Programa Oficial de Postgrado en
Desarrollo de Sistemas Software
Departamento de Lenguajes y Sistemas Informáticos

Universidad de Granada

BP-Octree: una estructura jerárquica de volúmenes envolventes.

Memoria presentada para la obtención del grado de
Doctor
presentada por

Francisco Javier Melero Rus

Directores
Dr. Pedro Cano Olivares

Dr. Juan Carlos Torres Cantero

Granada. Noviembre de 2008

Editor: Editorial de la Universidad de Granada
Autor: Francisco Javier Melero Rus
D.L.: GR. 2828-2008
ISBN: 978-84-691-8356-4

La memoria *BP-Octree: una estructura jerárquica de volúmenes envolventes* que presenta D. Francisco Javier Melero Rus para optar al grado de Doctor por la Universidad de Granada ha sido realizada dentro del programa de doctorado *Desarrollo de Sistemas Software* del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada, bajo la dirección de los doctores D. Pedro Cano Olivares y D. Juan Carlos Torres Cantero.

Granada, noviembre de 2007

El doctorando

Francisco Javier Melero Rus

Los Directores

Pedro Cano Olivares　　　　Juan Carlos Torres Cantero

A Mª Carmen,

que esta memoria compense

el poco tiempo dedicado a nosotros.

Abbá,

sin Ti nada habría sido posible.

Agradecimientos

Tras varios años de esfuerzo, son numerosas las personas que han aportado su granito de arena a que este trabajo vea la luz. Unos con sus escuchas atentas y sus palabras de ánimo, otros con sus ideas para resolver los numerosos problemas encontrados, algunos con críticas constructivas y los más expertos y veteranos con sus disertaciones sobre la importancia y necesidad de acabar la tesis. Sería temerario poner una lista exhaustiva, puesto que lo más probable es que involuntariamente me saltara a alguien. Seguro que al leer estas líneas cada uno sabrá reconocerse. A todos, gracias. Pero permítanme que detalle algunas referencias obligadas.

A mis directores les debo que este trabajo se publique. No soy una persona fácil de dirigir, y lo han sabido hacer estupendamente. Juan Carlos con su paciencia ante mi tozudez y mis irrupciones en el despacho a deshoras, y Pedro con sus revisiones ortográficamente milimétricas. Gracias a ambos.

Mi compañero de despacho Jose Miguel, mis *vecinos* Rosana y Alejandro y muchos otros compañeros de departamento han tenido que aguantar mis malos ratos por las numerosas frustraciones que la programación produce. Gracias, muchachos.

En las primeras semanas, Germán ayudó a montar el esqueleto de la interfaz gráfica para las pruebas, y en las últimas semanas, Paco ha colaborado en dar vida a la estructura de datos con una aplicación con el dispositivo háptico. Gracias a los dos *artistas*.

Parte de este trabajo se realizó en el *Max Planck Institure für Informatik* de Saarbrücken, Alemania. Gracias al prof. Hans-Peter Seidel por su invitación y gestiones para que dicha estancia fuese posible. En ese tiempo, las reflexiones del prof. Gumhold fueron imprescindibles para el resultado final. Gracias, Stefan, por tu implicación.

Gracias a los profesores Fabio Ganovelli, del ISTI/CNR de Italia, y Joaquim Jorge, del INESC de Portugal, por sus informes y comentarios sobre mi trabajo.

Finalmente, como no podia ser menos, agradecer a mis padres Ana y Diego por su esfuerzo en sacarnos adelante a mi hermano Luis y a mí.

Este trabajo ha estado parcialmente financiado por una beca del plan propio de investigación de la Universidad de Granada, una beca de formación de personal docente e investigador de la Junta de Andalucía, una beca de formación de profesorado universitario del Ministerio de Educación y Ciencia, por los proyectos de investigación TIC2001-2099-C03-02, TIN2004-06326-C03-02 y TIN2007-67474-C03-02 del Gobierno de España, así como por el proyecto de investigación PE-TIC-401 de la Junta de Andalucía.

Índice

1. Motivación y presentación de la memoria 1
1.1. Objetivos . . . 2
1.2. Organización de la memoria . . . 3

2. Introducción 5
2.1. Modelado de sólidos. . . . 5
2.1.1. El proceso de modelado . . . 6
2.2. Esquemas de representación. . . . 8
2.2.1. Representación basada en fronteras (B-Rep) . . . 10
2.2.2. Enumeraciones espaciales . . . 13
2.2.2.1. Octree . . . 14
2.2.2.2. Extensiones al Octree. Polytrees, Extended Octrees, SP-Octrees. 17
2.2.2.3. BSP Tree . . . 19

3. BP-Octree 21
3.1. Fundamentos . . . 23
3.2. Creación del índice espacial . . . 26
3.2.1. Código Morton . . . 27
3.2.2. Determinación de la profundidad del árbol . . . 28
3.2.3. Asignación de los polígonos a nodos. Algoritmo 3DDDA. . . . 28
3.2.4. Creación de los nodos hoja. . . . 29
3.3. Creación del árbol. . . . 29
3.3.1. Compactación de nodos . . . 30
3.4. Creación de la jerarquía de planos envolventes. . . . 33
3.4.1. Cálculo de planos envolventes en los nodos hoja. . . . 34
3.4.1.1. Etiquetado de las esquinas de los nodos. . . . 35
3.4.2. Selección de planos relevantes. . . . 37
3.4.3. Recorte de paralelepípedo con el conjunto de planos. . . . 39
3.4.4. Cálculo de planos envolventes en los nodos grises. . . . 40
3.4.5. Creación de nodos negros y blancos. . . . 41
3.5. Tiempos de construcción. . . . 42
3.5.1. Influencia del parámetro K . . . 45
3.6. Almacenamiento en memoria externa. . . . 54

4. BP-Octree como estructura multiresolución **59**
4.1. El BP-Octree como estructura multiresolución 60
4.1.1. Dibujado del BP-Octree 61
4.1.2. Dibujado de un nodo 61
4.1.3. Resultados 62
4.2. Transmisión progresiva 74
4.3. Visualización adaptativa 90
4.4. Uso de impostores en BP-Octrees 91
4.4.1. Aplicación de *impostores* 91
4.4.2. Generación de *impostores* 92
4.4.3. Visualización del BP-Octree con *impostores* 93

5. Detección de colisiones **95**
5.1. Introducción. 95
5.1.1. Sistemas de detección de colisiones. 95
5.1.1.1. Volúmenes envolventes. 96
5.1.1.2. Jerarquías de volúmenes envolventes. 97
5.1.1.3. Técnicas de división espacial. 99
5.2. Detección de colisiones con el BP-Octree. 99
5.2.1. Inclusión punto-en-sólido. 99
5.2.2. Detección de distancias y direcciones de colisión. 102
5.2.3. Aplicación del esquema para interacción háptica. 108

6. Conclusiones y trabajos futuros. **111**
6.1. Conclusiones y principales aportaciones. 111
6.2. Trabajos futuros. 112

A. English summary **113**
A.1. BP-Octree. Data Structure Overview 114
A.2. Building the BP-Octree 116
A.2.1. Spatial Indexation 117
A.2.1.1. Morton Code 117
A.2.1.2. Defining Maximum Depth 118
A.2.1.3. Addressing Polygons 119
A.2.2. Creation and coalescing of leaf nodes. 120
A.2.3. Computing Bounding Volumes 121
A.2.3.1. Labelling of leaf nodes corners. 123
A.2.3.2. Creation of black and white nodes. 125
A.2.3.3. Hierarchy of bounding volumes. 126
A.2.4. Building times 128
A.2.4.1. How the k parameter influences building times 128
A.2.5. External memory storing of BP-Octree 132
A.3. Using the BP-Octree as LOD Scheme 134
A.3.0.1. Volume of the BP-Octree at each level 135
A.3.1. Adaptive Visualization 138
A.3.2. Visualization of BP-Octree Nodes 138
A.3.3. Improved Visualization using Impostors 139

A.3.3.1. Applying *Impostors* 139
A.3.3.2. Impostors Generation 139
A.3.3.3. Impostors Application 140
A.4. Point Classification 142
A.5. Distance-to-solid computation 143
A.6. Haptic interaction with large polygonal models. 146
A.7. Conclusions and future works. 148
A.7.1. Future works. 148

Bibliografía **149**

Índice de figuras

2.1. Modelos matemáticos de un cono y un cilindro representados como la intersección de varios semiespacios. 6
2.2. Relación entre sólidos matemáticos y sus representaciones computacionales. . 8
2.3. Elementos B-Rep para definir un cubo. 10
2.4. Un toroide no es un poliedro simple, al no ser homeomorfo a una esfera. . . . 11
2.5. Representaciones de frontera basadas en aristas. Información almacenada para una única arista/semiarista. 13
2.6. Representaciones mediante enumeración espacial. 14
2.7. Octree. Subdivisión espacial a la izquierda. A la derecha, árbol representando los nodos existentes. 15
2.8. Aliasing debido al carácter discreto del octree. 16
2.9. Representación de un sólido (a la izquierda), mediane un octree (columna central) y un Extended Octree (derecha) [Can04] 17
2.10. Nodos blanco, negro, convexo, cóncavo y vértice del SP-Octree. 18
2.11. Stanford Bunny representado con SP-Octree. Cada color indica un tipo de nodo. 18
2.12. BSP Tree sólido en 2D 19

3.1. Detalle de un nodo hoja. En verde la geometría real. Cyan la envolvente convexa almacenada. En morado, los planos ficticios del nodo. 23
3.2. Esquema general de la estructura BP-Octree. 24
3.3. Seis primeros niveles del BP-Octree para el modelo Stanford Bunny. 25
3.4. El punto V_2 es clasificado en un voxel que no existe. 26
3.5. Numeración de los nodos 27
3.6. Adición de unos bits de nivel al código Morton 27
3.7. Un error en el algoritmo de reparto de triángulos (superficie azul) provoca que falte uno en el nodo (verde). 28
3.8. Representación gráfica del algoritmo de voxelización de triángulos. En azul los segmentos producidos por el corte con el plano P. En rojo, los puntos usados para determinar los voxels por los que pasa el triángulo. 30
3.9. Nodos creados hasta llegar al nodo hoja $17|354265701_8$ 31
3.10. Selección de planos envolventes. En cyan el volumen envolvente. 33
3.11. Envolvente resultante en un nodo hoja. 34

3.12. Configuración errónea de planos envolventes (verde). La superficie del sólido, en azul, es una concavidad formada por todas las caras que comparten un mismo vértice, estando contenido éste en el interior del voxel y los demás vértices fuera. 35
3.13. Esquema 2D de lo representado en la figura 3.12 35
3.14. Resolución de la inclusión de las esquinas del nodo 36
3.15. Corte arista con nodo 37
3.16. Alta densidad de planos debido a áreas del modelo extremadamente convexas. 38
3.17. Efectos del algoritmo k-medoides. 38
3.18. Recorte de la malla del nodo para crear la superficie envolvente. 39
3.19. Envolvente de un nodo gris y sus nodos hijos. 40
3.20. Etiquetado de esquinas de nodo 41
3.21. Corte del modelo Stanford Bunny donde se aprecia la superficie del sólido y la envolvente generada al máximo nivel de detalle. 41
3.22. A la izquierda, volumen envolvente en un nodo gris. El punto P puede estar fuera o dentro del sólido, según la configuración de la superficie del sólido, que da lugar a una misma envolvente en un nivel superior. 42
3.23. Tiempos de construcción del BP-Octree relacionado con el número de polígonos de cada modelo. 43
3.24. Tiempos de construcción del BP-Octre para cada uno de los modelos. 44
3.25. Reducción del tiempo de cálculo de envolventes variando el parámetro K. . . 46
3.26. Aspecto visual del BP-Octree para el modelo Blade con $k = 0{,}01$ y $k = 0{,}0005$. 48
3.27. Número medio de planos por nodo para el modelo Stanford Dragon 49
3.28. Porcentaje de planos seleccionados sobre el máximo permitido en el nodo raíz. 50
3.29. Porcentaje de planos seleccionados sobre el máximo permitido por nodo (nivel 1). 50
3.30. Influencia del parámetro K en el volumen del modelo Stanford Dragon. 51
3.31. Influencia del parámetro K en el volumen del modelo Happy Budda. 52
3.32. Influencia del parámetro K en el volumen del modelo Blade. 53
3.33. Stanford bunny *(71.040 polígonos)* 57
3.34. Golf *(100.243 polígonos)* 57
3.35. Armadillo *(150.000 polígonos)* 57
3.36. Egipcio *(161.909 polígonos)* 57
3.37. Teeth *(233.204 polígonos)* 57
3.38. Fertility *(483.226 polígonos)* 57
3.39. Angelo *(674.764 polígonos)* 58
3.40. Phlegmatic Dragon *(715.933 polígonos)* 58

3.41. Stanford Dragon
(871.414 polígonos) 58
3.42. Happy Buddha
(1.087.716 polígonos) 58
3.43. Blade
(1.765.388 polígonos) 58
3.44. Asian Dragon
(7.218.906 polígonos) 58

4.1. Tres resoluciones de un modelo poligonal obtenidas de una única estructura, las mallas progressivas [Hop96] 60
4.2. Nivel de detalle dependiente del observador [Hop97]. 60
4.3. Huecos entre nodos adyacentes. 62
4.4. Agujeros en las paredes de los nodos por la no coincidencia de convexidades en nodos adyacentes. 62
4.5. Modelo *Stanford Bunny* multirresolución. 63
4.6. Modelo *Armadillo* multirresolución. 64
4.7. Modelo *Egipcio* multirresolución. 65
4.8. Modelo *Teeth* multirresolución. 66
4.9. Modelo *Fertility* multirresolución. 67
4.10. Modelo *Angelo* multirresolución. 68
4.11. Modelo *Phlegmatic Dragon* multirresolución. 69
4.12. Modelo *Stanford Dragon* multirresolución. 70
4.13. Modelo *Happy Budda* multirresolución. 71
4.14. Modelo *Blade* multirresolución. 72
4.15. Modelo *Asian Dragon* multirresolución. 73
4.16. Distintos modelos a nivel 4, por debajo del umbral que supone el tamaño total de la geometría original. 75
4.17. Volumen por nivel de BP-Octree. 76
4.18. Transmisión progresiva para el modelo Stanford Bunny. 77
4.19. Transmisión progresiva para el modelo Golf. 78
4.20. Transmisión progresiva para el modelo Armadillo. 79
4.21. Transmisión progresiva para el modelo Egipcio. 80
4.22. Transmisión progresiva para el modelo Teeth. 81
4.23. Transmisión progresiva para el modelo Angelo. 82
4.24. Transmisión progresiva para el modelo Phlegmatic Dragon. 83
4.25. Transmisión progresiva para el modelo Stanford Dragon. 84
4.26. Transmisión progresiva para el modelo Happy Budda. 85
4.27. Transmisión progresiva para el modelo Blade. 86
4.28. Transmisión progresiva para el modelo Asian Dragon. 87
4.29. Influencia de k en la transmisión progresiva del modelo.Verde, $k = 0{,}01$. Rosa: $k = 0{,}0005$ 88
4.30. Influencia de k en la transmisión progresiva del modelo, sin el envío de la geometría. Verde, $k = 0{,}01$. Rosa: $k = 0{,}0005$ 89
4.31. Visualización adaptativa de los modelos *Fertility* y *Teeth*. 90
4.32. Transformaciones que se aplican a los vértices de la geometría (para su visualización y la obtención de coordenadas de textura). 91

4.33. Esfera de nivel 4 usada para tomar los *impostores* del modelo. 93
4.34. Visualización de un SP-Octree sin *impostores* (superior) y con *impostores* (inferior). 94
4.35. Visualización con *impostores* en el nivel 1 (fila central), y sin usar *impostores* en el nivel 1 (inferior). En la fila superior, SP-Octree a máximo detalle, representando la geometría original. 94

5.1. Distintos volúmenes envolventes para una nube de puntos. 96
5.2. Jerarquías de volúmenes envolventes 98
5.3. Intersección segmento –cyan– con sólido –azul– 100
5.4. Resultados visuales del test de inclusión punto en sólido. 103
5.5. Milisegundos necesarios para un test punto-en-sólido 104
5.6. Velocidad test distancia a solido 105
5.7. Tiempo de respuesta por consulta en función del número de polígonso. 106
5.8. Mapa de distancias para el modelo Happy Buddha. 106
5.9. Mapa de distancias para el modelo Asian Dragon, de 7M polígonos. 107
5.10. Interfaz háptico de seis grados de libertad 108
5.11. Aplicación de pintura háptica. Graffiti sobre los modelos Happy Buddha (1M triángulos) y Armadillo (150K triángulos). 109
5.12. Interacción háptica con el modelo Dragon de 7M de triángulos. 109

A.1. Example of node. Green, model geometry; cyan, computed bounding. 113
A.2. Phlegmatic Dragon and its bounding planes (cyan) at root node (a) and at level 4 of the BP-Octree (b). 114
A.3. General Scheme of BP-Octree data structure. 115
A.4. Boundings at level 0. 116
A.5. High density of planes due to strong convex bounding areas of the model. . . 117
A.6. Point V_2 is addressed in a wrong voxel. 117
A.7. Morton-code based nodes numbering 118
A.8. Morton-based locational code: a) Depth-first ordering; b) breadth-first ordering 119
A.9. Triangle voxelizing algorithm. *Blue:* segments produced by intersection with a plane P. *Red:* points used to detect traversed nodes. 119
A.10. A wrong triangles -blue- voxelization leads to surface inconsistencies in nodes. 120
A.11. Gray nodes created to reach leaf $17|354265701_8$ 121
A.12. Selecting bounding planes. 122
A.13. Effects of applying a k-medoids algorithm over convex surfaces. 122
A.14. Bounding volume at leaf node. 123
A.15. Error in bounding planes (green) configuration. 124
A.16. 2D scheme of Figure A.15 124
A.17. Labelling node corners when there are no triangles intersecting node edges. In blue, the volume defined by the model. 124
A.18. Black nodes (violet) allow to classify points in the interior of the model. . . . 125
A.19. Point P cannot be properly classified without having black or white nodes. . 125
A.20. Levels of the *Happy Buddha* BP-Octree. 126
A.21. Detail of the *Happy Buddha* BP-Octree. 127
A.22. BP-Octree building times with respect to number of polygons of original model. 128
A.23. BP-Octree building times per stages. 129

A.24. Bounding planes computing time changes as varying K. 131
A.25. Several models at level 4, using less than 10 % of planes. 134
A.26. Accumulated byte count for the progressive transmission of Happy Buddha. The dashed line shows the size of the original geometry. 136
A.27. Volume of BP-Octree representation with respect to original model. 137
A.28. Adaptive visualization of Fertility and Teeth. 138
A.29. Node mesh clipping 138
A.30. Sphere with 258 vertexes, used to take the impostors. 139
A.31. SP-Octree visualization without impostors (upper) and using impostors (lower). 141
A.32. SP-Octree at maximum LOD (upper), without impostors at level 1 (centre) and with impostors at level 1 (lower) 141
A.33. Intersection of segment –cyan– with surface –dark blue. 143
A.34. Visual display of point-in-solid test results. 144
A.35. Query time for distance test using different approaches. 145
A.36. Query time depending on the number of polygons. 145
A.37. Distance field for several models. Black points are those closer than 0.01 % of model width. 146
A.38. Closer look at distance values in Dragon model. 147
A.39. Haptic painting application. Graffiti on the Happy Buddha (1M triangles) and Armadillo (150K triangles) model. 147

Índice de tablas

3.1. Número de hojas para el modelo Stanford Bunny 32
3.2. Número de hojas para el modelo Fertility 32
3.3. Tiempos de construcción del BP-Octree para diversos modelos. 43
3.4. Influencia del parámetro k. 47
3.5. Planos por nodo según k 49
3.6. Volumen del BP-Octree para el modelo Stanford Dragon en función de k . . . 51
3.7. Volumen del BP-Octree para el modelo Happy Budda en función de k 52
3.8. Volumen del BP-Octree para el modelo Blade en función de k 53
3.9. KBytes necesarios para almacenar el BP-Octree relativo a cada modelo, y ratio ($.bpn + .bpl$)/$.ply$. 55

4.1. Número de planos en cada nivel del BP-Octree. 74
4.2. Porcentaje de referencias a planos en cada nivel sobre el total de polígonos del modelo. 74
4.3. Volumen representado del sólido en cada nivel (siendo el BREP el 100 %). . . 75
4.4. Datos del modelo Stanford Bunny. 77
4.5. Datos del modelo Golf. 78
4.6. Datos del modelo Armadillo. 79
4.7. Datos del modelo Egipcio. 80
4.8. Datos del modelo Teeth. 81
4.9. Datos del modelo Angelo. 82
4.10. Datos del modelo Phlegmatic Dragon. 83
4.11. Datos del modelo Stanford Dragon. 84
4.12. Datos del modelo Happy Budda. 85
4.13. Datos del modelo Blade. 86
4.14. Datos del modelo Asian Dragon. 87

5.1. Resultados test de inclusión punto en sólido. 102
5.2. Comparativa de resultados test de inclusión punto en sólido (tiempo medio de un test en milisegundos). 102

A.1. Average triangles per leaf 120
A.2. Volume enclosed by the BP-Octree at each level (B-rep volume=100 %). . . . 127
A.3. BP-Octree building times. 127
A.4. Influence of parameter k. 130

A.5. KBytes of BP-Octree files, and ratio $(.bpn + .bpl)/.ply$. 133
A.6. Number of planes at each level of the BP-Octree. 134
A.7. Percentage of total number of planes that are used at first time at each level. 135
A.8. Happy Buddha transmitted data. 136
A.9. Volume of the BP-Octree at each level(Brep model is 100 %). 137
A.10. *Point in solid* tests per second. 143

CAPÍTULO 1

Motivación y presentación de la memoria

En los últimos años el realismo conseguido en los gráficos por ordenador se acerca visualmente a la realidad hasta límites insospechados hace pocos lustros. Las industrias del cine y los videojuegos, así como los grupos de investigación de las universidades de todo el mundo, se han afanado en conseguir métodos de iluminación cada vez más cercanos al comportamiento físico de la luz, en simular las propiedades de los materiales más complejos, y desarrollar técnicas de digitalización que consiguen réplicas exactas de objetos reales con precisión microscópica.

Este aumento en la calidad de los gráficos obtenidos lleva aparejado un aumento en las prestaciones requeridas en cuanto a cómputo, siendo prácticamente imprescindible el uso de procesamiento paralelo para ejecutar los algoritmos de renderizado y el tratamiento de modelos cada vez más complejos en un tiempo razonable. Estas arquitecturas paralelas ya no son un privilegio de las grandes empresas, sino que cada usuario dispone de procesadores vectoriales en su tarjeta gráfica.

Podríamos decir que estamos llegando al límite de la perfección en cuanto a la percepción visual de los gráficos por ordenador, pero esta misma perfección lleva aparejados otros problemas que son los que en parte intentamos resolver en el trabajo que sustenta la presente memoria.

Los modelos tridimensionales se pueden representar de muy diversas formas, teniendo cada una de ellas sus puntos fuertes y débiles que las hacen más adecuadas a ciertas disciplinas. Sin embargo, las mismas arquitecturas gráficas anteriormente citadas están optimizadas para trabajar con una representación concreta, que se ha convertido en el estándar *de facto* de representación de modelos tridimensionales: las mallas poligonales, en concreto las mallas de triángulos. Las mallas son representaciones basadas en superficie (B-rep) que definen la frontera de un objeto con una colección de vértices y polígonos, habitualmente triángulos.

Para obtener el alto nivel de detalle geométrico que requieren las aplicaciones gráficas actuales, es necesario por tanto contar con un elevado número de primitivas geométricas, de triángulos, de forma que éstos se aproximen lo más exactamente posible a la superficie real

del objeto. Es por ello por lo que no es extraño encontrarse hoy en día con modelos virtuales de varios cientos de miles e incluso millones de triángulos.

Sobra decir que este elevado nivel de detalle supone un alto coste computacional, especialmente a la hora de visualizar los modelos y más aún si lo hacemos de forma realista. A mayor número de polígonos en el modelo, más operaciones serán necesarias para obtener el resultado de un trazado de rayos u otras técnicas de iluminación realista.

Pero no sólo en el aspecto visual se centra hoy en día la informática gráfica. Hay otros muchos aspectos que determinan el realismo de una escena, como pueden ser la interacción entre los objetos entre sí, mediante su comportamiento en el caso de colisiones entre ellos, o incluso la interacción humana con el mundo virtual que se representa mediante dispositivos hápticos que transmiten sensaciones táctiles al usuario.

Además, hoy en día se está dando un giro radical a la concepción de los sistemas informáticos, teniendo cada vez más presencia los sistemas distribuidos, y éste es un aspecto que la comunidad científica de informática gráfica está aún desarrollando. Estos grandes modelos geométricos ocupan varias decenas de megabytes que tardan un tiempo considerable en ser transmitidos a través de una red comercial, y por tanto la visualización de ellos dista mucho de ser interactiva.

Es en el marco de esta problemática donde se inicia el trabajo de investigación que ha dado lugar a la presente memoria, y en el cual se abordan las cuestiones relativas a la simplificación de grandes modelos poligonales de forma que se puedan realizar de forma eficiente la detección de colisiones entre objetos así como la visualización de estos grandes modelos a través de una red de comunicaciones de forma progresiva y/o adaptativa.

1.1. Objetivos

Partiendo de un análisis de las estructuras de datos y algoritmos existentes en la bibliografía, los objetivos básicos que se plantearon para este trabajo fueron:

1. Obtener un nuevo esquema de representación que permita gestionar grandes modelos geométricos con las siguientes restricciones:
   - Que sea capaz de manejar modelos de varios cientos de miles de polígonos.
   - Que permita una indexación espacial que acelere los tests de inclusión.
   - Que posibilite una visualización multirresolución del modelo.
   - Que el volumen definido por la representación obtenida, en cualquiera de sus niveles de detalle, nunca sea inferior al del modelo original y siga una secuencia de volumen decreciente conforme aumentamos el detalle.
   - Que el espacio en memoria y en disco requerido sea mínimo, para acelerar una transmisión del modelo a través de una red de comunicaciones.
2. Implementar tests de inclusión punto-en-sólido y otros algoritmos de detección de colisiones.
3. Implementar algoritmos de visualización del modelo, tanto de forma directa como adaptativa.

4. Implementar algoritmos de visualización basados en imágenes para mejorar la visualización del modelo.

El resultado es una estructura de datos que almacena una jerarquía de volúmenes envolventes que satisface los objetivos anteriormente citados, y que detallaremos en los siguientes capítulos de esta memoria.

1.2. Organización de la memoria

En la presente memoria hemos tratado de resumir de forma clara y concisa el trabajo realizado durante los últimos años y los resultados obtenidos a partir del mismo. Algunos de los trabajos que aquí se presentan ya han sido publicados o presentados en revistas y congresos nacionales e internacionales [MCT04, MCT05, MCT07, MCT08].

En el capítulo 2 se presentarán los conceptos básicos de manejo de grandes modelos geométricos así como un resumen de otras estructuras de datos existentes para la representación de sólidos, centrándonos en aquellas estructuras que sirven de base para este trabajo.

En el capítulo 3 se propone el nuevo esquema de representación, denominado BP-Octree (*Bounding-Planes Octree*). El nombre es debido a que se utiliza un octree como índice espacial, de forma que en cada nodo se almacena un conjunto de planos que, intersecando sus semiespacios interiores con el volumen del propio nodo forman un volumen que delimita de forma convexa a la parte del modelo contenida en el nodo.

En el capítulo 4 se estudia el espacio consumido por la estructura de datos y la forma en que la información se transmite a través de la red para su visualización remota de forma progresiva. Así mismo, se detalla el algoritmo de visualización de los nodos y se ofrece una solución para la mejora de la visualización de la representación en los niveles más altos del árbol mediante el uso de impostores dependientes del observador.

En el capítulo 5 se analiza el comportamiento del BP-Octree ante los tests de inclusión punto-en-sólido así como en el cálculo de distancias de un punto a la superficie del objeto, aplicándolo a la detección de colisiones en dispositivos hápticos.

Finalmente, en el capítulo 6 realizaremos un breve resumen de las aportaciones que se extraen del trabajo realizado y se presentan las líneas que quedan por elaborar y que serán los temas en los que se trabajará a partir de ahora.

CAPÍTULO 2

Introducción

En este primer capítulo realizaremos una revisión de diversos aspectos relacionados con la representación, gestión y organización de objetos desde el punto de vista espacial, que guardan relación con la estructura propuesta en el presente trabajo. Haremos un breve repaso por las herramientas y técnicas orientadas a la representación de objetos sólidos, centrándonos a continuación en los avances registrados en los últimos años para el manejo de grandes modelos geométricos. Estas técnicas hacen uso en la mayor parte de los casos de estructuras de organización espacial de la información geométrica, las cuales también detallaremos seguidamente. Finalmente, nos centraremos en aquellas estructuras espaciales que están específicamente orientadas a la aceleración de algoritmos de detección de colisiones, como son las jerarquías de volúmenes envolventes.

2.1. Modelado de sólidos.

El concepto de *modelado* se aplica en muchas áreas de la informática. Básicamente consiste en crear una representación virtual de algo real para realizar sobre dicho modelo virtual operaciones que permitan extrapolar los resultados al elemento real. Podemos entender el modelado de sólidos como un conjunto de herramientas y técnicas orientadas a la representación y manipulación de objetos sólidos. Un modelo sólido es la representación digital de un objeto real o imaginario [RR99], utilizando una determinada estructura de datos procesable por el ordenador. El resultado final es una aproximación digital detallada, completa y *no ambigua* de la geometría de un objeto o una composición de objetos (como puede ser un motor o cualquier otra pieza compleja). Entre las operaciones más habituales que se realizan sobre los sólidos destacan el procesamiento de la superficie del modelo y de su interior, la visualización, las operaciones entre distintos objetos, etc.

El modelado de sólidos es un campo que históricamente y aún hoy aglutina una gran actividad investigadora dentro de la Informática Gráfica [BEH79], [RV83]. Las dos definiciones clásicas más difundidas son las siguientes:

- El modelado de sólidos es *"el conjunto de teorías, técnicas y sistemas orientados a la representación completa en cuanto a la información de sólidos. Dicha representación debe permitir, al menos en principio, calcular automáticamente cualquier propiedad bien definida de cualquier sólido almacenado"* [RV83]

- Un modelo geométrico de un sólido es *"una representación matemática, no ambigua y completa, de la forma de un objeto físico tal que dicha representación pueda ser procesada por un ordenador"* [Mor85]

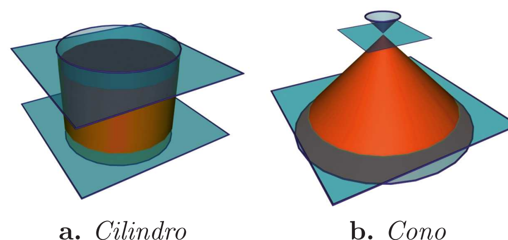
2.1.1. El proceso de modelado

El proceso de modelado es el resultado de una secuencia de abstracciones y aproximaciones: idealización, aproximación de la superficie y digitalización.

- **Idealización**. El primer paso es la abstracción del objeto físico real en un conjunto de puntos tridimensionales homogéneo y perfecto, ignorando en general sus estructuras internas e imperfecciones superficiales. Matemáticamente, un sólido es un subconjunto de $\mathbb{R}^3$ denominado *r-set* [Req80] que se caracteriza por ser:

    - *Limitado*. Un conjunto está limitado si tiene una extensión finita, esto es, si puede englobarse por una esfera de radio finito.
    - *Regular*. Todo el sólido debe estar compuesto por materia, sin tener caras, aristas o puntos aislados. Matemáticamente, un conjunto A es regular si $A = clausura(interior(A))$.
    - *Semi-algebráico*. El sólido es el resultado de combinar mediante operaciones de conjunto (unión, diferencia, intersección y complemento) un número finito de semiespacios, cada uno de ellos definido por una inecuación del tipo $(x, y, z)|f(x, y, z) \leq 0$ donde f es un polinomio. Por ejemplo, podemos ver en la figura 2.1 que un sólido cilíndrico finito es el resultado de intersecar tres semiespacios semialgebraicos: uno cilíndrico y los otros dos planos.

    Estas cualidades del *r-set* hacen que satisfaga las propiedades necesarias para que un modelo se considere un sólido:



a. *Cilindro* **b.** *Cono*

Figura 2.1: *Modelos matemáticos de un cono y un cilindro representados como la intersección de varios semiespacios.*

- *Rigidez*. El sólido tiene una forma fija e invariante que es independiente de su posición y orientación. Las transformaciones rígidas preservan distancias y ángulos entre los puntos de un conjunto en un espacio Euclídeo, que son fijos de por sí.
- *Finitud*. Un sólido debe tener una extensión finita.
- *Solidez*. El modelo de un sólido debe ser homogéneamente tridimensional, es decir, debe tener interior y no debe presentar caras o aristas aisladas, cuya existencia no es posible en un sólido real.
- *Cerrado bajo operaciones booleanas*. Operaciones booleanas aplicadas sobre sólidos deben generar otros sólidos.
- *Describible finitamente*. Los conjuntos de puntos usados para modelar sólidos deben ser almacenables en una cantidad finita de información.
- *Determinado por el contorno*. Un sólido debe poder modelarse sin ambigüedad por su frontera, que delimita el interior y el exterior del mismo. Sin esta propiedad no se pueden usar representaciones basadas en frontera, B-reps.

Los $r - sets$ no necesitan estar conectados. Por ejemplo, dos cubos disjuntos son un único $r - set$ si se considera su posición uno con respecto al otro rígida y se mueven juntos cuando se aplican transformaciones rígidas a ambos.

- **Aproximación de la superficie**. La segunda etapa es la aproximación de la frontera mediante un cierto número, relativamente pequeño, de caras, siendo cada cara una superficie del tipo concreto de superficies soportado por el sistema de modelado. Muchas de las variaciones entre distintas técnicas de modelado de sólidos se basan en las primitivas que utilizan para la representación de estas superficies. Estas primitivas pueden ser:

  - *Primitivas de caras planas*, como triángulos u otros polígonos más generales. Proporcionan una pobre aproximación a la forma real del objeto, a menos que se use un gran número de primitivas que detallen con gran precisión las geometrías con gran curvatura. Aunque los algoritmos que tratan con triángulos son sencillos y relativamente robustos, esta representación presenta problemas de eficiencia y gestión cuando se trata de manejar modelos complejos a un elevado nivel de detalle. Sin embargo, el hardware gráfico está altamente orientado al manejo de esta representación de sólidos, por lo que se ha convertido en cierto modo en un estándar *de facto*.
  - *Superficies paramétricas*, como las *Non-Uniform Rational B-spline Surface* (NURBS), que pueden ajustarse de una manera mucho más exacta a la superficie que miles de triángulos. Sin embargo, detectar y calcular intersecciones usando dichas superficies libres de forma supone elaborar unas técnicas y algoritmos matemáticos que pueden ser bastante más lentos y complejos que los existentes para geometrías triangulares.
  - *Superficies cuádricas naturales* (planos, cilindros, conos, esferas), que ofrecen un compromiso entre la representación matemática exacta para la mayoría de los objetos producidos en la industria, y un sencillo cálculo de intersecciones. Sin embargo, no permiten el modelado libre de forma ni otros objetos con requerimientos estéticos.

- *Superficies implícitas*, que usan una función tridimensional en la que los puntos que pertenecen a la superficie toman valor cero. Una de sus principales propiedades es que aportan una información precisa sobre el interior y exterior del objeto, proporcionando una rápida clasificación del espacio interno del objeto.

- **Digitalización**. Una vez elegido el método de aproximación de la superficie, se debe realizar la digitalización de los parámetros numéricos que definen la forma y posición de las superficies. Para ello se usan comunmente las representaciones de coma flotante, con los errores de precisión y redondeo inherentes en valores cercanos a cero o tendentes a infinito, o representación en aritmética entera, haciendo uso de escalado y desplazamiento. En la práctica, se usan números de coma flotante ya que no se requiere un conocimiento a priori del intervalo, y se pueden usar de forma homogénea para representar y calcular todos los parámetros del modelo de forma inmediata.

2.2. Esquemas de representación.

El concepto de representación está vinculado al concepto de abstracción mencionado con anterioridad, y puede entenderse como una relación entre sólidos abstractos y descripciones de éstos.

Una descripción sintáctica correcta de un sólido es una colección finita de símbolos (de un alfabeto finito) que describen un sólido siguiendo una serie de reglas sintácticas de un determinado esquema de representación [Män87].

Formalmente, un esquema de representación puede definirse como una relación $s : M \rightarrow R$, siendo M el espacio de modelos matemáticos (compuesto por sólidos abstractos) y R el conjunto de expresiones sintáctica y semánticamente correctas que definen el modelo en términos computacionales [Män87]. De todos los esquemas de representación posibles, los más interesantes son aquellos que son únicos y no ambiguos.

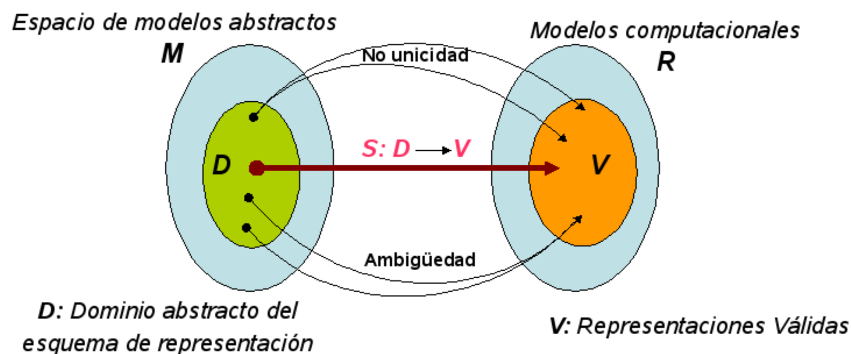


Figura 2.2: *Relación entre sólidos matemáticos y sus representaciones computacionales.*

En cualquier caso, todo esquema de representación elegido para el modelado de sólidos debe cumplir las siguientes condiciones [Req80], representadas gráficamente en la figura 2.2:

- **Dominio**. El dominio D es el conjunto de sólidos $D \subseteq M$ que se puede modelar con el esquema de representación. Cuanto mayor sea el conjunto de sólidos representables, mejor será considerado el esquema.

- **Validez**. Un método de representación es válido si el conjunto imagen coincide con el conjunto de representaciones. Una representación es *no válida* si no corresponde a ningún sólido real. El conjunto de todas las representaciones válidas se denomina el *rango del sistema* V. El propio esquema de representación debe asegurar de por sí la validez de todas las representaciones que se pueden crear, o habilitar métodos automáticos para comprobar la validez de las representaciones.

- **No ambigüedad**. Un esquema de representación es no ambiguo si cada representación se corresponde con un único objeto real.

- **Unicidad**. Un objeto del espacio de modelos debe tener una única representación posible utilizando el esquema elegido. Esta propiedad permite realizar comprobaciones de igualdad entre objetos.

Además de estas propiedades, es conveniente que el esquema de representación cumpla con una serie de propiedades adicionales, importantes a nivel práctico para su implementación:

- **Concisión**. La representación debe ser lo más compacta posible, evitando la redundancia en los datos. En ciertos casos la redundancia de datos puede servir para acelerar diversas operaciones sobre los modelos.

- **Facilidad de edición**. Habitualmente los datos son introducidos parcial o totalmente por un ser humano, por lo que es necesario un subsistema de entrada que permita la edición simplificada del modelo.

- **Eficacia**. Cada esquema de representación tiene un campo de aplicación preferente, por lo que debe permitir la realización de operaciones habituales (p.ej. visualización o cálculo de propiedades) en dicho ámbito de forma eficiente y eficaz.

Teniendo en cuenta todas estas cualidades, Requicha [Req80] distingue entre:

1. Instanciación de primitivas
2. Enumeración espacial
3. Descomposición en celdas
4. Geometría Constructiva de Sólidos (CSG)
5. Barrido y traslación
6. Representación basada en fronteras (B-rep)

Esta clasificación puede simplificarse agrupando en función de los dos modelos matemáticos de sólidos establecidos [Sha02]:

- **Representaciones implícitas y constructivas:** Establecen las reglas para **comprobar** qué puntos pertenecen al conjunto y cuáles no. Tales representaciones son naturalmente soportadas por el modelo de sólidos de conjuntos de puntos continuos. En este grupo se incluyen la instanciación de primitivas, la Geometría Constructiva de Sólidos y las operaciones de barrido y traslación.

  Un método muy general para definir un conjunto de puntos X es especificar un predicado A que pueda ser evaluado para cualquier punto p del espacio $X = \{p|A(p) = true\}$.

  Dicho de otro modo, X es definido *implícitamente* de forma que son todos los puntos que satisfacen la condición especificada por el predicado A. El predicado más simple es evaluar el signo de alguna función real $f(p)$, por ejemplo, si $f = ax + by + cz + d$, las condiciones $f(p) = 0$,$f(p) \geq 0$ y $f(p) < 0$ evalúan un plano, un semispacio lineal cerrado y un semiespacio lineal abierto respectivamente.

- **Representaciones enumerativas y combinatoriales:** especifican las reglas para **generar** puntos del conjunto (y no otros puntos). En este grupo se incluyen los esquemas de enumeración espacial y de descomposición de celdas, así como la representación basada en fronteras (B-rep).

Las representaciones enumerativas y combinatoriales se acercan más al concepto algebraico y topológico de sólido, mientras que las implícitas y constructivas están relacionadas directamente con el concepto de conjunto continuo de puntos. Afortunadamente, ambas teorías son consistentes y pueden usarse ambos modelos matemáticos indistintamente, apoyando en las propiedades de continuidad o combinatorial cuando sea necesario.

A continuación se presentan los esquemas de representación más relevantes por su relación con este trabajo.

2.2.1. Representación basada en fronteras (B-Rep)

Las representaciones basadas en frontera (*boundary representations, B-rep*, en inglés) definen los sólidos indirectamente, mediante una representación de la superficie que los delimita

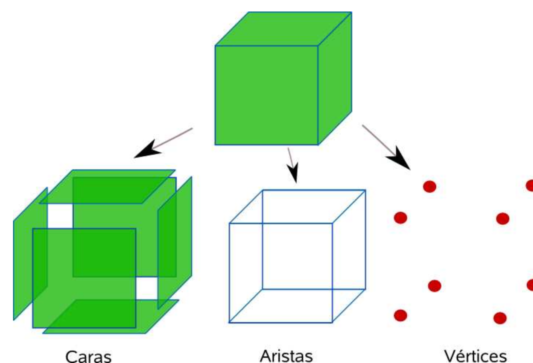


Figura 2.3: *Elementos B-Rep para definir un cubo.*

[BEH79][Req80][Män87][Sha02]. Esta superficie se divide en un número finito de elementos denominados *caras* si son planos, o *parches* si son superficies curvas. Los modelos *B-rep* delimitados por caras planas se denominan poliedros.

Los elementos básicos de una representación *B-rep*, mostrados en la figura 2.3, son las *caras*, los *vértices* –que son los puntos que definen las caras– y las *aristas*, que unen los vértices de las caras delimitando el exterior y el interior de las mismas.

Numerosos autores restringen el uso de *B-rep* a superfices cerradas, orientadas y 2-variedad, es decir, que cualquier punto de la superficie tiene conectados a él un subconjunto de puntos que podrían considerarse topológicamente análogos a un disco en el plano [FvFH90]. Dicho de otra forma: en un sólido 2-variedad, todas las aristas son compartidas por sólo dos caras. Una profunda disertación sobre las propiedades de los sólidos 2-variedad y sus propiedades se puede encontrar en [Män87].

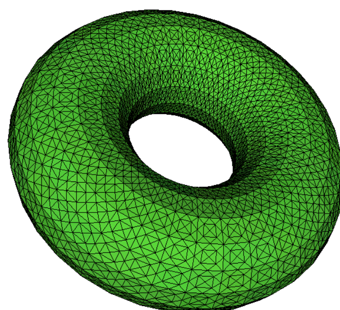


Figura 2.4: *Un toroide no es un poliedro simple, al no ser homeomorfo a una esfera.*

Un *poliedro* es un sólido limitado por caras planas –polígonos– donde cada arista es compartida por un número par de caras. Un *poliedro simple* es aquel que es homeomorfo a una esfera en $\mathbb{R}^3$. Esto implica que no puede tener agujeros como los de un toroide (figura 2.4). Toda representación *B-rep* de un poliedro simple cumple lo que se denomina la *fórmula de Euler*, que expresa una relación invariante entre el número de vértices, aristas y caras de un poliedro simple:

$$V - E + F = 2 \tag{2.1}$$

donde V es el número de vértices, E el de aristas y F el número de caras. Esta fórmula se aplica también a poliedros simples de caras no planas. Sin embargo, ésto no garantiza que sea un sólido lo que estamos representando. Además, se deben cumplir las siguientes restricciones: cada arista debe conectar dos vértices y sólo puede estar compartida por dos caras, al menos tres aristas deben coincidir en cada vértice y las caras no deben intersecar entre sí.

Para poliedros no simples, con agujeros, se puede formular una generalización de la fórmula de Euler como sigue:

$$V - E + F - H = 2(C - G) \tag{2.2}$$

donde H es el número de agujeros en las caras que no traspasan el sólido, C el número

de componentes separadas que forman el sólido y G el número de agujeros que atraviesan el sólido.

De entre las alternativas existentes para representar la geometría y la topología de un modelo de fronteras, destacamos las siguientes [Män87]:

- *Modelos de fronteras basados en polígonos.* El sólido S se representa como un conjunto de n caras, y para cada cara F se almacenan las coordenadas de los j vértices que la forman.

$$S = F_1, F_2, F_3, ..., F_n$$
$$F_i = (x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_j, y_j, z_j)$$

- *Modelos de fronteras basados en vértices.* El modelo basado en polígonos hace que para cada vértice v, se almacenen sus coordenadas tantas veces como el vértice aparezca en la descripción del modelo. Para eliminar esa redundancia, se almacena una lista V con las coordenadas de los vértices y cada cara F_i se describe con los j índices de sus vértices. Pese a esta mejora, seguimos sin tener información topológica del modelo.

$$S = F_1, F_2, F_3, ..., F_n$$
$$F_i = v_1, v_2, ..., v_j$$
$$V = (x_1, y_1, z_1), (x_2, y_2, z_2), ...(x_m, y_m, z_m)$$

- *Modelos de fronteras basados en aristas.* Estos modelos representan la superficie como una secuencia cerrada de aristas, de forma que los vértices de las caras se representan sólo por las aristas. La estructura de datos define una orientación para las aristas, y esta orientación es consistente de forma que una cara está definida por una secuencia cerrada de aristas en sentido horario según se ven desde el exterior de la cara. Obviamente, una arista es usada una vez en sentido horario y otra en sentido antihorario, para definir la cara opuesta. Encontramos diversas estructuras de datos para almacenar esta información:

  - *Aristas aladas* [Bau72]. Sólo sirve para sólidos 2-variedad, y almacena la siguiente información:

$$Arista = (V_{inicio}, V_{final}, A_{sigh}, A_{sigah})$$
$$Vertice = (x, y, z)$$
$$Cara = (A_{inicio}, Sentido)$$

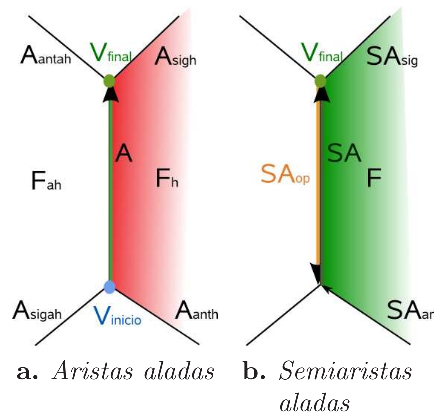
    donde para cada arista A, V_{inicio} y V_{final} son los vértices extremos de la arista, A_{sigh} es la siguiente arista en el sentido horario, A_{sigah} es la siguiente arista en el sentido antihorario (y por tanto está en la cara opuesta a la definida por el sentido horario de la arista). Los vértices se almacenan con sus coordenadas geométricas, y para cada cara sólo es necesario indicar una de sus aristas A_{inicio} y el $Sentido$ en que ésta se ha de recorrer para definir la cara.

    Otras extensiones a esta estructura añaden a la arista identificadores F_h y F_{ah} para identificar las dos caras compartidas por la arista, y las aristas anteriores en ambos sentidos A_{anth} y A_{antah}. Es el caso de la figura 2.5a.

- *Semi-aristas aladas.* Basada en la estructura de aristas aladas, elimina el concepto de sentido, dividiendo cada arista en dos semi-aristas, cada una con su sentido propio. Esta estructura de datos permite realizar de forma eficiente consultas habituales en el tratamiento de modelos poligonales, como las caras y aristas que comparten un vértice, las aristas que rodean una cara, etc. Para cada semiarista SA se almacena exclusivamente la siguiente información:

$$SA = (V_{final}, SA_{ant}, SA_{sig}, F, SA_{op})$$

donde V_{final} es el vértice en el que termina la semiarista, SA_{ant} y SA_{sig} son las semiaristas anterior y siguiente (SA_{ant} no es estrictamente necesaria, pero es conveniente para aumentar la eficiencia), F es la cara que contiene la semiarista y SA_{op} es la semiarista opuesta, que tiene como V_{final} el vértice origen de SA. Se puede apreciar esta estructura en la figura 2.5b



a. *Aristas aladas* **b.** *Semiaristas aladas*

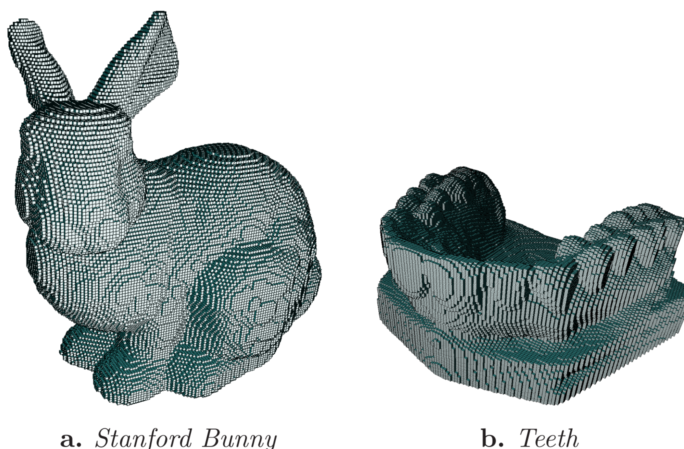
Figura 2.5: *Representaciones de frontera basadas en aristas. Información almacenada para una única arista/semiarista.*

Las representaciones basadas en fronteras son las más utilizadas por la comunidad gráfica hoy en día, hasta tal punto que el hardware gráfico está específicamente diseñado y optimizado para un tipo muy concreto: los modelos basados en vértices definidos por polígonos de tres vértices, las mallas triangulares. Lo habitual es encontrar formatos de archivo que almacenan los vértices y caras triangulares con referencias a estos vértices, y, en función del procesamiento a realizar con los modelos, utilizar en memoria principal una estructura basada en aristas o continuar con la lista de triángulos.

2.2.2. Enumeraciones espaciales

Las enumeraciones espaciales - o agrupamientos según [Sha02]- son la forma más simple de representar un conjunto de puntos. Ejemplos de enumeraciones espaciales son: conjuntos de cubos tridimensionales (denominados voxels), haces de rayos (columnas rectangulares de tamaño finito) [Men93], uniones de esferas, secuencias de poliedros tridimensionales, etc... El principio común es que todas las enumeraciones son agrupaciones de celdas cerradas que

representan pequeñas porciones del espacio. Estas celdas son lo suficientemente simples como para admitir representaciones implícitas y paramétricas, y como son todas iguales en dimensión y tipo, los algoritmos para su manejo son sencillos. Por ejemplo, el test de inclusión para un punto se reduce a un test de inclusión que se repite para cada celda. Dependiendo del tipo geométrico concreto, las enumeraciones se pueden agrupar usando estructuras de datos computacionalmente más compactas y eficientes (árboles, grafos jerárquicos, etc.).



a. *Stanford Bunny* **b.** *Teeth*

Figura 2.6: *Representaciones mediante enumeración espacial.*

Este esquema de representación conlleva en la mayoría de los casos una pérdida de información, ya que rara vez el sólido real se ajusta a las celdas usadas para su representación, como se aprecia en la figura 2.6. Por ejemplo, el uso de voxels sólo daría lugar a una representación exacta si el sólido tuviese todas sus caras perfectamente alineadas a los planos definidos por los ejes de coordenadas y cada una de las dimensiones del sólido fuese divisible por el tamaño de las celdas.

Para resolver en la medida de lo posible esta falta de precisión, se utilizan esquemas de representación que en lugar de utilizar subdivisiones regulares del espacio ocupado por el sólido, realizan una subdivisión adaptativa del mismo, aumentando el nivel de detalle –disminuyendo el volumen de la celda– allí donde sea estrictamente necesario.

2.2.2.1. Octree

El Octree [Mea82],[JT80] puede ser considerada la estructura clásica de subdivisión espacial. Podemos verlo como una compactación de los *voxels* de una enumeración por coherencia espacial de forma adaptativa.

Se basa en la división recursiva de la caja envolvente del sólido a representar en ocho octantes que cubren totalmente el volumen de su ancestro, organizando dicha estructura en un árbol octal [1] (figura 2.7). Si un octante está totalmente ocupado por el sólido se etiqueta como **negro**, mientras que si está totalmente fuera se marca como **blanco**. En ambos casos, los nodos negros y blancos del árbol son nodos hoja. Si por el contrario, el *voxel* está sólo

[1] En adelante se utilizarán los términos *octante*,*voxel* o *nodo* de forma indistinta para identificar cada una de estas divisiones.

parcialmente ocupado por el sólido, se marca como **gris**, en cuyo caso es necesario volver a dividir en ocho hijos que corresponden con la subdivisión regular del voxel, y se prosigue con el algoritmo recursivo en cada uno de estos ocho nuevos nodos.

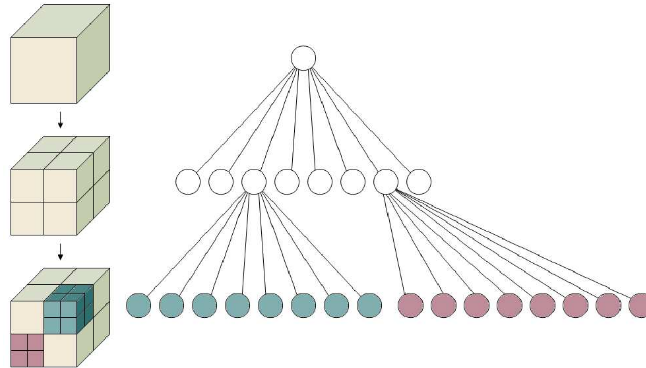


Figura 2.7: *Octree. Subdivisión espacial a la izquierda. A la derecha, árbol representando los nodos existentes.*

En caso de no poder alcanzar una representación exacta del sólido (lo que tan sólo puede ocurrir si todas las caras del sólido son paralelas a los planos que forman los ejes de coordenadas) la condición de parada para la recursión se produce al alcanzar una determinada profundidad en el árbol. En el caso de que se alcance este nivel máximo, será necesario un método para decidir la clasificación de los nodos hoja parcialmente ocupados, obteniendo en cualquier caso una aproximación del sólido, más o menos exacta en función de la resolución alcanzada.

Representación. El árbol octal que representa al sólido puede ser almacenado de muy diversas formas, que han ido apareciendo en la bibliografía con los objetivos comunes de aumentar la eficiencia y el rendimiento en memoria del *octree*:

- *Como un árbol explícito* [Män87][SW88]. Cada nodo almacena su tipo y ocho punteros a sus nodos descendientes. El nodo raíz del árbol almacena además las coordenadas de la caja envolvente. Se podrían usar una estructura como la mostrada en el código 1.

```
struct octree {
      double xmin,ymin,zmin;
      double xmax,ymax,zmax; // Caja envolvente
      struct octree *root;
};

struct octreenode {
      char tipo; // BLANCO, NEGRO o GRIS
      struct octreenode *hijo[8];
};
```

Algoritmo 1: *Estructura de datos para un árbol octal explícito.*

- *Codificación Lineal*[Gar82]. Se almacena una lista ordenada del camino a seguir para cada nodo negro. Hay un dígito octal para cada nivel, de forma que conforme se va leyendo el número de izquierda a derecha el código nos muestra qué hijo tomar para seguir la ruta.
- *Codificación Lineal en Preorden*[Oli84]. Se almacena una lista ordenada de nodos generada al recorrer el árbol en preorden. Para cada nodo se indica su tipo, de forma que tras un nodo gris, aparecerá la codificación de sus ocho hijos.

En general, el número de nodos necesarios para representar un sólido con un octree es proporcional al área de la superficie de dicho objeto [Mea82], por tanto, se puede asegurar que si bien los octrees no necesitan tanto espacio como las enumeraciones exhaustivas, el alcanzar un elevado nivel de detalle supone un coste extra de almacenamiento.

Propiedades. La propiedad esencial de los octrees es que recogen la información geométrica del objeto de una forma espacialmente ordenada, lo que permite la generación de algoritmos muy simples para su visualización [DT81].

De forma análoga, algoritmos para el análisis y cálculo de propiedades del objeto, tales como su volumen, momento de inercia o centro de masas, puede ser también calculado con sencillos recorridos por los nodos del árbol.

Además, las operaciones booleanas sobre octrees también son fácilmente implementables mediante algoritmos recursivos que recorren ambos operandos de forma síncrona. Dichos algoritmos se basan en comparar los *voxels* homólogos de los dos objetos operandos, aplicando nuevamente de forma recursiva el algoritmo si el resultado de la operación booleana es un nodo gris.

En el lado opuesto, el octree tiene serias limitaciones para ciertos algoritmos, la mayoría de ellas derivadas del carácter discreto de la representación. Por un lado, la visualización del sólido, aún al más alto nivel de detalle, produce un efecto de *aliasing* (ver figura 2.8). Dicho efecto puede subsanarse en cierto modo mediante la consulta a los nodos vecinos, pero es precisamente este algoritmo de acceso a nodos vecinos una de las carencias mayores de esta estructura, puesto que, salvo que se usen punteros auxiliares –con el consecuente aumento de espacio–, es necesario alcanzar la raíz del árbol y volver a descender hasta el nodo deseado.

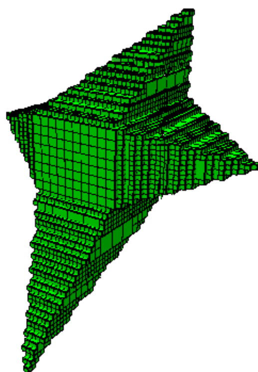


Figura 2.8: *Aliasing debido al carácter discreto del octree.*

2.2.2.2. Extensiones al Octree. Polytrees, Extended Octrees, SP-Octrees.

Para subsanar en la medida de lo posible la inexactitud inherente al octree, se han propuesto diversas extensiones al mismo de forma que se pueda representar de forma exacta el objeto sólido. Varias de ellas hace uso de información de la frontera, por lo que pueden ser considerados una aproximación híbrida. En [Sam90] se agrupan bajo el nombre común de *Polygonal Map Octrees* (PM-Octrees).

Polytrees [CCV85]. Incorpora en los nodos terminales información geométrica de la superficie del objeto, permitiendo una representación del mismo. Esta información se almacena en unos nuevos tipos de nodo, que son: nodos *cara*, que son aquellos atravesados por una única cara poligonal del sólido; nodos *arista*, que contienen dos caras adyacentes y parte de su arista común; y nodos *vértice* que contienen un vértice del poliedro y parte de las caras y aristas que convergen en él. La información de la geometría del objeto se calcula y almacena explícitamente en cada nodo. En [DK89] se presenta una extensión, denominada *polytrees integrados*, con tres nuevos tipos de nodos, *arista',vértice'* y *vértice"*, que intentan resolver casos especiales que conllevan a una recursión infinita en la construcción del árbol.

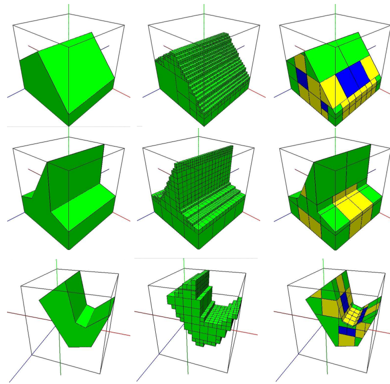


Figura 2.9: *Representación de un sólido (a la izquierda), mediane un octree (columna central) y un Extended Octree (derecha) [Can04]*

.

Extended Octrees [ABJN85][BN90]. Surgieron simultáneamente a los polytrees, y añaden también tres tipos de nodos al esquema clásico del octree: *cara, vértice y arista*, con idénticas características. Sin embargo, la información es calculada y almacenada de forma muy distinta. En los Extended Octrees se almacena en cada nodo la configuración y un conjunto de referencias a una única lista de semiespacios planos, eliminando las redundancias existentes en los polytrees. Por configuración se entiende la información necesaria para clasificar un punto con respecto al objeto [BN85]. La recursión infinita que también se presenta en los polytrees, que ocurre cuando un nodo es atravesado por un conjunto de caras que convergen en el exterior de nodo, se resuelve añadiendo lo que se denominan *nodos grises terminales*, que almacenan la configuración del nodo vértice en el que converge dicho conjunto de caras. En la figura 2.9 se muestran sólidos respresentados con octrees y con Extended Octrees, donde se observa que con éstos últimos son necesarios menos niveles y se consigue una representación exacta.

SP-Octrees [CTV03][Can04] *(Space Partition Octrees)*. Esta extensión de los octrees se basa en el uso de semiespacios planos para delimitar el interior y el exterior del sólido, tanto en nodos terminales como en nodos grises intermedios. Cuando un nodo está completamente dentro o fuera del sólido, se clasifica como *negro* o *blanco*. A estos dos tipos básicos, el SP-Octree añade los nodos *convexos* y *cóncavos*, cuando la intersección del sólido con un nodo del árbol sea un volumen convexo o cóncavo respectivamente. En estos casos, se almacena en el nodo una referencia a los planos que, intersecados con el volumen del nodo, crean el volumen convexo o cóncavo (éste último mediante sustracción de una convexidad al volumen completo). Se incluye también, al igual que en los Extended Octrees el nodo *vértice* para evitar recursiones infinitas, y el nodo *gris*, además de indicar una recursión necesaria por contener concavidades y convexidades en la geometría abarcada, contiene información de los planos que forman una envolvente convexa de la parte del sólido contenida en el nodo. En la figura 2.10 podemos ver los distintos tipos de nodo, así como un sólido representado con esta estructura en la figura 2.11. Esta estructura permite la realización de forma eficiente de tests de inclusión, así como una trasmisión progresiva del modelo poliédrico, ya que no almacena la información sólo en los nodos hoja, sino que anticipa la aparición de los planos a los niveles intermedios. Además, supone una reducción en el espacio necesario para el almacenamiento de los modelos.

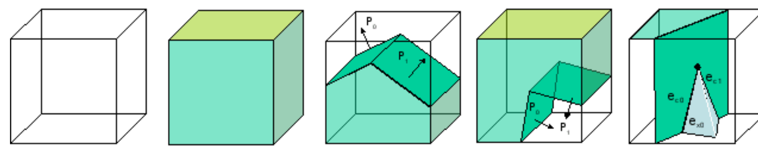


Figura 2.10: *Nodos blanco, negro, convexo, cóncavo y vértice del SP-Octree.*

El trabajo que se presenta en esta memoria se inspira en el SP-Octree para alcanzar los objetivos planteados en su inicio.

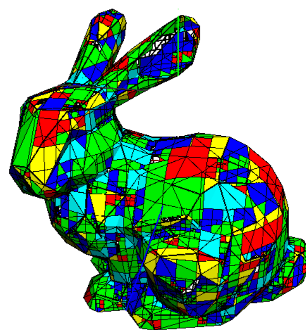


Figura 2.11: *Stanford Bunny representado con SP-Octree. Cada color indica un tipo de nodo.*

2.2.2.3. BSP Tree

Tal y como se ha visto, los octrees subdividen el espacio de forma ortogonal a los ejes de coordenadas, lo cual es un serio contratiempo a la hora de representar fielmente el sólido. Los *Binary Space Partition Trees* (BSP Trees), surgieron como una estructura para determinar la visibilidad en escenas tridimensionales, pero rápidamente fue aprovechada para representar poliedros arbitrarios [TN87].

El BSP-Tree es un árbol binario que divide recursivamente el espacio 3D en pares de semiespacios con respecto a un plano arbitrario. Estos dos semiespacios se suelen denominar como semiespacio *positivo* y *negativo*, estando el semiespacio positivo en la parte del plano hacia la que apunta su normal. Si estos planos coinciden con las caras de la geometría del sólido, estamos hablando de BSP-Trees *autoparticionados*.

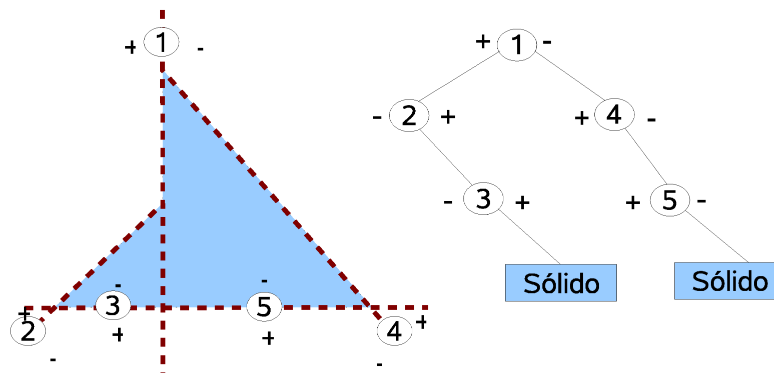


Figura 2.12: *BSP Tree sólido en 2D*

Los BSP Trees se pueden clasificar de diversas maneras: según se orientan los planos divisores, si se almacena la geometría en los nodos o en las hojas, o si se usan para división espacial o como representación volumétrica. En general, se suelen tratar tres tipos de BSP-Trees, según almacenen sus caras en los nodos, sólo en los nodos hoja o en ningún sitio:

- *BSP Trees con datos en los nodos.* Son BSP Trees autoparticionados y en cada nodo, además del plano divisor, se almacena la cara de la cual se extrae dicho plano, así como todas las caras coplanares. El resto de las caras se reparten adecuadamente entre los dos nodos hijos para construir recursivamente los subárboles del nodo. Si una cara es cortada por el plano divisor, se divide en dos y se reparten apropiadamente las nuevas caras resultantes

- *BSP Trees con datos en las hojas.* Son BSP Trees en los cuales la información geométrica se almacena únicamente en los nodos hoja, de forma que los nodos internos tan sólo contienen al plano divisor y las referencias a los dos subárboles.

- *BSP Trees sólidos.* Se realizan particiones del volumen del objeto en un conjunto de poliedros convexos, que se representan como el volumen de la intersección de un cierto número de semiespacios. En la figura 2.12 se puede observar cómo los nodos hoja tan sólo indican si el área (o volumen en 3D) que representan se encuentra en el semiespacio

exterior o en el interior. Si es en el interior, el nodo hoja se denomina *nodo sólido*. En ninguno de los nodos se almacena la geometría del poliedro.

Otros autores han propuesto estructuras de datos para conservar la información topológica del objeto geométrico inicial, como los *Topological BSP-Trees* [CN96].

Los principales problemas que presentan los BSP-Trees surgen por un lado de la excesiva dependencia en la elección del plano divisor, puesto que es una estructura que no cumple la propiedad de *unicidad* y por tando puede dar lugar a dos representaciones muy distintas, incluso con árboles totalmente desbalanceados, de un mismo sólido.

CAPÍTULO 3

BP-Octree

Como se ha descrito en el capítulo anterior, todo esquema de representación de un sólido ha de cumplir una serie de propiedades deseables:

- *Dominio*. El dominio D es el conjunto de sólidos $D \subseteq M$ que se puede modelar con el esquema de representación. En concreto, se debe intentar que D abarque al mayor número posible de sólidos.
- *Validez*. Un método de representación es válido si el conjunto imagen coincide con el conjunto de representaciones, por lo que se debe conseguir un esquema de representación que imposibilite representar sólidos no reales.
- *No ambigüedad*. Esta propiedad requiere que cada representación se corresponda con un único objeto real.
- *Unicidad*. Se requiere del esquema de representación que para cada sólido real exista una única forma de representarlo. Normalmente, los esquemas de representación existentes añaden restricciones para el cumplimiento de esta propiedad.

En el presente capítulo presentamos una estructura de datos, denominada **Bounding-Planes Octree** [MCT08], que intenta cumplir los objetivos que se marcaron en al inicio del trabajo cuya memoria se presenta en este volumen. Recordemos que el primero era obtener un nuevo esquema de representación que permita simplificar grandes modelos geométricos con las siguientes restricciones:

- Que sea capaz de manejar modelos de varios cientos de miles de polígonos.
- Que permita una indexación espacial que acelere los tests de inclusión.
- Que posibilite una visualización multirresolución del modelo.
- Que el volumen definido por la representación obtenida, en cualquiera de sus niveles de detalle, nunca sea inferior al del modelo original y siga una secuencia de volumen decreciente conforme aumentamos el detalle.

- Que el espacio en memoria y en disco requerido sea mínimo, para acelerar una transmisión del modelo a través de una red de comunicaciones.

Esta estructura de datos se basa en una descomposición espacial basada en un octree, pero almacenando información de la frontera B-Rep del modelo en sus nodos terminales y construyendo en los nodos intermedios envolventes convexas. En cierto modo, esta estructura toma de los SP-Octrees [Can04] el uso de los planos originales del modelo para construir esta envolvente.

Al diseñar la estructura de datos y los algoritmos para su construcción, se han debido tener en cuenta diversos aspectos para obtener una solución completa y correcta:

- *Garantizar que la totalidad de la geometría contenida en el nodo es usada al calcular su envolvente.* Como el octree es un entorno discreto, tenemos que asignar primitivas contínuas (polígonos) a voxels, y hacerlo sucientemente rápido. La solución obvia sería recortar todos los polígonos para que encajen exactamente en nuestros voxeles tridimensionales, creando nuevos polígonos, pero esto nos llevaría a un tiempo de cálculo y especialmente un incremento en el espacio de memoria necesario prohibitivos. En su lugar, usamos un algoritmo 3DDDA para recorrer cada polígono y determinar qué voxeles atraviesa.

- *Asegurar que la envolvente a nivel n está completamente contenida en la envolvente a nivel $n-1$.* Es importante mantener la coherencia entre niveles, ya que no tiene sentido que estemos en un nivel determinado del BP-Octree y al obtener un mayor detalle, resulte que la envolvente se ajusta peor al modelo -cuando debería ocurrir lo contrario. Para satisfacer este criterio, en los nodos hoja se calculan las envolventes usando la geometría real del modelo, y en los nodos internos, consideramos la geometría generada por la intersección de los planos envolventes de sus hijos.

- *Evitar huecos entre nodos adyacentes.* Al usar envolventes aproximadas, es casi imposible que dos nodos adyacentes compartan planos en su cara común, por lo que al visualizarlos, se vería un agujero en la unión. Hemos considerado conveniente añadir una serie de planos ficticios para taparlos, que no son más que las caras de cada nodo que son visibles en parte

- *Tratamiento de concavidades.* Cuando la geometría no es convexa, es bastante complicado encontrar una cara cuyo plano englobe a toda la geometría. Entonces, permitimos que los planos se "despeguen"de su cara origen con un desplazamiento tal que dejen en su semiespacio interior a todos los vértices de la geometría en cuestión, convirtiéndose en un plano envolvente por desplazamiento.

- *Las zonas convexas consiguen menor simplificación que las irregulares.* En el caso de una esfera, todas sus caras (planos) satisfacen el criterio de que son envolventes de la geometría, con lo que el 100 % de las caras formarían parte del nodo raíz. Para evitar dicha situación, limitamos el número de planos en cada nivel, y éstos son elegidos siguiendo un criterio de distribución estadística.

- *Determinación del interior y el exterior del sólido.* La creacion de los nodos negros y blancos se genera a partir de la información contenida en los nodos hoja, que cubren completamente la superficie del sólido. Para una correcta creación de estos nodos, como

es obvio, es necesario que el sólido sea cerrado y 2-variedad. En cualquier otro caso, el BP-Octree sólo podrá ser usado como representación de superficie.

En el presente capítulo analizaremos la estructura de datos propuesta, así como los algoritmos diseñados para su construccion. En los capitulos sucesivos analizaremos las prestaciones del BP-Octree para operaciones como la detección de colisiones o la visualización progresiva.

3.1. Fundamentos

La idea que subyace en esta estructura es realizar una aproximación híbrida entre los octrees y los BSP-Trees. Utilizando un octree como índice espacial, tendremos cuatro tipos de nodos:

- **blanco**, si el nodo está totalmente fuera del sólido,
- **negro**, si el sólido está totalmente dentro del nodo,
- **gris**, si el sólido ocupa parcialmente el volumen del nodo, y
- **hoja**, si es un nodo terminal del árbol atravesado por la frontera del sólido.

Tanto en los nodos grises como en los nodos hoja, almacenaremos un conjunto de planos cuyos semiespacios interiores, intersecados entre sí y con el volumen del nodo, forman un volumen convexo que engloba a la parte del sólido contenido en dicho nodo, como se aprecia en la figura 3.1. Además, en los nodos hoja, se almacenan referencias a los polígonos contenidos total o parcialmente por el nodo, de una manera similar a como lo hacian los polytrees [CCV85].

La existencia de los nodos negros y blancos es irrelevante si esta estructura se usa tan sólo para la visualización de los modelos, puesto que con los nodos grises y hoja conseguimos una representación de fronteras, aproximada y convexa celda a celda, del sólido en cuestión.

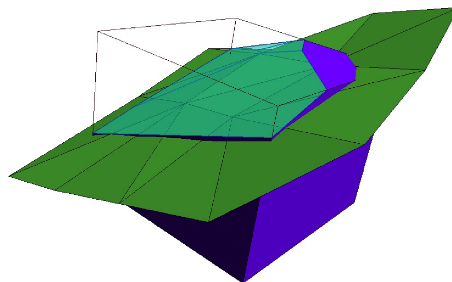


Figura 3.1: *Detalle de un nodo hoja. En verde la geometría real. Cyan la envolvente convexa almacenada. En morado, los planos ficticios del nodo.*

La particularidad de la estructura propuesta es que estos planos que forman las convexidades son planos que ya existen en el modelo B-Rep, ya que se extraen de las caras poligonales originales del modelo, al igual que hacen los SP-Octrees [CTV03].

```
typedef long int Index;

typedef struct {
        Index planeIndex;
        double d;
} BPlane;

typedef long int Octcode;
class BPNode {
         vector<BPlane> bounding;
         Octcode oct;
}

class BPLeafNode:public BPNode {
         vector<Index> geometry;
};
```

Algoritmo 2: *Estructuras de datos usadas para almacenar el BP-Octree*

Básicamente, la estructura de datos que maneja el BP-Octree se describe en el código 2, y gráficamente en la figura 3.2. Cada nodo tiene asignado un *octcode*, basado en la codificación Morton [Mor66] que sirve para identificar únicamente a un nodo en toda la jerarquía, y que nos permite movernos por el árbol con tan sólo leer los dígitos en octal, al estilo de lo presentado por Gargantini [Gar82]. Además de este *octcode*, el nodo almacena una serie de índices de planos, que pueden ir acompañados de un desplazamiento *d*, que son los que forman la envolvente convexa en dicho nodo. Adicionalmente, en los nodos hoja, se almacenan los índices de los polígonos que atraviesan dicho volumen.

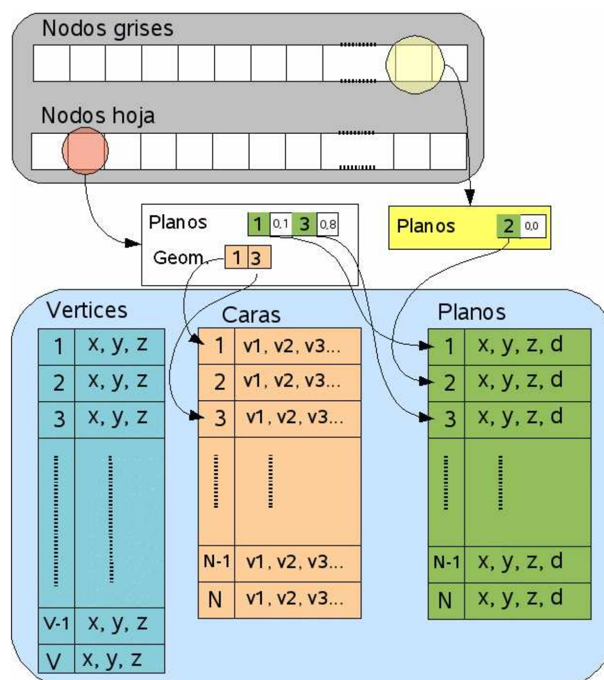


Figura 3.2: *Esquema general de la estructura BP-Octree.*

En las secciones sucesivas, describiremos el proceso de construcción del BP-Octree a partir de una representación *B-rep* de un sólido, en concreto a partir de mallas poligonales. A grandes rasgos, este proceso se puede describir en las siguientes etapas:

- Indexar los polígonos del modelo en un grid 3D cuya resolución sea la máxima del octree que se va a usar como índice espacial.

- Construir el octree, comenzando por la creación de los nodos hoja, que serán las celdas del grid ocupadas por polígonos, y siguiendo por los nodos grises,que se encuentran en la ruta desde la raíz a cada una de las hojas.

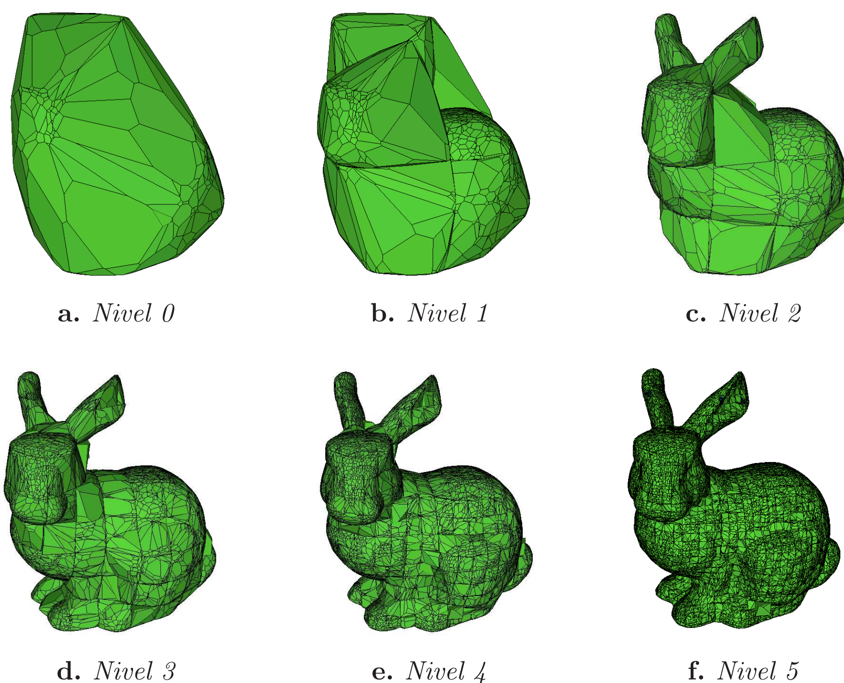
- Agrupar nodos hoja de forma que cada uno contenga un mínimo de polígonos. Esto reduce el número de nodos del árbol y elimina niveles innecesarios para modelos pequeños.

- Crear las envolventes en los nodos hoja, usando para ello los planos de los polígonos que atraviesan cada nodo.

- Crear las envolventes en los nodos grises, usando para cada nodo los planos que forman las envolventes de sus hijos.

- Crear los nodos blancos y negros, de forma que ahora cada nodo gris tiene ocho nodos hijos.

El resultado final es una secuencia de volúmenes envolventes como la que se muestra en la figura 3.3.



a. *Nivel 0* **b.** *Nivel 1* **c.** *Nivel 2*

d. *Nivel 3* **e.** *Nivel 4* **f.** *Nivel 5*

Figura 3.3: *Seis primeros niveles del BP-Octree para el modelo Stanford Bunny.*

3.2. Creación del índice espacial

De las dos aproximaciones clásicas a la construcción de un árbol, *top-down* o *bottom-up*, hemos optado por ésta última. Partimos de una indexación espacial en un grid 3D de los polígonos que representan al sólido original. Esta rejilla tridimensional se corresponde con el último nivel del árbol octal, por lo que es necesario definir a priori el máximo nivel de profundidad del árbol. De esta forma se sabe qué nodos del último nivel del árbol son atravesados por la frontera del sólido, siendo éstos los nodos hoja. A continuación, con los nodos hoja fijados e identificados con su octcode, es sencillo saber qué nodos internos tendrá el árbol, puesto que serán aquellos necesarios para alcanzar a cada nodo hoja desde la raíz.

Aunque los algoritmos funcionarían de forma idéntica con cualquier tipo de malla poligonal, se ha restringido al uso de mallas de triángulos por ser éstas las más extendidas en la comunidad de gráficos actual.

Al ser el octree una estructura discreta, y el espacio tridimensional un entorno contínuo, está claro que la primera tarea que hay que realizar es discretizar nuestro modelo, es decir, asignar cada polígono al conjunto de nodos hoja que atraviesa. Esto lo realizamos usando un octcode, calculado como un código Morton tradicional [Mor66].

El nodo raíz del árbol es una caja envolvente alineada a los ejes (*Axis-Aligned Bounding Box* (AABB), [Ben97]), es decir, no es necesariamente un cubo, sino que se ajusta al objeto en la dirección de los ejes y se determina por dos puntos signicativos: $BB_{min} = (x_{min}, y_{min}, z_{min})$ y $BB_{max} = (x_{max}, y_{max}, z_{max})$.

El criterio básico para indexar todos los polígonos es determinar la celda del grid discreto en la que un punto p está. Para cada dimensión d, en un nivel de profundidad l, la coordenada discreta N_D se calcula como:

$$N_d = floor(\frac{2^l}{w_d}(p_d - d_{min})) \tag{3.1}$$

donde w_d es la longitud del nodo raíz en dicha dimensión, d_{min} es el valor mínimo de la coordenada correspondiente a la dimensión d y p_d el valor de la coordenada d del punto p.

Esta fórmula tiene un comportamiento anómalo -para nuestros propósitos- cuando $p_d = d_{max}$. En este caso, al ser los intervalos abiertos por la derecha, se genera un N_d fuera de la caja envolvente del sólido, como se puede apreciar en la figura 3.4. Hay que controlar este caso especial convirtiendo el último intervalo discreto de cada dimensión en intervalo cerrado por ambos extremos. De esta forma, en la figura 3.4 el vértice V_1 sigue estando correctamente clasificado, y el V_2, en lugar de estar en el voxel que se aprecia rayado, fuera del bounding de la figura –marcado con lineas negras–, se incluiría en el voxel de color cyan. Otra opción es ampliar ligeramente el tamaño del nodo raíz.

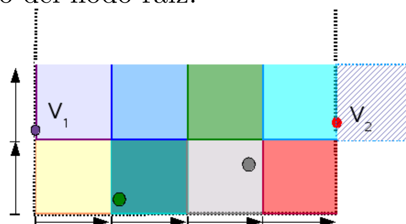


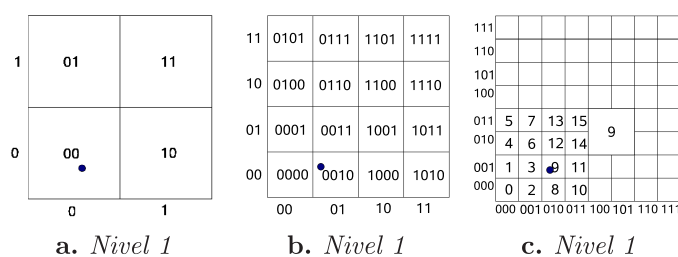
Figura 3.4: *El punto V_2 es clasificado en un voxel que no existe.*

Esta misma fórmula proporciona la numeración de cada uno de los ocho octantes en los que se divide un nodo.

3.2.1. Código Morton

El código Morton de un voxel se calcula entrelazando las coordenadas discretas (expresadas en binario) de cualquiera de los puntos que corresponden a dicho voxel.

En la Figura 3.5 podemos ver cómo se intercalan las coordenadas X e Y resultando un número binario, único para cada nodo de un nivel determinado, pero no único para cualquier nodo de cualquier nivel. Podemos ver en la Figura 3.5c como el número 9 identifica tanto una celda del segundo como otra distinta del tercer nivel, siendo éstas de diferentes tamaños y coordenadas.



a. *Nivel 1* **b.** *Nivel 1* **c.** *Nivel 1*

Figura 3.5: *Numeración de los nodos*

Para solucionarlo, añadimos al código información sobre el nivel al que pertenece, también en binario. Esto convierte el código Morton en un localizador, ya que nos permite con un simple número saber las coordenadas exactas que delimitan el voxel. Hay dos opciones a la hora de añadir la información del nivel:

- Añadir los bits de nivel como los más signicativos (figura 3.6a), lo que nos permite una ordenación primero en anchura de los octcodes.
- Añadir los bits de nivel como los menos signicativos (figura 3.6b), teniendo un orden en profundidad de los octcodes.

level	code

Level 1, node 0: 01 000000 (64)
Level 1, node 1: 01 000001 (65)
Level 2, node 0: 10 000000 (128)
Level 2, node 1: 10 000001 (129)

a. *Orden en anchura*

code	*level*

Level 1, node 0: 000000 01 (1)
Level 1, node 1: 000001 01 (5)
Level 2, node 0: 000000 10 (2)
Level 2, node 1: 000001 10 (6)

b. *Orden en profundidad*

Figura 3.6: *Adición de unos bits de nivel al código Morton*

En nuestra propuesta hemos elegido el orden en anchura para los octcodes, ya que al almacenarlos en dicho orden en un fichero, esta ordenación conserva la coherencia espacial para una transmisión progresiva del modelo.

3.2.2. Determinación de la profundidad del árbol

El hecho de que sean necesarios 3 bits por nivel del árbol, más $log_2 nivel_{maximo}$ bits para especificar el nivel, nos permite calcular de forma fácil la longitud del octcode en función de la profundidad máxima del árbol, y viceversa, fijando una profundidad máxima del árbol, saber qué longitud de palabra será necesaria.

Un árbol con profundidad 9 necesitará $3 * 9 = 27 bits$ para codificar el nodo, más 4 bits para identificar el nivel. Esto hace una longitud de 31 bits. Utilizando un valor entero signado de 4 bytes, `signed int`, podremos representar cualquier nodo de nuestro árbol. Si deseáramos ampliar la profundidad máxima de nuestro árbol, según los tipos de datos disponibles usualmente, habría que pasar a una profundidad de 18 niveles para optimizar los 64 bits del `long int`.

Por este motivo hemos predefinido el uso de un árbol de nivel 9 para nuestra implementación. Esto nos permite tener 8^9 nodos hoja potenciales. Según los modelos utilizados para los tests, resultan suficientes para modelos de varios millones de polígonos.

3.2.3. Asignación de los polígonos a nodos. Algoritmo 3DDDA.

Para garantizar que el plano que contiene a un triángulo es utilizado en todos y cada uno de los nodos atravesados por dicho triángulo, es necesario realizar un recorrido exhaustivo por él.

Intuitivamente se podría ver el algoritmo como la disección del triángulo por todos los planos que forman la rejilla del grid 3D, y la clasificación de cada uno de los puntos de corte.

Este algoritmo es uno de los más críticos de toda la construcción de la estructura de datos, puesto que la presencia de errores de redondeo o coma flotante pueden llevar a saltarnos algún voxel y generar inconsistencias en momentos posteriores del algoritmo de construcción, tal y como se muestra en la figura 3.7.

Como se aprecia en el pseudocódigo 3, ilustrado con una secuencia de su ejecución en la figura 3.8, se trata de realizar cortes al triángulo con planos paralelos a cada una de los planos cartesianos. Se realiza un primer corte, generando un segmento que se a su vez intersecado con otro plano, paralelo también a uno de los dos planos cartesianos restantes. El resultado de estos dos cortes son una serie de puntos en el segmento. Dichos puntos son clasificados en en

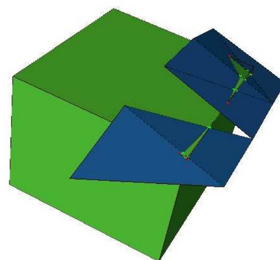


Figura 3.7: *Un error en el algoritmo de reparto de triángulos (superficie azul) provoca que falte uno en el nodo (verde).*

```
1  Para cada triangulo T
2   determinar min[X],min[Y],min[Z],max[X],max[Y],max[Z]
3   Para cada dimension d:{X,Y, Z}
4     Para cada plano P paralelo a d=0 y P_d:[min[d],max[d]]
5     S=T.cortarConPlano(P)
6     asignar(T,octcode(S.Origen))
7     asignar(T,octcode(S.Destino))
8     Para cada dimension d2:{X,Y, Z} Y d2!=d
9      para cada plano P2 PARALELOA d2=0 y P2_d2:[min[d2],max[d2]]
10        PT=S.cortarConPlano(P2);
11         asignar(T,octcode(PT));
12         asignar(T,vecino(octcode(PT),d2,-1);
13     asignar T a los octcode de cada vertice
```

Algoritmo 3: *Pseudocódigo de asignación de triángulos a nodos.*

el índice espacial, y se prosigue el algoritmo para todos y cada uno de los planos susceptibles de atravesar el triángulo.

Puede darse el caso en el que la operación de la línea 12 del pseudocódigo 3 (asignar el triángulo al nodo vecino a la izquierda en la dimensión que se esté realizando el segundo corte) sea incorrecta, esto es, que el triángulo sea cortado por el plano justo en el extremo inicial del segmento S y el vecino *a la izquierda* nunca sea visitado por *T*. En este caso, el error no es tal puesto que se solventa en un paso posterior, cuando se compruebe que el triángulo asignado no pasa por el nodo. Obviamente, se podría evitar realizando comprobaciones extra en este algoritmo, pero entendemos que supone elevar su complejidad cuando es sencillo incluirlo en una fase posterior del algoritmo de construcción del BP-Octree, cuando se construyen las envolventes.

Realizando el reparto según el algoritmo anterior, nos garantizamos que cada triángulo es asignado a todos y cada uno de los nodos por los que pasa. En la figura 3.8 podemos apreciar una secuencia de la evolución del algoritmo.

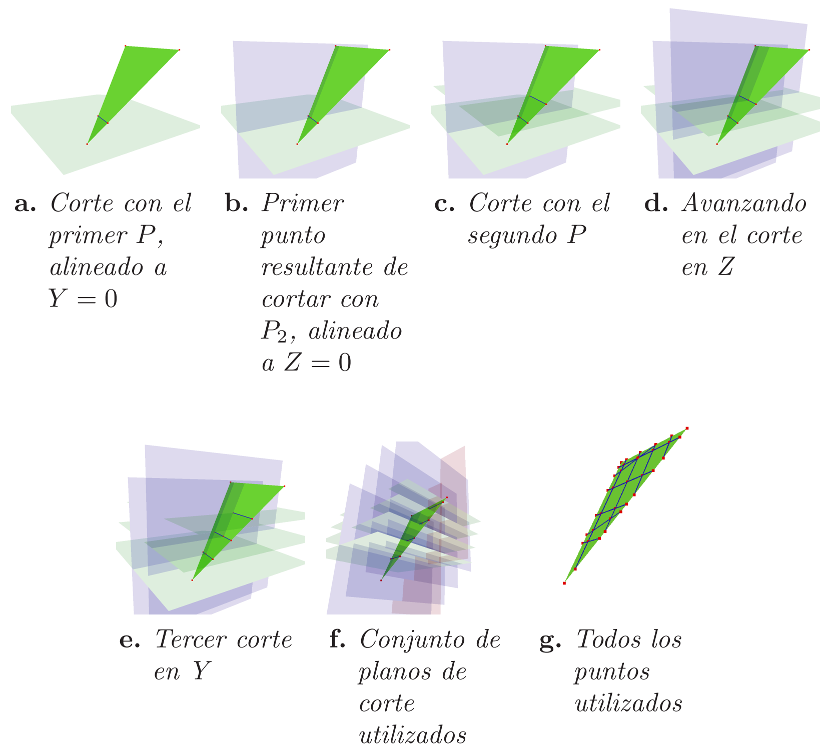
3.2.4. Creación de los nodos hoja.

Partiendo de un vector con duplas <*Polígono, Código*>, se ordena por el octcode y se realiza la reserva de memoria para los nodos hoja, asignando ya los índices de los polígonos.

3.3. Creación del árbol.

Una vez que tenemos identificados los nodos hoja de último nivel que son atravesados por la frontera del sólido, el siguiente paso es generar los nodos internos. Para ello, hacemos uso del octcode generado para los nodos hoja, y, al igual que en el trabajo de Gargantini [Gar82], éste nos indica para cada nivel el hijo que hay que tomar. Si éste no existe, se crea, y se continua leyendo el código.

Dado el octcode $17|354265701_8$ ($17_8 = nivel9$), la ruta a seguir es la que se representa en la figura 3.9



a. *Corte con el primer P, alineado a* $Y = 0$ **b.** *Primer punto resultante de cortar con* P_2*, alineado a* $Z = 0$ **c.** *Corte con el segundo P* **d.** *Avanzando en el corte en Z*

e. *Tercer corte en Y* **f.** *Conjunto de planos de corte utilizados* **g.** *Todos los puntos utilizados*

Figura 3.8: *Representación gráfica del algoritmo de voxelización de triángulos. En azul los segmentos producidos por el corte con el plano P. En rojo, los puntos usados para determinar los voxels por los que pasa el triángulo.*

3.3.1. Compactación de nodos

En algunos modelos, en funcion del número de polígonos y su tamaño relativo a la caja envolvente del objeto, se puede dar el caso de que los nodos hoja del máximo nivel alberguen tan sólo uno o dos polígonos.

Se introduce el concepto de *compactación* de nodos hoja para indicar el proceso por el cual se convierte un nodo interno en nodo hoja mediante la fusión de todos sus hijos, que han de ser hojas, y asignando al nuevo nodo todos los polígonos contenidos previamente en sus descendientes, tal y como se describe en el pseudocódigo 4. Este algoritmo recursivo evalúa, para cada nodo gris, la posibilidad de compactación para cada uno de sus hijos. Tras esta evaluación, si todos sus hijos son hoja, se verifica si cumple los criterios para compactarlos. Si es así, toma los polígonos asignados a ellos y convierte el nodo gris en un nuevo nodo hoja. Si por el contrario, se encuentra que alguno de sus hijos es gris, no se puede aplicar compactación de nodos y retorna la ejecución.

Hay diversos criterios que se pueden aplicar para decidir o no la compactación de los nodos hoja. La elección de uno u otro no afecta en gran medida a la envolvente en los niveles superiores del árbol, pero sí determina el número final de nodos hoja y la profundidad de éstos, como se puede ver en las tablas 3.1 y 3.2.

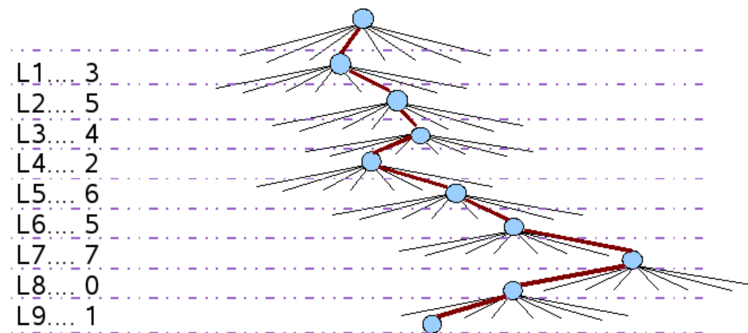


Figura 3.9: *Nodos creados hasta llegar al nodo hoja* $17|354265701_8$

```
BPONode condenseNode(BPOctree bpo,BPONode node)
    isCondensable=false;
    foreach child_i of node {
        if isGray(child) {
            nc=condenseNode(bpo,child);
            if (!isLeaf(nc))
                isCondensable=false;
            else {
                isCondensable=true;
                setChild(node,i,nc);
                delNode(child);
            }
        }
    }
    if (isCondensable && checkCriterion(node)) {
        leaf=newLeaf(bpo,node->getCode());
        leaf->setBBox(node->getBBox());
        foreach child of Node {
            leaf.addFaces(child.faces());
            delLeaf(child);
        }
        return leaf;
    } else
        return node;
}
```

Algoritmo 4: *Algoritmo recursivo de compactación de nodos*

Los dos criterios evaluados son los siguientes:

1. Agrupar mientras el nodo hoja resultante quede con menos de N polígonos asignados.
2. Agrupar hasta que la media de polígonos entre los nodos hermanos sea al menos AV.
3. No agrupar

Mostramos a continuación unas tablas con las estadísticas de nodos hoja para diversos modelos, con $N = 50$ y $AV = 20$.

	Criterio 1	Criterio 2	Criterio 3
Hojas en nivel 1	0	0	0
Hojas en nivel 2	1	0	0
Hojas en nivel 3	17	1	0
Hojas en nivel 4	189	10	0
Hojas en nivel 5	1706	177	0
Hojas en nivel 6	11694	3779	0
Hojas en nivel 7	13	12	0
Hojas en nivel 8	-	-	0
Hojas en nivel 9	-	-	1089598

Tabla 3.1: *Número de hojas para el modelo Stanford Bunny*

	Criterio 1	Criterio 2	Criterio 3
Hojas en nivel 1	0	0	0
Hojas en nivel 2	0	0	0
Hojas en nivel 3	0	3	0
Hojas en nivel 4	8	62	0
Hojas en nivel 5	898	781	0
Hojas en nivel 6	8280	20564	0
Hojas en nivel 7	66824	2533	0
Hojas en nivel 8	18187	-	0
Hojas en nivel 9	6	-	1551983

Tabla 3.2: *Número de hojas para el modelo Fertility*

A lo largo de esta memoria, los BP-Octrees generados se han construido con el primer criterio, y $N = 50$.

```
calcularEnvolvente(Node_T node){
 si esHoja(node)
  node->selectBoundingPlanes();
 si_no {
  para cada ch hijo de node {
   calcularEnvolvente(ch);
   node->addBPlanes(ch.getBPlanes());
   node->addBVertices(ch.getBVertices());
  }
  node->selectBoundingPlanes();
 }
}
```

Algoritmo 5: *Cálculo recursivo de la envolvente en cada nodo.*

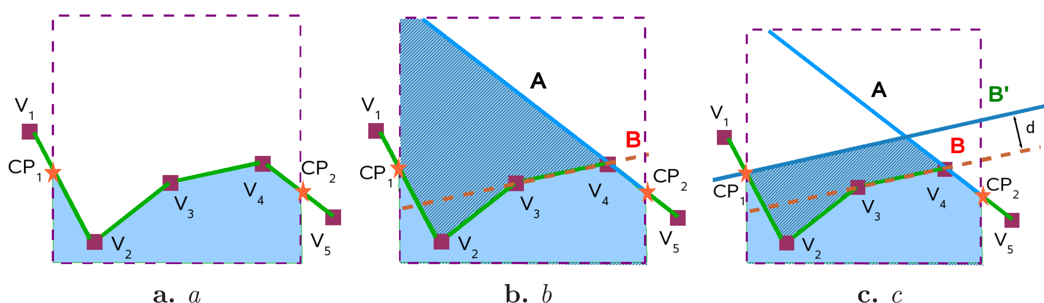
3.4. Creación de la jerarquía de planos envolventes.

Una vez que se tienen los polígonos repartidos de forma exhaustiva por los nodos hoja, se procede a construir las envolventes de éstos.

Como se ha comentado anteriormente, en cada nodo guardamos los índices de un conjunto de planos que delimitan completamente la parte del modelo 3D contenida en el nodo. Para determinar este conjunto de planos, seguimos un procedimiento recursivo ascendente, de forma que en cada nodo se realiza el menor número de cálculos posible, es decir, seguimos un paradigma *divide y vencerás*.

Como puede verse en el pseudocódigo 5, en cada nodo seleccionamos sólo los planos que engloban la geometría de las envolventes de sus hijos con `addBVPlanes` (o la geometría real del modelo si estamos en una hoja) y sólo se usan los vértices de los hijos (`addBVVertices`) para garantizar que el volumen de la envolvente aumenta conforme subimos en el árbol.

El método utilizado para determinar si un plano forma parte de la envolvente o no es relativamente simple, y se muestra gráficamente en 2D mediante la figura 3.10. Partimos de un conjunto de planos *candidatos*, que son los que forman las envolventes de los nodos hijos



a. *a* **b.** *b* **c.** *c*

Figura 3.10: *Selección de planos envolventes. En cyan el volumen envolvente.*

(o la geometría original en el caso de que estemos en un nodo hoja). Comprobamos para cada plano candidato -desplazado si tuviese offset d previamente asignado en niveles inferiores- la posición de los vértices con respecto a su orientación. Si todos los vértices están en el semiespacio interior o son coplanares, el plano es tomado tal cual para formar parte de la envolvente. Este es el caso del plano A en la figura 3.10a. El volumen envolvente resultante es el área cyan.

3.4.1. Cálculo de planos envolventes en los nodos hoja.

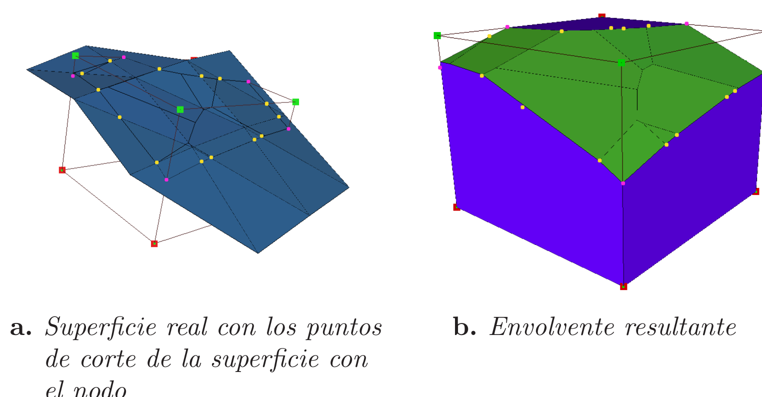
Para el cálculo de las envolventes en los nodos hoja, además de los vértices incluidos en el volumen del nodo, se tienen en cuenta los puntos de corte de los polígonos de la frontera con las caras del nodo. Este cómputo supone unas décimas de segundo más en la construcción del BP-Octree, pero es indispensable para obtener, como se explicará más adelante, la información necesaria para detectar los nodos *negros* y *blancos*.

Estos puntos calculados en las caras del nodo tienen dos orígenes:

- *Corte arista de cara-nodo*. Puntos de intersección entre las aristas de las caras de la frontera del sólido con las caras del nodo (puntos amarillos en la figura 3.11).
- *Corte cara-arista de nodo*. Puntos de intersección entre las caras de la frontera y las aristas del nodo (puntos rosa en la figura 3.11). Estos puntos están siempre en las aristas del nodo.

Además, éstos últimos puntos de corte nos permiten determinar si cada una de las ocho esquinas del nodo se encuentra dentro o fuera del sólido, ya que disponemos de la información sobre la orientación de las caras que produce cada uno de los cortes.

En la figura 3.11b se puede apreciar en verde la configuración de los planos seleccionados para formar la envolvente convexa, apreciándose claramente como ésta contiene a todos y cada uno de los puntos situados en el perímetro del nodo.



a. *Superficie real con los puntos de corte de la superficie con el nodo* **b.** *Envolvente resultante*

Figura 3.11: *Envolvente resultante en un nodo hoja.*

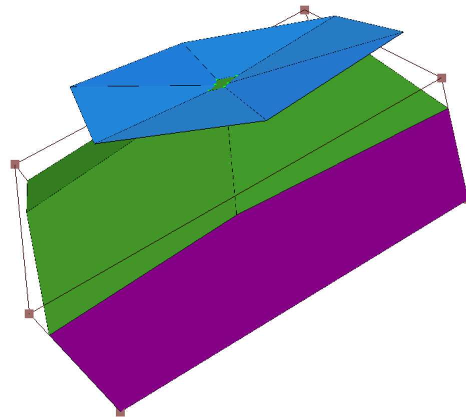
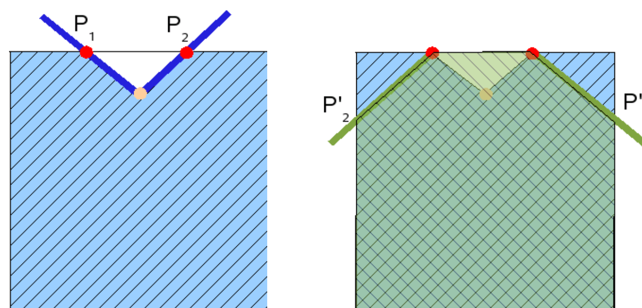


Figura 3.12: *Configuración errónea de planos envolventes (verde). La superficie del sólido, en azul, es una concavidad formada por todas las caras que comparten un mismo vértice, estando contenido éste en el interior del voxel y los demás vértices fuera.*

3.4.1.1. Etiquetado de las esquinas de los nodos.

El etiquetado de las esquinas de los nodos es un proceso necesario para la construcción del conjunto de planos envolventes, porque, pese a lo que pudiera parecer en un principio, se producen casos especiales en los que es necesario hacer uso de dicha información. Es el caso en el cual la frontera del sólido atraviesa el nodo sin cortar ninguna de sus aristas, tal y como se muestra en la figura 3.12. En la figura 3.13a se puede apreciar un corte de dicho nodo, donde se marca en azul rayado el interior del sólido, y el vértice de la geometría así como los puntos de corte de las aristas en la cara del nodo. en la figura 3.13b se puede ver cómo tan sólo con la informacion que disponemos, se genera una envolvente, marcada en color verde, que cumple los criterios indicados anteriormente (se obtiene desplazando los planos P_1 y P_2, y envuelve a los vértices de la geometría contenida), pero que es errónea.



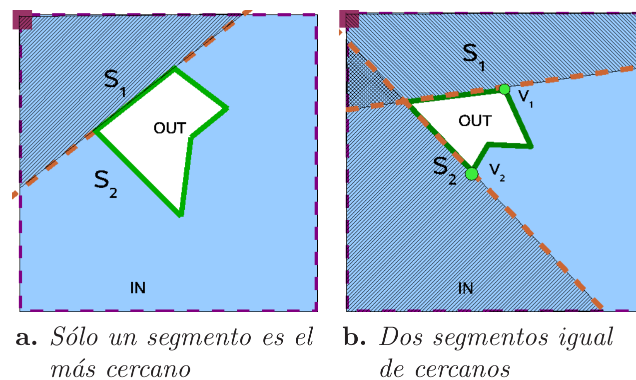
a. *El sólido tiene una concavidad que no cortan ninguna arista del nodo.*

b. *Se genera una envolvente errónea.*

Figura 3.13: *Esquema 2D de lo representado en la figura 3.12*

Este tipo de configuraciones conflictivas, que dan lugar a envolventes erróneas, se solucionan etiquetando adecuadamente las esquinas de los nodos. Al no haber información de cortes en las aristas del nodo, hay que recurrir a la información generada por los puntos de corte de aristas de caras con el nodo (en verde luminoso en la figura 3.12), pasando el problema a dos dimensiones y resolviéndolo como una cuestión de cálculo de inclusión punto en polígono.

El algoritmo para determinar si una esquina se encuentra dentro o fuera del polígono que forma la intersección de las caras del sólido con la cara del nodo, se representa gráficamente en la figura 3.14 (se muestran los semiespacios interiores con trama rayada). En la figura 3.14a se selecciona la arista del polígono más cercana a una de las esquinas, S_1 . Si el punto de dicho segmento más cercano a la esquina de la cara del nodo es uno de los extremos, como ocurre en la figura 3.14b, hay que tomar las dos aristas incidentes en dicho vértice, S_1 y S_2. Los planos de las caras del sólido que forman dichos segmentos se ortogonalizan con respecto a la cara del nodo que es atravesada, de forma que se puede determinar en 2D el semiespacio interior y el exterior de cada arista.



a. *Sólo un segmento es el más cercano* **b.** *Dos segmentos igual de cercanos*

Figura 3.14: *Resolución de la inclusión de las esquinas del nodo cuando el sólido atraviesa el nodo sin cortar aristas del mismo. En azul el interior del sólido. En blanco el exterior.*

El caso representado en la figura 3.14a es trivial, y la posición de las ocho esquinas del nodo concicide con la posición de la esquina seleccionada con respecto a la arista S_1. En el caso representado en la figura 3.14b, hay que tener en cuenta la posición de los vértices no compartidos por las aristas seleccionadas S_1 y S_2, V_1 y V_2 con respecto a la arista opuesta. En el caso representado, tanto V_1 como V_2 se encuentran en el semiespacio exterior de S_2 y S_1 respectivamente, por lo que las esquinas del nodo se encuentran en el interior del sólido.

Para los nodos cuyas aristas son atravesadas, la clasificación de las esquinas se hace en función de los cortes producidos en las aristas. En la figura 3.11a podemos apreciar en color *amarillo* los puntos de corte de las aristas de las caras del sólido con los límites del nodo, y en color *rosa* los cortes de las aristas del nodo con las caras del polígono. En la figura 3.11 se aprecia igualmente cómo las esquinas del nodo están coloreadas en verde o en rojo. Esto indica su posición con respecto a la frontera del sólido: verde es *exterior* y rojo es *interior*. Además del punto de corte en la arista del nodo, tenemos en cuenta el plano del polígono que produce dicho corte, puesto que será la orientación de la esquina del nodo con respecto a

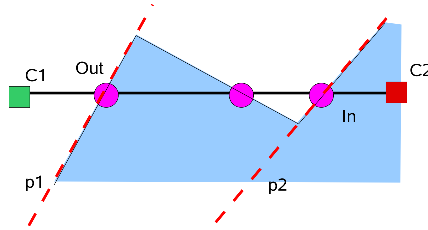


Figura 3.15: *El plano p_1 deja en su semiespacio exterior a la esquina del nodo C_1. Análogamente, C_2 es etiquetada como interior al estar en el semiespacio interior definido por p_2*

dicho plano la que determinará su posición. En la figura 3.15 se puede apreciar que el plano p_1 es el que produce el corte más cercano al vértice C_1, por lo que la orientación de este último se calcula con respecto al semiespacio interior definido por p_1. Análogamente, la esquina C_2 del nodo se etiqueta como *interior* al estar en el semiespacio interior del plano p_2

En el caso de que un punto sea generado por la intersección de una arista de polígono, se añade un offset al mismo en dirección a su normal, para que los dos polígonos adyacentes, si tienen orientación opuesta, no generen el mismo punto de corte.

Una vez que se han etiquetado las esquinas con el procedimiento anteriormente descrito, puede que haya algunas cuyas aristas incidentes no sean atravesadas por ningún polígono. En este caso, se realiza una propagación de los valores, puesto que si una esquina está *dentro*, las tres esquinas adyacentes estarán también dentro salvo que sean atravesadas por la frontera del sólido, en cuyo caso ya habrán sido evaluadas.

3.4.2. Selección de planos relevantes.

Una vez que se dispone del conjunto de vértices que delimitan la geometría contenida en el nodo, los propios de la geometría del sólido y los calculados al intersecarla con el nodo en sí, se aplica el algoritmo descrito en el pseudocódigo 5.

Este algoritmo tiene un buen comportamiento en superficies mixtas, con convexidades y concavidades, puesto que permite una simplificación del modelo bastante ajustada a la frontera original. Sin embargo, con superficies extremadamente convexas, como las que se aprecian en la figura 3.16, donde se utilizan un gran número de planos que, si bien ajustan perfectamente a la superficie, no permiten conseguir uno de los objetivos del *BP-Octree*, la simplificación de la geometría.

Para eliminar de una forma eficiente la redundancia de los planos, se utiliza un algoritmo de agrupamiento o *clustering*, que se encarga de clasificar los planos en grupos o *clusters* en función de su similitud. En nuestro caso, el **criterio de similitud** considerado es la normal del plano, de forma que en una superficie plana, todos los planos serían sustituidos por un único representante, y en una superficie esférica, se elegirían K representantes homogeneamente distribuidos a lo largo de todo el espacio, tal y como se muestra en la figura 3.17.

Utilizaremos para esta clasificación un algoritmo *k-medoides*, que es una variante del clásico *k-medias* [HKT01] en el cual, en lugar de tomar el centro del *cluster* como valor representativo, utiliza el objeto más centrado del grupo. Además, es menos sensible a ruido y valores anómalos

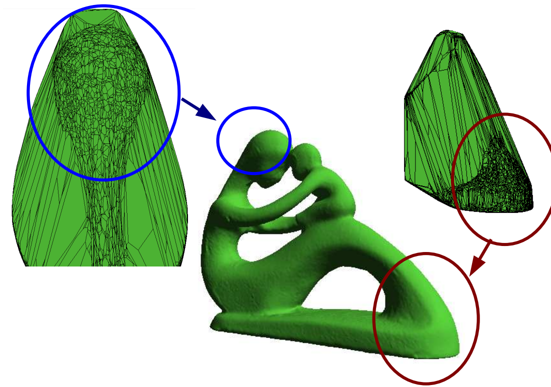


Figura 3.16: *Alta densidad de planos debido a áreas del modelo extremadamente convexas.*

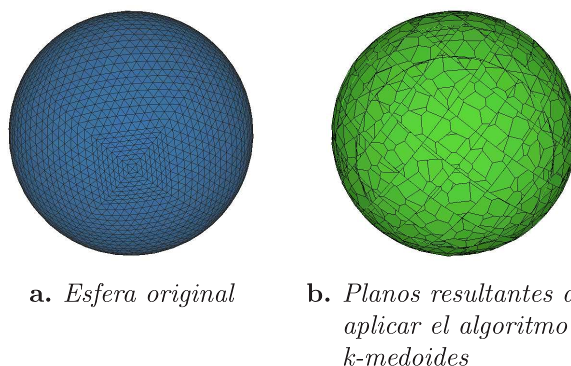
que el algoritmo *k-medias* [HKT01]. Se han propuesto diversos algoritmos para implementar el concepto del *k-medoides*, siendo el *PAM (Particionamiento Alrededor de los Medoides)* propuesto por Kaufman y Rousseeuw el más antiguo [KR90]. Se puede describir el algoritmo, aplicado a nuestro propósito, como sigue [NH94]:

1. Seleccionar k planos representativos de forma aleatoria.

2. Calcular el coste total TC_{ih} para todos los pares de planos $< P_i, P_h >$, donde P_i está seleccionado, y P_h no.

3. Seleccionar el par $< P_i, P_h >$ que tiene el valor mínimo para TC_{ih}. Si éste valor es negativo, sustituir P_i por P_h y volver al paso 2.

4. En caso contrario, para cada plano no seleccionado, encontrar su representante más cercano. Fin del algoritmo.



a. *Esfera original*

b. *Planos resultantes de aplicar el algoritmo k-medoides*

Figura 3.17: *Efectos del algoritmo k-medoides.*

Para calcular el coste de un par de planos usamos un valor de distancia euclídea en 2D. Para ello, convertimos el vector normal de cada plano a un sistema de coordenadas esférico. Al ser todos los vectores normalizados, con radio de la esfera $r = 1$, podemos comparar las normales tan sólo con sus ángulos θ y ρ, de forma que $TC_{ih} = (\theta_i - \theta_j)^2 + (\rho_i - \rho_j)^2$.

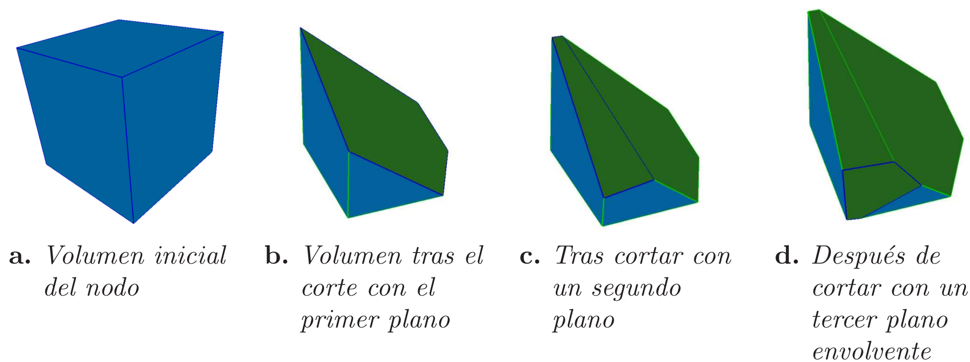
Uno de los inconvenientes de este algoritmo es que hay que definir el valor de K a priori, por lo que podría ser de utilidad en futuras implementaciones usar algoritmos evolutivos u otros que no fijen el número de clusters a buscar, como el *QT clustering (Quality Treshold clustering)* [HKY99]. Además, el valor elegido para K influye notablemente en el tiempo de construcción del árbol, como se analizará en la sección 3.5.1.

En cualquier caso, resulta claro que es necesario realizar una simplificación guiada del conjunto de planos inicialmente seleccionados en la envolvente para evitar la masificación de planos en superficies completamente convexas. El algoritmo a utilizar para ello puede ser objeto de un ulterior estudio detallado.

3.4.3. Recorte de paralelepípedo con el conjunto de planos.

Una vez se ha determinado el conjunto de planos envolventes, hay que construir el volumen que definen los semiespacios interiores de dichos planos. La geometría resultante nos proporcionará la información necesaria para construir las envolventes en el nivel superior.

El proceso se describe visualmente en la figura 3.18. Se parte de una malla formada por las seis caras del nodo, y se van aplicando cortes sucesivos con los distintos planos seleccionados como integrantes de la envolvente. Aquellas caras que conserven los vértices originales del nodo serán polígonos no integrantes de la envolvente –en azul en la figura 3.18– , pero que pueden ser necesarios para tapar discontinuidades entre nodos adyacentes. Los vértices de los polígonos dibujados en verde en la figura 3.18 son los considerados *vértices envolventes*, y son los que se pasan al nodo ancestro para calcular la envolvente en un nivel superior.



a. *Volumen inicial del nodo* **b.** *Volumen tras el corte con el primer plano* **c.** *Tras cortar con un segundo plano* **d.** *Después de cortar con un tercer plano envolvente*

Figura 3.18: *Recorte de la malla del nodo para crear la superficie envolvente.*

El pseudocódigo de recorte de una malla se presenta en el algoritmo 6. Básicamente el proceso es etiquetar cada uno de los vértices según su orientación con respecto al plano de corte p. Si el plano atraviesa una arista, se divide la arista y se etiquetan apropiadamente los tres vértices. Finalmente, se conectan todos los vértices coplanares, creando una nueva cara poligonal y se eliminan aquellas caras que tienen algún vértice fuera.

```
mesh::clipAgainstPlane(Plane p){

   para cada semiarista ei de la malla m
      etiquetar ei.orig y ei.dest segun su posicion respecto a p

   para cada semiarista ei de la malla m
      si p no corta ei
        etiquetar ei igual que ei.orig
      sino
        calcular el punto de corte cp
        dividir ei por el vertice cp -> ei1,ei2
        etiquetar ei1 y ei2 adecuadamente
    }
    crear una nueva cara poligonal que conecte los vertices coplanares
    eliminar las caras que tengan algun vertice etiquetado como OUT
}
```

Algoritmo 6: *Recorte de una malla con un plano*

Este proceso es el más costoso en tiempo de toda la construcción del árbol, puesto que hay que realizarlo para todos y cada uno de los nodos del árbol, tanto hojas como grises y su complejidad depende directamente del número de planos seleccionados.

Este algoritmo, además de para el cálculo de las envolventes, se utiliza para la visualización del árbol, ya que es la geometría resultante la que se visualiza para cada uno de los nodos.

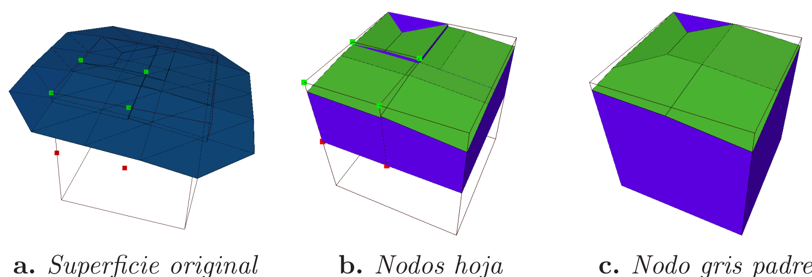
3.4.4. Cálculo de planos envolventes en los nodos grises.

El cálculo de las envolventes en los nodos grises se realiza tal y como se muestra en el pseudocódigo 5. Para cada nodo, se solicita de sus hijos el conjunto de planos que tienen en sus respectivas envolventes, y los vértices que generan dichos planos al aplicar el algoritmo 6, para calcular el volumen envolvente.

De esta forma se garantiza que:

a) Sólo se utilizan planos de los hijos para crear la envolvente del nodo.

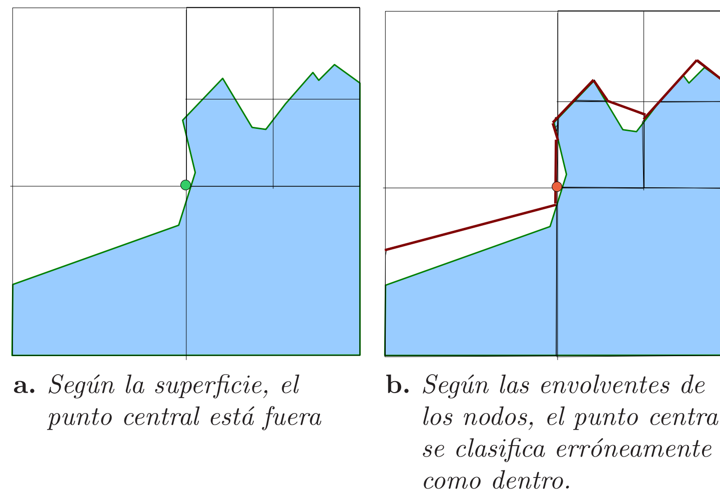
b) La envolvente del nodo contiene a las envolventes de los hijos, como se aprecia en la figura 3.19.



a. *Superficie original* **b.** *Nodos hoja* **c.** *Nodo gris padre*

Figura 3.19: *Envolvente de un nodo gris y sus nodos hijos.*

El proceso de etiquetado de las esquinas de los nodos grises se realiza heredando los valores de las esquinas análogas de los nodos hijos, por lo que se garantiza que, en todo momento, esta propiedad viene dada por la superficie real del sólido, y no por las envolventes de cada momento, como sucede en la figura 3.20, donde no coincide el valor según la frontera con el valor de la envolvente.



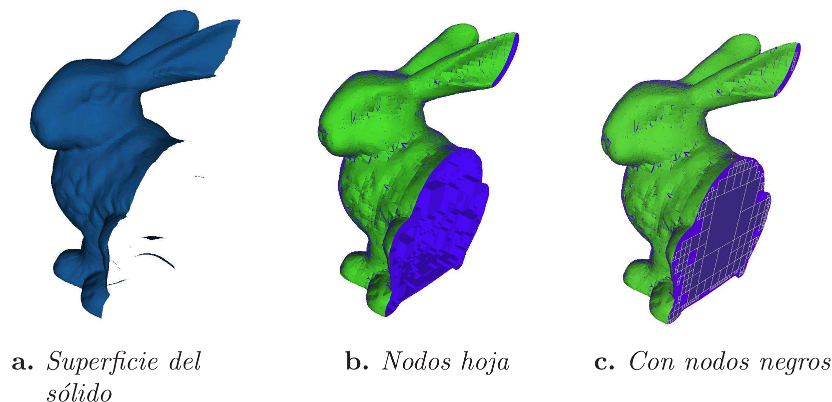
a. *Según la superficie, el punto central está fuera*

b. *Según las envolventes de los nodos, el punto central se clasifica erróneamente como dentro.*

Figura 3.20: *Configuraciones en 2D que ejemplifican cómo las esquinas se han de etiquetar siempre según los valores dados por la frontera del sólido.*

3.4.5. Creación de nodos negros y blancos.

Como se puede apreciar en la figura 3.21, los nodos hoja y grises son suficientes para tener una representación cerrada de la envolvente del modelo representado.



a. *Superficie del sólido*

b. *Nodos hoja*

c. *Con nodos negros*

Figura 3.21: *Corte del modelo Stanford Bunny donde se aprecia la superficie del sólido y la envolvente generada al máximo nivel de detalle.*

Sin embargo, estos nodos no permiten delimitar completamente el interior y el exterior del sólido, puesto que se pueden dar configuraciones como las mostradas en la figura 3.22. Se puede apreciar cómo del nodo gris que contiene en su volumen envolvente al punto P, pasamos a que no existe ningún nodo gris u hoja que contenga a dicho punto P, y que, con la información de que disponemos, no podemos asegurar que P está dentro o fuera del sólido, salvo que hagamos un recorrido exhaustivo por todos los descendientes del árbol.

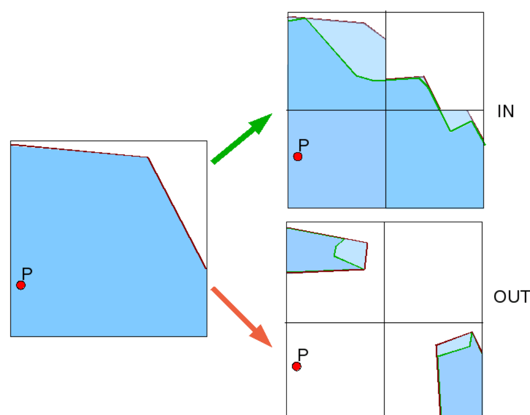


Figura 3.22: *A la izquierda, volumen envolvente en un nodo gris. El punto P puede estar fuera o dentro del sólido, según la configuración de la superficie del sólido, que da lugar a una misma envolvente en un nivel superior.*

Para resolver situaciones como estas, hacemos uso de la información almacenada relativa a las esquinas de los nodos, tanto los nodos hoja como los interiores. La cuestión es bien sencilla: de los ocho hijos que tiene un nodo gris, los que falten serán negros si el centro del nodo está en el interior del sólido, y blancos si está en el exterior. Si dicho punto central del nodo, que coincide con la esquina común a los ocho hijos, es atravesado por la superficie, se consulta con cualquiera de las esquinas ya evaluadas por ser compartidas con el nodo que falta.

En la figura 3.22, en el caso reflejado en la imagen derecha superior, la esquina compartida por los cuatro nodos es IN, por lo que el nodo en el que se encuentra el punto P es *negro*. En el inferior, la esquina sería etiquetada como OUT, por lo que queda fuera.

Es importante recordar que el etiquetado de las esquinas de los nodos se hace por herencia ascendente, tomando para cada esquina el valor que tiene en el nodo hijo que la contiene, y así recursivamente hasta obtener el valor por la frontera en un nodo hoja.

3.5. Tiempos de construcción.

El tiempo de construcción del BP-Octree depende, principalmente, del número de polígonos y la convexidad del objeto tridimensional a representar. En la tabla 3.3, y su representación gráfica en la figura 3.23, se puede apreciar cómo hay una correspondencia casi lineal entre el número de polígonos y los segundos invertidos en la construcción del árbol[1] . Los modelos poligonales utilizados se muestran al final del capítulo, en la figuras de la 3.33 a 3.44.

[1] Estos tiempos han sido tomados en un AMD 64bits, 2GB RAM. Ejecutable para Linux generado con el compilador *gcc* con optimización -O3

Modelo	*Poligonos*	*Segundos*
Esfera	8.192	6,58
Bunny	71.040	43,72
Golf	100.243	105,22
Armadillo	150.000	130,57
Egipcio	161.909	180,60
Teeth	233.204	320,28
Fertility	483.226	1.530,38
Angelo	674.764	1.271,99
Phlegmatic Dragon	715.933	1.182,52
Stanford Dragon	871.414	963,77
Happy Budda	1.087.716	1.573,40
Blade	1.765.388	1.565,60
Dragon	7.218.906	8.614,58

Tabla 3.3: *Tiempos de construcción del BP-Octree para diversos modelos.*

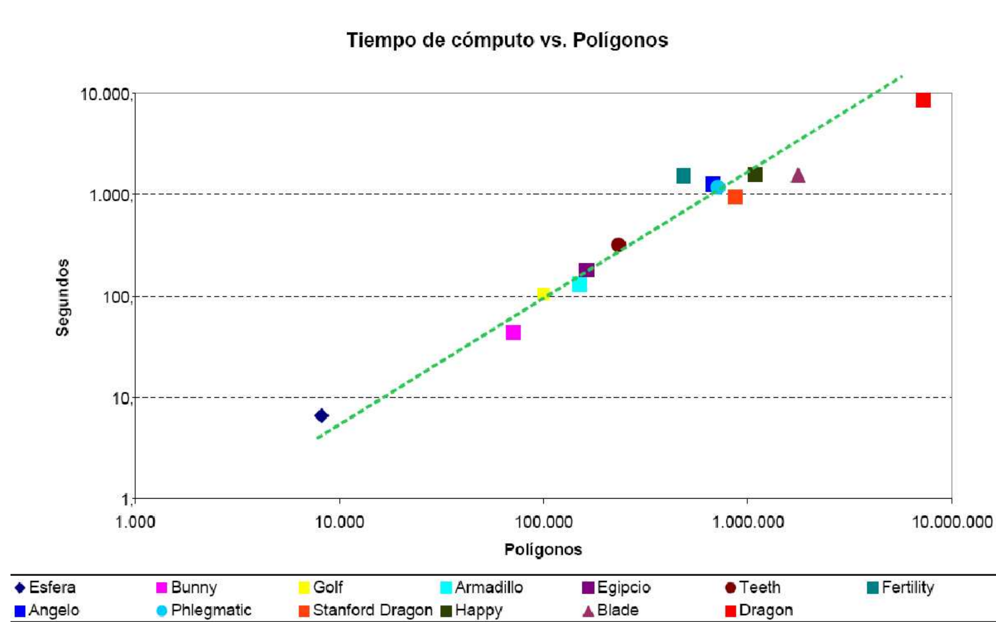
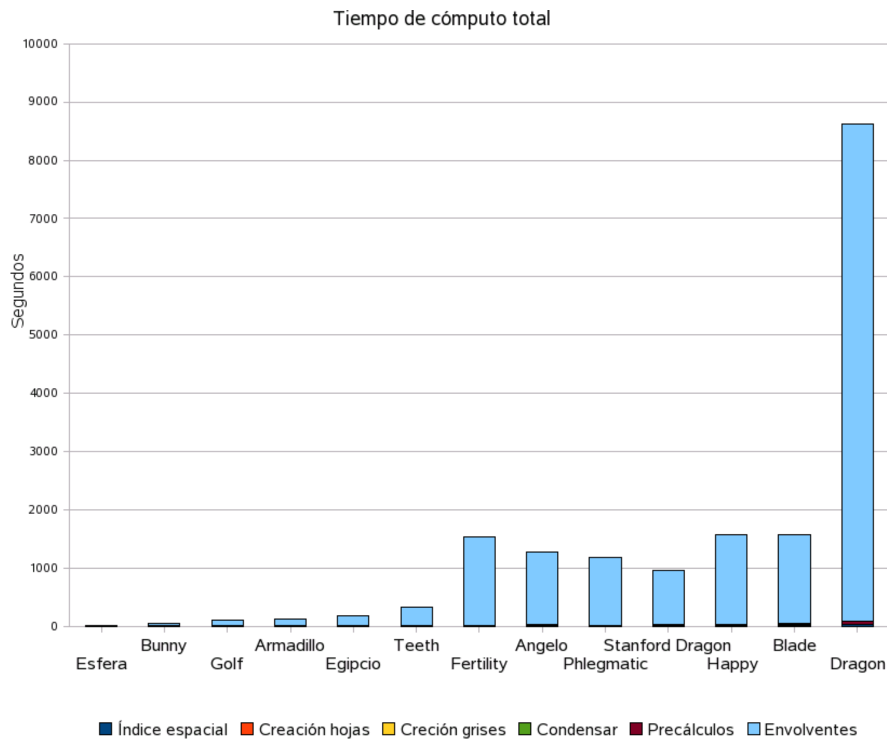
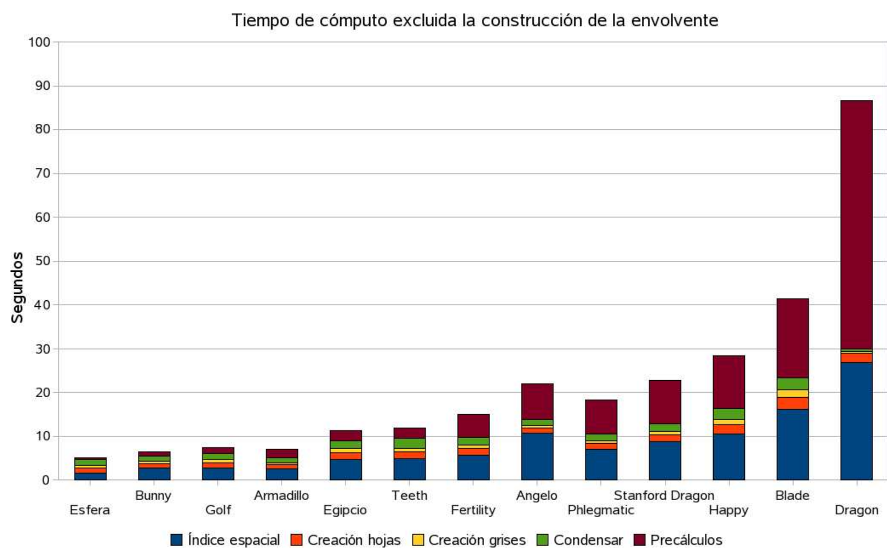


Figura 3.23: *Tiempos de construcción del BP-Octree relacionado con el número de polígonos de cada modelo.*

En la figura 3.24a podemos observar gráficamente cómo se reparten estos tiempos en cada una de las etapas de construcción del BP-Octree, aunque sólo se pueden discernir las primeras etapas excluyendo del gráfico la etapa de cálculo de las envolventes, lo que se muestra en la figura 3.24b.



a. *Tiempos de construcción por etapas*



b. *Tiempo necesario para cada etapa de precálculo*

Figura 3.24: *Tiempos de construcción del BP-Octre para cada uno de los modelos.*

Destacan algunos modelos que se apartan ligeramente de la linealidad, como puede ser el caso del modelo *Fertility*, que consume más tiempo en su construcción que otros modelos con más polígonos, como es el caso de Angelo (191K polígonos más, y tarda 260 segundos menos) o el Stanford Dragon (388K polígonos más y 568 segundos menos). Esto es debido a la elevada convexidad de la figura, que hace que un gran número de planos sean tenidos en cuenta para la construcción de las envolventes en los nodos hoja y en todos sus ancestros, lo que aumenta la complejidad de los algoritmos de selección y recorte de las mallas.

Queda claro que la mayor parte del tiempo de cómputo, más del 95 % en cualquier caso, se consume en el cálculo de las envolventes convexas, por lo que la optimización del algoritmo ha de realizarse en esta etapa.

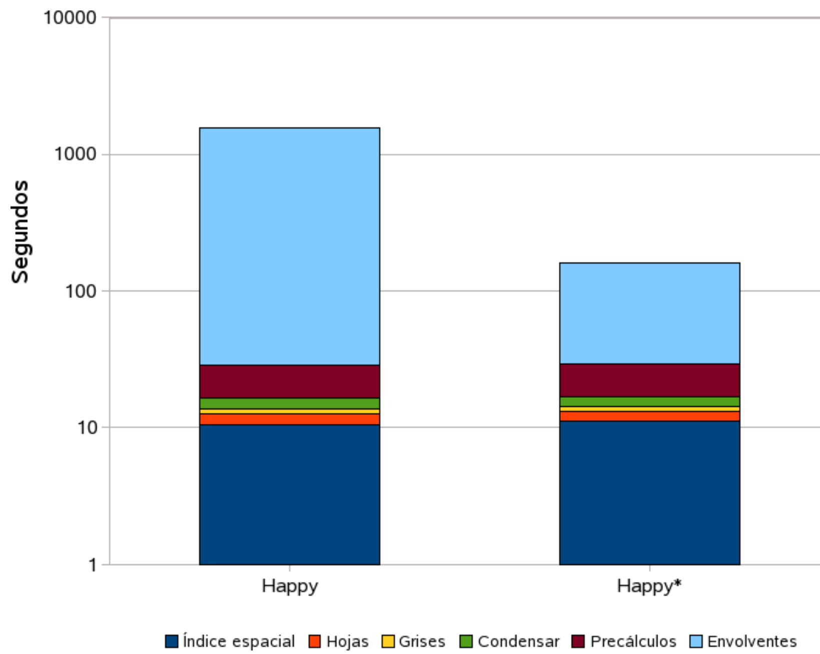
3.5.1. Influencia del parámetro K

Una mejora considerable en los tiempos de construcción del árbol se consigue variando el parámetro K del algoritmo k-medoides.

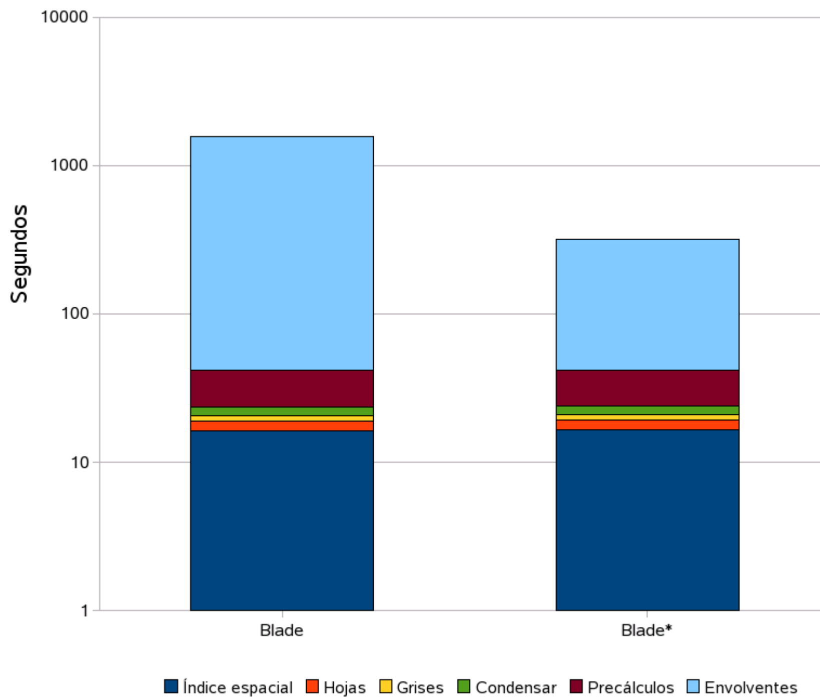
Los tiempos de la tabla 3.3 han sido tomados con una distribución de K tal que en el nivel L_0, $K(L_0) = k * NP$, $k = 1{,}00\,\%$, siendo NP el número de polígonos, y $K(L_i) = K(L_{i-1})/2$.

Como criterio añadido, y para garantizar que siempre se descartan planos al crear la envolvente de un nodo n correspondiente a un nivel L_i, si el número de planos susceptibles de formar parte de dicha envolvente es menor que el valor de $K(L_i)$, se divide $K(L_i)$ por dos sucesivamente hasta que el valor obtenido es menor que el número de planos candidatos de que disponemos. Este cambio en el valor de $K(L_i)$ no afecta a otros nodos del mismo nivel ni de niveles superiores, puesto que es una solución local para asegurar la simplificación del modelo en cada nodo.

En la tabla 3.4 podemos comprobar cómo reduciendo el valor $K(L_0)$ a $K(L_0) = 0{,}0005 * NP$, esto es un factor $k = 0{,}05\,\%$, se mejora notablemente el tiempo de cómputo en tres de los modelos más grandes utilizados para las pruebas, así como una reducción en el número de planos. Se aprecia gráficamente en la figura 3.25. En la figura 3.26 podemos ver cómo la mayoría de los planos que se han descartado son aquellos que se encuentran en las convexidades, de forma que el resultado final no se ve extremadamente afectado.



a. *Evolución del tiempo para el modelo Happy Budda con* $k = 0{,}01$ *(Happy) y* $k = 0{,}0005$ *(Happy*)*



b. *Evolución del tiempo para el modelo Blade con* $k = 0{,}01$ *(Blade) y* $k = 0{,}0005$ *(Blade*)*

Figura 3.25: *Reducción del tiempo de cálculo de envolventes variando el parámetro K.*

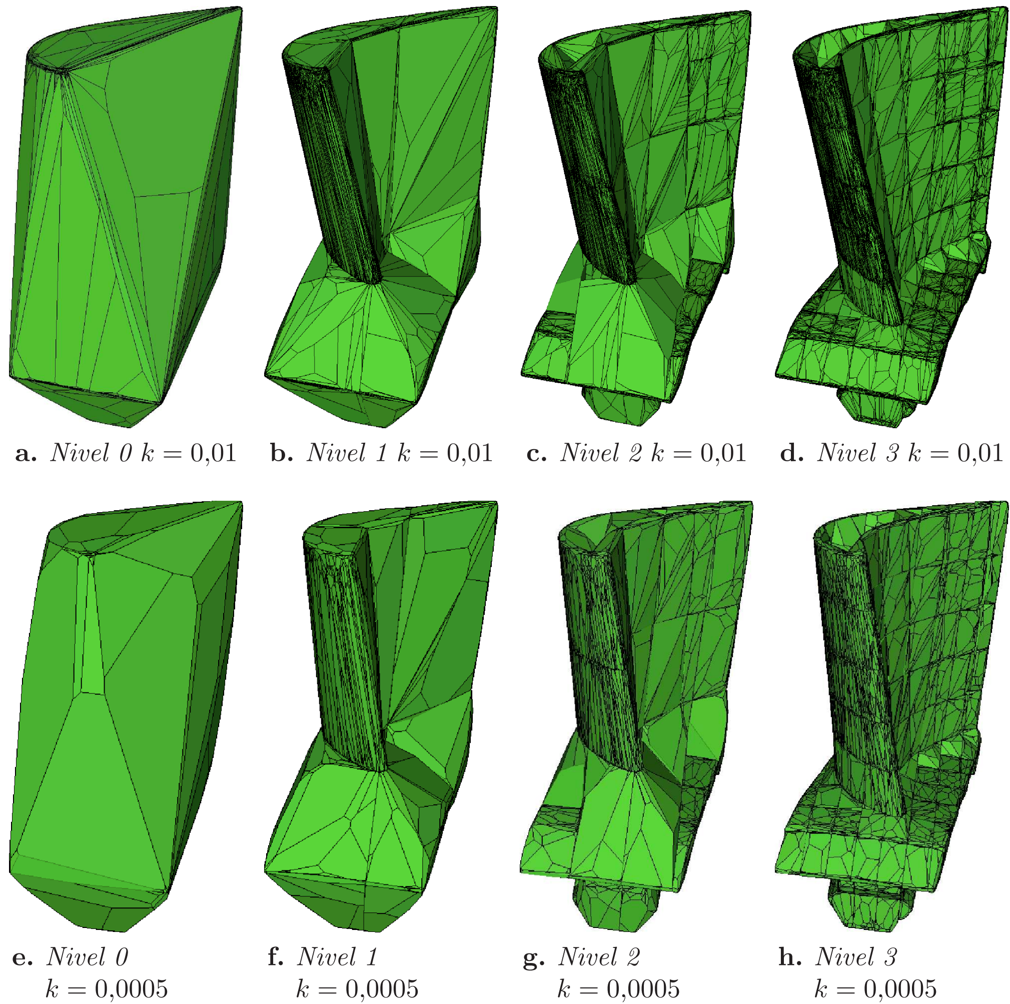
Se han realizado diversas pruebas variando los valores de k iniciales, obteniendo unos resultados bastante ilustrativos. En la figura 3.27 podemos ver el número de planos medio por nodo para cada nivel para el modelo Stanford Dragon, de 871K polígonos. Se puede apreciar como para los valores de k mayores, apenas hay diferencia (ver datos numéricos en tabla 3.5), mientras que a partir de $k = 0{,}001$ la diferencia es sustancial. Destaca cómo para valores muy pequeños de k, en los últimos niveles del árbol, se produce un comportamiento anómalo: para $k = 0{,}0001$, en el nivel 7 se usan 12.72 planos de media, mientras que en el nivel 6, la media es de 1.04. Ello se debe a que el algoritmo, si $K(L_i) < 1$, acepta todos los planos candidatos en el nivel i. Nótese que este comportamiento se produce en niveles más altos cuanto menor es el valor de $K(L_0)$.

Otro dato interesante que se extrae de la tabla 3.5 es que con $k = 0{,}001$, los valores medios de planos obtenidos se aproximan bastante al K requerido, mientras que con $k = 0{,}005$ y sobre todo con $k = 0{,}01$ es necesario usar K de niveles inferiores.

En la gráfica de la figura 3.28 mostramos cuántos planos han sido utilizados en el nodo raíz con respecto al valor original de $K(L_0)$, que sería el valor 100 %. Podemos apreciar que para todos los modelos en el nodo raíz es necesario reducir al menos a la cuarta parte el valor de $K(L_0)$ con $k = 0{,}01$ y para los dos más grandes también con $k = 0{,}005$, y para el modelo Asian Dragon, de más de siete millones de polígonos, hay que reducir $K(L_0)$ salvo con $k = 0{,}0001$. Recordemos que esta reducción se realiza dividiendo por dos el valor de $K(0)$ hasta que sea menor que el número de planos potencialmente seleccionables. De la misma manera, en la figura 3.29 se observa un comportamiento similar para el nivel 1, siendo incluso necesario reducir $K(L_1)$ también para $k = 0{,}005$ en todos los modelos.

	Happy		Blade		Dragon	
k raíz	0.01	0.0005	0.01	0.0005	0.01	0.0005
Segundos	1573,40	160,82	1565,60	315,91	8614,58	1599,98
Planos por nivel						
Nivel 0	4816	501	2431	587	3.970	1.777
Nivel 1	10.017	1.585	8.706	2.342	10.293	4.374
Nivel 2	21.006	3.972	19.198	5.808	33.266	12.645
Nivel 3	47.276	10.989	49.381	16.072	88.022	31.279
Nivel 4	105.454	28.720	126.592	45.474	192.631	71.103
Nivel 5	229.782	72.470	324.505	130.854	429.974	167.548
Nivel 6	462.378	158.779	765.858	337.321	938.582	397.750
Nivel 7	867.293	296.711	1.646.645	717.515	2.015.802	946.210
Nivel 8	904.894	287.321	1.318.353	605.366	4.197.030	2.176.897
Nivel 9	94.281	22.658	6.984	2.046	7.804.535	4.222.341

Tabla 3.4: *Influencia del parámetro k.*



a. *Nivel 0* $k = 0{,}01$ **b.** *Nivel 1* $k = 0{,}01$ **c.** *Nivel 2* $k = 0{,}01$ **d.** *Nivel 3* $k = 0{,}01$

e. *Nivel 0* $k = 0{,}0005$ **f.** *Nivel 1* $k = 0{,}0005$ **g.** *Nivel 2* $k = 0{,}0005$ **h.** *Nivel 3* $k = 0{,}0005$

Figura 3.26: *Aspecto visual del BP-Octree para el modelo Blade con* $k = 0{,}01$ *y* $k = 0{,}0005$.

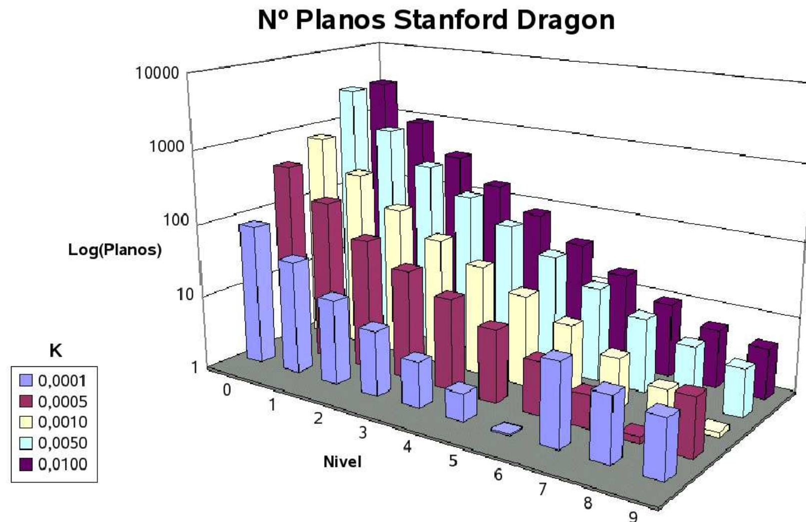


Figura 3.27: *Número medio de planos por nodo para el modelo Stanford Dragon*

	k=0.01		k=0.005		k=0.001		k=0.0005		k=0.0001	
i	Avg	$K(L_i)$	Avg	$K(L_i)$	Avg	$K(L_i)$	Avg	$K(L_i)$	Avg	$K(L_i)$
0	2955,00	8714,14	2880,00	4357,07	746,00	871,41	394,00	435,70	81,00	87,14
1	910,88	4357,07	888,25	2178,54	278,38	435,71	155,50	217,85	34,75	43,57
2	346,66	2178,54	329,83	1089,27	111,15	217,85	60,13	108,93	14,09	21,79
3	161,90	1089,27	153,30	544,63	54,91	108,93	30,70	54,46	7,44	10,89
4	78,09	544,63	76,44	272,32	30,95	54,46	17,44	27,23	4,19	5,45
5	37,86	272,32	37,57	136,16	17,03	27,23	9,47	13,62	2,33	2,72
6	18,66	136,16	18,64	68,08	9,37	13,62	5,10	6,81	1,04	1,36
7	9,89	68,08	9,88	34,04	4,91	6,81	2,79	3,40	12,72	0,68
8	5,98	34,04	5,97	17,02	2,73	3,40	1,23	1,70	7,32	0,34
9	4,75	17,02	4,35	8,51	1,18	1,70	5,87	0,85	5,87	0,17

Tabla 3.5: *Media de planos por nodo para el modelo Stanford Dragon. En las columnas $K(L_i)$ se refleja el máximo de planos para ese nivel en función del valor inicial de k.*

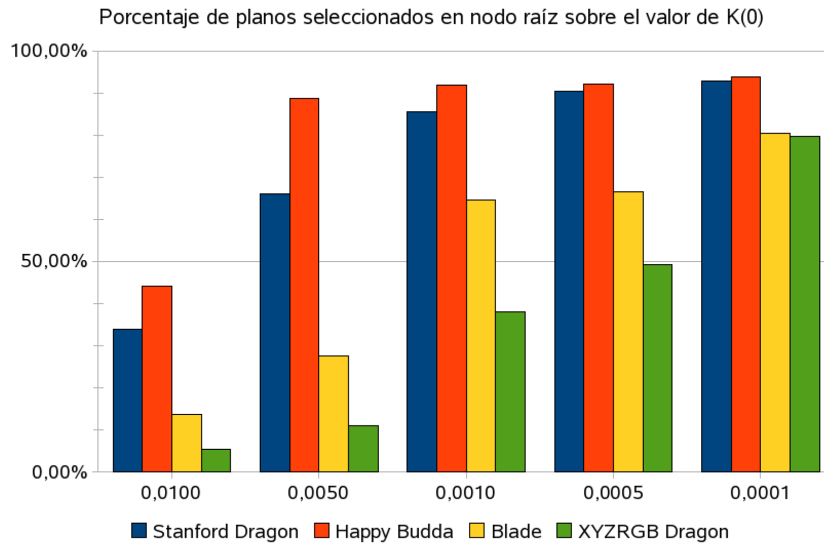


Figura 3.28: *Porcentaje de planos seleccionados sobre el máximo permitido en el nodo raíz.*

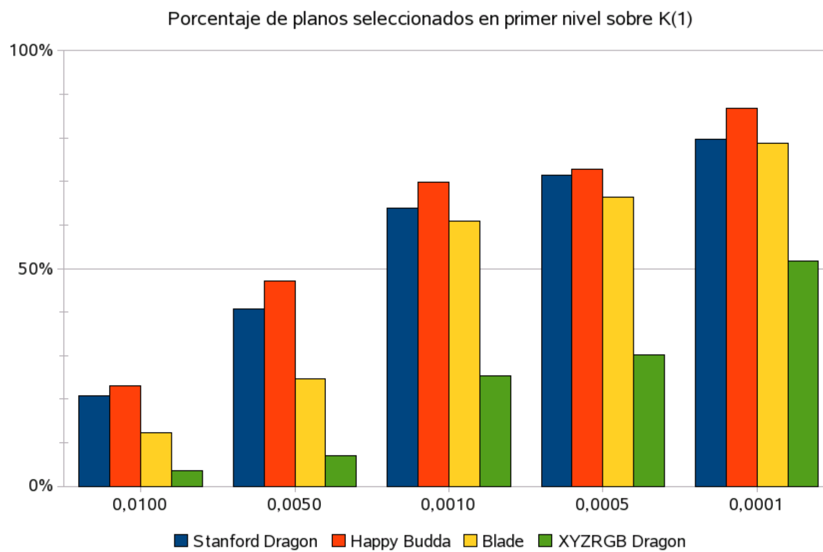


Figura 3.29: *Porcentaje de planos seleccionados sobre el máximo permitido por nodo (nivel 1).*

Obviamente, estos datos no indican nada acerca de la calidad de la envolvente generada, sino que se centran en mostrar la influencia de K sobre el tiempo de cálculo del BP-Octree generado. Una medida de la calidad de la envolvente es el volumen cubierto por la misma, lo que mostramos gráficamente en las figuras 3.30,3.31 y 3.32 y numéricamente en las tablas 3.6,3.7 y 3.8.

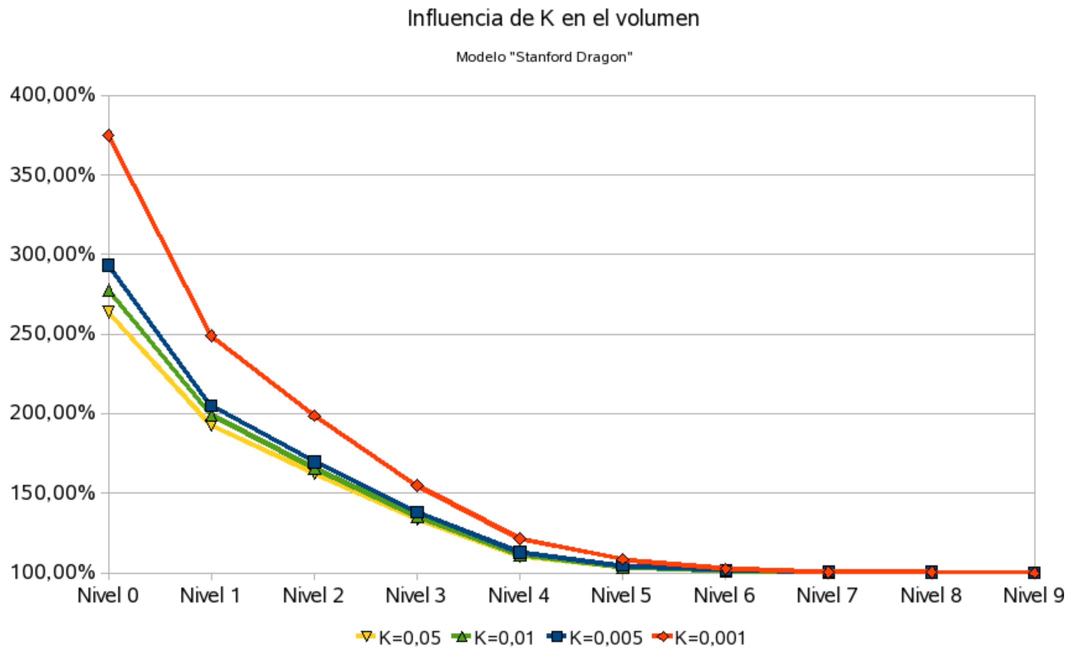


Figura 3.30: *Influencia del parámetro K en el volumen del modelo Stanford Dragon.*

	K=0,001	K=0,005	K=0,01	K=0,05
Level 0	374,83 %	293,14 %	277,82 %	263,65 %
Level 1	248,80 %	204,83 %	199,08 %	192,56 %
Level 2	198,60 %	169,79 %	165,60 %	162,63 %
Level 3	154,78 %	138,12 %	135,50 %	133,70 %
Level 4	121,45 %	112,97 %	111,44 %	110,62 %
Level 5	108,49 %	104,53 %	103,68 %	103,27 %
Level 6	102,55 %	101,74 %	101,30 %	101,11 %
Level 7	100,49 %	100,70 %	100,50 %	100,41 %
Level 8	100,36 %	100,41 %	100,32 %	100,27 %
Level 9	100,00 %	100,00 %	100,00 %	100,00 %

Tabla 3.6: *Volumen del BP-Octree para el modelo Stanford Dragon en función de k*

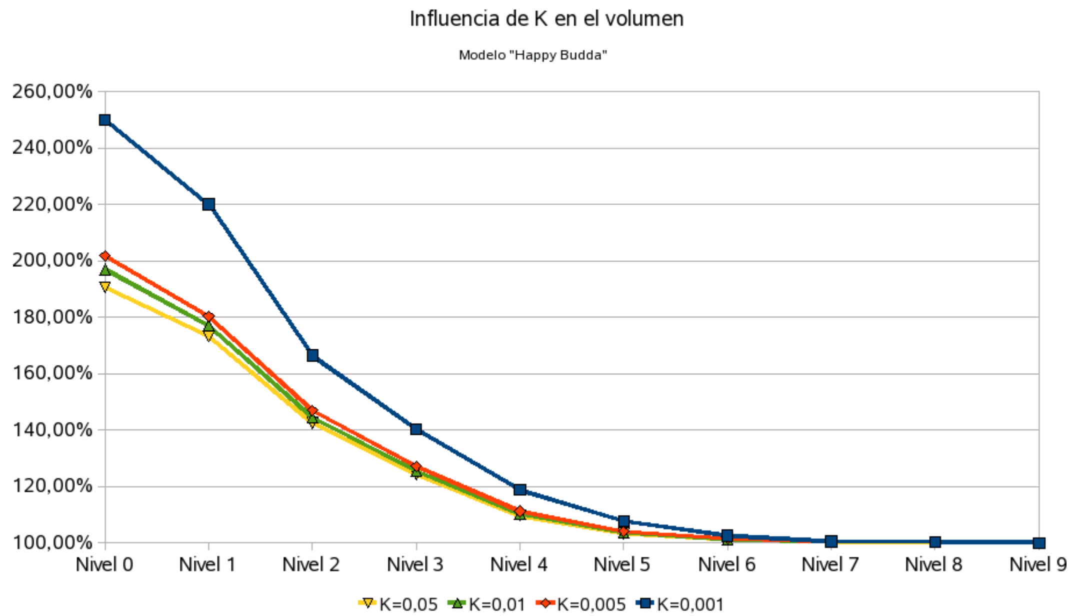


Figura 3.31: *Influencia del parámetro K en el volumen del modelo Happy Budda.*

	K=0,001	K=0,005	K=0,01	K=0,05
Level 0	249,85 %	201,72 %	196,91 %	190,62 %
Level 1	219,96 %	180,29 %	177,06 %	173,27 %
Level 2	166,38 %	146,87 %	144,42 %	142,51 %
Level 3	140,28 %	127,08 %	125,48 %	124,19 %
Level 4	118,76 %	111,15 %	110,23 %	109,67 %
Level 5	107,71 %	104,04 %	103,58 %	103,33 %
Level 6	102,62 %	101,49 %	101,27 %	101,16 %
Level 7	100,49 %	100,55 %	100,44 %	100,39 %
Level 8	100,34 %	100,32 %	100,26 %	100,23 %
Level 9	100,00 %	100,00 %	100,00 %	100,00 %

Tabla 3.7: *Volumen del BP-Octree para el modelo Happy Budda en función de k*

En la figura 3.31 se observa como tan sólo el valor $k = 0,001$ supone un aumento significativo del volumen en el nodo raíz, en torno al 20 %, disminuyendo este diferencial conforme profundizamos en el árbol.

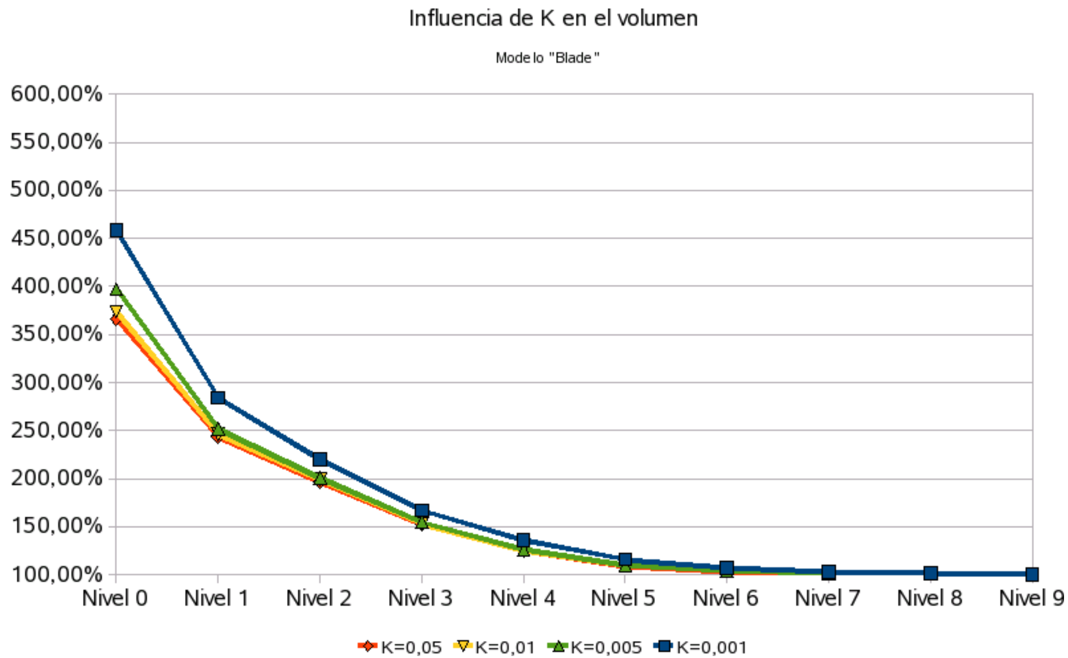


Figura 3.32: *Influencia del parámetro K en el volumen del modelo Blade.*

	K=0,001	K=0,005	K=0,01	K=0,05
Level 0	458,38 %	397,54 %	373,79 %	366,10 %
Level 1	284,13 %	251,52 %	246,02 %	243,31 %
Level 2	220,11 %	200,44 %	199,07 %	196,70 %
Level 3	166,98 %	154,66 %	152,86 %	152,19 %
Level 4	135,86 %	126,42 %	125,31 %	124,85 %
Level 5	115,74 %	109,65 %	108,97 %	108,70 %
Level 6	107,01 %	103,79 %	103,40 %	103,28 %
Level 7	102,61 %	101,57 %	101,37 %	101,31 %
Level 8	101,57 %	101,03 %	100,90 %	100,87 %
Level 9	100,00 %	100,00 %	100,00 %	100,00 %

Tabla 3.8: *Volumen del BP-Octree para el modelo Blade en función de k*

Independientemente del estudio realizado en esta sección, todos los datos mostrados en la presente memoria, salvo que se indique lo contrario, han sido tomados con $k = 0{,}01$.

3.6. Almacenamiento en memoria externa.

Una vez construido el árbol, con sus envolventes en nodos grises y hoja, con los nodos negros y blancos, y la geometría repartida por las hojas, el almacenamiento en memoria externa es sencillo. Se distribuye la información en tres archivos:

1. Archivo *.bpo*. Indica la ubicación de los archivos con los planos y los nodos, así como el modelo original.

2. Archivo *.bpo.bpl*. Almacena los planos utilizados en las envolventes, estando ordenados según su aparición en el recorrido en anchura del árbol.

3. Archivo *.bpo.bpn*. Almacena la estructura jerárquica siguiendo el orden indicado por los octcodes. Además, almacena al principio del mismo las coordenadas de la caja envolvente del modelo. La secuencia que encontramos se detalla a continuación.

   - Una cabecera, con la información necesaria para interpretar el árbol:

     ```
     xmin ymin zmin xmax yax zmax
     typecodeforBlack (1 byte)
     typecodeforWhite (1 byte)
     typecodeforLeaf  (1 byte)
     typecodeforGray  (1 byte)
     ```

   - Los nodos ordenados por su octcode. En función del tipo de nodo, encontramos una información distinta:

     a) Si el nodo es *blanco* o *negro* tan sólo se almacena su tipo (1 byte)

     b) Si el nodo es *gris* se almacena su tipo, la configuración de las esquinas, el número de planos que contiene, y la lista con los pares $< indicedeplano, offset >$. Si el índice de plano es negativo, indica que hay offset. Si es positivo, se omite:

        ```
        typecodeforGray  (1 byte)
        nrOfPlanes       (2 byte)
        planeIndex1      # <0
        offset1
        planeIndex2      # >0
        planeIndex3      # <0
        offset3
        ..
        planeIndexN      # >0
        ```

     c) Si el nodo es *hoja*, además de su tipo, número de planos y lista con los pares $< indicedeplano, offset >$ como en el nodo *gris*, se incluye la configuración de las esquinas, el número de polígonos que contiene y sus índices.:

```
typecodeforLeaf  (1 byte)
cornerConfig     (1 byte)
nrOfPlanes       (2 byte)
planeIndex1      # <0
offset1
planeIndex2      # >0
planeIndex3      # <0
offset3
..
planeIndexN      # >0
nrOfFaces       (2 byte)
faceIndex1
faceIndex2
..
faceIndexM
```

La información relativa a las esquinas se almacena en un bitset de tamaño 8, indicando en el bit i si la esquina i-ésima está *dentro* (true) o *fuera* (false).

Como se puede apreciar, se almacena en disco el mínimo de información necesaria, prescindiendo del valor del octcode, ya que éste puede ser calculado conforme vamos creando los nodos.

Todos los datos son almacenados en formato binario, y en la tabla 3.9 podemos consultar el espacio en disco necesario para cada uno de los modelos evaluados. En general, el modelo BPO supone una sobrecarga de entre 2 y 3 veces el espacio en disco necesario para el modelo poligonal.

Modelo	.bpn	.bpl	.ply	Ratio
Sphere	136,5	63,5	16,0	12,5
Stanford Bunny	2.038,5	908,6	1.331,4	2,60
Golf	3.097,1	1.307,1	1.765,4	2.49
Armadillo	4.774,1	2.017,5	2.636,7	2,57
Aegyptian	5.377,1	2.082,7	2.846,3	2.62
Teeth	6.929,1	2.929,7	4.099,3	2,40
Fertility	15.514,2	6.064,7	8.494,1	2,43
Angelo	21.523,2	9.014,6	12.520,2	2.57
Phleg. Dragon	21.903,3	9.442,2	13.298,9	2,35
Stanford Dragon	25.126,4	10.562,8	16.191,5	2.20
Happy Buddha	31.544,6	13.093,4	20.180,0	2,21
Blade	50.043,2	22.131,3	32.759,5	2,20
Asian Dragon	327.866,5	85.395,0	126.894,9	3,25

Tabla 3.9: *KBytes necesarios para almacenar el BP-Octree relativo a cada modelo, y ratio (.bpn + .bpl)/.ply.*

El algoritmo de lectura del fichero hace uso de una cola para realizar el recorrido en anchura, tal y como se describe en el pseudocódigo 7. Se cargan los planos y la malla de

los ficheros auxiliares. A continuación se lee la cabecera del archivo *bpn* (caja envolvente y valores constantes para identificar los tipos de nodos). Se generan los índices para los nodos blancos y negros y se utiliza una cola auxiliar para ir introduciendo los nodos leidos, de forma que se siga el recorrido en anchura que dispone el fichero.

```
load(FILE nodesF,FILE planesF,FILE meshF,BpOctree bpo) {

  loadPlanes(planesF);
  readMeshFile(meshF);

  char type; Index idx; Coord bboxLimits[6];

  nodesF.read((char*)bboxLimits,sizeof(bboxLimits[0])*6);
  bpo->bbox=BBox(bboxLimits);

  nodesF.read((char *)&WHITENODE, sizeof(char));
  nodesF.read((char *)&BLACKNODE, sizeof(char));
  nodesF.read((char *)&LEAFNODE, sizeof(char));
  nodesF.read((char *)&GRAYNODE, sizeof(char));

  __BLACK=newLeaf()->code();
  __WHITE=newLeaf()->code();

  queue<Index> nodesq;

  nodesFile.read((char *)&type,sizeof(type));
  BPONode *root=newNode(bpo);
  root->load(nodesFile);
  root->setCode(0);
  root->setBBox(bbox);
  nodesq.push(root);

  while (!nodesq.empty() && !nodesFile.eof()) {
      BpoNode *node=nodesq.front();nodesq.pop();
      for (int i=0;i<8;i++) {
          nodesF.read((char *)&type,sizeof(type));
          switch (type) {
            case WHITENODE:    idx=__WHITE; break;
            case BLACKNODE:    idx=__BLACK; break;
            case LEAF:
               BpoLeafNode *leaf=newLeaf(bpo);
               leaf->load(nodesF);
               leaf->setBBox(node->getBBox()->getSubBBox(i));
               leaf->setCode(getChildCode(node->code(),i));
               setChild(node,i,leaf);           break;
            case GRAY:
               BpoNode *graynode=newNode(bpo);
               graynode->load(nodesF);
               graynode->setBBox(node->getBBox()->.getSubBBox(i));
               graynode->setCode(getChildCode(node->code(),i));
               nodesq.push(graynode);
               setChild(node,i,graynode);             break;
          }
      }
  }
  closeAllFiles();
}
```

Algoritmo 7: *Carga de un BP-Octree de disco*

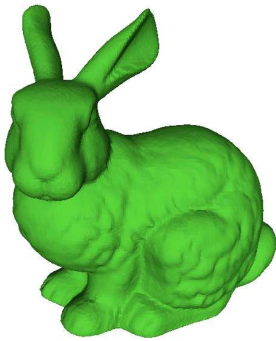


Figura 3.33: *Stanford bunny (71.040 polígonos)*

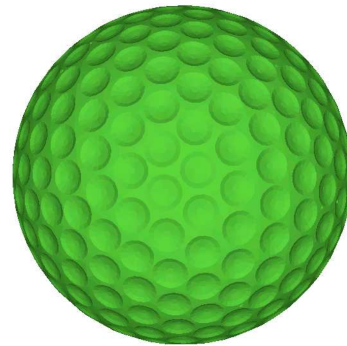


Figura 3.34: *Golf (100.243 polígonos)*

Figura 3.35: *Armadillo (150.000 polígonos)*

Figura 3.36: *Egipcio (161.909 polígonos)*

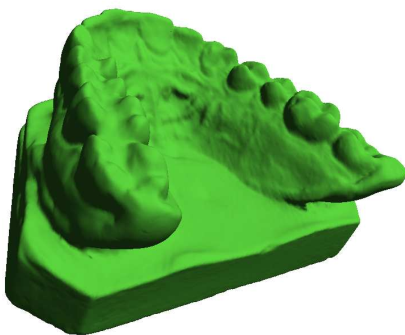


Figura 3.37: *Teeth (233.204 polígonos)*

Figura 3.38: *Fertility (483.226 polígonos)*

Figura 3.39: *Angelo (674.764 polígonos)*

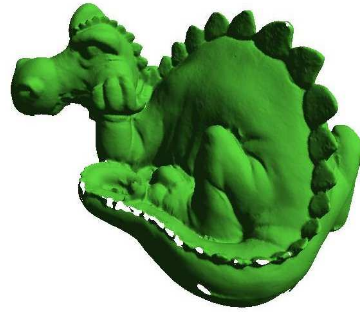


Figura 3.40: *Phlegmatic Dragon (715.933 polígonos)*

Figura 3.41: *Stanford Dragon (871.414 polígonos)*

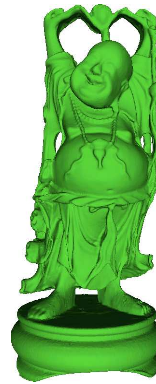


Figura 3.42: *Happy Buddha (1.087.716 polígonos)*

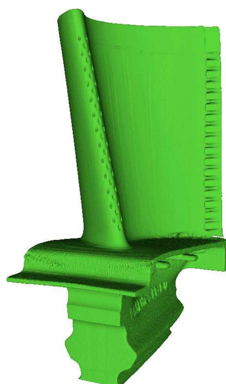


Figura 3.43: *Blade (1.765.388 polígonos)*

Figura 3.44: *Asian Dragon (7.218.906 polígonos)*

CAPÍTULO 4

BP-Octree como estructura multiresolución

La visualización interactiva de modelos complejos requiere un gran esfuerzo del hardware gráfico para conseguir el nivel de detalle y realismo esperado, y estas espectativas están reñidas con el concepto de interactividad, ya que la frecuencia de refresco ha de superar un umbral de 30fps. Este conflicto entre nivel de detalle y velocidad de visualización ha motivado el desarrollo de diversas técnicas que permitan compaginar ambos objetivos, mostrando el modelo simplificado cuando el usuario no puede apreciar los detalles por su lejanía y suministrando los detalles sólo cuando éstos son necesarios.

Ya en su clásico artículo de 1976, James Clark [Cla76] afirmó que no tenía sentido utilizar 500 polígonos para representarlos en apenas 20 píxeles, y propuso una estructura de datos jerarquíca que manejaba el grafo de escena para no sólo obtener diversos niveles de detalle (*level-of detail*, LOD, en inglés), sino para otras operaciones como la eliminación de partes ocultas.

Desde entonces, se han propuesto diversas técnicas para la visualización eficiente de grandes modelos geométricos. En un primer nivel, se podrían clasificar en: técnicas basadas en la geometría del objeto original –básicamente la simplificacion de mallas de triángulos–, técnicas basadas en la compresión de la geometría –mediante *wavelets* o codificación de polígonos– y técnicas basadas en el uso de imágenes –impostores–.

Las técnicas de nivel de detalle basadas en la geometría pueden a su vez clasificarse en:

- *discretas* [Cla76], cuando se tienen varias instancias del mismo objeto a distintos niveles de detalle;
- *progresivas* [Hop96], si el detalle se va obteniendo de una única estructura de datos, como es el caso de la figura 4.1;
- *dependientes del observador* [Hop97], que son una extensión de las anteriores, de forma que el nivel de detalle no es uniforme en todo el modelo, sino que es anisotrópico dependiendo del punto de vista. En la figura 4.2 se ve cómo la densidad de triángulos aumenta con respecto a la cercanía al observador.

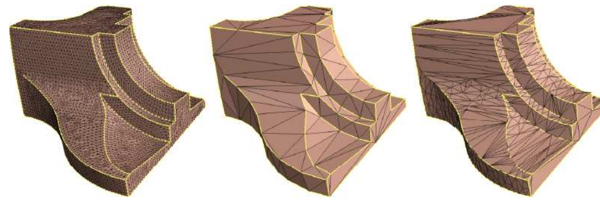


Figura 4.1: *Tres resoluciones de un modelo poligonal obtenidas de una única estructura, las mallas progressivas [Hop96]*

Cuando los modelos son de varios millones de triángulos, se utilizan técnicas con memoria secundaria, denominadas *out-of-core*, de forma que se procesa la malla por partes. Los primeros métodos se basaban en el *troceado de la malla* [Hop98][BRM+02], de forma que cada porción sea de tal tamaño que se puede manejar en memoria principal, simplificando el interior y dejando los bordes intactos para que suelden bien las piezas. Los métodos difieren entre sí en la forma en que se realizan las diversas iteraciones para simplificar a su vez estos bordes. Otras técnicas, denominadas de procesamiento por lotes, acceden a la malla ignorando la topología de ésta, considerándola como una secuencia de triángulos. Entre éstas técnicas podemos encontrar las basadas en la agrupación de vértices haciendo uso de un grid tridimensional y minimizando el error [Lin00]. También se han propuesto el uso de octrees [CMRS03] y otras estructuras de datos, así como la paginación virtual del propio sistema operativo, para la simplificación de mallas triangulares haciendo uso de la información topológica de la malla. En los últimos años han aparecido técnicas que hacen uso de estas tres aproximaciones, combinándolas y realizando tanto el procesamiento por lotes como el aprovechamiento de la informaticón topológica de los modelos [IG03][ILGS03], mediante la extracción de los polígonos de la malla de una forma ordenada y con información sobre el volumen y los rasgos principales de la misma.

4.1. El BP-Octree como estructura multiresolución

El uso de una estructura jérarquica que indexa espacialmente el objeto hace del BP-Octree un evidente candidato a ser utilizado como estructura para una representación multiresolución de grandes modelos poligonales.

Además, el BP-Octree permite su utilización para una visualización por niveles de detalle,

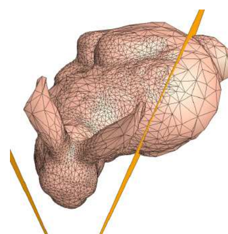


Figura 4.2: *Nivel de detalle dependiente del observador [Hop97].*

```
recursiveRenderBPO(BPOctree bpo, int upToLevel, Octcode node) {
  if (bpo->isLeaf(node))
    bpo->getLeaf(node)->render();
  else
    if (bpo->isGray(node))
      if (level(node)==upToLevel)
        bpo->getNode(node)->render();
      else
        for (int c=0;c<8;c++) {
          child=bpo->getChild(node,c);
          if (bpo->isNode(child))
            recursiveRenderBPO(bpo,upToLevel,child);
        }
}
```

Algoritmo 8: *Dibujado por niveles del BP-Octree*

esto es, dibujando todos los nodos de un determinado nivel del árbol, o bien la visualización adaptativa del mismo, descendiendo por unas ramas u otras hasta alcanzar el nivel de detalle deseado en función de la posición del observador.

Si bien hay otras posibilidades multiresolución que proporcionan resultados visualmente más adecuados, el BP-Octree combina la apariencia visual con la propiedad volumétrica, que hace que en cualquier nivel el volumen del sólido representado sea mayor o igual que el del modelo original, lo que nos permite usar esta misma representación para realizar sobre ella cálculos volumétricos, de inclusión u oclusiones.

4.1.1. Dibujado del BP-Octree

Cada uno de los niveles del BP-Octree es una representación del modelo, con mayor nivel de detalle conforme aumentamos la profundidad. De esta forma, el algoritmo de dibujado del modelo representado puede ser tan sencillo como una simple función recursiva, cuyo parámetro sea el nivel de detalle deseado (pseudocódigo 8).

4.1.2. Dibujado de un nodo

En definitiva, dibujar un BP-Octree es la visualización del volumen definido por la intersección de los semiespacios interiores delimitados por los planos envolventes de cada uno de los nodos.

En concreto, es el algoritmo explicado en el apartado 3.4.3 el que nos proporciona la geometría que nos permite visualizar este volumen. Recordemos que este algoritmo parte de una malla inicial, que es la caja englobante del nodo en cuestión, y le aplica sucesivos cortes con los planos que forman la envolvente, dando como resultado un conjunto de caras poligonales.

Cabe decir que el uso de envolventes convexas, y la ausencia total de informacion entre dichas envolventes en nodos adyacentes hace que la superficie que se visualiza no sea continua, como se aprecia en la figura 4.3. Estos huecos pueden ser tapados de una forma trivial dibujando la parte relativa a la cara del nodo, puesto que son polígonos existentes e identificados tras la aplicación del algoritmo 6. Obviamente, esto supone una sobrecarga en la visualización del modelo, pero es una primera aproximación que no requiere tiempo de cómputo extra. En

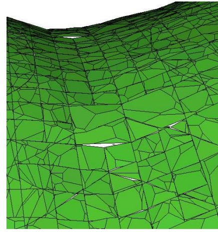
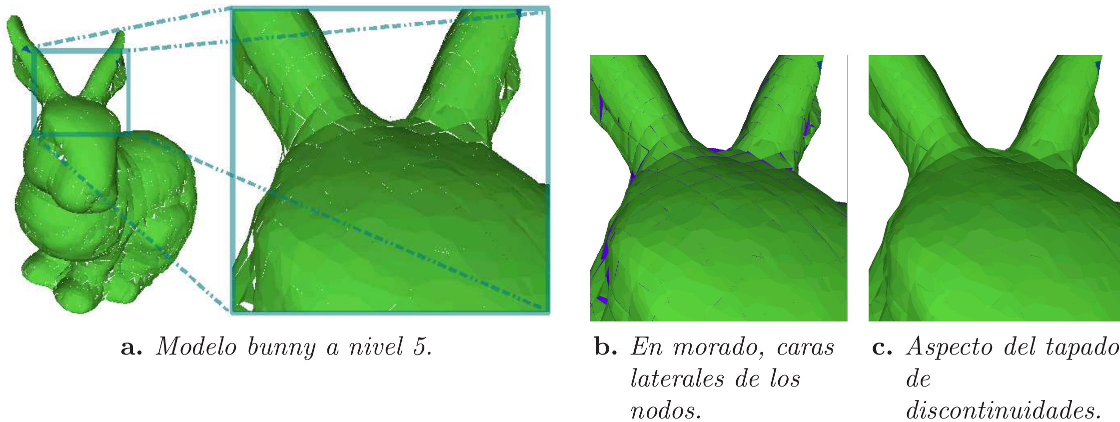


Figura 4.3: *Huecos entre nodos adyacentes.*

la figura 4.4b se muestran dichos polígonos *virtuales* en morado, y en la figura 4.4c se observa su impacto visual.



a. *Modelo bunny a nivel 5.*

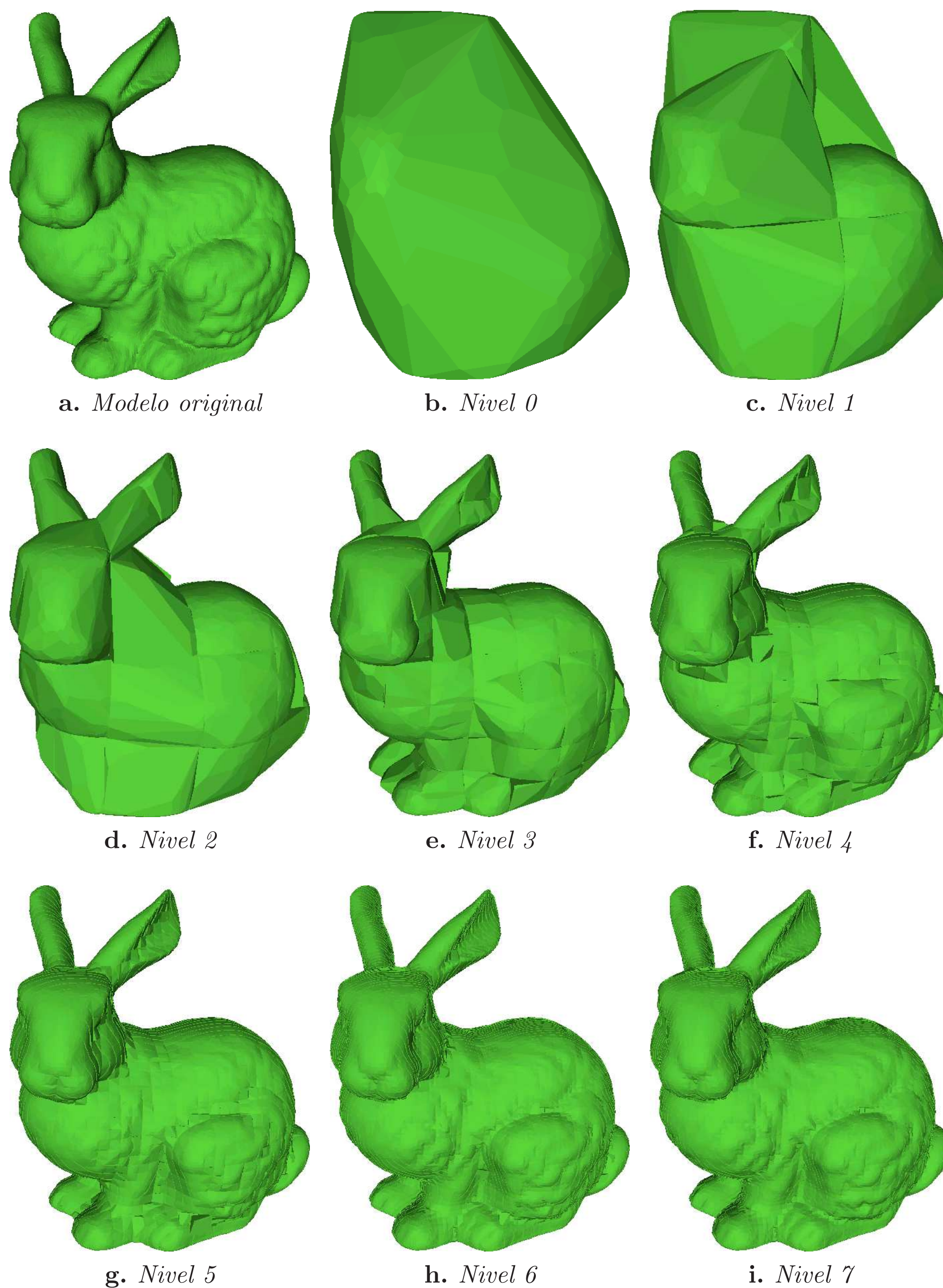
b. *En morado, caras laterales de los nodos.*

c. *Aspecto del tapado de discontinuidades.*

Figura 4.4: *Agujeros en las paredes de los nodos por la no coincidencia de convexidades en nodos adyacentes.*

4.1.3. Resultados

En las figuras 4.5 a 4.15 mostramos los distintos niveles de detalle para diversos modelos usados en las pruebas del presente trabajo. Las figuras han sido tomadas con el tapado de agujeros activado.



a. *Modelo original* **b.** *Nivel 0* **c.** *Nivel 1*

d. *Nivel 2* **e.** *Nivel 3* **f.** *Nivel 4*

g. *Nivel 5* **h.** *Nivel 6* **i.** *Nivel 7*

Figura 4.5: *Modelo Stanford Bunny multirresolución.*

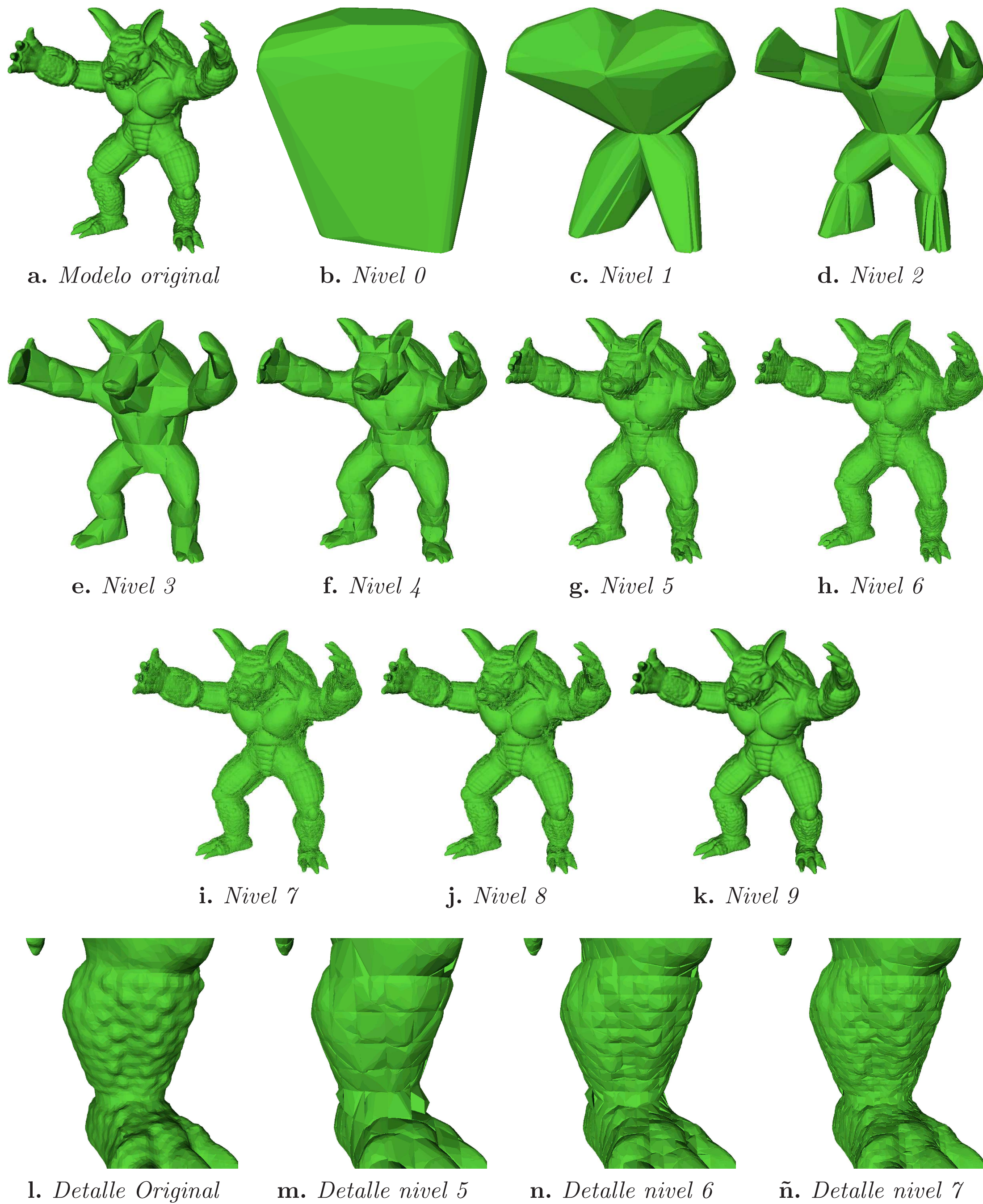
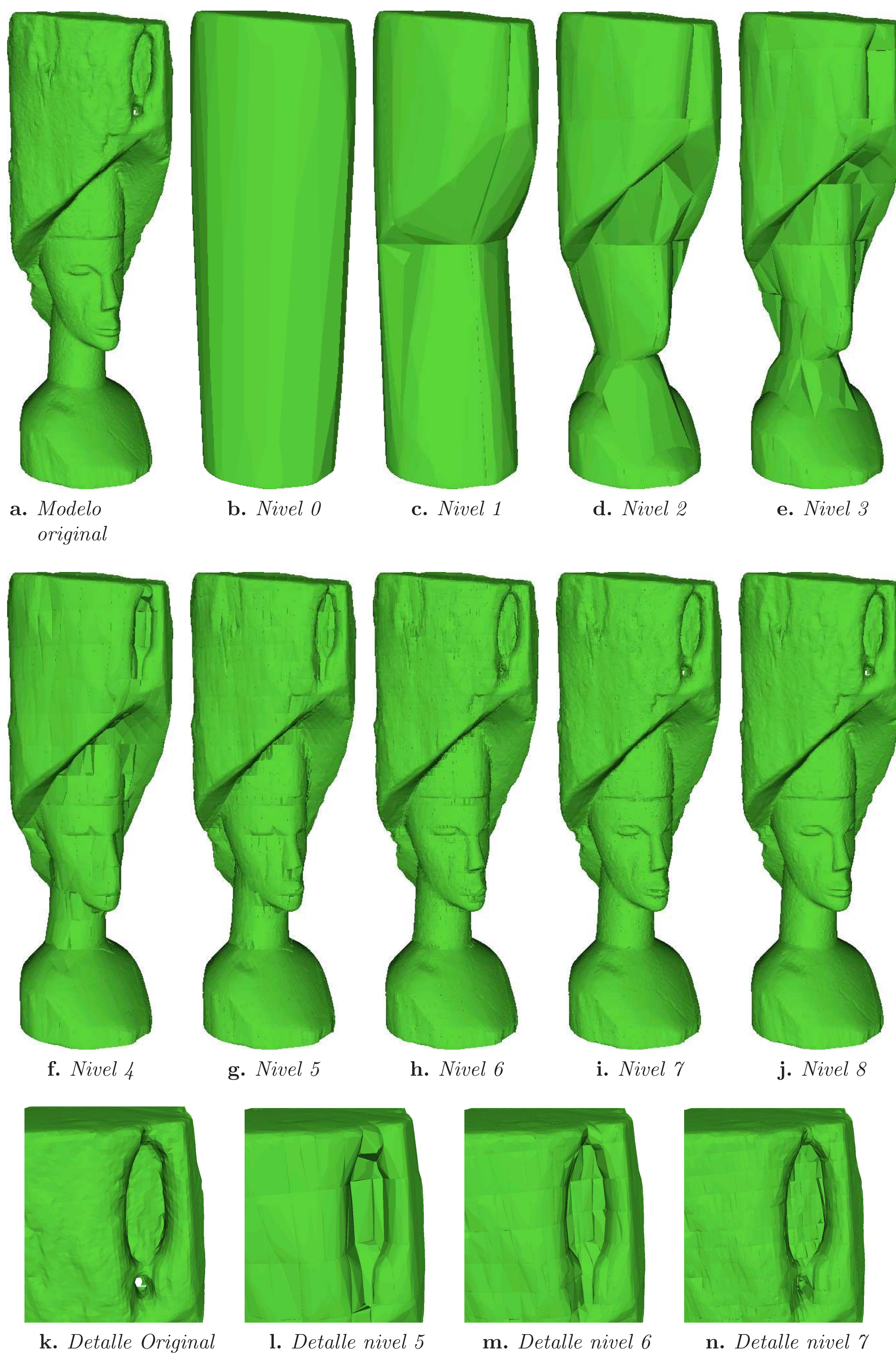


Figura 4.6: *Modelo Armadillo multirresolución.*

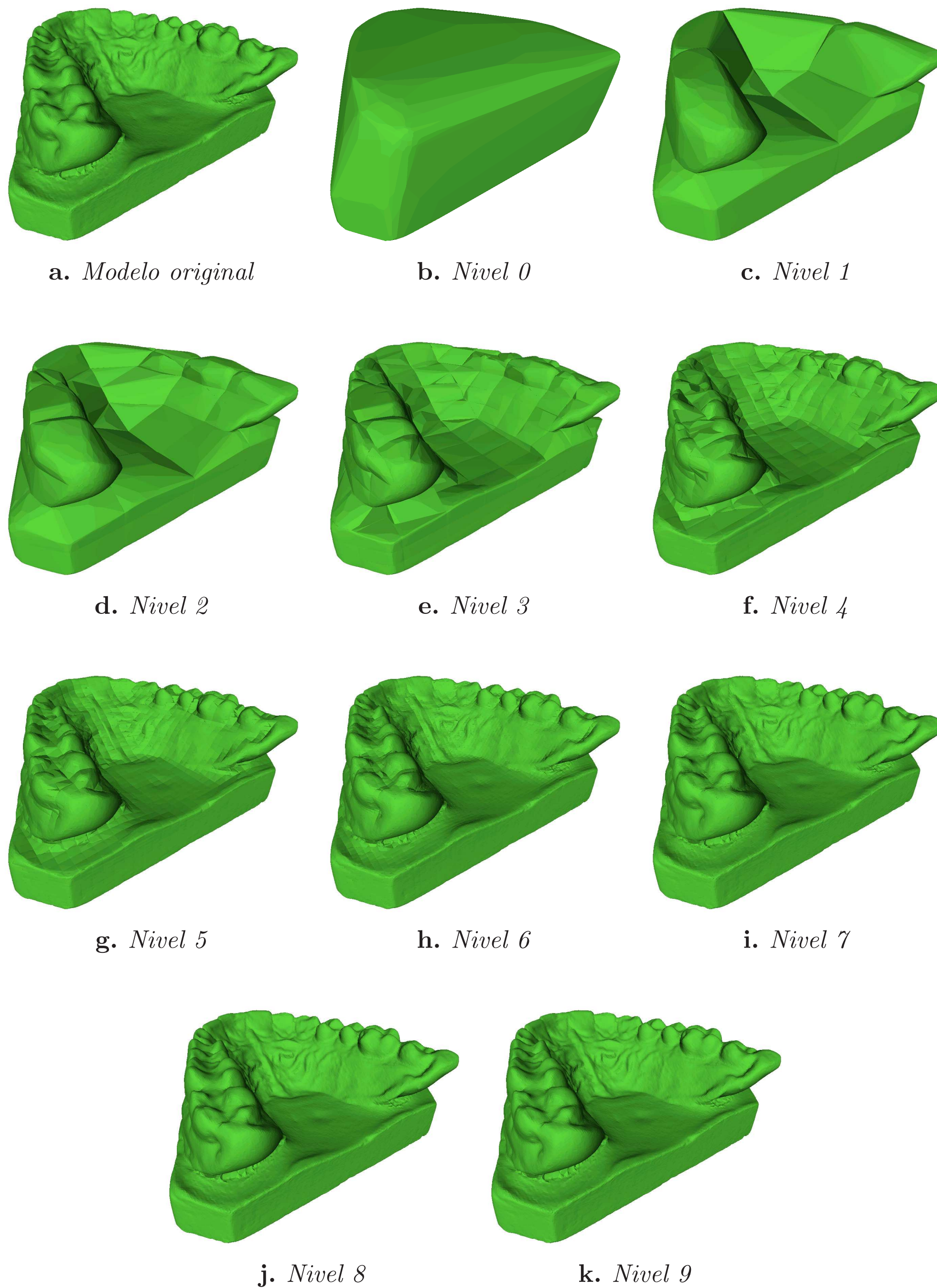


a. *Modelo original* b. *Nivel 0* c. *Nivel 1* d. *Nivel 2* e. *Nivel 3*

f. *Nivel 4* g. *Nivel 5* h. *Nivel 6* i. *Nivel 7* j. *Nivel 8*

k. *Detalle Original* l. *Detalle nivel 5* m. *Detalle nivel 6* n. *Detalle nivel 7*

Figura 4.7: *Modelo Egipcio multirresolución.*



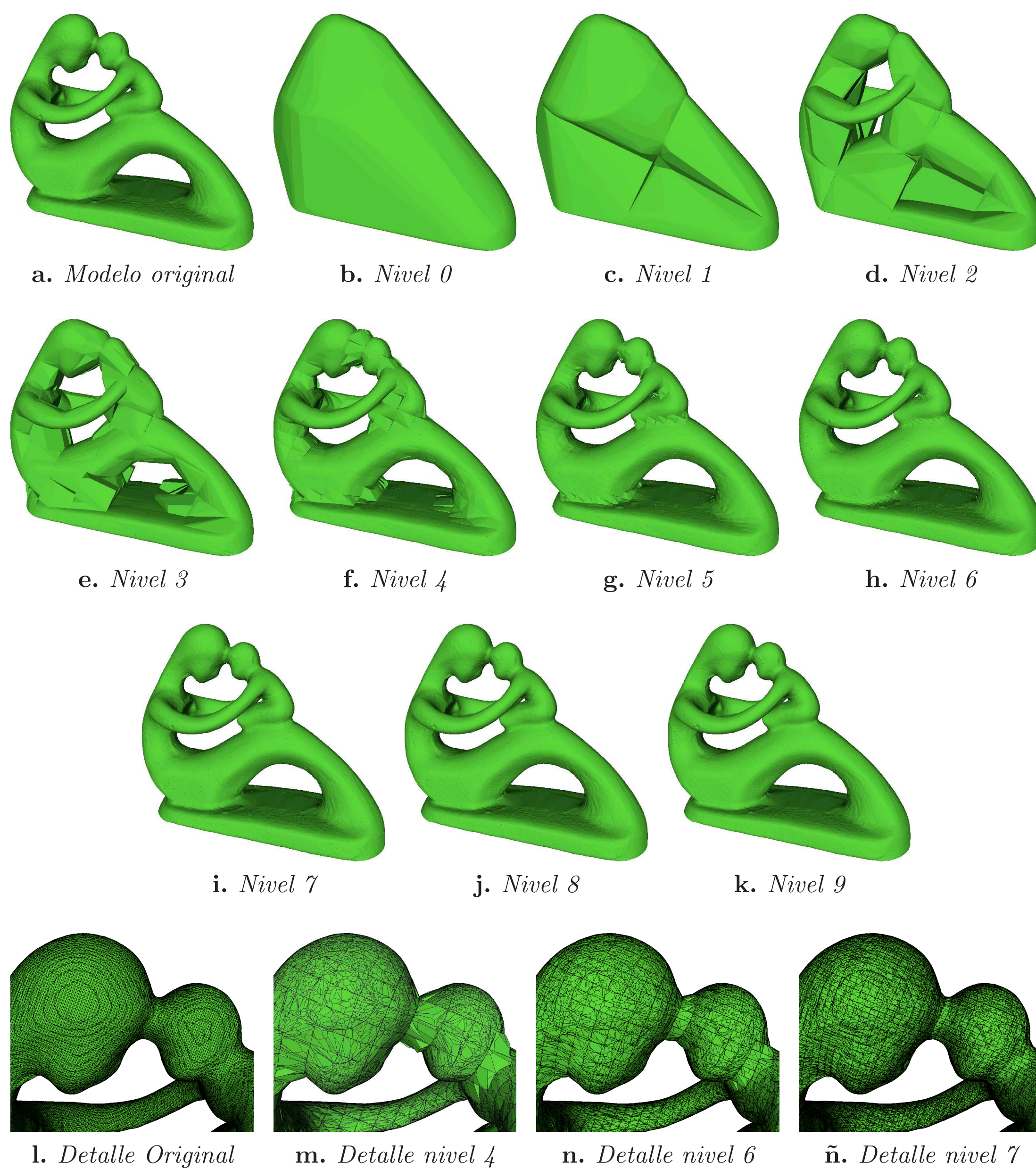
a. *Modelo original* **b.** *Nivel 0* **c.** *Nivel 1*

d. *Nivel 2* **e.** *Nivel 3* **f.** *Nivel 4*

g. *Nivel 5* **h.** *Nivel 6* **i.** *Nivel 7*

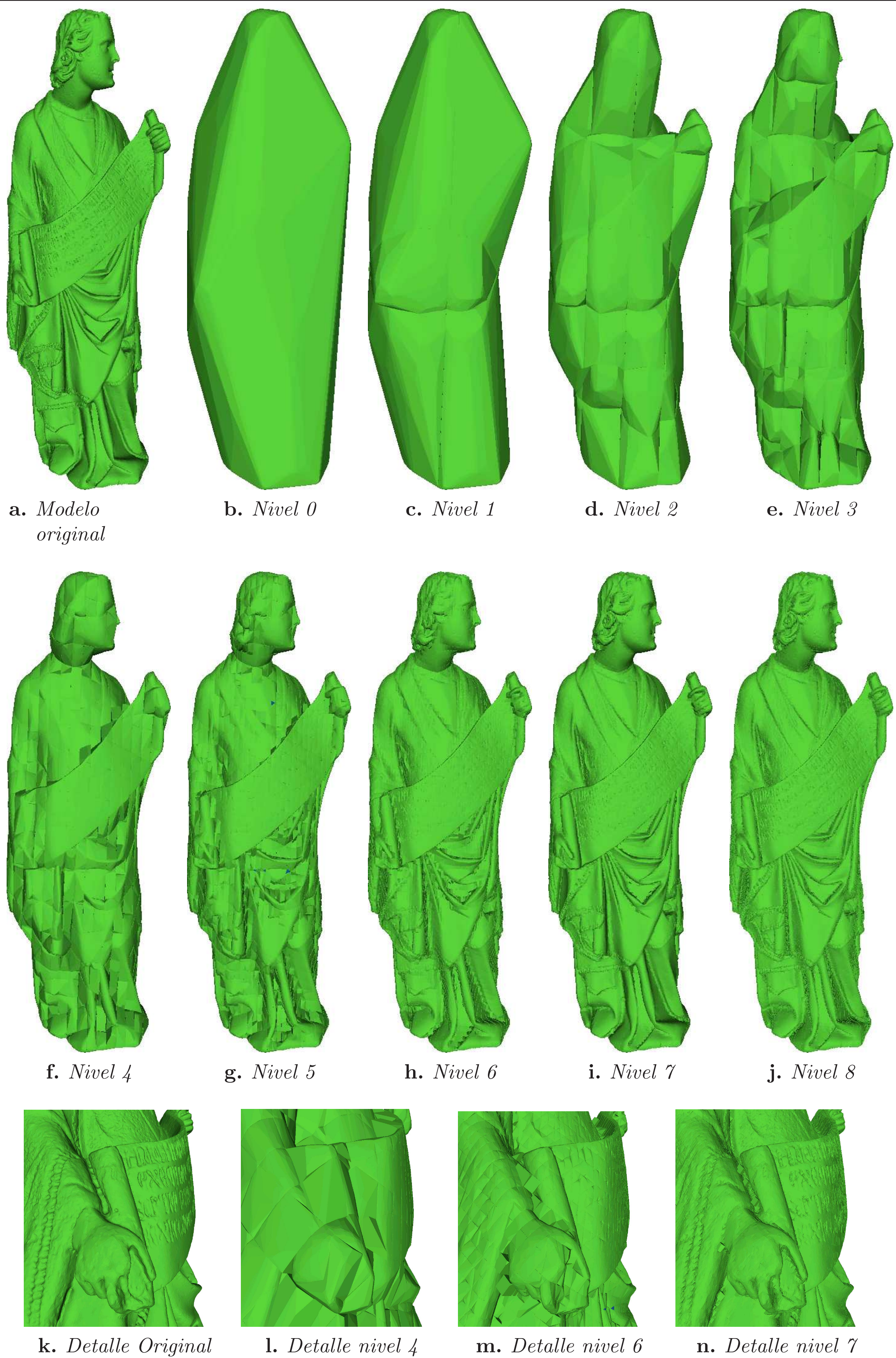
j. *Nivel 8* **k.** *Nivel 9*

Figura 4.8: *Modelo Teeth multirresolución.*



a. *Modelo original* b. *Nivel 0* c. *Nivel 1* d. *Nivel 2*
e. *Nivel 3* f. *Nivel 4* g. *Nivel 5* h. *Nivel 6*
i. *Nivel 7* j. *Nivel 8* k. *Nivel 9*
l. *Detalle Original* m. *Detalle nivel 4* n. *Detalle nivel 6* ñ. *Detalle nivel 7*

Figura 4.9: *Modelo Fertility multirresolución.*



a. *Modelo original* **b.** *Nivel 0* **c.** *Nivel 1* **d.** *Nivel 2* **e.** *Nivel 3*

f. *Nivel 4* **g.** *Nivel 5* **h.** *Nivel 6* **i.** *Nivel 7* **j.** *Nivel 8*

k. *Detalle Original* **l.** *Detalle nivel 4* **m.** *Detalle nivel 6* **n.** *Detalle nivel 7*

Figura 4.10: *Modelo Angelo multirresolución.*

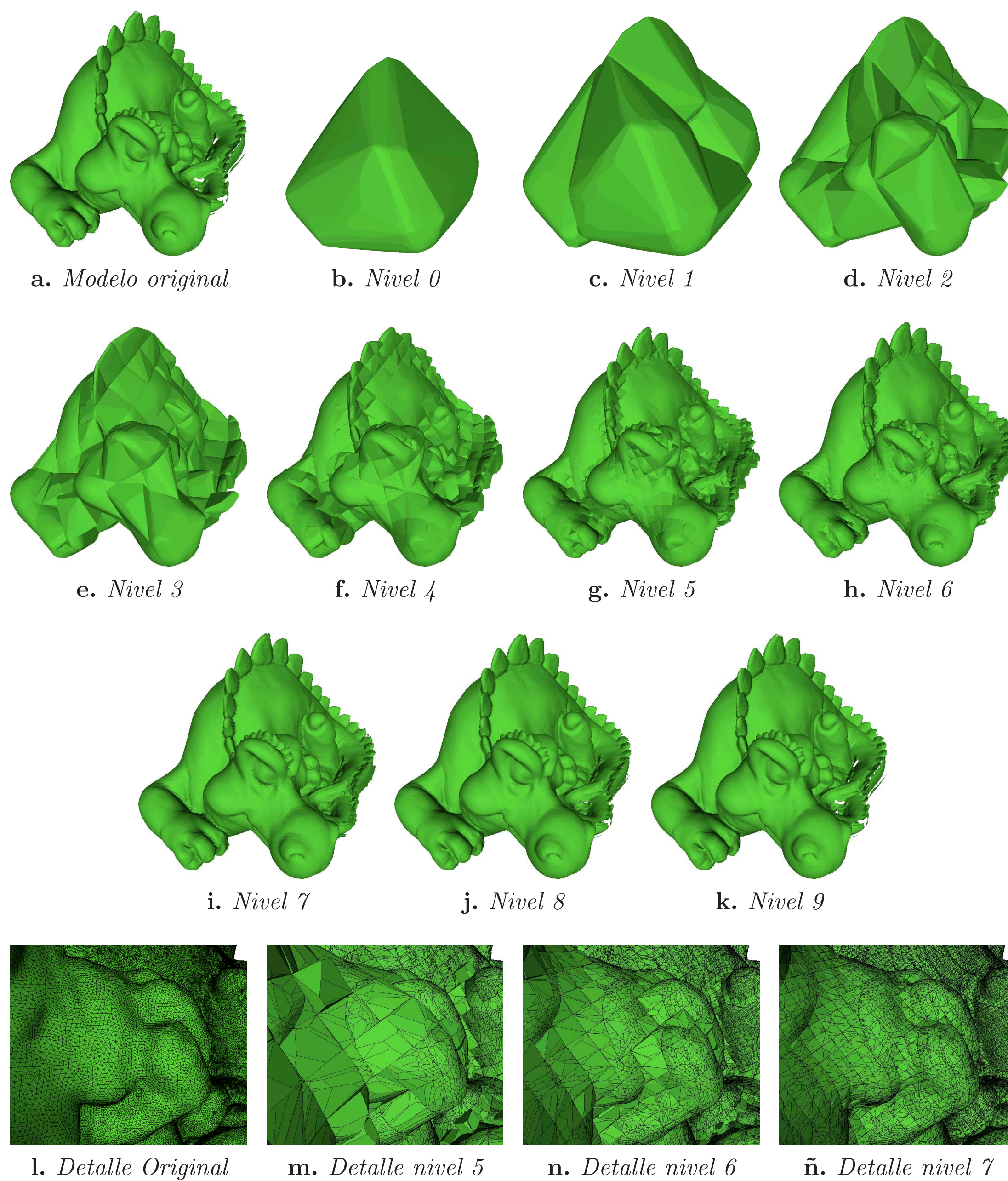


Figura 4.11: *Modelo Phlegmatic Dragon multirresolución.*

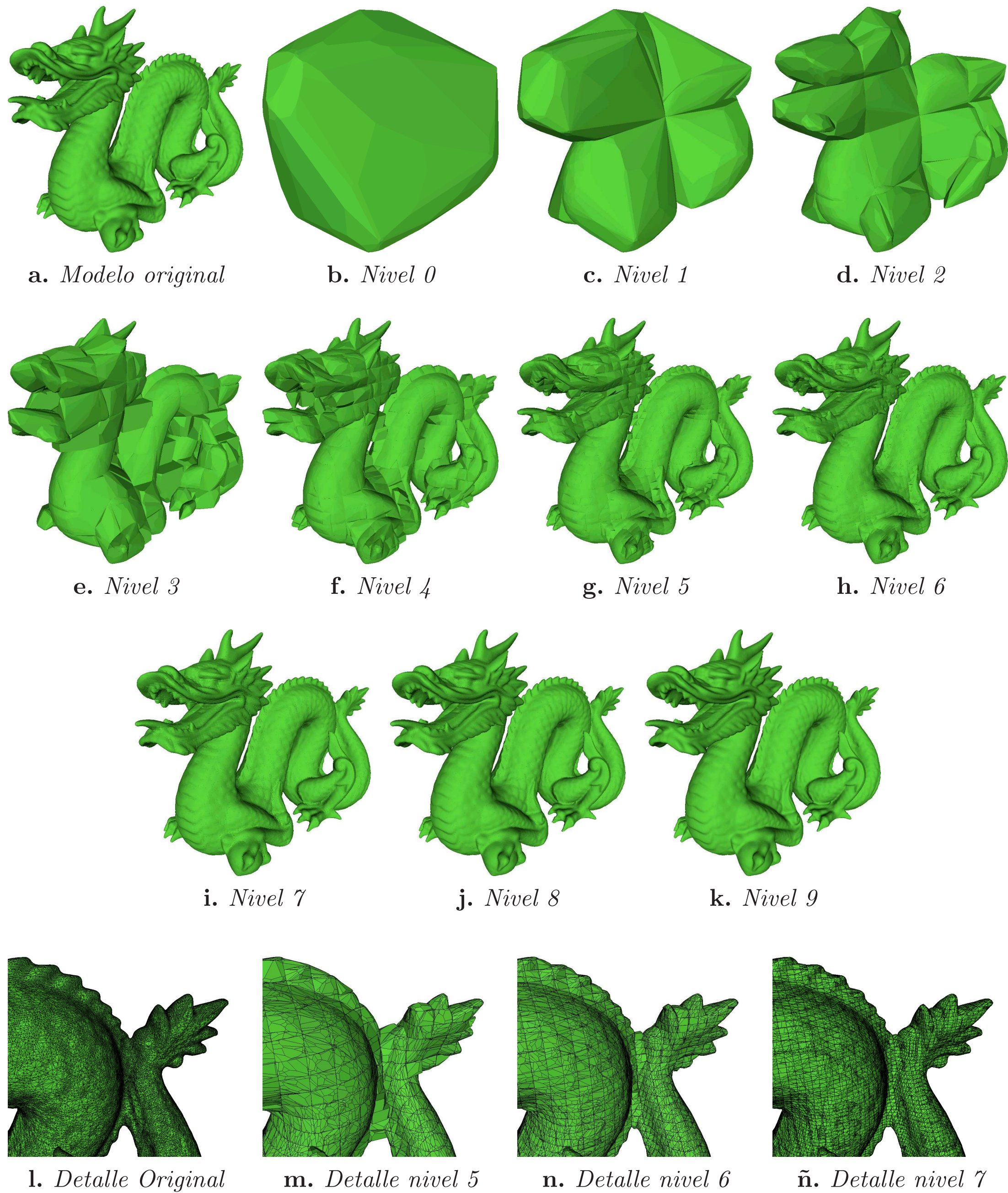
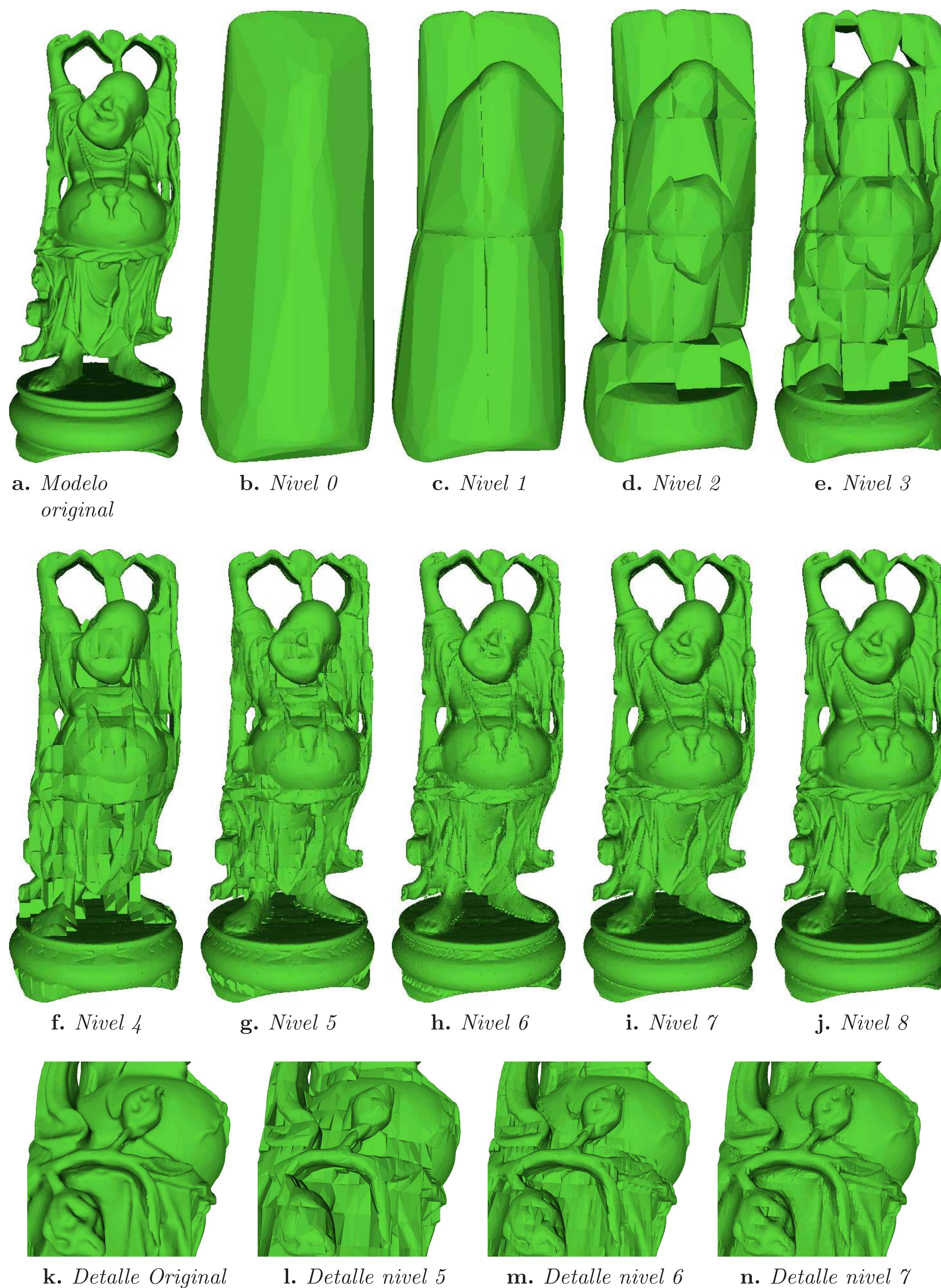


Figura 4.12: *Modelo Stanford Dragon multirresolución.*

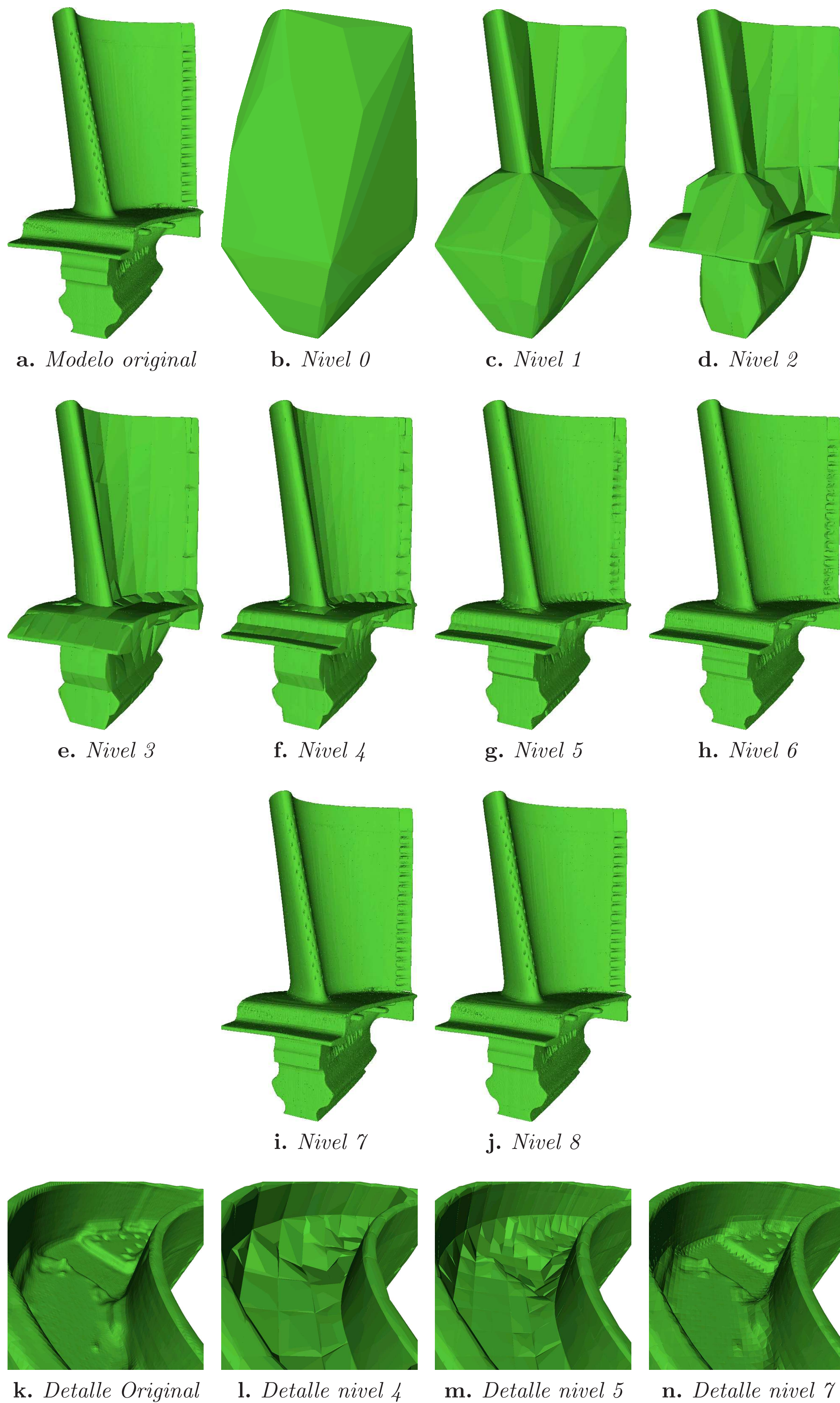


a. *Modelo original* **b.** *Nivel 0* **c.** *Nivel 1* **d.** *Nivel 2* **e.** *Nivel 3*

f. *Nivel 4* **g.** *Nivel 5* **h.** *Nivel 6* **i.** *Nivel 7* **j.** *Nivel 8*

k. *Detalle Original* **l.** *Detalle nivel 5* **m.** *Detalle nivel 6* **n.** *Detalle nivel 7*

Figura 4.13: *Modelo Happy Budda multirresolución.*



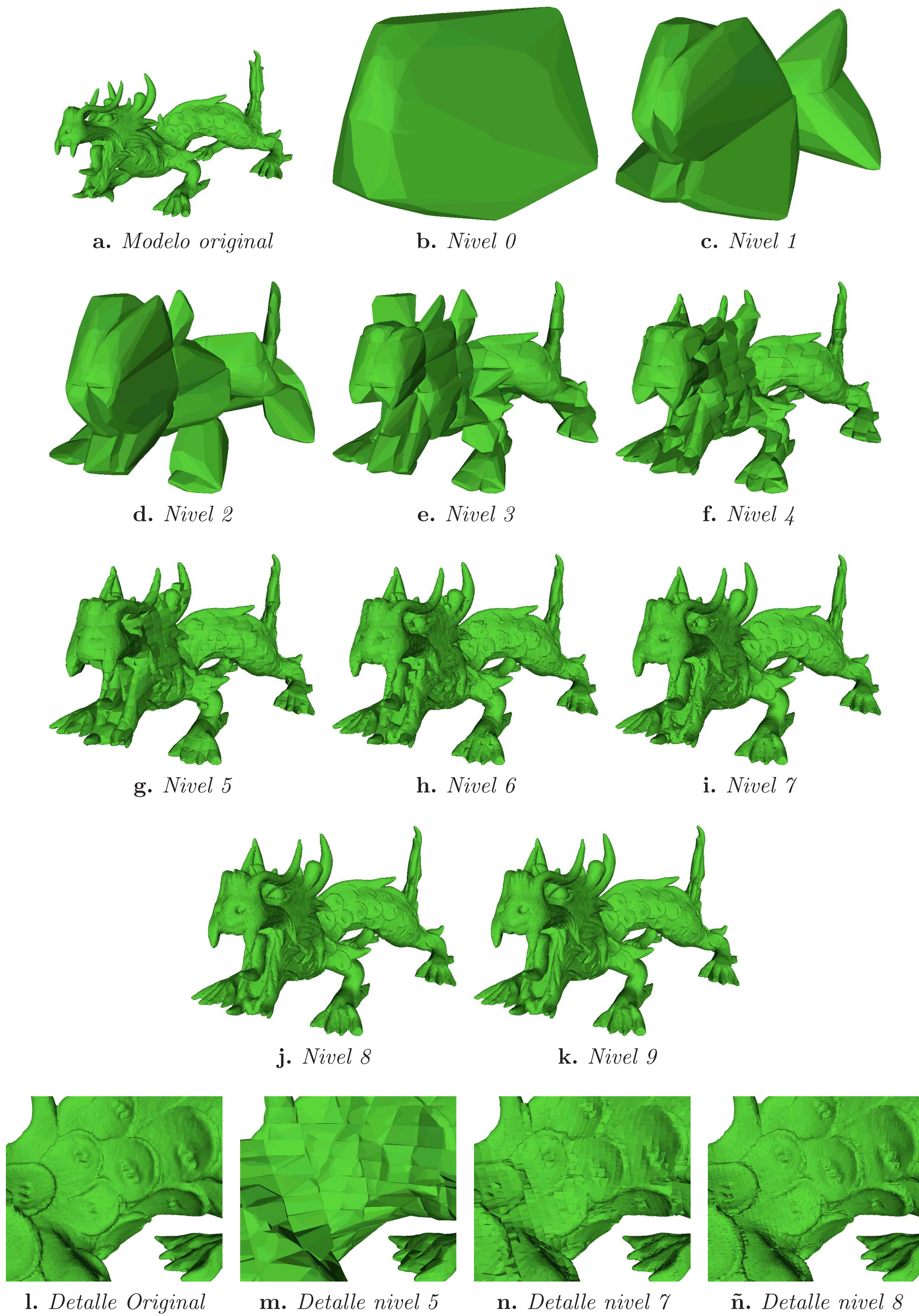
a. *Modelo original* **b.** *Nivel 0* **c.** *Nivel 1* **d.** *Nivel 2*

e. *Nivel 3* **f.** *Nivel 4* **g.** *Nivel 5* **h.** *Nivel 6*

i. *Nivel 7* **j.** *Nivel 8*

k. *Detalle Original* **l.** *Detalle nivel 4* **m.** *Detalle nivel 5* **n.** *Detalle nivel 7*

Figura 4.14: *Modelo Blade multirresolución.*



a. *Modelo original* **b.** *Nivel 0* **c.** *Nivel 1*

d. *Nivel 2* **e.** *Nivel 3* **f.** *Nivel 4*

g. *Nivel 5* **h.** *Nivel 6* **i.** *Nivel 7*

j. *Nivel 8* **k.** *Nivel 9*

l. *Detalle Original* **m.** *Detalle nivel 5* **n.** *Detalle nivel 7* **ñ.** *Detalle nivel 8*

Figura 4.15: *Modelo Asian Dragon multirresolución.*

4.2. Transmisión progresiva

La estructura que se presenta tiene como característica inherente la reutilización de datos, al ser los planos de un nivel dado el conjunto del cual se seleccionan aquellos que forman la envolvente en el nivel inmediatamente superior del árbol. Es por ello por lo que se puede realizar una transmisión del modelo desde el nodo raíz o desde un nivel más bajo, en función del ancho de banda disponible.

Modelo/Nivel	0	1	2	3	4	5	6	7	8	9
Bunny	656	2.004	4.976	11.716	25.093	53.180	86.010	50	-	-
Golf	999	3.846	11.021	17.624	33.550	69.229	125.085	12.566	-	-
Armadillo	1.228	3.497	7.543	16.548	35.022	72.610	144.834	160.059	645	-
Egipcio	1.501	4.107	10.976	23.362	49.927	99.869	175.344	113.897	66	-
Teeth	2.029	7.240	13.914	28.588	59.159	121.404	233.530	149.484	-	-
Fertility	4.462	12.053	25.689	49.840	96.595	184.336	349.661	592.770	88.237	28
Angelo	4.990	8.535	16.343	31.358	69.780	155.357	337.284	654.202	608.250	39.177
Phlegmatic	3.636	7.633	18.086	43.675	93.745	199.602	406.802	738.714	535.141	1.343
St. Dragon	2.955	7.287	18.373	43.876	99.023	203.160	408.036	754.485	672.812	26.074
Happy Budda	4.816	10.017	21.006	47.276	105.454	229.782	462.378	867.293	904.894	94.281
Blade	2.431	8.706	19.198	49.381	126.592	324.505	765.858	1.646.645	1.318.353	6.984
Dragon	3.970	10.293	33.266	88.022	192.631	429.974	938.582	2.015.802	4.197.030	7.804.535

Tabla 4.1: *Número de planos en cada nivel del BP-Octree.*

En la tabla 4.1 mostramos la distribución del número de planos en cada uno de los niveles para los modelos de evaluación mostrados en las figuras 3.33 a 3.44.

Modelo/Nivel	0	1	2	3	4	5	6	7	8	9
Bunny	0,92	2,82	7,00	16,49	35,32	74,86	121,07	0,07	0,00	0,00
Golf	1,00	3,84	10,99	17,58	33,47	69,06	124,78	12,54	0,00	0,00
Armadillo	0,82	2,33	5,03	11,03	23,35	48,41	96,56	106,71	0,43	0,00
Egipcio	0,93	2,54	6,78	14,43	30,84	61,68	108,30	70,35	0,04	0,00
Teeth	0,87	3,10	5,97	12,26	25,37	52,06	100,14	64,10	0,00	0,00
Fertility	0,92	2,49	5,32	10,31	19,99	38,15	72,36	122,67	18,26	0,01
Angelo	0,74	1,26	2,42	4,65	10,34	23,02	49,99	96,95	90,14	5,81
Phlegmatic Dragon	0,51	1,07	2,53	6,10	13,09	27,88	56,82	103,18	74,75	0,19
Stanford Dragon	0,34	0,84	2,11	5,04	11,36	23,31	46,82	86,58	77,21	2,99
Happy Budda	0,44	0,92	1,93	4,35	9,69	21,13	42,51	79,74	83,19	8,67
Blade	0,14	0,49	1,09	2,80	7,17	18,38	43,38	93,27	74,68	0,40
Asian Dragon	0,05	0,14	0,46	1,22	2,67	5,96	13,00	27,92	58,14	108,11

Tabla 4.2: *Porcentaje de referencias a planos en cada nivel sobre el total de polígonos del modelo.*

Se puede observar en la tabla 4.2 cómo en el primer nivel, se utiliza un porcentaje menor de planos conforme aumenta el tamaño del modelo, y en la figura 4.16 se muestran algunos modelos a nivel 4 que resultan una aproximación visual bastante buena del modelo original, oscilando entre el 35,32 % de los planos originales del modelo Stanford Bunny al 2,67 % del

total de planos posibles en el modelo Asian Dragon. Además, esta impresión visual se ve corroborada con los datos volumétricos de la tabla 4.3 y en la figura 4.17.

Hay niveles en los que hay más referencias a planos que polígonos tiene el modelo completo. Ello es debido a que un plano puede ser usado en varios nodos adyacentes. Este fenómeno suele producirse en los niveles inferiores del árbol, a partir del sexto.

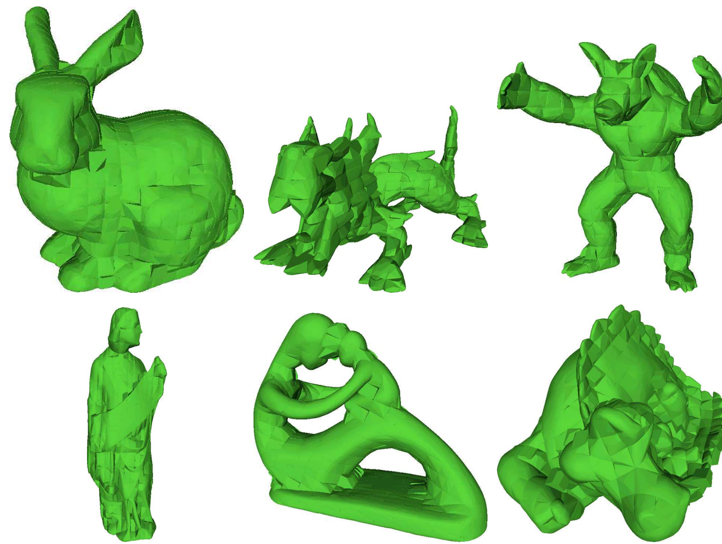


Figura 4.16: *Distintos modelos a nivel 4, por debajo del umbral que supone el tamaño total de la geometría original.*

	0	1	2	3	4	5	6	7	8
Bunny	171,41 %	144,71 %	119,92 %	107,33 %	103,11 %	101,14 %	100,55 %	100,55 %	100,55 %
Golf	103,34 %	101,84 %	101,74 %	101,68 %	101,17 %	100,50 %	100,21 %	100,21 %	100,21 %
Armadillo	388,73 %	214,88 %	139,57 %	116,78 %	107,99 %	103,45 %	101,57 %	100,92 %	100,92 %
Egipcio	150,02 %	133,02 %	113,27 %	105,79 %	102,18 %	101,03 %	100,51 %	100,41 %	100,41 %
Fertility	203,49 %	182,88 %	140,04 %	114,99 %	104,17 %	101,20 %	100,48 %	100,24 %	100,22 %
Teeth	161,69 %	117,11 %	112,34 %	105,93 %	102,53 %	101,07 %	100,44 %	100,35 %	100,35 %
Angelo	138,15 %	138,47 %	121,46 %	114,07 %	107,23 %	103,58 %	101,53 %	100,62 %	100,39 %
Phlegmatic	170,09 %	143,14 %	128,27 %	112,90 %	105,97 %	102,33 %	100,84 %	100,29 %	100,22 %
St. Dragon	277,82 %	199,08 %	165,60 %	135,50 %	111,44 %	103,68 %	101,30 %	100,50 %	100,32 %
Happy budda	196,91 %	177,06 %	144,42 %	125,48 %	110,23 %	103,58 %	101,27 %	100,44 %	100,26 %
Blade	373,79 %	246,02 %	199,07 %	152,86 %	125,31 %	108,97 %	103,40 %	101,37 %	100,90 %
Asian Dragon	583,10 %	320,75 %	193,04 %	133,70 %	114,69 %	106,61 %	102,90 %	101,19 %	100,43 %

Tabla 4.3: *Volumen representado del sólido en cada nivel (siendo el BREP el 100 %).*

Hay que destacar que el número de planos por nivel no supone la transmisión de cuatro coeficientes reales por plano más su respectivo desplazamiento, sino que la mayoría de ellos ya han sido usados en niveles superiores o han sido utilizados previamente por otros nodos del mismo nivel, en cuyo caso sólo se transmite un índice entero y el desplazamiento.

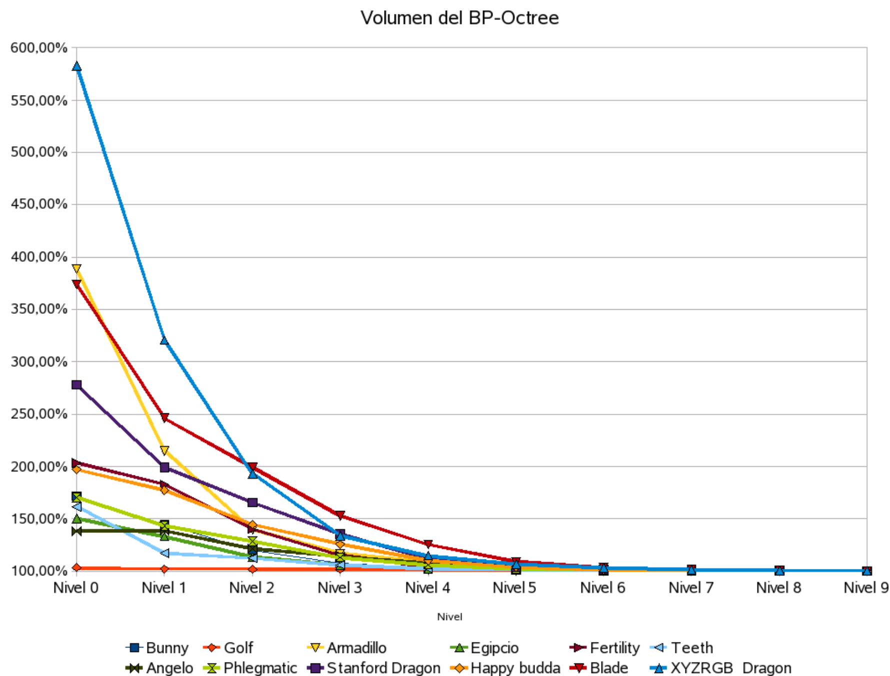


Figura 4.17: *Volumen por nivel de BP-Octree.*

En las figuras 4.18 a 4.28 podemos consultar la transferencia de datos de forma acumulada por nivel. Para ello se tienen en cuenta los siguientes valores:

- **1 byte** por cada nodo hoja para indicar *la posición de las esquinas del nodo*,
- **16 bytes** por cada plano la primera vez que se envía, que corresponden a los *cuatro coeficientes reales* del mismo.
- **4 bytes** por cada plano transmitido con anterioridad, ya que sólo se enviaría su *índice.*
- **4 bytes** por cada plano desplazado, indicando el *offset*
- **12 bytes** por cada triángulo de la geometría original, sólo computado así la primera vez que se envía, que corresponden a los *índices de sus tres vértices.*
- **4 bytes** por cada triángulo en la segunda y sucesivas apariciones, correspondiendo a su *índice.*
- **12 bytes** por cada vértice de la geometría original en su primera aparición, correspondiendo a sus *tres valores coordenados.*
- **4 bytes** por cada vértice en la segunda y sucesivas apariciones, correspondiendo a su *índice.*

De esta forma, y con los datos que se muestran acompañando a cada una de las figuras 4.18 a 4.28 se puede estimar la transferencia de información necesaria para la transmisión del BP-Octree a través de una red de comunicaciones. En la figuras 4.5 a 4.15 se puede observar cómo desde los primeros niveles del árbol se obtienen ya modelos con un aceptable nivel de detalle.

Level	Planes Total	Planes Used	Planes Displaced	Polygons Total	Polygons Used	Vertices Total	Vertices Used	Nodes Gray	Nodes Leaf	Transm. at Level
0	656	0	634	0	0	0	0	1	0	13.035
1	2.004	697	1.888	0	0	0	0	8	0	31.384
2	4.976	2.243	4.597	9	9	27	27	46	1	73.558
3	11.716	6.012	10.480	330	194	990	904	208	17	232.372
4	25.093	15.071	21.542	3.624	2.073	10.872	9.906	768	189	1.006.872
5	53.180	34.868	43.155	30.912	16.391	92.736	84.530	2.086	1.706	3.248.614
6	86.010	66.600	63.448	141.442	86.641	424.326	398.076	2	11.694	879.022
7	50	41	39	102	71	306	292	0	13	516

Tabla 4.4: *Datos del modelo Stanford Bunny.*

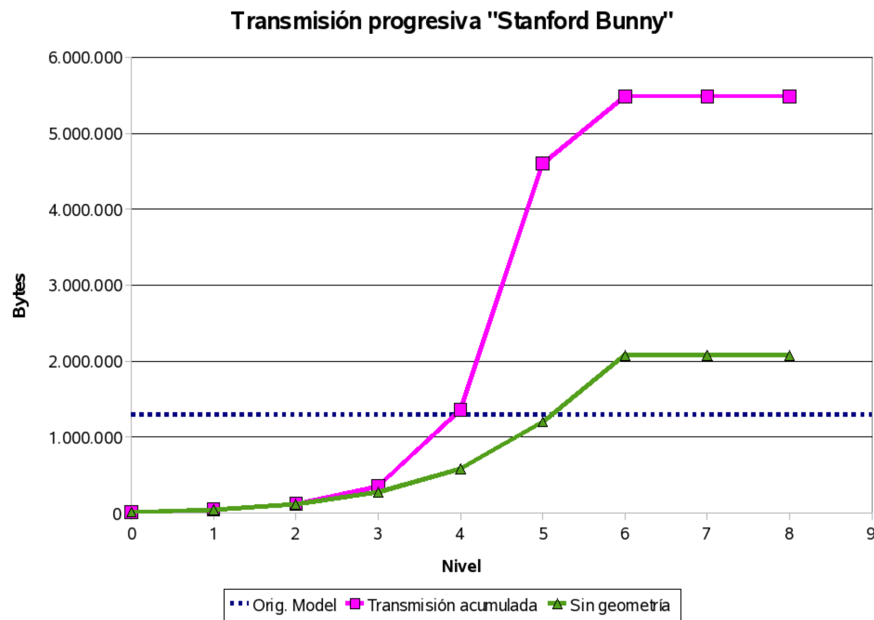


Figura 4.18: *Transmisión progresiva para el modelo Stanford Bunny.*

Un análisis de las figuras 4.18 a 4.28 nos lleva a la conclusión que cuanto mayor es el modelo representado con el BP-Octree, más tarde se alcanza el umbral de tamaño del modelo original. Además, el envío de la geometría contenida en los nodos hoja supone un grán coste en bytes, especialmente en los últimos niveles.

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	999	0	983	0	0	0	0	1	0	19.940
1	3.846	1.037	3.780	0	0	0	0	8	0	64.380
2	11.021	4.274	10.726	0	0	0	0	56	0	168.774
3	17.624	12.146	16.940	230	122	690	632	266	6	214.803
4	33.550	21.402	31.613	3.963	2.158	11.889	10.715	941	213	508.451
5	69.229	45.696	63.311	33.288	17.704	99.864	90.931	2.641	1.865	1.619.182
6	125.085	94.988	107.994	194.581	116.830	583.743	545.664	366	14.374	5.035.400
7	12.566	10.722	10.044	19.356	14.361	58.068	55.905	0	2.468	352.328

Tabla 4.5: *Datos del modelo Golf.*

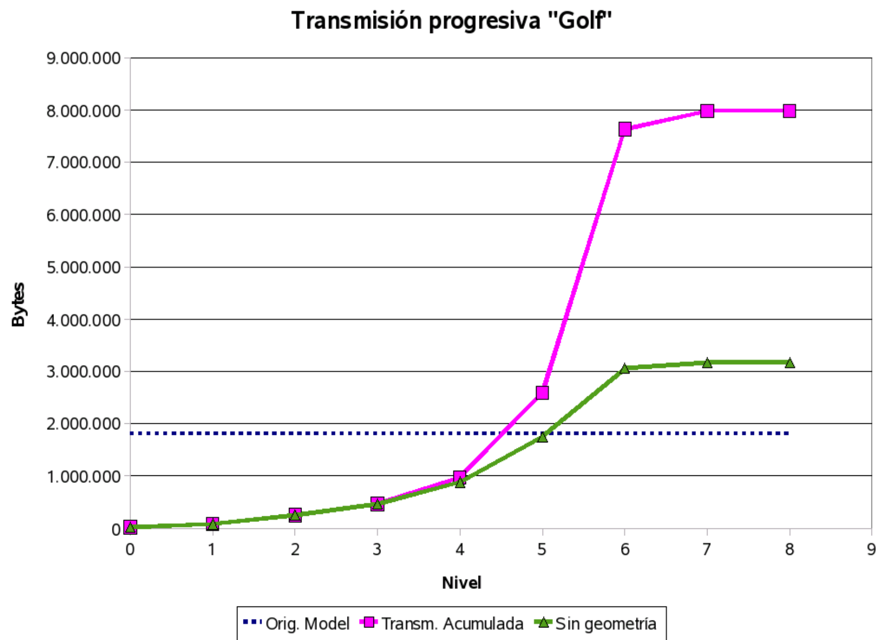


Figura 4.19: *Transmisión progresiva para el modelo Golf.*

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	1.228	0	1.189	0	0	0	0	1	0	24.407
1	3.497	1.269	3.304	0	0	0	0	8	0	53.964
2	7.543	3.755	6.940	19	9	57	51	45	1	103.635
3	16.548	8.450	14.909	176	63	528	439	173	11	229.999
4	35.022	19.893	30.569	1.840	969	5.520	4.940	673	93	488.111
5	72.610	45.671	59.971	14.640	8.093	43.920	39.998	2.308	768	1.177.844
6	144.834	103.570	108.563	133.237	73.214	399.711	367.734	3.954	7.885	4.433.262
7	160.059	129.735	111.412	245.672	163.529	737.016	698.713	21	24.276	5.489.803
8	645	521	395	1.081	788	3.243	3.118	0	153	20.324

Tabla 4.6: *Datos del modelo Armadillo.*

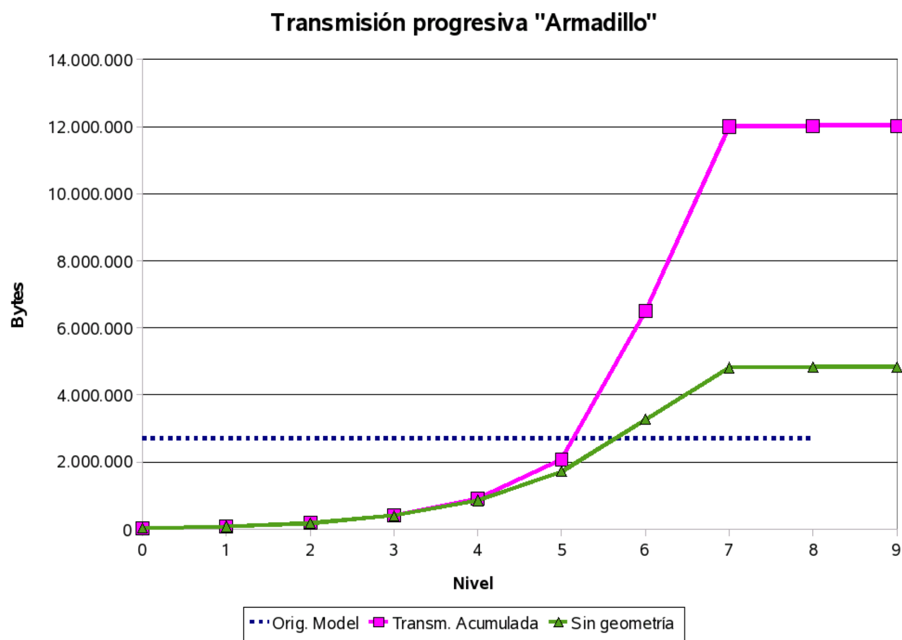


Figura 4.20: *Transmisión progresiva para el modelo Armadillo.*

	Planes			Polygons		Vertices		Nodes		Transm.
Level	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	1.501	0	1.448	0	0	0	0	1	0	29.811
1	4.107	1.561	4.002	0	0	0	0	8	0	63.012
2	10.976	4.508	10.659	0	0	0	0	57	0	164.327
3	23.362	12.570	22.467	246	136	738	648	283	22	321.213
4	49.927	28.889	47.500	3.214	1.548	9.642	8.483	1.107	202	726.261
5	99.869	66.699	92.410	34.325	17.935	102.975	93.321	3.327	1.759	1.970.893
6	175.344	133.988	153.946	216.556	125.946	649.668	603.337	3.068	13.799	6.227.112
7	113.897	97.488	93.093	176.357	123.248	529.071	505.343	2	19.401	3.651.710
8	66	53	58	111	87	333	327	0	16	1.868

Tabla 4.7: *Datos del modelo Egipcio.*

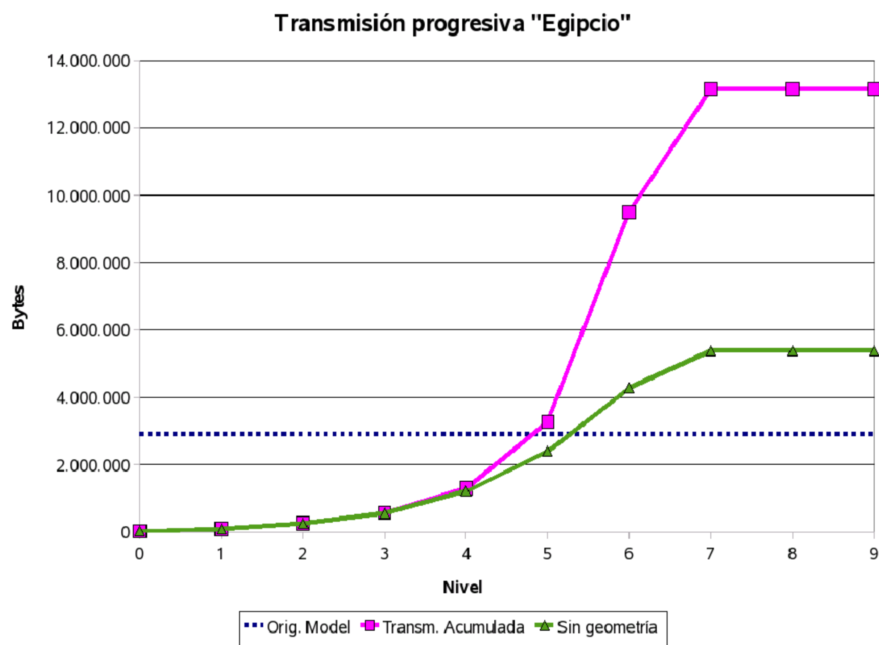


Figura 4.21: *Transmisión progresiva para el modelo Egipcio.*

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	2.029	0	1.982	0	0	0	0	1	0	40.395
1	7.240	2.101	6.869	0	0	0	0	8	0	118.128
2	13.914	7.634	13.108	0	0	0	0	62	0	183.634
3	28.588	15.610	26.535	246	137	738	659	318	16	384.670
4	59.159	34.249	53.510	2.493	1.266	7.479	6.692	1.335	119	812.973
5	121.404	78.165	105.966	28.790	15.148	86.370	78.337	4.529	1.526	2.102.855
6	233.530	169.764	193.700	293.041	164.373	879.123	814.196	4.564	17.373	8.733.360
7	149.484	117.122	121.471	260.248	170.690	780.744	737.966	0	23.405	5.864.568

Tabla 4.8: *Datos del modelo Teeth.*

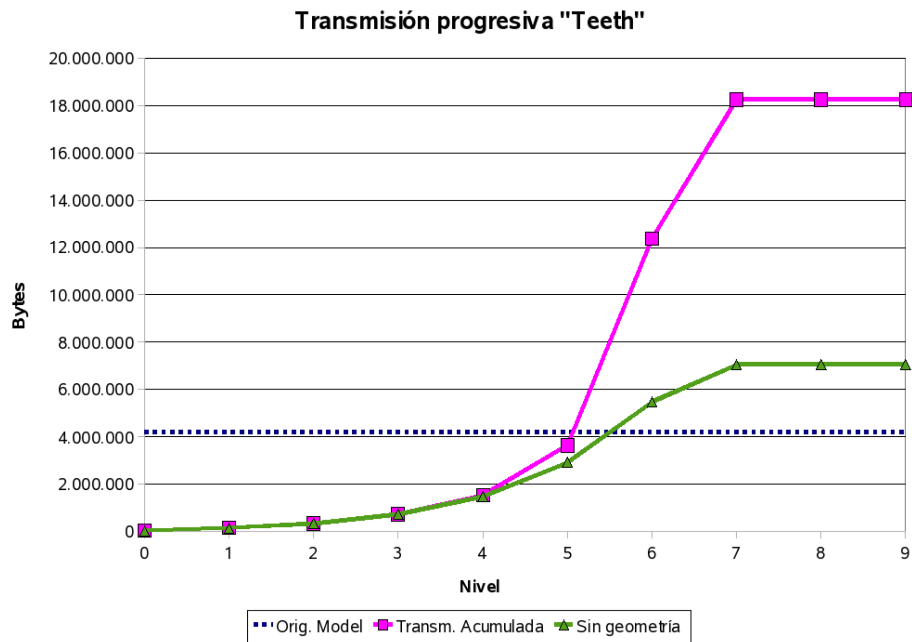


Figura 4.22: *Transmisión progresiva para el modelo Teeth.*

Level	Planes			Polygons		Vertices		Nodes		Transm. at Level
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	
0	4.990	0	4.877	0	0	0	0	1	0	99.351
1	8.535	5.090	8.365	0	0	0	0	8	0	108.964
2	16.343	8.847	15.906	0	0	0	0	42	0	219.074
3	31.358	17.450	30.191	192	91	576	519	201	7	420.027
4	69.780	35.726	66.434	1.040	542	3.120	2.829	843	54	980.153
5	155.357	85.935	145.248	10.405	5.521	31.215	28.488	3.346	544	2.282.130
6	337.284	207.878	305.617	93.951	49.266	281.853	257.515	10.702	5.101	6.321.866
7	654.202	465.330	562.429	668.910	376.918	2.006.730	1.856.655	14.287	42.489	21.361.421
8	608.250	489.043	495.106	917.505	599.928	2.752.515	2.598.963	1.103	87.067	21.439.181
9	39.177	33.041	32.456	60.304	45.277	180.912	174.570	0	7.555	933.708

Tabla 4.9: *Datos del modelo Angelo.*

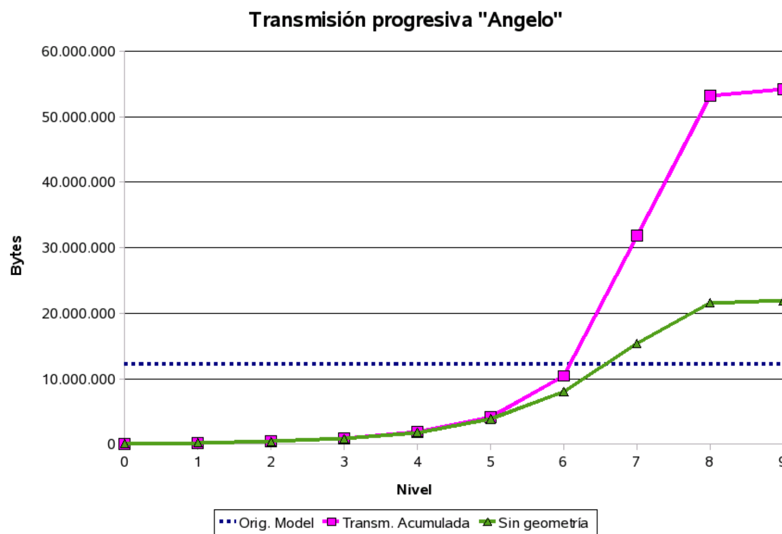


Figura 4.23: *Transmisión progresiva para el modelo Angelo.*

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	3.636	0	3.276	0	0	0	0	1	0	71.283
1	7.633	3.709	6.912	0	0	0	0	8	0	105.292
2	18.086	8.012	15.969	18	7	54	47	52	1	257.352
3	43.675	19.817	38.175	27	15	81	73	247	2	615.261
4	93.745	49.247	80.695	1.391	681	4.173	3.708	1.055	71	1.269.265
5	199.602	113.332	168.209	11.746	6.583	35.238	32.127	4.038	670	2.769.102
6	406.802	260.542	329.734	107.819	55.740	323.457	293.028	12.791	5.980	7.263.349
7	738.714	544.482	562.624	854.967	468.133	2.564.901	2.368.647	13.605	53.549	26.359.179
8	535.141	443.835	400.750	802.166	531.668	2.406.498	2.277.888	40	86.630	18.169.780
9	1.343	1.103	1.063	2.259	1.643	6.777	6.502	0	283	33.352

Tabla 4.10: *Datos del modelo Phlegmatic Dragon.*

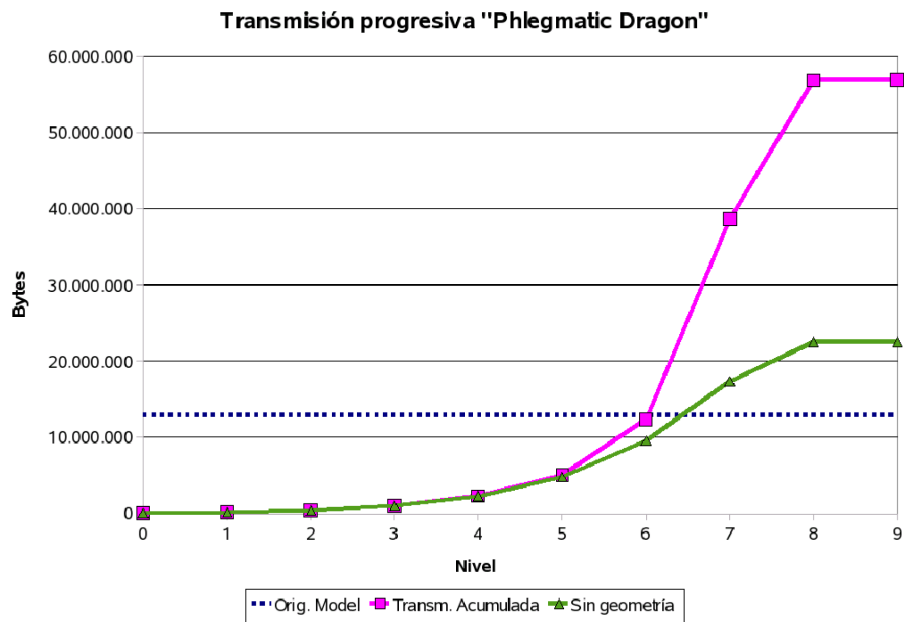


Figura 4.24: *Transmisión progresiva para el modelo Phlegmatic Dragon.*

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	2.955	0	2.689	0	0	0	0	1	0	58.039
1	7.287	3.023	6.510	0	0	0	0	8	0	106.380
2	18.373	7.694	16.187	0	0	0	0	53	0	266.547
3	43.876	20.090	38.487	129	19	387	307	267	4	621.285
4	99.023	49.949	85.445	1.038	630	3.114	2.870	1.206	62	1.350.210
5	203.160	118.868	170.583	11.086	6.178	33.258	30.132	4.727	639	2.758.797
6	408.036	266.429	328.052	124.837	67.019	374.511	340.361	14.931	6.933	7.491.425
7	754.485	545.746	576.341	900.007	506.341	2.700.021	2.493.912	18.409	57.878	27.010.879
8	672.812	528.315	494.001	1.155.304	757.590	3.465.912	3.280.744	864	111.654	25.940.696
9	26.074	19.949	19.855	52.722	35.932	158.166	149.398	0	5.486	710.344

Tabla 4.11: *Datos del modelo Stanford Dragon.*

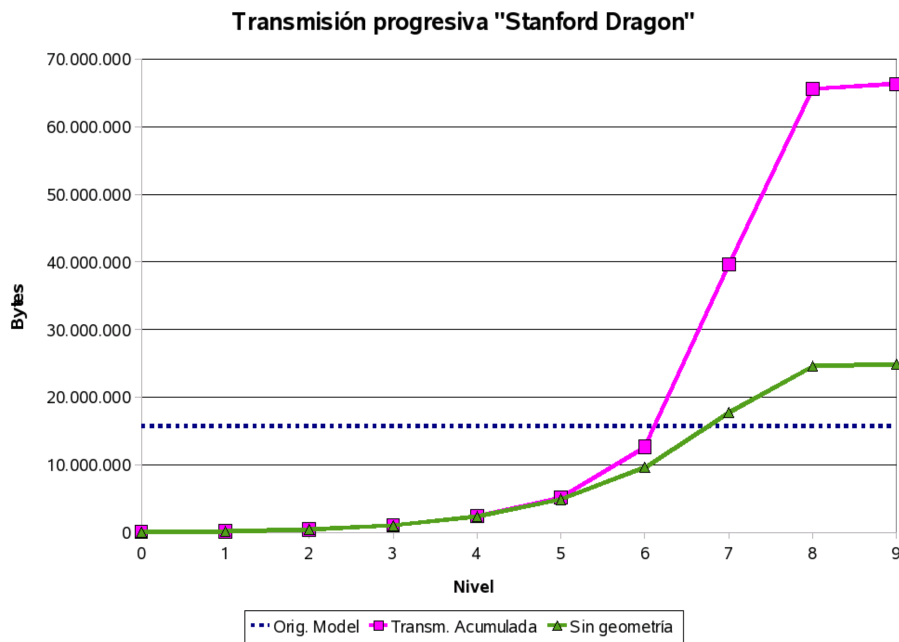


Figura 4.25: *Transmisión progresiva para el modelo Stanford Dragon.*

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	4.816	0	4.361	0	0	0	0	1	0	94.503
1	10.017	4.892	8.874	0	0	0	0	8	0	137.088
2	21.006	10.445	18.903	0	0	0	0	56	0	286.536
3	47.276	22.616	42.278	118	84	354	341	290	5	658.002
4	105.454	54.079	92.807	1.038	566	3.114	2.812	1.420	56	1.436.684
5	229.782	128.979	198.765	24.259	12.817	72.777	66.285	5.359	1.199	3.493.913
6	462.378	295.858	388.264	149.679	80.913	449.037	409.525	16.816	8.336	8.785.368
7	867.293	614.615	699.030	945.849	539.316	2.837.547	2.624.609	24.027	60.652	29.125.701
8	904.894	700.176	698.669	1.553.295	1.003.105	4.659.885	4.397.739	2.790	139.218	35.845.230
9	94.281	77.559	70.431	166.130	115.851	498.390	476.141	0	17.934	2.321.460

Tabla 4.12: *Datos del modelo Happy Budda.*

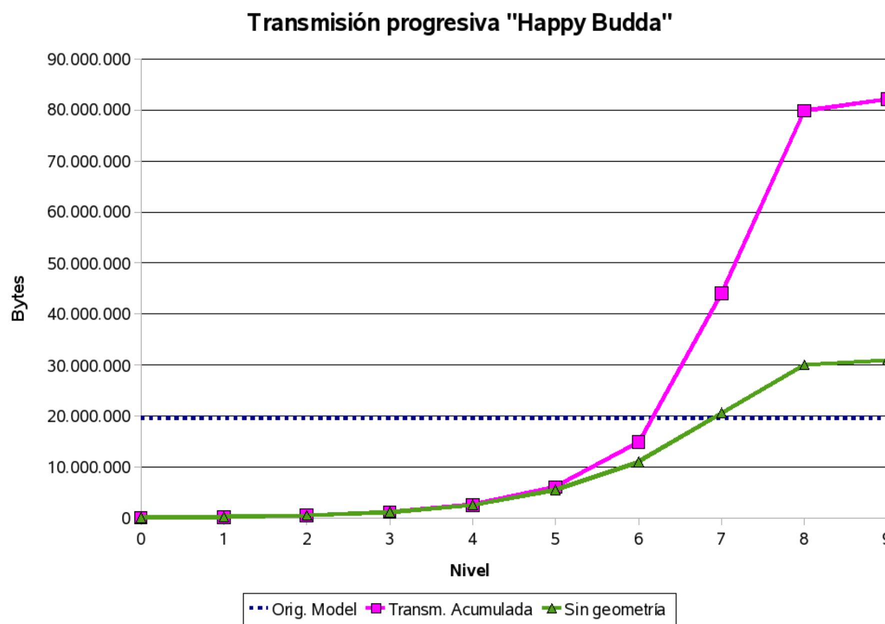


Figura 4.26: *Transmisión progresiva para el modelo Happy Budda.*

	Planes			Polygons		Vertices		Nodes		Transm.
Level	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	2.431	0	2.388	0	0	0	0	1	0	48.451
1	8.706	2.544	8.620	0	0	0	0	8	0	143.272
2	19.198	9.055	18.997	0	0	0	0	50	0	274.646
3	49.381	20.790	48.845	60	10	180	138	252	3	739.324
4	126.592	56.026	124.864	583	285	1.749	1.532	1.388	29	1.871.200
5	324.505	154.559	318.519	8.071	4.098	24.213	21.511	7.067	428	4.825.065
6	765.858	433.338	740.508	138.687	71.636	416.061	376.513	28.082	6.634	13.344.934
7	1.646.645	1.110.381	1.539.622	1.956.404	996.174	5.869.212	5.377.641	34.597	100.758	65.778.099
8	1.318.353	1.059.859	1.156.283	2.088.847	1.357.991	6.266.541	5.918.993	200	210.975	48.926.060
9	6.984	5.698	6.039	11.001	8.463	33.003	31.955	0	1.496	175.064

Tabla 4.13: *Datos del modelo Blade.*

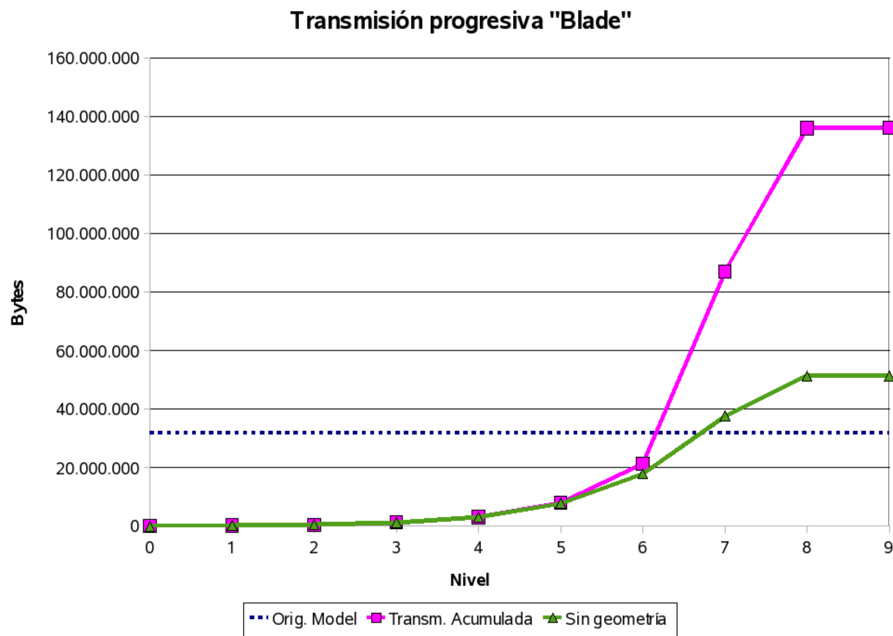


Figura 4.27: *Transmisión progresiva para el modelo Blade.*

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	3.970	0	3.804	0	0	0	0	1	0	78.739
1	10.293	4.016	9.835	0	0	0	0	8	0	155.860
2	33.266	10.588	31.695	0	0	0	0	35	0	532.114
3	88.022	34.410	83.611	0	0	0	0	161	0	1.330.710
4	192.631	92.077	182.729	145	103	435	416	691	7	2.715.617
5	429.974	208.162	405.170	1.914	995	5.742	5.152	2.935	105	6.085.595
6	938.582	484.657	877.611	16.098	8.531	48.294	43.506	11.890	860	13.365.312
7	2.015.802	1.129.825	1.849.838	146.256	77.920	438.768	396.446	44.341	7.762	31.599.056
8	4.197.030	2.634.363	3.725.166	1.332.424	694.197	3.997.272	3.629.577	134.978	69.693	98.471.321
9	7.804.535	5.650.728	6.531.042	13.594.294	7.090.479	40.782.882	37.588.841	0	707.693	306.029.933

Tabla 4.14: *Datos del modelo Asian Dragon.*

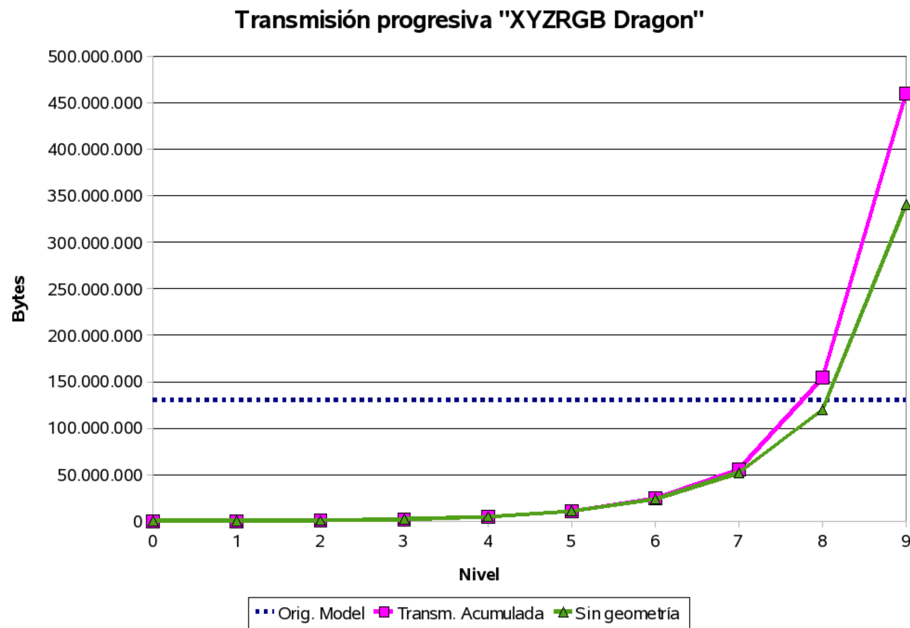
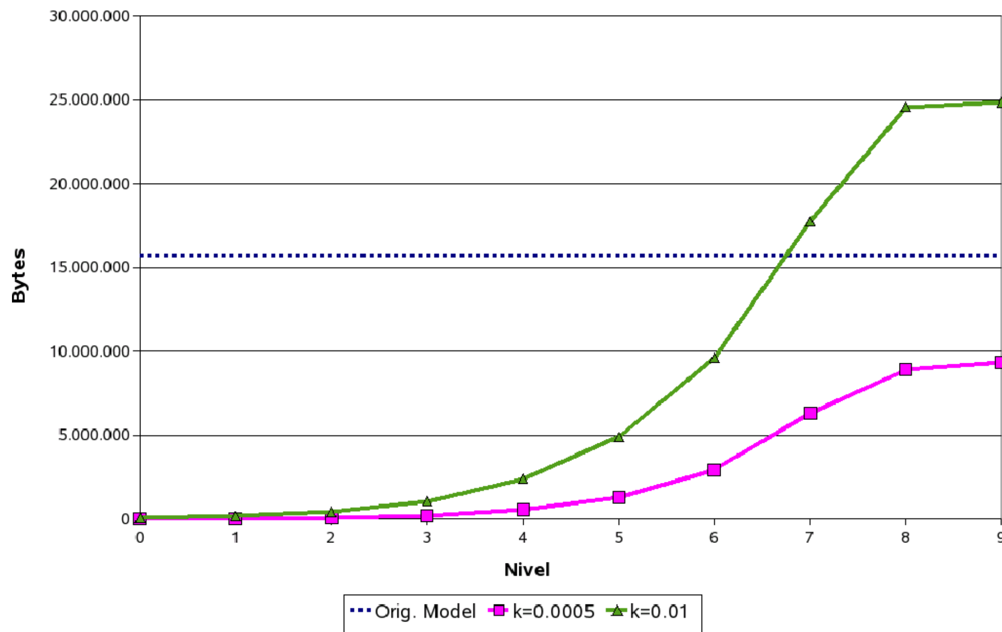
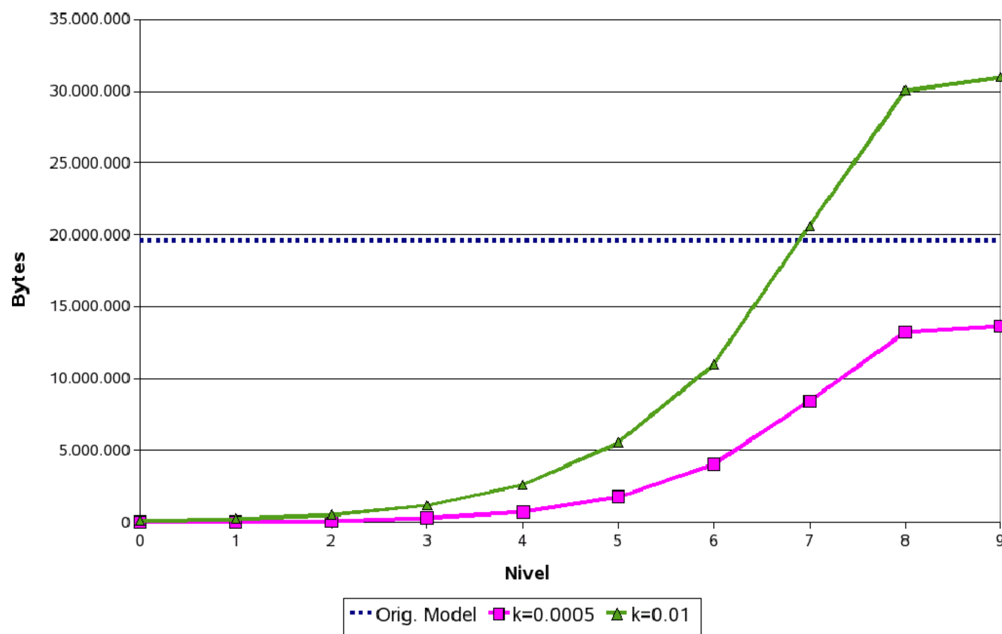


Figura 4.28: *Transmisión progresiva para el modelo Asian Dragon.*

En las figuras 4.29 y 4.30 se muestra la influencia del parámetro k también en la transmisión progresiva. En las gráficas se reflejan los valores de transmisión acumulada, sin enviar la geometría final, para un mismo modelo con $k = 0{,}01$ (usado en las figuras 4.18 a 4.28) y con $k = 0{,}0005$. Se puede observar como con $k = 0{,}0005$, enviando tan sólo la información de las envolventes, nunca se llega a alcanzar el valor umbral que define el tamaño del modelo original.

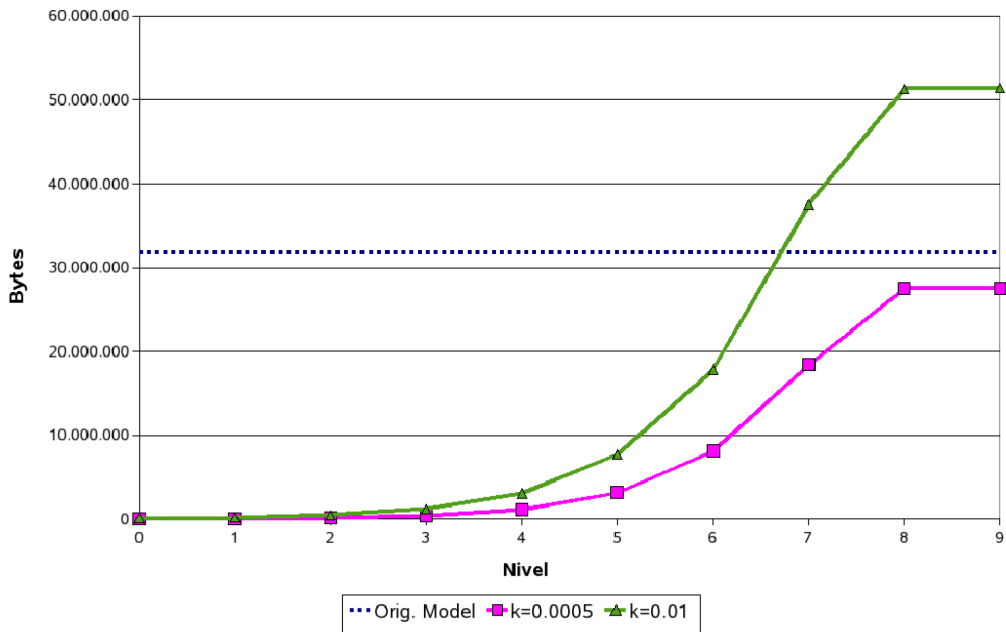


a. *Stanford Dragon*

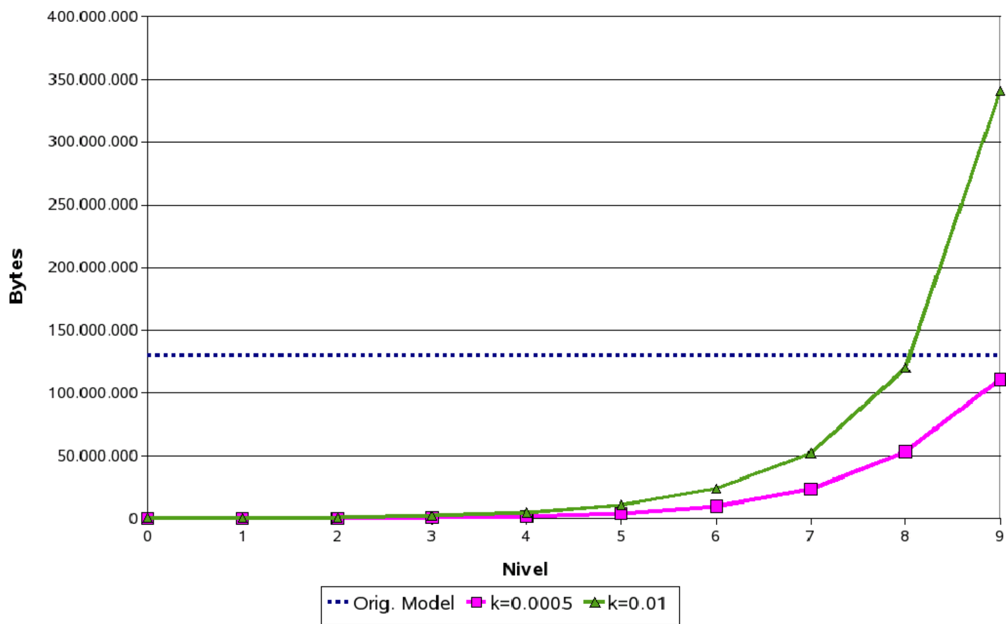


b. *Happy Budda*

Figura 4.29: *Influencia de k en la transmisión progresiva del modelo. Verde, $k = 0{,}01$. Rosa: $k = 0{,}0005$*



a. *Blade*



b. *Asian Dragon*

Figura 4.30: *Influencia de k en la transmisión progresiva del modelo, sin el envío de la geometría. Verde, $k = 0{,}01$. Rosa: $k = 0{,}0005$*

4.3. Visualización adaptativa

La estructura que sustenta a los BP-Octrees, permite de una forma sencilla y trivial realizar una visualización adaptativa de los modelos. Los criterios para decidir si se desciende o no de nodo son de nuevo variables, pudiendo ir desde la distancia al observador a un criterio de percepción como la resolución de cada nodo, tal y como se muestra en el algoritmo 9.

```
bool enoughDetail(BPONode n){
  bool enough=false;
  switch (::adaptiveCriterion){
    case DIAGWIDTH:
        double diagL=n.bbox().diagonalLength();
        enough=WorldToPixels(diagL)<MIN_WIDTH_PIXELS;
        break;
    case DIST_OBS:
        double dO=eucDistance(n.center(),observer.pos());
        enough=dO>MIN_DIST_OBSERVER;
 }
 return enough;
}

void adaptiveRender(BPONode n){
        if (isLeaf(n) || enoughDetail(n))
                n.render();
        else
            for_each hijo i de n
                adaptiveRender(i);
}

Llamada inicial: adaptiveRender(bpo.root());
```

Algoritmo 9: *Algoritmo de visualización adaptativa*

A modo de ejemplo las imágenes que se muestran en al figura 4.31 han sido tomadas de forma que la diagonal de cada nodo visible mide, como máximo 40 pixeles. En primer plano podemos observar la geometría a máximo nivel de detalle, mientras que en segundo plano se observan los nodos a inferior resolución.

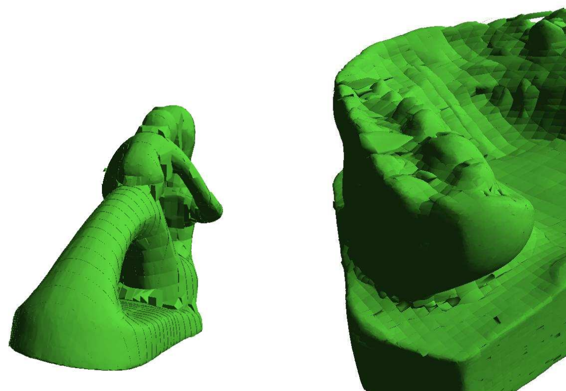


Figura 4.31: *Visualización adaptativa de los modelos Fertility y Teeth.*

4.4. Uso de impostores en BP-Octrees

De forma opuesta, aunque complementaria, a las técnicas multiresolución basadas en la geometría, se encuentran las técnicas de visualización basadas en imágenes. La idea que subyace bajo todas las técnicas es la sustitución de una geometría compleja por una imagen 2D de aquello que debería estar representado. Esta imagen es lo que se denomina un *impostor*.

Se han presentado muchas técnicas para representar sólidos basándose en imágenes [GD98, MS95, Ali96, SDB97, DSSD99, DYB98]. Todas ellas pueden considerarse dependientes del observador, ya que en todos los casos, en función del punto de vista en un instante, se selecciona una imagen entre las tomadas del modelo real desde posiciones bien conocidas. La complejidad del modelo geométrico sobre el que se aplican estas imágenes a modo de textura varía desde un simple plano hasta mallas más o menos complejas.

Se propone, para la visualización del BP-Octree a bajos niveles de detalle, el uso de *impostores* dependientes del observador, de forma análoga a lo presentado para el SP-Octree en [MCT04][MCT05].

4.4.1. Aplicación de *impostores*

La proyección de una textura (*projective texture mapping, PTM*) fue propuesta por Segal [SKv+92b], y es parte del estándar de OpenGL [SA94]. Aunque en el artículo original se usaba solamente para acelerar la generación de sombras y otros efectos de iluminación, esta técnica es directamente aplicable a la visualización basada en imágenes, ya que puede simular el proceso inverso de tomar fotografías con una cámara, es decir, permite proyectar imágenes sobre la escena como si fuese un cañón proyector.

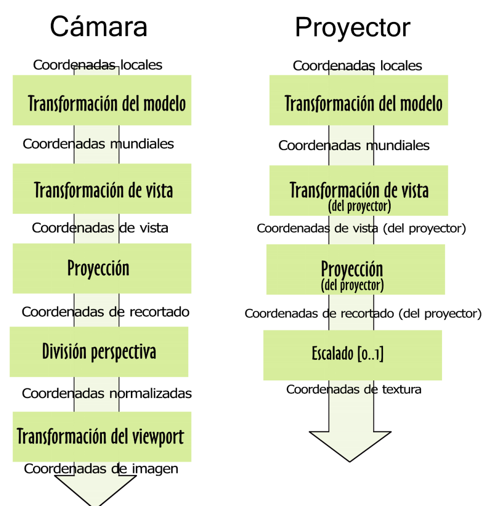


Figura 4.32: *Transformaciones que se aplican a los vértices de la geometría (para su visualización y la obtención de coordenadas de textura).*

Para realizar la proyección de la textura, el usuario especifica la posición y orientación del

proyector, así como un plano virtual donde se proyecta la imagen. Con estos datos, y las matrices de transformación y proyección de la escena actual, se calculan las coordenadas de textura adecuadas.

De hecho, las transformaciones que se realizan sobre los vértices para obtener sus coordenadas de textura son muy similares a las que se realizan sobre los propios vértices durante el proceso de visualización al pasarlos de coordenadas de mundo a coordenadas de imagen, tal y como se puede contemplar en la figura 4.32.

Como se ha comentado anteriormente, OpenGL soporta desde su versión 1.0 el *PTM*, y la clave de todo el proceso se encuentra en la función de generación automática de coordenadas de texturas, *texgen*. De esta forma, para proyectar en OpenGL una textura sobre un modelo (o parte de él), tan sólo es necesario controlar las matrices *modelview* y *perspective* del proyector de la textura y del observador de la escena. Con estos datos, y en función del modo de generación de las texturas, se combinan dichas matrices para obtener la matriz de textura, que es la que dirige las transformaciones que se aplican sobre los vértices para obtener las coordenadas de textura. Dicho proceso se puede reproducir fácilmente siguiendo lo dispuesto en [Eve, SA94].

4.4.2. Generación de *impostores*

La generación del conjunto de imágenes a utilizar como *impostores* se realiza una única vez por objeto a representar, y se obtienen desde diversos puntos de la esfera que contiene a la caja englobante del sólido. Las imágenes se han tomado renderizando el objeto a su máxima resolución, y colocando la cámara en los vértices que se obtienen al formar una esfera a partir de un tetraedro. Dicha esfera puede construirse a distinto nivel de detalle, lo que por tanto permite generar más o menos vistas del objeto según las necesidades.

Otra posibilidad hubiera sido realizar una determinación de los puntos de visibilidad de cada plano almacenado en la estructura del BP-Octree y realizar un muestreo de imágenes para cada uno de ellos desde dichos lugares. Sin embargo, consideramos que esto conlleva un aumento considerable de las necesidades de memoria, y un aumento en el tiempo de cálculo, ya que cada plano tiene su propia textura y la operación de intercambio de imágenes consume cierto tiempo.

Variando la resolución de la esfera (y por tanto el número de imágenes utilizadas), y comprobando la calidad de la visualización interactiva del modelo tridimensional en cada caso, observamos que con un nivel 4 de detalle (fig. 4.33), en el que se generan 258 vértices, es suficiente para que la transición de una textura a otra se pueda realizar con suavidad, incluso sin realizar fundido de texturas durante la transición entre ellas.

Dados los vértices, se coloca la cámara en dicho punto, orientada hacia el centro de la esfera en todo momento. No es necesario controlar el sentido del vector *up*, ya que toda la información acerca de la posición y orientación de la cámara se almacena en la matriz de transformación, con la que OpenGL trabaja para realizar el mapeado automático de la textura.

La proyección utilizada para tomar las imágenes es la ortográfica, para obtener una vista del objeto independiente de la distancia del plano de proyección a él. Lo único que hay que controlar es que el objeto quepa totalmente en el plano de proyección.

Todas las imágenes se almacenan en un único fichero binario, incluyendo en éste información acerca de la matriz de proyección utilizada para la toma de las imágenes, así como cada una las matrices de transformación utilizadas para posicionar la cámara en cada una de las

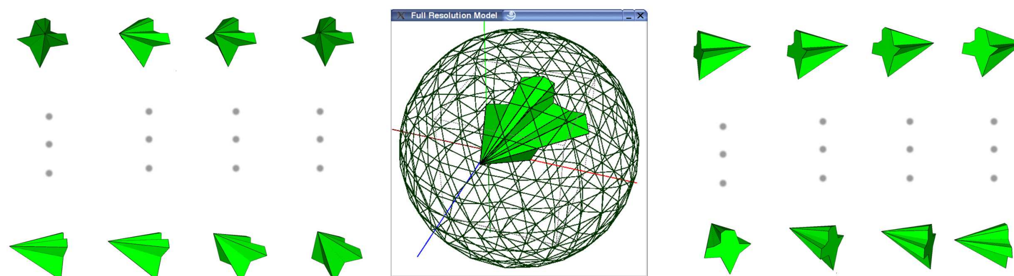


Figura 4.33: *Esfera de nivel 4 usada para tomar los impostores del modelo.*

tomas. Todos los píxeles pertenecientes al objeto tienen un valor *alpha* 0.0, mientras que los correspondientes al fondo, totalmente blancos, se almacenan con el valor *alpha* a 1.0, es decir son transparentes.

4.4.3. Visualización del BP-Octree con *impostores*

Para visualizar el modelo, se recorre el árbol de forma descendente, generando los polígonos correspondientes a cada nivel. Si el polígono generado no forma parte de la frontera del sólido, es decir, es un polígono de un nodo gris que se ha decidido pintar de forma aproximada, se le aplica el *impostor*; si, por el contrario, es un polígono contenido en la frontera del sólido, se le aplica su textura original.

Como para cada imagen almacenamos su matriz de transformación, es posible conocer el vector de vista utilizado cuando fue tomada. En lugar de trabajar con coordenadas cartesianas, y dado que el módulo de dicho vector es indiferente, trabajamos con las coordenadas polares (*azimut* y *zenit*) del mismo.

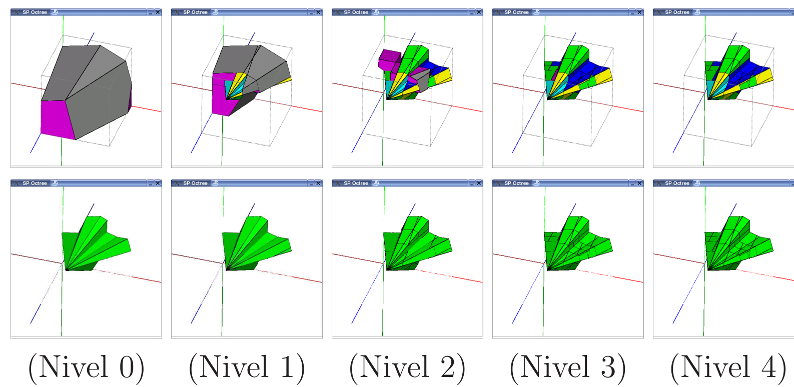
Más concretamente, seguimos el siguiente proceso:

1) En cada momento de la visualización, se comprueba la posición del observador con respecto al objeto. Esta posición determina el vector de vista que nos va a servir para localizar la imagen que fue tomada desde un punto de vista similar. Nótese como no tenemos en cuenta el lugar hacia donde mira el observador, sino la posición relativa del mismo con respecto al objeto, ya que de no hacerse así, se cargaría un *impostor* erróneo.

2) Se busca en la tabla de *impostores* aquél que mejor se ajusta a la vista que se va a dibujar, y se toma el identificador de textura que tiene asignado.

3) Se recorre el árbol en preorden hasta el nivel deseado, generando los polígonos que corresponda en cada nodo. Si el polígono forma parte de la frontera del objeto, se dibuja con su textura original (o con el código de colores si se carece de textura). Si, por el contrario, nos encontramos ante un plano de un nodo gris no perteneciente a la frontera, se le aplica el impostor seleccionado previamente.

En la figura 4.34 se puede observar cómo la selección del impostor adecuado en función de la posición relativa del observador con respecto al objeto hace que lo observado se ajuste al



(Nivel 0) (Nivel 1) (Nivel 2) (Nivel 3) (Nivel 4)

Figura 4.34: *Visualización de un SP-Octree sin impostores (superior) y con impostores (inferior).*

modelo final, y en todo momento, la sensación es estar trabajando con el objeto al máximo nivel de detalle. Idéntico comportamiento se daría usando un BP-Octree.

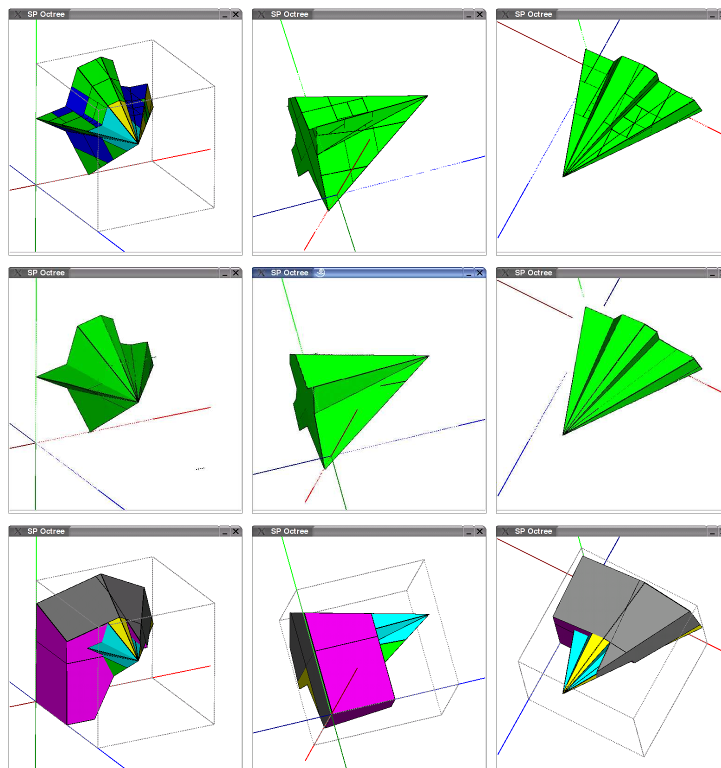


Figura 4.35: *Visualización con impostores en el nivel 1 (fila central), y sin usar impostores en el nivel 1 (inferior). En la fila superior, SP-Octree a máximo detalle, representando la geometría original.*

En la secuencia 4.35 podemos apreciar como desde distintos puntos de vista, en cada momento se selecciona el *impostor* más adecuado.

CAPÍTULO 5

Detección de colisiones

5.1. Introducción.

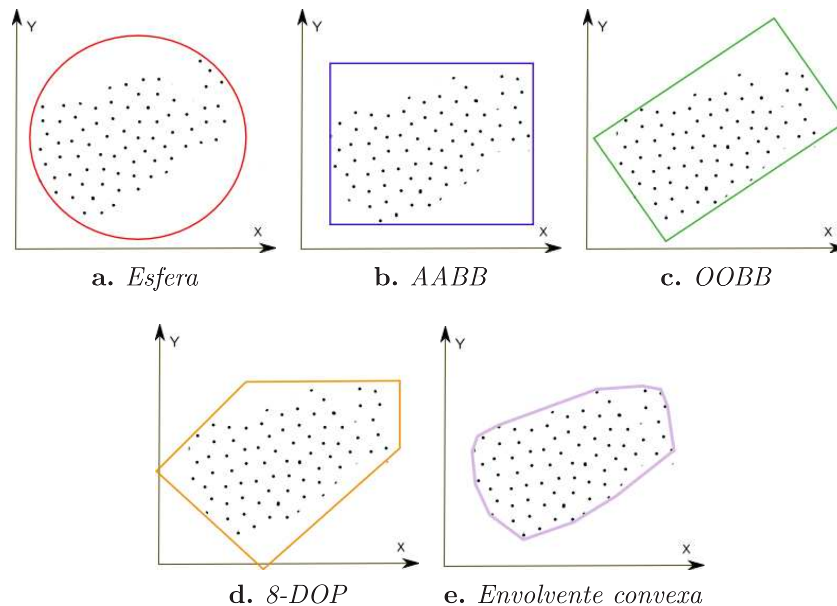
La detección de colisiones, tema recurrente en la comunidad gráfica desde hace varias décadas, trata de detectar si dos o más objetos intersecan entre sí. En realidad, el problema es algo más complejo, puesto que además de saber *si* colisionan, es interesante en la mayoría de las aplicacionies conocer *cuándo* colisionan y *dónde*, esto es, en qué punto entran en contacto los dos objetos. Otros parámetros que definen el *cómo* colisionan (fuerza, direccion, ángulo, etc...) son igulamente interesantes para la simulación realista de la interacción entre los objetos.

Las aplicaciones de la detección de colisiones son muy diversas: la industria de los videojuegos, la animación por ordenador, la simulación de recorridos, entornos de realidad virtual o sistemas de entrenamiento, entre otros.

De estas aplicaciones, hay algunas que requieren una respuesta lo más fiable físicamente posible, como es el caso de las animaciones por ordenador. En este caso, no es tan importante el tiempo de cómputo como la respuesta del elemento en función de las leyes físicas aplicables. En los entornos de realidad virtual, y en concreto aquellos con interacción táctil del usuario con el mundo virtual, además de fiabilidad física es necesario que la simulación se realice en tiempo real. En los denominados entornos hápticos [LD03, SCB04] es generalmente aceptado por la comunidad científica que la detección de colisión debe realizarse en menos de un milisegundo.

5.1.1. Sistemas de detección de colisiones.

El diseño de un sistema de detección de colisiones debe tener en cuenta diversos factores que influyen en su rendimiento. En primer lugar se debe considerar la representación del dominio, esto es, el formato en que los objetos gráficos están representados. Lo más común es el uso de una representación *B-rep* basada en triángulos, con la consecuente carencia de información explícita acerca del interior y el exterior del modelo. También hay que tener en cuenta el tipo de respuestas que se esperan del sistema (sólo *colisionan* o *no colisionan*,



a. *Esfera* **b.** *AABB* **c.** *OOBB*

d. *8-DOP* **e.** *Envolvente convexa*

Figura 5.1: *Distintos volúmenes envolventes para una nube de puntos.*

puntos de intersección, profundidad de penetración, ruta de salida, etc...). Además, no es lo mismo tener un punto y un objeto, o dos objetos y detectar su colisión, que un sistema con decenas de objetos en movimiento. Obviamente, los resultados esperados y la cantidad de tests a realizar difieren bastante de uno a otro modelo. En definitiva, según el tipo de aplicación en el que se va a utilizar el sistema de detección de colisiones se tenderá a uno u otro, puesto que es la aplicación de usuario la que nos marca en cierto modo la frecuencia con la que se realizarán las consultas y el tipo de éstas, el formato de los datos de entrada, la tolerancia a imprecisiones numéricas, etc.

No es el propósito de este capítulo realizar una extensa disertación sobre el tema. Un excelente libro introductorio a la detección de colisiones en tiempo real es el de [Eri04].

5.1.1.1. Volúmenes envolventes.

Es obvio que la realización de tests de colisión entre modelos de varios cientos o miles de polígonos es inviable de forma directa, esto es, confrontando cada polígono con cada uno de los restantes. Una primera estrategia para acelerar este proceso es usar *volúmenes envolentes*, descartando de forma rápida los casos de *no-colisión*.

La idea es simple: tener como volumen envolvente una zona del espacio lo suficientemente simple como para realizar tests de intersección de una forma muy rápida. Estas zonas del espacio van desde esferas -lo más simple- hasta la envolvente convexa mínima del modelo (el *convex hull*), como se puede apreciar en la figura 5.1. Cuanto más ajustada es la envolvente menos falsos positivos se producen, pero es mayor el tiempo y el espacio de almacenamiento requerido para realizar la consulta.

Los volúmenes envolventes más relevantes existentes en la bibliografía son:

- *Esferas envolventes* [Hub96]. Requieren la consulta más sencilla de todos los volúmenes envolventes, ya que dos objetos -sus esferas envolventes- intersecan si la distancia entre sus centros es menor que la suma de sus radios. Además, tiene la ventaja de que es invariante a la rotación.

- *Cajas envolventes alineadas a los ejes (Axis-aligned Bounding Boxes, AABBs)* [Ben97]. Es un paralelepípedo tridimensional que tiene sus caras alineadas a los planos del sistema de coordenadas. La principal ventaja es la rápida detección de intersección, ya que tan sólo hay que evaluar las proyecciones de las dimensiones sobre los planos coordenados, y si intersecan las tres proyecciones, las cajas intersecan entre sí. Además, su volumen está mucho más ajustado a la geometría que las esferas envolventes.

- *Cajas envolventes orientadas (Object-oriented Bounding Boxes,OBBs)* [GLM96]. Es un paralelepipedo tridimensional con orientación arbitraria, de forma que se ajusta de forma más exacta al modelo que los AABBs. Tiene algunos inconvenientes con respecto a los AABB, como puede ser el mayor espacio necesario para su almacenamiento y la mayor complejidad en el cálculo de intersecciones, ya que hay que aplicar el teorema de separación de ejes (más conocido en sus términos ingleses: *Separating axis theorem* [Got96]).

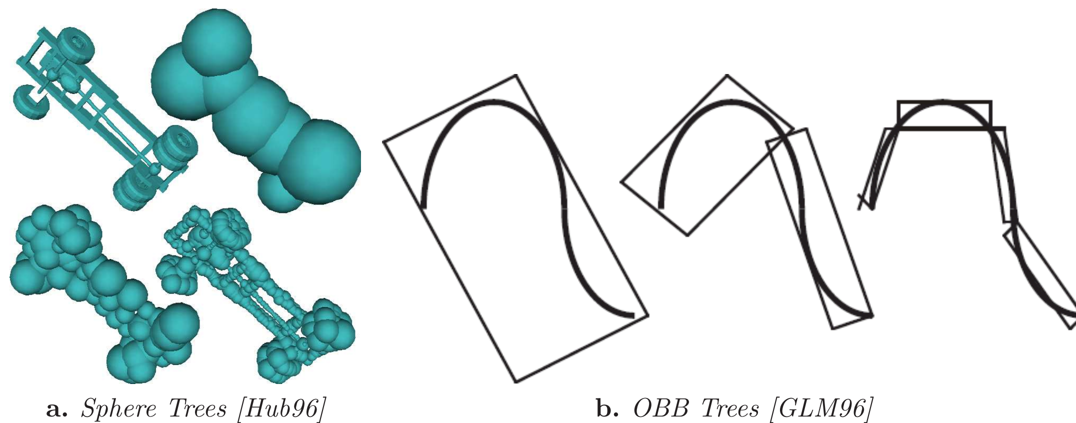
- *Envolventes de orientación fija (Discrete-orientation polytopes, k-DOPs)* [KZ97][KHM+98]. Se definen mediante un conjunto de planos cuya normal viene predeterminada en un conjunto de cardinalidad k, de forma que un 6-DOP tiene sus caras con normales en el conjunto $(\pm 1, 0{,}0), (0, \pm 1, 0), (0, 0, \pm 1)$ –un AABB es un caso especial de 6-DOP–, y un 8-DOP con las ocho direcciones $(\pm 1, \pm 1, \pm 1)$. Al estar predefinidas las orientaciones de los planos envolventes, los algoritmos de cálculo de colisiones son mucho más eficientes que en el caso de AABBs y OBBs. Una variante es elegir las orientaciones óptimas de los planos en funcion de un algoritmo evolutivo [Zac00].

- *Envolvente convexa de volumen mínimo (Convex Hull)*. El *convex hull* de un sólido está formado por el *conjunto convexo de puntos S* más pequeño que contiene completamente al sólido, siendo un conjunto convexo de puntos S aquel en el cual cualquier segmento que une un par de puntos queda completamente contenido en S.

En el caso de intersección entre los volúmenes envolventes, según el sistema de aplicación, puede ser suficiente dar un valor de *colisión* positivo aunque éste sea aproximado, como en el caso de videojuegos, o puede que haya que refinar hasta detectar la colisión real con la superficie del objeto, como en aplicaciones hápticas.

5.1.1.2. Jerarquías de volúmenes envolventes.

El uso de volúmenes envolventes supone una importante mejora sobre el método *de fuerza bruta* para detectar si dos o más objetos intersecan entre sí, puesto que se descarta de manera muy eficiente lo casos *negativos*. Sin embargo, en el caso de un positivo con el volumen envolvente, hay que recurrir a la comprobación primitiva a primitiva de la colisión entre los objetos, lo que nos lleva al peor de los casos. Para reducir la complejidad de los tests de colisión, se organizan los volúmenes envolventes en una jerarquía de forma que la colisión entre hijos no se comprueba si los padres no colisionan entre sí.



a. *Sphere Trees [Hub96]* **b.** *OBB Trees [GLM96]*

Figura 5.2: *Jerarquías de volúmenes envolventes*

Las jerarquías de volúmenes envolventes (*Bounding Volume Hierarchies, BVH*, en inglés) se aplican tanto a la organización de varios objetos en la escena, de forma que los nodos hoja son el volumen envolvente de cada uno de los objetos, como al refinamiento de una envolvente para un único objeto, de forma que la colisión entre dos objetos se realice enfrentando sus dos jerarquías de volúmenes.

Una característica de las *BVH* presentes en la literatura es que, por lo general, no restringen a que el volumen de los nodos hijos esté totalmente contenido en el del padre, sino que la única restricción es contener a la superficie original del modelo.

Algunas de las jerarquías de volúmenes envolventes existentes son:

- *OBB Trees*[GLM96]. Es una jerarquía binaria construida de forma descendente, donde el plano de división es el que contiene al eje más lárgo del OBB. Las primitivas atravesadas por dicho plano son asignadas a los nodos hijos que contengan los respectivos centroides (Figura 5.2b).

- *AABB Trees* [Ben97]. Se construyen de forma descendente y el plano de división, a lo largo del eje mayor, pasa por la mediana espacial de las primitivas.

- *Sphere Trees*. Se suelen construir con una filosofía descendente, usando un octree como divisor espacial [TC96] o usando como primitivas finales un conjunto de esferas que recubren totalmente la superficie [Qui94] (Figura 5.2a).

- *k-DOP Trees* [KHM+98]. Usando un método descendente, se parte de un k-DOP del conjunto de triángulos y se divide éste en dos partes de forma recursiva. Se detiene la recursión cuando se alcanza un número umbral de triángulos en el nodo.

Una diferencia sustancial entre las *BVH* y los métodos de división espacial, que presentaremos a continuación, es que estos últimos dividen el espacio en zonas disjuntas, mientras que las jerarquías de volúmenes permiten solapamientos entre los distintos nodos de un mismo nivel.

5.1.1.3. Técnicas de división espacial.

Las técnicas basadas en la división del espacio en zonas disjuntas nos permiten discriminar de forma rápida si dos objetos pueden intersecar o no, ya que sólo es posible que colisionen si ocupan la misma zona del espacio.

De una forma resumida, podemos enumerar las técnicas de subdivisión espacial más importantes:

- Las *rejillas regulares* (*grids* en inglés). El *grid* divide el espacio en un número de regiones de igual tamaño. Cada objeto se asocia con las celdas que ocupa, de forma que si dos objetos ocupan la misma celda, entonces se realiza un test de colisión clásico entre ellos. Gracias a la uniformidad del grid, el acceso a cada una de las celdas es sencillo y rápido.

- El *octree*, comentado profusamentee en la sección 2.2.2.1, además de una técnica de representación de sólidos permite realizar una división del espacio que acelera de manera importante la detección de colisiones [RUL00]. El octree, como estructura de división espacial, puede almacenar datos estáticos –las primitivas que forman el escenario– o dinámicos –las entidades que se mueven por el mundo virtual.

- Los *k-d Trees* [Ben75] son una generalización de los octrees, que en lugar de dividir el espacio simultáneamente en las tres dimensiones, lo hace de una en una, eligiendo en cada momento la dimensión y el punto de corte. Un nivel de un octree puede verse como un k-d tree recortado sucesivamente en X, Y y Z (en todo caso dividiendo el espacio por la mitad).

- El *BSP Tree* [FKN80], comentado en la sección 2.2.2.3, es una división binaria del espacio, y fue originalmente ideado para solucionar temas de visibilidad. La arbitrariedad en la orientación de los hiperplanos divisores permite una menor profundidad que con los *octrees* y los *k-d trees*, además de una rápida evaluación de las colisiones.

5.2. Detección de colisiones con el BP-Octree.

El BP-Octree puede enmarcarse dentro de las técnicas de subdivisión espacial para la detección de colisiones, ya que utiliza un octree como índice espacial primario, pero a su vez es una jerarquía de volúmenes envolventes disjuntos. Esta última propiedad, el hecho de que los volúmenes no se solapen, lo hace bastante singular con respecto a otras jerarquías de volúmenes presentes en la literatura ya que ofrece una gran versatilidad.

El hecho de tener acotada la profundidad máxima del árbol, y que la estructura sea capaz de almacenar grandes modelos, nos otorga una gran eficiencia en la detección de inclusiones y distancias a la frontera real del sólido.

En el trabajo que se presenta en esta memoria se han desarrollado los test de inclusión *punto en sólido* y el *cálculo de distancias*, con la idea de su posterior uso en un dispositivo háptico.

5.2.1. Inclusión punto-en-sólido.

Se han propuesto numerosos algoritmos para el cálculo de la inclusión de un punto en un sólido, ya que éste es uno de los problemas clásicos en Informática Gráfica. El test de

inclusión *punto en sólido* consiste en determinar si un punto está sobre la frontera, en el espacio interior delimitado por dicha frontera o fuera del mismo.

Existen varias propuestas para la resolución de este problema, normalmente cada una adaptada a un esquema de representación concreto. Entre los algoritmos de inclusión que dan resultados no exactos, esto es, con posibles falsos positivos o negativos en el entorno de la superficie, encontramos los basados en grids y otras estructuras espaciales, como los octrees. Su naturaleza discreta hace imposible determinar con exactitud la posición de un punto muy cercano a la frontera del sólido. Si queremos respuestas exactas y totalmente fiables a la consulta, hay que estudiar algoritmos basados en el Teorema de la Curva de Jordan [Jor87, HAM02], o los basados en símplices [FT97, OSF05].

En nuestro caso, el hecho de tener los nodos del octree codificados con su ruta desde la raíz, y que los nodos hoja tengan etiquetadas sus esquinas nos permite realizar el test de una forma muy rápida y eficiente. Se puede decir que combinamos la eficiencia del octree con la exactitud del teorema de Jordan.

En la figura 10 se muestra el pseudocódigo del algoritmo. En primer lugar, dado un punto p se calcula su *octcode*, o lo que es lo mismo, el nodo hoja de máximo nivel en el que estaría si dicho nodo hoja existiese. Con este *octcode*, se recorre el árbol desde la raíz hasta que, nos encontramos con un nodo *blanco* (fuera), *negro* (dentro) u *hoja*.

Sólamente en el caso de llegar a un nodo hoja, de último o superior nivel, se realiza un test clásico de intersección segmento-poliedro basado en el Teorema de la Curva de Jordan [O'R94]. Se construye un segmento $S\{p,C_n\}$ siendo C_n una de las esquinas del nodo etiquetadas como "dentro"(ver figura 5.3). De esta forma, se cuenta el número de veces que el segmento atraviesa la superficie y dará como resultado la posición de p: si es par, el punto queda dentro (recordemos que un extremo del segmento está dentro del sólido), si es impar, se considera fuera.

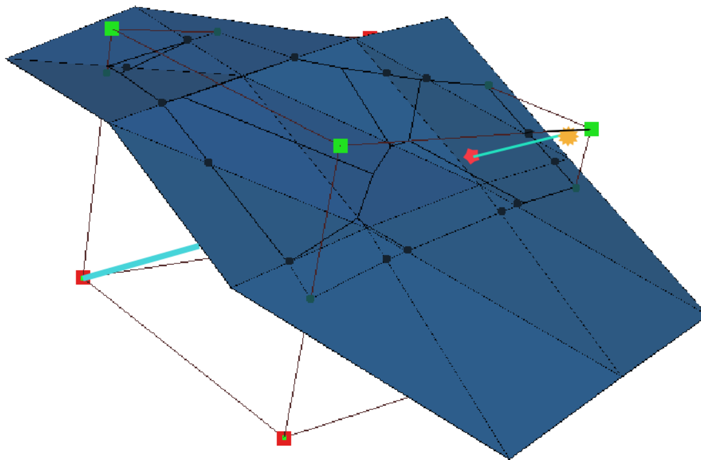


Figura 5.3: *Intersección segmento –cyan– con sólido –azul–*

En la tabla 5.1 mostramos los tiempos obtenidos en las pruebas realizadas. Dichas pruebas se han realizado lanzando diez millones de puntos aleatorios dentro de la caja envolvente del sólido, de forma uniforme. Se aprecia claramente como la influencia del tamaño del modelo es mínima. En los tiempos mostrados en la columna *modo directo* no se hace uso de los planos

```
bool inside(Point p, BPOctree bpo){
        OCTCODE_T code=bpo.octcode(p);
        node=bpo.root();
        int level=0;
        while (isGray(node)) {
                c=bpo.getChildIndex(level);
                node=node.getChild(c);
        }
        if (isBlack(node))
                isinside=true;
        if (isWhite(node))
                isinside=false;
        if (isLeaf(node))
                isinside=insideLeaf(p,node);
        return isinside;
}

bool insideLeaf(Point p, LeafBPONode node) {
        Point c=getCornerInside();
        Segment s(p,c);
        int nrI=0;
        nrI+=nrFacesIntersect(s,node.Faces);

        if (nrI MOD 2 == 0)
                isinside=true;
        else
                isinside=false;

        return isinside;
}
```

Algoritmo 10: *Test de inclusión punto en sólido.*

envolventes en niveles superiores ni siquiera en los nodos hoja, puesto que al tener las envolventes tan ajustadas a la superficie real del objeto, los puntos interiores serían doblemente comprobados –con los planos de la envolvente y posteriormente con la geometría real– y el test de intersección segmento-triángulo no es computacionalmente caro como para evitarlo a toda costa. Se puede apreciar que la diferencia de uno a otro método oscila alrededor del $\pm 2{,}0\,\%$.

Mostramos en la figura 5.4 los puntos interiores detectados por el algoritmo para algunos modelos.

En la figura 5.5 podemos apreciar gráficamente como el número de polígonos no afecta en el tiempo necesario para realizar una consulta. Si nos fijamos en los modelos que más tiempo requieren, coinciden con aquellos que tienen más concavidades, por lo que podemos deducir que la complejidad de la superficie del objeto sí es un factor determinante. En cualquier caso, los tiempos obtenidos son sensiblemente inferiores a los que se encuentran en la literatura, como los mostrados en la tabla 5.2. Nótese que los algoritmos publicados en [Ogá06] están implementados en GPU, mientras que nuestro método tan sólo hace uso de un procesamiento puro en CPU. Sólo se incluyen los modelos Armadillo y Stanford Dragon por ser los utilizados tanto en las pruebas del presente trabajo como en las de [Ogá06].

Las dos primeras columnas con tiempos son sendas implementaciones optimizadas del algoritmo presentado en [FT97]. Es un algoritmo basado en símplices que clasifica puntos en poliedros genéricos. En la columna del BSP se aprecia cómo los autores no pudieron trabajar con grandes modelos, puesto que faltó memoria. En el modelo armadillo, los tiempos se acer-

Modelo	*Poligonos*	*Tests/seg* Con envolventes	*Tests/seg* *Modo directo*	Directo/Envolventes
Bunny	71.040	2.840.909	2.747.253	96,70 %
Golf	100.243	2.202.643	2.173.913	98,70 %
Armadillo	150.000	3.424.658	3.521.127	102,82 %
Egipcio	161.909	2.369.668	2.314.815	97,69 %
Teeth	233.204	1.879.699	1.845.018	98,15 %
Fertility	483.226	2.793.296	2.777.778	99,44 %
Angelo	674.764	4.098.361	4.166.667	101,67 %
Phlegmatic Dragon	715.933	3.164.557	3.144.654	99,37 %
Stanford Dragon	871.414	2.941.176	2.923.977	99,42 %
Happy Buddha	1.087.716	2.336.449	2.283.105	97,72 %
Blade	1.765.388	2.380.952	2.347.418	98,59 %
Dragon	7.218.906	5.376.344	5.747.126	106,90 %

Tabla 5.1: *Resultados test de inclusión punto en sólido.*

Modelo	Polígonos	Ogayar-FT (GPU)	Ogayar-FT (GPU+Tetragrid32)	BSP	Jordan Oct6	BP-Octree
Armadillo	150.000	840	27	0,00043	0,18	0,00028
Stanford Dragon	871.414	5.219	87	Stack overflow	1,19	0,00034

Tabla 5.2: *Comparativa de resultados test de inclusión punto en sólido (tiempo medio de un test en milisegundos).*

can considerablemente a los presentados en este trabajo. El algoritmo de Jordan optimizado devuelve valores bastante buenos, aunque sin llegar a la calidad de BSP y BP-Octree.

5.2.2. Detección de distancias y direcciones de colisión.

La detección de colisiones entre diversos objetos se puede realizar mediante el uso de mapas de distancias. Los vértices de un modelo se comparan con el mapa de distancias del otro objeto y viceversa. En general, un mapa de distancias de una superficie S es una función escalar

$$D : \mathbb{R}^3 \longrightarrow \mathbb{R}, D(p) = \min_{q \in S}\{|p - q|\}, \forall p \in \mathbb{R}^3 \tag{5.1}$$

Si además la superficie es cerrada, se puede definir una función signo tal que

$$sign(p) = \begin{cases} -1 & si\, p\, está\, dentro \\ 1 & si\, p\, está\, fuera \end{cases} \tag{5.2}$$

de forma que junto con la función de distancia nos devuelva un valor de distancia signado.

Se detecta colisión si $D(p) < 0$ para algún punto p, y se puede detectar la cercanía al objeto si $D(p) < \epsilon$.

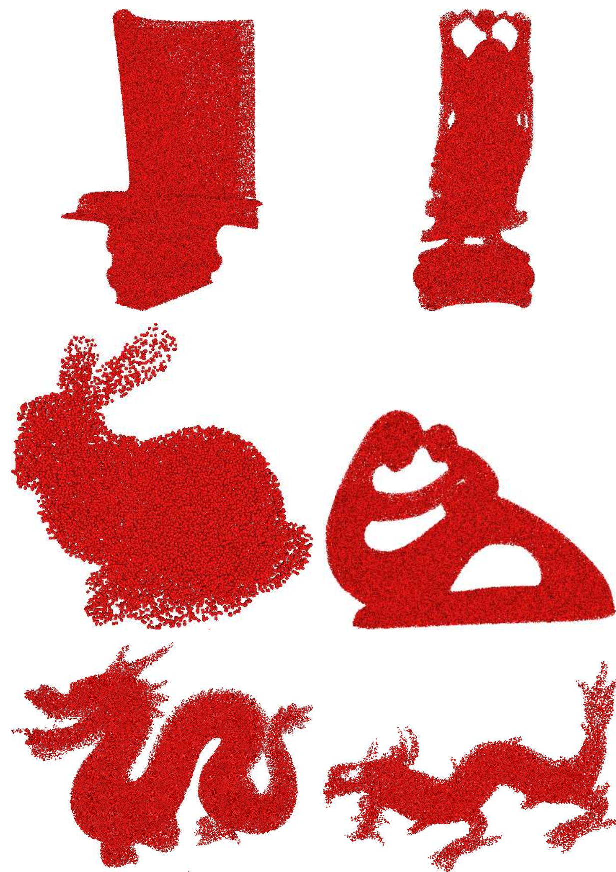


Figura 5.4: *Resultados visuales del test de inclusión punto en sólido.*

En general, hay tres grandes líneas de trabajo para construir un mapa de distancias de forma eficiente: métodos basados en diagramas de Voronoi [BM01, ICK+99], métodos de propagación de distancias [Set96] y métodos que usan árboles y grids. Respecto a estos últimos, en la literatura podemos encontrar mapas de distancia almacenados en grids tridimensionales [FSG03], de forma que se interpolan los valores para posiciones intermedias. También se encuentran mapas de distancia adaptativos (ADFs, [FPRJ00]) que son unas estructuras multirresolución (octrees) que se refinan hasta conseguir un nivel de detalle adecuado, o incluso el uso de *BSP-Trees* [WK03].

En en nuestro caso, utilizaremos el octree como índice espacial para localizar de forma rápida el punto de forma relativa al objeto con el cual queremos calcular la distancia, de forma análoga a cómo se obtiene el valor de inclusión. Calculamos el *octcode* del punto, lo cual nos direcciona directamente a un nodo n. En el caso de que n sea un nodo hoja, se busca la distancia al triángulo más cercano, y se devuelve tanto el valor signado como la normal de dicho triángulo. Si n es un nodo blanco o negro, tenemos varios criterios:

1) *Modo recursivo.* Ascender hasta el ancestro *gris* del nodo, A_n, y realizar un cálculo de la distancia en todos los nodos hoja que descienden de A_n mediante un algoritmo

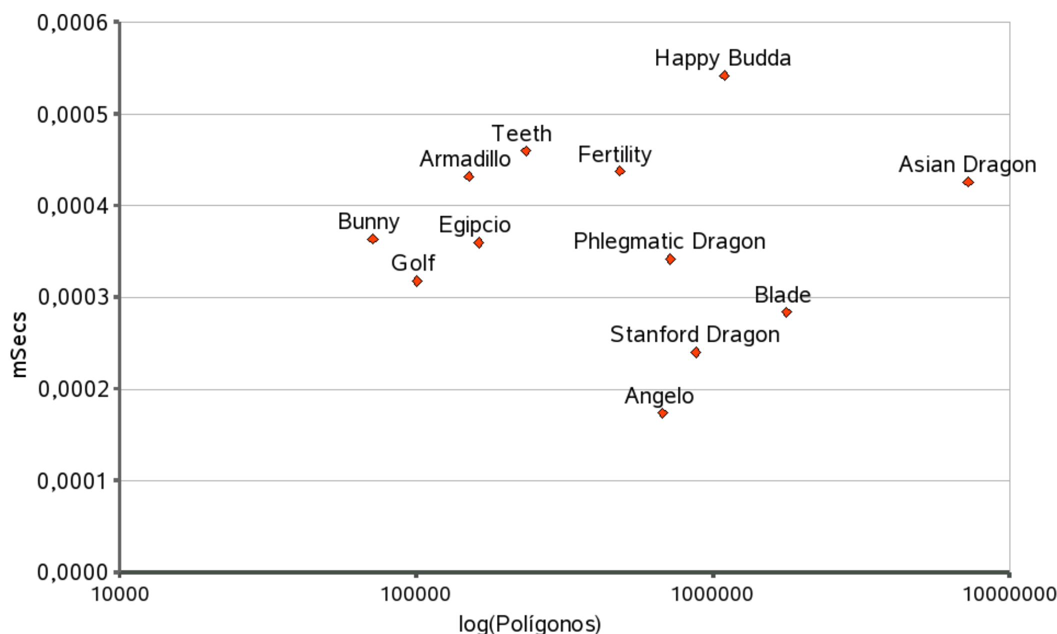


Figura 5.5: *Milisegundos necesarios para un test punto-en-sólido*

recursivo descendente. Ésto nos proporcionará la distancia al triángulo más cercano de entre los que atraviesan los nodos hoja que comparten el mismo ancestro gris A_n.

2) *Nodo de nivel superior*. Ascender hasta el ancestro *gris* del nodo, A_n, y realizar un cálculo de la distancia a los planos que forman la envolvente convexa en dicho nodo A_n. Obviamente, ésto nos dará un resultado "conservador", en el sentido que cuando estemos *fuera* del modelo, nos dará un valor de distancia menor que el real, y si estamos dentro, nos indicará que estamos más lejos de la frontera de lo que en realidad está el punto p, ya que las envolventes de un nodo gris tienen un mayor volumen que las suma de los volúmenes de sus hijos, es decir, tiene los planos más alejados de la superficie real.

3) *Nodo de nivel superior acotado*. Una variante del anterior es limitar el ascenso por el árbol hasta un nivel máximo, L, de forma que si en dicho nivel, el punto sigue estando en un nodo blanco o negro, se descarta. Ésto nos permite limitar el error, puesto que con un $L = 4$, estaríamos evaluando tan sólo una franja de grosor $1/16$ del ancho total del modelo en cada dimensión.

4) *Sólo hojas*. El criterio anterior puede refinarse aún más ignorando el cálculo si el nodo ancestro A_n es gris, es decir, teniendo en cuenta sólo los valores de los nodos hoja hasta un nivel determinado, L.

5) *Búsqueda en entorno*. Realizar un desplazamiento por todos los nodos de último nivel en un entorno delimitado de p (3-entorno, 5-entorno), para ver si se encuentra un nodo

hoja que nos proporcione un valor exacto de distancia. Esto funcionaría si el árbol estuviese completo con todas las hojas en el mismo nivel, pero no es el caso, ya que se produce una compactación de nodos hoja para reducir el espacio.

Como se puede deducir, el algoritmo devuelve un valor de distancia que es exacto en una estrecha franja en torno a la superficie, y debe ser considerado aproximado cuanto maś nos alejamos de ésta, puesto que lo más probable es que se haya recurrido a la información de los nodos grises.

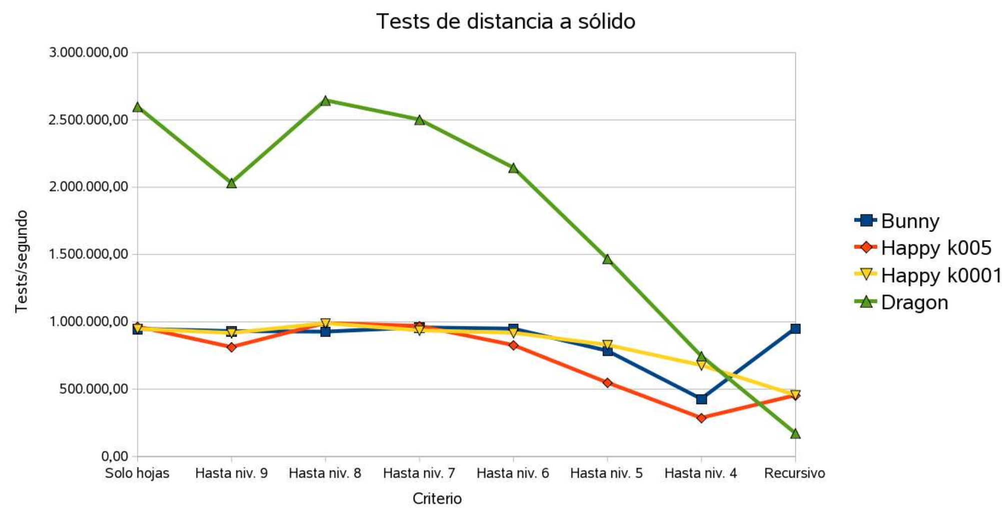


Figura 5.6: *Velocidad test distancia a solido*

Se han realizado pruebas con tres de estos criterios: el modo *recursivo*, teniendo sólo en cuenta los nodos hoja y ascendiendo por el árbol hasta encontrar un ancestro (con limitación de nivel). Los resultados se muestran en la figura 5.6. Podemos apreciar cómo claramente el algoritmo más lento es el recursivo, y los que ascienden por el árbol hasta encontrar un nodo ancestro tienen un comportamiento que empeora cuanto más alto se pone el límite.

En la figura 5.7 se aprecia que, al igual que ocurre con el test de inclusión, el tamaño del modelo no es un factor determinante, aunque se aprecia como el rendimiento mejora con el aumento del número de polígonos.

Otro detalle interesante es que los mejores rendimientos se obtienen con el modelo más grande, el Dragon de 7 millones de polígonos (figura 5.9). Ésto es debido al gran número de nodos del árbol, lo que hace que esté muy completo y haya muchos nodos hoja en los últimos niveles, lo que implica muy pocas búsquedas de ancestros. Por otro lado, se aprecia como el factor k influye también en esta aplicación del BP-Octree, puesto que el modelo *Happy Buddha* (ver ejemplo en la figura 5.8) obtiene mejores resultados en el uso de nodos grises hasta niveles 4 o 5 con un k menor.

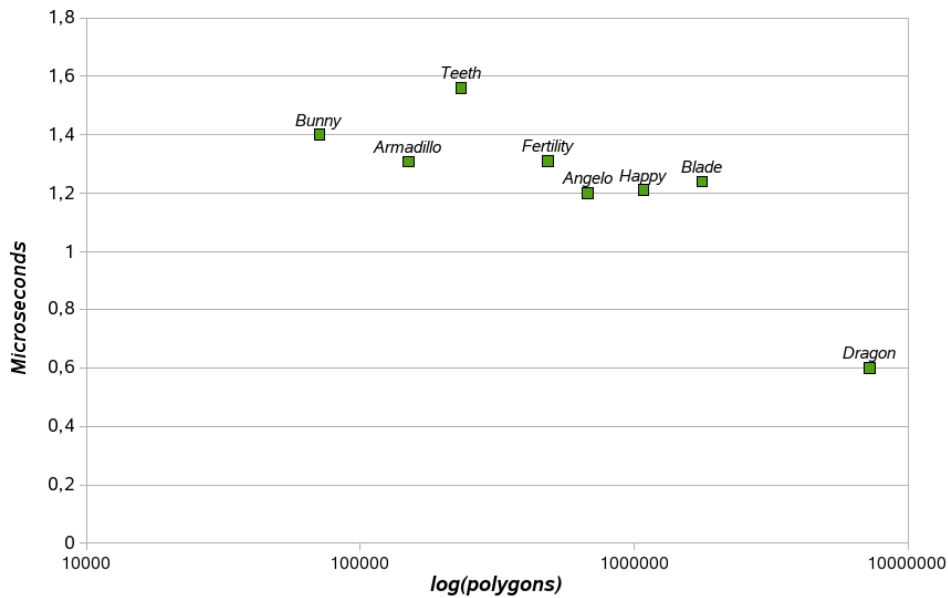
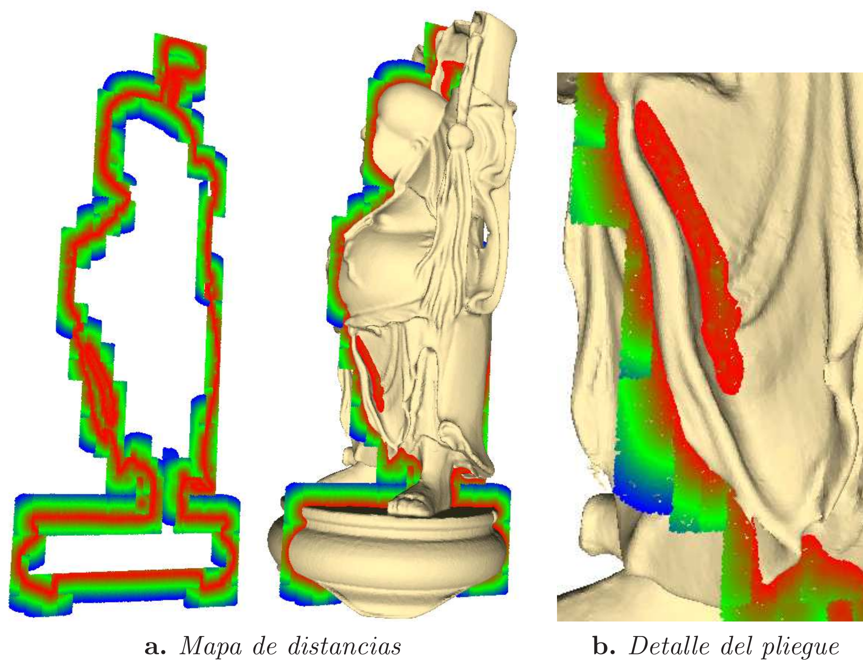
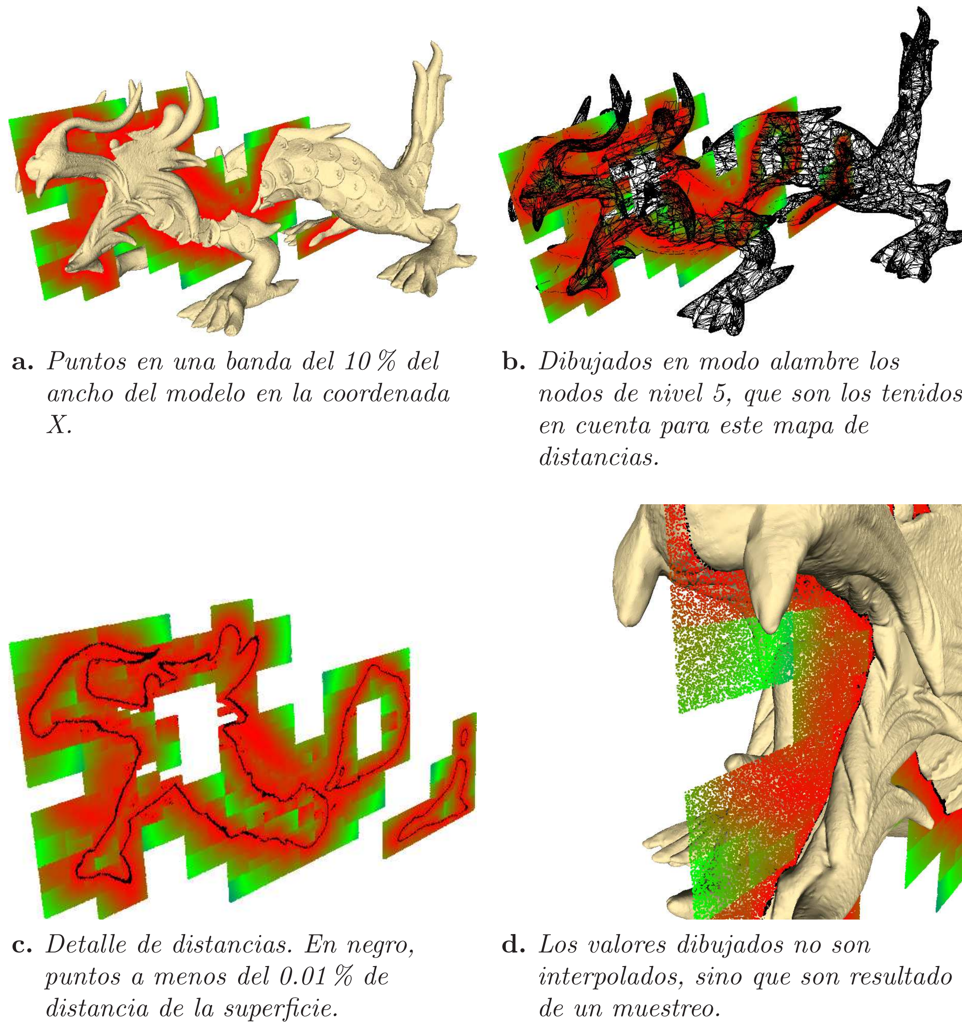


Figura 5.7: *Tiempo de respuesta por consulta en función del número de polígonso.*



a. *Mapa de distancias* **b.** *Detalle del pliegue*

Figura 5.8: *Mapa de distancias para el modelo Happy Buddha.*



a. *Puntos en una banda del 10 % del ancho del modelo en la coordenada X.*

b. *Dibujados en modo alambre los nodos de nivel 5, que son los tenidos en cuenta para este mapa de distancias.*

c. *Detalle de distancias. En negro, puntos a menos del 0.01 % de distancia de la superficie.*

d. *Los valores dibujados no son interpolados, sino que son resultado de un muestreo.*

Figura 5.9: *Mapa de distancias para el modelo Asian Dragon, de 7M polígonos.*

Figura 5.10: *Interfaz háptico de seis grados de libertad*

5.2.3. Aplicación del esquema para interacción háptica.

Para realizar una comprobación empírica de los datos anteriormente indicados, se ha desarrollado un simple prototipo de aplicación háptica, usando un interfaz háptico de 6 grados de libertad, como el mostrado en la figura 5.10. Este prototipo básico simula el proceso de pintar sobre la superficie del modelo, de forma análoga a lo publicado en [GEL00, KSD03]. El puntero del dispositivo háptico se comporta como un rotulador que, al contacto con la superficie del modelo virtual, libera una cantidad de tinta sobre el objeto. Cuanto más fuerza se aplica sobre la superficie, mayor es la gota liberada. Además, el punto que se marca en la superficie tiene la orientación de la normal que devuelve el algoritmo, así que el resultado es de puntos de tinta completamente adheridos a la superficie.

Se ha incluído también al puntero virtual la posibilidad de que muestre mediante una escala de color la cercanía a la superficie, mostrándose en rojo cuando está en contacto con el modelo.

La figura 5.11 muestra unas capturas de pantalla de la aplicación ejecutándose en un Pentium IV 3.2GHz, 1GB RAM, con una tarjeta gráfica Nvidia Quatro 1400. En todo momento, la frecuencia de refresco de la hebra háptica es de 1kHz.

Durante las pruebas realizadas, se ha dado la situación de que la hebra háptica detectaba colisiones con el modelo *Asian Dragon*, de 7 millones de polígonos, y sin embargo, la tarjeta apenas soportaba la visualización del mismo a 0.2 frames por segundo. Se muestra una captura de pantalla en la Figura 5.12.

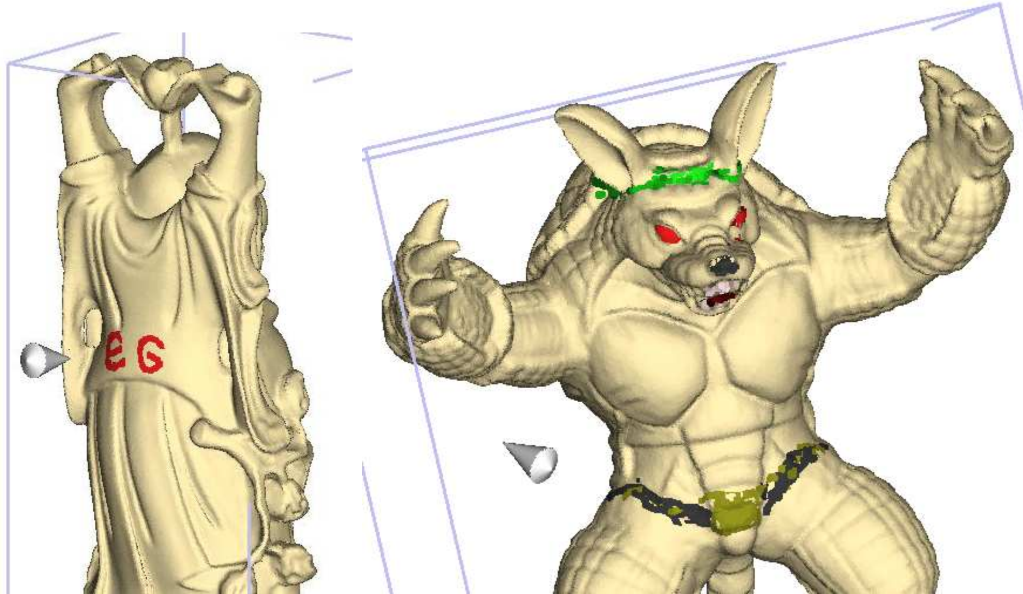


Figura 5.11: *Aplicación de pintura háptica. Graffiti sobre los modelos Happy Buddha (1M triángulos) y Armadillo (150K triángulos).*

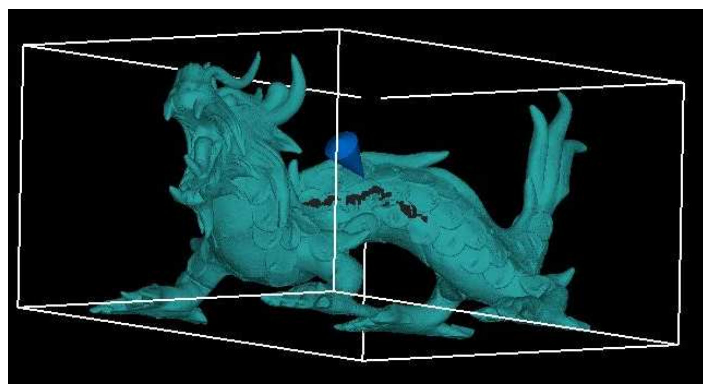


Figura 5.12: *Interacción háptica con el modelo Dragon de 7M de triángulos.*

CAPÍTULO 6

Conclusiones y trabajos futuros.

6.1. Conclusiones y principales aportaciones.

En la presente memoria hemos tratado de resumir de forma clara y concisa el trabajo realizado durante los últimos años y los resultados obtenidos a partir del mismo [MCT04, MCT05, MCT07, MCT08]. El BP-Octree y los algoritmos desarrollados sobre esta estructura de datos han proporcionado una serie de prestaciones que lo hacen muy atractivo para aplicaciones de visualización remota e interacción háptica. Las principales aportaciones de este trabajo son:

1) Se ha presentado un esquema de representación de sólidos jerárquico basado en volúmenes envolventes que permite realizar una visualización progresiva del modelo, así como servir de estructura para una detección rápida de inclusiones de punto en sólido y de distancias al objeto.

2) Se ha aplicado un algoritmo de indexación espacial rápida de polígonos para la localización de puntos en el volumen del octree, permitiendo una rápida creación de los nodos hoja y grises del árbol. Se han conseguido crear de una forma eficiente y rápida los nodos *negros* y *blancos* que permiten clasificar cualquier punto contenido en el volumen del nodo raíz del árbol.

3) Además, se han obtenido buenos resultados en la aplicación de un algoritmo de *clustering* para la eliminación de planos redundantes, sin que el volumen contenido por las envolventes se vea afectado.

4) Se ha desarrollado un algoritmo de visualización progresiva multirresolución del esquema de representación presentado, así como otro algoritmo de visualización adaptativa del mismo, mejorando la apariencia de los modelos a alto nivel con el uso de impostores.

5) El BP-Octree ha proporcionado muy buenos resultados en la clasificación de puntos con respecto a la frontera de un sólido, así como en el cálculo de distancias a dicha frontera.

Los datos obtenidos mejoran en varios órdenes de magnitud los publicados en artículos recientes por otros autores, a pesar de haber sido implementado esta estructura íntegramente en CPU.

6) Se ha desarrollado un prototipo de aplicación usando un dispositivo háptico y se han manipulado modelos de hasta 7 millones de polígonos en tiempo real, respondiendo adecuadamente a las colisiones del brazo robótico.

6.2. Trabajos futuros.

Obviamente, este trabajo no es más que el comienzo de una línea de investigación, que ha ido descubriendo nuevas vías de desarrollo que pasamos a comentar brevemente:

- *Desarrollo de un algoritmo de detección de colisiones sólido-sólido.* Se implementará un algoritmo para la detección de colisiones entre dos objetos complejos, usando el BP-Octree como estructura jerárquica auxiliar. Para ello, será necesario almacenar en disco la geometría calculada para cada uno de los nodos intermedios, de forma que se minimice el tiempo de carga y cómputo.

- *Implementación en GPU.* Se realizará una implementación en GPU del algoritmo de inclusión punto en sólido, para evaluarlo con otros algoritmos GPU que pudieran aparecer para el mismo o similar propósito.

- *Mejora de la visualización.* Realizar una mejora en la visualización del modelo multirresolución. Quizá realizando una propagación de las normales originales del modelo o usando la geometría obtenida en cada nodo para realizar un remuestreo global.

- *Modelos deformables.* Estudiar la viabilidad del uso del BP-Octree para modelos deformables.

APÉNDICE A

English summary

A summary of this Thesis is included as partial fulfillment of the requirements for the European Doctorate Award

In this work we present the BP-Octree as a novel multiresolution volume hierarchy, which can be used both for rendering and collision detection, by either standalone or distributed applications.

Our proposed data structure uses an octree, assigning to each node a set of planes that define a convex bounding volume of the part of the model contained in that node (see a detailed node in figure A.1, and two different bounding levels in figure A.2). These planes are restricted to either face planes or planes parallel to faces in order to avoid generating more geometric information. This is why it is named *BP-Octree* (Bounding-Planes Octree).

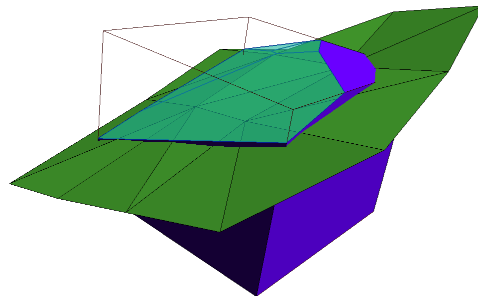
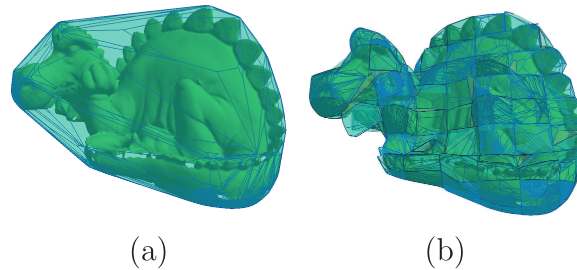


Figure A.1: *Example of node. Green, model geometry; cyan, computed bounding.*



(a) (b)

Figure A.2: *Phlegmatic Dragon and its bounding planes (cyan) at root node (a) and at level 4 of the BP-Octree (b).*

A.1. BP-Octree. Data Structure Overview

As mentioned above, we store a set of planes at each node. The halfspaces defined by these planes when intersected with the cell at that node, completely enclose the part of model contained in the octree cell. Finally, leaf nodes contain the geometry information proper. Pseudocode 11 shows the pseudocode describing this data-structure, which is schematized in Figure A.3. Figure A.4 shows the root node of the BP-Octree for several models.

```
typedef long int Index;
typedef struct {
      Index planeIndex;
      double d;
} BPlane;
typedef int Octcode; // 4 bytes long

class BPNode {
     vector<BPlane> boundingPlanes;
     Octcode oct;
}

class BPLeafNode:public BPNode {
     vector<Index> faces;
};
```

Algoritmo 11: *Basic Data Structure of our BP-Octree*

We would like to re-use planes defined by model polygons as far as possible.While this works well for convex geometry, non-convex models require additional planes. In the latter case, we use planes parallel to model faces (field d of the `BPlane` structure stores the plane-to-face distance).

To save space, we eliminate redundancies by storing planes in a separate data structure and access them via indices from octree nodes. The same idea applies to the final geometry stored at leaf nodes. When using large models, this approach makes it easier to keep the data structure in main memory and to access both planes and geometry information via external files, delegating to the operating system the task of caching.

When designing the data structure, several problems required our attention to arrive at a complete and correct solution:

- *Management of large datasets.* Our data structure should be able to deal with large models. Besides optimizing space consumption we wanted to reduce building time. To this end, we index polygons spatially making it easier to address and classify them and also to manage large datasets via external memory.

- *Guarantee that every polygon contained in the node is contained in its bounding volume.* As the octree is a discrete structure, we need to assign continuous primitives (polygons) to voxels, and do it quickly enough. The obvious solution would be to clip all polygons to fit them exactly into our three-dimensional voxels, but this would lead to prohibitive computation time. We use a 3DDDA algorithm [FTI88] to traverse each polygon and determine which voxels contain it.

- *Ensure that bounding volumes at level* n *are completely contained in the upper bounding volume (at level* n-1). It is important to maintain coherence between levels, because it makes no sense to obtain looser bounds at more detailed levels of the BP-octree. To achieve this objective, we compute bounding volumes from real geometry at leaf nodes, while for internal nodes we compute them using the bounding planes of their children.

- *Avoid cracks between adjacent nodes.* As we use approximated bounding, it is almost impossible that two adjacent nodes get identical bounding planes at their common face, which would result in holes showing up in the rendered model. We add a set of fictitious planes, which are never transmitted, corresponding to the node bounding box faces that are still visible (see blue polygons in figure A.1).

- *Which plane to use when polygons in the node create a concavity?.* When node geometry is not purely convex, it is rather complex to find a face whose supporting plane bounds

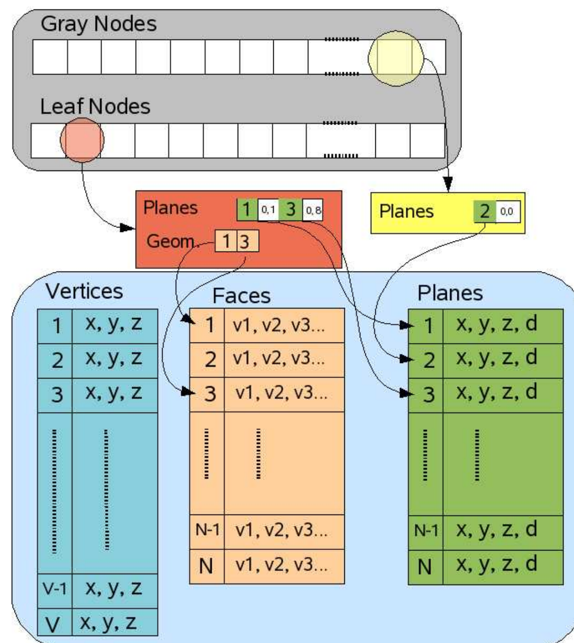
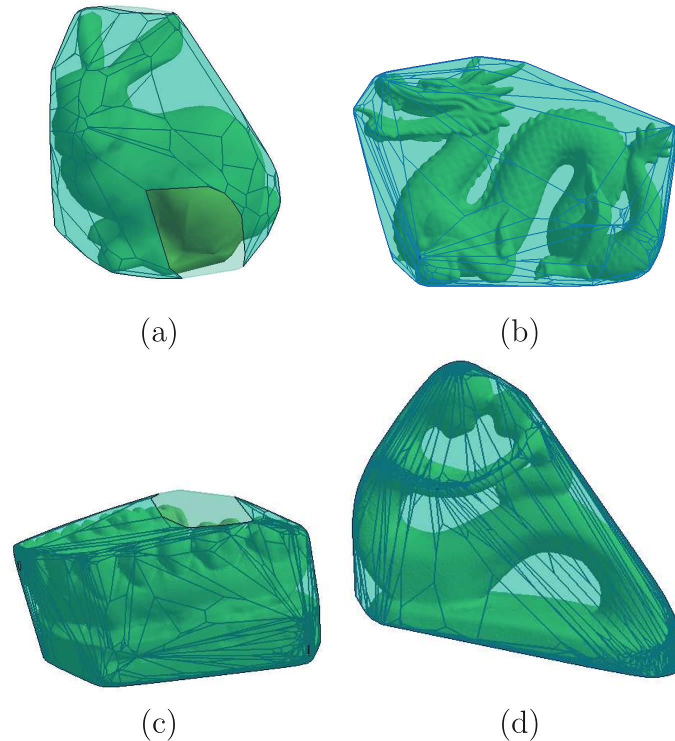


Figure A.3: *General Scheme of BP-Octree data structure.*



(a) (b) (c) (d)

Figure A.4: *Boundings at level 0.*

the whole geometry. In these circumstances, we allow these planes to be translated along its normal direction in order to become a bounding plane. We name them *displaced planes.*

- *Convex meshes get less simplification rates than irregular ones.* If we have a sphere-like object, all its face planes satisfy the bounding criterion, so we would obtain as bounding volume the sphere itself. To avoid such situations, we limit the number of planes at each level, which are selected through statistical criteria. An example of this situation can be found in figure A.5

A.2. Building the BP-Octree

One important aim of this work is to build the whole data structure in a reasonable computation time, although this is done just once per model. This is carried out by a bottom-up approach: first, we classify the polygons in a three-dimensional grid which is the deepest level of the octree; then, we assign these polygons to each leaf node and group all leaf nodes in order to have a minimum number of planes per node. After selecting the bounding planes at leaves, we build internal nodes by tracing the path from the root node to each leaf, computing their bounding set of planes by a bottom-up recursion.

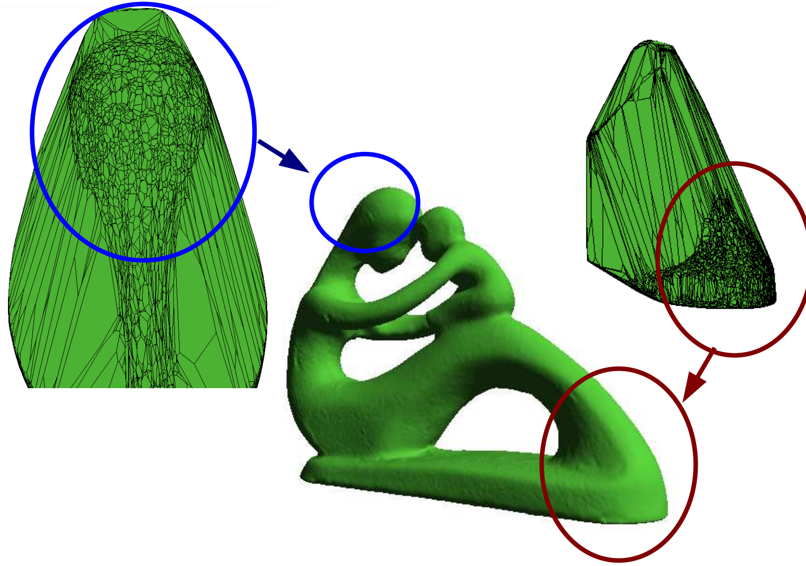


Figure A.5: *High density of planes due to strong convex bounding areas of the model.*

A.2.1. Spatial Indexation

Since the octree is a discrete data structure, using as the root node the *Axis Aligned Bounding Box* [Ben97] of the model, we need to discretize our model. We use *Morton codes* [Mor66] to locate every point in the octree space, and hence to identify each node (i.e. each node is identified by the *octcode* assigned to any point inside its volume).

A.2.1.1. Morton Code

For a given point p and each dimension D, and for an octree node at level l the discrete coordinate N_D is computed by the following formula:

$$N_D = floor(\frac{2^l}{w_d}(p_d - d_{min})) \tag{A.1}$$

where w_d is the length of the root node at that dimension, d_{min} is the minimum value of the octree node for the dimension D and p_d is the coordinate at dimension D of the point p.

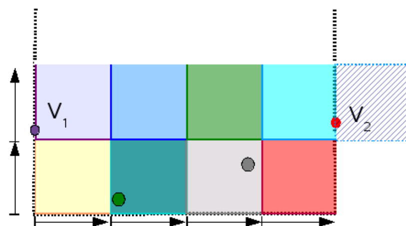
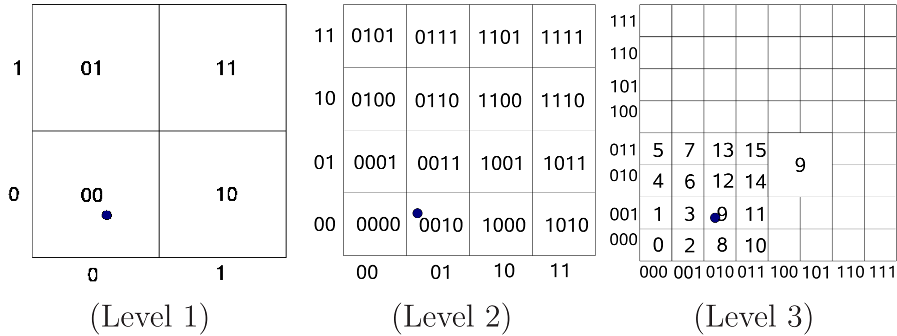


Figure A.6: *Point V_2 is addressed in a wrong voxel.*

Equation A.1 fails when $p_d = d_{max}$, as shown in Figure A.6, because intervals are right-open. with the aim of solving this there are two choices: check if $p_d = d_{max}$ and close the interval, or make the root voxel a bit wider than the model, thus avoiding these special cases.



(Level 1) (Level 2) (Level 3)

Figure A.7: *Morton-code based nodes numbering*

Figure A.7 shows Morton Codes in two dimensions. For each node, X and Y coordinates are interleaved, resulting in an integer number, which is unique for each voxel at its level. However, Morton Codes are not unique across all levels. Indeed, we can see that the number 9 can identify cells in the second or third level, which differ both in size and location. To solve this ambiguity, we append the level that the code belongs to, also in binary. This converts the Morton Code into a locational code, the *octcode*. There are two options:

- Append the level bits as the most significant. (figure A.8a). This induces a breadth-first ordering of the octcodes.

- Append the level bits as the least significant (figure A.8b). This causes a depth-first ordering of the octcodes.

In our approach, we chose the breadth-first ordering of the octcodes, because this order preserves spatial coherence which is useful when storing values to a file and allows a progressive transmission of the model.

A.2.1.2. Defining Maximum Depth

Given the locational code expressed in the previous section, we can easily calculate the memory required to store each voxel. The octcode can be split into two parts: the level and the cell index related to that level (figure A.8). Each depth level l adds three bits to the cell index (so it is $3l$ bits), and level l requires $log_2 l$ bits.

Therefore, a four byte long code allows indexing $2^{27} = 134217728$ leaf nodes, i.e. slightly over 134 million leaf nodes. From our experience, models over 1.5M polygons have never used more than 5 % of of these nodes. So we could assume that 4 bytes, a `int` in most of the implementations, is a safe code length for most current models.

level	code

Level 1, node 0: 01 000000 (64)
Level 1, node 1: 01 000001 (65)
Level 2, node 0: 10 000000 (128)
Level 2, node 1: 10 000001 (129)

(a)

code	*level*

Level 1, node 0: 000000 01 (1)
Level 1, node 1: 000001 01 (5)
Level 2, node 0: 000000 10 (2)
Level 2, node 1: 000001 10 (6)

(b)

Figure A.8: *Morton-based locational code: a) Depth-first ordering; b) breadth-first ordering*

A.2.1.3. Addressing Polygons

We index every triangle by identifying the cells of the discrete grid that it traverses. The discrete grid is defined by the deepest level of the octree, which in this case is fixed to 9 because it can be addressed with only 31 bits (4 to identify the level and 27 for the location itself), and a four bytes integer has been proved to be enough for handling models up to 10M triangles.

We apply an exhaustive 3DDDA algorithm over each triangle, intersecting it by all node-separating planes contained in its bounding box, as is shown in Figure A.9. Each segment/plane intersection returns a point, which is located using the *octcode*, and therefore the triangle is assigned to the node addressed by that *octcode* and also to the surrounding nodes, avoiding any miss that would lead to inconsistency (as the one shown in Figure A.10). The 3DDDA algorithm ensures that all nodes spanned by a polygon will use its information to compute their bou

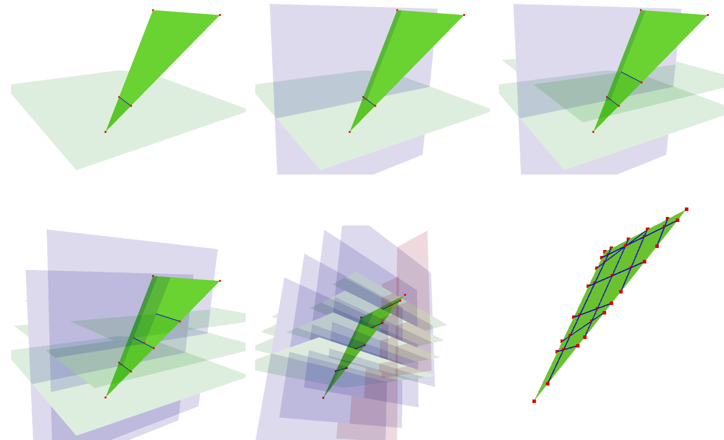


Figure A.9: *Triangle voxelizing algorithm. Blue: segments produced by intersection with a plane P. Red: points used to detect traversed nodes.*

If it were not necessary to have at each node a bounding volume of the model, it would be faster and less memory consuming to select just a small subset of these nodes, e.g. by selecting only the nodes actually containing polygon vertices.

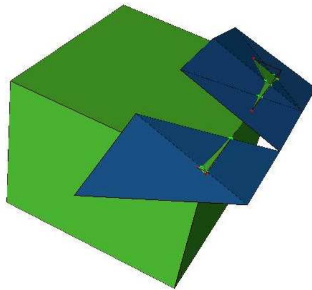


Figure A.10: *A wrong triangles -blue- voxelization leads to surface inconsistencies in nodes.*

A.2.2. Creation and coalescing of leaf nodes.

After the triangles indexation, we have defined absolutely the leaf nodes at the maximum depth of the octree. But such depth is not necessary unless we have very large models, so we coalesce sibling leaves until a defined criterion is reached. In this work, leaves have been merged while the new leaf contains less than 50 triangles. Table A.1 shows as the average values moves around 20 triangles per leaf.

Level	Bunny	Armadillo	Happy Buddha	Dragon
0	-	-	-	-
1	-	-	-	-
2	9,00	19,00	-	-
3	19,41	16,00	23,60	-
4	19,17	19,78	18,54	20,71
5	18,12	19,06	20,23	18,23
6	12,10	16,90	17,96	18,72
7	7,85	10,12	15,59	18,84
8	-	7,07	11,16	19,12
9	-	-	9,26	19,21

Table A.1: *Average triangles per leaf*

By applying these methods we can achieve a significant reduction in the number of leaf nodes, up to 85 %. Furthermore, choosing a particular criterion only affects the lower levels of the octree, and is not relevant at top levels. Additionally, applying the selected criterion yields an average of about six polygons per leaf node across all models in our experiments.

After creating leaves, it is possible to recreate the internal tree nodes needed to reach a given leaf from the octree root (Figure A.11) by examining its *octcode* as depicted in [Gar82].

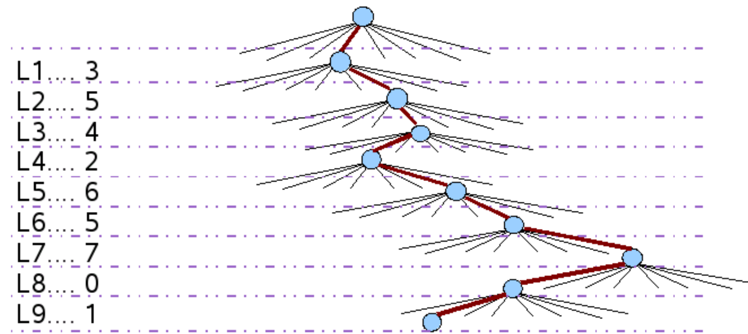


Figure A.11: *Gray nodes created to reach leaf* $17|354265701_8$

A.2.3. Computing Bounding Volumes

At each node we store a set of planes that completely bounds the portion of the 3D model contained in that node. To compute this set, we apply a recursive bottom-up procedure, applying a divide-and-conquer paradigm such that only a small number of operations are required at each node.

```
computeBounding(Node_T node){
 if isLeaf(node)
  node->selectBoundingPlanes();
 else {
  for each ch child of node {
   computeBounding(ch);
   node->addBPlanes(ch.getBPlanes());
   node->addBVertices(ch.getBVertices());
  }
  node->selectBoundingPlanes();
 }
}
```

Algoritmo 12: *Computing recursively the bounding at each node*

As is briefly described in figure 12, at each node we select the planes that bound the geometry of its children (or the real geometry of the model contained in the node if we are at a leaf) using the `addBVPlanes` method. Furthermore, only planes belonging to bounding volumes of its descendants are used to compute the current node bounding volume. Additionally, only bounding vertices of its descendants (`addBVVertices`) are used to guarantee that the size of bounding volumes never decreases as we climb the octree.

We apply a simple method to compute the bounding volume as Figure A.12 illustrates in two dimensions. For each node n we create a set of *candidate* planes, which are those selected as bounding planes at children of n (or the original geometry in case n is a leaf node). Then, we test sequentially every candidate plane against contained bounding vertices (V_2, V_3 and

V_4) and the intersecting points between edges of the node and the triangles, or edges of the triangles and the faces of the node (CP_1 and CP_2). The plane is taken as it was in the child node, using the displacement value if any. If all vertices lie inside or on the plane, i.e., signed distance is less or equal to zero, the plane can be added to the bounding volume (BV) as it is –this is the case of plane A in figure A.12.b, which is the only plane that bounds all the bounding points. The resulting BV is depicted in blue in Figure A.12.b.

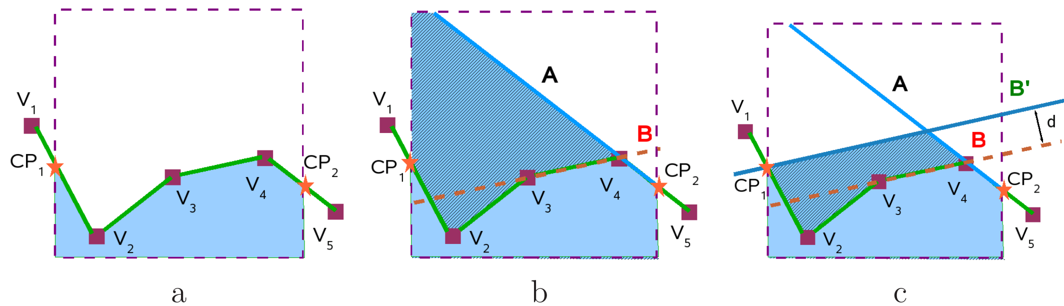


Figure A.12: *Selecting bounding planes.*

As we search for a closer BV, we can insert an extra plane B' into the set of bounding planes, which is the plane B displaced such that all vertices lie inside the region delimited by the two planes as depicted in Figure A.12.c. Note that vertex CP_1 initially lies outside B (A.12.b), but displacing the plane to B', CP_1 will lie in the inner halfspace. The displacement d is exactly the distance from CP_1 to B.

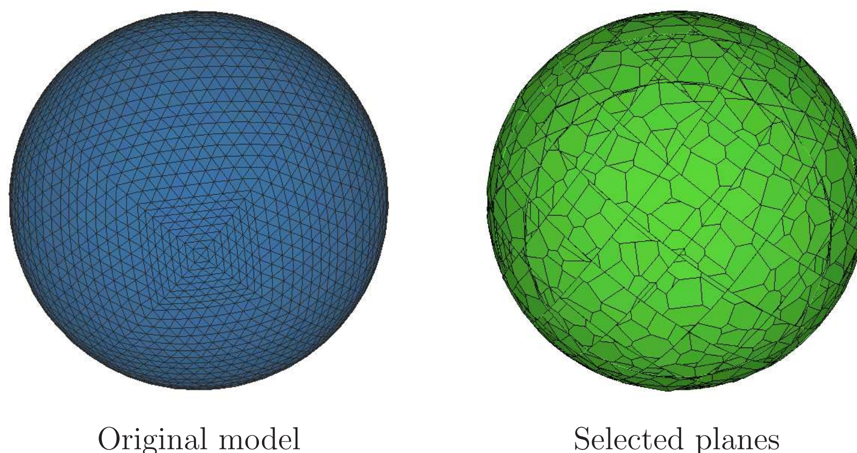


Figure A.13: *Effects of applying a k-medoids algorithm over convex surfaces.*

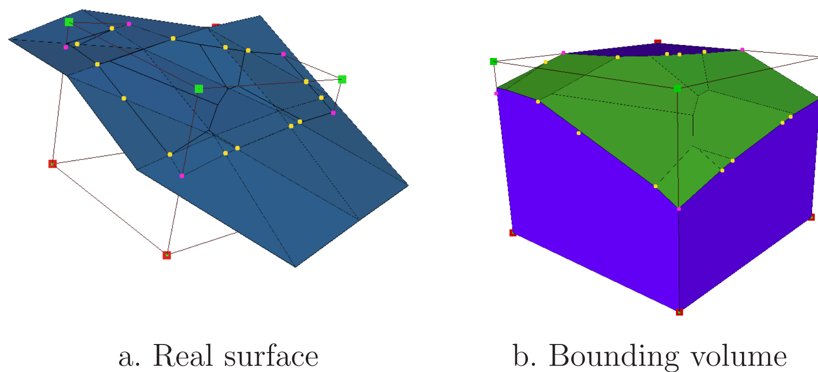
Each relevant plane -displaced or not- is selected as part of the bounding set. The key point is to define which planes are relevant to the bounding set. We could consider that all bounding planes are relevant, but this would make our method very inefficient when

applied to convex models. To avoid such inefficiency, we apply a classification technique to determine which planes best circumscribe the original surface. This is done by using a k-medoids algorithm [KR90], using the distance among unit normals as the grouping function. By doing so, we discard planes with similar orientations and select only the most relevant k planes homogeneously distributed in the 2D normal space, as shown in Figure A.13.

We can define the bounding volume in a node as the intersection of all the inner halfspaces defined by the selected planes with the node volume. This bounding volume has some interesting properties, such as its convexity and the fact that all planes are extracted from the original surface of the model.

A.2.3.1. Labelling of leaf nodes corners.

As it has been noted above, we compute all intersecting points of the surface with the node box, depicted in Figure A.14a (yellow: triangle edges/node faces intersections, purple: node edges/triangles intersections).



a. Real surface b. Bounding volume

Figure A.14: *Bounding volume at leaf node.*

Also, in Figure A.14 the node corners classification is displayed: *green* if it is outside the solid surface, *red* if inside. This classification is achieved by looking at the orientation of the triangles that intersect with node box edges (purple points). As usually not every node edge is traversed by a triangle, some of the corners may remain unlabelled. Then, a propagation of the values is done until the eight corners are properly labelled.

This classification of the node corners allows us also to avoid some specific cases as shown in Figure A.15. In this case, the surface of the model penetrates into the node without intersecting any node edge, so there is just one vertex inside the node in a concavity (as shown in 2D in Figure A.16) and a few triangle edge/node face intersecting points. With this configuration it is not possible to label corners as explained above, so we have to use an alternative algorithm.

The algorithm to label corners in these special cases (similar to *vertex nodes* in [BN90]) is based on analyzing the configuration of the intersecting triangles in the node face that they traverse with respect to any of the four corners of that face, as displayed in Figure A.17. If the closest point to corner C is in a segment, as S_1 in Figure A.17a, corner C is classified

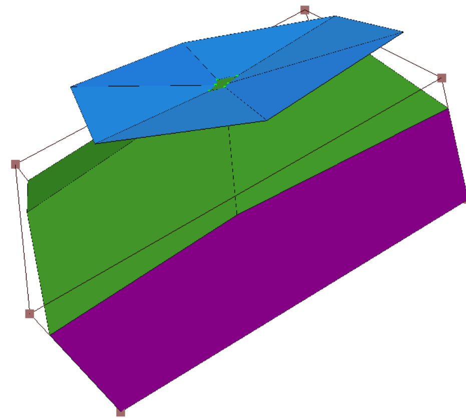
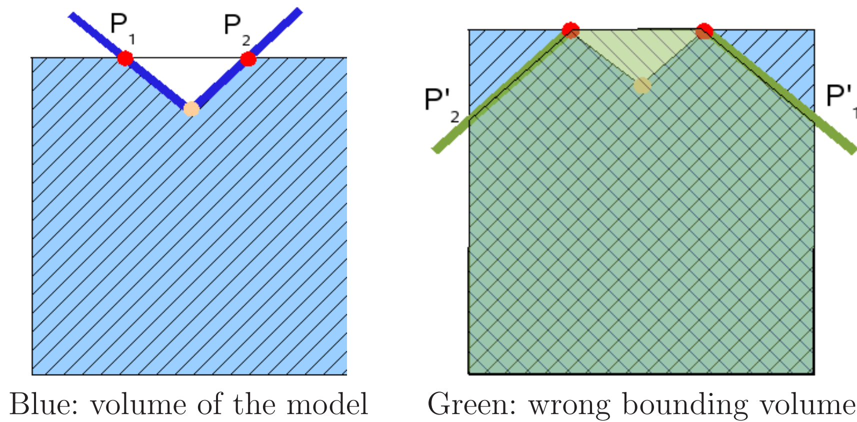
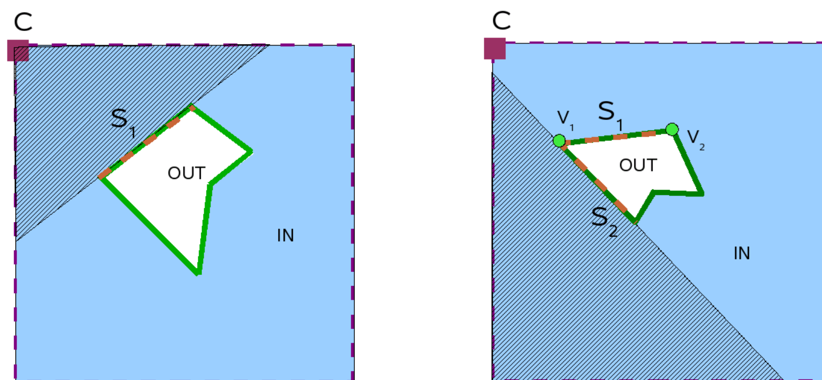


Figure A.15: *Error in bounding planes (green) configuration.*



Blue: volume of the model   Green: wrong bounding volume

Figure A.16: *2D scheme of Figure A.15*



a. Closest point to corner is in S_1 b. Closest point to corner is vertex V_1.

Figure A.17: *Labelling node corners when there are no triangles intersecting node edges. In blue, the volume defined by the model.*

with respect to the orthogonalization of the supporting plane of the triangle that creates the intersecting segment S_1. If the closest point to corner C is one of the vertices, V_1, as in Figure A.17b, two segments S_1 and S_2 are taken into account. Then, vertex V_2 is classified with respect to segment S_2, and corner C is labelled with the contrary value, i.e. if V_2 is *outside* with respect to S_2, then C is inside with respect to the solid surface.

A.2.3.2. Creation of black and white nodes.

The leaf corners labelling is a key point of our data structure, because, apart from solving peak convexities or concavities as shown in Figure A.15, it provides us enough information to create black and white nodes, because these do not exist as they are not traversed by any polygon.

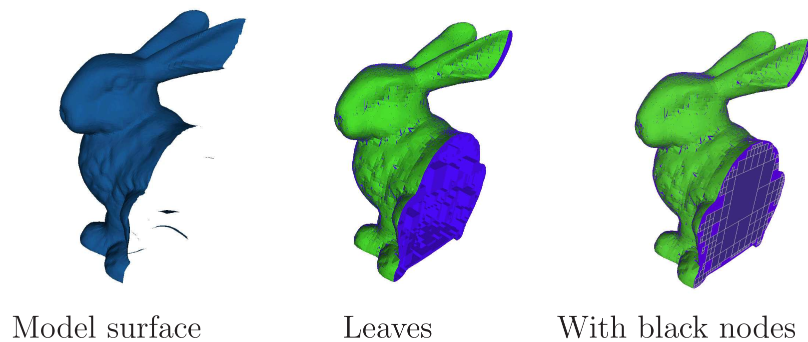


Figure A.18: *Black nodes (violet) allow to classify points in the interior of the model.*

Figure A.18 shows that gray and leaf nodes are enough to get a closed bounding of the original model. But these nodes do not give information to classify every point in the space, as in the case displayed in Figure A.19, it is not possible to classify point P because two different surfaces lead to the same gray node bounding volume.

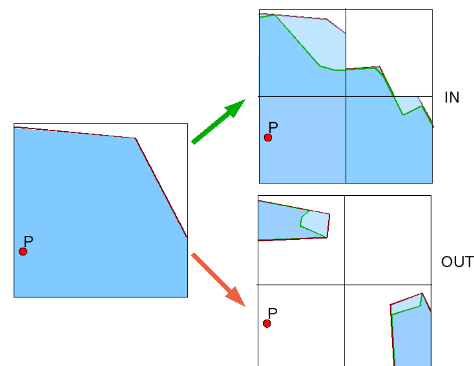


Figure A.19: *Point P cannot be properly classified without having black or white nodes.*

To avoid such classification errors, we use the information from corners of leaves, by propagating it upwards and creating missing nodes according to the value of adjacent siblings

corners. The algorithm is simple: for each gray node, its eight corners are labelled with the value of the same corner of the corresponding descendant. If this descendant does not exist, the value of any of the shared corner among all siblings is taken to decide whether it is a black or a white node and hence the value of its height corners.

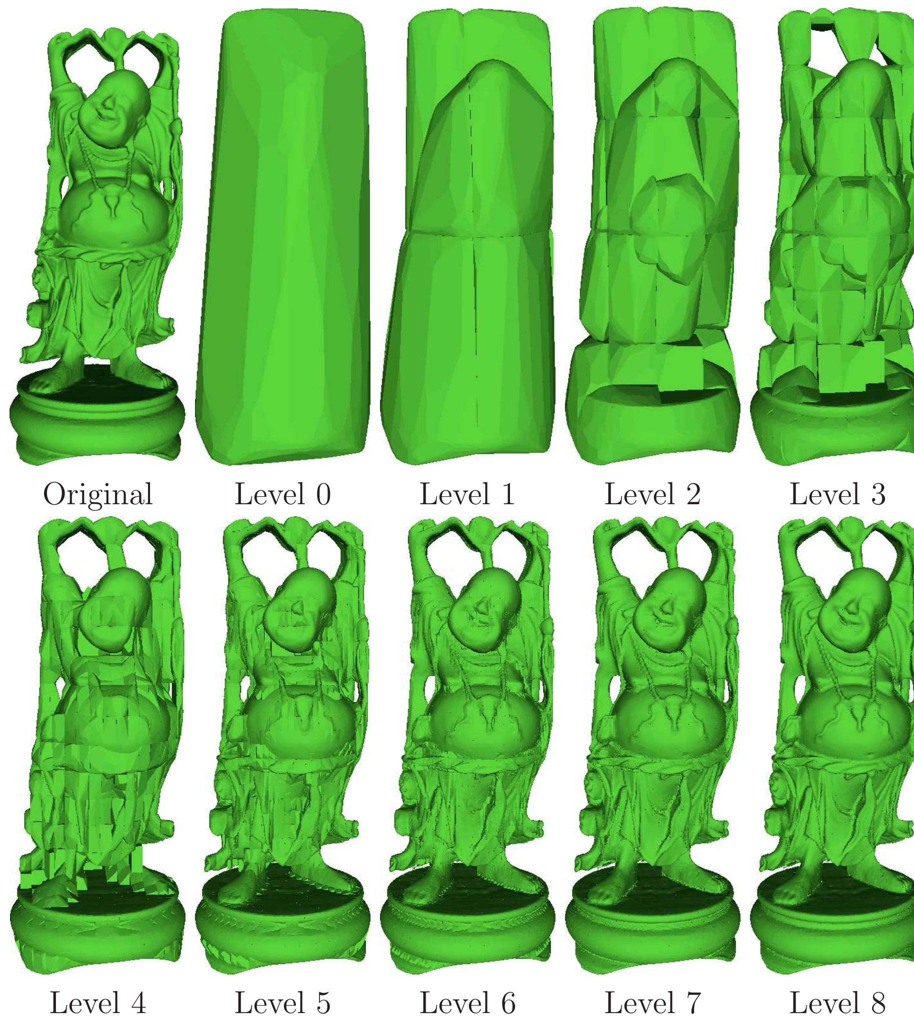


Figure A.20: *Levels of the Happy Buddha BP-Octree.*

A.2.3.3. Hierarchy of bounding volumes.

Following the process described in previous sections, the final result is a hierarchy of bounding volumes that decreases its volume as the tree is traversed. The visualization of these volumes for the *Happy Buddha* model is displayed in Figure A.20

Table A.2 shows the volume enclosed by each level of the BP-Octree, being 100 % the total volume of the original model. It can be seen as level 5 offers a volume just 5 % higher than the high resolution model, and the visual perception at that level also approximates very well to the original model, as Figure A.21 shows.

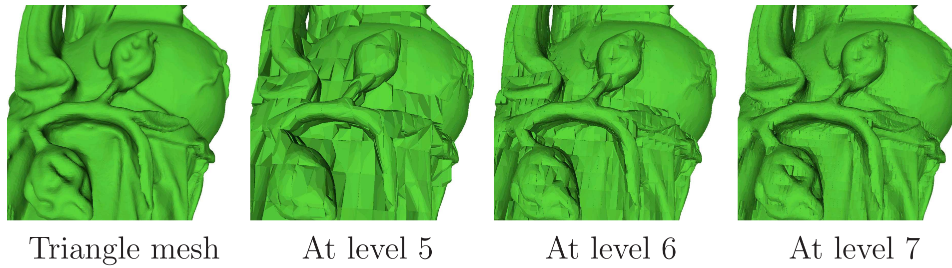


Figure A.21: *Detail of the Happy Buddha BP-Octree.*

Level	Bunny	Armadillo	Buddha	Dragon
0	171,41 %	388,73 %	196,91 %	583,10 %
1	144,71 %	214,88 %	177,06 %	320,75 %
2	119,92 %	139,57 %	144,42 %	193,04 %
3	107,33 %	116,78 %	125,48 %	133,70 %
4	103,11 %	107,99 %	110,23 %	114,69 %
5	101,14 %	103,45 %	103,58 %	106,61 %
6	100,55 %	101,57 %	101,27 %	102,90 %
7	–	100,92 %	100,44 %	101,19 %
8	–	100,21 %	100,26 %	100,43 %
9	–	–	100,03 %	100,01 %

Table A.2: *Volume enclosed by the BP-Octree at each level (B-rep volume=100 %).*

Model	*Polygons*	*Secs.*
Sphere	8.192	6,58
Bunny	71.040	43,72
Golf	100.243	105,22
Armadillo	150.000	130,57
Egyptian	161.909	180,60
Teeth	233.204	320,28
Fertility	483.226	1.530,38
Angelo	674.764	1.271,99
Phlegmatic Dragon	715.933	1.182,52
Stanford Dragon	871.414	963,77
Happy Buddha	1.087.716	1.573,40
Blade	1.765.388	1.565,60
Dragon	7.218.906	8.614,58

Table A.3: *BP-Octree building times.*

A.2.4. Building times

Building time of BP-Octree depends mainly on the number of polygons of the original model and its global convexity. In table A.3 (and its graphical representation in Figure A.22) it can be appreciated as there is an almost lineal correspondence between polygons and seconds [1] .

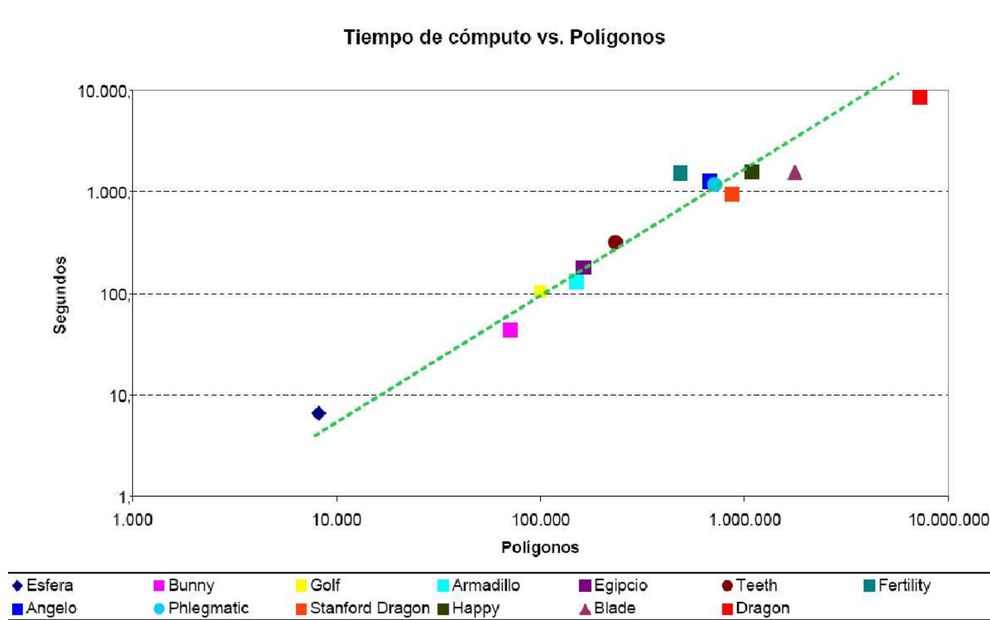


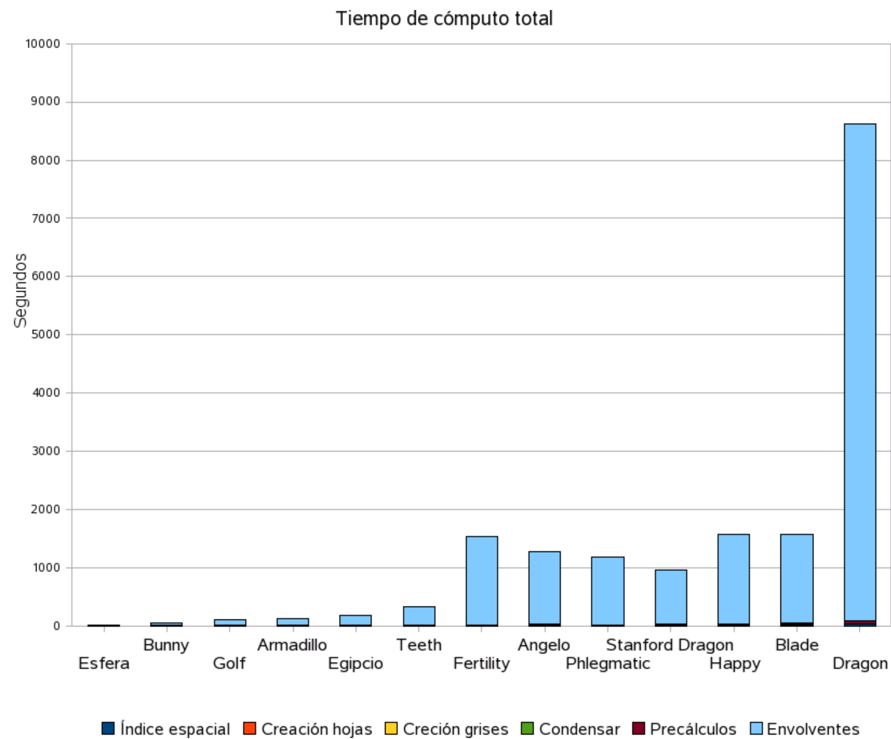
Figure A.22: *BP-Octree building times with respect to number of polygons of original model.*

Figure A.23a shows the distribution of computation time among stages, and it is clear that over 95 % of total time is spent in compute bounding planes. Figure A.23b clarifies the time at each one of the stages, excluding the most time consuming. *Fertility* model has a strange behaviour, consuming more time than other larger models. This is because its very convex and smooth surface causes a lot of planes to be selected and propagated to the upper levels of the tree, where they are pruned by applying the k-medoids algorithm .

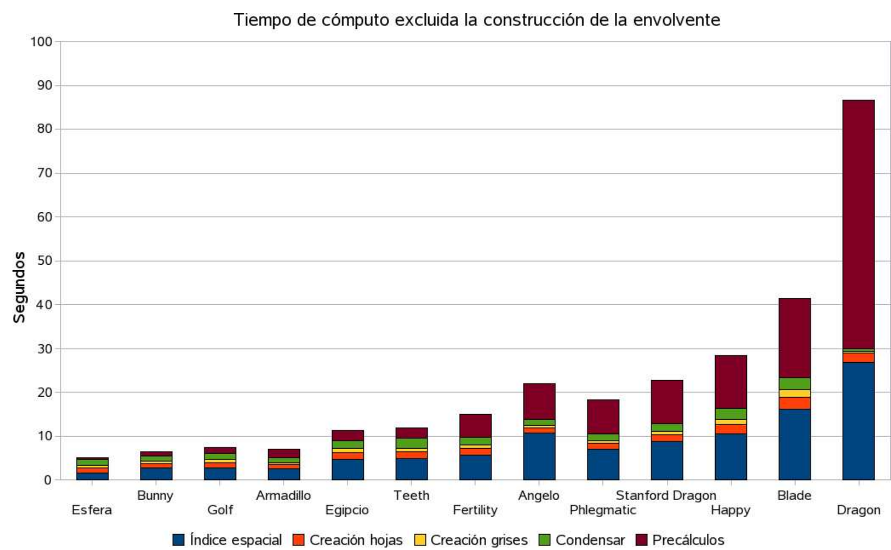
A.2.4.1. How the k parameter influences building times

By varying the k parameter of the *k-medoids* algorithm, it is possible to achieve a notable improvement on the algorithm performance. Timings in table A.3 have been obtained with a K distribution such as at level L_0, $K(L_0) = k * NP$, $k = 1{,}00\,\%$, being NP the number of polygons, and $K(L_i) = K(L_{i-1})/2$. As an additional criterion, in order to guarantee that there are always pruned planes at any node n of level L_i, if the number of planes (clusters) to select $CP(n) > K(L_i)$, it is reduced until $K(L_i) = K(L_j)/j > i$ and $K(L_j) < CP(n)$, i.e.,

[1] Tests running on a AMD 64bits, 2GB RAM. Executable file compiled with *gcc* using optimization option -O3



a. *Building times per stage*



b. *Building time per stage, excluded computing boundings*

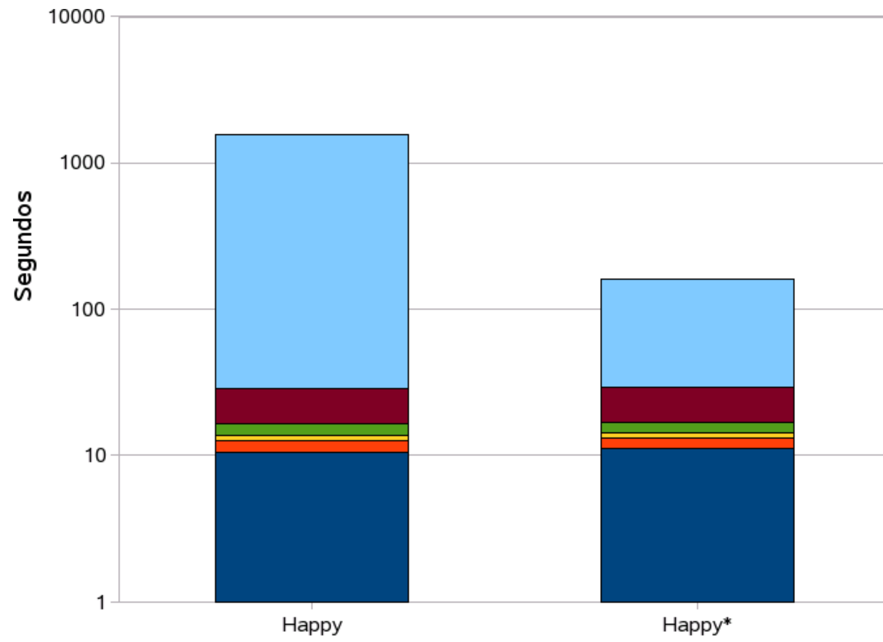
Figure A.23: *BP-Octree building times per stages.*

it is divided by two until the obtained value $K(L_i)'$ is greater than the number of planes we have in that node n.

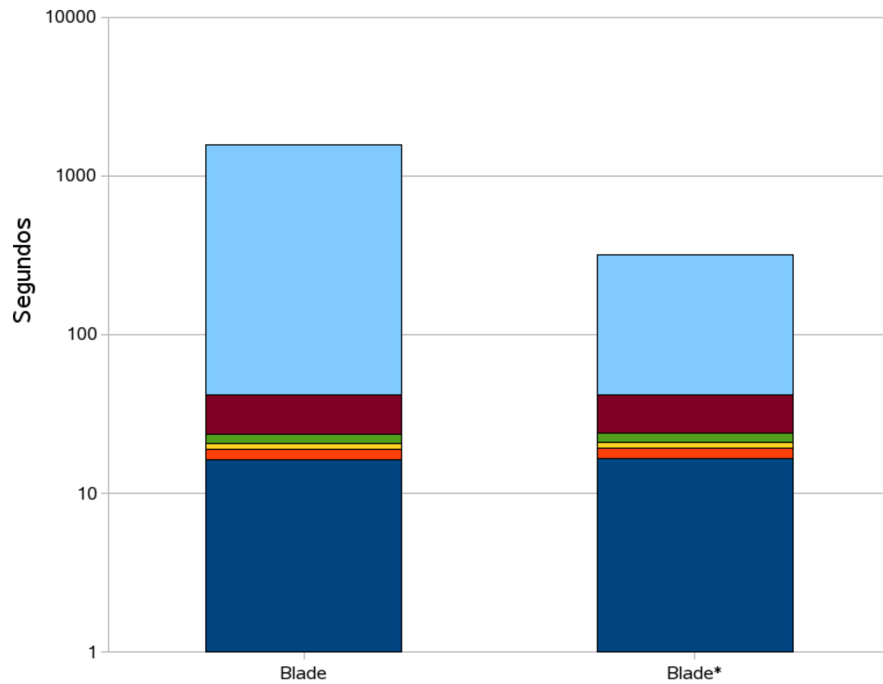
In table A.4 it is shown that reducing $K(L_0)$ to $K(L_0) = 0{,}0005 * NP$, i.e. $k = 0{,}05\,\%$, the building time for large models is greatly improved (Figure A.24), and also results in a smaller set of planes.

	Happy		Blade		Dragon	
k root	0.01	0.0005	0.01	0.0005	0.01	0.0005
Seconds	1573,40	160,82	1565,60	315,91	8614,58	1599,98
Planes at level						
Level 0	4816	501	2431	587	3.970	1.777
Level 1	10.017	1.585	8.706	2.342	10.293	4.374
Level 2	21.006	3.972	19.198	5.808	33.266	12.645
Level 3	47.276	10.989	49.381	16.072	88.022	31.279
Level 4	105.454	28.720	126.592	45.474	192.631	71.103
Level 5	229.782	72.470	324.505	130.854	429.974	167.548
Level 6	462.378	158.779	765.858	337.321	938.582	397.750
Level 7	867.293	296.711	1.646.645	717.515	2.015.802	946.210
Level 8	904.894	287.321	1.318.353	605.366	4.197.030	2.176.897
Level 9	94.281	22.658	6.984	2.046	7.804.535	4.222.341

Table A.4: *Influence of parameter k.*



a. *Happy Buddha* $k = 0{,}01$ *(Happy) and* $k = 0{,}0005$ *(Happy*)*



b. *Turbine Blade model* $k = 0{,}01$ *(Blade) and* $k = 0{,}0005$ *(Blade*)*

Figure A.24: *Bounding planes computing time changes as varying K.*

A.2.5. External memory storing of BP-Octree

Once the tree is built, it is straightforward to save it into a file to avoid repeating the building process. Data are distributed into three files:

1) *.bpo* file. It just references the location of BP-Octree planes and nodes files, and the original polygonal model file.

2) *.bpo.bpl* file. Stores planes used in node boundings. These planes are in the same order as they appear while traversing the BP-Octree in breath-first order.

3) *.bpo.bpn* file. Stores the hierarchical data structure, following the order given by the octcodes. Also, at the beginning the coordinates of the bounding box of the model are stored. Data in the file are organized as follows:

   - Header, helpful for understanding the codes in the rest of the file:

   ```
   xmin ymin zmin xmax yax zmax
   typecodeforBlack (1 byte)
   typecodeforWhite (1 byte)
   typecodeforLeaf  (1 byte)
   typecodeforGray  (1 byte)
   ```

   - Nodes ordered by its octcode. Depending on the node type, there are three possible types of information to be saved:

     a) If node is *white* or *black*, only its type is stored (1 byte)

     b) If node is *gray*, its type is stored, as well as the number of planes, and a list of pairs $< planeIndex, offset >$. If *planeIndex* is negative, it means that there is offset. If it is positive, there is no *offset*.

     ```
     typecodeforGray  (1 byte)
     nrOfPlanes       (2 byte)
     planeIndex1      # <0
     offset1
     planeIndex2      # >0
     planeIndex3      # <0
     offset3
     ..
     planeIndexN      # >0
     ```

     c) If node is *leaf*, apart from data included in gray nodes, it is included the corner configuration, the number of polygons and their indices:

     ```
     typecodeforLeaf  (1 byte)
     cornerConfig     (1 byte)
     nrOfPlanes       (2 byte)
     planeIndex1      # <0
     offset1
     planeIndex2      # >0
     ```

```
planeIndex3      # <0
offset3
..
planeIndexN      # >0
nrOfFaces       (2 byte)
faceIndex1
faceIndex2
..
faceIndexM
```

The configuration of the corners is stored in a *bitset(8)*, being the value of bit i *true* if corner i is *inside* or *false* if it is *outside*. The *octcode* value is not stored in the file, as it can be computed while reading the nodes file.

All data are stored in binary format, and table A.5 shows the disk space needed for each one of the evaluated models. BP-octree model needs around 2.5 times the disk space of the polygonal model.

Model	.bpn	.bpl	.ply	Ratio
Sphere	136,5	63,5	16,0	12,5
Stanford Bunny	2.038,5	908,6	1.331,4	2,60
Golf	3.097,1	1.307,1	1.765,4	2.49
Armadillo	4.774,1	2.017,5	2.636,7	2,57
Aegyptian	5.377,1	2.082,7	2.846,3	2.62
Teeth	6.929,1	2.929,7	4.099,3	2,40
Fertility	15.514,2	6.064,7	8.494,1	2,43
Angelo	21.523,2	9.014,6	12.520,2	2.57
Phleg. Dragon	21.903,3	9.442,2	13.298,9	2,35
Stanford Dragon	25.126,4	10.562,8	16.191,5	2.20
Happy Buddha	31.544,6	13.093,4	20.180,0	2,21
Blade	50.043,2	22.131,3	32.759,5	2,20
Asian Dragon	327.866,5	85.395,0	126.894,9	3,25

Table A.5: *KBytes of BP-Octree files, and ratio* $(.bpn + .bpl)/.ply$.

A.3. Using the BP-Octree as LOD Scheme

The intrinsic hierarchical feature of the BP-Octree makes it easy to use it to transmit progressively the model over a network, by sending planes starting either at the root level or one of its descendants, depending on the available bandwidth.

Table A.6 shows the number of planes stored at each level for four representative models: Stanford Bunny (69K polygons), Stanford Dragon (202K polygons), Fertility (483K polygons) and Phlegmatic Dragon (715K polygons). It can be seen from Table A.7 that the topmost level uses less than 0.50 % of the total planes, and Figure A.25 shows tight approximations to models which require less than 10 % of total planes.

Level	Bunny	Dragon	Fertility	Phlegmatic
0	391	529	1777	743
1	1225	1658	3267	2166
2	3087	4230	7418	6031
3	7120	10822	18691	17663
4	16019	27637	42056	43295
5	33502	61272	91574	102967
6	46534	116649	185686	230607
7	-	52331	296797	423267
8	-	67	3399	144246

Table A.6: *Number of planes at each level of the BP-Octree.*

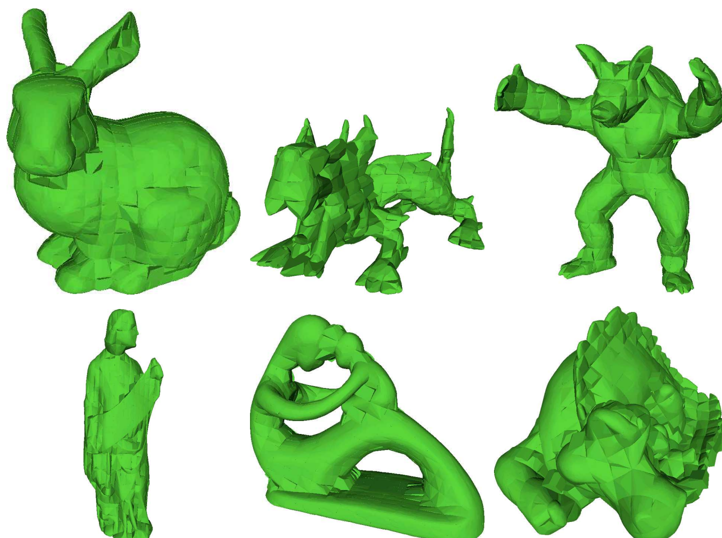


Figure A.25: *Several models at level 4, using less than 10 % of planes.*

Level	Bunny	Dragon	Fertility	Phlegmatic
0	0,56 %	0,26 %	0,37 %	0,10 %
1	1,16 %	0,54 %	0,30 %	0,19 %
2	2,45 %	1,20 %	0,82 %	0,52 %
3	5,05 %	2,90 %	2,17 %	1,52 %
4	10,27 %	7,11 %	4,35 %	3,24 %
5	19,11 %	13,24 %	8,65 %	7,21 %
6	19,82 %	21,08 %	15,39 %	14,57 %
7	-	7,77 %	20,57 %	22,75 %
8	-	0,01 %	0,22 %	7,27 %

Table A.7: *Percentage of total number of planes that are used at first time at each level.*

We remark that transmitting a number of planes at one level does not mean it requires sending four floating point coefficients per plane, plus its offset value. Indeed, since most of them have been used at upper levels or appear more than once at that level, each reuse of a plane requires that both the index and the offset values are sent.

So far in this section we have just discussed bounding-planes transmission, assuming that the final geometry is never transmitted until the transmitted data reaches a fine approximation of the real model. To send the final geometry, each polygon needs to be sent only once, the first time it is found in a leaf node. Then, at client side, it is assigned to every leaf node where it is reused, as described in subsection A.2.1.3. This means that it is not necessary to transmit any redundant data over the network. Furthermore, our structure overhead is low enough to achieve a good approximation to models even at early stages of communication.

Figure A.20 shows the appearance of Happy Buddha model at each level of the BP-Octree, and the accumulated volume of data necessary to reach each level is displayed in Figure A.26. It can be seen that transferring information up to the sixth level of the multiresolution hierarchy is equivalent to sending the original full detailed model. Figure A.20 shows that it is not necessary to reach such level to obtain a good approximation of the model. More tables and figures are available in the Spanish text (Tables 4.4 to 4.14 and Figures 4.18 to 4.28).

A.3.0.1. Volume of the BP-Octree at each level

Table A.9 shows the volume enclosed by the bounding planes at each level of the BP-Octree. It is clearly shown in Figure A.27 as from level 3, the volume enclosed by the BP-Octree is less than 150 % of the original model, being at level 5 less than 4 % greater in most of the models.

Level	Planes			Polygons		Vertices		Nodes		Transm.
	Total	Used	Displaced	Total	Used	Total	Used	Gray	Leaf	at Level
0	4.816	0	4.361	0	0	0	0	1	0	94.503
1	10.017	4.892	8.874	0	0	0	0	8	0	137.088
2	21.006	10.445	18.903	0	0	0	0	56	0	286.536
3	47.276	22.616	42.278	118	84	354	341	290	5	658.002
4	105.454	54.079	92.807	1.038	566	3.114	2.812	1.420	56	1.436.684
5	229.782	128.979	198.765	24.259	12.817	72.777	66.285	5.359	1.199	3.493.913
6	462.378	295.858	388.264	149.679	80.913	449.037	409.525	16.816	8.336	8.785.368
7	867.293	614.615	699.030	945.849	539.316	2.837.547	2.624.609	24.027	60.652	29.125.701
8	904.894	700.176	698.669	1.553.295	1.003.105	4.659.885	4.397.739	2.790	139.218	35.845.230
9	94.281	77.559	70.431	166.130	115.851	498.390	476.141	0	17.934	2.321.460

Table A.8: *Happy Buddha transmitted data.*

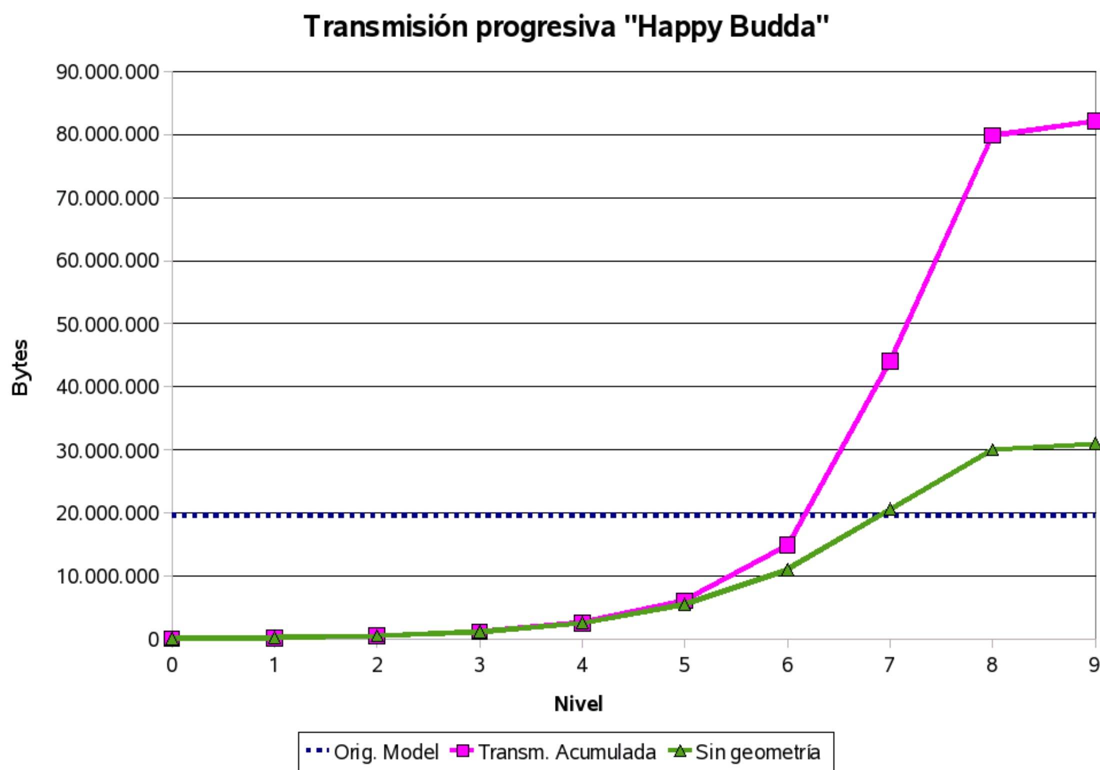


Figure A.26: *Accumulated byte count for the progressive transmission of Happy Buddha. The dashed line shows the size of the original geometry.*

	0	1	2	3	4	5	6	7	8
Bunny	171,41 %	144,71 %	119,92 %	107,33 %	103,11 %	101,14 %	100,55 %	100,55 %	100,55 %
Golf	103,34 %	101,84 %	101,74 %	101,68 %	101,17 %	100,50 %	100,21 %	100,21 %	100,21 %
Armadillo	388,73 %	214,88 %	139,57 %	116,78 %	107,99 %	103,45 %	101,57 %	100,92 %	100,92 %
Egiptian	150,02 %	133,02 %	113,27 %	105,79 %	102,18 %	101,03 %	100,51 %	100,41 %	100,41 %
Fertility	203,49 %	182,88 %	140,04 %	114,99 %	104,17 %	101,20 %	100,48 %	100,24 %	100,22 %
Teeth	161,69 %	117,11 %	112,34 %	105,93 %	102,53 %	101,07 %	100,44 %	100,35 %	100,35 %
Angelo	138,15 %	138,47 %	121,46 %	114,07 %	107,23 %	103,58 %	101,53 %	100,62 %	100,39 %
Phlegmatic	170,09 %	143,14 %	128,27 %	112,90 %	105,97 %	102,33 %	100,84 %	100,29 %	100,22 %
St. Dragon	277,82 %	199,08 %	165,60 %	135,50 %	111,44 %	103,68 %	101,30 %	100,50 %	100,32 %
Happy Buddha	196,91 %	177,06 %	144,42 %	125,48 %	110,23 %	103,58 %	101,27 %	100,44 %	100,26 %
Blade	373,79 %	246,02 %	199,07 %	152,86 %	125,31 %	108,97 %	103,40 %	101,37 %	100,90 %
Asian Dragon	583,10 %	320,75 %	193,04 %	133,70 %	114,69 %	106,61 %	102,90 %	101,19 %	100,43 %

Table A.9: *Volume of the BP-Octree at each level(Brep model is 100 %).*

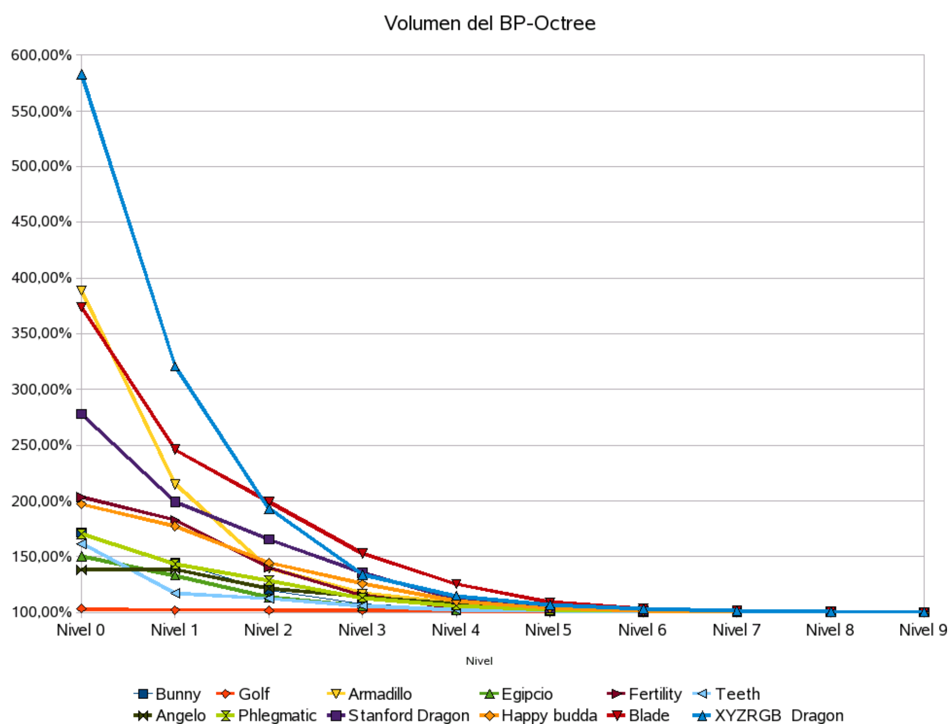


Figure A.27: *Volume of BP-Octree representation with respect to original model.*

A.3.1. Adaptive Visualization

As we can readily see, the BP-Octree data structure makes it possible to render geometric models adaptively. Furtermore, since the criteria to decide whether or not to visit the descendants of a given node are not fixed, it is possible to use the observer distance to the object or any other criterion such as the screen size of the node being traversed.

Indeed, Figure A.28 shows an adaptive visualization of both models using the criterion that each node should be displayed if it is at least 40 pixels wide. In the foreground, we can recognize the original geometry at the maximum level of detail, while in the background, nodes are displayed at lower resolution.

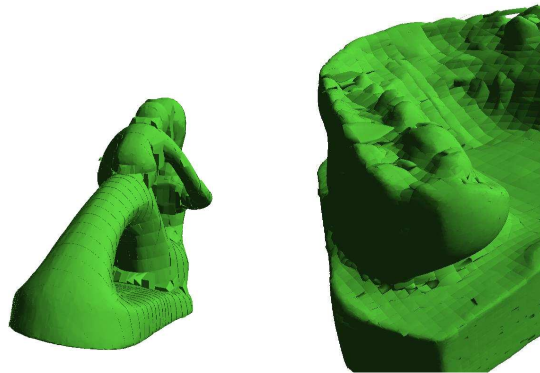
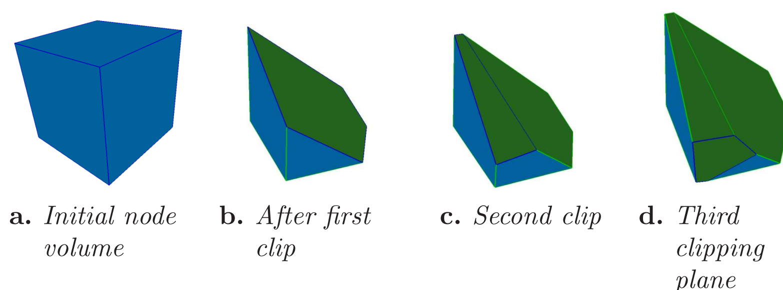


Figure A.28: *Adaptive visualization of Fertility and Teeth.*

A.3.2. Visualization of BP-Octree Nodes

The easiest way to render the set of bounding planes constrained to the node volume is via a mesh clipping algorithm, starting with the node parallelepiped, and clipping it against each bounding plane in succession.



a. *Initial node volume* **b.** *After first clip* **c.** *Second clip* **d.** *Third clipping plane*

Figure A.29: *Node mesh clipping*

Figure A.29 illustrates how this algorithm is applied to a node with three bounding planes. As the original node faces are well known, it is easy to decide whether to display them or

not. In case we want to render a model without visible holes in the mesh, we must display the node faces –depicted in blue– and fill the cracks between two adjacent nodes.

A.3.3. Improved Visualization using Impostors

In [MCT05] an approach was presented using SP-Octrees for adaptive visualization, that improves the visualization of these multiresolution models by applying impostors over those planes of the hierarchical structure belonging to the convex hull but not part of the solid boundary (ficticious planes). By doing so, it was possible to obtain a better approximating visualization of the object, although the maximum LOD is not used. At every moment, the impostor is selected depending on the viewpoint.

We propose to use the same technique over BP-Octrees in order to improve visualization at higher levels of the octree.

A.3.3.1. Applying *Impostors*

Projective texture mapping was proposed by Segal [SKv+92a], and is part of the OpenGL standard [SA94]. Although in the original paper it was used only to accelerate the shadow and lighting generation, it is possible to apply *Projective Texture Mapping* directly to image-based rendering, because it can simulate the inverse process of taking photographs with a camera, allowing projection of images over the scene like a light projector.

To project the texture, the user specifies the projector position and orientation, and a virtual projection plane where the image is projected. With these data, the model transformation and projection matrices of the scene we can calculate the right texture coordinates [Eve, SA94]

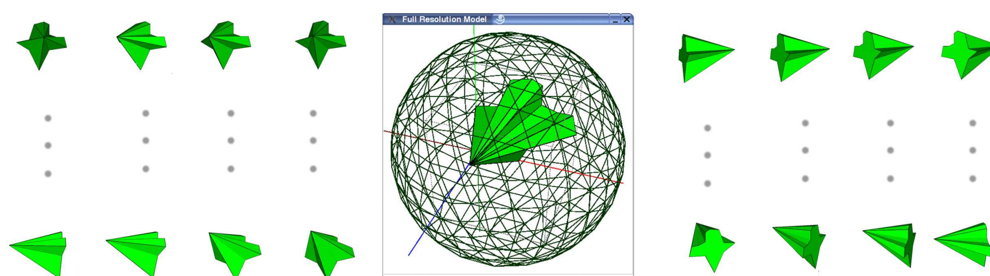


Figure A.30: *Sphere with 258 vertexes, used to take the impostors.*

A.3.3.2. Impostors Generation

The set of images used as impostors is generated once in a preprocessing time. The images are taken from the vertice of a sphere that contains the solid bounding box. These images are taken by rendering the real solid at the maximum LOD. By varying the LOD of the sphere, it is possible to control the amount of images or impostors to be used.

Another approach is to determine the visibility points for each plane stored in the SP-Octree, and take a set of images for each plane from different view points. We think that such approach needs a higher amount of memory, as well as an increase of rendering time, because each plane has its own texture and texture switching spends CPU time.

Having done several tests at different sphere resolution, and checking the quality of the transitions between impostors and the subjective appearance of the interactive visualization of the three-dimensional model, we think that using 258 images (fig. A.30) could be enough to obtain the propoer appearance of the SP-Octree, especially when the solid is far enough so as not to have to descend in the hierarchy and it is rendered quite small.

We position the camera at each vertex of the sphere, oriented towards the centre. By using the orthographic projection, we only have to ensure that the object fits entirely within the projection plane. All the images are stored in a binary file, as well as information about all the OpenGL matrices used when the image was taken. The object was rendered with a controlled background, so that transparent pixels can be easily recognized.

A.3.3.3. Impostors Application

To render the model the tree is traversed in a descendent way, generating the corresponding polygons for each level. If the generated polygons do not belong to the solid boundary, we apply the selected impostor, otherwise, if it is a boundary polygon, we apply its original texture.

Since we store each image modelview matrix, it is possible to know where it was taken from, and to select as the impostor the one that best fits the view seen by the observer.

1) Before rendering the scene, we check the observer position relative to the object. This relative position determines the vector that will be used to locate the image from the set of impostors. Note that the position is not based on the *lookAt* vector, but it is based on the position of the SP-Octree relating to the observer.

2) From the table of impostors, we select the one that was taken in a similar position, and take its texture identifier (this impostor will be used for all ficticious planes).

3) The tree is traversed in preorder until the desired level, generating those polygons that are seen at each node.

4) The boundary polygons at the original model are rendered with its original texture or assigned colour. But, if we are on a non-boundary plane of a gray node, we apply the impostor previously selected.

This approach offers an improvement of visualization capabilities, being able to recognize the model from the first steps of visualization. As the observer gets closer to the object, the tree refines the nearest nodes to the camera, leaving at low resolution those that are not being seen.

In figure A.31 we can see as the proper impostor is selected at each moment, and as from the first level of the SP-Octree, the user perceives the original solid.

In sequence A.32 we can appreciate that from several points of view, the right impostor is selected, the result being very similar to the last level of the SP-Octree.

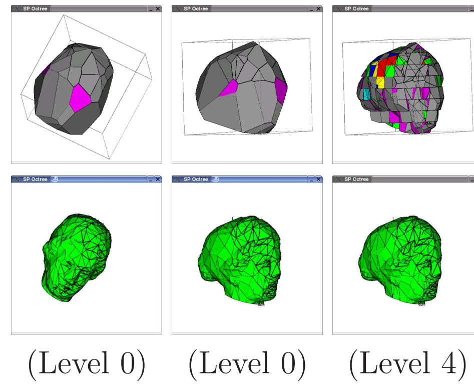


Figure A.31: *SP-Octree visualization without impostors (upper) and using impostors (lower).*

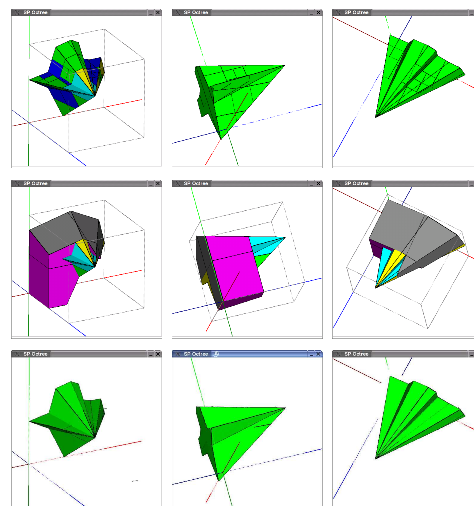


Figure A.32: *SP-Octree at maximum LOD (upper), without impostors at level 1 (centre) and with impostors at level 1 (lower)*

A.4. Point Classification

Using the BP-Octree representation scheme, it is straightforward to implement a point classification algorithm. To achieve this, we just have to locate the tested point p within the volume defined by the root node of the BP-Octree. This is done by computing $c = octcode(p)$ as described in section A.2.1. Then, the BP-Octree is directly traversed by using c as described in [Gar82] until a terminal node (*white, black* or *leaf*) is reached. As shown in Pseudoalgorithm 13, if `node` is *white* or *black* the returning value is trivial.

```
 bool inside(Point p, BPOctree bpo){
   OCTCODE_T c=bpo.octcode(p);
   node=bpo.root();
   int l=0;
   while (isGray(node)) {
     c=bpo.getChildAtLevel(l);
     node=node.getChild(c);
   }
   if (isBlack(node)) isinside=true;
   if (isWhite(node)) isinside=false;
   if (isLeaf(node))
      isinside=insideLeaf(p,node);
   return isinside;
}

bool insideLeaf(Point p, LeafBPONode node) {
   Point c=getCornerInside();
   int nrI=0;

   Segment s(p,c);
   nrI+=nrFacesIntersect(s,node.Faces);

   if (nrI MOD 2 == 0)  isinside=true;
   else isinside=false;
   if (c.label==OUTSIDE)
        isinside=!isinside;
   return isinside;
}
```

Algoritmo 13: *Pseudoalgorithm for point classification test*

If the algorithm reaches a *leaf* node, it uses the extra information about the classification of the corners. We build a segment between the point to be classified and one of the corners labelled as *inside*, as shown in Figure A.33. If none of the corners is *inside*, `getCornerInside()` returns an *outside* corner, and the result is the opposite as if it were *inside*. Then, a classic segment-polyhedron test based on the Jordan Curve Theorem [Jor87, O'R94] is applied.

We have performed several tests of this algorithm on a Athlon 64 3.2GHz, 2GB RAM

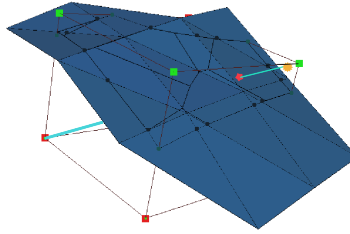


Figure A.33: *Intersection of segment –cyan– with surface –dark blue.*

computer, launching 10M points for each experiment. Results over several well known models are displayed in table A.10, and query rates are between 1.8M tests per second for model Teeth and 5.3M tests per second for model Dragon, which is the largest model among those used in the experiments. These results are clearly below others existing in the literature [OSF05, HTG04], as we move in several orders of magnitude below, i.e. less than half a microsecond per query. Another important conclusion is that the size of the model has no effect on the efficiency of the algorithm. *Inside* labelled points are displayed for several models in Figure A.34.

Model	*Polygons*	*Tests/seg*	*µsecs./test*
Bunny	71.040	2.840.909	0,35
Armadillo	150.000	3.424.658	0,29
Teeth	233.204	1.879.699	0,53
Fertility	483.226	2.793.296	0,35
Happy Buddha	1.087.716	2.336.449	0,42
Turbine Blade	1.765.388	2.380.952	0,42
Dragon	7.218.906	5.376.344	0,18

Table A.10: *Point in solid tests per second.*

A.5. Distance-to-solid computation

In haptic environments, it is necessary not only to define whether a point is inside or outside the virtual model, but also the distance from the haptic device's end effector, also known as Haptic Interface Point, HIP. For this purpose, distance fields and other techniques have been proved to be useful in achieving good update rates.

The BP-Octree data structure allows us to obtain the exact distance to the model and the normal of the nearest triangle when the point to test lies in the narrow band defined around the polygonal surface by the leaf nodes. If the point is outside this band, there are several approaches to obtain a wider band. We have tested the behaviour of two of them:

- To use the information of the bounding planes at gray nodes, taking into account that it is a *conservative* value that is smaller when the point is outside, and greater when it is inside.

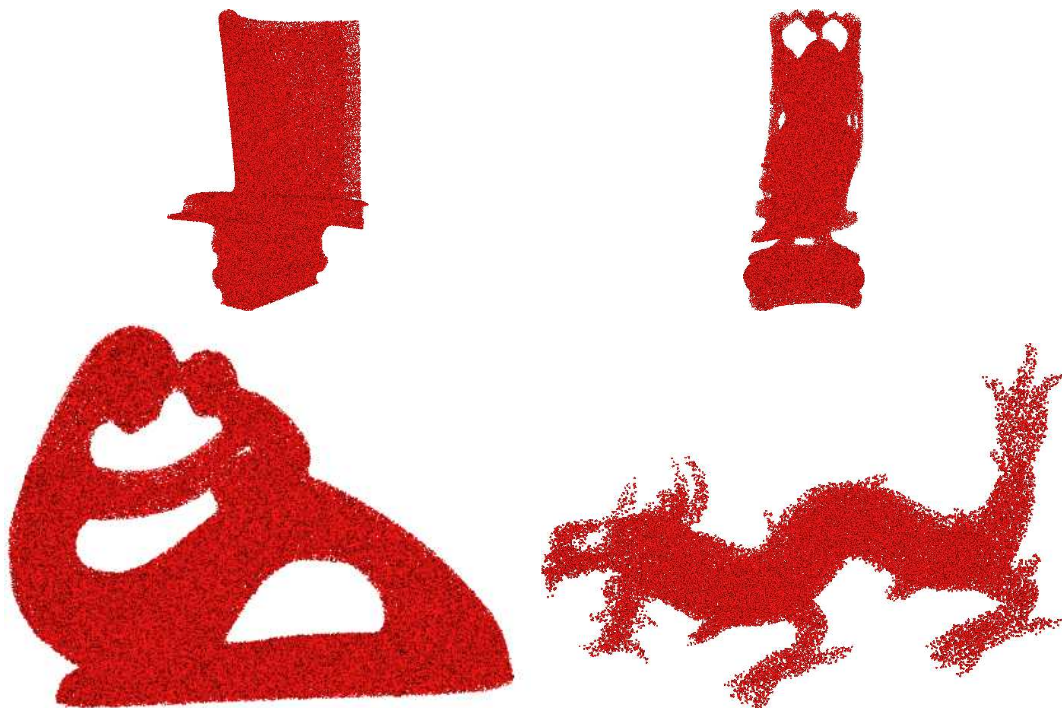


Figure A.34: *Visual display of point-in-solid test results.*

- To use the information of sibling leaf nodes, which would make it necessary to ascend several levels up to a gray node and then search recursively through all its descending leaves for the closest surface triangle.

The latter approach is necessarily slower than the former, due to the exhaustive recursive search of leaves. Furthermore, it becomes slower as the area within which we are searching is wider. However, the former one runs reasonably fast when limiting the maximum level where the algorithm should stop searching for gray nodes. Figure A.35 shows that the query time is significantly higher in recursive algorithm, but the algorithm that uses gray nodes decreases its performance slowly while increasing the size of the valid gray node, i.e. limiting the algorithm in a higher level of the tree reduces performance. Moreover, it can be appreciated that querying only at leaves does not affect to the query time considerably. In Figure A.36 it can be appreciated that the number of polygons of each model is not relevant for the query time, the algorithm behaving in an extraordinary manner with largest model.

In any event, all the algorithms proposed above achieve the desirable rate for haptic rendering, ranging between one and two million queries per second. These tests have been performed launching ten million points over the model and computing the distance and the normal of the closest triangle for each one of them. We have executed the algorithm on the same computer as the point classification tests discussed in section A.4.

Figure A.37 shows some slices from the distance fields of the *Happy Buddha* and *Dragon* models. A values band, which is 10 % of the model extent width, is displayed.

Figure A.38 shows that displayed colours are not interpolated, but they are really computed point by point. That is why it can be appreciated that there are some smaller discontinuities

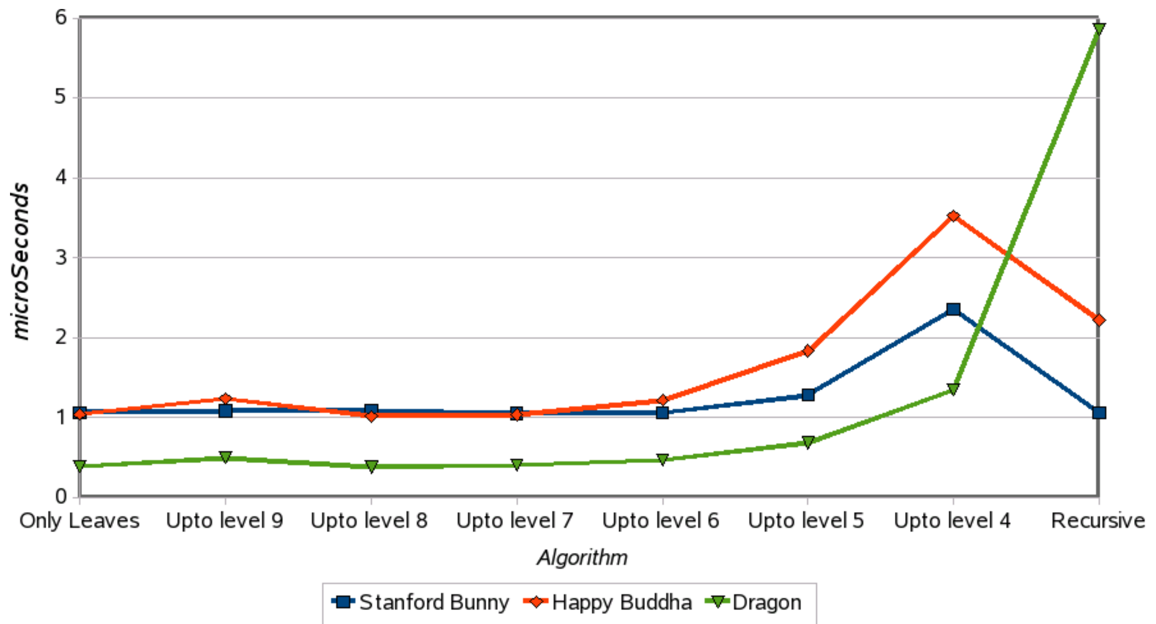


Figure A.35: *Query time for distance test using different approaches.*

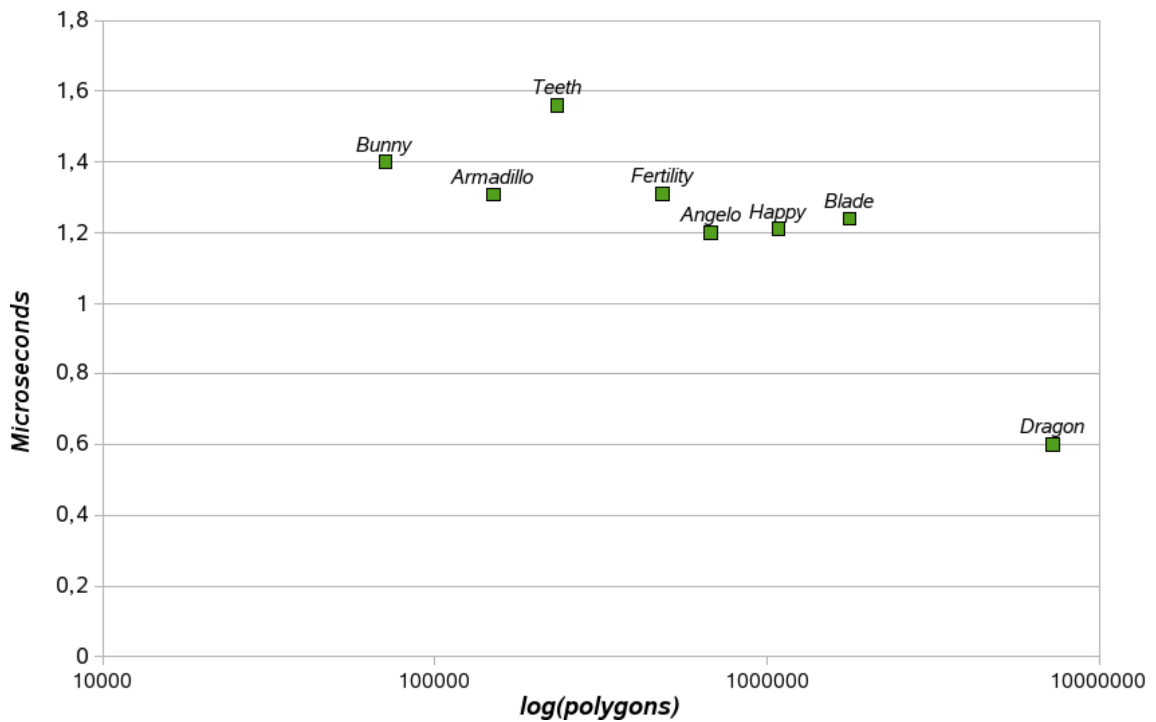


Figure A.36: *Query time depending on the number of polygons.*

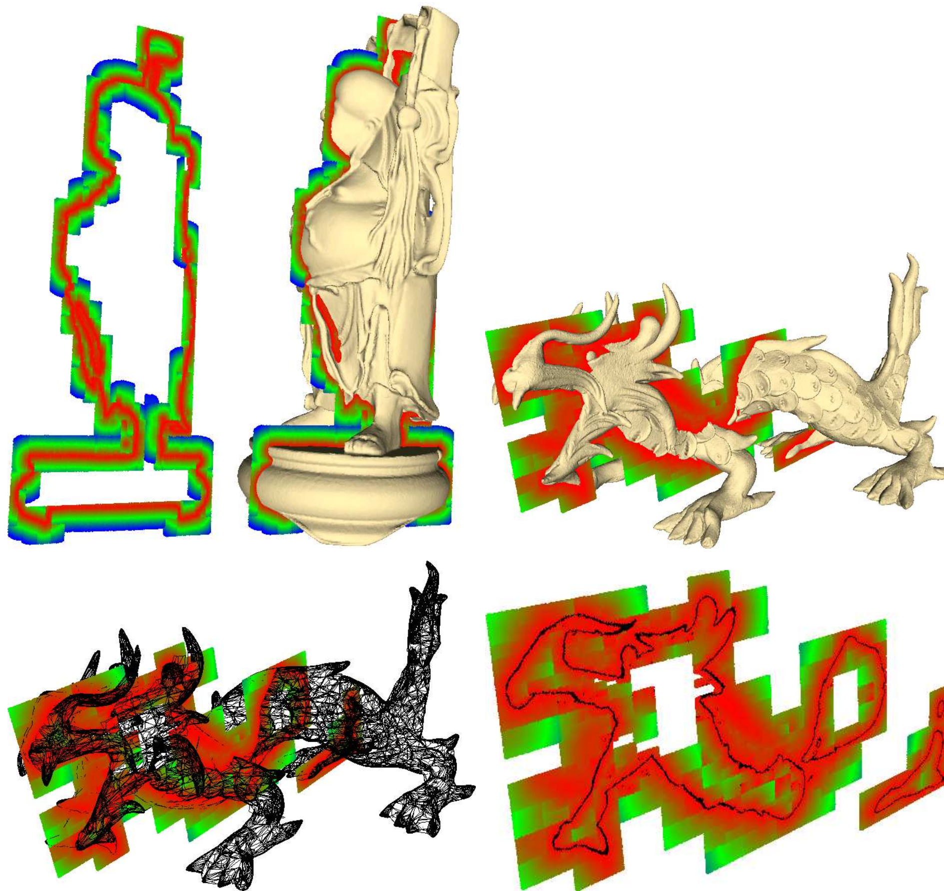


Figure A.37: *Distance field for several models. Black points are those closer than 0.01 % of model width.*

in the distance values inherent to the octree based data structure used by the algorithm. These discontinuities are overcome in haptic rendering by using a proxy that detects them and forces checking the distance in recent nodes as well, or in a recursive manner.

A.6. Haptic interaction with large polygonal models.

In order to test in a real time application the results described above, we have implemented a simple application prototype using a 6 DOF haptic interface. This basic prototype simulates the painting process over the surface, in a similar manner as [GEL00, KSD03] do. The haptic interface pointer behaves as an ink pen that when it contacts the surface of the virtual model, it releases a small portion of ink over the sculpture. The stronger the force applied to the surface, the wider the released spot. Furthermore, the spot is oriented with the same normal as the one given by the distance algorithm, so the resulting effect is a spot completely attached to the model surface.

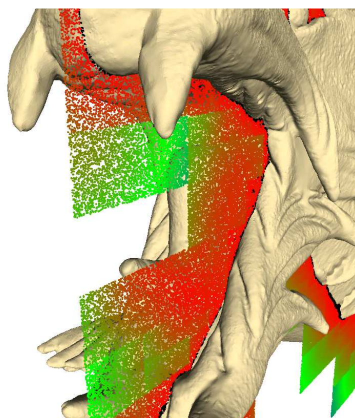


Figure A.38: *Closer look at distance values in Dragon model.*

Also, the visual display of the HIP changes its colour depending on the distance to the surface.

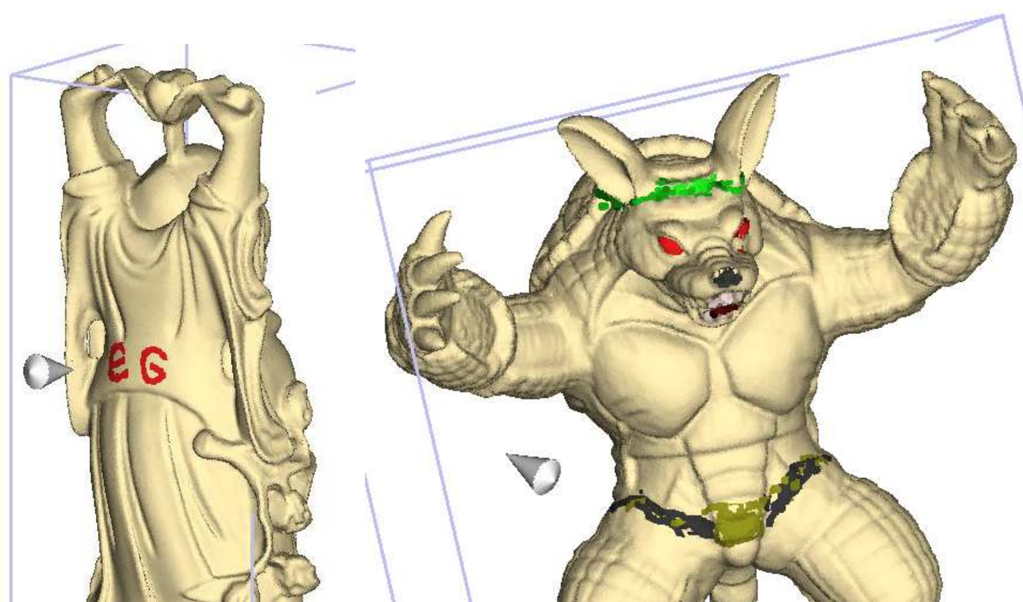


Figure A.39: *Haptic painting application. Graffiti on the Happy Buddha (1M triangles) and Armadillo (150K triangles) model.*

Figure A.39 shows some screenshots of the application running on a Pentium IV 3.2GHz, 1GB RAM computer, with a Nvidia Quatro 1400 graphics card. At all times, the haptic rendering update rate is what the HIP needs (1kHz).

A.7. Conclusions and future works.

We have developed a novel data structure, the BP-octree, which can handle large polygonal models, regardless of whether they have holes or not, or whether their faces are defined by triangles or polygons. This structure can be computed in a reasonable time, even though this need only be done once per model. Furthermore, planes are arranged at different levels in such a way as to make it possible for data to be transmitted progressively and depending on the observer.

While the technique features a reasonable image quality vs. bandwith tradeoff, it is still possible to improve the visualization of the hierarchical model, either by using the original normals of the model or through image based rendering techniques, such as impostors [MCT05].

We have implemented algorithms over the BP-Octree data structure to make it capable of handling large polygonal models in a haptic rendering system. Point classification query rates are below 1 microsecond, and distance queries take between one and two microseconds to be performed. All these processes run completely in CPU, and precomputing can be done just once per model and then the BP-Octree is stored in a file, retrieving data from disk rather than computing it again each time the model is loaded.

A prototype of haptic application has been developed using our algorithms, handling models over one million polygons in real time.

A.7.1. Future works.

This work is the starting point for new projects, new research lines that might lead to a better and wider use of BP-Octrees. Some of them are:

- *Development of a solid-to-solid collision detection algorithm.* We will implement an algorithm that, using the BP-Octree as data structure, detects collisions among several large polygonal models. It might be necessary to store in external memory not only the bounding planes, but also the bounding vertices at each level.
- *GPU Implementation.* We are studying how to implement our data structure and algorithms in graphics hardware processors, focusing on point-in-solid and distance tests.
- *Improvement of rendering quality.* We plan to study methods to improve the visual quality of the multiresolution visualization of the model. It might be necessary to propagate normals from the original model or remesh each level of the BP-Octree without taking into account the spatial index.
- *Deformable models.* We are going to study the suitability of BP-Octrees in deformable models collision detection.

Bibliografía

[ABJN85] D. Ayala, P. Brunet, R. Joan, and I. Navazo. Object representation by means of nonminimal division of quadtrees and octrees. *ACM Transaction on Graphics*, 4(1), 1985.

[Ali96] Daniel G. Aliaga. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96*, pages 101–106, October 1996.

[Bau72] Bruce G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford, CA, USA, 1972.

[BEH79] A. Baer, C. Eastman, and M. Henrion. Geometric modeling : A survey. *Computer Aided Design*, 11(5):253–272, 1979.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[Ben97] Van Der Bengen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[BM01] David E. Breen and Sean Mauch. 3d metamorphosis between different types of geometric models. *Computer Graphics Forum*, 20(3):36–48, 2001.

[BN85] Pere Brunet and Isabel Navazo. Geometric modelling using exact octree representation of polyhedral objects. In *Proc. of Eurographics'85*, pages 159–169, 1985.

[BN90] Pere Brunet and Isabel Navazo. Solid representation and operation using extended octrees. *ACM Trans. Graph.*, 9(2):170–197, 1990.

[BRM+02] Fausto Bernardini, Holly Rushmeier, Ioana M. Martin, Joshua Mittleman, and Gabriel Taubin. Building a digital model of michelangelo's florentine pietà. *IEEE Computer Graphics & Applications*, 22(1):59–67, January/February 2002.

[Can04] P. Cano. *SP-Octree: Representación Jerárquica de Sólidos Poliédricos.* PhD thesis, Univ. Granada, 2004.

[CCV85] I. Carlbom, I. Chakravarty, and D.A. Vandershcel. A hierarchical data structure for representing the spatial decomposition of 3d objects. *IEEE Comp. Graphics and Applications*, 5(4):24–31, 1985.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[CMRS03] Paolo Cignoni, Claudio Montani, Claudio Rocchini, and Roberto Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[CN96] Joao L. D. Comba and Bruce F. Naylor. *Conversion of Binary Space Partitioning Trees to Boundary Representation*, chapter Geometric Modeling: Theory and Practice: Chapter II - Representations, pages 286–301. Springer Verlag, 1996. ISBN 3-540-61883-X.

[CTV03] P. Cano, J.C. Torres, and F. Velasco. Progressive transmission of polyhedral solids using a hierarchical representation. *Journal of WSCG*, 11(1):81–86, 2003.

[DK89] M. J. Dürst and T. L. Kunii. Integrated polytrees: a generalized model for the integration of spatial decomposition and boundary representation. pages 329–348, 1989.

[DSSD99] Xavier Decoret, François Sillion, Gernot Schaufler, and Julie Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–72, 1999.

[DT81] L. J. Doctor and J. G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics & Applications*, 1:29–38, 1981.

[DYB98] Paul E. Debevec, Yizhou Yu, and George D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *Eurographics Rendering Workshop 1998*, pages 105–116, June 1998.

[Eri04] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann, December 2004.

[Eve] C. Everitt. *Projective Texture Mapping*.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM.

[FPRJ00] Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[FSG03] A. Fuhrmann, G. Sobottka, and C. Gross. Distance fields for rapid collision detection in physically based modeling. In *Proceedings of GraphiCon 03*, pages 58–65, Sept 2003.

[FT97] Francisco R. Feito and Juan Carlos Torres. Inclusion test for general polyhedra. *Computers & Graphics*, 21(1):23–30, 1997.

[FTI88] A. Fujimoto, Takayuki Tanaka, and K. Iwata. *ARTS: accelerated ray-tracing system*, pages 148–159. Computer Science Press, Inc., New York, NY, USA, 1988.

[FvFH90] J. D. Foley, A. vanDamm, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 2 edition, 1990.

[Gar82] I. Gargantini. Linear octrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20(4):365–374, 1982.

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *Eurographics Rendering Workshop 1998*, pages 181–192, June 1998.

[GEL00] Arthur D. Gregory, Stephen A. Ehmann, and Ming C. Lin. inTouch: Interactive multiresolution modeling and 3d painting with a haptic interface. In *VR*, pages 45–, 2000.

[GLM96] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Proceedings SIGGRAPH'96*, pages 171–180, 1996.

[Got96] S. Gottschalk. Separating axis theorem. Technical Report TR96-024, Department of Computer Science, UNC Chapel Hill, 1996.

[HAM02] Eric Haines and Tomas Akenine-Moller. *Real-Time Rendering (2nd Edition).* AK Peters, Ltd., July 2002.

[HKT01] J. Han, M. Kamber, and A. K. H. Tung. *Spatial Clustering Methods in Data Mining: A Survey.* Taylor and Francis, 2001.

[HKY99] Laurie J. Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring expression data: Identification and analysis of coexpressed genes. *Genome Res.*, 9(11):1106–1115, November 1999.

[Hop96] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM Press.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.

[Hop98] Hugues H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization '98*, pages 35–42, October 1998.

[HTG04] Bruno Heidelberger, Matthias Teschner, and Markus Gross. Detection of collisions and self-collisions using image-space techniques. In *Journal of WSCG*, pages 145–152, 2004.

[Hub96] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, 15(3):179–210, 1996.

[ICK+99] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286, August 1999.

[IG03] Martin Isenburg and Stefan Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM Transactions on Graphics*, 22(3):935–942, July 2003.

[ILGS03] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large mesh simplification using processing sequences. In *In Visualization'03 Proceedings*, pages 465–472, 2003.

[Jor87] C. Jordan. *Course DÁnalyse.* École Polytchnique de Paris, 1887.

[JT80] C.L. Jackins and S.L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. 14(3):249–270, November 1980.

[KHM+98] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[KR90] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data – An Introduction to Cluster Analysis.* John Wiley & Sons, 1990.

[KSD03] Laehyun Kim, Gaura S. Sukhatme, and Mathieu Desbrun. Haptic editing of decoration and material properties. In *HAPTICS '03: Proceedings of the 11th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (HAPTICS'03)*, page 213, Washington, DC, USA, 2003. IEEE Computer Society.

[KZ97] Petr. Konecný and Karel Zikan. Lower bound of distance in 3d. In *Proceedings of WSCG*, volume 3, pages 640–649, 1997.

[LD03] Stephen D. Laycock and A. M. Day. Recent developments and applications of haptic devices. *Comput. Graph. Forum*, 22(2):117–132, 2003.

[Lin00] Peter Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 259–262, July 2000.

[Män87] Martti Mäntylä. *An introduction to solid modeling.* Computer Science Press, Inc., New York, NY, USA, 1987.

[MCT04] F.J. Melero, P. Cano, and J.C. Torres. Visualización interactiva de sp-octrees utilizando impostores. In *Congreso Español de Informática Gráfica (CEIG)*, Sevilla, July 2004.

[MCT05] F.J. Melero, P. Cano, and J.C. Torres. Combining sp-octrees and impostors for multiresolution visualization. *Computer and Graphics*, 29:225–233, 2005.

[MCT07] F.J. Melero, P. Cano, and J.C. Torres. Transmisión progresiva de grandes modelos usando una jerarquía de planos envolventes. In E. Cerezo and F. Feito, editors, *XIV Congreso Español de Informática Gráfica (CEIG)*. Thomson-Paraninfo, 2007.

[MCT08] F.J. Melero, P. Cano, and J.C. Torres. Bounding-planes octree: A new volume-based lod scheme. *Computer and Graphics*, 32, 2008.

[Mea82] D.J.R. Meagher. Geometric modeling using octree encoding. *CGIP*, 19(2):129–147, June 1982.

[Men93] Jai Menon. The ray casting engine and ray representations for solid modeling. a research synopsis. In *ICCG '93: Proceedings of the IFIP TC5/WG5.2/WG5.10 CSI International Conference on Computer Graphics*, pages 241–258, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[Mor66] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[Mor85] Michael E. Mortenson. *Geometric modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[MS95] Paulo W. C. Maciel and Peter S. Shirley. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, pages 95–102, April 1995.

[NH94] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. Technical report, Vancouver, BC, Canada, Canada, 1994.

[Ogá06] C. Ogáyar. *Optimización de Algoritmos Geométricos Básicos Mediante el Uso de Recubrimientos Simpliciales*. PhD thesis, Univ. Granada, 2006.

[Oli84] M. A. Oliver. Two display algorithm for octrees. In *Eurographics '84*, pages 251–264, 1984.

[O'R94] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1st edition, 1994.

[OSF05] Carlos J. Ogáyar, Rafael Jesús Segura, and Francisco R. Feito. Point in solid strategies. *Computers & Graphics*, 29(4):616–624, 2005.

[Qui94] Sean Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 3324–3329, May 1994.

[Req80] Aristides A. G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Comput. Surv.*, 12(4):437–464, 1980.

[RR99] J. Rossignac and A. Requicha. *Encyclopedia of Electrical and Electronics Engineering*, chapter Solid modeling. John Wiley and Sons, 1999.

[RUL00] J. Revelles, C. Ureña, and M. Lastra. An efficient parametric algorithm for octree traversal. In *Proc. Winter School on Computer Graphics*, pages 212–219, 2000.

[RV83] A. G. Requicha and H. Voelcker. Solid modeling: Current status and research directions. *IEEE Comput. Graph. Appl.*, 3(7):25–37, 1983.

[SA94] Mark Segal and Kurt Akeley. The design of the opengl graphics interface. Technical report, Silicon Graphics Computer Systems, 1994.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures (Addison-Wesley series in computer science).* Addison-Wesley Pub (Sd), 1990.

[SCB04] K. Salisbury, F. Conti, and F. Barbagli. Haptic rendering: introductory concepts. *Computer Graphics and Applications, IEEE*, 24(2):24–32, 2004.

[SDB97] François X. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):207–218, August 1997.

[Set96] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proc. Nat. Acad. Sci*, pages 1591–1595, 1996.

[Sha02] Vadim Shapiro. *Handbook of Computer Aided Geometry Design*, chapter Solid Modeling, pages 473–518. Elsevier, 2002.

[SKv+92a] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH'92*, pages 249–252, 1992.

[SKv+92b] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 249–252, July 1992.

[SW88] Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics. part i. *IEEE Comput. Graph. Appl.*, 8(3):48–68, 1988.

[TC96] C. Tzafestas and P. Coiffet. Real-time collision detection using spherical octrees. In *Proceedings 1996 IEEE International Workshop on Robot and Human Communication (ROMAN 96)*, November 1996.

[TN87] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph.*, 21(4):153–162, 1987.

[WK03] Jianhua Wu and Leif Kobbelt. Piecewise linear approximation of signed distance fields. In *VMV*, pages 513–520, 2003.

[Zac00] Gabriel Zachman. *Virtual Reality in Assembly Simulation: Collision Detecion, Simulation Algorithms and Interaction Techniques.* PhD thesis, Dept. Computer Science, 2000.