.000
Intel QuadCore Q9550 + NVIDIA GTX 275 (896 MB)	14.000.000
Intel Core2Duo 6600 + NVIDIA GeForce 7600 GT (256 MB)	4.000.000
Intel Atom N280 HT + 945 GME (shared memory)	100.000

Tabla 1: tamaño de modelos máximo con el que la visualización consigue unos 25 fps

Este es el reto al que intenta enfrentarse este PFC: buscar, implementar y evaluar un algoritmo de simplificación de modelos grandes para poder permitir que el usuario pueda interactuar con ellos en plataformas modestas. Se pretende que esta simplificación dependa de criterios geométricos y de atributos nodales de la malla, ya sean resultados o coordenadas de texturas. Se hará especial hincapié en su consumo de recursos y en su rapidez, pues al depender de la vista y de los atributos que el usuario seleccione, se tendrá que volver a simplificar el modelo.

1.2. Objetivos

El objetivo principal del proyecto de fin de carrera es el análisis, diseño, implementación y prueba de un algoritmo que permita simplificar rápidamente una malla de triángulos muy densa, tratando de minimizar tanto el error geométrico como el error de atributos definidos en la superficie, para que el usuario pueda interaccionar con ella en plataformas modestas.

Este algoritmo formará parte de una librería y se usará en la aplicación comercial de pre y post proceso gráfico GiD, www.gidhome.com.

Otros requerimientos del algoritmo son:

- el usuario pueda seleccionar el nivel de simplificación para obtener una pérdida 'aceptable' de calidad,
- establecimiento de un criterio de simplificación geométrico y / o de atributos,
- que sea lo más rápido posible, pues dependiendo de la visualización del atributo o de la vista del modelo, se tendrá que simplificar de nuevo el modelo,
- que el uso de memoria es aceptable,
- multiplataforma: el algoritmo se probará tanto en MS Windows como en Linux,
- robusto: se usará una batería de ejemplos para validar el funcionamiento del algoritmo.

La lista de los ejemplos se ha confeccionado a partir de los ejemplos mostrados en los artículos, así como de modelos específicos de simulación y modelos grandes hallados en la internet, realizando las conversiones de formato necesarias para poderlos leer y se puede encontrar en el anexo del proyecto.

Para poder cumplir con los objetivos y los requerimientos se desarrollará una aplicación que servirá de marco de trabajo para las diferentes etapas del proyecto y se usarán las siguientes plataformas:

- *plataforma 1, de trabajo*: Intel Quad Core Q9550 con una tarjeta gráfica NVIDIA GTX 275 o con una tarjeta integrada Intel G45 y con sistema operativo MS Windows 7 de 64 bits y Linux Ubuntu 8.04.4 LTS;
- *plataforma 2, de testeo*: Intel i7 920 con una tarjeta gráfica ATI Radeon 5870 y Linux Ubuntu 10.04.2 LTS;
- *plataforma 3, de testeo*: Intel Pentium Dual SU4100 con gráficos Intel y MS Windows 7;
- *plataforma 4, de testeo*: módulo con 2 procesadores Intel Xeon E5410 QuadCore 2.33 GHz y RedHat 5.1 ELS 64 bits.

2. Antecedentes

La simplificación de mallas y construcción de niveles de detalle ha sido un campo bastante activo en las últimas dos décadas, desde el algoritmo de agrupación de vértices (vertex clustering) de Rossignac and Borrell en 1993 *[Rossignac1993]*, pasando por el algoritmo de contracción de aristas de Michael Garland en 1997 *[Garland1997]*, hasta los últimos basados en octrees *[Schaefer2003]* y portados a las tarjetas gráficas *[DeCoro2007]*.

El objetivo es estudiar e implementar un algoritmo que sea fácilmente paralelizable para cpu y que, más adelante, se pueda llevar a la tarjeta gráfica, para poder hacer comparaciones equitativas.

El esquema de simplificación que se acerca más a este objetivo es del de agrupamiento de vértices, intuitivamente paralelizable, pues el de contracción de aristas funciona con una cola de prioridades según el coste de simplificación de la arista.

"Adaptive Vertex Clustering Using Octrees"

Entre los primeros artículos estudiados está el de Scott Schaefer y Joe Warren *[Schaefer2003] que,* para simplificar mallas grandes sin almacenarlas enteramente en memoria, "out of core", proponía un enfoque de agrupación de vértices adaptivo usando un octree dinámico con una función de error cuádrica en cada nodo para controlar el error y poder colapsar ciertas ramas. Dado un límite de memoria, o especificando un número de nodos, el algoritmo iba construyendo el árbol insertando vértices en las hojas y cada nodo del árbol contiene la función de error cuádrica para representar la superficie agregada del subárbol que cuelga de ese nodo. Cuando se llega al límite de memoria, o se llega al número de nodos especificado y aún quedan vértices por procesar, se colapsan los sub-árboles con menos error, colapsando los vértices de las hojas a un único vértice usando la función de error cuádrica guardada. Para asegurar que en los sub-árboles colapsados no se insertará ningún vértice nuevo, previamente se ordenan los vértices de la malla para asegurar un orden de llenado de las hojas del árbol.

La función de error cuádrica para un vértice v se define como la suma de distancias cuadradas de este vértice al conjunto de planos asociados a este vértice *[Garland1997] [Schaefer2003] [DeCoro2007]*:

$$E(v) = \sum_{p \in planos(v)} (p^T v)^2 = v^T \left(\sum_{p \in planos(v)} p^T p \right) v = v^T Q_v v$$

Inicialmente *planos(v)* se construyen a partir de los triángulos adyacentes a *v*. Cuando se colapsan dos vértices *(a, b)* → *c*, asignamos *planos(c) = planos(a)* ∪ *planos(b)*. En *[Garland1997]* se explica que podemos guardar simplemente la matriz 4x4 Q_v en vez de guardar el conjunto de planos incidentes. De esta manera, la unión se reduce a la suma de cuádricas. Y así, la cuádrica de un nodo del árbol es la acumulada de las cuádricas de su subárbol.

Para encontrar el punto que mejor aproxima a al grupo de vértices cuya función de error cuádrica es Q_v simplemente se busca el punto que minimiza este error. Esto se puede hacer de esta manera:

$$Q_v = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ q_{14} & q_{24} & q_{34} & q_{44} \end{bmatrix} \quad entonces \quad \tilde{v} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

En el artículo, en vez de invertir la matriz, se usa la factorización QR propuesta por Tao Ju, Frank Losasso, Scott Schaefer y Joe Warren en *[Ju2002]*, que se muestra numéricamente más estable para calcular $E(v)$.

Para generar la malla simplificada, se numeran los vértices de las hojas del octree y se escriben, después se procesan los triángulos de la malla. Si los tres vértices del triángulo caen en tres hojas diferentes el triángulo se escribe en disco. Si dos o los tres vértices del triángulo caen dentro de la misma hoja del octree, el triángulo se convierte en una línea, en un punto o sencillamente se descarta.

El algoritmo de Schaefer y Warren tiene la ventaja de simplificar mucho la malla concentrando los vértices en las zonas con más detalles, proporcionando mallas de mayor calidad que un simple algoritmo de agrupamiento uniforme. Su desventaja radica en el pre-procesamiento de los vértices, necesario para el algoritmo, y que puede llegar a ser tres veces superior al tiempo de construcción del octree en los ejemplos del artículo, pues su algoritmo tiene coste *O(n·log(n))* .

"Real-time Mesh Simplification using the GPU"

El algoritmo de DeCoro y NatalyTatarchuk *[DeCoro2007]* porta el algoritmo de agrupación de vértices de la CPU a la tarjeta gráfica, simplificando mallas en tiempo real. Además de adaptar el algoritmo al pipeline gráfico, propone un octree probabilístico para, dado un criterio de error, simplificar al máximo allí donde se pueda y concentrar vértices en las zonas con más detalle. En el artículo explica tres algoritmos: un "vertex clustering" uniforme, otro "vertex clustering" pero usando una función de deformación para mantener los detalles en las 'zona de interés' y el tercero usa un octree probabilístico para concentrar vértices en las zonas con más detalles.

Los tres algoritmos usan los siguientes recursos de la tarjeta gráfica:

- **texturas:** tabla en la que se guarda el grid 3d de cuádricas y donde cada celda está identificada por un *cluster_id* construido a partir de la tripleta *(i, j, k)* que identifica la celda dentro del grid;
- **procesador de vértices (PV):** se usa para calcular en qué celda cae cada vértice y asignarle su *cluster_id*;
- **procesador de geometría (PG):** se procesan los triángulos de la malla;
- **procesador de píxeles (PP):** se encarga de acumular los puntos y las cuádricas en cada celda y obtener el vértice que mejor aproxima a los que caen en la celda.

El primer algoritmo, agrupación de vértices uniforme, se siguen estas tres etapas:

1. *creación del mapa de cuádricas*: a partir de la caja contenedora del modelo, la malla original y el tamaño del grid:
   - **PV**: para cada punto calcula qué celda le corresponde y le asigna un *cluster_id*

- **PG**: para cada elemento calcula la cuádrica que tiene asociada a sus vértices
- **PP**: acumula la cuádrica calculada al *cluster_id* del vértice.

2. *cálculo de los representantes óptimos*: para cada celda del grid, si la cuádrica es invertible, la invierte y optiene el punto que mejor aproxima a los vértices de la celda. Si no lo es, el representante es la media de los vértices de la celda.
3. *simplificación de la malla*:
   - **PV**: para cada punto calcula qué celda le corresponde y le asigna un *cluster_id*
   - **PG**: para cada elemento si los cluster_id son todos diferentes, se buscan las coordenadas de los vértices y el triángulo se pinta.

El segundo algoritmo simplemente incrusta la función de deformación (enfoque de las "zonas de interés") en el procesador de vértices, que se encarga de asignar el cluster_id a los vértices de la malla, asignando más celdas a las zonas de interés y menos celdas fuera de ellas.

El tercer algoritmo usa un *octree probabilístico* que no es más que un multigrid donde cada nivel tiene el doble de celdas en cada dimensión respecto al anterior, esto es sigue una subdivisión tipo octree. Cada celda de este grid guarda la función de error cuádrica acumulada para ese nivel y todos los inferiores, como en el octree the Chaefer y Warren *[Schaefer2003]*. Dado un nivel, en vez de guardar todas las celdas, hay un límite de memoria y utiliza una función hash para acceder a las celdas en tiempo constante. Pero no se pueden guardar todas las celdas, con lo que cada celda tiene una 'probabilidad' de ser almacenada. De ahí viene el nombre *octree probabilístico*. Este árbol tiene una profundidad máxima *lmax*, definida por el usuario.

Para simplificar la malla se sigue la estructura del primer algoritmo con los siguientes cambios:

- **en el primer paso**: el acumular las cuádricas en cada celda se transforma en guardar la cuádrica en las celdas correspondientes de cada nivel del octree. En vez de guardar esta información *lmax* veces, si el modelo es suficientemente detallado, se escoge un nivel al azar. No se unas una función aleatoria lineal si no ponderada según la profundidad del árbol.
- **en el tercer paso**: al buscar el *cluster_id* donde ha caído un vértice se selecciona el nivel del árbol según una *tolerancia de error* dado por el usuario. Como no todas las celdas de un nivel están guardadas, se busca en el nivel superior, que "seguramente" sí está almacenada.

Al usar funciones de hash, que pueden provocar colisiones, cada celda no sólo guarda la cuádrica correspondiente si no también su *cluster_id*, para poderla identificar cuando se recupera su información.

Este esquema de tres pasos: creación de mapa de cuádricas, búsqueda de la aproximación óptima para cada clúster, y simplificación de la malla es el que se ha adoptado en este proyecto, pues es fácilmente adaptable a la cpu y paralelizable. La primera versión de la aplicación de simplificación se ha basado en el primero de estos tres algoritmos: agrupamiento de vértices uniforme.

Este algoritmo guarda en memoria el grid entero con una ocupación bastante baja. Según el esquema propuesto y como la tarjeta gráfica sigue un mecanismo unidireccional, pues hay muchos problemas para usar una textura para leer y escribir a la vez, cada celda ha de guardar: 10 floats para

la función de error cuádrica, 4 floats para acumular los vértices del clúster y 4 floats para guardar la mejor aproximación de la celda, y su error. Un grid de 256^3 ocupa algo más de 1GB de memoria.

La siguiente tabla muestra los requerimientos de memoria para cada subdivisión del grid y la ocupación real del grid con el modelo lucy de 14.027.872 puntos y 28.055.742 triángulos:

Tamaño grid	Millones de celdas	MBytes	Celdas ocupadas	% ocupación
64^3	0,262	18	5.437	2,074%
128^3	2,097	144	22.784	1,086%
256^3	16,777	1.152	90.776	0,541%
512^3	134,218	9.216	351.329	0,262%

Tabla 2: requerimientos de memoria según el tamaño del grid y ocupación real del grid por el modelo lucy.

Hay algunos modelos, como el pensatore (997.875 puntos), el gargolyde (863.692 puntos) y el blade (882.954 puntos) que doblan el factor de ocupación en estos niveles pero en niveles más refinados mantienen prácticamente el número de celdas ocupadas, reduciendo así su factor de ocupación, debido a la resolución de las mallas

El algoritmo del artículo sólo obtenía triángulos. Una posible mejora es extraer además las líneas que resultan de los triángulos colapsados.

Al implementar esta versión del agrupamiento de vértices uniforme apareció un problema: aunque la función de error cuádrica se pueda invertir, el vértice que se obtiene puede salir de la celda que estamos tratando de resolver.

En el artículo de DeCoro y NatalyTatarchuk *[DeCoro2007]* no menciona que dada una celda, con su función de error cuádrica, aunque ésta se pueda invertir el vértice obtenido puede quedar fuera de la celda, porque la superficie es localmente plana o tiene curvatura Gausiana cero, o la matriz está mal condicionada.

En diversos artículos se proponen varias alternativas: Schaefer en *[Schaefer2003]* usa la factorización QR, Lindstrom en *[Lindstrom2000]* busca el vértice proyectado ortogonalmente en el espacio de soluciones que incluye el centro de la celda y para ello calcula los valores propios y vectores propios (descomposición de valor singular, singular value decomposition)

Para poder superar el límite de 256^3 el artículo propone el octree probabilístico, pero se han buscado alternativas para poder guardar todas las celdas ocupadas de un nivel. Se hizo un estudio con la batería de ejemplos del tanto por ciento de ocupación según el nivel del de división del grid y de sus requerimientos de memoria y en la siguiente tabla se muestran los resultados para el modelo lucy, el de mayor tamaño hasta ese momento.

Tamaño grid	Celdas ocupadas	% ocupación	MBytes
64^3	5.437	2,074%	0,373
128^3	22.784	1,086%	1,564
256^3	90.776	0,541%	6,233
512^3	351.329	0,262%	24,124
1.024^3	1.317.591	0,123%	90,472
2.048^3	4.555.555	0,053%	312,805
4.096^3	10.287.739	0,015%	706,403

Tabla 3: requerimientos de memoria guardando sólo las celdas ocupadas por el modelo lucy.

Para poder hacer este estudio simplemente se ha usado un bitmap 3D con un bit por celda del grid para marcar si estaba libre o ocupado. Para la resolución más alta, 4.096^3, el bitmap ya ocupaba 8 GBytes de memoria.

En los artículos leídos hasta el momento tampoco se puede encontrar ningún ejemplo con una profundidad de octree más grande de 8.

Para poder montar un octree, o poder llegar a un error muy pequeño, se necesita poder definir una profundidad del octree de más de 8, esto significa poder usar grid mayores que 256^3.

"Perfect Spatial Hashing"

El objetivo del método que proponen Lefebvre and Hoppe *[Lefebvre2006]* es minimizar el espacio usado por datos dispersos como por ejemplo los vóxeles útiles de una textura 3d que se aplicarán a una malla de superficie o los píxeles útiles de un bitmap.

Para guardar sólo las celdas ocupadas de un grid en la tabla H, usa una función hash multidimensional que es una combinación de otras dos funciones hash imperfectas y una tabla de offsets:

$$h(p) = h_0(p) + \Phi[h_1(p)]$$

Según el artículo, usando como h_0 y h_1 simples funciones módulo y siendo $\Phi[\,]$ una tabla pequeña de offsets, con valores que maximizan la coherencia de h, consiguen lo que llaman una función hash perfecta, que no tiene colisiones, pudiendo llegar a ser una función hash perfecta mínima, donde además la tabla no tiene celdas vacías.

La función de este offset es "agitar" la función imperfecta h_0 hasta conseguir hacerla perfecta con sólo un 15-25 % de las entradas de la tabla H, y cada celda de solo 8 bits por coordenada.

A continuación se describe el proceso de construcción de esta función de hash perfecta:

Si el dominio es un grid de u^3 celdas donde sólo n están ocupadas, entonces el tamaño de la tabla H, que contendrá los datos de las celdas ocupadas del grid, será de $m^3 < 1{,}01 \cdot n$. El factor 1,01 es para tener un poco de espacio libre si $m > 256$, pues el offset es de sólo 8 bits.

El tamaño de la tabla de offsets $\Phi[\,]$ es $r^3 \geq s \cdot n$ donde $s = \frac{1}{2 \cdot 3}$ y cada celda es un vector de 3 componentes de 8 bits cada una. Para escoger una r óptima, es decir que la tabla de offsets sea lo más compacta posible, el artículo propone hacer una búsqueda binaria.

Las funciones h_0 y h_1 se escogen de manera que datos de diferentes celdas se mapeen en posiciones diferentes de la tabla H, es decir que si $h_0(p_1) = h_0(p_2)$ y $h_1(p_1) = h_1(p_2)$ entonces $h(p_1) = h(p_2)$ independientemente del valor de $\Phi[\,h_1(p_1)\,]$ y no será un hash perfecto si $p_1 ! = p_2$. Para ello el artículo sugiere escoger si m y r co-primos y de manera que $r \cdot m \geq u$.

Para crear la tabla de offset $\Phi[\,]$, el artículo desaconseja llenar la tabla de 0 o de valores aleatorios, buscar colisiones y allí donde las haya, reajustar los offsets.

Según el artículo de Fox *[Fox1992]* las entradas más problemáticas de ajustar son las que tienen el mayor número de colisiones, así que Lefebvre, siguiendo esta heurística, asigna offsets prioritariamente (greedily) a las celdas con más colisiones, usando 'bucket sort'. Para cada entrada q se busca un valor de offset tal que ${h_1}^{-1}(q)$ no colisione con ningún dato previamente asignado, es decir que toda celda p ocupada del grid original que tengan el mismo valor para $h_1(p) = q$ su entrada en la tabla $H\big[\,h_0(p) + \Phi[\,h_1(p)]\big] = H\big[\,h_0(p) + \Phi[\,q]\big] = libre$. También recomienda comenzar usando un valor aleatorio para el offset, y si no se puede encontrar un offset válido, se hace un backtracking. Los offsets finales son los más fáciles de asignar pues corresponden a una única celda ocupada del grid original.

Para poder utilizar este algoritmo hay que conocer cuántas y cuáles son las celdas del grid original que están ocupadas, para poder calcular los tamaños de las tablas, los parámetros necesarios y los valores de la tabla de offset.

En el caso que nos ocupa, se ha de recorrer los puntos de la malla original y distribuirlos en el grid de simplificación y conseguir sus *cluster_id* o tripletas *(i, j, k)* que serán las claves usadas en este hash.

La parte de buscar los valores para la tabla de offset sugiere un proceso altamente secuencial como así se confirma en el siguiente artículo *[Alcantara2009]*.

"Real-Time Parallel Hashing on the GPU"

En este artículo Alcántara y su equipo relajan la condición de usar el mínimo espacio a usar y optan por conseguir una tabla de hash perfecta, en la que los datos se pueden acceder en tiempo constante.

Su algoritmo está enfocado a la construcción rápida y acceso rápido de las tablas hash, a velocidad interactiva usando de media un 40 % más del espacio útil.

De los diferentes esquemas de hash existentes el artículo propone un método híbrido: primero se usa un esquema de hashing perfecto FKS de *[Fredman1984]* adaptado que es sencillo y rápido pero no eficiente en ocupación. Después se usa el método de reciente desarrollo llamado hash de cuco (cuckoo hashing) *[Pagh2001]*. Hash de cuco es un algoritmo de hashing perfecto de 'selección múltiple' que consigue altas tasas de ocupación a costa de un mayor tiempo de construcción. El artículo también explica cómo este algoritmo ha sido llevado a la tarjeta gráfica.

En el algoritmo de cuco se usa un número pequeño $d \geq 2$ de funciones hash con sus respectivas d sub-tablas. Cada bucket de la tabla hash guarda un único dato. Suponiendo un cuco hash de 2 sub-tablas, un elemento mira en la primera tabla si su posición está libre, si no lo está, mira en la segunda. Si su posición en la segunda tabla no está libre, vuelve a la primera y expulsa al elemento que está ocupando su posición y se coloca él. El elemento que ha sido "echado de su nido" (de ahí el nombre del algoritmo) intenta meterse en la segunda tabla. Si no puede, echa al que está allí y se coloca él. Y así iterativamente hasta llevar al número máximo de iteraciones.

En el artículo, Alcántara usa 3 tablas hash, de tamaño $n(1+\gamma)/3$ con γ suficientemente grande, concretamente sus tablas son de $3 * 192 = 576$ para guardar $n = 512$ elementos.

El procedimiento explicado está muy orientado a CUDA, pero los pasos que se siguen son:

- primero agrupar los elementos en buckets de 512 elementos como mucho, pero con una media de ocupación de 409 elementos, 80 % de ocupación, para dejar espacio para el hash cuco, calculando la posición donde se guardarán;
- después las parejas (clave, valor) se guardan en un único búfer en las posiciones anteriormente calculadas, donde cada bucket tendrá 576 posiciones (3 tablas cuco de 192 elementos);
- ahora, cada bucket se procesa independientemente usando tres funciones hash diferentes g_1, g_2, g_3 para cada una de las tres tablas T_1, T_2, T_3 colocando los elementos como se ha descrito anteriormente. Si se hacen demasiadas iteraciones, 25, echando a elementos de las tablas se generan de nuevo las tres funciones de hash;

- finalmente las tablas T_1 de todos buckets se copian consecutivamente al buffer final, seguidas de las T_2 y de las T_3. De esta manera se pueden hacer búsquedas paralela concentradas en la misma área en lugar de recorrer toda la tabla, según el artículo. Junto con cada bucket se guardan las semillas para crear las funciones g_1, g_2, g_3 de cada bucket.

Como función hash para el primer paso, el artículo usa simplemente la función $mod\ NumBuckets$, con $NumBuckets = \lceil n/409 \rceil$. Si esta falla, entonces usa la función $h(k) = [\ (\ c_0 + c_1 \cdot k) mod\ 1900813] mod\ NumBuckets$, con c_0 y c_1 números aleatorios. Hay que hacer notar que tanto 409 como 1900813 son números primos.

Las funciones de hash cuco g_1, g_2, g_3 se construyen de esta manera $g_i(k) = [\ (\ c_{i0} + c_{i1} \cdot k) mod\ 1900813] mod\ 192$. Las seis constantes c_{ij} se generan haciendo XOR de un único número aleatorio con diferentes constantes fijas. Empíricamente hacen 5,5 iteraciones por las tres tablas para tener todos los elementos colocados. Sólo unos pocos buckets no han podido contener a los elementos que les tocaban, es decir han realizado más de 25 iteraciones colocando y echando elementos, pero después de crear g_1, g_2, g_3 de nuevo ya se han podido colocar.

El factor de carga de este hash es 409 / 576, es decir un 40,8 % del espacio está desocupado.

Para recuperar los elementos primero se calcula $h(k)$ para saber a qué bucket acceder. Después a partir de la semilla guardada se crean de nuevo las tres funciones de hash g_1, g_2, g_3 y se acceden a las tablas. Si no está en la primera tabla, estará en la segunda y si no, en la tercera. Así obtenemos la pareja *(clave, valor)* con sólo dos, tres o cuatro accesos.

Como en el artículo anterior, antes de crear este hash hay que obtener la lista con celdas del grid original que están ocupadas, para poder calcular los tamaños, repartir los elementos y generar las funciones de hash de cuco.

El artículo propone que la media de ocupación de los buckets sea de 409 para dejar espacio al hash cuco. Pero dentro de cada bucket ya dispone de 576 - 512 entradas vacías, un 12,5% de espacio libre. Una duda que surge es si se pueden llenar más los buckets sin perjudicar el éxito del hash cuco.

Se ha podido observar que los ejemplos presentados en los artículos usan tamaños de grid hasta 1024^3, excepto un único ejemplo en el artículo "Perfect Spatial Hashing" *[Lefebvre2006]* pero con una ocupación muy baja.

La siguiente tabla muestra los requerimientos de memoria y ocupación de este algoritmo guardando en cada celda: matriz de la función de error cuádrica (10 floats) + acumulación de vértices (4 floats)

+ vértice óptimo (4 floats) + indices (1 entero 32 bits) = 76 bytes. Se ha usado el modelo lucy de 14.027.872 puntos y 28.055.742 triángulos.

Tamaño grid	Celdas ocupadas	Num buckets (ocup / 409)	Buckets * 576 elementos	MBytes nivel	MBytes acumulados
64^3	5.437	14	8.064	0,584	0,584
128^3	22.784	56	32.256	2,338	2,922
256^3	90.776	222	127.872	9,268	12,190
512^3	351.329	859	494.784	35,862	48,052
1.024^3	1.317.591	3.222	1.855.872	134,512	182,564
2.048^3	4.555.555	11.139	6.416.064	465,031	647,596
4.096^3	10.287.739	25.154	14.488.704	1.050,130	1.697,726
1 punto/celda	14.027.872	34.298	19.755.648	1.431,875	3.129,601

Tabla 4: requerimientos de memoria usando el hash híbrido de Alcántara a diferentes nivel de resolución.

La pregunta que aparece es si se podrán usar grid más grandes de 1024^3 para poder alcanzar una resolución mayor.

En el momento de estudiar el artículo de Schaefer "Adaptive Vertex Clustering Using Octrees", también se estudió este otro: "Simplifying Surfaces with Color and Texture using quadric error metrics" *[Garland1998]* que extiende su función de error cuádrica de $\mathbb{R}^3$ a $\mathbb{R}^n$, con es $n = 3 + k$, y k siendo el número de las componentes del color del vértice, o el número de coordenadas de textura o el número de atributos a simplificar.

El esquema de tres etapas que se sigue en el algoritmo de simplificación también facilita que, una vez el algoritmo de simplificación siguiendo el criterio geométrico esté listo, sea fácilmente adaptable para usar una combinación de simplificación según geometría y atributos definidos en los vértices siguiendo el artículo *[Garland1998]* o ponderando la función de error cuádrica de la geometría y la de los atributos.

3. Desarrollo técnico

Para realizar este proyecto se ha desarrollado una aplicación para implementar y probar el algoritmo de simplificación de mallas en sus diferentes etapas.

En éste apartado se dividirá en dos partes. En la primera se detallarán los requerimientos de esta aplicación, su diseño y en la segunda parte se explicarán las diferentes etapas del algoritmo de simplificación que se han seguido y su correspondiente pseudo-código.

3.1. Requerimientos

Los requerimientos funcionales de la aplicación son estos:

- lectura y visualización de modelos;
- selección del nivel de simplificación por parte del usuario;
- simplificación del modelo original y visualización del modelo simplificado;
- escritura del modelo.

En cuanto a los requerimientos no funcionales el algoritmo tendrá que:

- ser multiplataforma: que funcione tanto en MS Windows como en Linux;
- ser rápido: haga uso de las plataformas "multicore" actuales;
- ser robusto: en el anexo se puede encontrar una lista de los ejemplos usados para validar el algoritmo;
- hacer un uso responsable de la memoria;
- de fácil adaptación a otros programas, por ejemplo mediante el mecanismo de librería.

3.2. Diseño

Formato de los modelos:

A la hora de buscar modelos para probar el algoritmo buena parte de ellos se han encontrado en formato PLY, **Polygon File Format** o también conocido como **Stanford Triangle Format**. Éste es el formato que se ha escogido para leer modelos con la aplicación desarrollada. Los modelos que no estaban en este formato, se han convertido a PLY. Para la lectura y escritura en este formato se ha adaptado la librería desarrollada por Greg Turk en 1998 en Georgia Institute of Technology *[Gatech2011]*. Para la escritura Además de adoptar este formato también se ha implementado el formato de mallas ascii para GiD *[GiD2011]*, programa pre y postprocesador para poder diagnosticar y verificar las simplificaciones obtenidas.

Arquitectura:

Como la aplicación no es muy compleja y que la parte de lectura y escritura son bastante sencillos se ha seguido el patrón de arquitectura de dos capas:

- *capa de presentación*: que hará toda la interacción con el usuario y la lectura y escritura de los modelos
- *capa de dominio*: que se encargará de visualización de la escena, manejar y simplificar los objetos

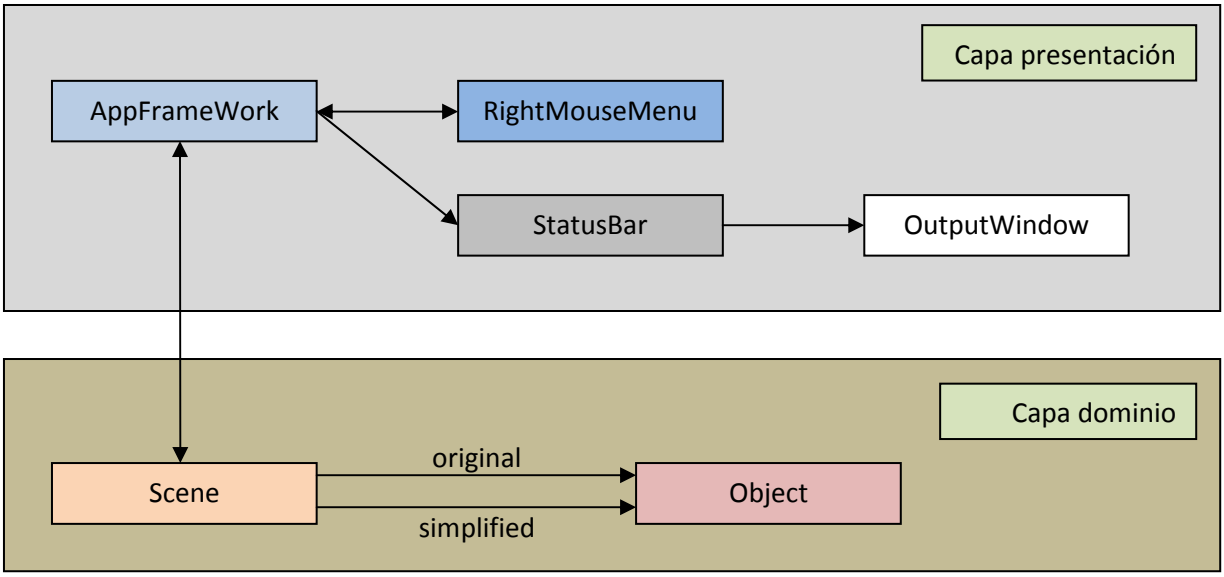


Figura 2: capas de la aplicación desarrollada

En la capa de presentación:

- *clase AppFrameWork*: se encarga de crear la ventana principal, controlar los eventos de teclado y de ratón, y de leer y escribir los modelos
- *clase RightMouseMenu*: se encarga de crear y tratar las diferentes entradas del menú del botón derecho del ratón, filtrando las entradas del usuario cuando sea necesario y hacer las llamadas correspondientes
- *clase StatusBar*: se encarga de mostrar y gestionar los mensajes de texto y la pseudo barra de avance, en la parte inferior de la ventana principal; también se encarga, si el usuario lo desea, de mostrar y esconder la ventana de salida, OutputWindow, que muestra mensajes adicionales y de diagnóstico.

En la capa de dominio:

- *clase Scene*: se encarga de montar y visualizar la escena, con sus luces, matrices de proyección y visualización, tanto del objeto original como del objeto simplificado
- *clase Object*: contiene las listas de coordenadas, triángulos y líneas que conforman el objeto así como una serie de flags para permitir la visualización de sus características; es en esta clase donde se han implementado las diferentes versiones del algoritmo de simplificación.

El aspecto de la aplicación es éste:

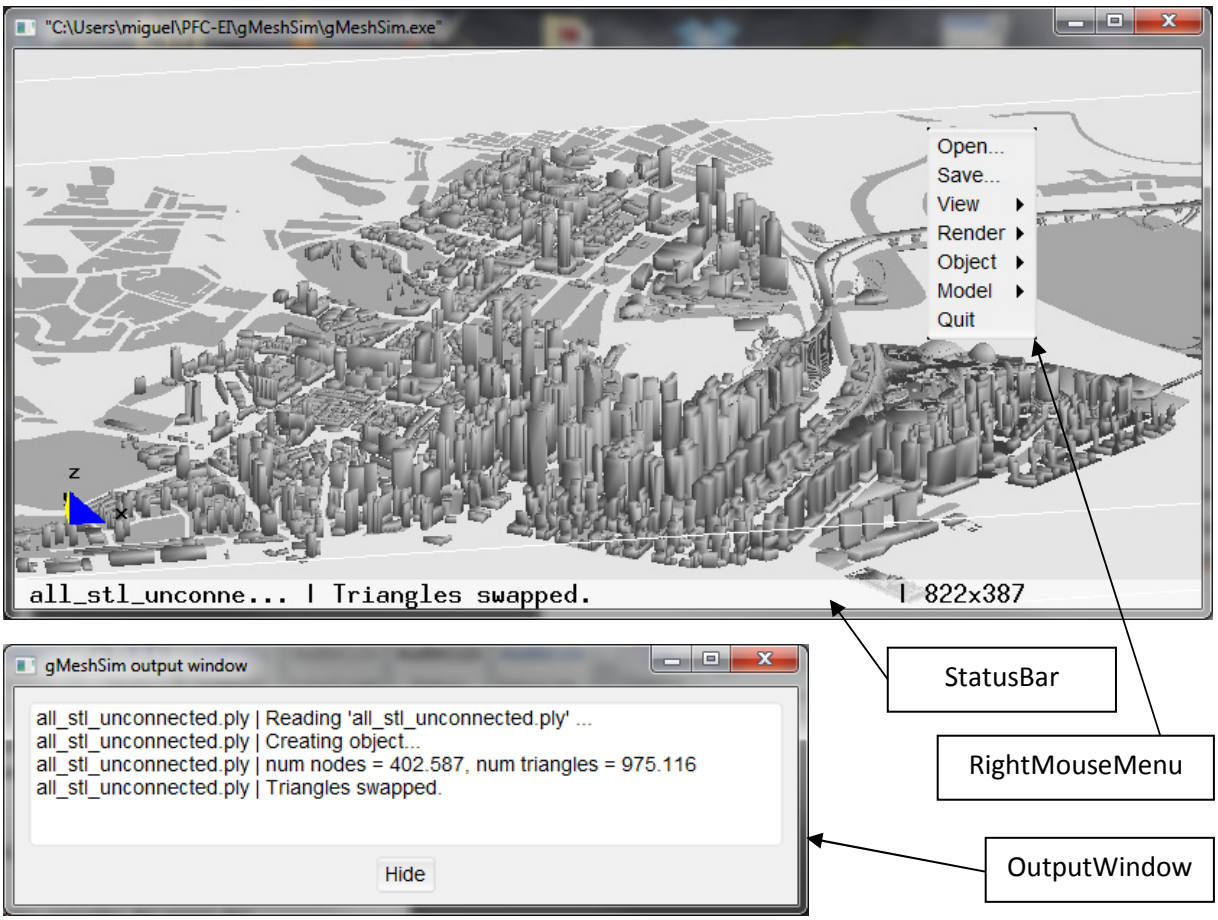


Figura 3: ventana principal de la aplicación y ventana de salida de texto con mensajes de diagnóstico.

Casos de uso:

La aplicación es bastante sencilla en cuanto a casos de uso, que serían estos:

1. el usuario carga el modelo y éste se visualiza, mostrándole información de éste;
2. el usuario escoge diferentes propiedades de visualización tanto de la escena como del objeto:
   - tipo de renderizado: display lists, vertex arrays o vertex buffer objects,
   - seleccionar la visualización con una o dos luces,
   - ver el modelo con triángulos llenos o como modelo de alambres,
   - seleccionar si se pintan los triángulos, las líneas o ambos,
   - si se filtran los triángulos con la normal apuntando al usuario, los que no apunten al usuario o ninguno, también conocido como cullling,
   - mostrar o esconder las aristas de la caja contenedora del modelo,
   - mostrar o esconder las aristas de las celdas usadas para simplificar el modelo,

- volver a las opciones iniciales de visualización (reset);

3. el usuario selecciona la opción de invertir el sentido de las normales, y se invierten las normales del modelo;
4. el usuario selecciona la opción de invertir el sentido de los triángulos, y éston invierten su conectividad;
5. el usuario escoge el nivel de simplificación del modelo, se le pide confirmación, éste se simplifica mostrándole información de diagnóstico y al final del proceso el modelo simplificado se visualiza;
6. el usuario sale de la aplicación.

Tecnología:

Para cumplir con el requisito de portabilidad, la aplicación se ha desarrollado en C++ usando OpenGL para la visualización 3D.

Para la creación de la ventana gráfica, del menú de la derecha de ratón y la recogida de los eventos de ratón y teclado se ha usado la librería GLUT.

Para las ventanas de diálogo que usa la aplicación:

- ventanas modales de texto
- ventanas para seleccionar el archivo a leer o escribir
- ventana de salida de texto

se han valorado tres librerías: glui, Qt y fltk escogiendo finalmente ésta por ser más completa que la glui, y más fácil de usar y compilar en las diferentes plataformas que Qt.

3.3. Implementación

Etapas seguidas

Los pasos que se han seguido para la implementación y que se describen son estos:

- *FullUC*: primera versión serial con el algoritmo de DeCoro *[DeCoro2007]*,
- *FullUC*: paralelización de esta primera versión con OpenMP,
- *HashUC*: segunda versión con el algoritmo de almacenamiento hash de Alcantara *[Alcantara2009]*,
- *HashUC*: paralelización de esta segunda versión con OpenMP.

Primera versión (FullUC): implementación agrupamiento uniforme con grid lleno

Como ya se ha comentado en los *Antecedentes* se ha implementado la versión de DeCoro *[DeCoro2007]*, para la simplificación uniforme agrupando vértices.

La simplificación consiste en 'meter' la malla en un grid uniforme con un número de celdas especificado por el usuario para cada eje y de este grid sacar la malla simplificada. La información de todos los vértices que caen en la misma celda se acumula de una determinada manera, en este caso se acumulan por un lado las funciones de error cuádricas y por otro los vértices para que a la hora de simplificar se pueda encontrar un único vértice representativo de la celda ya sea minimizando la función de error cuádrica o haciendo la media de los vértices acumulados.

El algoritmo de DeCoro sigue estos tres pasos:

1. *creación del mapa de cuádricas*: donde se acumulan las diferentes funciones de error cuádricas en las celdas de un grid 3d uniforme;
2. *cálculo de representantes óptimos*: aquí se recorren las celdas del grid3d y en aquellas donde hay puntos acumulados se calcula el punto que minimiza la función de error cuádrica o, si no es posible conseguirlo, se usa la media de los vértices acumulados como vértice representativo;
3. *simplificación de la malla:* éste es el paso que simplifica la malla, para cada triángulo se busca en qué celdas caen sus vértices y si las tres son distintas se genera un triángulo a pintar.

Para adaptar este algoritmo a la cpu se ha seguido el símil de las tarjetas gráficas y la nomenclatura que DeCoro usa en su artículo:

- *Render Texture*: texturas donde almacena la información del grid: funciones de error cuádrica y vértices acumulados en la celda;
- *CPUfloat2, CPUfloat3 y CPUfloat4*: tipos de datos similares a los float2, float3 y float4 del lenguaje CUDA, OpenCL, lenguajes de sombreado;
- *método ping-pong*: como se recomienda al programar las tarjetas gráficas no se usa una misma tabla, textura, para leer y para escribir si no que se lee de una tabla y se escribe en otra.

Siguiendo estas reglas, el algoritmo adaptado de DeCoro sería este:

Algoritmo 1: implementación simplificación uniforme básico

```
entrada:
        lst_points: lista de puntos de la malla original;
        lst_triangs: lista de triángulos de la malla original;
        bbox: caja contenedora del modelo original;
        dim_x, dim_y, dim_z: dimensiones del grid usado para simplificar la malla;
salida:
        lst_points_out: lista de puntos de la malla simplificada;
        lst_triangs_out: lista de triángulos de la malla simplificada;
local:
        acc_coords, quadric_A1, optimal_coords: RenderTexture2D< CPUfloat4>;
        quadric_A2, quadric_B: RenderTexture2D< CPUfloat3>;
        grid: DecimationGrid; // con la información del grid de soporte usado para simplificar
begin:
        grid.crear( bbox, dim_x, dim_y, dim_z);
        Initialize( acc_coords, quadric_A1, quadric_A2, quadric_B, optimal_coords);
```

```
        CreateQuadricMap( acc_coords, quadric_A1, quadric_A2, quadric_B,
                                lst_puntos, lst_triangulos, grid);
        FindOptimalPositions( optimal_coords,
                                acc_coords, quadric_A1, quadric_A2, quadric_B, grid);
        DecimateMesh( lst_points_out, lst_triangs_out, lst_points, lst_triangs, optimal_coords, grid);
end.
```

La clase *DecimationGrid* contiene la información del grid de soporte usado para simplificar la malla y se usa para:

- a partir de un punto calcular en qué celda cae, identificada por si *cluster_id*. A partir del punto se encuentra la tripleta *(i, j, k)* según los ejes *(x, y, z)* que identifica la celda y el *cluster_id* es *(k * dim_y + j) * dim_x + i*;
- dado un punto y un cluster_id saber si está dentro de la celda o no;
- dado un cluster_id devolver las coordenadas de la celda.

La clase *RenderTexture2D< Tipo>* simula una textura 2D cuadrada y potencia de dos de una tarjeta gráfica donde cada *texel* es del tipo *Tipo*. Esta clase se usa para leer y acumular la información de una celda del grid 3D identificada por su *cluster_id*, con lo que *RenderTexture2D* hace la conversión del *cluster_id* a las coordenadas *(s, t)* de la textura.

El refinamiento del primer paso del algoritmo de DeCoro, la creación del mapa de cuádricas sería este:

Algoritmo 1.1: creación del grid con las funciones de error cuádricas acumuladas en cada celda

```
entrada:
        lst_points: lista de puntos de la malla original;
        lst_triangs: lista de triángulos de la malla original;
        bbox: caja contenedora del modelo original;
        grid: grid de soporte usado para simplificar la malla;

salida:
        acc_coords, quadric_A1, optimal_coords: RenderTexture2D< CPUfloat4>;
        quadric_A2, quadric_B: RenderTexture2D< CPUfloat3>;
begin:
        for each triang in lst_triags do
                calculate_plane_coeficients( triang, a, b, c, d);
                for each node in triag do
                        Point p = lst_points[ node];
                        int idx_p = grid.get_cluster_id( p);
                        // construir la qef
                        CPUfloat4 dataA0( a * a, a * b, a * c, b * b);
                        CPUfloat3 dataA1( b * c, c * c, d * d), dataB( a * d, b * d, c * d);
                        CPUfloat4 acc( p, 1.0); // acc = ( p.x, p.y, p.z, 1.0);
                        quadric_A1 → add( idx_p, dataA0);
                        quadric_A2 → add( idx_p, dataA1);
                        quadric_AB → add( idx_p, dataB);
```

```
                    acc_coords → add( idx_p, acc);
                end for
        end for
end.
```

El detalle del segundo paso es este:

Algoritmo 1.2: búsqueda de las posiciones óptimas

```
entrada:
        acc_coords, quadric_A1, optimal_coords: RenderTexture2D< CPUfloat4>;
        quadric_A2, quadric_B: RenderTexture2D< CPUfloat3>;
        grid: grid de soporte usado para simplificar la malla;
salida:
        optimal_coords: RenderTexture2D< CPUfloat4>;
begin:
        for each cluster_id in grid do
                CPUfloat4 cur_point = acc_coords → get( cluster_id);
                // la componente w contiene el número de puntos acumulados en la celda
                if cur_point.w >= 1 then
                        CPUfloat4 optimal_position; // representante de la celda
                        CPUfloat4 dataA0 = quadric_A1 → get( cluster_id);
                        CPUfloat3 dataA1 = quadric_A2 → get( cluster_id);
                        CPUfloat3 dataB = quadric_B → get( cluster_id);
                        CPUMatrix quadric(      dataA0.x, dataA0.y, dataA0.z, dataB.x,
                                                dataA0.y, dataA0.w, dataA1.x, dataB.y,
                                                dataA0.z, dataA1.x, dataA1.y, dataB.z,
                                                0.0, 0.0, 0.0, 1.0);
                        if quadric.determninant() > SINGULAR_THRESHOLD then
                                CPUMatrix inv = quadric.get_inverse();
                                optimal_position = inv * ( 0.0, 0.0, 0.0, 1.0);
                        else
                                // no se puede invertir,
                                // el representante es la media de los vértices acumulados
                                optimal_position = cur_point / cur_point.w;
                        end if
                        optimal_coords → set( cluster_id, optimal_position);
                end if
        end for
end.
```

En el tercer paso, DeCoro filtra los triángulos colapsados, aquellos cuyos vértices no caen en tres celdas diferentes, mientras deja pasar los no colapsados para que la tarjeta gráfica los pinte en pantalla o en la textura de destino. El objetivo del proyecto es recuperar estos triángulos y, por tanto, lo que se hace es guardar los triángulos y los vértices en sendas listas.

Algoritmo 1.3: simplificar la malla original

```
entrada:
        lst_points: lista de puntos de la malla original;
        lst_triangs: lista de triángulos de la malla original;
```

```
        optimal_coords: RenderTexture2D< CPUfloat4>;
        grid: grid de soporte usado para simplificar la malla;
salida:
        lst_points_out: lista de puntos de la malla simplificada;
        lst_triangs_out: lista de triángulos de la malla simplificada;
begin:
        for each triangs in lst_triangs do
                Point p1 = lst_points[ triang.v1()];
                Point p2 = lst_points[ triang.v2()];
                Point p3 = lst_points[ triang.v3()];
                int idx_p1 = grid.get_cluster_id( p1);
                int idx_p2 = grid.get_cluster_id( p2);
                int idx_p3 = grid.get_cluster_id( p3);
                // si los vértices están en tres celdas diferentes, entonces guardamos el triángulo
                if ( idx_p1 != idx_p2) && ( idx_p2 != idx_p3) && ( idx_p3 != idx_p1) then
                        // el algoritmo original de DeCoro pinta el triángulo
                        CPUfloat4 sim_p1 = optimal_coords → get( idx_p1);
                        CPUfloat4 sim_p2 = optimal_coords → get( idx_p2);
                        CPUfloat4 sim_p3 = optimal_coords → get( idx_p3);
                        int new_idx_p1 = lst_points_out → add( sim_p1);
                        int new_idx_p2 = lst_points_out → add( sim_p2);
                        int new_idx_p3 = lst_points_out → add( sim_p3);
                        lst_triangs_out → add( new_idx_p1, new_idx_p2, new_idx_p3);
                end if
        end for
end.
```

Primera versión: problemas y mejoras

Inmediatamente se ha mejorado esta primera versión con:

- eliminación de los triángulos repetidos
- eliminación y re enumeración de los puntos
- recuperación de los triángulos que se colapsan en líneas, y su eliminación de repetidos.

Para eliminar los triángulos repetidos y los nodos repetidos, una de las primeras implementaciones fue guardar todos los triángulos, después ordenarlos y quitar los repetidos.

Para recuperar las líneas colapsadas, se ha seguido la misma idea pero marcando de forma diferente las líneas provenientes de triángulos colapsados de las líneas provenientes de los triángulos no colapsados. De esta manera, a la vez que se eliminan los repetidos, también se eliminan las líneas provenientes de los triángulos no colapsados.

En principio el algoritmo parecía funcionar bien, y la decisión de incorporar la extracción de líneas fue muy acertada como se puede ver en las siguientes figuras:

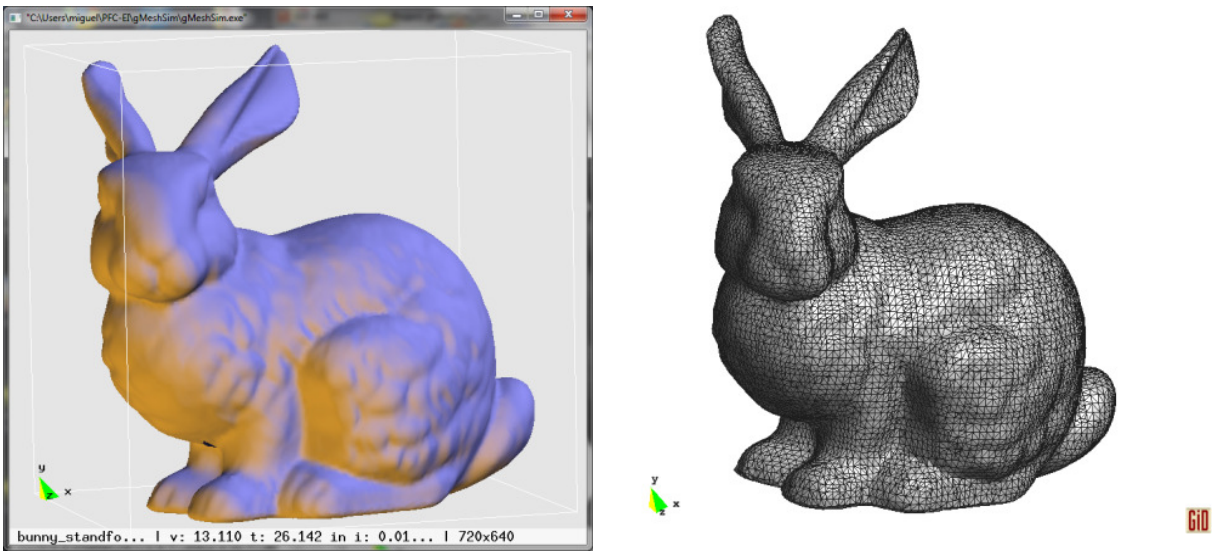


Figura 4: simplificación usando un grid de 64 x 64 x 64 celdas del conejo de Stanford con unos 70k triángulos.

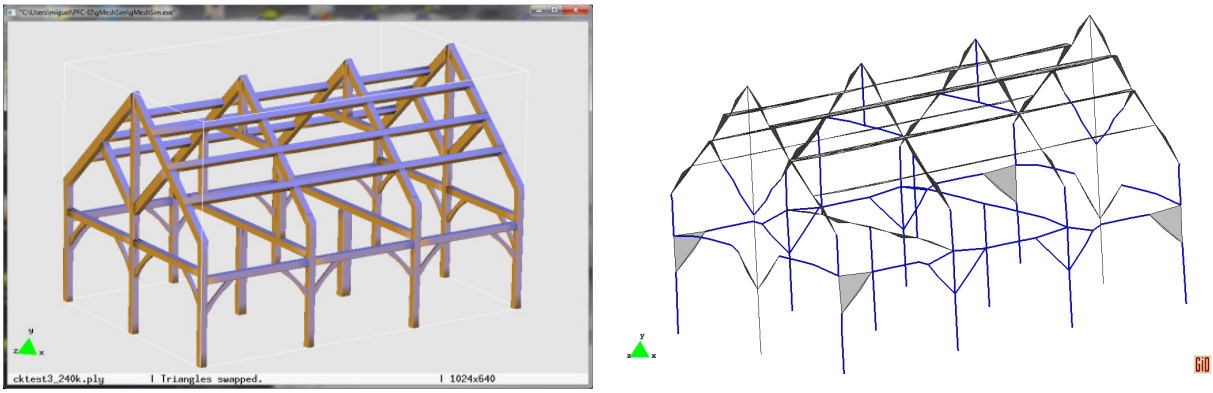


Figura 5: simplificación usando un grid de 12 x 11 x 16 celdas donde los triángulos originales colapsados son recuperados, cuando el algoritmo original los hubiese descartado.

Pero al probar más ejemplos, incluso el más sencillo de la tetera, aparecieron algunos problemas. Concretamente el algoritmo de búsqueda del representante óptimo podía posicionar este punto fuera de la celda, como se puede comprobar en las figuras 6 y 7.

Los artículos de Garland *[Garland1997], [Garland1998], [Heckbert1999],* Lindstrom *[Lindstrom2000*] y Scheffaer *[Ju2002]* y *[Schaefer2002]* proponen diferentes alternativas para solucionar este problema:

1. proyectar la solución de la QEF dentro del espacio de soluciones que incluye el centro de la celda, calculando los vectores y valores propios de la matriz,
2. usar la factorización QR que se muestra más estable numéricamente,
3. si el punto cae fuera de la celda, escoger la media de los vértices acumulados en la celda.

Para este proyecto se ha escogido la tercera opción, dejando las dos primeras para una posterior valoración entre el coste computacional y el de memoria o se puede compensar con una resolución más alta del grid o una profundidad mayor del octree.

Un hecho interesante es que la mayoría de los artículos, incluido el de Shaeffer *[Schaefer2003]*, normalizan el modelo al rango [0, 1]^3 "sin perder generalidad". Siguiendo esta tendencia el algoritmo implementado también escala el modelo al rango [-1, 1]^3 antes de simplificar pero lo "desescala" de nuevo después de simplificar. De esta manera se ahora el uso epsilons dependientes del modelo y, de hecho, cambia la simplificación.

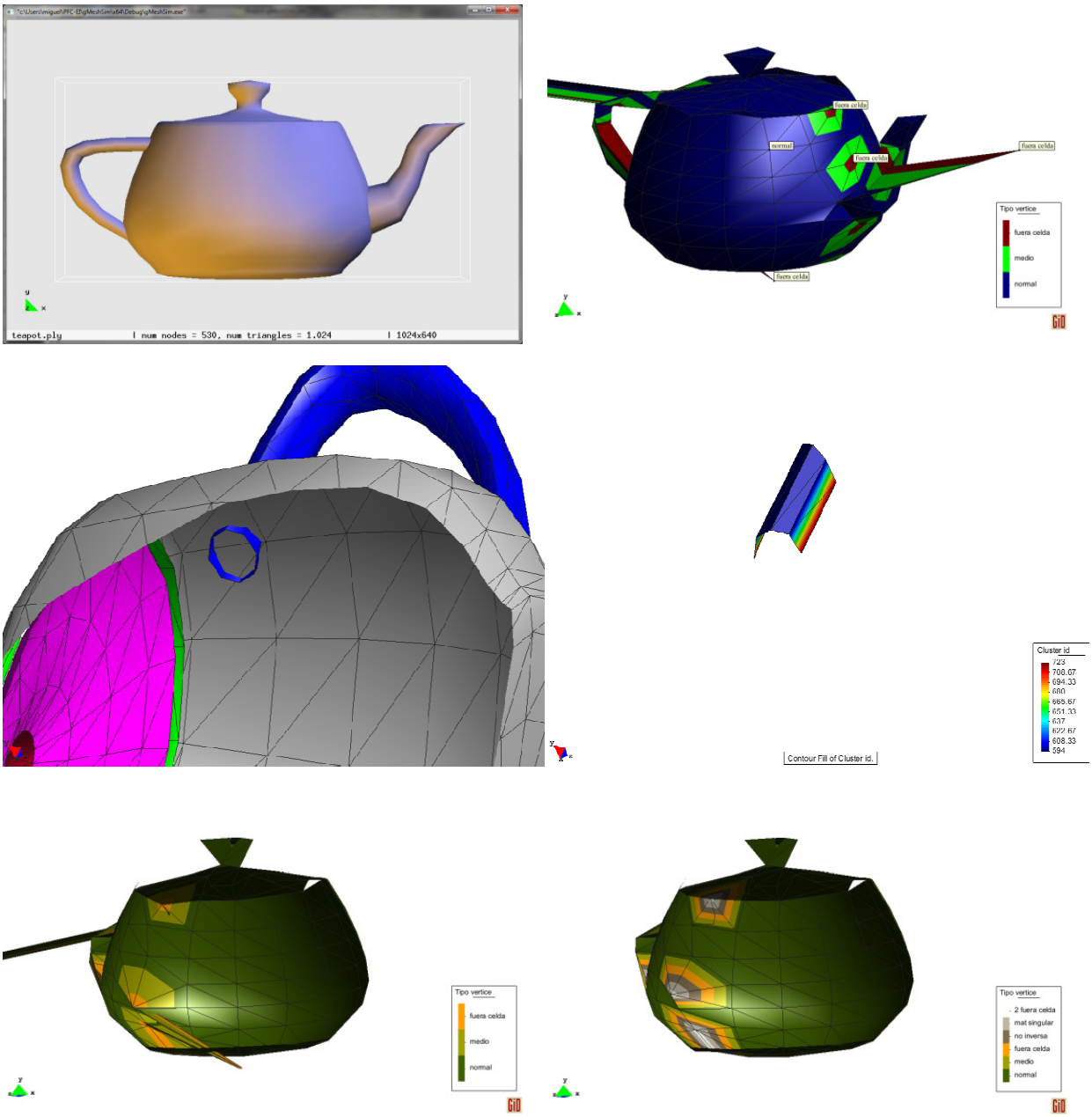


Figura 6: simplificación de la tetera con un grid de 10 x 10 x 10 celdas. Arriba a la izquierda de puede ver el modelo original, a la derecha cómo el vértice que minimiza la QEF de las celdas con la superficie localmente plana se sale de la celda. En el medio se puede observar el detalle de el asa de la tetera que sólo presenta una curvatura en una dirección. en las imágenes inferiores se puede observar el interior de la tetera, izquierda, y a la derecha la corrección usando la media de los vértices.

El siguiente ejemplo muestra el problema con más dramatismo:

Figura 7: simplificación de la estatua 803_Neptune de más de 4 millones de triángulos usando un grid de 59 x 99 x 44 celdas. El problema aparece en algunos puntos del modelo, pero sobretodo en la base de la estatua. El fondo gris de la imagen de la derecha son los triángulos grises 'simplificados' de la base de la estatua.

Otro problema que ha aparecido es el de los slip-overs, el de los triángulos que al simplificarse salen de la superficie porque sus nodos se han movido. El efecto que aparece es que las normales aparecen volteadas, cambiadas de sentido. Se ha añadido una corrección de normales para que en la extracción de la malla simplificada la normal de los nuevos triángulos apunten al mismo semiplano que la de los triángulos originales.

Primera versión (FullUC): paralelización

Como ya se ha comentado, el algoritmo es fácilmente paralelizable:

- *creación del mapa de cuádricas*: se paraleliza el bucle externo de triángulos con un simple *#pragma omp parallel for*
- *cálculo de representantes óptimos*: se paraleliza el bucle externo que recorre todas las celdas del grid con un simple *#pragma omp parallel for*
- *simplificación de la malla:* este es un poco más complicado. Se puede paralelizar el bucle externo de triángulos con *#pragma omp parallel for* pero para evitar que todos los threads escriban en la misma lista de triángulos y puntos, se usa una lista para cada thread que después del bucle se fusionan:

Algoritmo 1.3omp: simplificar la malla original con OpenMP

```
entrada:
        lst_points: lista de puntos de la malla original;
        lst_triangs: lista de triángulos de la malla original;
        optimal_coords: RenderTexture2D< CPUfloat4>;
        grid: grid de soporte usado para simplificar la malla;
salida:
        lst_points_out: lista de puntos de la malla simplificada;
        lst_triangs_out: lista de triángulos de la malla simplificada;
begin:
        create_local_lists( local_points_out, local_triangs_out, omp_get_max_threads());
#pragma omp parallel
        {
        int mi_id = omp_get_thread_num();
#pragma omp for
        for each triangs in lst_triangs do
                Point p1 = lst_points[ triang.v1()];
                Point p2 = lst_points[ triang.v2()];
                Point p3 = lst_points[ triang.v3()];
                int idx_p1 = grid.get_cluster_id( p1);
                int idx_p2 = grid.get_cluster_id( p2);
                int idx_p3 = grid.get_cluster_id( p3);
                // si los vértices están en tres celdas diferentes, entonces guardamos el triángulo
                if ( idx_p1 != idx_p2) && ( idx_p2 != idx_p3) && ( idx_p3 != idx_p1) then
                        // el algoritmo original de DeCoro pinta el triángulo
                        CPUfloat4 sim_p1 = optimal_coords → get( idx_p1);
                        CPUfloat4 sim_p2 = optimal_coords → get( idx_p2);
                        CPUfloat4 sim_p3 = optimal_coords → get( idx_p3);
                        // local lists
                        int new_idx_p1 = local_points_out[ mi_id] → add( sim_p1);
                        int new_idx_p2 = local_points_out[ mi_id] → add( sim_p2);
                        int new_idx_p3 = local_points_out[ mi_id] → add( sim_p3);
                        local_triangs_out[ mi_id] → add( new_idx_p1, new_idx_p2, new_idx_p3);
                end if
        end for
        }
        lst_points_out = join_local_lists( local_points_out);
        lst_triangs_out = join_local_lists( local_triangs_out);
end.
```

Primera versión (FullUC): tiempos, problemas detectados y corregidos

Aquí se muestra una primera tabla de tiempos al simplificar el modelo lucy de 14.027.872 puntos y 28.055.742 triángulos:

	Single	4 cores	Speed-up
Inicialización	0,967	0,757	1,277
P1: crear mapa QEF	3,703	0,972	3,810
P2: buscar optimo	0,163	0,066	2,470
P3: simplificar malla	3,520	2,437	1,444
Total	**8,353**	**4,232**	**1,974**

Tabla 5: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 1*: QuadCore Q9550, extrayendo triángulos y líneas

Como se puede observar, el speed-up que obtenemos es más bien modesto. Las partes que menos escalaban eran la inicialización de las tablas y la de la simplificación de malla. De la parte de la inicialización no se puede sacar más, pues está limitado por el ancho de banda de la memoria, pero la parte de simplificación de malla parece que el factor limitante es la ordenación y eliminación de triángulos y líneas repetidas. Quitando el cálculo de la líneas se puede observar que el tiempo mejora mucho:

	Single	4 cores	Speed-up
Inicialización	0,807	0,757	1,066
P1: crear mapa QEF	3,717	0,985	3,774
P2: buscar optimo	0,160	0,067	2,388
P3: simplificar malla	1,627	0,623	2,613
Total	**6,310**	**2,433**	**2,593**

Tabla 6: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 1*: QuadCore Q9550 extrayendo sólo triángulos

Se ha cambiado las listas donde se guardaban y calculaban los triángulos y las líneas únicas por tablas hash. Este cambio también modifica la numeración de los puntos únicos, pero es aceptable. Se ha seguido con el mecanismo de que cada thread tuviese su hash de trabajo que luego se fusionan en una única tabla hash.

Como función hash de estas tablas simplemente se ha usado el módulo*: h(k) = k mod (NumberOccupiedCells / NumThreads)* tanto para el hash de líneas, el de triángulos. Para el de nodos el módulo usado es este: *h(k) = k mod (NumberOccupiedCells / NumThreads / 5*).

Cada entrada de estas tablas hash es una lista encadenada de buckets de 6 elementos: de media habrán 6 líneas o triángulos incidentes a un punto.

Para asegurar que un único thread escriba en la entrada de la tabla hash correspondiente se ha usado un lock para cada entrada de la tabla.

Estos son los tiempos conseguidos con los semáforos:

	Single (noomp)	4 cores	Speed-up
Inicialización	0,844	0,762	1,108
P1: crear mapa QEF	3,786	0,994	3,809
P2: buscar optimo	0,167	0,061	2,738
P3: simplificar malla	2,567	2,158	1,190
Total	**7,364**	**3,975**	**1,853**

Tabla 7: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 1*: QuadCore Q9550 extrayendo sólo triángulos

Aunque el speed-up no es muy alto, los tiempos han mejorado respecto a la primera versión con hash de líneas y de triángulos. También hay que tener en cuenta que la versión serial es la versión del programa compilada sin OpenMP, pues los locks suponen un gran sobrecoste como se vió en seguida.

Evitar data-races al acumular las funciones de error cuádricas en el grid de simplificacion:

A la hora de crear el mapa QEF también se ha puesto un semáforo para evitar escrituras simultáneas, pero parece ser que no es tan crucial si en una misma celda caen suficientes puntos. Es un punto a estudiar en el futuro pues no se había detectado su falta hasta que se colocaron los semáforos en las tablas hash.

Al comenzar a usar semáforos en la parte de crear el mapa de cuádricas se han probado diferentes opciones como poner un semáforo para cada celda, lo que equivale a 16 millones de semáforos en un grid de tamaño 256^3 lo cual son muchos semáforos, o por cada n celdas. Se ha observado que la mejor opción es poner un semáforo cada 16 celdas.

A la evaluar los tiempos con y sin semáforos se ha visto que el sobrecoste de los lock usando MS Visual C++ es muy alto.

Para compararlo, en vez de trocear el bucle que recorre los triángulos para cada thread, se ha implementado una segunda versión donde cada thread recorre todos los triángulos pero sólo escribe en su parte del grid.

	Single (noomp)	4 cores	Speed-up
Inicialización	0,844	0,765	1,103
P1: crear mapa QEF	3,786	1,891	2,002
P2: buscar optimo	0,167	0,067	2,493
P3: simplificar malla	2,567	1,639	1,566
Total	**7,364**	**4,364**	**1,687**

Tabla 8: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 1*: QuadCore Q9550 + MS Visual Strudio 2008, con la versión de escritura restringida.

	Single (noomp)	Single (omp)	4 cores	Speed-up (omp)	Speed-up (noomp)
Inicialización	0,844	0,775	0,738	1,050	1,144
P1: crear mapa QEF	3,786	8,627	2,558	3,373	1,480
P2: buscar optimo	0,167	0,156	0,063	2,476	2,651
P3: simplificar malla	2,567	3,988	1,643	2,423	1,059
Total	**7,364**	**13,546**	**5,002**	**2,708**	**1,472**

Tabla 9: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 1*: QuadCore Q9550 + MS Visual Strudio 2008, con la versión de locks para cada 16 celdas.

En cambio las mismas tablas pero usando gcc 4.4.5 en Linux y en la misma plataforma

	Single (omp)	4 cores	Speed-up
Inicialización	0,438	0,368	1,190
P1: crear mapa QEF	3,658	1,990	1,838
P2: buscar optimo	0,137	0,100	1,37
P3: simplificar malla	3,049	1,153	2,644
Total	**7,282**	**3,611**	**2,017**

Tabla 10: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 1*: QuadCore Q9550 + gcc 4.4.5, con la versión de escritura restringida.

	Single (noomp)	Single (omp)	4 cores	Speed-up (omp)	Speed-up (noomp)
Inicialización	0,433	0,427	0,365	1,170	1,186
P1: crear mapa QEF	3,141	6,031	1,656	3,642	1,897
P2: buscar optimo	0,137	0,137	0,056	2,446	2,446
P3: simplificar malla	3,247	3,258	1,191	2,719	2,726
Total	**7,364**	**9,832**	**3,269**	**3,008**	**2,128**

Tabla 11: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 1*: QuadCore Q9550 + gcc 4.4.5, con la versión de locks para cada 16 celdas.

Para ver si el algoritmo escalaba bien se ha ejecutado la aplicación en dos plataformas más, simplificando el modelo lucy con un grid de 256^3 :

plataforma 2: Intel i7 920 con 4 cores e hyperthreading, y con Linux Ubuntu 10.04.2 LTS

plataforma 4: módulo con 2 procesadores Intel Xeon E5410 QuadCore 2.33 GHz y RedHat 5.1 ELS 64 bits.

	Single (noomp)	4 cores	Speed-up	4 cores + HT	Speed-up
Inicialización	0,269	0,121	2,223	0,118	2,280
P1: crear mapa QEF	3,027	1,128	2,684	0,972	3,114
P2: buscar optimo	0,108	0,033	3,273	0,027	4,000
P3: simplificar malla	3,230	1,066	3,030	1,042	3,100
Total	**6,550**	**2,350**	**2,787**	**2,158**	**3,035**

Tabla 12: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 2*: intel i7 920 + gcc 4.4.3, con la versión de locks para cada 16 celdas.

	Single-omp	2 cores	Speed-up	4 cores	Speed-up	6 cores	Speed-up
Inicialización	0,624	0,495	**1,261**	0,39	**1,600**	0,335	**1,863**
crear mapa QEF	7,883	3,993	**1,974**	2,029	**3,885**	1,531	**5,149**
buscar optimo	0,183	0,098	**1,867**	0,067	**2,731**	0,071	**2,577**
simplificar malla	4,059	2,34	**1,735**	1,475	**2,752**	1,354	**2,998**
Total	**12,749**	6,926	**1,841**	3,960	**3,219**	3,290	**3,875**

Tabla 13: tiempos al simplificar el modelo lucy con un grid de 256^3 en la *plataforma 4*: 2 x Xeon E5410 QuadCore + gcc 4.4.5, con la versión de locks para cada 16 celdas.

Poniendo estos números es una gráfica se puede apreciar la escalabilidad del algoritmo:

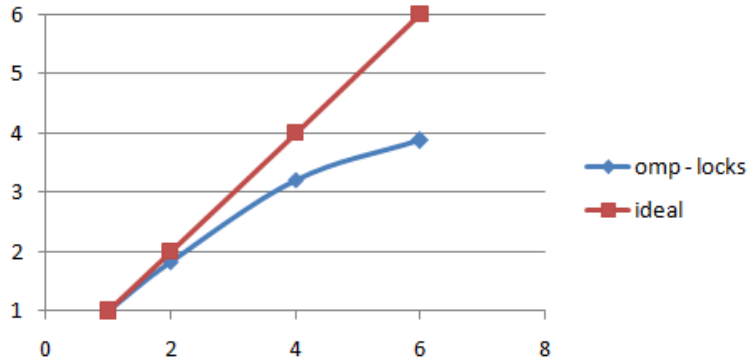


Figura 8: gráfica de escalabilidad en la *plataforma 4*: 2 x Intel Xeon E5410 QuadCore

Primera versión (FullUC): limitaciones

Este algoritmo, al guardar todo el grid lleno supone un consumo importante de memoria.

La siguiente tabla muestra el consumo de memoria para varios tamaños de grid, donde cada en celda se almacena: (10 reales de la matriz cuádrica + 4 reales para acumular puntos + 4 reales para guardar el representante óptimo) * 4 bytes = 72 bytes por celda:

Tamaño	Millones de celdas	MBytes
128^3	2,097	144
256^3	16,777	1.152
512^3	134,218	9.216
1024^3	1.073,742	73.728

Tabla 14: requerimientos de memoria según el tamaño del grid

La resolución de 256^3 no es suficiente pues en algunos modelos se pueden apreciar imperfecciones, como por ejemplo la mano de la estatua lucy:

Figura 9: ejemplo de imperfección no deseada al simplificar lucy con un grid de 256^3

Si bien es cierto que el grid no ha de ser potencia de dos, un grid de 406^3 ocupa ya 4,5 GB. La idea final es posibilitar el montar un octree sieguiendo el esquema propuesto por DeCoro con diferentes niveles de resolución precisamente para minimizar estos errores. Siguiendo con dicho enfoque, se guarda cada nivel pero las celdas tienen una 'probabilidad' de ser guardadas.

Se ha hecho un estudio de ocupación con diferentes modelos y tamaños de grid para observar cuan vacías están los diferentes niveles de un posible octree y si es posible guardar todas las celdas.

En la siguiente tabla se muestra la ocupación del modelo lucy para los diferentes niveles y su ocupación en memoria si se guardan todas las celdas (72 bytes):

Tamaño	Celdas ocupadas	% ocupación	Memoria (MB)
64^3	5.437	2,074%	0,373
128^3	22.784	1,086%	1,564
256^3	90.776	0,541%	6,233
512^3	351.329	0,262%	24,124
1024^3	1.317.591	0,123%	90,472
2048^3	4.555.555	0,053%	312,805
4096^3	10.287.739	0,015%	706,403

Tabla 15: requerimientos de memoria guardando sólo las celdas ocupadas del modelo lucy, con 14.027.872 puntos y 28.055.742 triángulos.

Parece factible crear niveles de grid de hasta 4096^3 de resolución y poder montar de esta manera un octree que pueda refinarse hasta una profundidad de 12.

Después de ver el éxito obtenido con el hash de líneas y triángulos, se han estudiado artículos sobre hash espacial, entre ellos: "Perfect spatial hashing" *[Lefebvre2006]* y "Real-Time Parallel Hashing on the GPU" *[Alcantara2009]*.

Segunda versión (HashUC): implementación con hash espacial de cuco

Como se ha comentado en los antecedentes, el algoritmo de cuco consiste en agrupar elementos en buckets de cómo mucho 512 elementos, después se guardan en un buffer las parejas *(clave, valor)* según esta distribución en un único buffer según esta agrupación. Seguidamente se procesa cada bucket, subdiviendo el bucket en tres tablas y buscando las correspondientes tres funciones de hash de cuco. Según el artículo de Alcántara, para que el hash de cuco pueda tener éxito hay que proporcionar espacio suficiente para asegurar el éxito de la creación del hash de cuco. El artículo propone usar tres tablas de 192 elementos, que suman 576 elementos por bucket. 576 elementos, de los que como mucho se usarán 512, pero que la media será de 409 elementos. Al final la tabla tendrá un sobrecoste del 576 / 409 = 40,83 % en memoria.

En la implementación realizada se ha obviado la creación de este buffer único y se trabaja directamente con la tabla hash inicial y en cada bucket no vacio, calculo el hash de cuco. En esta tabla hash se guardan las parejas *(clave, valor)*, que en el caso del proyecto serán:

- clave = *cluster_id* que para poder llegar a grids de más de 1024^3 será un entero de 64 bits,
- valor = matriz función de error cuádrica + vértices acumulados.

Esta tabla hash es la que se usa para acumular la información de las funciones de error cuádricas y los vértices de las celdas ocupadas, para encontrar los vértices óptimos y para simplificar la malla original, en vez del grid lleno que se usaba en la primera versión.

El esquema se puede observar en la siguiente figura:

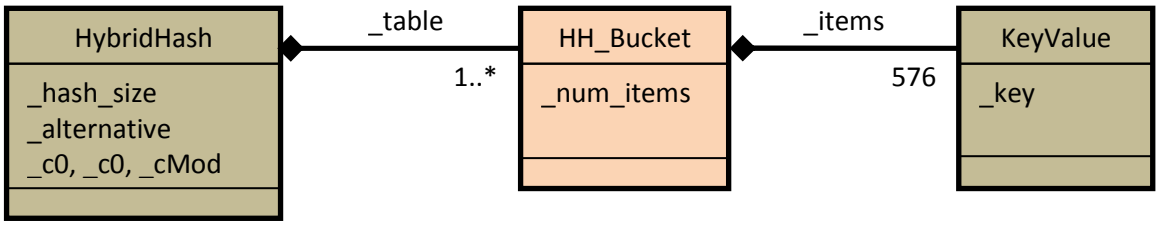


Figura 10: esquema de la tabla híbrida usada para guardar las celdas ocupadas del grid.

Para poder construir esta tabla hash híbrida primero hay que obtener la lista de *cluster_id* únicos. Para crear esta lista en la primera versión se usó un bitmap que ocupaba 128 MBytes para un grid de 1024^3 , 1 GByte para un grid de 2048^3 y 8 GBytes para un grid de 4096^3 cuando sólo hay 10 millones de celdas ocupadas. Se observó que se desperdiciaba mucha memoria.

A la hora de simplificar la malla original ya se calcula la lista de cluster_id únicos, así que simplemente se trasladó la creación de esta lista al principio del algoritmo. Esta lista también es una tabla hash donde cada elemento *(clave, valor)* es *(cluster_id, número asignado)*.

De esta manera el algoritmo de tres pasos se convierte en uno de cuatro pasos:

Algoritmo 2: implementación simplificación uniforme con hash espacial

```
entrada:
        lst_points: lista de puntos de la malla original;
        lst_triangs: lista de triángulos de la malla original;
        bbox: caja contenedora del modelo original;
        dim_x, dim_y, dim_z: dimensiones del grid usado para simplificar la malla;
salida:
        lst_points_out: lista de puntos de la malla simplificada;
        lst_triangs_out: lista de triángulos de la malla simplificada;
local:
        ht_qef_pos: KeyValueHybridHash;
        ht_cluster_ids: NodeHashTable;
        grid: DecimationGrid; // con la información del grid de soporte usado para simplificar
begin:
        grid.crear( bbox, dim_x, dim_y, dim_z);
        CreateHybridHashTable( ht_qef_pos, ht_cluster_ids, lst_puntos, grid);
        CreateQuadricMap( ht_qef_pos, lst_puntos, lst_triangulos, grid);
        FindOptimalPositions( ht_qef_pos, grid);
        DecimateMesh( lst_points_out, lst_triangs_out, lst_points, lst_triangs,
                                ht_qef_pos, ht_cluster_ids, grid);
end.
```

Para la creación de la tabla hash híbrida el artículo *[Alcantara2009]* propone estos pasos:

- calcular el número de entradas de la tabla hash simplemente haciendo $NumBuckets = [numCeldasOcupadas/409]$;
- repartir los elementos en la tabla hash con la función $mod\ NumBuckets$ y, si esta falla, se usa $h(\ clave) = [\ (\ c_0 + c_1 \cdot clave)\ mod\ 1900813] mod\ NumBuckets$. Como c_0 y c_1 he usado dos primos, si ambos fallan, dos números aleatorios y si fallan se avisa al usuario;
- cada bucket busca el número aleatorio para construir las funciones de cuco g_1, g_2, g_3 haciendo XOR con seis constantes para obtener los coeficientes c_{ij} de las funciones $g_i(k) = [\ (\ c_{i0} + c_{i1} \cdot k) mod\ 1900813] mod\ 192$ que consiguen guardar todos los elementos del bucket en las tres sub-tablas.

Las seis constantes que se han elegido son seis números primos entre 1 y 3.000.000;

El algoritmo es este:

Algoritmo 2.1: creación de la tabla hash híbrida: hash normal y de cuco en cada bucket

```
entrada:
        lst_points: lista de puntos de la malla original;
        grid: grid de soporte usado para simplificar la malla;
salida:
        ht_qef_pos: KeyValueHybridHash;
        ht_cluster_ids: NodeHashTable;
begin:
        for each p in lst_points do
                long long int idx_p = grid.get_cluster_id( p);
                ht_cluster_ids → add( idx_p);
        end for
        // numerates points and resunts number of unique cluster_ids
        int num_occupied_cells = ht_cluster_ids → numerate();
        for each cluster_id in ht_cluster_ids do
                // create a ( key, value) pair with empty value
                KV_QefPos tmp( cluster_id, 0);
                int num_items_in_bucket = ht_qef_pos → add( tmp);
                if num_items_in_bucket > 512 then
                        ht_qef_pos → clear();
                        ht_qef_pos → set_alternative_key( random_c0, random_c1, 1900813);
                        restart loop;
                end if
        end for
        for each bucket in ht_qef_pos do
                bucket → create_cuckoo();
        end for
end.
```

Y el pseudo código que crea el cuco en cada bucket es éste:

Algoritmo 2.1.1: creación de las tablas de cuco hash dentro de cada bucket

```
entrada:
        bucket: los elementos del bucket, como mucho habrá 512, pero hay espacio para 576;
salida:
        bucket: los elementos del bucket repartidos en las tres sub-tablas de 192 elementos;
begin:
        int seed = 0;
        int num_seeds_used = 0;
        int num_iterations = 0;
        bool all_stored = false;
        do
                num_iterations = 0;
                seed = Ran.randomi(); // get a random seed
                num_seeds_used++;
                create_sub_tables( t1, t2, t3);
                create_cuckoo_hashes( seed, g1, g2, g3);
                lst_not_stored = bucket; // list of elements to be stored
                do
```

```
                    // try to store all elements in the first subtable using g1
                    for each item in lst_not_stored do
                        int idx = cuckoo_g1( item.key);
                        t1[ idx] = item; // if there is already one, evict it
                    end for
                    actualize_not_stored_elements( lst_not_stored, t1);

                    // try to store all elements in the first subtable using g2
                    for each item in lst_not_stored do
                        int idx = cuckoo_g2( item.key);
                        t2[ idx] = item; // if there is already one, evict it
                    end for
                    actualize_not_stored_elements( lst_not_stored, t2);

                    // try to store all elements in the first subtable using g3
                    for each item in lst_not_stored do
                        int idx = cuckoo_g3( item.key);
                        t3[ idx] = item; // if there is already one, evict it
                    end for
                    actualize_not_stored_elements( lst_not_stored, t3);
                    num_iterations++;
                while ( lst_not_stored != {}) && ( num_iterations < MAX_ITERATIONS);
                all_stored = ( lst_not_stored == {});
            while ( !all_stored);
end.
```

Para obtener números aleatorios se ha implementado la clase Ran propuesta por el Numerical Recipes *[Numerical2007]* que promete ser más aleatorio que el generador de números aleatorios de MS Visual Studio 2008.

Los otros tres pasos del algoritmo de DeCoro se han modificado para reflejar el uso de esta nueva estructura:

Algoritmo 2.2: creación del grid con las funciones de error cuádricas acumuladas en cada celda

```
entrada:
        lst_points: lista de puntos de la malla original;
        lst_triangs: lista de triángulos de la malla original;
        ht_qef_pos: KeyValueHybridHash;
        grid: grid de soporte usado para simplificar la malla;
salida:
        ht_qef_pos: KeyValueHybridHash, con las qef y los vértices acumulados;
begin:
        for each triang in lst_triags do
            calculate_plane_coeficients( triang, a, b, c, d);
            for each node in triag do
                Point p = lst_points[ node];
                int idx_p = grid.get_cluster_id( p);
                // construir la qef
                CPUfloat4 dataA0( a * a, a * b, a * c, b * b);
                CPUfloat3 dataA1( b * c, c * c, d * d), dataB( a * d, b * d, c * d);
```

```
                    CPUfloat4 acc( p, 1.0); // acc = ( p.x, p.y, p.z, 1.0);
                    // ( key, value) = ( cluster_id, QefInfo+AccumulatedCoords)
                    ht_qef_pos → add_value( idx_p, QefAccInfo( dataA0,
                                                        dataA1, dataB, acc);
            end for
        end for
end.
```

El paso de buscar las posiciones óptimas se ha modificado para usar la tabla hash híbrida. Para ahorrar memoria, el representante óptimo de la celda se guarda donde originalmente se acumulaban los vértices de la celda.

Algoritmo 2.3: búsqueda de las posiciones óptimas

```
entrada:
        ht_qef_pos: KeyValueHybridHash
        grid: DecimationGrid; // con la información del grid de soporte usado para simplificar
salida:
        ht_qef_pos: KeyValueHybridHash con el representante óptimo calculado;
begin:
        for each bucket in ht_qef_pos do
                for each item in bucket do
                QefAccInfo data = item.value;
                int cluster_id = item.key;
                CPUfloat4 cur_point = data.acc_coords;
                // la componente w contiene el número de puntos acumulados en la celda
                if cur_point.w >= 1 then // algunos item están vacios por el cuco hash
                        CPUfloat4 optimal_position; // representante de la celda
                        CPUfloat4 dataA0 = data.quadric_A1;
                        CPUfloat3 dataA1 = data.quadric_A2;
                        CPUfloat3 dataB = data.quadric_B;
                        CPUMatrix quadric(      dataA0.x, dataA0.y, dataA0.z, dataB.x,
                                                dataA0.y, dataA0.w, dataA1.x, dataB.y,
                                                dataA0.z, dataA1.x, dataA1.y, dataB.z,
                                                0.0, 0.0, 0.0, 1.0);
                        if quadric.determninant() > SINGULAR_THRESHOLD then
                                CPUMatrix inv = quadric.get_inverse();
                                optimal_position = inv * ( 0.0, 0.0, 0.0, 1.0);
                                if !grid.inside( cluster_id, optimal_position) then
                                // si no está dentro de la celda, congemos la media de los vértices
                                        optimal_position = cur_point / cur_point.w;
                                end if
                        else
                                // no se puede invertir,
                                // el representante es la media de los vértices acumulados
                                optimal_position = cur_point / cur_point.w;
                        end if
                        // usar el acumulado para guardar la posición óptima
                        data.acc = optimal_position;
                        item.value = data;
                end if
        end for
end.
```

En el último paso de simplificar la malla, ya no hace falta calcular los nodos únicos, pues ya se han calculado para poder crear la tabla de hash híbrido. Para simplificar sólo se muestra el algoritmo de obtención de los triángulos únicos.

Algoritmo 2.4: simplificar la malla original

```
entrada:
        lst_points: lista de puntos de la malla original;
        lst_triangs: lista de triángulos de la malla original;
        ht_qef_pos: KeyValueHybridHash;
        ht_cluster_ids: NodeHashTable;
        grid: grid de soporte usado para simplificar la malla;
salida:
        lst_points_out: lista de puntos de la malla simplificada;
        lst_triangs_out: lista de triángulos de la malla simplificada;
local:
        ht_triangs: hash para obtener los triángulos únicos
begin:
        for each triang in lst_triangs do
                Point p1 = lst_points[ triang.v1()];
                Point p2 = lst_points[ triang.v2()];
                Point p3 = lst_points[ triang.v3()];
                int idx_p1 = grid.get_cluster_id( p1);
                int idx_p2 = grid.get_cluster_id( p2);
                int idx_p3 = grid.get_cluster_id( p3);
                // si los vértices están en tres celdas diferentes, entonces guardamos el triángulo
                if ( idx_p1 != idx_p2) && ( idx_p2 != idx_p3) && ( idx_p3 != idx_p1) then
                        // guardamos el triangulo
                        // obtenemos los representantes de las celdas
                        Point sim_p1 = ht_qef_post → get_cuckoo( idx_p1) → value().opt_pos();
                        Point sim_p2 = ht_qef_post → get_cuckoo( idx_p2) → value().opt_pos();
                        Point sim_p3 = ht_qef_post → get_cuckoo( idx_p3) → value().opt_pos();
                        // obtenemos el número del puntos ( los habíamos enumerado)
                        int new_idx_p1 = ht_cluster_ids → get( idx_p1) → get_num();
                        int new_idx_p2 = ht_cluster_ids → get( idx_p2) → get_num();
                        int new_idx_p3 = ht_cluster_ids → get( idx_p3) → get_num();
                        Triangle new_triang( new_idx_p1, new_idx_p2, new_idx_p3);
                        correct_normal( new_triang, triang);
                        ht_triangs → add( new_triang);
                end if
        end for
        lst_points_out = ht_cluster_ids → get_all_nodes();
        lst_triangs_out = ht_triangs → get_all_triangles();
end.
```

Segunda versión (HashUC): problemas y mejoras

A parte del problema ya comentado sobre el uso del bitmap para conseguir los *cluster_id* únicos que se ha solventado usando un hash de *cluster_ids*, sólo ha aparecido un problema grave.

Al usar los números que propone Alcántara para las tablas de cuco hash, el algoritmo tardaba mucho en encontrar las funciones de cuco adecuadas para guardar todos los elementos del bucket.

Recordemos que se usaba un número aleatorio, haciendo XOR con seis constantes para construir las funciones de hash de cucko: $g_i(k) = [\ (\ c_{i0} + c_{i1} \cdot k) mod\ 1900813] mod\ 192$.

El tamaño de las subtablas era de 192 elementos y se ha observado que la mayor parte de las veces sólo las entradas impares se llenaban. Esto parece indicar que había algún acoplamiento entre este número par y los seis primos escogidos como constantes.

Siguiendo con los primos, se ha cambiado el tamaño de las subtablas a 191, otro primo, y el algoritmo funciona perfectamente.

Así pues los buckets del algoritmo se llenan hasta 512 elementos, pero tienen 3 * 191 = 573 elementos de tamaño, para dejar espacio libre suficiente y poder crear las funciones de hash de cuco. Con estos números la tabla hash usa un 40,01 % como espacio para los hashes, respecto al anterior 40,83 %.

Después de ver esto, ha surgido la duda de si se pueden llenar más los buckets, entre 409 y 512, para conseguir un uso más eficiente de memoria. En el apartado de resultados se mostrará el resultado de los experimentos realizados en este sentido.

Según los pruebas de Alcántara, casi todas las veces se conseguía encontrar las funciones de hash de cuco a la primera después de 5,5 iteraciones y en pocos casos se ha tenido que escoger una única nueva semilla para las funciones de cuco. Con los ejemplos seleccionados para este proyecto y probando con grids de diferentes resoluciones, 256^3, 512^3y 1024^3, la media de iteraciones es de 5,3 y sólo en tres casos se ha vuelto a escoger una nueva semilla: un ejemplo diferente para cada resolución. Al final del proyecto se usó un ejemplo, mapa de terreno Puget Sound, para el cual no se ha podido construir el hash de cuco para el grid de resolución 8192^3.

El hecho de haber escogido el cluster_id como un entero, limita la resolución del grid a 1024^3. Al ver estos buenos resultados, se ha ampliado el algoritmo original para poder trabajar con grid de mayor resolución, cambiando el cluster_id a un entero de 64 bits. Esto ha permitido llegar hasta grids de 131.072^3 en algunos ejemplos concretos.

Segunda versión (HashUC): paralelización

De la misma manera que antes, el seguir las etapas propuestas por DeCoro facilita mucho la paralelización del algoritmo. A esto se añade que las tablas hash ya indican donde han de ponerse los semáforos: al menos hay que poner uno para cada bucket.

Las siguientes tablas y gráficas muestran la escalabilidad del algoritmo:

Plataforma 1: Intel Q9550 QuadCore + Ubuntu 8.04 + gcc 4.4.5

Tamaño grid	1 core	2 cores	4 cores
16.384^3	166,494	106,091	78,225
8.192^3	152,491	97,367	71,523
4.096^3	122,611	78,820	58,408
2.048^3	51,484	33,605	25,282
1.024^3	20,414	12,602	8,891
512^3	12,683	7,274	4,014
256^3	10,767	5,775	3,370

Tabla 16: tiempos, en segundos, para simplificar el modelo de la estatua lucy usando diferentes tamaños de grid en la *plataforma 1*: Intel Q9550 QuadCore.

Tamaño grid	1 core	2 cores	4 cores
16.384^3	1	1,569	2,128
8.192^3	1	1,566	2,132
4.096^3	1	1,556	2,099
2.048^3	1	1,532	2,036
1.024^3	1	1,620	2,296
512^3	1	1,744	3,160
256^3	1	1,864	3,195

Tabla 17: Speed-up para simplificar el modelo de la estatua lucy usando diferentes tamaños de grid en la *plataforma 1*: Intel Q9550 QuadCore.

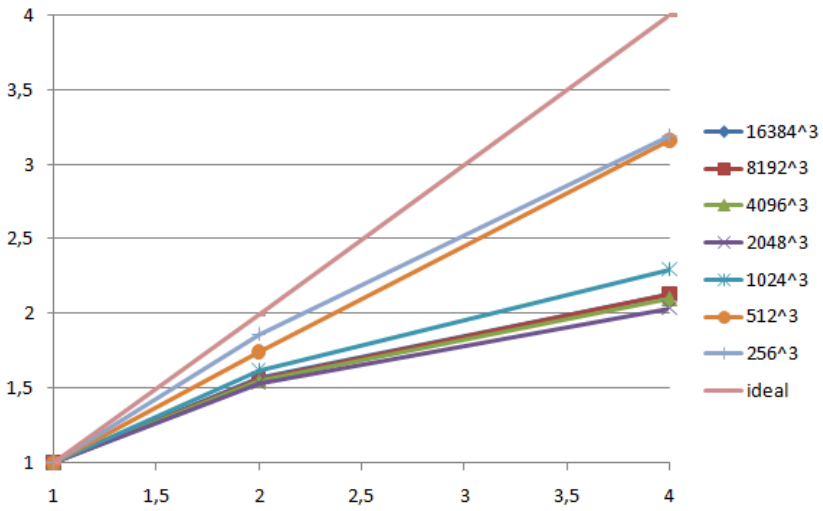


Figura 11: Escalabilidad del algoritmo en la *plataforma 1*: Q9550 QuadCore

Plataforma 2: Intel i7-920 QuadCore + HT + Ubuntu 10.04 + gcc 4.4.3

Tamaño grid	1 core	2 cores	4 cores	4 cores + HT
16.384^3	102,333	60,170	37,450	29,677
8.192^3	93,396	55,055	33,969	27,210
4.096^3	75,136	45,354	27,854	22,341
2.048^3	33,809	20,084	13,021	10,450
1.024^3	15,248	8,935	5,591	4,359
512^3	10,261	5,635	3,522	2,854
256^3	8,653	4,856	2,796	2,657

Tabla 18: tiempos, en segundos, para simplificar el modelo de la estatua lucy usando diferentes tamaños de grid en la *plataforma 2*: i7-920.

Tamaño grid	1 core	2 cores	4 cores	4 cores + HT
16.384^3	1	1,701	2,733	3,448
8.192^3	1	1,696	2,749	3,432
4.096^3	1	1,657	2,697	3,363
2.048^3	1	1,683	2,596	3,235
1.024^3	1	1,707	2,727	3,498
512^3	1	1,821	2,913	3,595
256^3	1	1,782	3,095	3,257

Tabla 19: Speed-up para simplificar el modelo de la estatua lucy usando diferentes tamaños de grid en la *plataforma 2*: i7-920.

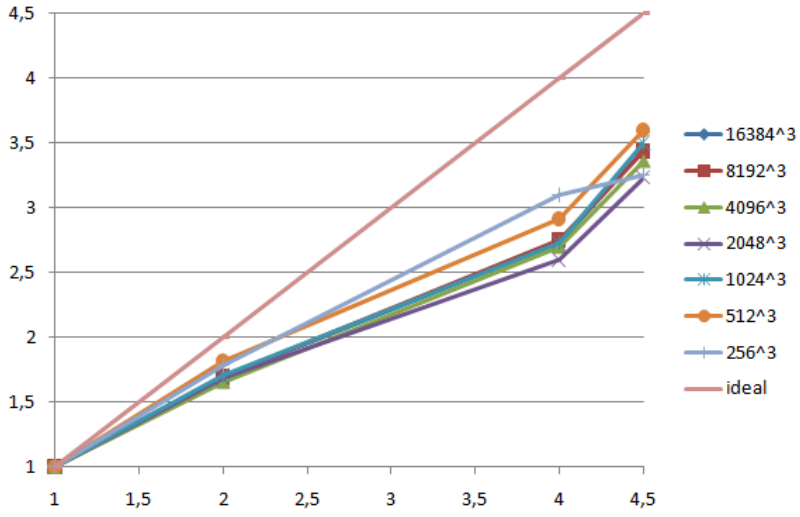


Figura 12: Escalabilidad del algoritmo en la *plataforma 2*: i7-920

3.4. Resumen

Se ha seguido las etapas propuestas por DeCoro, pero se ha mejorado el algoritmo incorporando:

- verificación de que el representante óptimo esté dentro de la celda,
- corrección de normales,
- extracción de líneas resultados de los triángulos colapsados,
- eliminación de puntos, líneas y triángulos repetidos.

Para poder aumentar la resolución de la simplificación más allá de 256^3, ahorrar memoria y no perder calidad se ha incorporado el hash espacial de Alcántara con la siguiente mejora:

- respecto al artículo de Alcantara, se ha usado la tabla hash híbrida para simplificar una malla,
- uso de enteros de 64 bits como claves de hash para romper el límite de 1.024^3 ,
- se propone aumentar el factor de ocupación de los buckets para ahorrar memoria

Con estas mejoras una línea futura es montar un octree con:

- niveles 1..6 completos con el algoritmo desarrollado en la primera versión de este proyecto,
- niveles 7..N usando el hash espacial de cuco implementado en la segunda versión del proyecto.
- la profundidad máxima vendría dada por estos criterios:
    - tamaño mínimo de celda o tamaño mínimo de triángulo seleccionable por el usuario
    - cantidad de puntos mínima en la celda, esto implica contar los puntos de cada celda al construir el hash de *cluster_ids* únicos
    - cantidad de memoria disponible, la memoria que necesita el algoritmo de hash espacial depende del tamaño del modelo inicial y, asintóticamente es de 108 bytes por punto, si cada punto de la malla se le asigna una celda única.

4. Resultados

4.1. Imágenes

La siguiente secuencia de imágenes muestra los diferentes niveles de simplificación del modelo lucy, con 14.027.872 puntos y 28.055.742 triángulos. Los tiempos mostrados han sido tomados en la *plataforma 1*: Intel QuadCore Q9550 usando MS Windows 7 64 bits.

Figura 13: Visualización del modelo original Lucy y de detalles que se mostrarán cómo afecta la simplificación a la calidad del modelo.
14.027.872 puntos
28.055.742 triángulos
0 líneas
0 segundos

La siguiente tabla muestra un resumen de los niveles de simplificación del modelo y sus tiempos:

Tamaño grid	Puntos	Triángulos	Líneas	T HashUC (s)	T FullUC (s)
1^3	1	0	0		
2^3	8	14	0	62,134	62,385
4^3	18	32	2	61,439	12,777
8^3	68	138	6	59,356	6,788
16^3	284	660	6	60,345	5,073
32^3	1.212	2.800	4	22,921	4,294
64^3	5.437	11.526	8	7,535	4,314
128^3	22.785	46.569	12	4,386	4,544
256^3	90.780	182.902	11	3,977	6,132
512^3	351.327	705.207	17	5,008	n/a
1.024^3	1.317615	2.642.388	23	10,512	n/a
2.048^3	4.555.618	9.128.466	6	30,673	n/a
4.096^3	10.287.708	20.568.519	7	71,750	n/a
original	14.027.872	28.055.742	0		

Tabla 20: tamaño de las mallas simplificadas y tiempo tomado para imsplificarlas usando el hash híbrido de Alcántara (HashUC) y el grid lleno de DeCoro.

Nivel 12 de grid: 4096 x 4096 x 4096 celdas

Figura 14: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

10.287.708 puntos
20.568.519 triángulos
7 líneas

71,750 segundos con HashUC

Nivel 11 de grid: 2048 x 2048 x 2048 celdas

Figura 15: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

4.555.618 puntos
9.128.466 triángulos
6 líneas

30,673 segundos con HashUC

Nivel 10 de grid: 1024 x 1024 x 1024 celdas

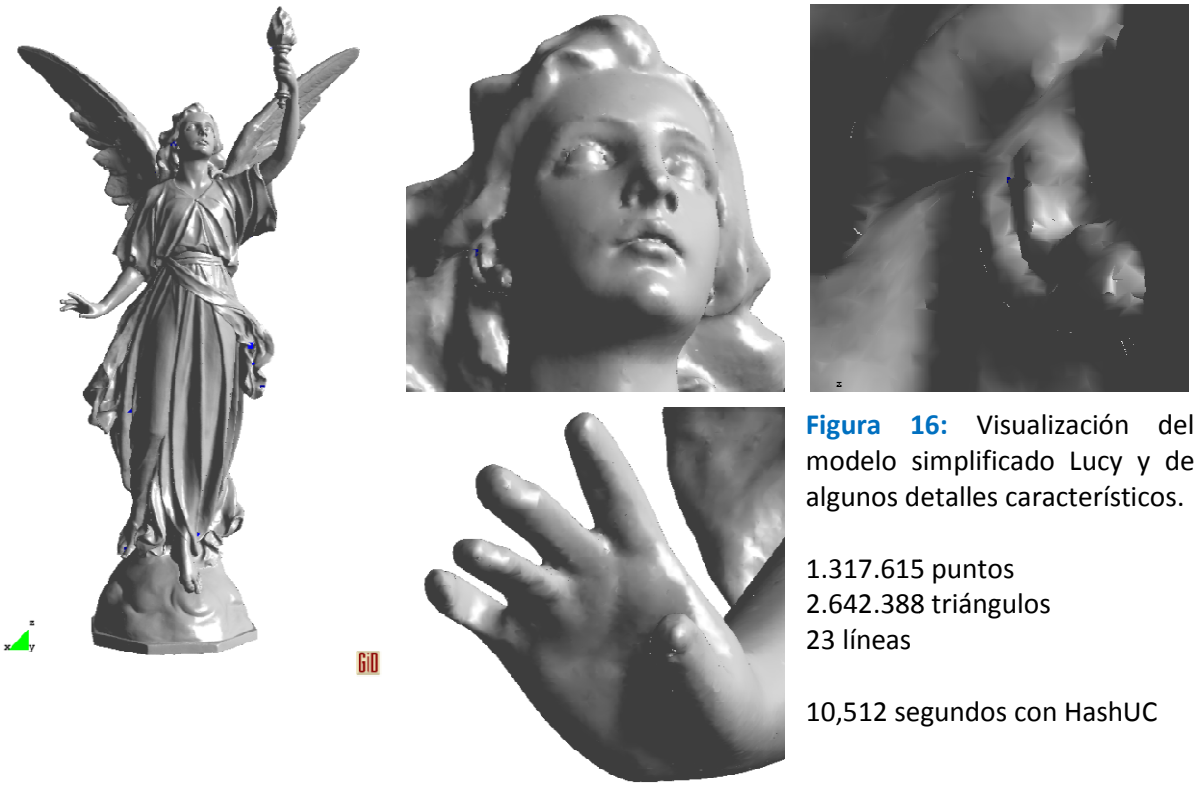


Figura 16: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

1.317.615 puntos
2.642.388 triángulos
23 líneas

10,512 segundos con HashUC

Nivel 9 de grid: 512 x 512 x 512 celdas

Figura 17: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

351.327 puntos
705.207 triángulos
17 líneas

5,008 segundos con HashUC

Nivel 8 de grid: 256 x 256 x 256 celdas

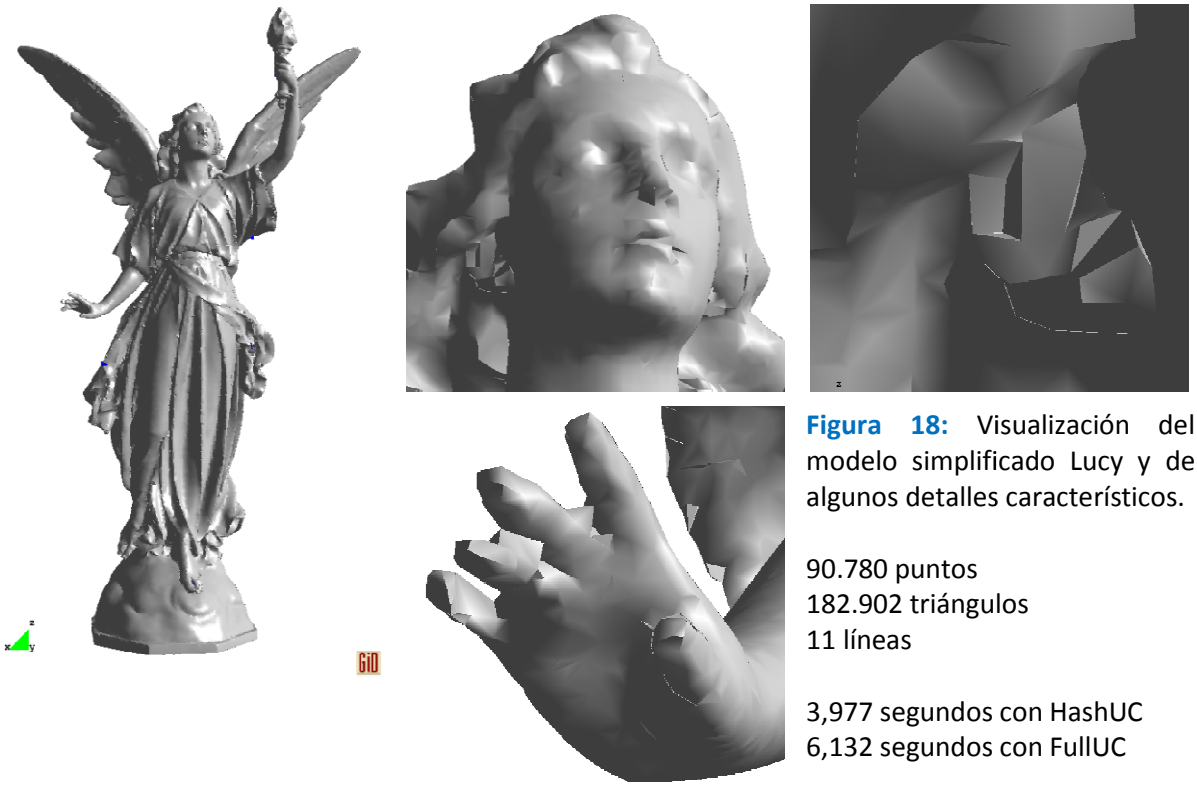


Figura 18: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

90.780 puntos
182.902 triángulos
11 líneas

3,977 segundos con HashUC
6,132 segundos con FullUC

Nivel 7 de grid: 128 x 128 x 128 celdas

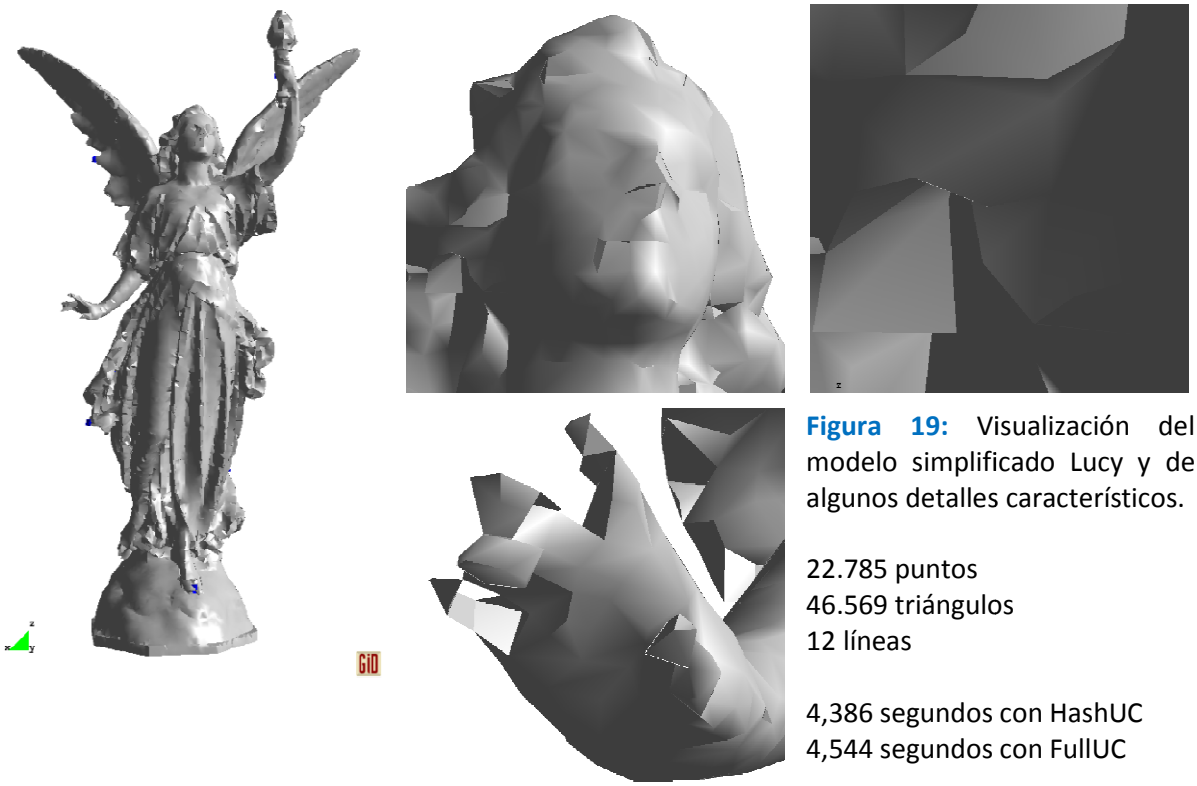


Figura 19: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

22.785 puntos
46.569 triángulos
12 líneas

4,386 segundos con HashUC
4,544 segundos con FullUC

Nivel 6 de grid: 64 x 64 x 64 celdas

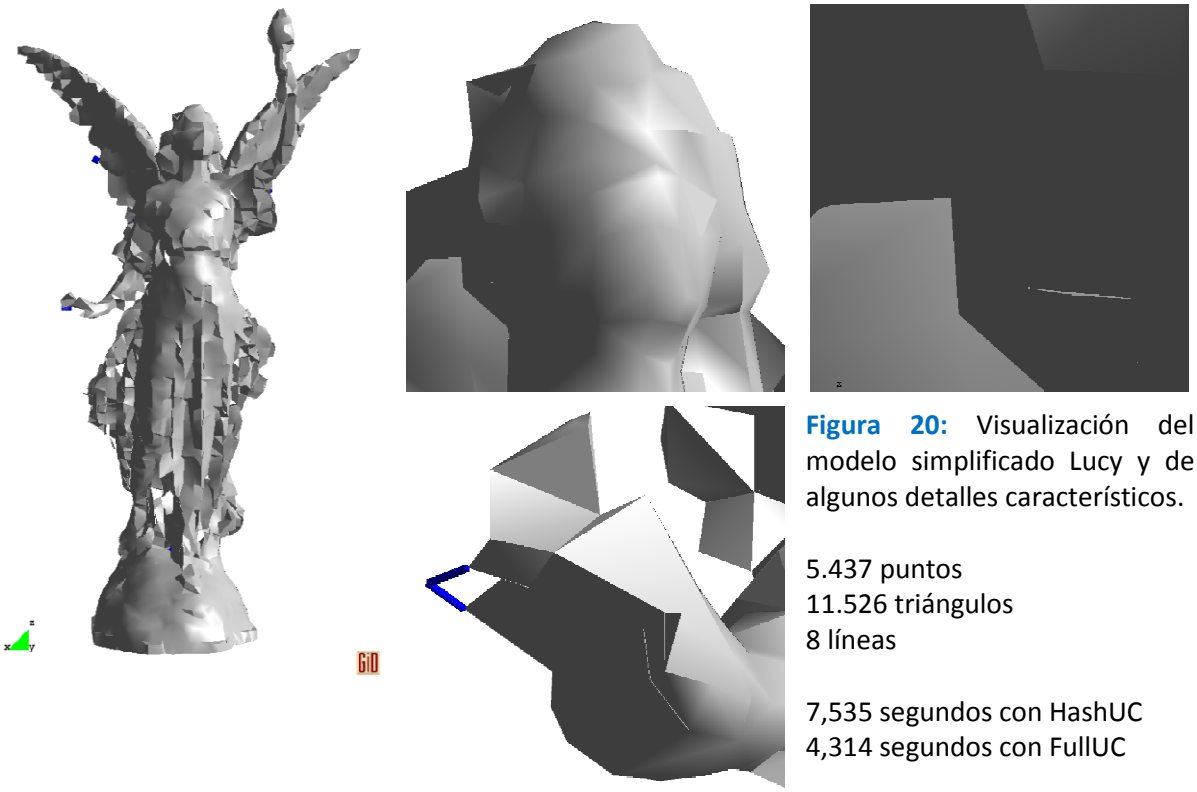


Figura 20: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

5.437 puntos
11.526 triángulos
8 líneas

7,535 segundos con HashUC
4,314 segundos con FullUC

Nivel 5 de grid: 32 x 32 x 32 celdas

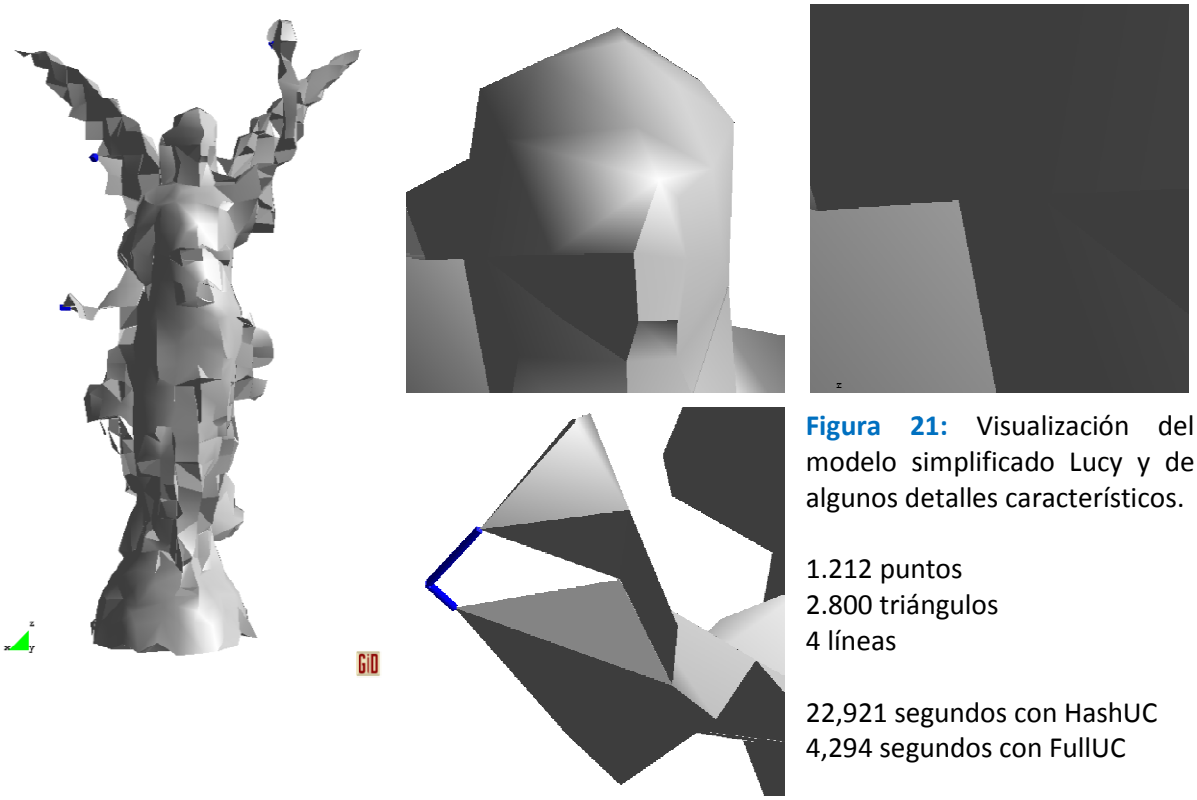


Figura 21: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

1.212 puntos
2.800 triángulos
4 líneas

22,921 segundos con HashUC
4,294 segundos con FullUC

Nivel 4 de grid: 16 x 16 x 16 celdas

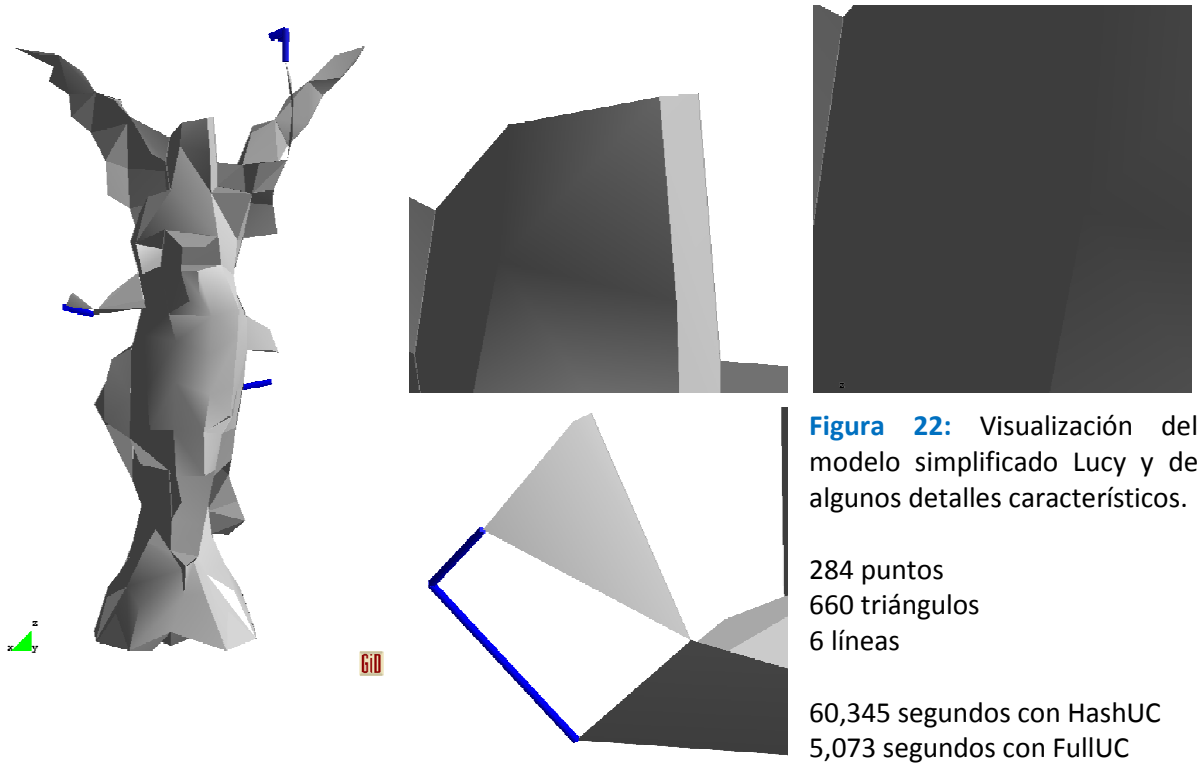


Figura 22: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

284 puntos
660 triángulos
6 líneas

60,345 segundos con HashUC
5,073 segundos con FullUC

Nivel 3 de grid: 8 x 8 x 8 celdas

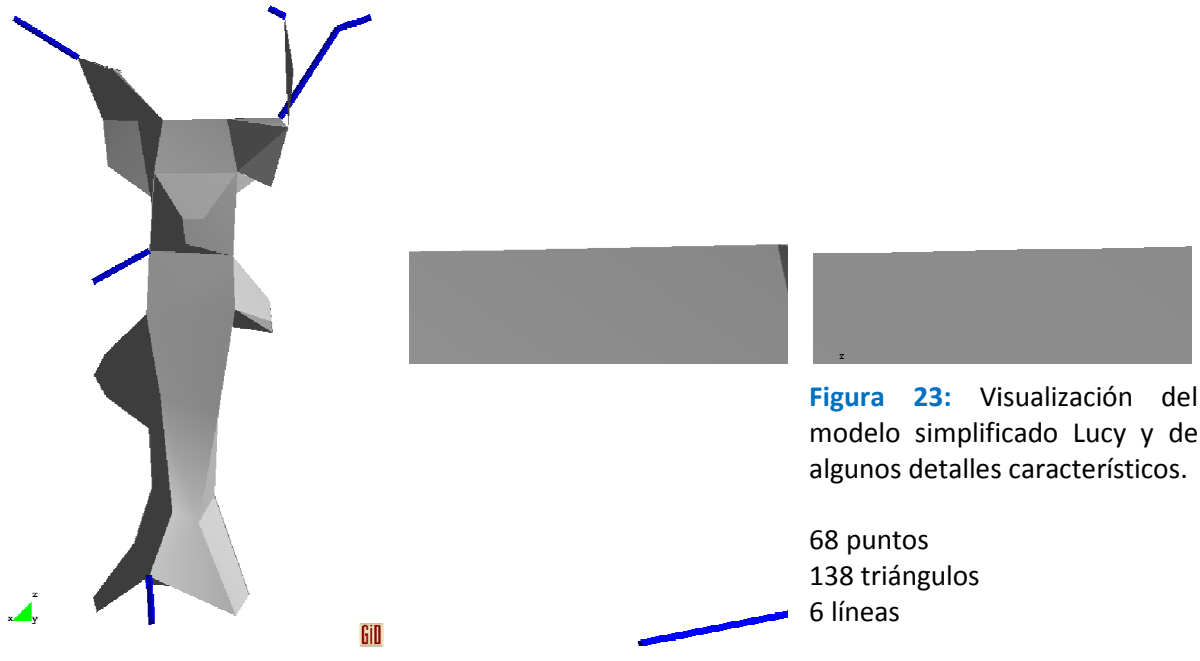


Figura 23: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

68 puntos
138 triángulos
6 líneas

59,356 segundos con HashUC
6,788 segundos con FullUC

Nivel 2 de grid: 4 x 4 x 4 celdas

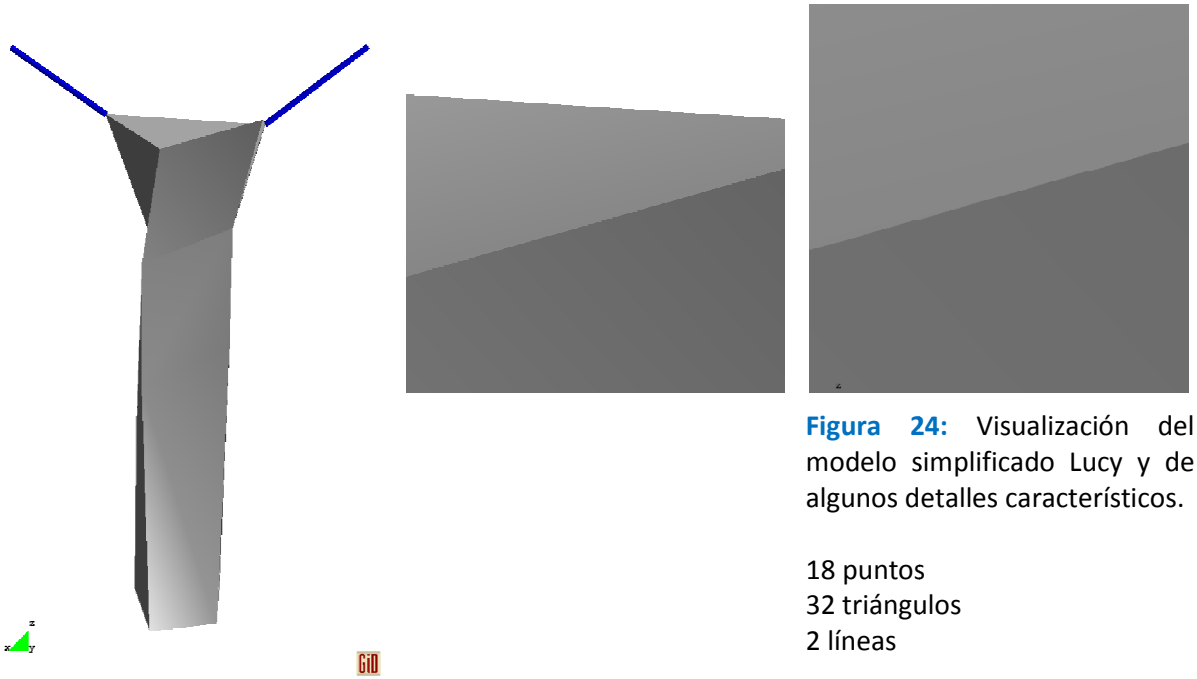


Figura 24: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

18 puntos
32 triángulos
2 líneas

61,439 segundos con HashUC
12,777 segundos con FullUC

Nivel 1 de grid: 2 x 2 x 2 celdas

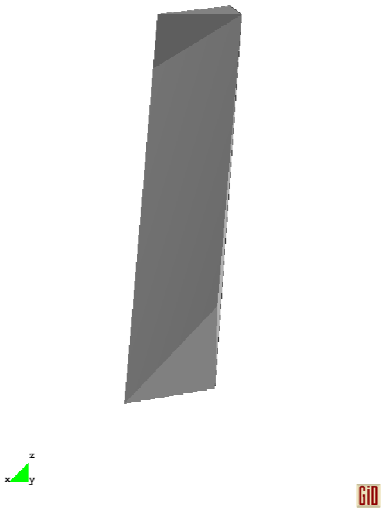


Figura 25: Visualización del modelo simplificado Lucy y de algunos detalles característicos.

8 puntos
14 triángulos
0 líneas

62,134 segundos con HashUC
62,385 segundos con FullUC

Nivel 0 de grid: 1 x 1 x 1 celda = 1 punto.

En el ejemplo del plano de una ciudad se puede ver el acierto de recuperar los triángulos colapsados como líneas, que se muestran como líneas gruesas de color azul:

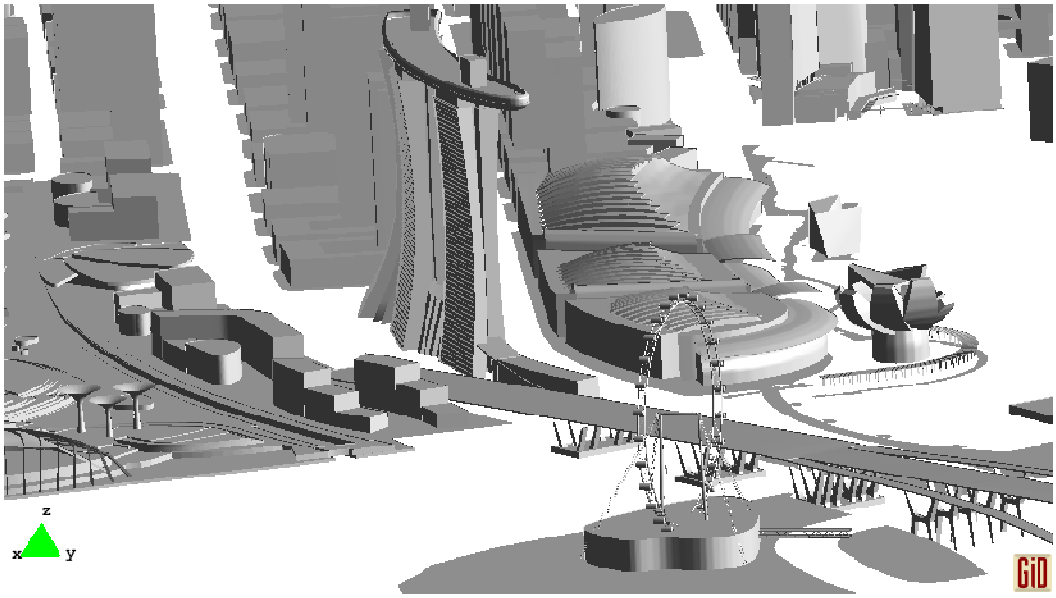


Figura 26: Imagen del modelo original de una ciudad con 6.107.478 puntos y 15.601.856 triángulos

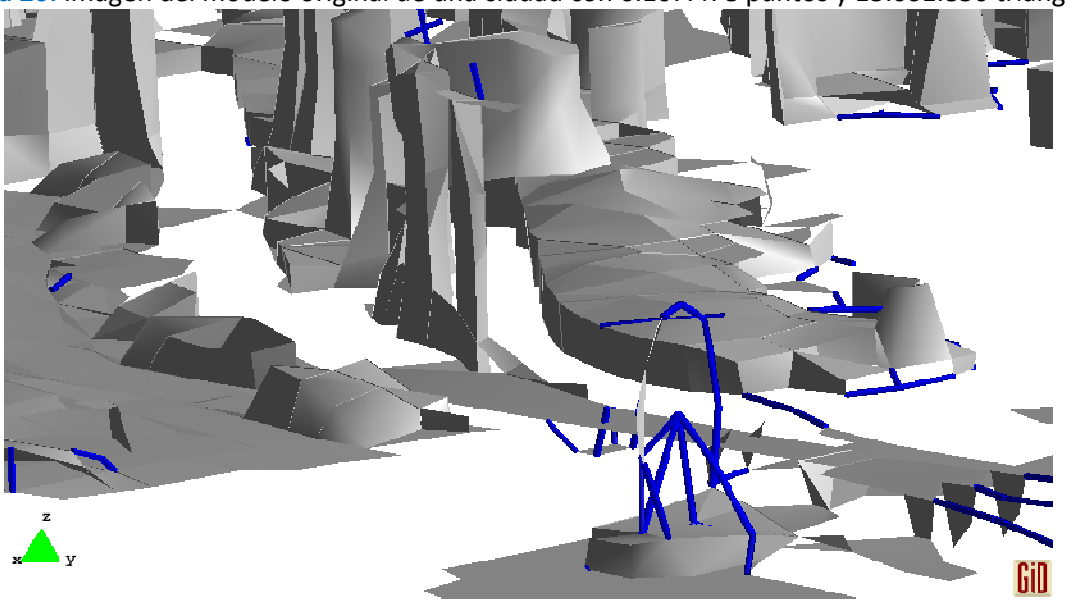


Figura 27: Nivel 7 de grid malla con 12.672 puntos, 34.435 triángulos y 380 líneas

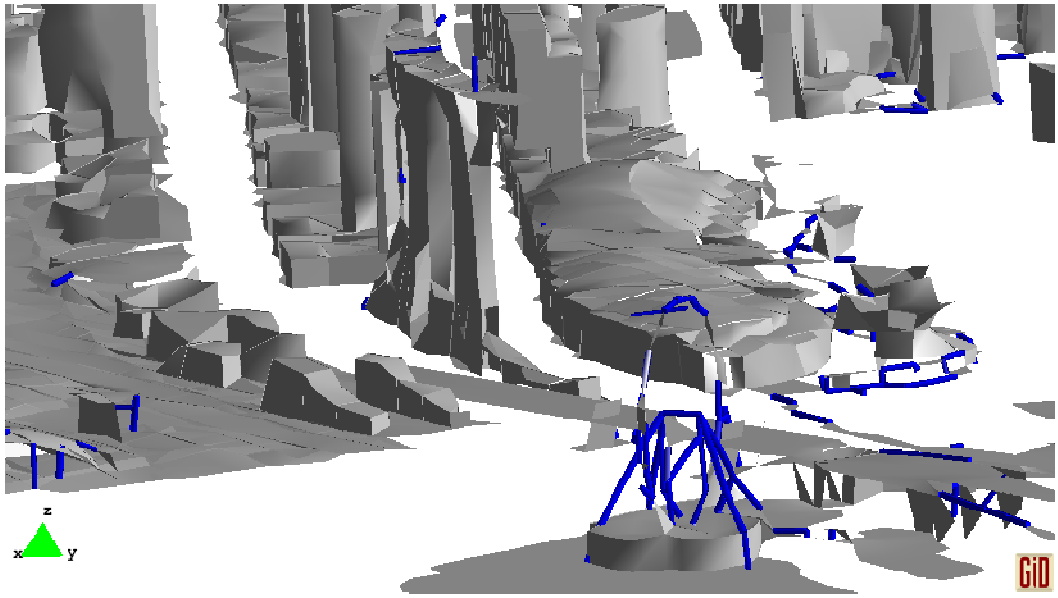


Figura 28: Nivel 8 de grid malla con 44.276 puntos, 113.235 triángulos y 548 líneas

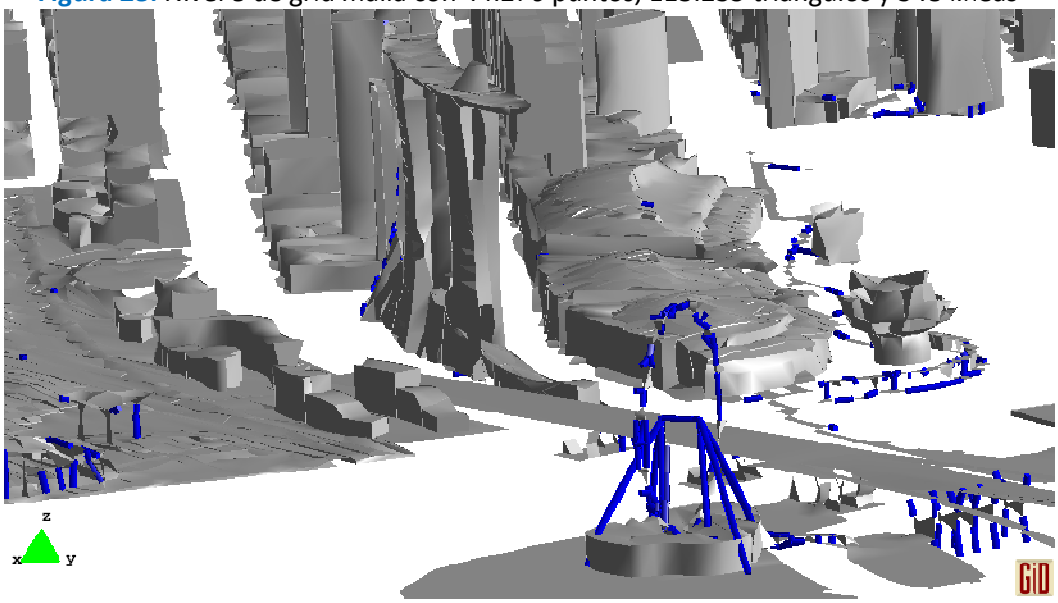


Figura 29: Nivel 9 del grid malla con 137.922 puntos, 332.334 triángulos y 952 líneas

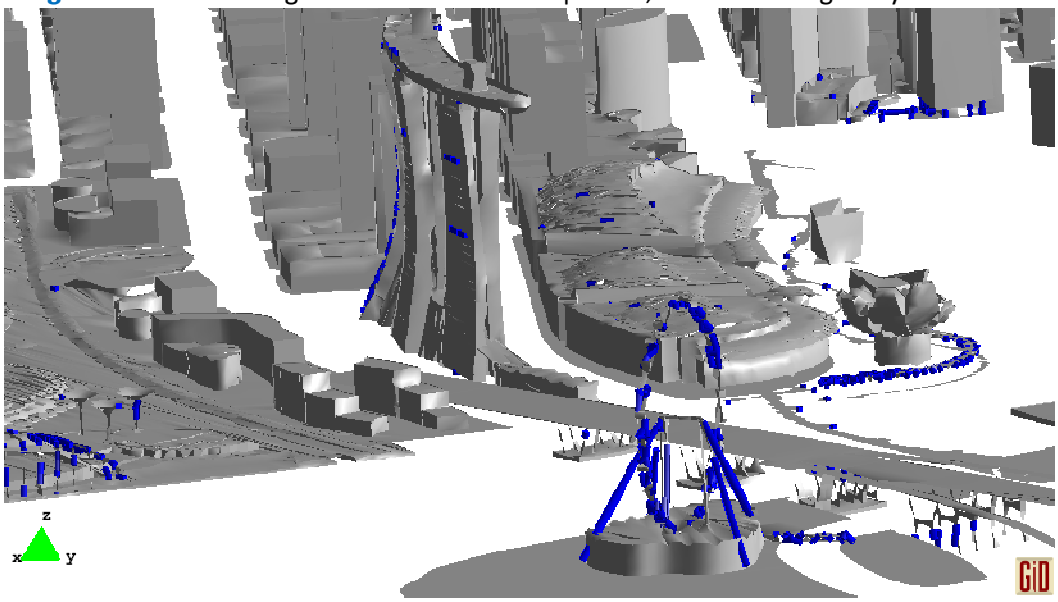


Figura 30: Nivel 10 del grid malla con 386.344 puntos, 881.256 triángulos y 1.492 líneas

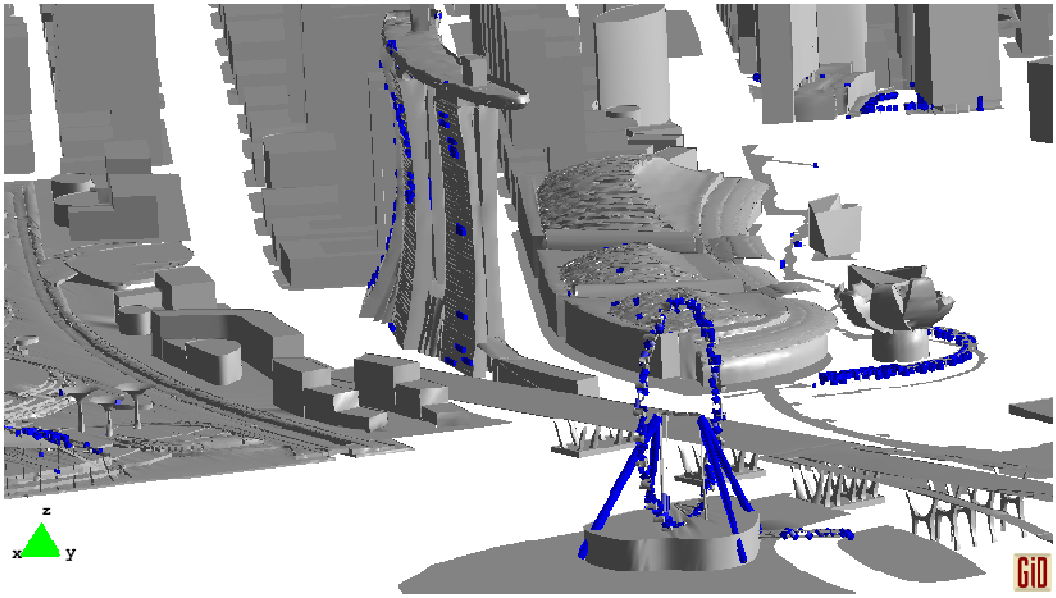


Figura 31: Nivel 11 del grid malla con 779.960 puntos, 1.682.503 triángulos y 1.760 líneas

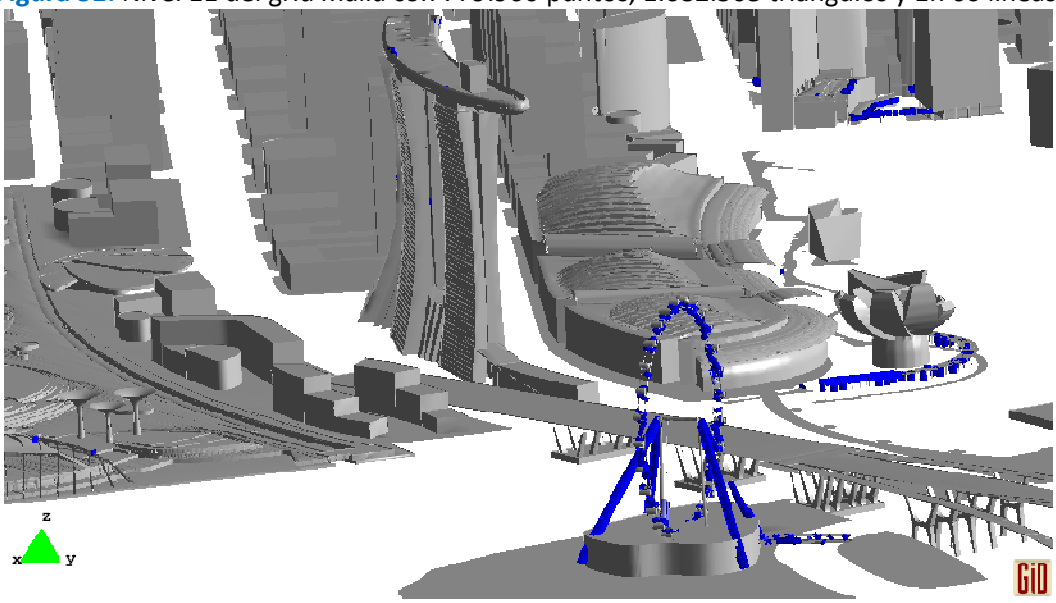


Figura 32: Nivel 12 del grid malla con 1.289.987 puntos, 2.679.289 triángulos y 1.822 líneas

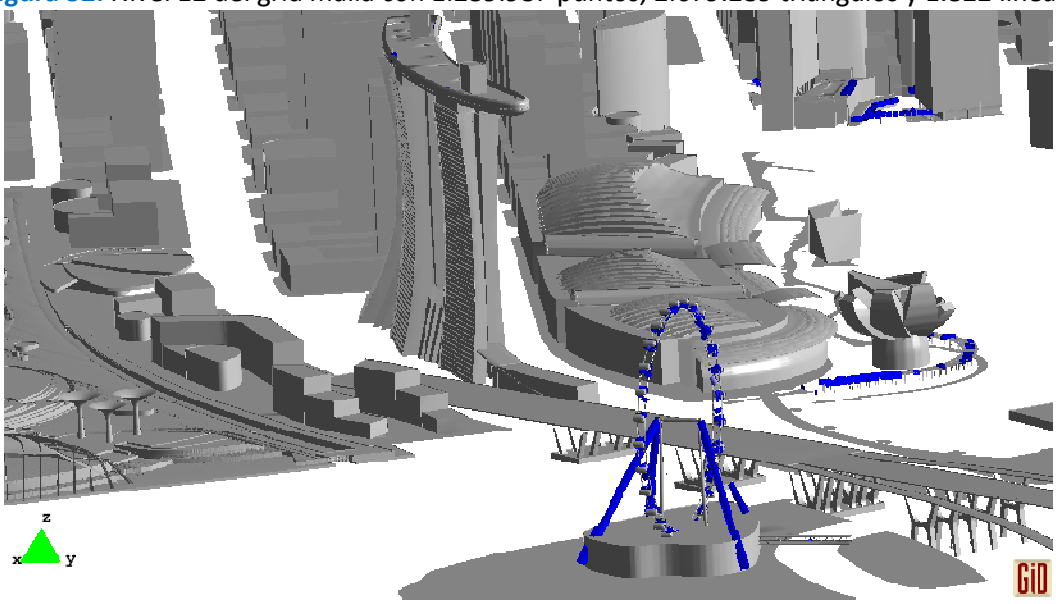


Figura 33: Nivel 13 del grid malla con 1.881.515 puntos, 3.827.405 triángulos y 1.487 líneas

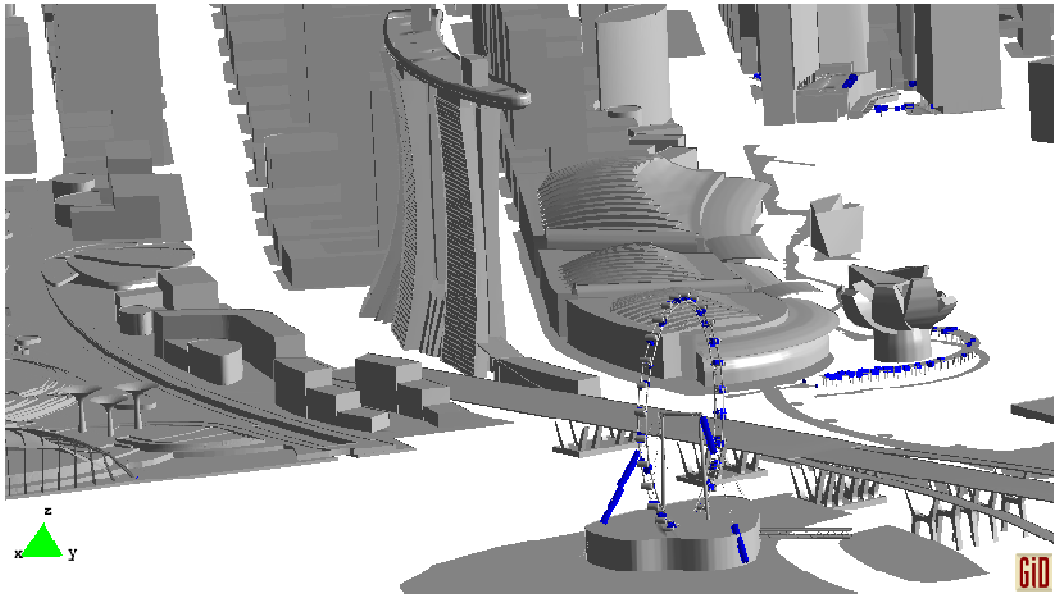
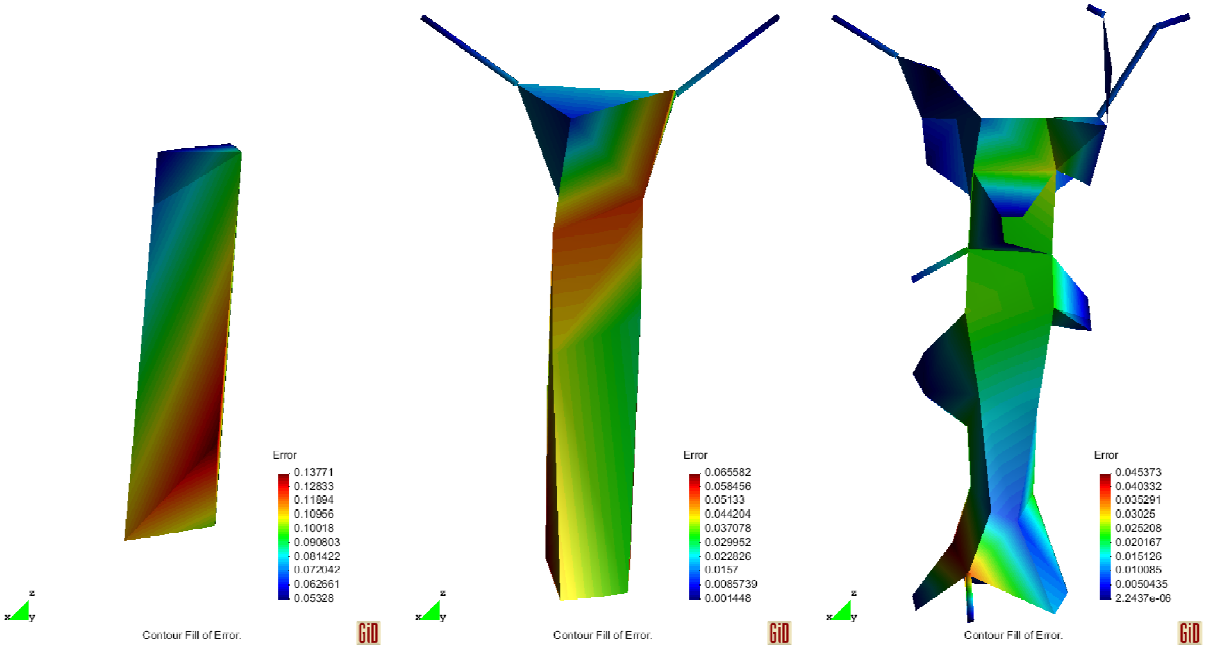


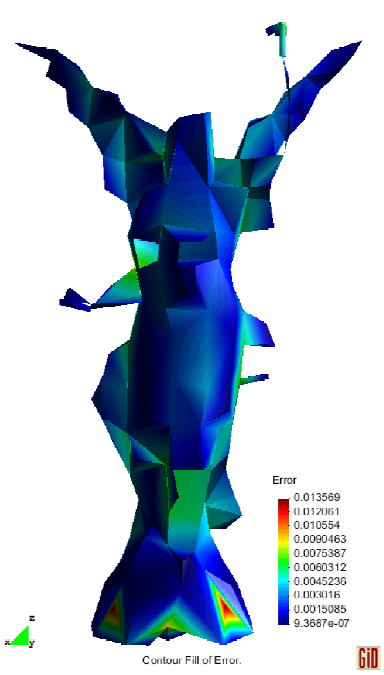
Figura 34: Nivel 14 del grid malla con 2.575.495 puntos, 5.158.613 triángulos y 739 líneas

En las próxima figura se puede apreciar la evolución del error según los diferentes niveles de refinamiento. El error en cada representante de la celda está definido como:

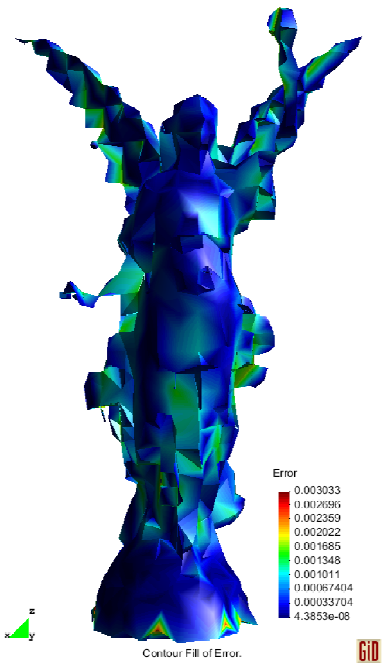
$$E(v) = \sqrt[2]{\sum_{p \in planos(v)} (p^T v)^2}$$



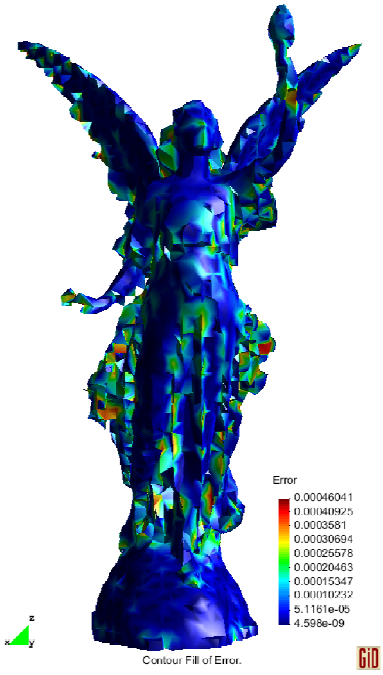
Nivel 1: 2^3 Nivel 2: 4^3 Nivel 3: 8^3



Nivel 4: 16^3

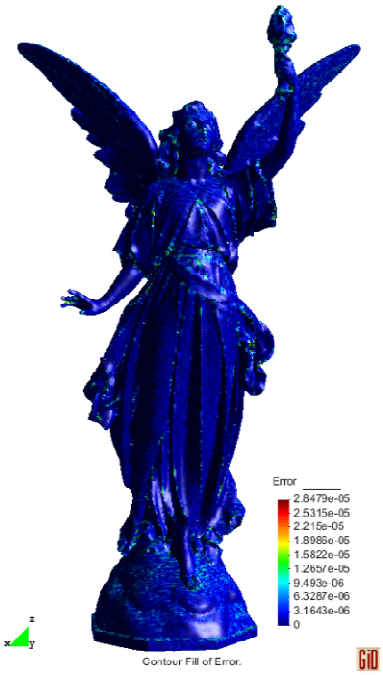


Nivel 5: 32^3

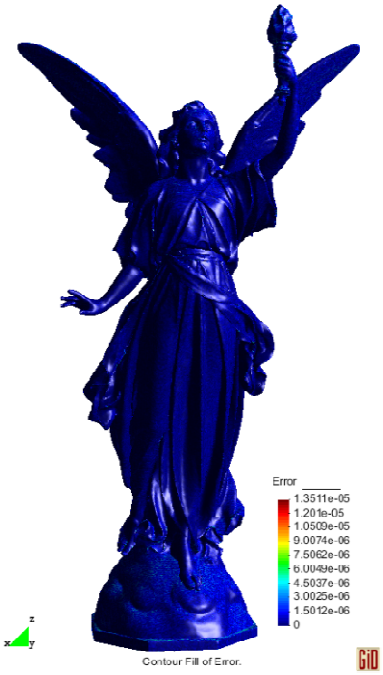


Nivel 6: 64^3

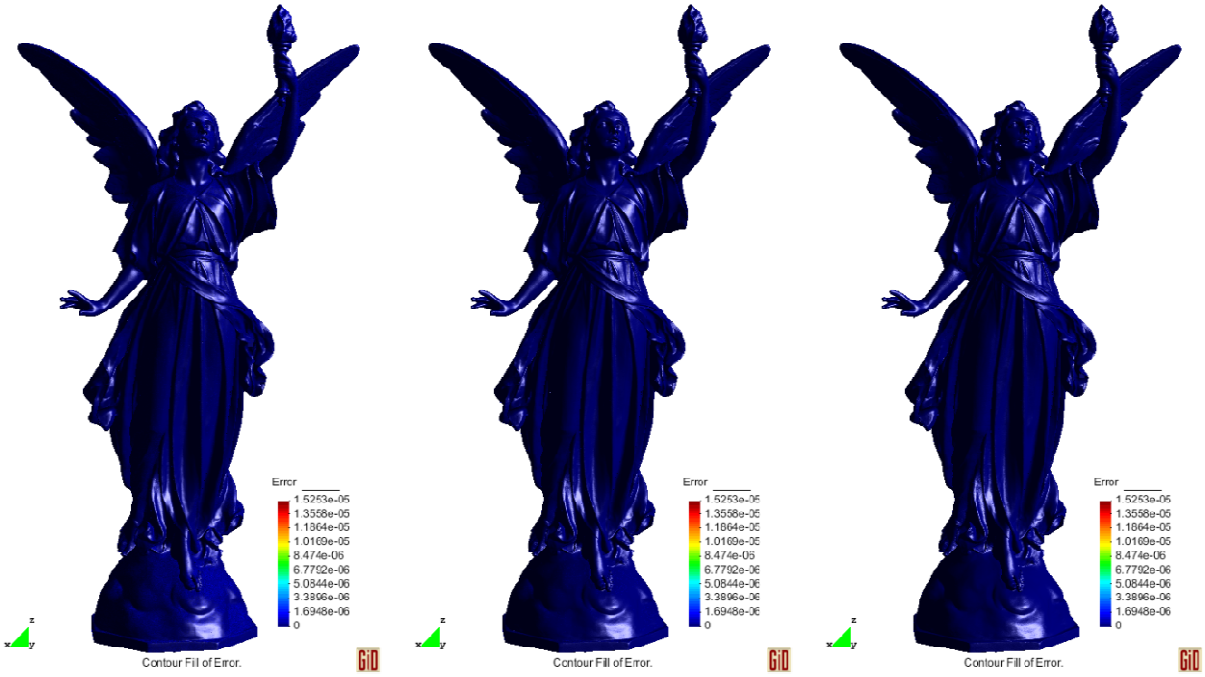
Nivel 7: 128^3



Nivel 8: 256^3



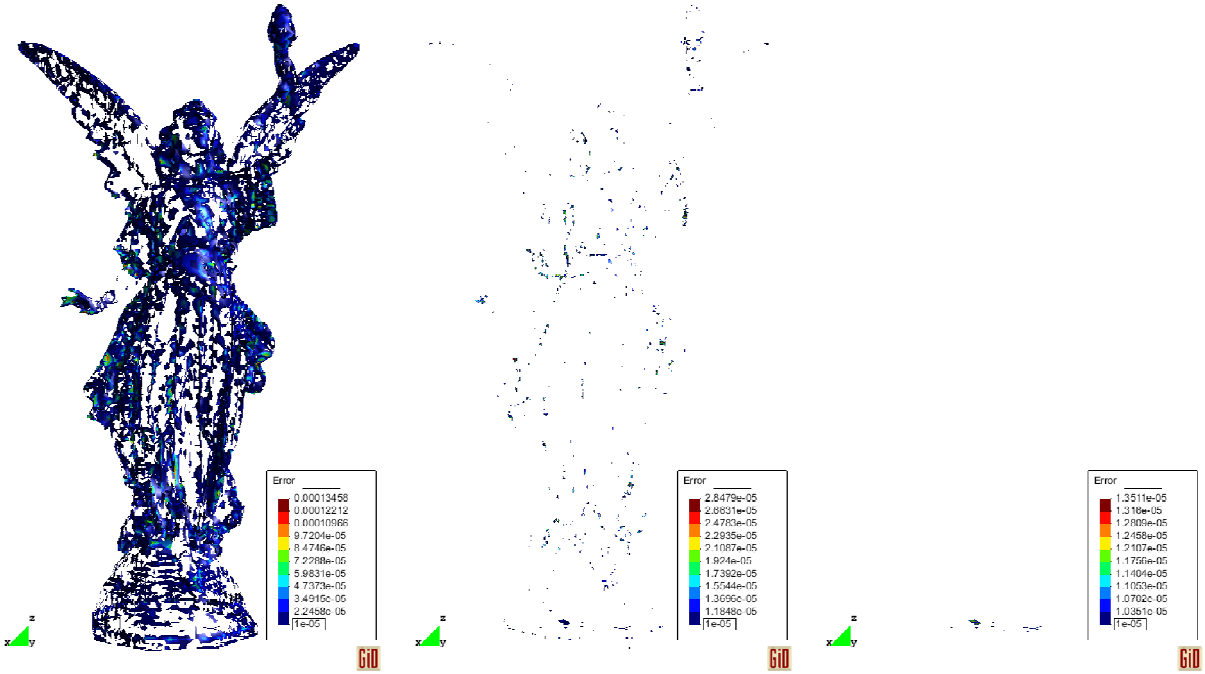
Nivel 9: 512^3



Nivel 10: 1.024^3 Nivel 11: 2.048^3 Nivel 12: 4.096^3

Figura 35: Evolución de la distribución del error en los diferentes niveles de refinamiento, desde el nivel 1 arriba a la izquierda hasta el nivel 12, la imagen de abajo a la derecha.

Se puede observar que a partir del nivel 9 prácticamente toda la estatua es de color azul oscuro, con error prácticamente 0. Si se filtran los errores más pequeños de 10e-5 a partir del nivel 7, se puede observar que a partir del nivel 9, incluido éste, los errores se concentran en la base de la estatua. La base de la estatua es prácticamente un plano, con lo que la función de error cuártica no se puede invertir y el representante de la celda es la media de los vértices.



Nivel 7: 128^3 Nivel 8: 256^3 Nivel 9: 512^3

Figura 36: Evolución de la distribución del error mostrando sólo los errores superiores a 10e-5.

Si se acepta un error de 10e-5, sólo hace falta llegar al nivel 9 de profundidad del octree para simplificar el modelo lucy.

4.2. Tiempos

A continuación se muestra una tabla de tiempos en función de la resolución del grid usado para los modelos de 2 millones de triángulos o más:

	Modelo	pensatore	neptune	manuscript	F1 car	xyz_dragon
	puntos	997.875	2.003.932	2.155.617	3.005.848	3.609.600
	triángulos	1.995.746	4.007.872	4.305.818	6.011.696	7.219.045
grid						
256^3	puntos	248.940	66.242	45.319	83.739	89.828
	triángulos	498.653	133.215	90.047	168.453	180.801
	tiempo	**1,108**	**0,851**	**0,672**	**1,288**	**1,254**
512^3	puntos	752.382	244.733	179.746	331.411	344.137
	triángulos	1.505.771	491.619	358.426	665.791	390.760
	tiempo	**3,206**	**1,686**	**1,107**	**2,465**	**2,270**
1.024^3	puntos	933.498	782.595	698.818	689.029	1.218.777
	triángulos	1.866.995	1.570.908	1.394.135	1.387.314	2.442.976
	tiempo	**4,400**	**4,497**	**3,364**	**4,512**	**6,422**
2.048^3	puntos			2.149.448	1.061.731	3.283.354
	triángulos			4.293.500	2.152.570	6.567.145
	tiempo			**13,110**	**8,825**	**20,637**
4.096^3	puntos				1.726.342	
	triángulos				3.463.322	
	tiempo				**11,243**	
8.192^3	puntos				2.508.245	
	triángulos				5.024.283	
	tiempo				**17,465**	

Tabla 21: coste en tiempo de las simplificaciones de diferentes modelos de menos de 10 millones de triángulos

	Modelo	Thai statuete	ciudad 15M	lucy	Puget Sound
	Puntos	4.999.996	6.107.478	14.027.872	16.785.409
	triángulos	10.000.000	15.601.856	28.055.742	33.554.432
grid					
256^3	Puntos	118.555	44.276	90.780	91.547
	triángulos	240.919	113.235	182.902	190.034
	tiempo	**1,680**	**3,011**	**3,977**	**4,984**
512^3	Puntos	451.199	137.922	351.327	362.345

	triángulos	907.865	332.334	705.207	735.245
	tiempo	**3,116**	**2,891**	**5,008**	**5,969**
1.024^3	Puntos	1.496.860	386.344	1.317.615	1.422.113
	triángulos	3.000.344	881.256	2.642.388	2.854.207
	tiempo	**8,128**	**4,129**	**10,046**	**11,311**
2.048^3	Puntos	3.442.412	779.960	4.555.618	5.235.018
	triángulos	6.887.683	1.682.503	9.128.466	10.467.345
	tiempo	**22,376**	**8,852**	**33,616**	**32,186**
4.096^3	Puntos	4.877.974	1.289.987	10.287.708	16.778.871
	triángulos	9.756.009	2.679.289	20.568.519	33.541.359
	tiempo	**29,518**	**9,028**	**71,750**	**105,611**
8.192^3	puntos		1.881.515	12.179.672	no se ha
	triángulos		3.822.405	24.356.752	podido crear
	tiempo		**12,423**	**88,987**	hash de cuco
16.384^3	puntos		2.575.495	13.105.189	16.785.409
	triángulos		5.158.613	26.209.546	33.554.432
	tiempo		**16,815**	**96,010**	**145,255**

Tabla 22: coste en tiempo de las simplificaciones de diferentes modelos de 10 millones o méas de triángulos

Se puede observar que el coste temporal de simplificación depende en gran medida del tamaño de la malla simplificada, además de la malla original: para un tamaño de grid de 1.024^3 la simplificación de la malla de 4 millones de triángulos de la estatua de neptuno a una de 1,6 millones ha tardado 4,5 segundos, los mismos que se ha tardado en simplificar la malla de 6 millones de triángulos del ejemplo "f1 car" a unos 1,4 millones. Los 2 millones de triángulos de más de la malla original "se han compensado" con los cerca de doscientos mil triángulos de menos en la malla simplificada.

También se ha desglosado los tiempos de simplificación para saber donde radicaba el coste del algoritmo y se ha visto que el tiempo se ha invertido en crear la tabla hash híbrida y en simplificar la malla, como se puede ver en esta tabla:

Tamaño grid / celdas ocupadas	Tiempo creación hash	Tiempo creación mapa QEF	Tiempo búsqueda óptimo	Tiempo simplificación malla	Total
256^3 / 90.780	0,452	2,842	0,020	1,062	4,436
512^3 / 351.327	0,944	2,822	0,060	1,762	5,680
1.024^3 / 1.317.615	3,149	2,967	0,132	4,184	10,512
2.048^3 / 4.555.618	11,344	4,473	0,361	14,295	30,673
4.096^3 / 10.287.708	27,980	7,062	0,792	35,496	71,750
8.192^3 / 12.179.672	34,402	8,105	0,912	45,038	88,987

Tabla 23: Distribución del coste de tiempo, en segundos, según las diferentes partes del algoritmo y la resolución del grid. Han sido tomados al simplificar el modelo de la estatua lucy.

Se puede observar que buena parte del tiempo se consume para crear la tabla de hash híbrida y para simplificar la malla.

Dentro de la creación del hash híbrido, la distribución de tiempos es la mostrada en la siguiente tabla:

Tamaño grid / celdas ocupadas	T creación hash nodos	Tiempo extracción de índices	T distrib. buckets < 512	Tiempo creación hash cuco	T verif. hash ok	Total creación hash
256^3 / 90.780	0,412	0,000	0,030	0,000	0,010	0,452
512^3 / 351.327	0,452	0,010	0,449	0,013	0,020	0,944
1.024^3 / 1.317.615	0,632	0,020	2,352	0,061	0,083	3,149
2.048^3 / 4.555.618	1,482	0,100	8,983	0,209	0,353	11,344
4.096^3 / 10.287.708	4,682	0,340	21,474	0,472	1,002	27,980
8.192^3 / 12.179.672	6,530	0,420	25,844	0,558	1,276	34,402

Tabla 24: Distribución del coste de tiempo, en segundos, para crear la tabla de has híbrida. Han sido tomados al simplificar el modelo de la estatua lucy.

Se puede apreciar que la mayor parte del tiempo de creación del hash híbrido lo toma el reparto de elementos en buckets de menos de 512 elementos. Hay que recordar que este tiempo también incluye la inicialización de los elementos a 0, para que más tarde se pueda acumular las funciones de error cuádricas y los vértices dentro de la misma celda.

Si se dibuja el tiempo en función del número de celdas ocupadas, se crea la siguiente gráfica:

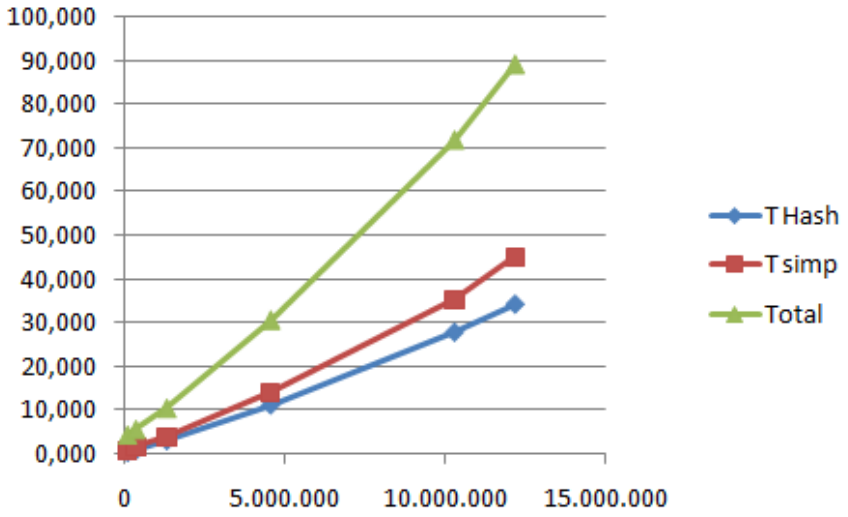


Figura 37: comparación entre el número de celdas ocupadas por el modelo lucy dependiendo del nivel de resolución del grid y el tiempo para crear el hash de cuco y simplificar la malla.

Parece que el coste es lineal!

También se ha comparado el rendimiento de la aplicación en tres plataformas:

- *plataforma 1*: Intel Quad Core Q9550 en MS Windows 7 de 64 bits y Linux Ubuntu 8.04.4 LTS y gcc 4.4.5;
- *plataforma 2*: Intel i7 920 en Linux Ubuntu 10.04.2 LTS y gcc 4.4.3;
- *plataforma 3*: Intel Pentium Dual SU4100 MS Windows 7 y Linux Ubuntu 10.04.2 y gcc 4.4.3;

El modelo escogido es la estatua thai de 10 millones de triángulos. El tamaño de las mallas simplificadas según el nivel de resolución del grid es éste:

Tamaño grid	Puntos	Triángulos	Líneas
Original	4.999.996	10.000.000	0
64^3	6.957	14.865	27
128^3	29.365	60.822	28
256^3	118.555	240.919	60
512^3	451.199	907.865	76
1.024^3	1.496.860	3.000.344	63
2.048^3	3.442.412	6.887.683	3
4.096^3	4.877.974	9.756.009	0

Tabla 25: tamaño de las mallas simplificadas de la estatua thai.

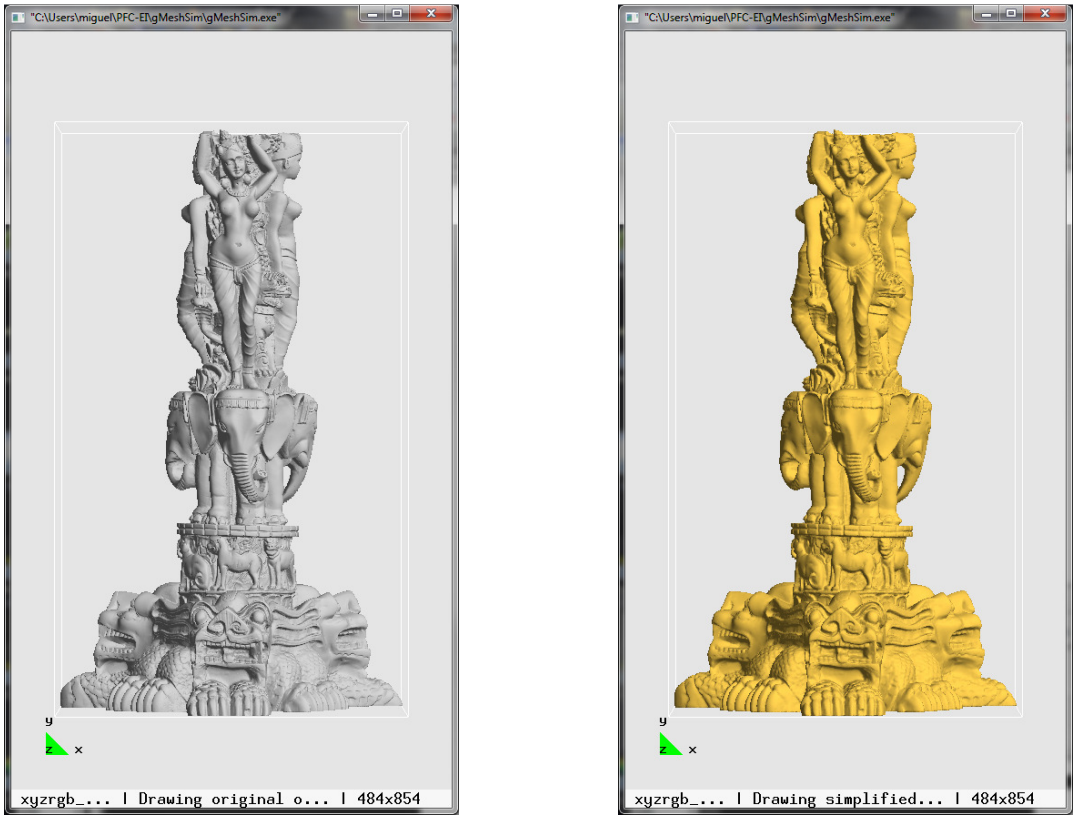


Figura 38: estatua thai original a la izquierda y simplificada con el grid de 512^3, aproximadamente un 10 % del tamaño de la malla original.

La primera tabla compara el coste en tiempo del algoritmo usando todos los núcleos posibles de las plataformas:

Tamaño grid	Plataforma 1 (Windows 7)	Plataforma 1 (Linux)	Plataforma 2 (Linux)	Plataforma 3 (Windows 7)	Plataforma 3 (Linux)
CPU	Intel Q9550	Intel Q9550	Intel i7-920	Intel SU4100	Intel SU4100
Núcleos	4	4	4 + HT	2	2
64^3	2,397	2,230	1,683	5,699	4,827
128^3	1,643	1,366	1,360	5,892	4,480
256^3	1,721	1,333	1,224	6,916	5,161
512^3	3,114	2,457	1,629	10,795	7,991
1.024^3	8,263	6,843	3,502	23,343	17,242
2.048^3	19,417	16,404	7,206	48,719	38,616
4.096^3	28,938	24,731	10,437	72,457	57,960

Tabla 26: Tiempos usando todos los núcleos de las tres plataformas.

Ahora se compara el rendimiento de un único núcleo de las diferentes plataformas:

Tamaño grid	Plataforma 1 (Windows 7)	Plataforma 1 (Linux)	Plataforma 2 (Linux)	Plataforma 3 (Windows 7)	Plataforma 3 (Linux)
CPU	Intel Q9550	Intel Q9550	Intel i7-920	Intel SU4100	Intel SU4100
Núcleos	1	1	1	1	1
64^3	2,964	2,964	2,381	6,521	4,842
128^3	3,146	3,146	2,527	6,983	5,211
256^3	3,671	3,671	2,916	8,175	6,300
512^3	5,990	5,990	4,375	12,683	10,369
1.024^3	14,103	14,103	9,270	28,117	24,256
2.048^3	33,436	33,436	20,627	63,144	56,436
4.096^3	51,246	51,246	31,324	94,744	85,732

Tabla 27: Tiempos usando un único núcleo de las tres plataformas.

4.3. Memoria

También he analizado el coste en memoria del algoritmo.

Antes de nada hay que advertir que el tamaño que se ha tomado para crear el hash de cluster_ids únicos es el número de puntos del modelo dividido por 60. Se había supuesto que el usuario normalmente desea simplificar el modelo a un 10 % de su tamaño original y en el hash de nodos, los buckets son de 6 elementos. Esta es una de las primera mejoras a realizar.

La siguiente tabla muestra la memoria requerida para construir el hash híbrido y después para simplificar la malla. También se muestra los bytes por celda ocupada como comparación.

Tamaño grid / celdas ocupadas	Memoria creación hash cuco (MB)	Bytes / nodo	Memoria simplificar (MB)	Bytes / nodo	Memoria total (MB)	Bytes / nodo
256^3 / 90.780	34,52	398,77	13,76	158,93	48,28	557,70
512^3 / 351.327	56,82	169,60	53,10	158,49	109,93	328,09
1.024^3 / 1.317.615	147,31	117,23	199,32	158,62	346,63	275,85
2.048^3 / 4.555.618	478,25	110,08	685,69	157,83	1.163,94	267,91
4.096^3 / 10.287.708	1.063,35	108,38	1.557,67	158,77	2.621,02	267,15
8.192^3 / 12.179.672	1.256,45	108,17	1.820,20	156,71	3.076,65	264,88
16.384^3 / 13.105.189	1.350,95	108,09	1.995,28	159,65	3.346,23	267,74
32768^3 / 13.566.701	1.398,03	108,05	2.064,41	159,56	3.462,43	267,61

Tabla 28: Coste en memoria del algoritmo de simplificación usando hash de cuco para el modelo lucy y diferentes resoluciones del modelo.

Si se desglosan las diferentes partes se puede ver que aún hay espacio para mejorar el algoritmo.

	Memoria creación hash de cuco			Bytes / nodo
Tamaño grid / celdas ocupadas	Hash QEF (de cuco)	Hash nodos	Total	
256^3 / 90.780	7,77	26,76	34,52	398,77
512^3 / 351.327	30,05	26,77	56,82	169,60
1.024^3 / 1.317.615	112,73	34,58	147,31	117,23
2.048^3 / 4.555.618	389,72	88,54	478,25	110,08
4.096^3 / 10.287.708	880,05	183,30	1.063,35	108,38
8.192^3 / 12.179.672	1.041,90	214,55	1.256,45	108,17
16.384^3 / 13.105.189	1.121,07	229,88	1.350,95	108,09
32768^3 / 13.566.701	1.160,54	237,49	1.398,03	108,05

Tabla 29: Desglose del consumo de memoria para la estructura de hash híbrido y la tabla has de *cluster_ids* únicos.

	Memoria durante la simplificación del modelo					Bytes / nodo
Tamaño grid / celdas ocupadas	Hash líneas	Hash triángulos	Lista triángulos	Lista puntos	Total	
256^3 / 90.780	5,55	4,72	2,79	0,69	48,28	158,93
512^3 / 351.327	21,41	18,25	10,76	2,68	109,93	158,49
1.024^3 / 1.317.615	80,35	68,60	40,32	10,05	346,63	158,62
2.048^3 / 4.555.618	277,01	234,63	139,29	34,76	1.163,94	157,83

4.096^3 / 10.287.708	633,80	531,54	313,85	78,49	2.621,02	158,77
8.192^3 / 12.179.672	736,46	619,16	371,66	92,92	3.076,65	156,71
16.384^3 / 13.105.189	809,91	685,45	399,93	99,98	3.346,23	159,65
32768^3 / 13.566.701	836,43	710,45	414,02	103,51	3.462,43	159,56

Tabla 30: Desglose del consumo de memoria durante la etapa de simplificación de la malla original.

Podemos observar que en la etapa de creación de la tabla de hash de cuco híbrida, la tendencia es a ocupar unos 108 bytes de memoria por nodo. El alto consumo inicial parece ser debido a la mala elección del tamaño de la tabla de cluster_ids únicos como ya he comentado al principio de esta sección.

También se puede obervar que para la simplificación del modelo, la memoria consumida es de unos 159 bytes por nodo.

En total, el consumo de memoria tiende a ser de unos 270 bytes por celda ocupada del grid de simplificación y no depende del tamaño del modelo original.

4.4. Experimentación con el hash híbrido

Como se ha comentado en el capítulo anterior, el algoritmo de hash propuesto por Alcántara propone primero repartir los elementos en buckets de 512 elementos como máximo pero con una ocupación media de 409 elementos por bucket. Su argumento era que necesitaba dejar espacio libre para poder tener éxito al crear las funciones de hash de cuco para cada bucket. Pero más adelante los buckets para las tablas de hash de cuco aumentaban de tamaño a 576 elementos, 573 en la implementación del proyecto.

Se ha hecho un experimento para probar si se puede aumentar la capacidad media de estos buckets, pues ya disponen de espacio para crear las funciones de hash de cuco.

Se ha ido aumentandola capacidad media de los buckets usando todos los primos que hay entre 409 y 512 para diferentes resoluciones del grid de simplificación.

La siguiente tabla muestra para cada modelo y tamaño de grid el primo más grande con el que se ha conseguido una creación exitosa de las funciones hash. Se ha considerado que el cálculo de las funciones hash falla después de 25 iteraciones para repartir los elementos y después de usar 25 semillas diferentes, como se ha explicado en el capítulo anterior.

	Grid de 256^3		Grid de 512^3		Grid de 1024^3	
Modelo	**# celdas**	**Primo**	**# celdas**	**Primo**	**# celdas**	**Primo**
bunny	35.727	467	35.931	467	35.944	457
armadillo	109.385	467	171.324	449	172.912	449
dragon	121.866	463	281.895	439	358.605	439
happy	101.024	461	260.573	443	426.117	439
cktest_240k	28.653	467	61.625	457	92.055	463
Rotor16_500k	124.100	467	262.461	449	276.112	443
Rotor16_1500k	137.603	467	485.310	463	755.724	421
gear_583k	178.671	479	291.828	457	291.840	443
maxilar	182.695	463	365.064	449	422.377	439
colada_797k	35.530	467	77.160	457	153.691	443
gargo	212.943	449	580.707	443	723.110	431
pensatore	248.940	457	752.382	439	933.498	439
xyzrgb_dragon	89.828	467	344.137	443	1.218.777	443
neptune	66.242	479	244.733	461	782.595	439
lucy	90.780	463	351.327	443	1.317.615	439
singapur	25.398	487	50.404	463	85.450	463
singapur 15M	44.276	461	137.922	467	386.344	457
angel	58.665	467	148.422	457	228.842	443
hand	68.361	463	215.081	443	277.314	433
statuette thai	118.555	449	451.199	457	1.496.860	433
manuscript	45.319	503	179.746	479	698.818	463
Blade	225.496	449	573.143	439	766.650	439
Plataforma	18.441	479	42.347	467	122.862	463
plat. iso 142	11.554	479	30.329	467	80.367	463
Telescope	88.336	463	115.526	449	144.000	449
f1 bound	83.739	479	331.411	491	689.029	461
Media		467,77		455,31		445,92

Tabla 31: Pruebas con diferentes factores de ocupación para los buckets de las tablas de hash de cuco. Según tamaño de grid y para diferentes modelos se muestra el primo más grande, menor de 512 para el que se consiguió una creación con éxito de las funciones de hash de cuco.

Habría que realizar más pruebas, o modificar el algoritmo para que, dependiendo del modelo, escoja el factor de ocupación. Pero en el caso de subir la ocupación media de 409 a 445, el espacio libre necesario se podría reducir del 40 % al 30 % y el factor de ocupación de la estructura pasaría del 71 % al 77 %.

5. Conclusiones

5.1. Aplicación gMeshSim

Se ha desarrollado en C++ una aplicación, llamada gMeshSim, multiplataforma con la que se ha probado diferentes algoritmos de simplificación y se ha experimentado con ellos. La aplicación está 'lista para usar' y presenta estas características:

- se puede simplificar modelos grandes;
- el espacio de memoria usado por el algoritmo de simplificación está limitado a 270 bytes por cada nodo de la malla original;
- el coste temporal es lineal:
    - se hace una pasada por los nodos de la malla original para construir el hash híbrido
    - se hace una parada por los triángulos de malla original para acumular las funciones de error cuádricas y los vértices en cada una de las celdas
    - se hace una última pasada por los triángulos para simplificar la malla original
- el formato de lectura y escritura es PLY;
- también se puede usar remotamente (tunel ssh + X11);
- depende de librerías de dominio público: OpenGL, GLUT y FLTK;
- escala aceptablemente bien en sistemas multiprocesadores;

simplifica una malla de 28 millones de triángulos en 4 segundos.

También se han desarrollado las herramientas para poder montar un Octree de simplificación como el desarrollado por Schaefer *[Schaefer2003]* y poder conseguir mallas con un error menor de 10e-5.

5.2. Estado del arte y mejoras realizadas

Se han estudiado diversos artículos para tener conocimiento del estado del arte en el tema de simplificación de mallas y sacar lo mejor de cada uno de ellos.

Respecto al artículo de DeCoro *[DeCoro2007]*:

- se ha aprovechado su enfoque de tres etapas para la simplificación de mallas;
- se han recuperado los triángulos colapsados en líneas que ayudan al reconocimiento visual del modelo;
- se ha controlado que el representante óptimo de la celda no se posicione fuera de ella, usando la media de los vértices en tal caso;
- se han eliminado las líneas y los triángulos repetidos;
- se han corregido las normales de los triángulos que cambian de orientación al simplificarlos para evitar errores visuales.

Respecto al artículo de Alcántara *[Alcantara2009]*:

- se ha usado su hash híbrido, combinación de hash perfecto modificado y hash de cuco, para poder almacenar eficientemente las celdas dispersas con información de un grid 3d cuasi vacio;
- se ha ampliado el algoritmo para poder trabajar con claves más grandes y así poder refinar los grid de simplificación más allá de 1024^3;
- se han realizado pruebas de capacidad media de los buckets y parece ser que se puede aumentar dicha media;
- con la combinación de las etapas de DeCoro y el hash de Alcántara se pueden conseguir mallas simplificadas con un error menor de 10e-5.

De los demás artículos se ha conseguido una mejor compresión del funcionamiento de los algoritmos de simplificación de mallas y de los criterios que se usan para la elección del representante óptimo de los clústeres.

5.3. Líneas futuras

Éste ha sido un trabajo que tendrá continuidad pues por un lado se incluirá en una librería y por otro se usará en una aplicación comercial.

Debido al alto coste de los semáforos, sobretodo usando en compilador MS VisualStudio, y a la cantidad de información que acumula cualquiera de las celdas del grid de simplificación, se ha estudiar si es realmente necesario usarlos para asegurar una correcta acumulación de información en la celda. En resoluciones bajas del grid, cada celda acumula información de cientos o de miles de vértices. Aunque sólo haya un vértice en una celda en concreto, la celda será visitada, en media, seis veces, una por cada triángulo adyacente a dicho vértice.

Una acción inmediata que se realizará en las próximas semanas es montar un octree con las dos estructuras implementadas:

- niveles 1..6 completos con el algoritmo desarrollado en la primera versión de este proyecto,
- niveles 7..N usando el hash espacial de cuco implementado en la segunda parte del proyecto,
- la profundidad máxima vendría dada por estos criterios:
    - tamaño mínimo de celda o tamaño mínimo de triángulo seleccionable por el usuario,
    - cantidad de puntos mínima en la celda, esto implica contar los puntos de cada celda al construir el hash de *cluster_ids* únicos,
    - cantidad de memoria disponible, la memoria máxima que necesita el algoritmo de hash espacial depende del tamaño del modelo.

Se intentará publicar un artículo cuando el octree esté montado.

Una de las ventajas del enfoque adoptado de DeCoro es que permite ampliar el algoritmo de simplificación fácilmente:

- incorporando atributos (resultados, colores, coordenadas de textura) a la simplificación ya sea como la propuesta por Garland en *[A.6]* o usando dos funciones de error cuádricas: una para la geometría y otra para los atributos y ponderarlas;
- experimentar con los criterios de búsqueda del representante óptimo del clúster, como los propuestos por Schaefer en *[Ju2002]* y *[Schaefer2002]* o por Lindstrom en *[Lindstrom2000]*;

Otras líneas futuras a valorar son:

- adaptar el algoritmo para tratar mallas que no quepan en memoria: "Out of core simplification"
- portar el algoritmo a las tarjetas gráficas, usando OpenCL o CUDA;
- simplificar mallas volumétricas como en el artículo de Garland *[Garland2004]*.
- mantener las aristas únicas y vivas del modelo original, mediante conservación de volúmen, u otro método como el propuesto por Schaeffer en *[Ju2002]* y *[Schaefer2002]*.

6. Bibliografía

[Aim2011] " Aim@SHAPE Shape Repository", http://shapes.aim-at-shape.net/viewgroup.php?id=803

[Alcantara2009] "Real-Time Parallel Hashing on the GPU " by Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens and Nina Amenta, ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH Asia 2009

[DeCoro2007] "Real-time Mesh Simplification Using the GPU" by Christopher DeCoro and Natalya Tatarchuk, Proceedings of the 2007 symposium on Interactive 3D graphics and games, ACM New York, NY, USA, 2007.

[Fox1992] "Practical minimal perfect hash functions for large databases" by Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, Amjad M. Daoud, CACM33(1), p. 105-121, 1992.

[Fredman1984] "Storing a spare table with O(1) worst case access time" by Michael L. Fredman, János Komlós and Endre Szemerédi, Journal of the ACM 31, 3 (July), 538-544, 1984.

[Garland1997] "Surface simplification using quadric error metrics" by Michael Garland and Paul S. Heckbert, ACM Siggraph Conference, 209-216, 1997.

[Garland1998] "Simplifying Surfaces with Color and Texture using Quadric Error Metrics" by Michael Garland and Paul S. Heckbert, Ninth IEEE Visualization 1998 (VIS '98) proceedings, p. 263-269, 1998.

[Garland2004] "Quadric-Based Simplification in Any Dimension" by Michael Garland and Yuan Zhou, Technical Report no. UIUCDCS-R-2004-2450, June 2004

[Gatech2011-2] "Large Geometric Models Archive", Georgia Institute of Technology, http://www.cc.gatech.edu/projects/large_models/

[GiD2011] GiD - the personal pre and postprocessor, http://www.gidhome.com

[Heckbert1999] "Optimal triangulation and quadric-based surface simplification" by Paul S. Heckbert and Michael Garland, Journal of Computational Geometry: Theory and Applications, vol. 14 no. 1-3, pages 49-65, November 1999.

[Ju2002] "Dual contouring of hermite data." Tao Ju, Frank Losasso, Scott Schaefer and Joe Warren, ACM Siggraph Conference 2002, 339-346, 2002.

[Lefebvre2006] "Perfect Spatial Hashing" by Sylvain Lefebvre and Hugues Hoppe, ACM Transactions on Graphics 25, 3 (July), 579–588, 2006.

[Lindstrom2000] "Out of core Simplification of Large Polygonal Models" by Peter Lindstrom, ACM Siggraph Conference, p. 259-262, 2000.

[Luebke2003] "Level of Detail for 3D Graphics" by David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson y Robert Huebner, published by Morgan Kaufmann, Elsevier, 2003

[Numerical2007] "Numerical Recipes, The art of Scientific Computing C++", Third edition, 2007, Cambridge Press

[Pagh2001] "Cuckoo hashing" by Rasmus Pagh and Flemming Friche Rodler, 9th Annual European Symposium on Algorithms, Springer, vol.2161 of Lecture Notes in Computer Science, 121-133, 2001. [Gatech2011] "The PLY File Format", http://www.cc.gatech.edu/projects/large_models/ply.html, searched on 2011 January

[Pasenau2009] "Informe sesión 1 laboratorio: Geometría en OpenGL" asignatura de Visualizción avanzada, Octubre 2.009

[Rossignac1993] "Multi-Resolution 3D Approximations for Rendering Complex Scenes" by Jarek R. Rossignac and Paul Borrell, in Modeling in Computer Graphics, pp. 455-465, Springer Verlag, 1993.

[Schaefer2002] "Dual Contouring: 'The SecretSauce'" by Scott Schaefer and Joe Warren, 2002

[Schaefer2003] "Adaptive Vertex Clustering Using Octrees" by Scott Schaefer and Joe Warren, proceedings of SIAM Geometric Design and Computation 2003, SIAM, New York, NY, USA, vol. 2, p. 491-500, 2003.

[Shilov2010] " ATI Steals Market Share from Nvidia Due to World-Class DirectX 11 Strategy Execution." by Anton Shilov, 07/29/2010, http://www.xbitlabs.com/news/graphics/display/20100729095348_ATI_Becomes_World_s_Largest_Supplier_of_Discrete_Graphics_Chips.html

[Standford2011] "The Stanford 3D Scanning Repository", http://graphics.stanford.edu/data/3Dscanrep

[Yilmaz2011] "Mesh Multiplication Technique With Surface Correction" by Erdal Yilmaz and Shahrouz Aliabadi, Parallel CFD 2011 congress, Barcelona, Catalonia, Spain, 2011.

7. Agradecimientos

Quisiera agradecer al Profesor Carlos Andujar por dirigir el proyecto, por sus sugerencias y recomendaciones, a pesar de su dilatada carga de trabajo.

También quisiera agradecer el soporte y las facilidades ofrecidas por CIMNE para poder desarrollar este proyecto de fin de carrera.

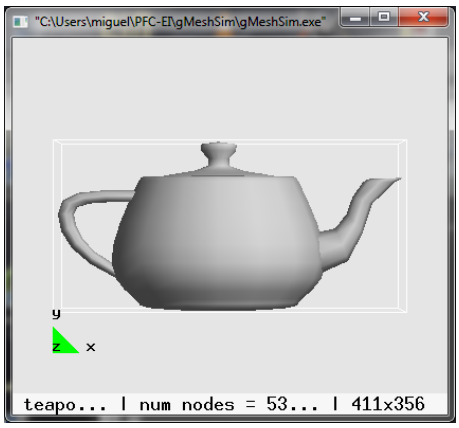
Y sobretodo quisiera agradecer a mi familia y mis amigos por la paciencia, comprensión y soporte que han tenido conmigo estos últimos seis meses.

8. Anexo

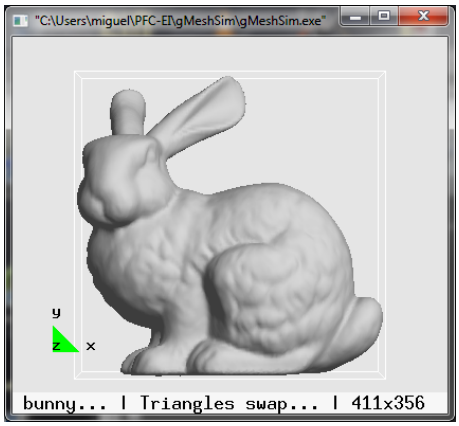
Modelos usados

La siguiente lista muestra los modelos usados para validar la aplicación gMeshSim y su tamaño. Buena parte de estos modelos se han obtenido del repositorio de Standford University *[*Standford2011*]*, de Aim@Shape *[*Aim2011*]*, de Georgia Institute of Technology *[*Gatech2011-2*]*, de prácticas de laboratorio, de modelos de simulaciones realizados en CIMNE o han sido creados expresamente para esta aplicación, como el ejemplo del cilindro o del rotor.

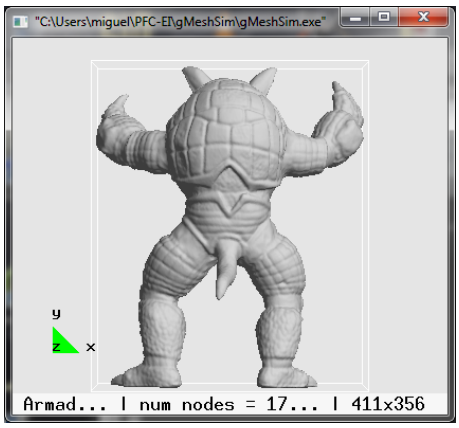
Modelo	Puntos	Triángulos
bunny	35.947	69.451
armadillo	172.974	345.944
dragon	437.645	871.414
happy	543.652	1.087.716
cilindro_z	14.722	29.440
cilindro_z_5g.	14.715	29.426
teapot	530	1.024
cktest_240k	118.762	242.688
Rotor16_90k	44.836	89.668
Rotor16_500k	277.934	555.864
Rotor16_1500k	768.112	1.536.220
gear_583k	291.840	583.676
maxilar	457.164	855.562
colada_797k	398.692	797.408
gargo	863.210	1.726.420
pensatore	997.875	1.995.746
xyzrgb_dragon	3.609.600	7.219.045
neptune	2.003.932	4.007.872
lucy	14.027.872	28.055.742
ciudad	402.587	975.116
ciudad 15M	6.107.478	15.601.856
ps_height	16.785.409	33.554.432
horse	48.485	96.966
angel	237.019	474.048
hand	327.323	654.666
statuette thai	4.999.996	10.000.000
manuscript	2.155.617	4.305.818
blade	882.954	1.765.388
plataforma	198.763	397.546
plat. iso 142	326.770	913.474
telescope	391.984	783.984
f1 bound	3.005.848	6.011.696



Teapot

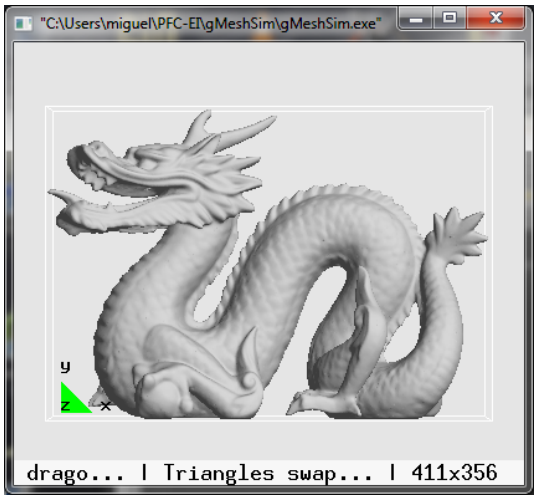


Bunny

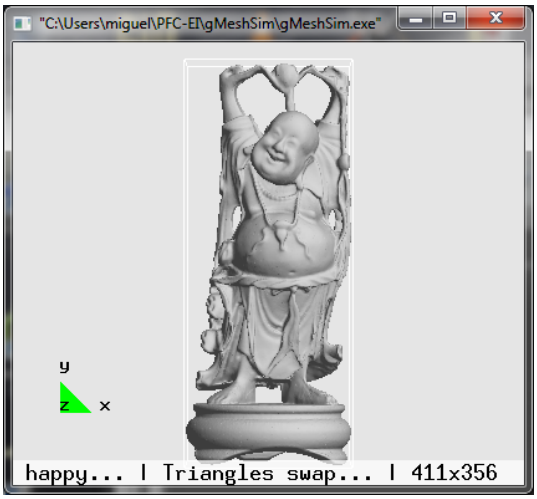


Armadillo

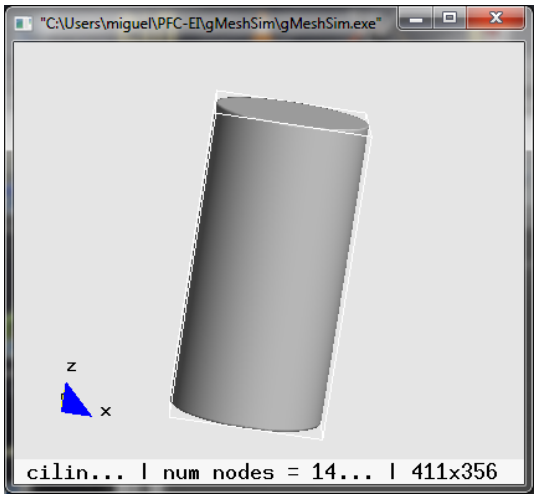
Tabla 32: Lista de modelos usados en el proyecto



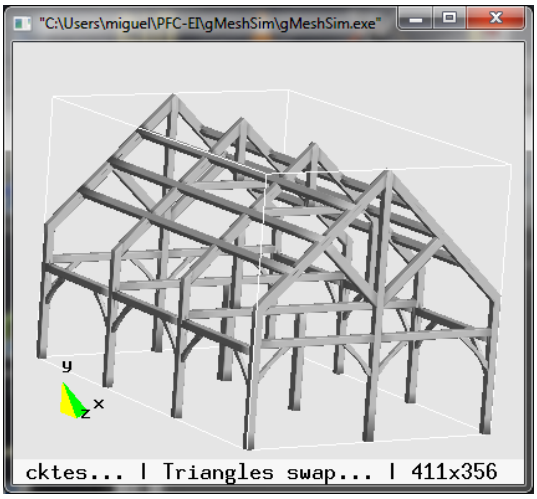
Dragon



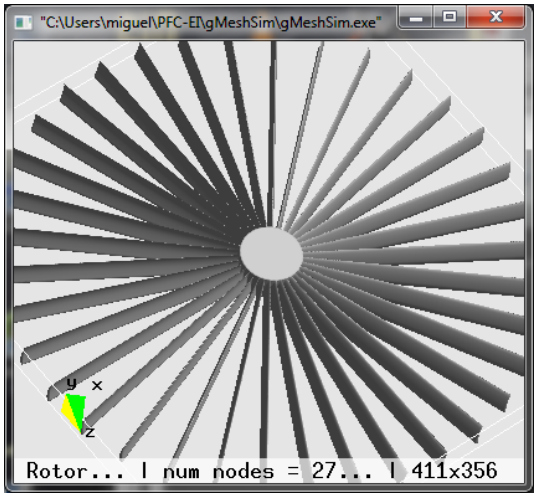
Happy budha



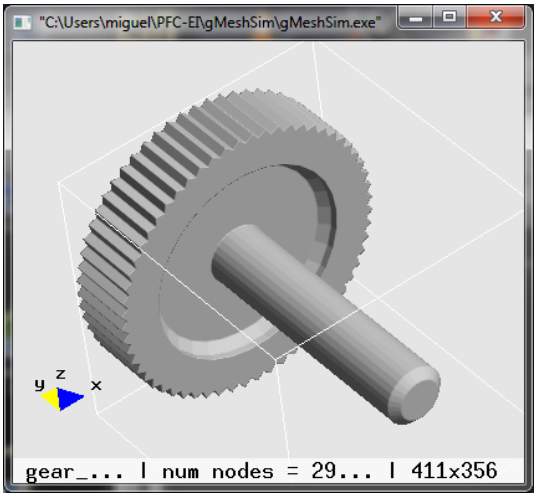
Cilindro, cilindro_5g.



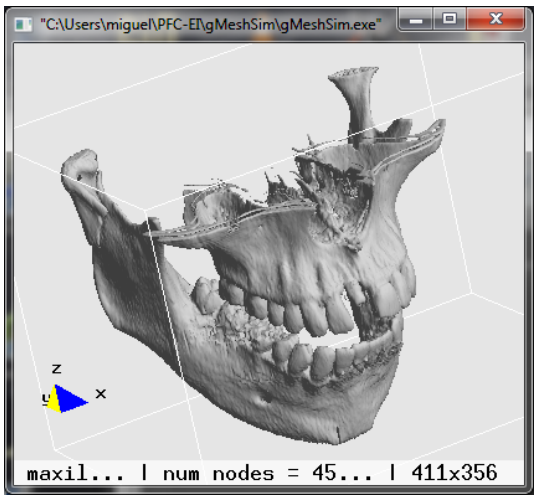
CkTest_240k



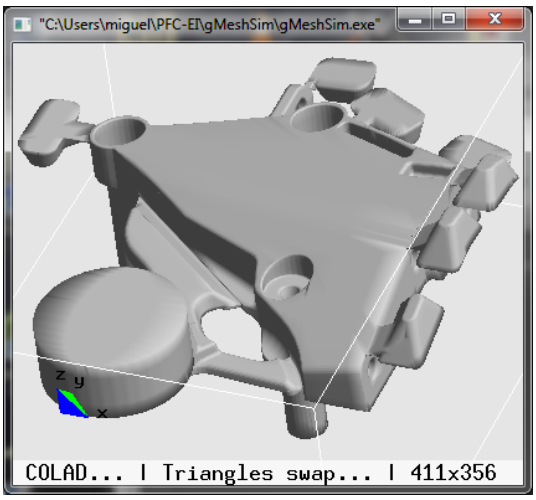
Rotor16_90k, Rotor16_500k, Rotor16_1500k



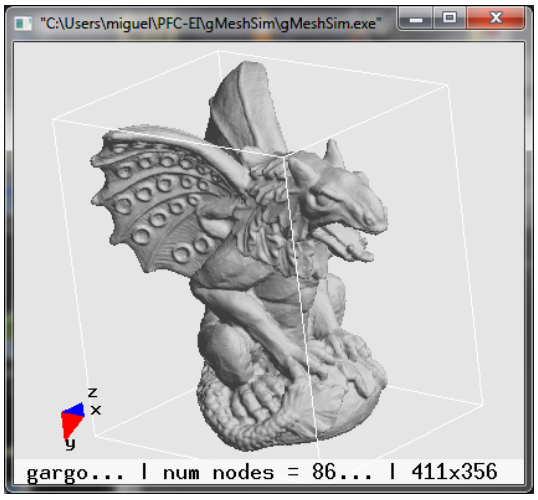
Gear_583k



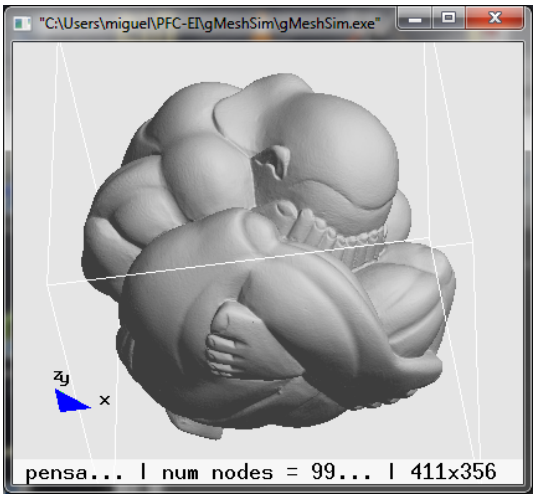
Maxilar



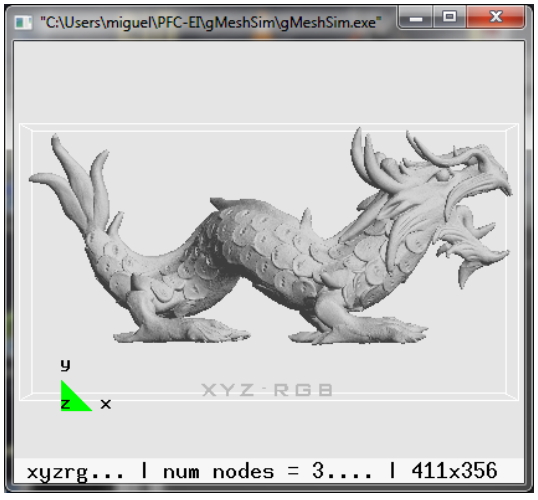
Colada_797k digitalización de una pieza de fundición



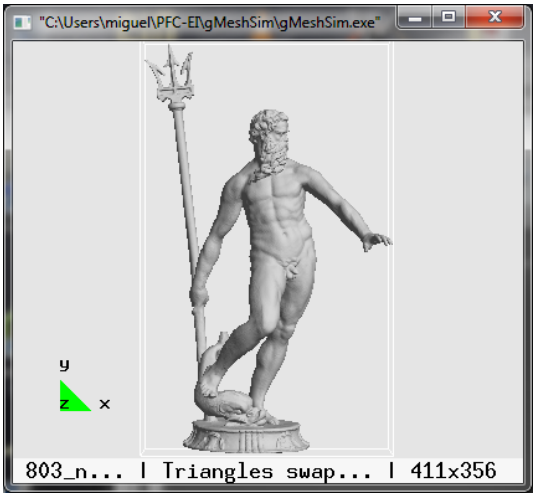
Gargoyle



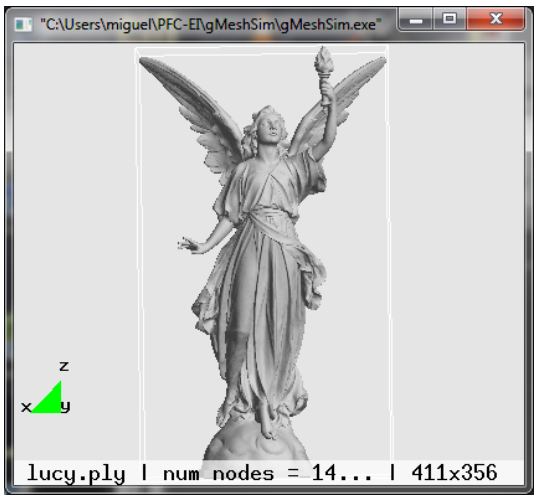
Pensatore



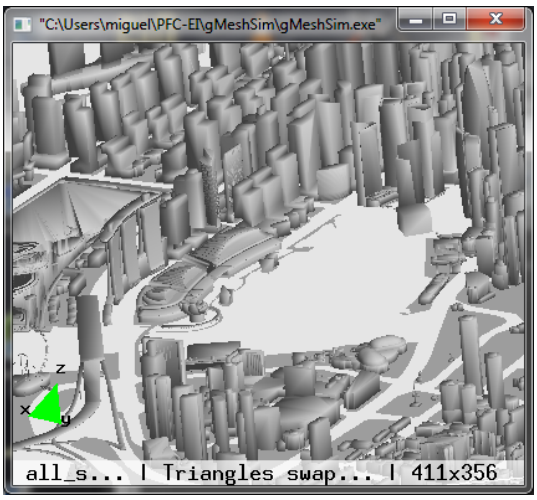
Xyzrgb dragon



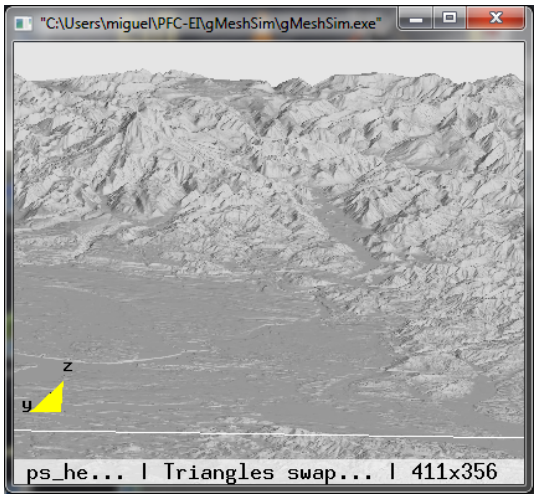
803 neptune



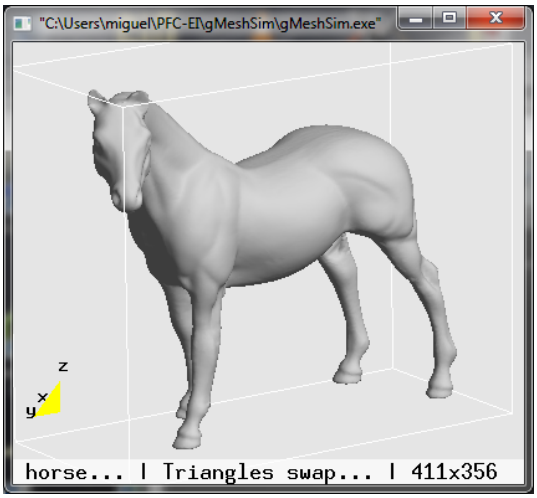
Lucy



Ciudad, ciudad 15M

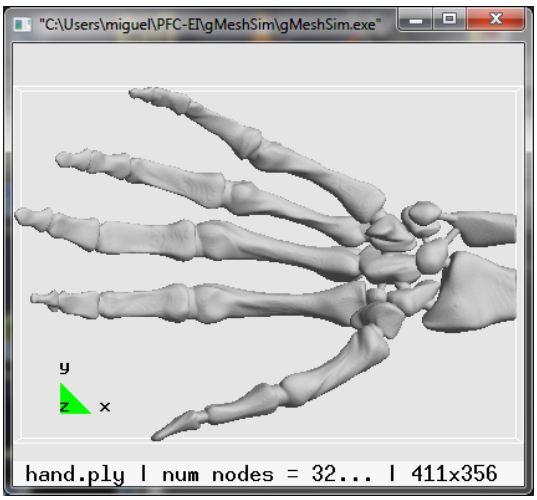


Puget Sound digitalización regular de un terreno

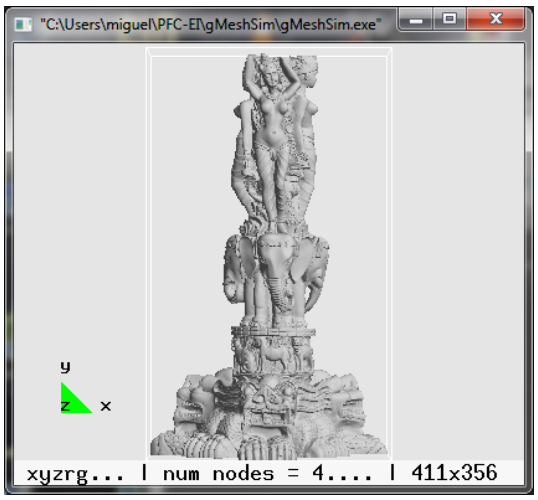


Horse

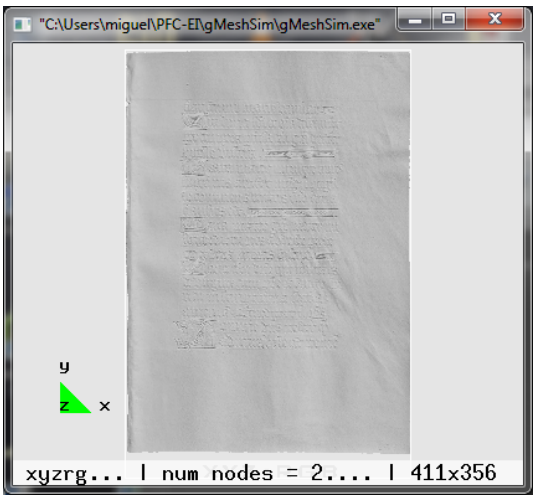
Angel



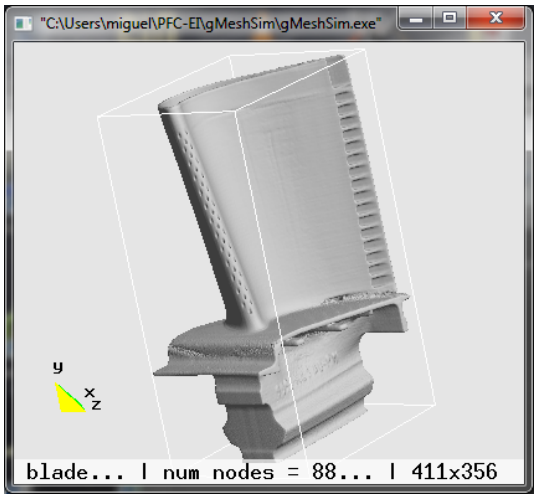
Hand



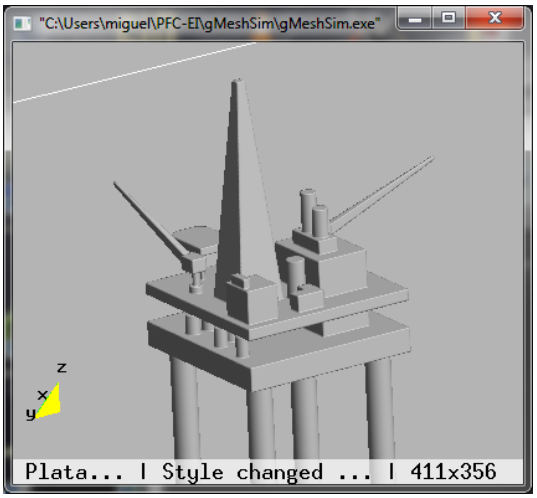
Statuette Thai



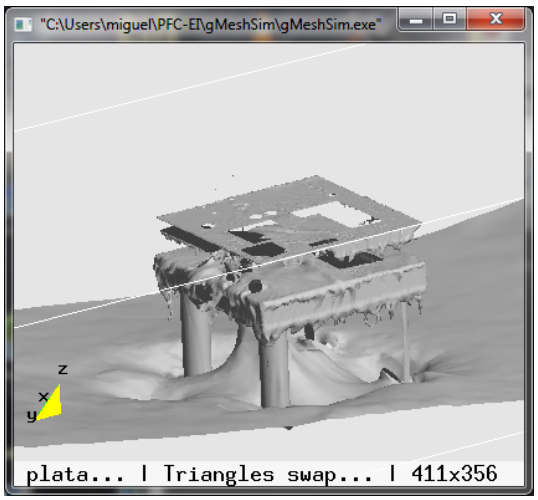
Manuscript



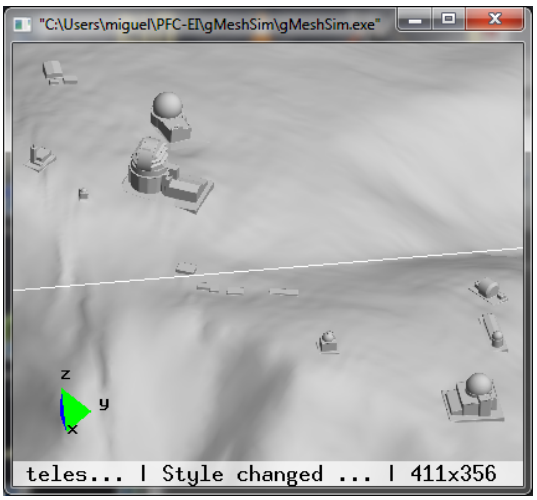
Blade



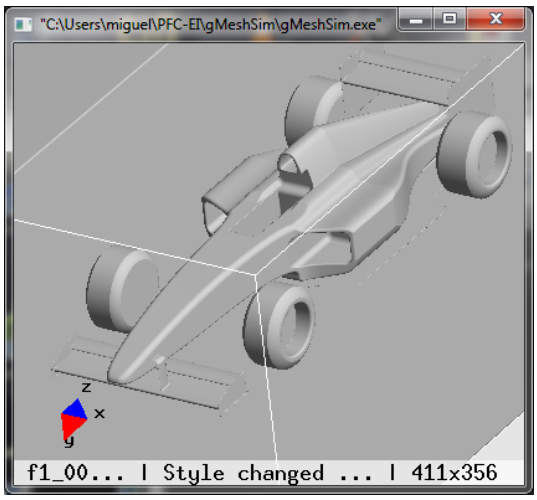
Plataforma: malla anisotrópica



Iso-superficie de oleaje de la plataforma



Telescopio: malla anisotrópica



Coche f1: malla anisotrópica

Figura 39: las imágenes de los modelos usados en este proyecto

Además de estos modelos, ocasionalmente se ha probado el algoritmo con otros modelos pero no se han considerado suficientemente relevantes para incluirlos en este informe.