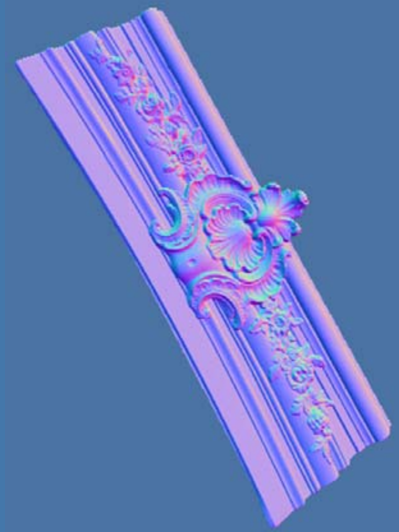
Tesis Doctoral

Interacción háptica en tiempo real con modelos gráficos de alta resolución



Autor:
Ángel Aguilera García

Directores:
Francisco Ramón Feito Higueruela
Francisco Javier Melero Rus

Dpto. Lenguajes y sistemas informáticos
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

2017

Programa de doctorado: Tecnologías de la Información y la Comunicación (D18.56.1)

Editor: Universidad de Granada. Tesis Doctorales
Auitor: Ánguel Aguilera García
ISBN: 978-84-9163-431-7
URI: http://hdl.handle.net/10481/48065

Tesis Doctoral
Ingeniería de Informática

UNIVERSIDAD
DE GRANADA

Interacción háptica en tiempo real con modelos gráficos de alta resolución

Autor:

Ángel Aguilera García

Directores:

Francisco Ramón Feito Higueruela

Catedrático de Universidad

Francisco Javier Melero Rus

Profesor Contratado Doctor

Dpto. de Lenguajes y Sistemas Informáticos

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Programa Oficial de Doctorado en Tecnologías de la Información y la Comunicación (D18.56.1)

Universidad de Granada

Granada, 2017

Tesis doctoral: Interacción háptica en tiempo real con modelos gráficos de alta resolución

Autor: Ángel Aguilera García

Tutores: Francisco Ramón Feito Higueruela

Francisco Javier Melero Rus

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente: Pere Bruinet Crosa

Vocales: Miguel Ángel Otaduy Tristan

Lidia Ortega Alvarado

Sergio Damas Arroyo

Secretario: Carlos Ureña Almagro

Acuerdan otorgarle la calificación de:

Granada, 2017

El Secretario del Tribunal

El doctorando / The *doctoral candidate* **[Ángel Aguilera García]** y los directores de la tesis / and the thesis supervisor/s: **[Francisco Ramón Feito Higueruela y Francisco Javier Melero Rus]**

Garantizamos, al firmar esta tesis doctoral, que el trabajo ha sido realizado por el doctorando bajo la dirección de los directores de la tesis y hasta donde nuestro conocimiento alcanza, en la realización del trabajo, se han respetado los derechos de otros autores a ser citados, cuando se han utilizado sus resultados o publicaciones.

/

Guarantee, by signing this doctoral thesis, that the work has been done by the doctoral candidate under the direction of the thesis supervisor/s and, as far as our knowledge reaches, in the performance of the work, the rights of other authors to be cited (when their results or publications have been used) have been respected.

Lugar y fecha / Place and date:

Granada, 14 de junio de 2017

Director/es de la Tesis / *Thesis supervisor/s;* Doctorando / *Doctoral candidate*:

FEITO HIGUERUELA FRANCISCO RAMON -

Firmado digitalmente por FEITO HIGUERUELA FRANCISCO RAMON -
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-
givenName=FRANCISCO RAMON, sn=FEITO HIGUERUELA, cn=FEITO HIGUERUELA FRANCISCO RAMON -
Fecha: 2017.06.14 19:35:33 +02'00'

MELERO RUS FRANCISCO JAVIER -
c=ES,
serialNumber:
givenName=FRANCISCO JAVIER, sn=MELERO RUS, cn=MELERO RUS FRANCISCO JAVIER -
2017.06.16 09:25:03 +01'00'

NOMBRE AGUILERA GARCIA ANGEL INOCENCIO - NIF

Firmado digitalmente por NOMBRE AGUILERA GARCIA ANGEL INOCENCIO - NIF
Nombre de reconocimiento (DN): c=ES, o=FNMT, ou=FNMT Clase 2 CA, ou=, cn=NOMBRE AGUILERA GARCIA ANGEL INOCENCIO -
Fecha: 2017.06.15 11:19:08 +02'00'

Firma / Signed Firma / Signed

A Lupe, Ángel y Pablo por todo el tiempo que no les he dedicado debido a la realización de este trabajo.

A mis padres y hermanos por todos los ánimos y apoyos recibidos.

Agradecimientos

Tras muchos años de esfuerzo y muchas horas de dedicación he completado este trabajo son muchas las personas a las que les tengo que agradecer todo el apoyo que me han prestado. La lista es muy amplia y necesitaría muchos folios para poder nombrarlos a todos, además de tener el despiste de no incluir a alguien. Por esto quiero daros las gracias a todos los que en algún momento me habéis dado vuestro apoyo, vuestro consejo y vuestros ánimos. Muchas gracias de todo corazón.

A pesar de todo permitirme dedicarles unas líneas a aquellas que más de cerca han vivido el desarrollo de este trabajo.

En primer lugar, a mis directores de tesis por la gran paciencia que han tenido conmigo, ya que sin ellos todo este trabajo nunca se habría podido realizar. A Paco por estar siempre detrás de mí y prestarme todo su apoyo y comprensión. Y de Javier, ¿qué puedo decir de él? Me faltan las palabras para poder agradecer todos los momentos dedicados al desarrollo de este trabajo, sabiendo que siempre estaba ahí para solucionar cualquier problema, pudiendo contar con él cualquier día y hora de la semana.

A Paco Soler, por todos los buenos ratos que pasamos juntos y los cafés que nos tomamos en mis estancias de investigación, y cómo no, por todos sus consejos y ayudas recibidas para la implementación de la aplicación y para la mejora de la estructura, todo ello de manera altruista. Muchas gracias Paco. No sé cómo podré compensarte por tanto esfuerzo.

A todos mis compañeros de departamento de informática de la Universidad de Jaén y en especial a los de Linares, ya que ellos han sabido animarme y escucharme siempre que lo he necesitado.

A mis suegros Concha y Manuel. Gracias por estar siempre al quite y cuidar de mis hijos, ya que sin ellos no le habría podido dedicar tanto tiempo a este trabajo.

Una mención muy especial a mis padres Carmen e Inocencio, gracias por ser como sois, por estar ahí siempre que os he necesitado, por todos los ánimos recibidos, por confiar en mí y por mostrarme siempre vuestro cariño y comprensión.

A mi querido hermano. ¿Qué te puedo decir? Necesitaría otra memoria para agradecerte todo lo que has hecho por mí a lo largo de mi vida. Muchas gracias por estar siempre pendiente y dispuesto a ayudarme en todo lo que he necesitado, por todos tus consejos, tus explicaciones, por ayudarme en la toma de decisiones transcendentales a lo largo de mi vida, sin las cuales nunca hubiera llegado hasta aquí.

Y por último a mi esposa Lupe y a mis hijos Ángel y Pablo. Ellos han sido los que han sufrido más de cerca la realización de este trabajo y a los que no les he podido dedicar todo el tiempo que se merecían. Lupe muchas gracias por toda tu comprensión y todos tus ánimos, por compartir conmigo todos los momentos, buenos y malos, de nuestra vida juntos. Espero poder compensártelo tal y como tú te lo mereces.

Agradecer también la cesión de los modelos utilizados para realizar las pruebas de este trabajo al Museo Histórico de Écija por el modelo de la Amazona Herida, a la empresa AGEO por el modelo de la Moldura y al Stanford 3D Scanning Repository por los modelos de Lucy y Armadillo.

Este trabajo ha sido parcialmente subvencionado con los proyectos de investigación del Ministerio Español de Educación, Cultura y Deporte y la Union Europea (via ERDF) codificados como TIN2014-58218-R y TIN2013-47276-C6-3R.

Resumen

En nuestros días, la utilización de la tecnología informática se ha generalizado en muchas disciplinas, como pueden ser la medicina, los videojuegos, los dispositivos hápticos, la realización de películas, el diseño y la fabricación de objetos, la planificación de rutas para la movilidad, etc. Principalmente destacan, el desarrollo de la realidad aumentada, la realidad virtual y la tecnología háptica.

Para todas estas tecnologías cada día van apareciendo nuevos dispositivos que permiten interactuar dentro de escenarios reales o virtuales por lo que se hace necesario trabajar con modelos geométricos cada vez mayores y más complejos

En la actualidad los escáneres para la captura de modelos geométricos han abaratado mucho los precios y han aumentado mucho sus prestaciones, permitiendo obtener modelos geométricos en tres dimensiones con gran resolución, recogiendo de esta forma los más mínimos detalles de los objetos representados.

También cabe destacar que al igual que la tecnología ha avanzado mucho en el desarrollo de los escáneres de modelos geométricos también ha evolucionado bastante en las tecnologías complementarias de los escáneres, como puede ser las impresoras en tres dimensiones. Actualmente en el mercado se pueden encontrar una gran variedad de éstas, las cuales se están utilizando en una gran diversidad de campos, destacando sus usos dentro de la industria y de la medicina.

Por todo lo expuesto anteriormente, hoy en día los modelos geométricos con los que se trabaja tienen una gran resolución y aunque los ordenadores cada vez sean más rápidos y potentes, el trabajar con estos modelos directamente sin utilizar ningún método de optimización se hace inviable si se espera que el tiempo de respuesta sea muy rápido o cercano al tiempo real. En esta tesis se propone la utilización de una estructura jerárquica de volúmenes envolventes para la detección de colisiones entre grandes modelos geométricos. Dicha estructura permite trabajar con modelos

formados por varias decenas de millones de triángulos obteniendo un tiempo de respuesta muy rápido y válido para ser utilizado con dispositivos hápticos.

Como formación previa a esta tesis se estuvo trabajando en la representación de terrenos multirresolución (Aguilera, Feito, and Torres 2010; Aguilera, Torres, and Feito 2003; Aguilera, Torres, and Feito 2001) lo que me sirvió para familiarizarme con el uso de mallas y la representación de estas a diferente nivel de detalle.

Parte del trabajo expuesto en esta memoria de tesis doctoral ha sido publicado previamente en el Congreso Español de Informática Gráfica (CEIG'13) (Aguilera, Feito, and Melero 2013) y en la revista Computer-Aided Design (CAD August 2016) (Aguilera, Melero, and Feito 2016).

Abstract

Nowadays, computer graphics technologies have become widespread in many disciplines, such as medicine, video games, haptic devices, filmmaking, design and manufacture of objects, planning of routes for mobility, etc. Mainly highlight, the development of augmented reality, virtual reality and haptic technology.

For all these technologies, new devices that allow interaction within real or virtual scenarios appear daily, making it necessary to work with geometric models that are increasingly larger and more complex

Currently the 3D scanners for the capture of geometric models have greatly reduced prices and have greatly increased their performance, allowing to obtain geometric models in three dimensions with high resolution, thus collecting the smallest details of the represented objects.

It should also be noted that technology has advanced a lot not only in the development of acquisition devices, but also it has also evolved quite a bit in the complementary technologies of scanners, such as 3D printers. Currently, in the market we can find a great variety of these devices, which are being used in a great diversity of fields, highlighting its benefits in the industry and medicine.

For all the above, nowadays the geometric models have a high resolution and although computers are faster and more powerful, working with these models without using any method of optimization becomes unnafordable if me expect the response time to be very fast or close to real time. This thesis proposes the use of a hierarchical structure of bounding volumes for the detection of collisions between large geometric models, as those coming from 3D scanning processes. This structure allows to work with models formed by several tens of millions of triangles obtaining a very fast response time, even faster than the required haptic rendering refresh frequency (1KHz).

As previous research duties to this thesis I have been researching in the representation of multiresolution terrains (Aguilera, Feito, and Torres 2010; Aguilera, Torres, and Feito 2003; Aguilera, Torres, and Feito 2001). These works helped me to familiarize with the use of meshes and their representation at different levels of detail.

A part of the work presented in this dissertation report has previously been published in the Spanish Congress of Computer Graphics (CEIG'13) (Aguilera, Feito, and Melero 2013) and Journal on Computer-Aided Design (CAD August 2016) (Aguilera , Melero, and Feito 2016).

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xv**

Índice de Figuras **xix**

Índice de tablas **xxvii**

Índice de Gráficos **xxxi**

Índice de algoritmos **xxxiii**

Índice de ecuaciones **xxxv**

1. Introducción **37**

1.1. ¿Qué queremos resolver? *38*

1.2. ¿Qué hemos conseguido? *40*

1.3. Estructura de la memoria. *41*

2. Detección de colisiones **43**

2.1. Definiciones previas. *44*

2.1.1. Colisión. Definición y ámbito del problema 44

2.1.2. Objeto sólido. 45

2.1.3. Representación de sólidos más utilizadas. 47

2.2. Introducción. *51*

2.2.1. Detección de colisiones con objetos no convexos. 53

2.3. Elección de la representación de los objetos. *68*

2.3.1. Estructuras de datos para la detección de colisiones. 70

2.3.2. Conclusiones sobre las estructuras para la detección de

colisiones. 99
2.4. Estructuras de datos y algoritmos para detección de colisiones entre objetos convexos. 100
2.4.1. Detección de colisiones en objetos B-rep 102
2.4.2. Algoritmos para detección de colisiones con objetos convexos 104
2.5. Librerias y aplicaciones de código abierto para la Detección de Colisiones 108
2.6. Dispositivos hápticos 121
2.6.1. Interfaces hápticos 121
2.6.2. Sistema háptico 122
2.6.3. Percepción háptica en las personas 122
2.6.4. Aspectos algorítmicos del renderizado háptico. 124
2.6.5. Principales dispositivos hápticos en el mercado 125
2.6.6. Clasificación de los dispositivos hápticos atendiendo a la funcionalidad de los mismos 126

3. La estructura EBP-Octree 131
3.1. Fundamentos del EBP-Octree 132
3.2. Cálculos preliminares para la construcción del EBP-Octree. 140
3.2.1. Cálculo de la caja envolvente del modelo. 140
3.2.2. Creación del índice espacial. 141
3.2.3. Código Morton. 143
3.2.4. Cálculo del número de niveles del EBP-Octree. 145
3.3. Cálculo de los nodos hoja. 148
3.3.1. Clasificación de los polígonos en nodos hoja. 148
3.3.2. Gestión out of core de la lista de nodos hoja. 153
3.3.3. Construcción de los nodos hoja. 154
3.3.4. Creación de las envolventes convexas de los nodos hoja. 154
3.4. Construcción del EBP-octree completo. 162
3.4.1. Cálculo de la envolvente de los nodos grises por niveles del EBP-Octree. 165
3.4.2. Cálculo de los nodos blancos y negros. 165
3.5. Gestión del EBP-Octree con memoria externa. 166
3.5.1. Inversión del EBP-Octree calculado out-of-core. 168
3.5.2. Carga del EBP-Octree en memoria. 169
3.6. Elección del método de selección de los planos relevantes para el cálculo de las envolventes. 171

3.6.1. Resultados visuales por niveles de las envolventes obtenidas. 172
3.6.2. El EBP-Octree en disco. 180
3.6.3. Tamaño del EBP-Octree por niveles. 183
3.6.4. Tiempo de creación del EBP-Octree. 186
3.6.5. Tiempo de carga del EBP-Octree en memoria principal. 191
3.6.6. Volumen por nivel de las envolventes calculadas. 192
3.6.7. Estudio de los resultados obtenidos. 198
3.7. Elección del porcentaje de planos relevantes para el cálculo de las envolventes. 199
3.7.1. Resultados visuales por niveles de las envolventes obtenidas. 200
3.7.2. El EBP-Octree en disco. 207
3.7.3. Tamaño del EBP-Octree por niveles. 208
3.7.4. Tiempo de creación del EBP-Octree. 210
3.7.5. Tiempo de carga del EBP-Octree en memoria. 211
3.7.6. Volumen por nivel de las envolventes calculadas. 212
3.7.7. Conclusión obtenida tras el estudio de los resultados. 213
3.8. Resumen. 214

4. Inclusión punto en sólido 215
4.1. Inclusión punto en sólido utilizando el EBP-Octree. 216
4.2. Test de inclusión punto en sólido. 216
4.2.1. Test de inclusión punto en sólido usando las envolventes de los nodos. 217
4.2.2. Test de inclusión punto en solido sin usar las envolventes de los nodos. 219
4.2.3. Test de inclusión punto en sólido usando una caché de subárboles. 220
4.2.4. Test inclusión punto en sólido con carga de sólo los nodos que se encuentran en el camino para llegar al nodo que contiene al punto. 222
4.2.5. Test de inclusión punto en sólido con carga del subárbol que contiene al punto que se desea comprobar. 223
4.2.6. Test inclusión punto en sólido que llega hasta un nivel fijo del EBP-Octree. 224
4.3. Tipos de test de inclusión punto en sólido realizados. 224
4.3.1. Elección del conjunto de puntos para la realización del test de inclusión punto en sólido. 226
4.3.2. Resultados obtenidos por los diferentes tipos de test de

inclusión punto en sólido. 226
4.3.3. Estudio comparativo de tiempos para la elección del mejor test de inclusión punto en sólido. 229
4.3.4. Gestión de la memoria principal por parte del EBP-Octree durante el renderizado háptico. 235
4.4. Test de cálculo de distancias y direcciones de colisión entre un punto y un modelo. 236
4.4.1. Resultados obtenidos por los diferentes tipos de test de cálculo de distancias. 238
4.5. Utilización de los EBP-Octree en aplicaciones con interacción háptica. 244

5. Detección de colisiones entre dos modelos 247
5.1. Introducción. 248
5.2. Detección de colisiones entre dos modelos utilizando EBP-Octree. 249
5.3. Test de detección de colisiones entre modelos utilizando EBP-Octree. 250
5.4. Algoritmos de detección de colisiones entre modelos. 252
5.4.1. Algoritmo de detección de colisiones para modelos con igual número de niveles. 255
5.4.2. Algoritmo detección de colisiones para modelos con distinto número de niveles. 260
5.5. Optimización sobre los algoritmos de detección de colisiones. 261
5.6. Estudio de tiempos obtenidos en el cálculo de la detección de colisiones. 262
5.6.1. Cálculo de la escena 265
5.6.2. Movimiento de los modelos por la escena 266
5.6.3. Resultados de los test con solamente detección de colisiones. 267
5.6.4. Resultados de los test con detección e identificación de todas las colisiones 269
5.7. Comparación de los resultados obtenidos con el EBP-Octree con otras aplicaciones o librerías de detección de colisiones. 275

6. Conclusiones y trabajos futuros. 277
6.1. Conclusiones y principales aportaciones. 277
6.2. Trabajos futuros. 279

Bibliografía 281

ÍNDICE DE FIGURAS

Figura 1: a) Objeto sólido. b) Caras en las que se descompone el sólido. 47

Figura 2. Composición de objeto con celdas básicas. 49

Figura 3. Representación de un toroide mediante voxels. © A.H.J. Christensen, 1980 50

Figura 4. Ejemplo en 2-D de tipos de estructuras con forma de rejilla (Schroeder, Martin, and Lorensen 1996). 72

Figura 5. a) Subdivisión recursiva de un cubo en octantes. b) Octree correspondiente.("Octree," n.d.) 74

Figura 6. Numeración de los nodos hijo de un nodo padre (Samet and Webber 1988). 75

Figura 7. Ejemplos de división en un point-quatree (Brunet et al. 1999). 77

Figura 8. Ejemplo de kd-tree en dos dimensiones (Bentley 1975). 78

Figura 9. Ejemplo de particionamiento del espacio llevado a cabo por un kd-tree ("Árbol Kd," n.d.)..78

Figura 10. BSP de un poliedro (Brunet et al. 1999). 81

Figura 11. BPS de clasificación de un conjunto de objetos en 2D (Naylor, Amanatides, and Thibault 1990; Fuchs, Kedem, and Naylor 1980). 82

Figura 12. Ejemplo de un 4-gono en un plano 2D (Held, Klosowski, and Mitchell 1996). ..83

Figura 13. Dibujado de volúmenes obtenidos con diferentes esferas de barrido: a) Un punto con una esfera de barrido. b) Línea obtenida con una esfera de barrido. c) Rectángulo obtenido con una esfera de barrido. (Larsen et al. 1999) 85

Figura 14. Ejemplo de cálculo de regiones de Voronoi entre dos objetos (M. C. Lin 1993)...86

Figura 15. Formas de los volúmenes envolventes (Mohd Suaib, Bade, and Mohamad 2008).

.... 87

Figura 16. Esferas sin colisión y con colisión. 88

Figura 17. Caja AABB con los puntos máximo y mínimo que la representan. 89

Figura 18. Punto central, tamaño de los lados y las nuevas coordenadas. 90

Figura 19. Proyección de dos OBB sobre un eje separador("Collision Detection (Advanced Methods in Computer Graphics)," n.d.). 90

Figura 20. Ejemplos de K-dop de un objeto ("Add a K-DOP Collision Hull to a Static Mesh," n.d.). 91

Figura 21. Ejemplo de un R-tree de un objeto (Brunet et al. 1999). 92

Figura 22. Construcción top-down de la jerarquía de datos. 93

Figura 23 . Construcción bottom-up de la jerarquía de datos. 94

Figura 24. Construcción por Inserción de la jerarquía de datos. 94

Figura 25. Ejemplo de Sphere Tree basado en Octree y Sphere Tree basado en Medial Axis. (Hubbard 1996) 95

Figura 26. Modelo original y los tres primeros niveles del Sphere tree basados en la división planteada por Hubbard (Hubbard 1996). 96

Figura 27. Malla triangular de una pieza y el AABBTree construido (Alliez, Tayeb, and Camille, n.d.). 97

Figura 28. Proceso de construcción del OBBTree. (Gottschalk et al. 1996). 98

Figura 29. (a) Dos objetos convexos: la distancia mínima local entre dos puntos es siempre un mínimo global. (b) Dos objetos cóncavos: la distancia mínima local entre dos puntos (en gris) no es necesariamente un mínimo global (en negro). (Ericson 2005) 102

Figura 30. (i) Búsqueda de vector separador, (ii) la idea, (iii) en el caso de círculos eligiendo S_{i+1} (Chung and Wang 1996) 105

Figura 31. Arriba a la izquierda: Un poliedro cúbico. Entre sus características se encuentran la cara F, la arista E y el vértice V. Arriba a la derecha: la región Voronoi VR (V). Uno de los planos de Voronoi que delimitan esta región es VP (V, E), correspondiente a la arista vecina E. Abajo izquierda: la región de Voronoi VR (E). Dos de los planos de Voronoi que delimitan esta región son VP (V, E) y VP (F, E), correspondientes respectivamente a las características

vecinas de E, V y F. Abajo a la derecha: la región de Voronoi VR (F). Uno de los planos de Voronoi que delimitan esta región es VP (F, E), correspondiente a la arista E vecina de F. El plano de soporte de F mismo también limita VR (F). (Mirtich and June 1997) 106

Figura 32. Ejemplo de construcción de una jerarquía de Dobkin-Kirkpatrick. (Dobkin and Kirkpatrick 1990) 107

Figura 33. Clasificación frecuencial frente a los estímulos (Shimoga, n.d.). 123

Figura 34. Esquema de la división en bloques del renderizado háptico y de la visualización (Salisbury, Conti, and Barbagli 2004). 124

Figura 35. Sistema bidireccional de un dispositivo háptico. 126

Figura 36. Distintos dispositivos clasificados según su funcionalidad. 127

Figura 37. 3D Touch™ Haptic 3D Stylus 128

Figura 38. Geomagic® Touch™ Haptic Device. 129

Figura 39. Geomagic® Touch™ X Haptic Device 129

Figura 40. Phantom Premium Haptic Devices 130

Figura 41. 3D Systems Phantom® Premium™ 6DOF 130

Figura 42. Posición de los nodos grises separadores en el EBP-Octree. 134

Figura 43. Nodo hoja. Los triángulos rosas se corresponden con los polígonos del sólido, los polígonos de color cian forman la envolvente convexa calculada y los de azul determinan los polígonos añadidos de las paredes del nodo. 135

Figura 44. Esquema gráfico de la estructura de un EBP-Octree. 136

Figura 45. Ejemplo de las envolventes generadas para los primeros 10 niveles del modelo de la Amazona Herida de 28 millones de polígonos 139

Figura 46. Puntos significativos para representar una AABB. 141

Figura 47. Clasificación de puntos en nodos. 142

Figura 48. Código Morton con información del nivel. 144

Figura 49. Ejemplo de código Morton con información del nivel a la izquierda 144

Figura 50. Octcode de 64 bit. 145

Figura 51. Proceso de calculo de nodos hoja cortados por un polígono. 151

Figura 52. Ejemplo del proceso de cálculo de todos los nodos hoja que son cortados por un polígono. 151

Figura 53. Estructura de archivos temporales para la indexación espacial. 153

Figura 54. Arista común a dos nodos. Sólo se calculan los puntos de corte una vez. 155

Figura 55. Distribución de las aristas en tres grupos. 156

Figura 56. Nodo hoja cuyas aristas no son cortadas por los polígonos de la frontera del sólido. 158

Figura 57. Parte de atrás de la Moldura 26 M.P. 160

Figura 58. Octree que no cabe en memoria y ha sido generado partido. 163

Figura 59. Creación de nodos para el octcode 37|34256543456)$_8$ 164

Figura 60. Estructura de archivos para almacenar un EBP-Octree en disco. 168

Figura 61. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 10 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 172

Figura 62. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 10 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 173

Figura 63. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 10 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 173

Figura 64. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 20 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 174

Figura 65. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 20 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 174

Figura 66. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 20 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente

con respecto al volumen real del modelo. 175

Figura 67. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 30 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 175

Figura 68. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 30 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 176

Figura 69. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 30 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 176

Figura 70. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 40 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 177

Figura 71. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 40 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 177

Figura 72. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 40 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 178

Figura 73. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 50 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 178

Figura 74. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 50 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 179

Figura 75. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 50 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo. 179

Figura 76. Moldura de 26 M.P. obtenida con el método de selección de planos k-mediana y tomando el 30% de los planos. 201

Figura 77. Moldura de 26 M.P. obtenida con el método de selección de planos k-mediana y

tomando el 40% de los planos. 202

Figura 78. Detalle de la moldura de 26 M.P. a nivel 6 obtenido con el método de selección de planos k-mediana y tomando el 30% de los planos. 202

Figura 79. Lucy de 28 M.P. obtenida con el método de selección de planos k-mediana y tomando el 30% de los planos. 204

Figura 80. Lucy de 28 M.P. obtenida con el método de selección de planos k-mediana y tomando el 40% de los planos. 205

Figura 81. Detalle del ojo de Lucy de 28 M.P. a nivel 6 obtenido con el método de selección de planos k-mediana y tomando el 30% y 40% de los planos. 206

*Figura 82. Nodo central donde se encuentra el punto y los veintisiete nodos de alrededor.*221

Figura 83. Planificación de la caché en un EBP-Octree. 222

Figura 84. Almacenamiento en memoria principal de los nodos necesarios para saber si un punto está incluido o no en un objeto. 223

Figura 85. Almacenamiento en memoria principal del subárbol que contiene al punto. 223

Figura 86. Nivel del árbol en el que se da la respuesta a la inclusión o no de un punto en un objeto. 224

Figura 87. Resultado de los test de inclusión punto en sólido sobre un millón de puntos aleatorios. 232

Figura 88. Resultado de ejecutar los test de inclusión punto en sólido por niveles sobre un millón de puntos aleatorios. 233

Figura 89. Resultado de ejecutar los test de inclusión punto en sólido sobre un millón de puntos de movimiento continuo. 234

Figura 90. Mapa de distancias para el modelo Amazona Herida de 28 M.P. 242

Figura 91. Mapa de distancias para el modelo Moldura de 26 M.P. 243

Figura 92. Mapa de distancias para el modelo, Lucy de 28 M.P. 244

Figura 93. Amazona Herida con el ojo pintado por la aplicación de simulación háptica. .. 245

Figura 94. En verde primer modelo, en azul segundo modelo y en rojo los polígonos que intersecan. 248

Figura 95. EBP-Octrees con similares niveles de corte y máximos. 252

Figura 96. Primer EBP-Octree con nivel de corte 5 y máximo 11 y segundo con nivel de corte 4 y máximo 9. 252

Figura 97. Detección de colisiones entre dos objetos en 2D. 254

Figura 98. Máquina de estados para calcular colisión entre dos modelos con igual número de niveles en el EBP-Octree. 256

Figura 99. Máquina de estados para calcular colisión entre dos modelos con distinto número de niveles en el EBP-Octree. 260

Figura 100. Cola sin sus celdas ordenadas. 261

Figura 101. Cola ordenada por el valor del primer número de cada pareja. 261

Figura 102. Tabla hash con los nodos del segundo modelo. 262

Figura 103. Posiciones inicial y final de las Amazonas Heridas en el escenario 1. 264

Figura 104. Posiciones inicial y final de las Lucys en el escenario 2. 264

Figura 105. Posiciones inicial y final de Amazona Herida y Armadillo en el escenario 3. 264

Figura 106. Posiciones inicial y final de Moldura y Armadillo en el escenario 4. 265

Figura 107. Cajas envolventes del escenario y de movimiento para los modelos. 266

Figura 108. Armadillo con la mano metida dentro de Amazona Herida. 273

Figura 109. Armadillo con los pies dentro de la Moldura. 274

Figura 110. Armadillo con manos y piernas metidas dentro de Lucy. 274

ÍNDICE DE TABLAS

Tabla 1. Tamaño en Kbyte del árbol raíz, los nodos en disco y los subárboles. 170

Tabla 2. Tamaño, en Bytes, de los archivos obtenidos para la Amazona Herida de 28 M.P. seleccionando los planos con menor offset. 180

Tabla 3. Tamaño, en Bytes, de los archivos obtenidos para la Amazona Herida de 28 M.P. seleccionando los planos aleatoriamente. 181

Tabla 4. Tamaño, en Bytes, de los archivos obtenidos para la Amazona Herida de 28 M.P. seleccionando los planos con el método de K-mediana. 181

Tabla 5. Ocupación en memoria, en Kbyte, del EBP-Octree por niveles utilizando el método de selección de planos de menor offset. 184

Tabla 6. Ocupación en memoria, en Kbyte, del EBP-Octree por niveles utilizando el método de selección de planos aleatorio. 184

Tabla 7. Ocupación en memoria, en Kbyte, del EBP-Octree por niveles utilizando el método de selección de planos k-mediana. 185

Tabla 8. Distribución de tiempos, en segundos, para construir el EBP-Octree utilizando la selección de planos con menor offset. 187

Tabla 9. Distribución de tiempos, en segundos, para construir el EBP-Octree utilizando la selección de planos aleatorios. 188

Tabla 10. Distribución de tiempos, en segundos, para construir el EBP-Octree utilizando la selección de planos con el método de k-mediana. 189

Tabla 11. Tiempos de carga, en segundos, del EBP-Octree en memoria principal. 191

Tabla 12. Tanto por ciento de ocupación de las envolventes con respecto al volumen real del modelo. Datos obtenidos utilizando el método de selección de los planos con menor offset. 193

Tabla 13. Tanto por ciento de ocupación de las envolventes con respecto al volumen real del

modelo. Datos obtenidos utilizando el método de selección de los planos de manera aleatoria. 194

Tabla 14. Tanto por ciento de ocupación de las envolventes con respecto al volumen real del modelo. Datos obtenidos utilizando el método de selección de los planos k-mediana. 195

Tabla 15. Tamaño en Bytes de los archivos generados. 207

Tabla 16. Ocupación en Mbyte del EBP-Octree por niveles. 209

Tabla 17. Datos relevantes con respecto a la ocupación de memoria del EBP-Octree. 209

Tabla 18. Tiempos, en segundos, en construir el EBP-Octree para los tres modelos utilizando el método del k-mediana y seleccionando el 30% y el 40% de los planos. 211

Tabla 19. Tiempo, en segundos, en cargar el EBP-Octree en memoria. 211

Tabla 20. Porcentajes del volumen de las envolventes con respecto al volumen real del modelo. 212

Tabla 21. Tiempo en segundos de los tests sin nivel límite y distribución puntos aleatoria. 227

Tabla 22. Tiempo en segundos de los tests sin nivel límite y distribución puntos movimiento continuo. 227

Tabla 23. Tiempo en segundos de los tests con nivel límite y distribución puntos aleatoria. 228

Tabla 24. Tiempo en segundos de los tests con nivel límite y distribución puntos movimiento continuo. 229

Tabla 25. Tiempo medio en segundos de los tests sin nivel límite. 230

Tabla 26. Tiempo medio en segundos de los tests con nivel límite. 230

Tabla 27. Tiempo medio, en segundos, de ejecutar 1 millón de test de cálculo de distancias sobre los tres modelos. 240

Tabla 28. Media de tiempos obtenidos en el cálculo de la distancia entre un punto y un modelo. 241

Tabla 29. Escenarios creados para el estudio de tiempos de colisión entre modelos. 263

Tabla 30. Resultados obtenidos con los algoritmos de detección de colisiones sin optimizar.

.... 270

*Tabla 31. Resultados obtenidos con los algoritmos de detección de colisiones optimizados.*270

Tabla 32. Modelos incluidos en los distintos escenarios. 276

Tabla 33. Tiempo medio en segundos para realizar el cálculo de la colisión entre dos modelos. 276

ÍNDICE DE GRÁFICOS

Gráfico 1. Comparación del tamaño de los archivos obtenidos para la Amazona Herida de 28 M. P. por los tres métodos de cálculo. 182

Gráfico 2. Tamaño de los archivos ".bpl", en Megabytes, para la Amazona Herida de 28 M.P. 182

Gráfico 3. Tamaño de los archivos ".bvp", en Megabytes, para la Amazona Herida de 28 M.P. 183

Gráfico 4. Suma del tamaño de los 5 primeros niveles del EBP-Octree. 186

Gráfico 5. Tiempos por etapas, en segundos, para construir el EBP-Octree de la Amazona Herida de 28 M. P. seleccionando los planos con menor offset. 188

Gráfico 6. Tiempos por etapas, en segundos, para la construcción del EBP-Octree de la Amazona Herida de 28 M. P. seleccionando los planos de manera aleatoria. 189

Gráfico 7. Tiempos por etapas, en segundos, para la construcción del EBP-Octree de la Amazona Herida de 28 M. P. seleccionando los planos utilizando el método de k-mediana. 190

Gráfico 8. Comparación de tiempos totales, en segundos, en construir el EBP-Octree para la Amazona Herida de 28 M. P. 191

Gráfico 9. Tiempos de carga, en segundos, del EBP-Octree en memoria principal. 192

Gráfico 10. Porcentaje de volumen de la envolvente con respecto al volumen real de la Amazona Herida de 28 M. P. seleccionando los planos con menor offset. 193

Gráfico 11. Porcentaje de volumen de la envolvente con respecto al volumen real de la Amazona Herida de 28 M. P. seleccionando los planos de manera aleatoria. 194

Gráfico 12. Porcentaje de volumen de la envolvente con respecto al volumen real de la Amazona Herida de 28 M. P. seleccionando los planos con el método de k-mediana. 195

Gráfico 13. Comparación del porcentaje de ocupación del volumen de la envolvente con respecto al volumen del modelo, para los tres métodos, seleccionando un 30% de planos.. 196

Gráfico 14. Comparación del porcentaje de ocupación del volumen de la envolvente con

respecto al volumen del modelo, para los tres métodos, seleccionando un 40% de planos.. 197

Gráfico 15. Suma de volúmenes envolventes obtenidos para cada uno de los métodos, agrupados por porcentaje de planos seleccionados. 198

Gráfico 16. Suma de la ocupación de los archivos en disco para los tres modelos calculados por el método de k-mediana y seleccionado el 30% y el 40% de los planos. 208

Gráfico 17. Tamaño de los EBP-Octree en memoria... 210

Gráfico 18. Suma de los porcentajes de volúmenes de las envolventes para los tres modelos calculados por el método de k-mediana y seleccionando un 30% y un 40% de los planos. 213

Gráfico 19. Tiempo medio, en segundos, de ejecutar los test siguiendo los puntos una distribución aleatoria.... 231

Gráfico 20. Tiempo medio, en segundos, de ejecutar los test siguiendo los puntos una distribución de movimiento continuo.... 232

Gráfico 21. Tiempo en segundos de la detección de colisión entre dos Amazonas Heridas de 28 M.P... 267

Gráfico 22. Tiempo en segundos de la detección de colisión entre dos Lucys de 28 M.P.... 268

Gráfico 23. Tiempo en segundos de la detección de colisión entre Amazona Herida de 28 M.P. y Armadillo de 150 K.P. 268

Gráfico 24. Tiempo en segundos de la detección de colisión entre Moldura de 26 M.P. y Armadillo de 150 K.P.. 269

Gráfico 25. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 1 con el algoritmo optimizado... 271

Gráfico 26. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 2 con el algoritmo optimizado... 271

Gráfico 27. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 3 con el algoritmo optimizado... 272

Gráfico 28. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 4 con el algoritmo optimizado... 272

ÍNDICE DE ALGORITMOS

Algoritmo 1. Procedimiento para calcular el número de niveles del octree. 147

Algoritmo 2. Cálculo del nodo padre común que engloba un polígono. 149

Algoritmo 3. Cálculo de los nodos hoja cortados por un polígono. 152

Algoritmo 4. Cálculo puntos de corte de las aristas de los nodos hoja. 158

Algoritmo 5. Seudocódigo del algoritmo utilizado para calcular el k-mediana. 161

Algoritmo 6. Seudocódigo para el cálculo de la envolvente de un nodo gris en un octree partido. 165

Algoritmo 7. Seudocódigo para el cálculo del test inclusión punto en sólido utilizando las envolventes. 218

Algoritmo 8. Seudocódigo para el cálculo del test de inclusión punto en sólido sin utilizar las envolventes. 220

Algoritmo 9. Procedimiento Recursivo para el cálculo de la distancia mínima desde un nodo de un subárbol. 237

Algoritmo 10. Procedimiento Recursivo para el cálculo de la distancia mínima desde un nodo cargado en memoria. 238

Algoritmo 11. Cálculo recursivo de la distancia mínima entre un punto y un modelo. 239

Algoritmo 12. Seudocódigo recursivo del cálculo de colisiones entre dos modelos similares. 255

Algoritmo 13. Seudocódigo del estado de la máquina A, A. 257

Algoritmo 14. Seudocódigo del estado de la máquina NS, NS. 257

Algoritmo 15. Seudocódigo del estado de la máquina SA, SA. 258

Algoritmo 16. Seudocódigo del estado de la máquina PNH, PNH. 258

Algoritmo 17. Seudocódigo del estado de la máquina NH, NH. 258

Algoritmo 18. Cálculo de colisiones entre dos modelos con igual número de niveles en el EBP-Octree. 259

ÍNDICE DE ECUACIONES

Ecuación 1. Ecuación generalizada de Euler. 48

Ecuación 2. Numero de bits necesarios para codificar hasta un nivel. 145

Ecuación 3. Calculo del nivel máximo del EBP-Octree. 147

Ecuación 4. Comparación de normales. 161

Ecuación 5.- Número de clústeres para la selección de planos relevantes. 161

Ecuación 6. Función escalar de un mapa de distancias. 236

Ecuación 7. Cálculo de la matriz de posición relativa entre dos modelos. 251

1. INTRODUCCIÓN

En este capítulo se plantea una introducción del problema que se quiere resolver y lo que se ha conseguido. También se realiza una breve descripción del contenido de esta memoria, dando una reseña de los distintos capítulos que la componen.

Esta tesis parte de los trabajos previos desarrollados por Francisco Javier Melero Rus (Melero 2008; Melero, Cano, and Torres 2008) en los que presenta una estructura de datos, el BP-Octree, basado en una jerarquía de volúmenes envolventes con indexación espacial, que implementa un algoritmo de inclusión punto en sólido para la detección de colisiones.

Las ventajas que presenta esta estructura son:

- Utiliza la indexación espacial para acelerar el cálculo de los tests de inclusión punto en sólido.
- Posibilita la visualización del modelo a diferente resolución.

- El volumen de las envolventes decrece cuando se desciende por los niveles de la jerarquía, es decir se aumenta el detalle, y este nunca es menor al volumen del modelo original.

- Permite el diseño de algoritmos que realicen test de inclusión punto en sólido.

Los problemas que presenta esta estructura son:

- No permite la representación de modelos mayores de dos millones de polígonos.

- Para poder representar modelos grandes necesita una gran cantidad de memoria principal.

- No plantea ninguna propuesta sobre como diseñar algoritmos que permitan la detección de colisiones entre modelos.

1.1. ¿Qué queremos resolver?

En la actualidad hay un gran número de áreas como puede ser la medicina, el entretenimiento o la educación en las que se están utilizando aplicaciones de realidad aumentada y realidad virtual. Estas aplicaciones se pueden utilizar como herramientas de trabajo o de aprendizaje. Estos sistemas tienen la limitación de que la información llega a los usuarios finales solamente a través de dos sentidos: el oído y la vista. Por ejemplo, si utilizamos unos cascos o gafas de realidad virtual (HMD Head-Mounted Display ("Wikipedia, 'Head-Mounted Display,'" n.d.)) faltaría la percepción táctil, la cual es fundamental para que el usuario pueda apreciar, por ejemplo, la rugosidad de la superficie del objeto que está tocando.

Por todo esto, el uso de los sistemas de realidad aumentada o realidad virtual no es el más apropiado cuando se pretenden simular modelos en los que se quiera que el usuario pueda interactuar físicamente con el entorno, ya que estos carecen de una parte fundamental como es la percepción táctil. Este problema se podría solucionar en gran medida si a los sistemas anteriores se le añadiera un dispositivo háptico, el cual haría que la información llegase a los usuarios por tres sentidos: oído, vista y tacto. Con esto el usuario tendría una percepción mucho más realista del entorno

virtual, ya que por ejemplo podría chocar con los objetos, percibir la textura de una superficie o la temperatura de un objeto.

Por lo visto anteriormente se hace necesario el uso de algún método que, a partir de la representación gráfica de modelos geométricos, se pueda conocer la posición relativa de unos sobre otros. Estos métodos que detectan la posición relativa de los objetos suelen tener asociado un coste computacional bastante elevado. Las aplicaciones que detectan la colisión entre objetos deben tener una respuesta en tiempo real. Estas aplicaciones además de detectar la colisión, tienen que hacer otras tareas como puede ser encargarse de la visualización. Además, si los objetos están en movimiento, estos cálculos los tendrá que repetir entre frames.

Dentro de la informática gráfica existe la disciplina de Detección de Colisiones que se encarga de estudiar las posiciones de los modelos dando como resultado la distancia a la que se encuentran estos, o bien calcula la colisión entre los modelos representados. Un parte de esta disciplina se encarga de estudiar la modificación que debe sufrir la trayectoria de los objetos cuando se produce una colisión.

La tecnología en el desarrollo de escáneres 3D ha evolucionado mucho en los últimos años, permitiendo recoger los más mininos detalles de los objetos escaneados. El aumento de la resolución lleva asociado un incremento en el tamaño de los modelos geométricos obtenidos, debido al gran número de polígonos que forman el modelo resultante. Esto ha provocado que para ciertas aplicaciones sea imposible trabajar directamente con el modelo geométrico, debido a su coste computacional, por ejemplo, en la industria aeronáutica o naval.

El presente trabajo de tesis pretende desarrollar una estructura de datos que sirva para poder representar modelos geométricos formados por varias decenas de millones de triángulos. La estructura propuesta servirá para almacenar modelos formados por un gran número de polígonos y se podrá utilizar para interactuar con dispositivos hápticos. Esto va a permitir, trabajar con ordenadores de uso doméstico, pudiendo representar el más mínimo detalle de los objetos a los que simboliza. La estructura de datos que se ha propuesto es una estructura jerárquica de volúmenes envolventes.

La superficie de los modelos con los que se trabaja en esta tesis viene representada por un conjunto de triángulos. La estructura de datos diseñada en este trabajo esta preparada para funcionar con polígonos por ser estos más genéricos e incluyen a los anteriores.

1.2. ¿Qué hemos conseguido?

La estructura de datos presentada en esta tesis permite almacenar de forma eficiente modelos geométricos formados por varias decenas de millones de polígonos. Esta estructura jerárquica de volúmenes envolventes permite la simplificación de grandes modelos geométricos, permitiendo la detección de colisiones entre objetos muy grandes de una forma muy eficiente y rápida. La estructura se ha probado en ordenadores domésticos y se ha confirmado su rendimiento en contextos hápticos, demostrando tiempos de respuesta para la detección de colisiones muy pequeños.

La idea principal del trabajo realizado es la utilización de la estructura jerárquica de volúmenes envolventes con indexación espacial para poder realizar en tiempo real test de inclusión punto en sólido, test de cálculo de distancias entre un punto y un modelo y test de detección de colisiones entre modelos. Esta estructura se ha utilizado en la simulación de una aplicación háptica de tipo puntero con modelos que requerían una gran precisión, procedentes del Museo Histórico de Écija para el modelo de la Amazona Herida, de la empresa AGEO para el modelo de la Moldura y del Stanford 3D Scanning Repository para los modelos de Lucy y Armadillo.

Las pruebas realizadas con varios modelos geométricos formados por entre veintiséis y veintiocho millones de triángulos han sido muy satisfactorias, obteniendo unos tiempos de respuesta, tanto en los test inclusión punto en sólido y de cálculo de distancia como en los test de colisión entre modelos, asimilables a tiempo real.

Los objetivos que se plantearon en esta tesis y se han conseguido han sido:

1. Definir una estructura de datos que permite trabajar con varios modelos geométricos formados por decenas de millones de triángulos.
2. Descomponer el modelo geométrico en una secuencia de volúmenes envolventes.
3. Permitir la indexación espacial del modelo representado que optimice la detección de la colisión y la inclusión.
4. Poder trabajar en ordenadores de uso doméstico.
5. Diseñar e implementar un algoritmo que permite ejecutar test punto en sólido.

6. Diseñar e implementar un algoritmo que permite calcular la distancia mínima y el ángulo de contacto entre un punto y un modelo.
7. Desarrollar una aplicación que permite utilizar un dispositivo háptico tipo puntero con modelos escaneados a una gran resolución.
8. Diseñar e implementar un algoritmo que puede detectar la colisión de dos modelos.
9. Diseñar e implementar un algoritmo que permite seleccionar el nivel para visualizar las diferentes envolventes que forman los modelos.

1.3. Estructura de la memoria.

Esta memoria se ha estructurado en seis capítulos. A continuación, se da una breve descripción del contenido de cada uno de ellos.

En el capítulo 1 se hace una pequeña introducción del trabajo que se pretende desarrollar.

En el capítulo 2 se hace un estudio de las diferentes estructuras de datos utilizadas en la representación de sólidos que permiten encontrar colisiones, así como de las distintas estrategias utilizadas en el cálculo de la detección de colisiones. También se define el concepto de la percepción háptica en las personas y se analizan los diferentes dispositivos hápticos tipo puntero que se pueden encontrar actualmente en el mercado.

En el capítulo 3 se presenta la estructura de datos propuesta en este trabajo, la cual permite representar modelos formados por varias decenas de millones de polígonos.

En el capítulo 4 se utiliza la estructura de datos propuesta para resolver el problema de la detección de inclusión punto en sólido y del cálculo de la distancia y ángulo de contacto entre un punto y un sólido. Para ello se han diseñado una serie de algoritmos que permiten realizar estos cálculos. También se ha diseñado una aplicación que implementa estos algoritmos en un dispositivo háptico tipo puntero.

En el capítulo 5 se han diseñado una serie de algoritmos que permiten resolver el problema de la detección de colisiones entre dos modelos.

En el capítulo 6 se hace un resumen en el que se presentan las conclusiones del trabajo desarrollado, así como de las líneas de trabajos futuros que se pueden desarrollar a partir de todo lo expuesto en esta tesis.

2. DETECCIÓN DE COLISIONES

En este capítulo se hace un estudio de las distintas estructuras de datos utilizadas para la representación de sólidos. Se analizan los diferentes algoritmos empleados en la detección de colisiones entre modelos. Además, se muestran la principales librerías y aplicaciones existentes en el campo de la detección de colisiones. Por último, se hace una introducción a la percepción háptica en las personas y una clasificación de los principales dispositivos hápticos tipo puntero que actualmente existen en el mercado.

La detección de colisiones ha sido uno de los temas más investigados en las últimas décadas dentro del campo de la informática gráfica. La detección de colisones entre objetos no es un tema simple, ya que no solamente consiste en ver si los objetos intersecan sino que en todo el proceso hay que responder las siguiente preguntas:

- El "si" intersecan o no, devolviendo un valor booleano dependiendo a la presunta de si los dos objetos se cruzan o no.
- El "cuándo", indicando el momento exacto, en el movimiento de los objetos, en que estos se tocan.
- El "dónde", proporcionando el lugar exacto en el que se ha producido la colisión.

En la actualidad existe una gran demanda de algoritmos cada vez más rápidos y robustos, que resuelvan estas cuestiones. En el mercado se pueden encontrar una gran variedad de aplicaciones que utilizan estos algoritmos tales como, las que simulan objetos rígidos o deformables, videojuegos, simulaciones físicas, simulación quirúrgica, robótica, animación por ordenador, prototipos virtuales, realidad virtual, realidad aumentada, simulación de recorridos, entre otros muchos.

Gracias a la detección de colisiones se pueden emular entornos del mundo real de una manera sólida, manteniendo la ilusión de realismo, ya que, por ejemplo, no permite que dos objetos se superpongan, o que un personaje de un video juego atraviese una pared.

En la bibliografía se encuentran algunas aplicaciones que necesitan una respuesta exacta sobre la detección de colisiones, no importando mucho el tiempo que conlleve su cálculo, como por ejemplo en la planificación de rutas. Otras, sin embargo, necesitan que se dé una respuesta muy rápida a la detección de colisiones como pueden ser los videojuegos. Estas limitaciones hacen que una gran parte del tiempo de los motores de juegos o de simulación se dedique al cálculo de la colisión. Por esto se hace fundamental tener algoritmos que resuelvan este problema de manera eficiente y que los resultados devueltos sean lo más robustos posibles.

2.1. Definiciones previas.

Este trabajo se basa en la detección de colisiones entre objetos sólidos. Por ello se van a definir los siguientes conceptos con el fin de aclarar la terminología utilizada.

2.1.1. Colisión. Definición y ámbito del problema

En el campo de la informática, y más concretamente en el de la informática gráfica se define la colisión entre objetos como aquella situación en la que, dentro de una escena y en un instante dado, dos o más objetos ocupan una misma posición.

Dentro de una escena se pueden encontrar dos tipos de objetos: los que no se mueven o están estáticos y los que se encuentran en movimiento. Cuando dos objetos inmóviles colisionan se dice que se ha producido una interferencia. Los objetos móviles se desplazan por la escena siguiendo una trayectoria a una velocidad

determinada. La animación de una escena simula un flujo continuo de configuraciones entre objetos, cada una de las cuales se corresponde con un momento de la animación.

El cálculo de la colisión entre objetos tiene asociadas varias limitaciones en función del ordenador donde se realice. Estas limitaciones no son sólo el tiempo de cálculo de la colisión, que en algunos casos se desearía una respuesta en tiempo real, sino otras como la de la memoria necesaria para almacenar los objetos y las que presentan las maquinas con respecto la precisión numérica, ya que estas trabajan con una aritmética finita. Este hecho provoca que los resultados obtenidos por los algoritmos a veces no sean robustos y correctos.

El cálculo de la colisión entre objetos se puede simplificar si se utilizan objetos con cara planas, ya que esto permite desarrollar un modelo matemático sencillo para calcular la colisión.

2.1.2. Objeto sólido.

Se puede definir un cuerpo u objeto sólido como aquel que opone resistencia a cambios de forma y de volumen. Sus partículas se encuentran juntas y correctamente ordenadas. Las moléculas de un objeto sólido tienen una gran cohesión y adoptan formas bien definidas.

Dentro del campo de la informática gráfica, un sólido se tiene que plasmar utilizando un esquema de representación. Para que todo esquema de representación de un sólido sea autónomo y no se pierdan ninguna de las características del sólido que intenta representar debe cumplir una serie de propiedades formales. Según A. Requicha (Requicha 1980) estas propiedades son:

- **Rigidez:** Un sólido abstracto debe tener una configuración o forma invariante que sea independiente de la ubicación y orientación del sólido.
- **Tridimensionalidad homogénea:** Un sólido debe tener un interior y sus límites no pueden tener partes aisladas.
- **Finitud:** Un sólido debe ocupar una porción finita del espacio.
- **Clausura bajo movimientos rígidos y ciertas operaciones booleanas:** Los movimientos rígidos (translaciones y/o rotaciones) u operaciones que

añaden o eliminan material (por ejemplo una soldadura) deben producir otros sólidos cuando se aplican a sólidos.

- **Descripción finita:** En el modelado de sólidos debe haber algún aspecto finito (por ejemplo, un número finito de "caras") para asegurar que se pueden representar utilizando ordenadores.
- **Determinismo de frontera:** El límite de un sólido debe determinar inequívocamente lo que es "interior" y, por tanto, lo que comprende el sólido.

Además, según Requicha (Requicha 1980), de las definiciones anteriores se deducen las siguientes propiedades formales de la representación de esquemas:

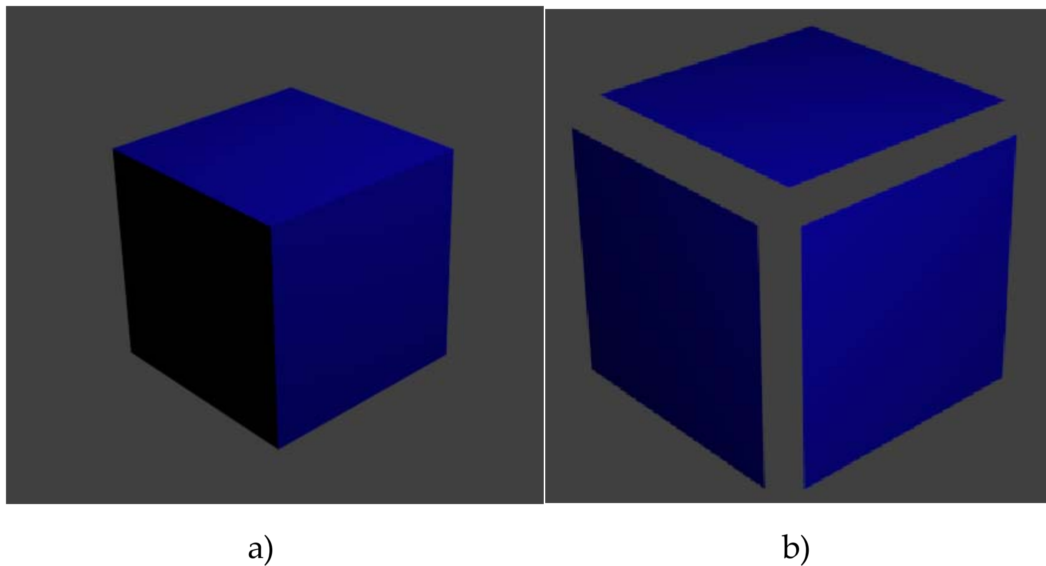
1. **Dominio.** Hace referencia al poder expresivo del esquema de representación que se está definiendo. El dominio D es el conjunto de sólidos $D \subseteq M$ que se pueden modelar con el esquema de representación. Una de las características deseables es que con él se puedan representar el mayor número de tipos de sólidos. Otro aspecto que se debe considerar es que el domino no solo recoge las propiedades de imagen de un sólido (que son las que se utilizan para modelar), sino que en este se tienen que tener en cuenta otras.
2. **Validez**. Dado un método de representación se dice que es válido si el conjunto imagen coincide con el conjunto de representaciones. Un método de representación no es válido si la representación del sólido no se corresponde con ningún sólido real. El método de representación debe garantizar que todas las representaciones de sólidos que genere recojan sólidos válidos. Si la representación no puede garantizar esto, entonces esta debe proporcionar una serie de procedimientos que permitan comprobar si la representación de un sólido es válida o no.
3. **Completitud o no ambigüedad**. Cada sólido real representado por un método debe ser no ambiguo, ya que no se pueden tener dos sólidos reales diferentes representados con la misma estructura. Si el método de representación no cumple esta propiedad entonces se imposibilita la posibilidad de poder comparar los sólidos representados.
4. **Unicidad**. Cada sólido real que se representa con el método debe ser único, es decir no se pueden tener dos sólidos diferentes que tengan la misma

representación. Si el método de representación no cumple esta propiedad entonces se imposibilita el poder comparar dos sólidos para ver si estos son iguales.

2.1.3. Representación de sólidos más utilizadas.

De todas las técnicas encontradas en la bibliografía especializada para la representación de sólidos las más utilizadas son:

- **Representación de superficies frontera**. En esta representación el objeto se simboliza por las superficies que lo delimitan. Las superficies que delimitan a un objeto son la frontera del mismo ya que esta es la se separa los puntos que están dentro de los que están fuera del sólido. La frontera del objeto está formada por un conjunto finito y disjunto de superficies que pueden ser curvas o planas. Cada una de estas superficies está acotada por un perímetro anular de aristas, las cuales pasan por un conjunto de vértices. Cuando la superficie tiene agujeros estos se representan por un anillo interno de aristas. Ver *Figura 1*.



a) b)

Figura 1: a) Objeto sólido. b) Caras en las que se descompone el sólido.

Cada una de las superficies que delimitan al sólido da información topológica y geométrica. La información topológica contiene información de cómo se conectan cada uno de los elementos que la forman. La

información geométrica incluye la localización y la dimensión de cada componente de la superficie.

La representación de todo sólido mediante su frontera debe definir un sólido válido. Para ello cualquier superficie que se utilice para definir la frontera de un sólido debe dividirlo en dos partes una interior y otra exterior, debe de ser cerrada no dejando ni aristas ni caras sueltas y por último no puede haber caras que intersecten entre sí.

La verificación de que una superficie cumple las propiedades de ser cerrada y que no haya caras que intersectan entre sí se puede comprobar mediante la ecuación generalizada de Euler (Ver *Ecuación 1*)

$$2S + R - 2H = C + V - A$$

Ecuación 1. Ecuación generalizada de Euler.

donde S es el número de parte separadas que forman el objeto, R es el número de anillos dentro de las caras, H es el número de agujeros que atraviesan el objeto, C el número de caras que representan el objeto, V el número de vértices y A el número de aristas.

La característica de esta representación es que describe a los objetos de forma poco concisa, ambigua y no única. La realización de operaciones booleanas entre sólidos representados por este método es muy costosa por necesitar un gran tiempo de cálculo, ya que la única forma que hay para saber si dos objetos interseccionan es comprobar si todas las caras de un sólido interseccionan con las del otro objeto.

La visualización de los objetos representados mediante esta técnica es directa, ya que los algoritmos para la visualización trabajan sobre la geometría del objeto, la cual, como se ha visto, está incluida en su representación. La mayoría de tarjetas gráficas que se pueden encontrar actualmente en el mercado ofrecen aceleración hardware para el dibujado de polígonos.

Esta representación es la más utilizada para la representación de sólidos (Stroud 2006; Mäntylä 1988) encontrándose en numerosas referencias bibliográficas. También se pueden encontrar en la bibliografía un gran número de referencias a artículos que detallan algoritmos que sirven para

detectar la colisión entre objetos.

- **Representación de subdivisión espacial**. Esta técnica consiste en subdividir o descomponer el espacio que ocupa el sólido en una colección de entidades geométricas o celdas mucho más simples, como pueden ser paralepípedos, cubos, etc. La yuxtaposición de todas estas entidades geométricas o celdas llena todo el espacio ocupado por el objeto. Las celdas en las que se subdivide el espacio que ocupa el objeto son no intersecantes y adjuntan, siendo estas etiquetadas para el objeto al que representan.

  Existen varios modelos dentro de esta técnica, como pueden ser:

  - *Descomposición en celdas:* Esta técnica consiste en definir un conjunto de celdas básicas las cuales se utilizarán para formar el objeto. La representación del objeto que se obtiene es como si se modelara el mismo utilizando las fichas del juego Lego. La única operación lógica que se permite es la unión. El objeto queda representado por un array de celdas, indicando para cada una de ellas su posición y tamaño. Ver por ejemplo en la *Figura 2*.

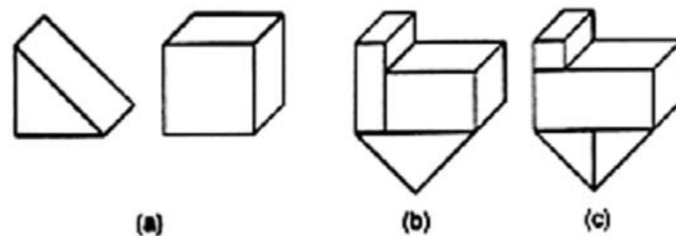


Figura 2. Composición de objeto con celdas básicas.

  - *Enumeración de la ocupación espacial*: En esta técnica se divide el espacio que ocupa el objeto se divide en celdas idénticas y alineadas según una malla regular. Las celdas que se utilizan para representar al objeto se denominan voxels. La forma que más se suele utilizar para representar al voxel es el cubo.

    En esta técnica todas las celdas tienen el mismo tamaño siendo este invariable y único y depende de la resolución de la malla seleccionada. Cada celda tiene asignada una posición de la malla. Para cada celda o voxel se indica si pertenece al objeto (cuando este está por completo dentro del objeto) o está ausente (el voxel está

fuera por completo o tiene parte dentro y fuera del objeto). Por esto los objetos que se representan con esta técnica presentan en su frontera el efecto de escalera o aliasing. Ver *Figura 3*.

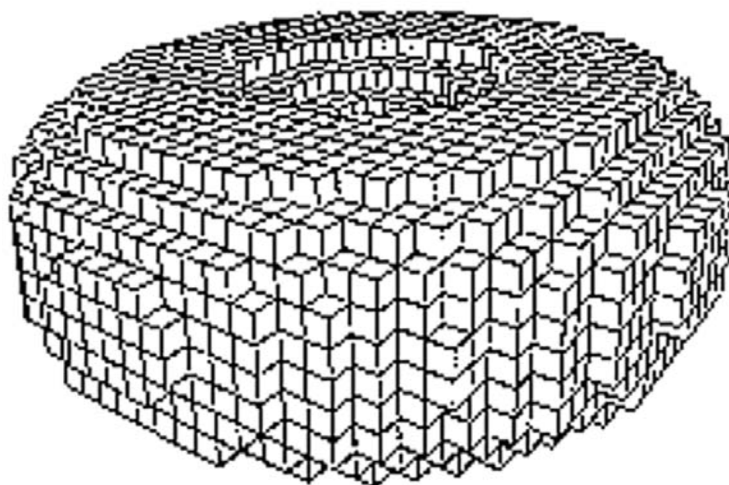


Figura 3. Representación de un toroide mediante voxels. © A.H.J. Christensen, 1980

Para evitar el efecto de escalera se puede bajar la resolución de la malla de voxels aumentando así la precisión de la representación. El aumento de la resolución lleva asociado un gran aumento de la memoria que se necesita para representar el objeto.

Para representar un objeto utilizando esta técnica solo tenemos que definir el tamaño de los voxels y ver para cada uno de ellos si está por completo dentro del objeto o no. Con esta técnica el objeto queda representado mediante un array de voxels ocupados.

La gran ventaja de esta representación es que es muy fácil calcular si un punto pertenece al objeto o si dos objetos son adyacentes. También otra ventaja de esta representación es que se guarda información del interior del objeto, a diferencia de la representación con B-Rep que sólo guarda información de la frontera. Los voxels que caen dentro del objeto no son visibles, pero están, con lo cual se les puede asignar propiedades diferentes. Por ejemplo, si el objeto está compuesto por diferentes materiales se le podría asignar un color diferente a cada uno de ellos, pudiendo visualizar los voxels que pertenezcan a un mismo material.

El mayor inconveniente de esta técnica es que no permite representar voxels con ocupación parcial, provocando esto que los únicos objetos que se puedan representar sean los que sus caras sean paralelas a las caras del voxels y sus vértices coincidan con la malla.

2.2. Introducción.

La detección de colisiones entre objetos es un problema que inicialmente parece muy sencillo; simplemente consiste en saber si uno o varios objetos intersectan entre sí. Más concretamente, para saber si se produce una colisión se tiene que determinar: si se produce intersección, cuándo (si los objetos están en movimiento) y dónde se ha producido dicha colisión.

Gracias a la detección de colisiones se pueden garantizar la simulación y las propiedades sólidas de los objetos reales en un mundo virtual.

Tal y como se comentó en el capítulo anterior, la detección de colisiones es fundamental para muchas aplicaciones como pueden ser la robótica, simulaciones físicas, la creación de prototipos virtuales, las simulaciones de ingeniería, los juegos de ordenador y la generación de escenas realistas.

En la animación por ordenador se utiliza la detección de colisiones para dotar de realismo el movimiento de los objetos por una escena. Por ejemplo, para emular el comportamiento de la tela cuando un personaje se mueve por un escenario, emulando los pliegues y las arrugas que se formarían en la tela cuando esta choca con las extremidades de una persona. En robótica se puede utilizar para planificar el movimiento de los robots por una escena esquivando los obstáculos que se puedan encontrar en su camino, pudiendo trazar rutas de movimiento para las diferentes partes que forman el robot. En ingeniería se puede utilizar para simular el comportamiento de dos objetos cuando chocan. Por ejemplo, se pueden emular los test de seguridad que pasan los coches ahorrándose todos los costes, ya que no se tienen que utilizar y destruir coches reales.

Las aplicaciones que utilizan la detección de colisiones se pueden clasificar en dos: las que necesitan una respuesta en tiempo real y las que no. Los juegos por ordenador son aplicaciones que necesitan tener una respuesta muy rápida a la detección de colisiones cerca del tiempo real, ya que se necesita tener la sensación de continuidad en los movimientos, debiendo de refrescarse la imagen en pantalla más de treinta

frames por segundo. Otras aplicaciones que necesitan una respuesta tan rápida o incluso mayor que la de los juegos de ordenador son simuladores quirúrgicos, simuladores de realidad virtual, sistemas hápticos, …

Para la generación de escenas realistas se utiliza la técnica de Ray-Tracing permite sintetizar imágenes tridimensionales. Para ello se lanzan una serie de rayos desde la posición del observador hasta la escena a través del plano de la imagen y se calcula la colisión de cada uno de ellos con los objetos de la escena. Una herramienta utilizada para acelerar el Ray-Tracing es el uso de volúmenes envolventes, ya que sólo es posible que un rayo interseccione con un objeto si antes lo ha hecho con su volumen envolvente. Esta técnica permite clasificar los objetos de manera eficiente quedándose solo con aquellos se son susceptibles de presentar intersección con el rayo.

El diseño de un sistema eficiente de detección de colisiones no es una tarea sencilla ya que hay que tener en cuenta una serie de factores que afectan directamente al diseño del sistema, tales como:

- **Representación del dominio de la aplicación**. La representación geométrica de los objetos o la escena condiciona en gran medida el tipo de algoritmo que se podrán utilizar para determinar la colisión entre objetos dentro de una escena. La elección adecuada de la representación geométrica es fundamental dependiendo del tipo de aplicación donde se quieran resolver las cuestiones de detección de colisiones.

- **Diferentes tipos de consultas**. Dependiendo del tipo de consulta que la aplicación necesite, se debe elegir una representación geométrica de los objetos u otra, ya que no todas las representaciones permiten todos los tipos de consultas. Cuanto más detallados sean los tipos de consulta y los resultados esperados, más esfuerzo computacional se requerirá para obtenerlos.

- **Parámetros de simulación del entorno**. Los parámetros que se deben tener en cuenta a la hora de diseñar un sistema de detección de colisiones son: cuántos objetos hay en la escena, sus tamaños y posiciones relativos, si estos se mueven y cómo se mueven, si se les permite interpenetrar y si son rígidos o flexibles.

- **Rendimiento**. Es un factor fundamental de los sistemas de detección de colisiones, ya que muchos de ellos necesitan trabajar en tiempo real. Este

motivo obliga a estos sistemas a operar bajo estrictas restricciones de tiempo y tamaño de los objetos.

- **Robustez**. No todas las aplicaciones requieren el mismo nivel de simulación física. En algunas basta con una aproximación en la detección de colisiones mientras que en otras, esta debe ser lo más precisa posible.
- **Facilidad de implementación y uso**. Es otro factor a tener en cuenta, ya que a veces se dispone de poco tiempo para realizar un proyecto, haciendo esto que se tengan que escoger representaciones que permitan la implementación de los algoritmos de manera rápida.

En los siguientes apartados, profundizaremos los aspectos relativos a los factores anteriormente indicados y cómo se han abordado en este trabajo u otros existentes en la literatura.

2.2.1. Detección de colisiones con objetos no convexos.

En este apartado se presenta uno de los temas fundamentales de esta tesis: la detección de colisiones entre objetos.

En este apartado también se realiza un estudio de los métodos encontrados en la literatura que más se han referenciado y han sido eficientes a la hora de detectar la colisión. Además, se especifican las diferentes aplicaciones donde se ha utilizado la detección de colisiones. Y por último se detalla una lista de librerías que se encuentran en internet libres para poder utilizarse en la detección de colisiones.

2.2.1.1. Ámbito del problema

La detección de colisiones se debe englobar dentro del término "manipulación de la colisión". La manipulación de la colisión se puede dividir según (Akenine-Möller, Haines, and Hoffman 2008) en las siguientes tres fases o etapas:

- Detección de la colisión: comprueba si dos objetos colisionan entre si, devolviendo un valor lógico de verdadero si colisionan y de falso en caso contrario.
- Determinación de la colisión: se calcula la zona de intersección entre un par de objetos que han colisionado.

- Respuesta a la colisión: se indica cómo tienen que reaccionar los objetos ante la colisión.

De las tres fases anteriores la última de ellas, donde se da la respuesta a la colisión, es quizás la más complicada ya que esta puede ser abordada de diferentes maneras. Por ejemplo, una respuesta a la colisión puede ser la deformación o la ruptura de los objetos (Van Den Bergen 2004; Teschner et al. 2004). Otro tipo de respuesta a la colisión puede ser que los objetos cuando choquen reboten y cambien su trayectoria (Eberly 2012). En estos casos para poder conocer la nueva trayectoria de los objetos será necesario conocer las trayectorias que siguen, las fuerzas (por ejemplo la gravedad) y las masas o pesos de los objetos que colisionan en la escena. Esta última fase no es objetivo de esta tesis.

2.2.1.1.1. Aplicaciones en las que se utiliza la detección de colisiones.

Existen muchas aplicaciones donde se suele utilizar la detección de colisiones dentro de un escenario en el cual se colocan varios objetos, los cuales pueden estar estáticos o móviles. Las aplicaciones se pueden clasificar o agrupar en tres ámbitos diferentes, destacando las que usan la informática gráfica, los entornos de simulaciones físicas y en la robótica. Más concretamente se pueden detallar estos tres grandes grupos obtenidos tras la clasificación anterior como:

Aplicaciones en informática gráfica.

Dentro de este grupo se pueden encontrar una gran variedad de aplicaciones, las cuales se pueden clasificar dentro de tres grandes subgrupos:

- la realidad virtual, realidad aumentada y entornos inmersivos,
- las aplicaciones dedicadas al Computer-Aided Desing and Manufactured (CAD/CAM) y
- la animación por ordenador.

Vamos a detallar cada uno de estos apartados.

- Realidad virtual.

  Con ella se pueden simular escenarios de manera que se pueda interactuar, visualizar y explorar los objetos que se incluyen en el escenario (J. D. Cohen et al. 1995; Wilson et al. 1999). Estas aplicaciones definen un escenario estático sobre el cual se introduce un avatar que se mueve por la escena, siendo capaz de detectar la colisión con los objetos que la forman, por

ejemplo, deteniendo su movimiento cuando se produzca una colisión con alguno de ellos.

Dentro de la realidad virtual también se pueden incluir:

- Los juegos de ordenador que emulan entornos 3D (Blow 2011), los cuales tienen que dar respuesta y comprobar una serie de colisiones entre múltiples objetos, y todo ello en tiempo real.
- Los simuladores, por ejemplo, los de vuelo o de coches. Los simuladores tienen que calcular la colisión y dar respuesta a la misma, todo ello con unas grandes restricciones de tiempo, ya que de ellos se espera una respuesta en tiempo real
- Aplicaciones CAD/CAM.

  En la producción y diseño de piezas CAD, se pueden analizar, evaluar y validar objetos diseñados, evitando la construcción de prototipos, lo que supone una reducción en los costes de producción (Zachmann 1997). El prototipado virtual permite emular procesos de mantenimiento dentro de la industria y comprobar el ensamblaje de piezas (Gomes de Sá and Zachmann 1999). La carencia fundamental que presentan los modelos de realidad virtual es la falta de percepción a la hora de interactuar con los objetos. En la bibliográfica se pueden encontrar artículos en los que al entorno virtual se le ha incorporado un dispositivo de reflexión de fuerzas (A. Cohen and Chen 1999; McNeely, Puterbaugh, and Troy 1999) que añade un grado más de realismo.
- Animación por ordenador.

  Aquí se encuentran una serie de aplicaciones que no tienen que dar una respuesta exacta a una colisión, sino lo que pretenden es emular el comportamiento de una serie de objetos de la forma más realista posible. Dentro de este grupo se encuentran una serie de aplicaciones que emulan el comportamiento de telas cuando chocan con objetos (Volino and Magnenat-Thalmann 1997; Zhang and Yuen 2002). Por ejemplo, se pueden emular como se comportaría una prenda de ropa cuando se la pone una persona y esta se mueve por un escenario. Otro uso muy común de la animación es la

realización de películas en las que se intenta emular y captar o grabar de la forma más realista posible escenarios donde se mueven una serie de personajes que interactúan con otros personajes u objetos que se encuentren dentro del escenario.

Emulación de entornos físicos.

Dentro de este grupo se encuentran aplicaciones que emulan el movimiento de objetos siguiendo una serie de leyes de la dinámica. Esto hace que los objetos se muevan siguiendo una serie de restricciones físicas. Una de las utilidades de estas aplicaciones es que posibilitan la simulación de choques entre objetos, por ejemplo, la de los coches, pudiéndose analizar el comportamiento de estos comprobando cómo rebotarían, se romperían o se deformarían. Otro uso es la creación de aplicaciones que permiten emular la construcción y el funcionamiento de mecanismos (Garcia-Alonso, Serrano, and Flaquer 1994), así como para poder interpretar el comportamiento de un sistema (Barriuso 1998). Para poder emular de una manera correcta un entorno en el cual los objetos se mueven siguiendo unas leyes físicas, hace falta recoger una serie de información extra de los objetos como puede ser la velocidad que llevan, la masa, etc.

Colisión de objetos en robótica.

En la robótica el problema de determinar cuándo dos objetos colisionan ha sido ampliamente estudiado. Los algoritmos utilizados por los robots para planificar su movimiento tienen que calcular las posibles colisiones entre el robot y los objetos que se encuentran dentro de una escena. La planificación de caminos consiste en concebir el movimiento de un robot por una escena de manera que no se produzca la colisión de este con ninguno de los objetos incluidos en la escena. Esta planificación se puede realizar off-line para sistemas sin tiempo real (Hayward 1986; Latombe 1991; Cameron and Quin 1998). Los robots actuales son autónomos, por tanto, no se conoce con anterioridad el movimiento de los mismos por la escena ni el entorno por el que se mueven. Por estos motivos la detección de colisiones tiene que ser on-line (Shaffer and Herb 1992).

2.2.1.2. Diferentes tipos de detección de colisiones

La detección de colisiones no consiste solamente en determinar si dos objetos se superponen en un momento dado, sino que a veces lo que interesa es conocer otro tipo de información sobre la colisión, como podría ser la zona que estan

intersectando, la distancia a la que se encuentran dos objetos o si estos en el futuro chocarán. Dependiendo del uso que se le dé al cálculo de la zona donde colisionan dos objetos, se necesitará mayor o menor precisión, obteniendose la zona exacta de intersección o bien solamente se calcula de manera aproximada, pudiendose tolerar un cierto error. De la detección de la colisión se puede obtener una serie de información como puede ser: la distancia de separación para cada pareja de objetos que se incluyan en la escena, la normal y el punto exacto donde colisionan los objetos, la distancia de penetración de un objeto sobre otro y el tiempo que debe transcurir para que dos objetos colisionen.

La elección del algoritmo que calcula la colisión dependerá de la información que se necesite. Por esto los algoritmos se pueden clasificar en cuatro grupos, dependiendo de la cuestión que resuelvan (M. Lin and Gottschalk 1998). Estos cuatro grupos son:

- *Los que dan respuesta a la colisión*. Aqui nos encontramos una serie de algoritmos que devuelven un valor booleano indicando si dos objetos colisionan o no. Por ejemplo se puede utilizar en el diseño de piezas mecanicas móviles, de las que se quiere saber si el movimiento de las mismas provoca la colisión con otras piezas (Zachmann 1997).

- *Los que devuelven la distancia entre dos objetos*. Los algoritmos que permiten conocer la distancia exacta a la que se encuentran dos objetos se puede utilizar, por ejemplo, para el diseño de sistemas mecánicos. De esta forma, el ingeniero puede calcular en cada instante la distancia entre las diferentes piezas (Barriuso 1998). Otra aplicación donde se suelen utilizar algoritmos de este grupo es en la planificación de recorridos de un robot, ya que con estos algoritmos se pueden planificar los movimientos de un robot para que este no colisione con ningún objeto dentro de la escena donde se desplaza.

- *Los que calculan la zona exacta de intersección de dos objeto.* Estos algoritmos suelen ser utilizados por aplicaciones que necesiten dar una respuesta correcta a una colisión (Moore and Wilhelms 1988; Hahn 1988). Por este motivo son de gran utilidad dentro de sistemas de animación y de simulación física.

- *Los que predicen las posibles colisiones*. Estos algoritmos sirven para detectar si el siguiente desplazamiento de los objetos por la escena provocaría una colisión. La predicción de posibles colisiones se puede utilizar para mover

un robot por una escena (M. C. Lin 1993), de manera que si el siguiente movimiento del robot provoca que este colisione con algun objeto, el robot decidirá cambiar el movimiento para evitar esa colisión.

2.2.1.3. Problemas en la detección de colisiones

En la detección de colisiones se plantean una serie de problemas a la hora de decidir qué algoritmo utilizar para comprobar si dos objetos colisionan. Estos son:

2.2.1.3.1. Uso de información de situaciones previas.

Este problema surge cuando no se reaprovecha la información obtenida de la detección de colisiones en situaciones o instantes anteriores, ya que lo normal es que muchos de los cálculos que se tienen que hacer sean los mismos que en el estado o situación previa. De esta manera se emplea mucho tiempo en obtener información previamente calculada. Este problema se soluciona almacenado la información y los cálculos que se realizan en cada instante y acceder a ellos siempre que se necesite. En la bibliografía se pueden encontrar algoritmos (Baraff 1992; M. C. Lin and Canny 1991) que reutilizan información entre frames.

2.2.1.3.2. Desplazamiento de los objetos de manera secuencial.

Los objetos se mueven de manera simultánea en el mundo real. Para poder emular el desplazamiento de los objetos dentro de un entorno virtual, estos se tienen que desplazar secuencialmente dentro de la escena. El movimiento secuencial de los objetos por la escena puede producir situaciones no deseables, por ejemplo, que no se descubra la colisión entre dos objetos que si se produciría en el mundo real. O el caso contrario, que se detecte una colisión entre dos objetos que en verdad no se produciría si los objetos se movieran simultáneamente. Estos problemas se suelen resolver fijando un tiempo de comprobación de colisiones lo suficientemente pequeño con el fin de evitar que un objeto pueda pasar entre otro sin que se compruebe si colisionan (Ericson 2005; Schulman et al. 2013).

2.2.1.3.3. Movimiento a intervalos de los objetos por la escena.

Los objetos se mueven por la escena siguiendo una trayectoria continua. Esta trayectoria continua no se puede emular con un ordenador, ya que estos trabajan de manera discreta. Para solucionar este problema lo que se hace es discretizar el movimiento de los objetos en intervalos de tiempo finitos. En estos casos lo más difícil es establecer un incremento de tiempo en el que sea viable detectar todas las posibles colisiones, ya que en la escena puede haber objetos con tamaño y velocidad

muy dispares. En la bibliografía se pueden encontrar una serie de artículos que solucionan este inconveniente utilizando un intervalo de tiempo adaptativo. Para calcular dicho intervalo se debe conocer previamente el recorrido de cada uno de los objetos, obteniéndose la ecuación del movimiento en función del tiempo (Canny 1986; Culley and Kempf 1986; Snyder 1992). Resolviendo estas ecuaciones se puede predecir cuándo se producirá la siguiente colisión. Otra solución consiste en calcular el volumen de barrido de los objetos entre dos instantes de tiempo consecutivos. Para la detección de colisiones se comprueba si los volúmenes de barrido colisionan. Si estos no colisionan se puede afirmar que los objetos tampoco van a hacerlo, pero si colisionan entonces puede que los objetos lo hagan o no. Por último, otra posible solución consiste en dividir el espacio en regiones y controlar cuándo un objeto cambia de región. De esta forma se obtiene el intervalo de tiempo adaptativo (Samet and Tamminen 1985; Duff 1992).

2.2.1.3.4. Cálculo exacto de la zona de colisión entre todos los objetos.

La eficiencia de estos cálculos depende de cómo se representen los objetos sobre los que hay que calcular la colisión. Los algoritmos utilizados normalmente no tienen que calcular sólo si se produce interpenetración entre los objetos, si no que tienen que obtener más información, como puede ser la zona exacta de penetración, ya que esta información suele ser empleada por otros módulos de la aplicación para calcular la respuesta a la colisión. Una técnica utilizada en varios trabajos (Herman 1986; Duff 1992; Moore and Wilhelms 1988; Lubachevsky 1991; Hubbard 1993b; Ortega and Feito 2005) **consiste** en hacer un filtrado previo de los objetos que podrían colisionar para no tener que comprobar la colisión exacta entre todos los incluidos en la escena. Los objetos que pasen este primer filtrado son los que se van a comprobar. Para ello los objetos se descomponen siguiendo técnicas de jerarquías de volúmenes envolventes, optimizando así el cálculo de la zona exacta de la colisión.

2.2.1.3.5. Test de colisión entre todos los pares de objetos.

Ya que los objetos que se pueden colocar en la escena podrían ser miles, comprobar la colisión entre todos los pares de objetos podría ser muy costoso, del orden de $O(n^2)$. Una forma de minimizar el tiempo para comprobar la colisión entre todos los objetos de una escena, es reducir el número de comprobaciones entre pares de objetos. Para ello se puede realizar un filtrado previo, comprobando solamente los pares de objetos que están próximos o que ocupan una misma zona en el espacio. De esta manera sólo se comprueba la colisión exacta entre unos cuantos pares de objetos. Los siguientes artículos (Herman 1986; Duff 1992; Moore and Wilhelms 1988;

Lubachevsky 1991; Hubbard 1993b; Ortega and Feito 2005) utilizan este filtrado previo. Para ello dividen el espacio en diferentes zonas o regiones, calculando la colisión exacta entre los pares de objetos que ocupen la misma zona o región espacial.

2.2.1.3.6. Cantidad de objetos simultáneos en una escena.

Como el número de objetos que se pueden incluir en una escena no tiene límite se pueden encontrar en la bibliografía aplicaciones que son capaces de manejar sin ningún problema decenas de objetos (Garcia-Alonso et al. 1994) y otras aplicaciones que incluso permiten trabajar con centenares o miles de objetos (Cohen et al. 1995, Wilson et al. 1999).

El número de test entre pares de objetos para detectar la colisión entre n objetos se puede calcular como el combinatorio de n sobre 2 $\binom{n}{2}$, con lo cual el número de comprobaciones que se tiene que realizar es $\binom{n}{2} = \frac{n(n-1)}{2}$. El orden de eficiencia de este algoritmo es cuadrático O (n^2). Para intentar optimizar este tiempo de ejecución, las aplicaciones tienen que intentar disminuir el número de test necesarios para detectar la colisión. Para ello hay aplicaciones que hacen un filtrado previo de los objetos (Hubbard 1993a), y otras que utilizan técnicas basadas en volúmenes contenedores o partición espacial (J. D. Cohen et al. 1995), descartando los pares de objetos sobre los que se tienen que realizar los test. Estos algoritmo pueden alcanzar ordenes logarítmicos O ($n \log n$) e incluso lineales O (n) (J. D. Cohen et al. 1995), aunque en el peor de los casos el orden que se obtiene es cuadrático O (n^2) (Lawlor and Kalée 2002).

2.2.1.3.7. Robustez de los algoritmos de detección de colisiones.

El problema de la robustez surge debido a la limitación que tienen los ordenadores a la hora de representar internamente la información. En un mundo real los datos y las operaciones que se realizan sobre estos son exactas. Los ordenadores utilizan una aproximación de la representación en coma flotante de los números reales, lo cual hace que la aritmética no sea exacta, produciendo un redondeo cuando se tienen que representar números con una gran precisión. Este error puede conducir a que los resultados obtenidos por los algoritmos de detección de colisiones no sean correctos y no coincidan con los que se obtendrían si se utilizara los valores exactos.

La robustez que afecta a los algoritmos de detección se colisiones viene dada por dos situaciones totalmente diferentes: la producida por los errores de los cálculos al utilizar datos con representación finita (conocida como robustez numérica) y la producida por tener una representación de los objetos sin una correcta topología

(conocida como robustez geométrica).

Un algoritmo robusto es aquel que trata las situaciones degeneradas tal y como cabría esperar, proporcionando resultados consistentes.

2.2.1.3.8. Robustez geométrica

Al calcular la detección de colisiones, los objetos deben tener una geometría libre de errores, es decir, no tener: caras con área cero, caras no convexas, caras con autointersecciones, caras, aristas o vértices repetidos, caras no planas, caras con agujeros o uniones en T, etc.

En (Ericson 2005) se presentan algunas técnicas para obtener objetos con una geometría correcta y proporciona una serie de pasos para solucionar los errores y corregir todas las anomalías que se pueden presentar en la geometría los objetos:

1. Unir vértices que estén muy próximos,
2. Unir agujeros entre caras adyacentes,
3. Unir caras coplanarias en una sola cara,
4. Descomponer en piezas convexas.

2.2.1.3.9. Robustez numérica.

La robustez numérica se puede clasificar en dos tipos:

- Robustez numérica debida a una precisión inadecuada de los datos. El problema se presenta porque los ordenadores, al representar números reales, no son capaces de utilizar aritmética real, teniendo que utilizar aritmética en coma flotante. Esta imprecisión hace que se produzcan situaciones de error como: errores por redondeo, errores de representación y de conversión, errores por cancelación de dígitos y errores debidos a underflow y overflow.

- Robustez numérica por degradación en los cálculos. Se da cuando los algoritmos no están preparados para afrontar casos con condiciones especiales, que necesitan un tratamiento especial distinto del caso general. Un ejemplo de este problema se puede dar cuando se quiere calcular la posición de un punto con respecto a un plano. Si el punto está muy próximo al plano, cualquier pequeña variación puede llevar a obtener un resultado inadecuado.

En la bibliografía se pueden encontrar diferentes soluciones que intentan resolver o solventar en parte el problema de la robustez numérica. Una solución sería utilizar tolerancia para calcular la colisión, de manera que en vez de hacer comparaciones exactas (Si x == 0.0) se hagan comparaciones con un umbral (Si abs(x) <= épsilon). Otra solución podría ser utilizar librerías específicas que permitan la representación con mayor precisión. El problema de esta solución es el consiguiente aumento del espacio ocupado por los datos en memoria, así como el aumento del tiempo para poder realizar los cálculos, haciendo que en sistemas que necesiten una respuesta en tiempo real no sea adecuado su uso (Fortune and Van Wyk 1993). Por último en (Traub 1967) se propone utilizar aritmética de intervalos para realizar los cálculos con números reales, y en (Knuth 1997) se propone utilizar aritmética entera de mayor precisión.

2.2.1.3.10. Precisión en el cálculo de detección de colisiones.

En la bibliografía se pueden encontrar una serie de aplicaciones que requieren un cálculo más preciso a la hora de detectar la colisión entre objetos, frente a otras en las que lo fundamental es dar una respuesta muy rápida en tiempo real. Por ejemplo, un algoritmo que controle el movimiento de un robot (Shaffer and Herb 1992) por una escena debe dar una respuesta en tiempo real impidiendo de esta manera que el robot colisione con algún otro objeto y provoque un accidente.

Para solucionar este problema y poder dar una respuesta a la colisión en un tiempo razonable en (Hubbard 1996) proponen dar respuesta a la colisión sin llegar al nivel máximo de precisión, de manera que el módulo de colisión marca un tiempo máximo de respuesta. La aplicación permite calcular la colisión dando respuestas parciales antes de llegar a la máxima precisión. Si en un momento dado en el cálculo de la colisión se sobrepasa el tiempo máximo fijado, el sistema responde con la solución aproximada que tenga calculada hasta ese instante. Los algoritmos que utilizan aproximaciones para dar una respuesta en un tiempo máximo se pueden clasificar en dos tipos: los de tiempo crítico (Hubbard 1993a; Hubbard 1995b; Hubbard 1995a; Hubbard 1996) y los que tienen interrupción (O'Sullivan et al. 2001).

La ventaja fundamental que presentan estos algoritmos es que la respuesta a la colisión se puede adaptar a las condiciones de la aplicación donde se vaya a utilizar. De esta manera su uso es apropiado en aplicaciones que necesiten una respuesta en tiempo real o en aplicaciones que necesiten un frame rate constante (Zachmann 1997). El inconveniente que presentan estos algoritmos es la imprecisión en los casos en los que no se calcula la colisión a máxima resolución. Para intentar mitigar este

problema en (Dingliana and O'Sullivan 2000) se presenta un algoritmo que permite dar prioridad a las colisiones más importantes.

Otros ejemplos de aplicación donde la respuesta a la colisión tiene que ser exacta, aunque se tarde más tiempo en calcularla, son los que se utilizan en ingeniería mecánica para hacer simulación de piezas (Mirtich 2000). Otros ejemplos de aplicaciones donde tienen igual importancia el cálculo de la colisión y el tiempo de respuesta, son las de realidad virtual, ya que estas deben tener un frame rate constante para que se obtenga la sensación de inmersión. Si a la realidad virtual se la añade un módulo de respuesta, como podría ser un dispositivo háptico, entonces la precisión tiene que ser doble. Por un lado habrá que obtenerse el cálculo exacto de la colisión y por otro que la respuesta sea los más rápida posible (Smith et al. 1995; Held, Klosowski, and Mitchell 1995; J. D. Cohen et al. 1995; Gottschalk et al. 1996). En este último caso la exactitud en la detección de la colisión es fundamental, ya que de ella se alimenta el módulo de respuesta a la colisión para calcular la fuerza necesaria con la que responder.

2.2.1.4. Densidad de los objetos incluidos en una escena.

La densidad de objetos se define como el número de objetos por unidad de volumen. Los escenarios se pueden clasificar como compactos y dispersos (Garcia-Alonso 1990). Un escenario es compacto cuando la densidad de los objetos es muy alta y un escenario es disperso cuando la densidad de los objetos es muy baja.

La densidad de los objetos que se incluyen en una escena se puede parametrizar calculándose como el porcentaje que está siendo ocupado por el volumen de todos los objetos con respecto al volumen del espacio total de la escena.

Por ejemplo, se pueden tener escenas con densidad baja que estén formadas por miles de objetos cuyo volumen sea muy pequeño en comparación con el volumen total de la escena (J. D. Cohen et al. 1995; Vemuri, Cao, and Chen 1998). También se pueden encontrar trabajos (Wilson et al. 1999) en los que, aunque se trabaje con objetos formados por una decena de millones polígonos, estos siguen formando un escenario disperso, ya que la escena ocupa un espacio de varios centenares de metros cuadrados.

Dentro de los escenarios compactos se pueden encontrar ejemplos (Gottschalk et al. 1996; Klosowski et al. 1998; Zachmann 1998; McNeely, Puterbaugh, and Troy 1999; Savall et al. 2002) que, aunque trabajen sólo con unos miles de polígonos, estos están

confinados dentro de un escenario cuyo volumen es muy pequeño.

A la hora de elegir el algoritmo que se va a utilizar para detectar la colisión dentro de un escenario, es fundamental saber si el escenario es compacto o disperso, ya que si los objetos se representan con un volumen envolvente, en un escenario disperso lo normal es que estos volúmenes no colisionen. Pero en un escenario compacto ocurre todo lo contrario y es que todos los volúmenes envolventes colisionan entre si, no siendo adecuado utilizar la técnica de volúmenes envolventes para representar los objetos.

2.2.1.5. Optimización en la detección de colisiones

Para aumentar la velocidad de respuesta a la hora de detectar una colisión entre objetos se pueden utilizar diferentes técnicas, como pueden ser aumentar la cantidad de memoria principal o reducir la complejidad de los objetos, con el fin de hacer más sencillo el proceso.

Otra forma de optimizar la detección de colisiones, y de esta forma disminuir los tiempos de respuesta, es utilizar la coherencia. Esta se puede clasificar en dos tipos: por un lado la coherencia temporal, en la que se aprovecha que los cambios en los objetos en dos instantes consecutivos son mínimos, y la coherencia geométrica o espacial, que permite conocer la zona de ocupación de los objetos en la escena siendo dicha zona mínima en comparación con el resto de la escena. La mayoría de las aplicaciones utilizan la coherencia geométrica o espacial de los objetos, lo que proporciona una disminución del número de comprobaciones entre pares de objetos para ver si colisionan. Las aplicaciones también, deben tener en cuenta la coherencia temporal, haciendo uso de cálculos realizados en los instantes anteriores, ahorrando una gran cantidad de operaciones.

2.2.1.5.1. Coherencia temporal entre frames consecutivos.

Si se comparan los cálculos realizados para obtener la colisión de objetos entre dos frames consecutivos, se puede comprobar que la variación es mínima. Por este motivo se almacenan los cálculos realizados para detectar la colisión en un frame y se utilizan para calcular la colisión en el siguiente, se puede reducir bastante el tiempo de cálculo para cada uno de los frames. La medida de los cálculos provenientes de frames anteriores que son reutilizados es lo que se denomina coherencia temporal entre frames (Van Den Bergen 2004). La coherencia temporal suele ser utilizada por aplicaciones para el cálculo de la animación (Zhang and Yuen

2002) o en el ámbito de la robótica (M. C. Lin 1993). En la bibliografía se pueden encontrar una serie de artículos (Jimenez, Thomas, and Torras 2001; J. D. Cohen et al. 1995; Ponamgi, Manocha, and Lin 1995; Hudson et al. 1997), que utilizan la coherencia temporal y la geométrica para disminuir el tiempo de cómputo en detectar las colisiones. En estos artículos se presentan algoritmos que calculan las distancias entre los elementos de dos objetos, seleccionando los elementos más cercanos. De esta forma en el siguiente cálculo sólo se tiene que calcular la distancia entre estos elementos cercanos y los vecinos a estos.

En otros artículos (J. D. Cohen et al. 1995; Ponamgi, Manocha, and Lin 1995) se utiliza la coherencia temporal para hacer un filtrado entre los pares de objetos comprobando que los cambios de los objetos entre dos frames consecutivos son mínimos.

2.2.1.5.2. Coherencia espacial o geométrica.

La coherencia espacial o geométrica de los objetos permite saber cuál es la distribución de los objetos dentro de una escena, permitiendo que se pueda calcular el grado de separabilidad de todos los objetos que forman el modelo (Van Den Bergen 2004). Los objetos normalmente están en una zona del espacio con pocos vecinos alrededor (Lawlor and Kalée 2002). En este caso sólo se tiene que comprobar la colisión entre objetos que ocupen la misma zona y entre objetos que sean vecinos de zona. Dos objetos se pueden separar si y sólo si las regiones definidas por sus envolventes convexas son disjuntas.

La coherencia espacial permite utilizar volúmenes envolventes para simplificar objetos, haciendo que la comprobación de la colisión entre los objetos sea más sencilla, ya que en vez de utilizar el objeto tal cual se utiliza la envolvente simplificada del mismo. Como la probabilidad de que dos volúmenes envolventes colisionen es baja esto hace que se reduzca el tiempo total de cálculo, ya que sólo se tiene que calcular la colisión exacta entre objetos cuyas envolventes colisionen.

2.2.1.6. Detección de colisiones en escenarios estáticos frente a escenarios dinámicos

La detección de colisiones en un escenario estático consiste en encontrar, en un instante dado, los objetos que se encuentran en colisión. En un escenario dinámico los objetos describen una trayectoria a lo largo del tiempo por lo tanto habrá que calcular la colisión a lo largo de la trayectoria. El cálculo de la colisión en escenarios

dinámicos mediante el uso de un ordenador no es posible, ya que estos funcionan de manera discreta. Una solución para poder emular un escenario dinámico es mediante una sucesión de escenarios estáticos (Held, Klosowski, and Mitchell 1995). A esta solución se le da el nombre de seudo-dinámica. La solución estática se diferencia de la seudo-dinámica en que en la estática no se tiene información del movimiento en los instantes anteriores, mientras que en la seudo-dinámica si. El gran inconveniente que presenta la solución seudo-dinámica es la elección del intervalo de discretización del tiempo. Si este es muy pequeño el cálculo de la colisión será muy grande, mientras si es muy grande puede darse el caso de que no se detecten algunas colisiones, por producirse estas en medio del intervalo. Una solución al problema de calcular el tamaño del intervalo de discretización del tiempo se presenta en (Jimenez, Thomas, and Torras 2001) donde se propone un método de cálculo que adapta el tamaño del intervalo al movimiento de los objetos. En (Smith et al. 1995) se resuelve el problema enfocando la solución de una manera diferente, ya que este parte de la premisa de que el desplazamiento de los objetos entre frames es muy pequeño, con lo cual no se producen saltos y se detectan todas las colisiones. El problema que presenta esta última solución es cuando en el escenario se encuentran objetos que se mueven muy rápidamente y su desplazamiento es muy largo. Para estos objetos los autores utilizan volúmenes contenedores a lo largo del tiempo.

En la bibliografía se pueden encontrar otros métodos que resuelven el problema seudo-dinámico, como por ejemplo el planteado en (Aliyu and Al-Sultan 1999) y explicado en (Rozas Merino 1997; Jimenez, Thomas, and Torras 2001), donde el cálculo se realiza a lo largo de un intervalo de tiempo.

2.2.1.6.1. Objetos estáticos frente a objetos móviles

Diferentes autores han tratado de clasificar el problema de la detección de colisiones entre los objetos que se incluyen dentro de una escena. Hay autores que hablan de simulación de mecanismos (Garcia-Alonso 1990), otros de simulación dinámica (Mirtich 2000) y por último otros que clasifican el problema en función del número de objetos que se incluyen dentro de la escena.

En la bibliografía especializada lo común es encontrar algoritmos que resuelven dos tipos de escenarios:

- Escenarios en donde sólo hay un objeto en movimiento y el resto de los objetos están estáticos. Este problema se conoce con el nombre de *n-body* estático. Ejemplos de algoritmos que resuelven este problema se pueden

encontrar en (J. D. Cohen et al. 1995; Wilson et al. 1999). Estos algoritmos se suelen utilizar en aplicaciones que emulan un entorno virtual en el que el usuario puede tener una interacción directa con el entorno. En este problema se encuentran aplicaciones que tratan todos los objetos (los no móviles y el móvil) como objetos independientes (Wilson et al. 1999) y otras que unen todos los objetos no móviles en un solo objeto (McNeely, Puterbaugh, and Troy 1999).

- Escenarios en donde hay *n* objetos (con *n* mayor que dos) que se pueden desplazar siguiendo una trayectoria. Este problema se conoce con el nombre de *multi-body* o *n-body* (Selim and Almohamad 1999). Ejemplos que resuelvan este problema se pueden encontrar en (Smith et al. 1995) donde el algoritmo de detección de colisiones puede trabajar con decena de objetos, en (Mirtich 2000) que puede detectar colisión en escenarios formados por cientos de objetos y en (J. D. Cohen et al. 1995) donde se puede llegar a trabajar hasta con miles de objetos.

2.2.1.6.2. Información previa sobre el movimiento de los objetos.

Un criterio útil a la hora de determinar la colisión de objetos que se desplazan dentro de un escenario es saber, a priori, las posiciones y las orientaciones de los objetos incluidos en la escena, así como las rutas que van a seguir cada uno de ellos. Conocer esta información a priori sobre el movimiento permite calcular la función paramétrica del tiempo. Esta permite calcular las colisiones entre objetos a priori, pudiéndose, por ejemplo, utilizar tal función para planificar el camino que seguirán los robots por una escena, evitando que colisionen (Latombe 1991; Selim and Almohamad 1999).

En entornos donde no se conozca a priori el movimiento de los objetos no se podrá calcular la función de tiempos. Ejemplos de este caso pueden ser aplicaciones que simulen fuerzas dinámicas (Moore and Wilhelms 1988; Hahn 1988; Pentland 1990), ya que en este caso los objetos se mueven por la escena siguiendo leyes dinámicas. Otro ejemplo de este tipo de algoritmos son lo que emulan la realidad virtual (McNeely, Puterbaugh, and Troy 1999). Aquí los objetos se mueven siguiendo las órdenes que dé el usuario que esté utilizando la aplicación en un instante dado.

2.3. Elección de la representación de los objetos.

De todos los tipos de representaciones de objetos que se pueden encontrar dentro de la bibliografía, la representación poligonal es la más natural, tanto para los objetos como para la escena, ya que todas las tarjetas gráficas actuales utilizan el triángulo como primitiva de renderización.

La representación poligonal más utilizada es la B-Rep, que suele estar formada por una colección desordenada de polígonos que no tienen información de cómo se conectan ni de cómo se relacionan unos con otros. Los algoritmos que funcionan con este tipo de representaciones suelen se aquellos que se pueden aplicar a cualquier colección de polígonos. Pero estos tienen el inconveniente de que suelen ser menos eficientes y menos robustos que aquellos que incluyen información adicional, tales como qué aristas se conectan con qué vértices, qué caras se forman con qué aristas, qué caras son vecinas de una cara, si el objeto es un sólido cerrado y si el objeto es cóncavo o convexo.

La representación geometría de un objeto se puede utilizar para la detección de colisiones, aunque esto no es lo ideal, ya que la representación geométrica está pensada para el renderizado del objeto, no para ser utilizada por un sistema de detección de colisiones. La mayoría de sistema de detección de colisiones utilizan dos representaciones para los objetos, una para su renderizado y otra para la detección de colisiones. La separación de la representación de renderizado de la de detección de colisiones se puede justificar por los siguientes motivos:

- Las tarjetas gráficas actuales utilizan una geometría de representación muy compleja, que no es la más adecuada para la utilización de los sistemas de representación de colisiones, ya que, por ejemplo, para una aplicación que necesite una respuesta rápida y no muy precisa a la detección de la colisión se podría utilizar una geometría simplificada.
- El hardware gráfico actual está diseñado para trabajar con formatos de representación muy concretos que posibilitan el renderizado de los objetos de una forma muy rápida. Estos formatos no pueden ser utilizados directamente por los sistemas de detección de colisiones y necesitan transformarlos a una representación adecuada para poder utilizarse. Por este motivo se suele usar una geometría especial para la detección de las colisiones.

- Los datos requeridos para el renderizado de un objeto varían radicalmente de los que se utilizan para la detección de colisiones. Por ejemplo, los datos para el renderizado se suelen ordenar por tipo de material, mientras que los de detección de colisiones se suelen ordenar de manera espacial.
- El diseño de la geometría de la representación difiere en muchos casos de la geometría para el renderizado.
- En una simulación los datos utilizados para la detección de la colisión deben permanecer siempre, mientras que los datos para el renderizado no, por ejemplo cuando estos no sean visibles.
- La geometría original del objeto se suele dar como una malla o como un conjunto de polígonos mientras que la geometría necesaria para la detección se colisiones se suele dar como una representación del objeto sólido.

La utilización de dos geometrías diferentes tiene asociada una serie de inconvenientes, tales como:

- El coste de tener en memoria dos estructuras de datos diferentes.
- El tener que mantener estas dos estructuras.
- La gestión por separado de ambas puede dar lugar a diferencias entre las mismas.
- El tener que controlar varias versiones de las mismas.

Lo ideal es tener sistemas de colisión específicos para cada una de las estructuras de datos elegidas para la representación de los objetos, ya que es inviable el diseñar un sistema general que pueda recoger objetos y escenarios de cualquier representación.

Otro dato a tener en cuenta es el tamaño de los mundos que se van a simular. Cuando estos son pequeños, se pueden mantener en memoria, pero cuando son muy grandes hay que mantenerlos en memoria masiva y se va cargando solo la parte del mundo que se necesita en un instante dado.

Dependiendo de la aplicación, la detección de colisiones puede consistir en decir si dos objetos interseccionan o no, pero en otros casos puede que lo que interese calcular sea las partes de los objetos que interseccionan. Otras aplicaciones solamente requieren el cálculo de un punto de contacto, mientras que otras necesitan todos los puntos de contacto. Algunas aplicaciones, cuando los objetos intersecan, necesitan

conocer la profundidad de la penetración o la distancia mínima de traslación (la longitud del vector de traslación más corto que separaría los objetos).

2.3.1. Estructuras de datos para la detección de colisiones.

Para poder detectar la colisión entre dos objetos es necesario realizar un gran número de test entre parejas de objetos: bien entre distintos objetos o entre todos los polígonos que forman los objetos. Si utilizamos el método de la fuerza bruta, el número de pares de test que se tienen que realizar en el peor de los casos es del orden O (n^2). Para reducir el número de pares de test se pueden utilizar heurísticas o estructuras de datos espaciales. En la bibliografía se pueden encontrar una gran cantidad de técnicas (Garcia-Alonso, Serrano, and Flaquer 1994; Jimenez, Thomas, and Torras 2001; Ericson 2005) que permiten conseguir este objetivo. También se puede encontrar documentación de algoritmos y estrategias que permiten reducir el número de test (Jiménez, Feito, and Segura 2003; Jiménez, Feito, and Segura 2004) que se tendrían que realizar para detectar la colisión, utilizando tetra-trees y recubrimientos simpliciales.

Un estudio comparativo de tiempos de ejecución de estos algoritmos demuestra que la mayor parte se empleaba en comparar polígonos y objetos que no interseccionan. Para conseguir el objetivo de reducir el tiempo que se dedica para la detección de colisiones entre objetos que no interseccionan, se han desarrollado un gran número de técnicas, las cuales utilizan diferentes estructuras de datos que permiten clasificar los test de colisión entre objetos o polígonos, realizando sólo los test entre los objetos o polígonos que están cercanos. Las técnicas se pueden agrupar en los siguientes tipos:

- **Técnicas que subdividen o particionan el espacio**. En esta técnica se divide el espacio que ocupa toda la escena en pequeños volúmenes, restringiendo la dificultad y simplificando el número de test que se tendría que realizar entre los objetos y polígonos. Dentro de estas se encuentra las técnicas de vecindad, cuya característica principal es que guardan información de la proximidad de cada uno de los objetos que se encuentran en la escena.
- **Técnicas que utilizan volúmenes envolventes**. Los objetos que están dentro de la escena se envuelven con un poliedro cuya forma es mucho más simple que la del objeto. La detección de colisiones entre estos poliedros que

envuelven a los objetos es muy sencilla y rápida.

- **Técnicas de jerarquía de volúmenes**. Estas técnicas descomponen el objeto en una jerarquía de volúmenes. Se diferencian de las de volúmenes envolventes en que estas no descomponen la escena donde están incluidos los objetos.

En esta tesis se presenta un método que utiliza ideas de varias de estas técnicas, trabajando con objetos rígidos. A continuación, se detallan más en profundidad cada uno de estos grupos de técnicas.

2.3.1.1. Estructuras que subdividen o particionan el espacio.

Estas técnicas consisten en subdividir el espacio que ocupan todos los objetos que forman parte de la escena. Cada objeto se clasifica asignándole una zona de la partición. Para cada escena se guarda una lista de objetos con la zona o zonas que ocupan. Con la lista anterior se pueden conocer los objetos que están cercanos geométricamente. Los test de colisión sólo se realizarán entre los pares de objetos que compartan la misma zona o región espacial en un momento dado.

En la literatura específica se encuentran diferentes técnicas que utilizan la subdivisión o particionamiento del espacio, pudiéndose clasificar en dos grandes grupos:

- Los que utilizan una rejilla uniforme de subdivisión espacial.
- Los que utilizan una jerarquía de subdivisiones espaciales.

Seguidamente se detallan algunas de las técnicas más destacadas que se utilizan para cada uno de estos dos grupos.

2.3.1.1.1. Rejilla uniforme de subdivisión espacial.

En este método se subdivide el espacio en celdas idénticas, las cuales están unidas entre si, son disjuntas y se encuentran ordenadas dentro de una malla tridimensional regular y fija llamada rejilla (Hughes et al. 2013; Garcia-Alonso, Serrano, and Flaquer 1994). Las celdas en las que se divide el espacio pueden tener forma de paralelepípedo (Garcia-Alonso, Serrano, and Flaquer 1994) o de cubo (Held, Klosowski, and Mitchell 1995; McNeely, Puterbaugh, and Troy 1999), siendo todas las celdas iguales entre si y recibiendo el nombre de voxels. En la *Figura 4* se pueden ver algunos ejemplos en 2-D de tipos de estructuras con forma de rejilla.

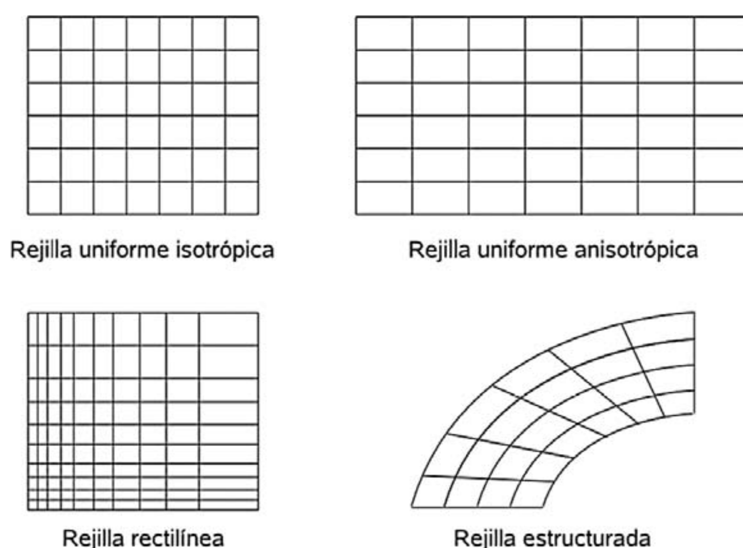


Figura 4. Ejemplo en 2-D de tipos de estructuras con forma de rejilla (Schroeder, Martin, and Lorensen 1996).

A cada objeto que se encuentra dentro de la escena se le asocia una lista con los voxels que ocupa. Para saber qué voxels ocupa un objeto hay que dividir cada uno de los puntos que lo determinan entre el tamaño del voxel, dando esto los distintos números de voxels. El tiempo necesario para calcular el/los voxels que ocupa un objeto es contante O (*1*). Además, utilizando esta técnica es muy fácil calcular los voxels adyacentes a uno dado. Para saber si dos objetos colisionan entre si se tiene que cumplir que estos deben ocupar el mismo voxel en un instante de tiempo dado. Con estas técnicas se clasifican los objetos por el voxel que ocupan, comprobándose solo la colisión entre objetos que ocupen el mismo voxel. Por la simplicidad de los cálculos para clasificar los objetos, esta técnica se ha utilizado mucho como método de descomposición espacial.

La técnica de división espacial se puede utilizar tanto en escenarios en los que se tienen todos los objetos en movimiento (Lawlor and Kalée 2002) como en aquellos en lo que se tengan varios objetos fijos y uno en movimiento (Savall et al. 2002).

Los problemas principales que presenta esta técnica son:

- La elección del tamaño del voxel para cada escena puede ser bastante complejo y costoso. Si, por ejemplo, se toma un tamaño muy pequeño, el número de voxels a actualizar es muy alto y muy ineficiente en tiempo de respuesta, ya que se tiene que emplear mucho tiempo para realizar

cualquier operación y se necesita un gran espacio de memoria para poder almacenarlo. Por contra, un tamaño del voxel muy grande con respecto a los objetos, provoca que muchos de ellos puedan caer dentro del mismo voxel, produciendo muchos falsos positivos, lo que implica tener que realizar muchos pares de test entre objetos y, por tanto, se ralentiza mucho la respuesta.

- Cuando los objetos que se tienen en la escena son muy complejos el tamaño del voxel debe ser más pequeño de lo normal, dividiendo incluso al objeto en diferentes partes o piezas más pequeñas.
- Cuando los objetos que se incluyen en la escena tienen tamaños muy dispares, siendo unos muy grandes con respecto a los otros, si se elige un tamaño del voxel adecuado a los objetos grandes este no funciona bien para los pequeños y si se escoge el tamaño adecuado para los objetos pequeños, los grandes ocupan muchos voxels, aumentando innecesariamente la memoria utilizada.

Debido a todos estos problemas, se deduce que para que los métodos que utilizan una rejilla uniforme de subdivisión espacial funcionen eficientemente es muy importante una elección óptima del tamaño del voxel, tarea que resulta bastante compleja, ya que se deben tener en cuenta todas las consideraciones comentadas en los puntos anteriores.

2.3.1.1.2. Jerarquía de particiones o subdivisiones espaciales.

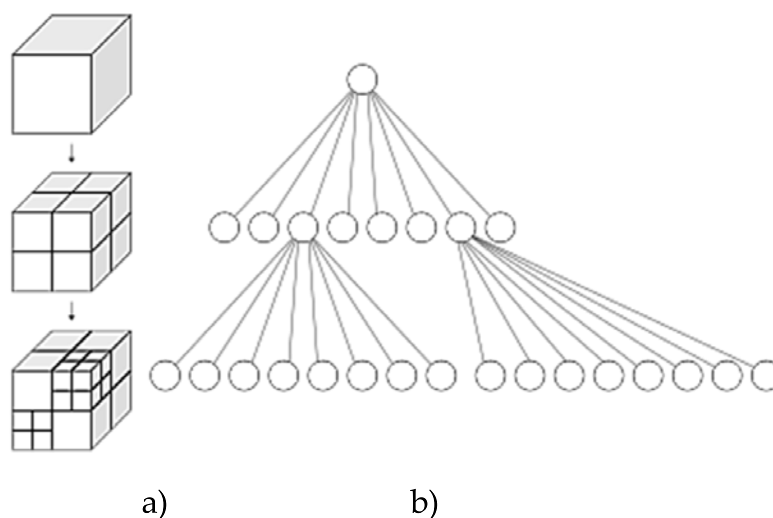
En la bibliografía se encuentran una serie de algoritmos que dividen todo el espacio ocupado por los objetos en una jerarquía de particiones o subdivisiones espaciales. Los volúmenes que se utilizan para particionar el espacio no intersectan entre ellos, salvo el que utiliza como jerarquía un árbol de esferas, que si intersectan por la forma geométrica de la esfera.

Los métodos que se estudian en este apartado aparecen para resolver el problema que presentan los métodos de subdivisiones uniformes, los cuales no tienen en cuenta la distribución en el espacio de los objetos ni su forma geométrica. En los métodos que utilizan subdivisiones uniformes las zonas del espacio en las que no hay ningún objeto ocupan el mismo espacio en memoria que las zonas en las que si están los objetos. Por esto aparecen estos nuevos métodos que utilizan una jerarquía para solo subdividir o particionar las zonas en las que están los objetos.

La utilización de particiones jerárquicas del espacio permite dividir más las zonas del espacio en la que se encuentran más objetos y menos las zonas donde no haya objetos, adaptándose de esta forma a las densidades locales del modelo.

2.1.2.1.2.1. Octree.

La palabra octree viene de la composición de dos palabras “Oct” + “tree”: árbol octal. Un *octree* parte del cubo que engloba el espacio que ocupa toda la escena. Este cubo tiene que estar alineado a los ejes de coordenadas. El cubo se divide en ocho cubos hijos alineados también a los ejes de coordenadas. Los cubos hijos que contengan a algún objeto de la escena se subdividen en otros ocho cubos hijos (ver *Figura 5*). Este proceso se va repitiendo de forma recursiva hasta que se llegue a la resolución deseada (Ayala et al. 1985; Hayward 1986; Vemuri, Cao, and Chen 1998; Swan 1993; Samet and Webber 1988).



a) b)

Figura 5. a) Subdivisión recursiva de un cubo en octantes. b) Octree correspondiente.(“Octree,” n.d.)

Los cubos en los que se subdivide la escena se numeran y se asignan a uno de los hijos del nodo. La numeración de cada uno de los hijos sirve para saber la posición de los hijos en la escena. En la *Figura 6* se puede ver la numeración de los nodos hijos según la posición que ocupan en la escena.

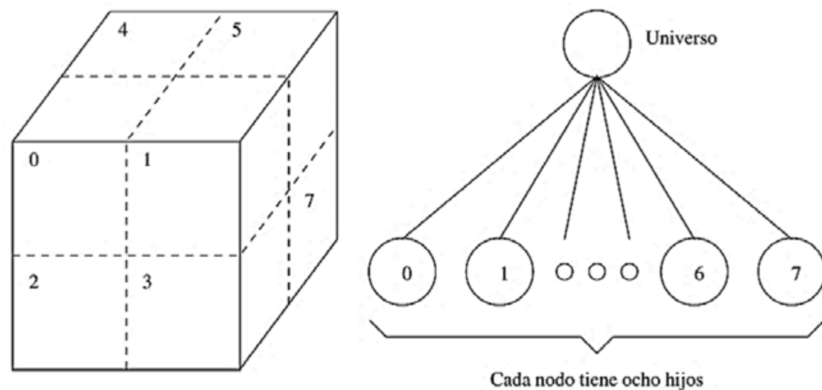


Figura 6. Numeración de los nodos hijo de un nodo padre (Samet and Webber 1988).

Cada nodo que se encuentra representado en el octree puede tener tres estados:

- Vacío (blanco): el nodo está totalmente fuera del objeto, es decir no contiene ningún volumen del objeto.
- Lleno (negro): el nodo está totalmente dentro del objeto.
- Parcial (gris): el nodo está parcialmente dentro del objeto, es decir contiene parte del volumen del objeto

Los nodos vacíos y los llenos son nodos finales o nodos *hoja* del *octree*, lo que significa que estos nodos no se particionan porque no tiene sentido hacerlo. Solamente se particionan los nodos parciales.

La gran ventaja que presentan los métodos basados en esta técnica es el gran ahorro en el almacenamiento de la escena, ya que el *octree* sólo guarda información de las zonas donde hay objetos que son las que están particionadas.

En cada uno de los nodos *hoja* del *octree* se guarda una lista de los objetos (Shaffer and Herb 1992) o los polígonos (Smith et al. 1995) incluidos en ese nodo.

La gran desventaja que presentan estos métodos es que el acceso a los objetos no se puede hacer directamente, se tiene que seguir un camino del *octree*, a diferencia de los métodos de partición uniforme en los cuales el acceso es directo. Los algoritmos que se utilizan para recorrer el *octree* suelen ser algoritmos recursivos. La complejidad de estos algoritmos en el peor de los casos es logarítmica $O(\log N)$, donde N es el número de nodos hoja. Por ejemplo, el coste medio de insertar un nuevo nodo en el *octree* es $O(\log N)$, y una vez construido el *octree* el coste de buscar un nodo hoja es $O(\log N)$. Como dentro del nodo hoja hay una lista con los objetos

que contiene el coste de buscar dentro de esta es O (q) siendo q el número de objetos que incluye.

2.1.2.1.2.2. Bintrees.

Los bintrees (Samet and Webber 1988) son estructuras de datos geométricas, más concretamente árboles binarios, los cuales clasifican todos los objetos que hay dentro de la escena según la posición que ocupan.

La construcción del bintree se realiza subdividiendo de forma recursiva el espacio finito del escenario en el que se distribuyen los objetos. Cada nivel del bintree representa una partición del espacio en dos, obteniéndose dos paralelepípedos que se asocian a cada uno de los dos hijos del nodo.

Los nodos se parten utilizando un plano de bisección (XY, YZ y ZX), que es distinto para cada uno de los nodos intermedios del bintree.

Los nodos que se obtienen pueden ser de cuatro tipos:

- Vacío (blanco): el nodo está totalmente fuera del objeto, es decir no contiene ningún volumen del objeto.
- Lleno (negro): el nodo está totalmente dentro del objeto.
- Parcial (gris): el nodo está parcialmente dentro del objeto, es decir contiene parte del volumen del objeto.
- Parcial Terminal (gris terminal): el nodo es igual que un nodo gris aunque siguiendo algún criterio, se decide que no se va a dividir más, pasando a ser un nodo hoja dentro del bintree.

Si se parte de una representación del objeto CGS se puede construir un bintree con un coste O (n), siendo n el número de nodos del bintree. Tanto el coste medio de insertar un objeto como el de buscarlo en el bintree es del orden de O ($log_2 n$). Al coste de buscarlo hay que añadir el coste de buscar en la lista de objetos que tiene el nodo hoja, el cual sería de O (q) siendo q el número de objetos que están en ese nodo.

Todos los bintree cumplen la regla de que $G = T - 1$, siendo G el número de nodos grises y T el número total de nodos hoja.

2.1.2.1.2.3. Point-Quadtrees.

Los point-quadtrees son estructuras de datos jerárquicas en las que se subdivide el espacio en cuatro subespacios.

La construcción del point-quardtree se realiza dividiendo de forma recursiva el espacio ocupado por la escena donde se incluyen los objetos. El espacio se divide utilizando líneas o planos paralelos a los ejes de coordenadas. La elección de los puntos por donde se va a cortar el espacio se puede realizar siguiendo varios criterios, por ejemplo, se podría tomar como criterio el punto más centrado. En la *Figura 7* se observa un ejemplo de división de un plano mediante un point-quatree.

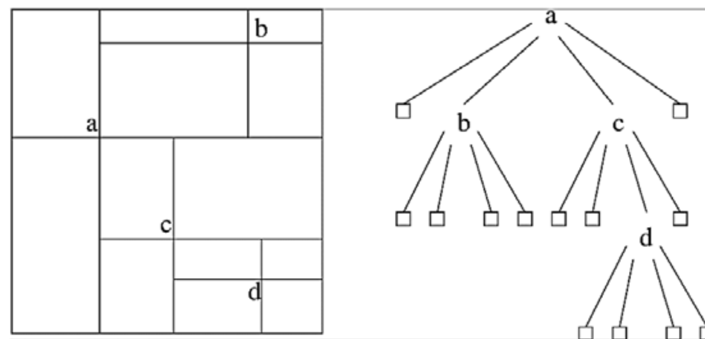


Figura 7. Ejemplos de división en un point-quatree (Brunet et al. 1999).

El coste para construir un point-quatree con n nodos en el peor de los casos es de $O((n * (n - 1))/2)$.

2.1.2.1.2.4. K-d trees.

El *kd-tree* (Held, Klosowski, and Mitchell 1995)(Zachmann 1997) subdivide el espacio en dos zonas y la subdivisión se realiza a lo largo de uno de los ejes de coordenadas. Para ello utiliza planos paralelos a los ejes de coordenadas del objeto. La letra k se refiere a la dimensión en la que se trabaja. Por tanto, el *kd-tree* organiza los puntos en un espacio euclideo de k-dimensiones. Todos los nodos que forman el kd-tree almacenan un punto o un objeto. Un ejemplo de un kd-tree de dos dimensiones se puede ver en la *Figura 8*.

El *kd-tree* puede realizar particiones variables o uniformes del espacio. Las particiones variables se adaptan mejor a los objetos. La estructura necesaria para almacenar los *kd-trees* con particiones variables es mucho mayor que la que utiliza particiones uniformes.

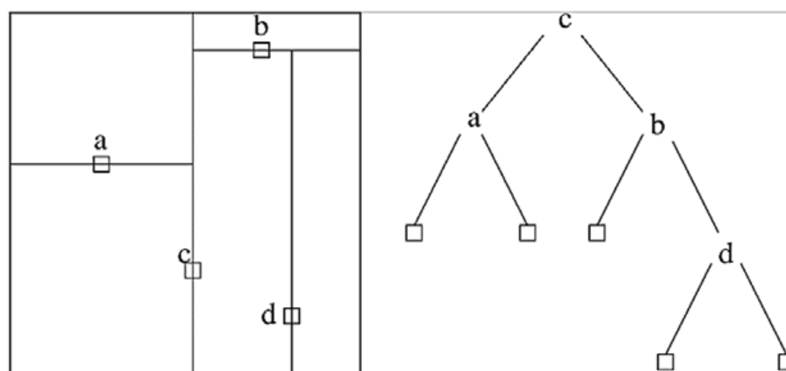


Figura 8. Ejemplo de kd-tree en dos dimensiones (Bentley 1975).

La creación del kd-tree se realiza partiendo del nodo que engloba toda la escena. Para ello se tiene que elegir el plano de corte ortogonal a uno de los ejes (X, Y ó Z) y la posición por donde pasará el plano de corte elegido por el eje de coordenadas.

Un ejemplo de construcción de un kd-tree tridimensional se puede ver en la *Figura 9*. La primera división (rojo) corta la celda raíz (blanco) en dos subceldas, que son divididas a su vez (verde) en dos subceldas. Finalmente, cada una de esas cuatro es dividida (azul) en dos subceldas. Dado que no hay más divisiones, las ocho finales se llaman hojas. Las esferas amarillas representan los nodos del árbol.

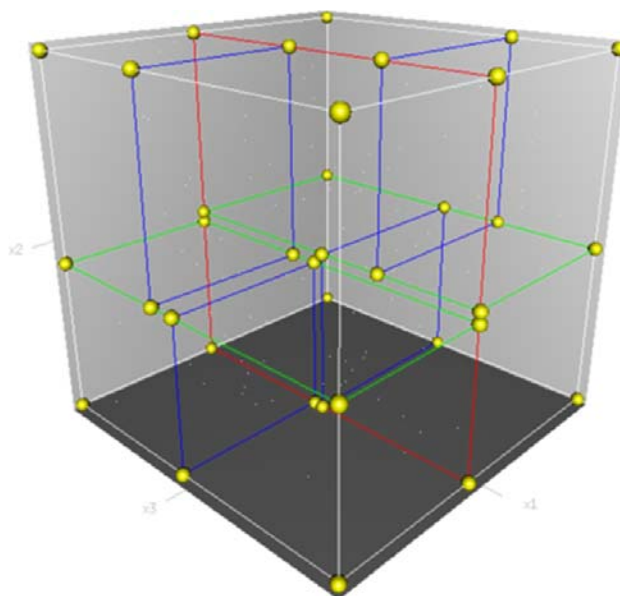


Figura 9. Ejemplo de particionamiento del espacio llevado a cabo por un kd-tree ("Árbol Kd," n.d.).

En el artículo (Van Den Bergen 1999b) realizan un estudio comparando los kd-tree con los *octree* demostrando que con los kd-tree se pueden tener menos celdas que con un *octree*.

El kd-tree puede dividir el espacio de muchas formas, ya que todo depende de cómo se cojan los planos de corte para cada uno de los nodos.

Un kd-tree está balanceado si todos los nodos hoja están a la misma distancia del nodo raíz.

Se puede construir un kd-tree balanceado si se cumplen las dos siguientes reglas:

- Conforme se va bajando por el kd-tree se escoge la orientación de los planos de corte siguiendo una serie. Por ejemplo, el nodo raíz se corta por un plano alineado al eje X, sus hijos se cortan por un plano alineado al eje Y y sus nietos al eje Z. Repitiendo esta serie por todos los niveles del kd-tree.
- La posición del punto en el eje de coordenadas por donde va a pasar el plano de corte se toma como la mediana entres los puntos ya incluidos en el kd-tree.

2.1.2.1.2.5. Vector octrees.

Un *vector octrees* (Brunet and Navazo 1992; Samet and Webber 1988) es un *octree* en el cual los nodos hoja pueden incluir información geométrica. En un vector *octree* se almacenan grupos de objetos u objetos de caras curvas aproximables por caras planas.

La construcción del *vector octrees* se suele hacer de arriba hacia abajo (*top-down*). Los nodos hoja en estos árboles guardan información geométrica de la cara. Por ejemplo pueden almacenar una parte de la frontera del objeto (un trozo de la cara). Por tanto, esto nodos no se tienen que dividir hasta llegar a nodos llenos o vacíos. Esta es la diferencia fundamental con los *octree* normales. Con esta representación se ahorra en espacio de almacenamiento, ya que estos árboles necesitan menos nodos para guardar la información.

Los nodos hoja ocupan mucho más espacio en memoria que los nodos hoja de un *octree* normal. La ventaja fundamental que tiene esta estructura es que necesita muchos menos nodos para representar un objeto que si se utilizase un *octree* normal.

En el *vector octrees* se encuentran los siguientes tipos de nodos:

- Nodos vacíos (blancos): no contienen ninguna parte del objeto. Estos nodos contienen un solo color.
- Nodos llenos (negros): están totalmente dentro del objeto. Estos nodos contienen un solo color.
- Nodos parciales (grises): están en la frontera del objeto. Estos nodos tienen ocho punteros a sus hijos.
- Nodos hoja cara: nodo hoja que contiene un puntero a una cara plana.
- Nodos hoja arista: nodo hoja que contiene una arista. Estos nodos tienen los punteros a las caras e información del semiespacio donde se encuentra el objeto.
- Nodo hoja vértice: nodo hoja que contiene un vértice. Almacena un puntero a las caras y la configuración del vértice.

El orden de eficiencia de estas estructuras es más o menos igual que el de un *octree* normal, ya que estos *vectores octrees* tienen menos nodos, pero el tratamiento de estos nodos es más complejo que en el caso de un *octree* normal.

2.1.2.1.2.6. BSP.

Un árbol de partición binaria del espacio *BSP* (Binary Space Partitioning tree) (Fuchs, Kedem, and Naylor 1980; Thibault and Naylor 1987; Naylor, Amanatides, and Thibault 1990) es una estructura de datos jerárquica. En dicha estructura se particiona recursivamente el espacio de la escena o los objetos en dos partes obteniendo celdas convexas. La partición se realiza seleccionando un plano con orientación y posición arbitraria. La selección del plano y su posición se suele realizar utilizando algunas técnicas heurísticas que intentan optimizar la construcción del *BSP* resultante.

Los *BSPs* inicialmente se utilizaban para trabajar con objetos (Naylor, Amanatides, and Thibault 1990) formados por poliedros. En estos casos los planos solían coincidir con los planos del objeto representado, aunque a veces se podían elegir otros planos arbitrarios para obtener un árbol más óptimo. En la *Figura 10* se puede ver un ejemplo de un poliedro y el *BSP* que se obtendría.

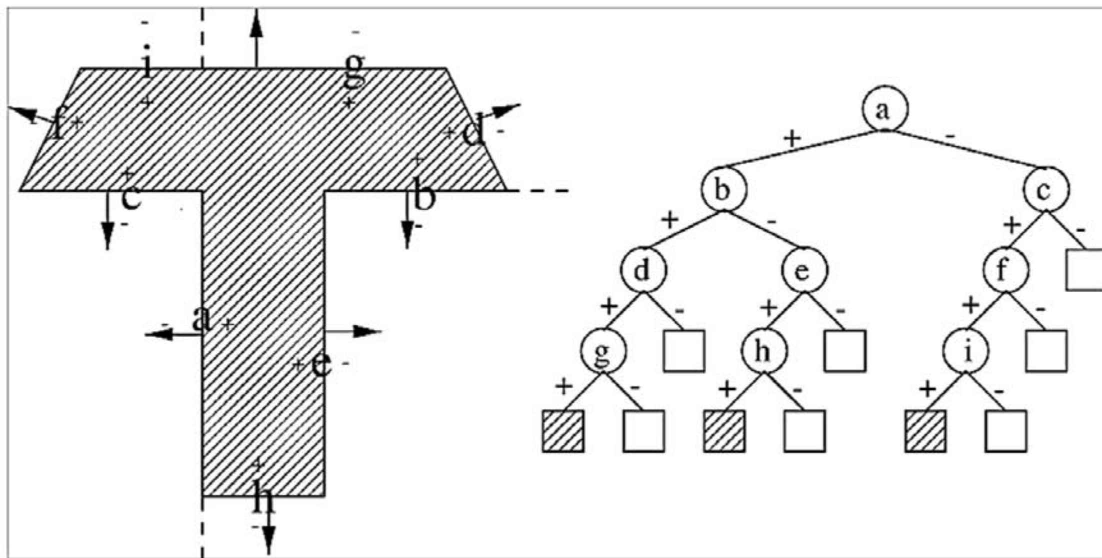


Figura 10. BSP de un poliedro (Brunet et al. 1999).

La naturaleza intrínseca de la estructura jerárquica de los *BSP* hace que su uso sea adecuado también en el modelado de sólidos, ya que es fácil de realizar operaciones (unión, diferencia, intersección, …) sobre ellos. Las operaciones geométricas se pueden aplicar sobre los *BSP* directamente transformando los planos contenidos dentro del nodo interior.

Para la construcción del *BSP* se parte del espacio del escenario. En el nodo raíz se introduce el plano seleccionado y la posición de este. A continuación, se estudian las dos partes obtenidas, viendo si cumplen una determinada condición. Si la cumplen se etiquita el nodo como un nodo hoja, si no la cumplen entonces se crean dos nodos internos. Este proceso se repite recursivamente hasta que todos los nodos sean hoja.

Cuando se utilizan los *BSPs* para clasificar un conjunto de objetos (Naylor, Amanatides, and Thibault 1990; Fuchs, Kedem, and Naylor 1980), los planos se seleccionan arbitrariamente de manera que los objetos no intersecciones. El objetivo es conseguir que el *BSP* tenga en cada nodo hoja un objeto. En la *Figura 11* se puede ver un ejemplo donde se clasifican un conjunto de objetos en 2D.

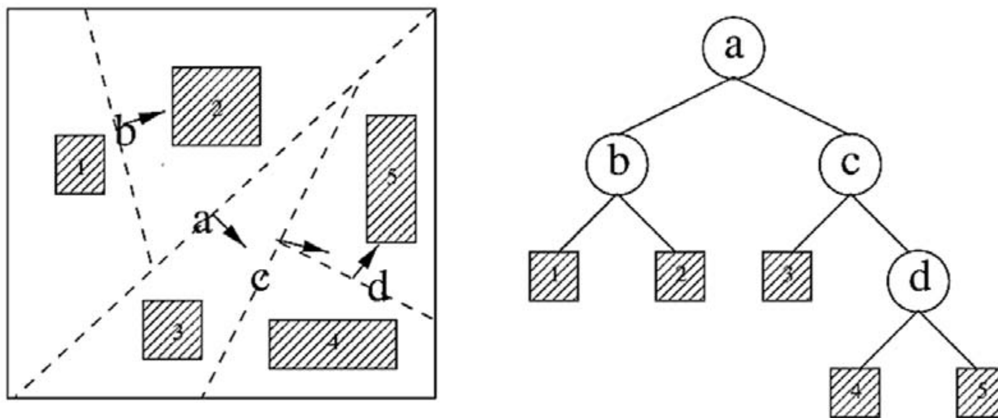


Figura 11. BPS de clasificación de un conjunto de objetos en 2D (Naylor, Amanatides, and Thibault 1990; Fuchs, Kedem, and Naylor 1980).

El problema fundamental que tienen los *BSP* es el coste que lleva asociado elegir bien los planos de corte para obtener un árbol con el menor número de nodos. El orden de eficiencia para crear un *BSP* a partir de una representación B-rep de un objeto con n poliedros es O (n). Para optimizar el *BSP* se pueden utilizar técnicas de *Backtracking* o Vuelta Atrás. El coste para insertar o buscar un nodo dentro de un *BSP* es de O ($log_2\, n$)

2.1.2.1.2.7. Box tree.

Los *box tree* (Barequet et al. 1996) son árboles en los que sus nodos son cajas en las que se incluyen los objetos.

La construcción del *box tree* se realiza de abajo hacia arriba. Se crea un nodo hoja para cada uno de los objetos que forman parte de la escena. A cada nodo hoja se le asigna la caja envolvente del objeto. Dos nodos hoja con sus cajas vecinas se fusionan en una sola caja que engloba a las dos. Con la nueva caja se crea un nodo intermedio. Este proceso se repite de forma recursiva hasta que se tenga la caja envolvente de toda la escena.

El orden de eficiencia de los algoritmos que trabajan con estas estructuras es de O ($log\, n$).

Con los *box trees* la detección de colisiones entre los objetos de una escena es muy eficiente, ya que sólo se tiene que comprobar si las cajas interseccionan. El cálculo de la intersección de dos cajas es muy rápido y sencillo.

2.1.2.1.2.8. BV-trees.

Los *BV-trees* (Held, Klosowski, and Mitchell 1996; Klosowski et al. 1998) son árboles binarios en los que cada nodo almacena una caja. Un *BV-tree* es una jerarquía de cajas envolventes. Para calcular las cajas envolventes se escoge un número fijo de planos n, los cuales se utilizan para cortar el espacio. De esta forma se crea la caja que contiene la escena. Los planos de corte tienen una orientación fija y lo único que se puede variar es la posición de estos en el espacio. Los planos se colocan tangencialmente a la escena con sus normales apuntando en dirección opuesta a la escena. Las intersecciones de todos los planos forman un semiespacio cerrado que envuelve a la escena. El volumen que forma el semiespacio interior cerrado de intersección de los planos se llama n-gono (ver *Figura 12*), donde *n* representa el número de planos utilizados.

La construcción del *BV-tree* comienza calculando el volumen envolvente de la escena con los planos seleccionados, el n-gono, el cual se asigna al nodo raíz del *BV-tree*. A continuación, se le asigna un punto a cada uno de los elementos que forman la escena (por ejemplo, los objetos). Los puntos se proyectan sobre los ejes de coordenadas, eligiendo el eje coordenado en el cual las proyecciones de los puntos tengan una mayor varianza. Sobre el eje coordenado seleccionado se busca la posición media de los puntos, siendo este punto medio el que se utilizará para dividir la escena en dos partes. Para cada una de las dos nuevas partes de la escena se calcula el n-gono (*Figura 12*). Con los dos nuevos n-gono se crean dos nodos que se colocan como nodos hijos del nodo raíz. Este proceso se repite recursivamente hasta llevar al nivel de las hojas, construyéndose así el *BV-tree*.

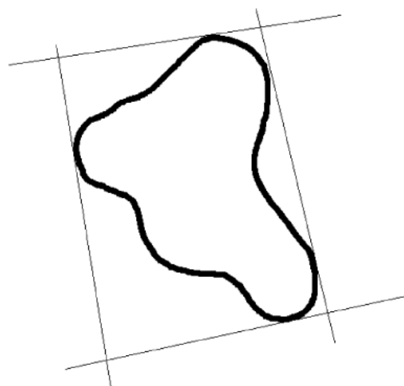


Figura 12. Ejemplo de un 4-gono en un plano 2D (Held, Klosowski, and Mitchell 1996).

El orden de complejidad de los algoritmos que trabajan con este tipo de árboles es de O (*log n*). Los *BV-tree* se pueden utilizar de una forma eficiente en aplicaciones en las cuales se tengan varios objetos en movimiento y se necesite una respuesta en tiempo real.

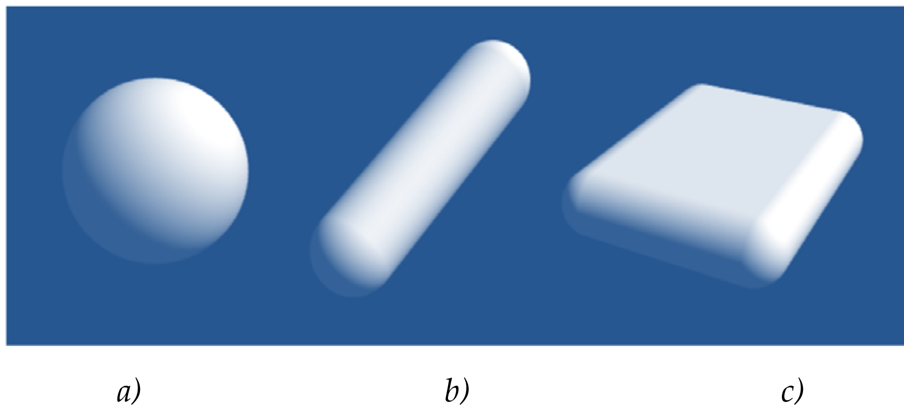
2.3.1.1.3. Estructuras que utilizan propiedades de vecindad

Las técnicas de vecindad consisten en clasificar los objetos que se encuentran en el espacio de tal forma que sólo se tengan que comprobar las colisiones entre los objetos que estén más próximos, descartando los objetos que no se encuentren cerca.

2.1.2.1.3.1. Swept spheres (Esferas de barrido).

En (Larsen et al. 1999) para calcular la cercanía de varios objetos se utilizan los volúmenes generados con una esfera de barrido. Los volúmenes se construyen utilizando una primitiva simple como puede ser un punto, una línea o un rectángulo. A la primitiva se le aplica una convolución o una suma de Minkowski, con una esfera. Los volúmenes que se pueden obtener utilizando esta técnica son (ver *Figura 13*):

- PSS (Point Swept Sphere) o esfera de barrido con un punto. Como la forma primitiva del núcleo es un punto, el volumen generado es una esfera. El volumen se representa con el punto del centro de la esfera y el radio.
- LSS (Line Swept Sphere) o esfera de barrido con una línea. Como la forma primitiva del núcleo es una línea, el volumen generado es un cilindro con los extremos redondeados. El volumen se representa con el segmento de la línea, el punto del centro y el radio de la esfera.
- RSS (Rectangle Swept Sphere) o esfera de barrido con un rectángulo. Como la forma primitiva del núcleo es un rectángulo 3D, el volumen generado es una caja con las esquinas redondeadas. El volumen se representa con el rectángulo, el centro y el radio de la esfera.



a) b) c)

Figura 13. Dibujado de volúmenes obtenidos con diferentes esferas de barrido: a) Un punto con una esfera de barrido. b) Línea obtenida con una esfera de barrido. c) Rectángulo obtenido con una esfera de barrido. (Larsen et al. 1999)

Esta técnica presenta varias ventajas entre las que se pueden destacar la facilidad para calcular la distancia y que; debido a las diferentes primitivas, se puede elegir la que mejor se adapte a la forma del objeto.

2.1.2.1.3.2. Regiones de Voronoi.

Las regiones de Voronoi es una técnica que utiliza la geometría computacional de un mapa de vecindad para clasificar los objetos que estén más próximos. Cada objeto que se incluye en la escena se representa con un punto. Cada punto se clasifica dentro de la región de Voronoi obteniendo la distancia de cada punto a todos los otros.

La técnica para calcular la colisión entre objetos se describe en (M. C. Lin 1993) (Ponamgi, Manocha, and Lin 1995). Para calcular la colisión se crea una región de Voronoi para cada una de las partes que forman un objeto: los vértices, las aristas y las caras. El proceso para detectar la colisión entre dos objetos consiste en tomar un par de elementos, uno de cada uno de los objetos que se quieren chequear. Si los dos elementos seleccionados están dentro de la misma región de Voronoi, entonces se puede afirmar que los objetos colisionan. Para todos los objetos que se tienen en la escena se guarda el par de elementos que son más cercanos.

Un ejemplo de las regiones de Voronoi asociadas a cada uno de los elementos de un objeto se puede ver en la *Figura 14*. Donde R_1 es la región asociada a la cara F_a y R_2 es la región asociada a la arista E_a y CP es el "Constraint Plane" entre las regiones R_1 y R_2.

El CP permite conocer la región vecina a una dada, hecho que se utiliza para saber

los test de detección que se tienen que hacer, ya que solo se realizan los test entre objetos que tiene regiones vecinas.

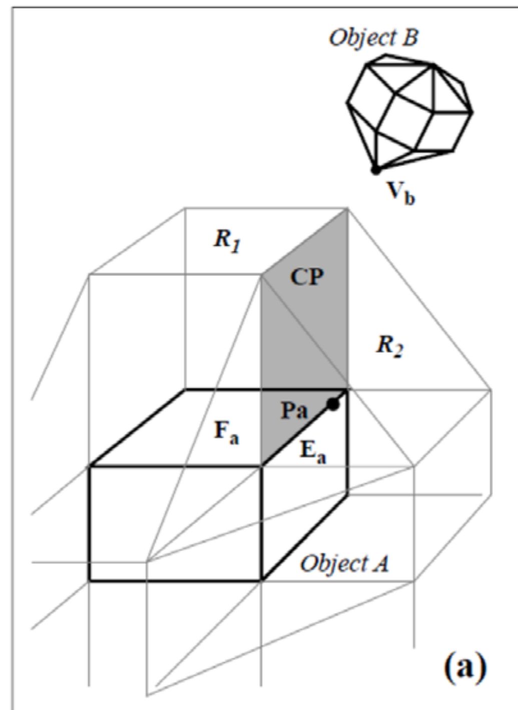


Figura 14. Ejemplo de cálculo de regiones de Voronoi entre dos objetos (M. C. Lin 1993).

2.3.1.2. Estructuras de simplificación del objeto.

La detección de colisiones entre objetos formados por varios miles de polígonos es muy costosa. Una técnica muy utilizada para resolver este problema es la de envolver los objetos en un volumen contenedor. Estos volúmenes envolventes suelen tener una forma simple de manera que comprobar la colisión entre dos volúmenes envolventes es una operación rápida y muy eficiente. Estas técnicas se utilizan para reducir el número de comparaciones que se tienen que hacer entre los diferentes objetos que se incluyen en una escena, ya que solo se realizan los test de colisión para los objetos cuyo volumen envolvente colisione.

Los métodos basados en esta técnica añaden un pequeño coste al proceso de detectar la colisión, pero si se compara este coste con respecto a la ganancia que proporcionan es mínimo, ya que se evita tener que realizar los test completos de colisión entre los objetos que no colisionan. En 1998 Suri y otros (Suri, Hubbard, and Hughest 1998) enumeran un teorema en el que se demuestra que el número de interseciones entre

volúmenes envolventes es muy similar al número de intersecciones entre objetos.

Los volúmenes envolventes que se utilizan para la detección de colisiones según esta técnica deben cumplir las siguientes propiedades:

- El coste para calcular el volumen envolvente debe ser mínimo.
- Tiene que ser muy sencillo el rotado o la transformación del volumen envolvente.
- El espacio en memoria para poder almacenar el volumen envolvente debe ser muy pequeño.
- El volumen envolvente se debe ajustar tanto como sea posible al objeto.
- El cálculo para la detección de colisiones entre los volúmenes envolventes tiene que ser mucho más rápido que si se tuviera que hacer entre los dos objetos completos.

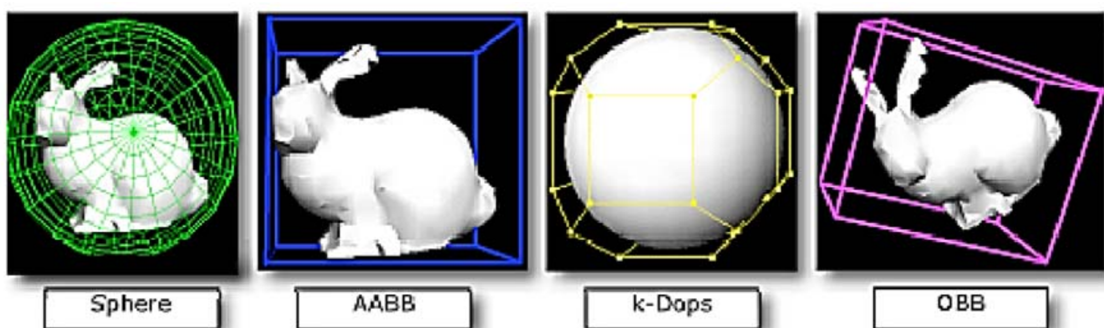


Figura 15. Formas de los volúmenes envolventes (Mohd Suaib, Bade, and Mohamad 2008).

2.3.1.2.1. Esfera envolvente.

En esta técnica los objetos se envuelven en una esfera. Para calcular el tamaño del radio de la esfera se necesita, según (Ritter 1990), recorrer dos veces todos los vértices de los poliedros que forman el objeto. Existe un algoritmo (Welzl 1991) que calcula la esfera de tamaño más ajustado que envuelve al objeto en tiempo lineal (Ver *Figura 15*). Este algoritmo presenta un problema: que es recursivo, lo que puede llevar a que la pila se desborde. En 2001 Capen (Capens 2001) diseñaron una reimplementación del algoritmo que evita el problema del desbordamiento. Aquí una esfera se representa con las coordenadas de un punto y el tamaño del radio.

La ventaja fundamental de utilizar el volumen envolvente en forma de esfera es que

el cálculo de la detección de colisiones es muy sencillo y rápido, sólo se necesita calcular la distancia entre los dos centros de las esferas y si este es mayor que la suma de los dos radios entonces es que las esferas no colisionan. Gráficamente se puede ver en la *Figura 16*.

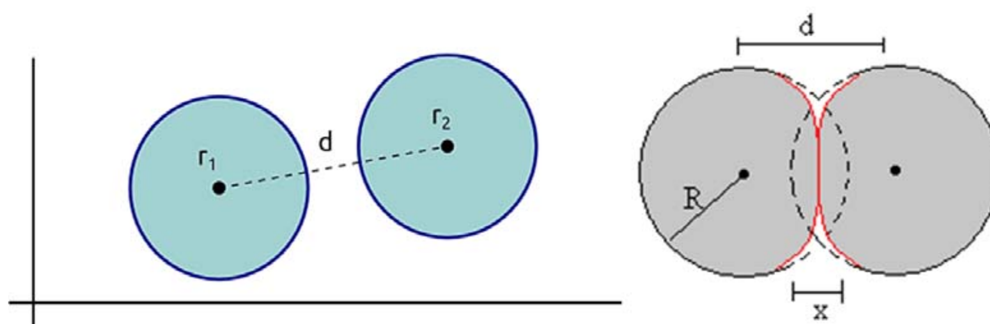


Figura 16. Esferas sin colisión y con colisión.

Otra ventaja que presenta el utilizar esferas para envolver a los objetos es que las rotaciones no afectan al resultado del test de colisión, solamente le afectan las translaciones y el escalado.

La desventaja fundamental que presenta el envoltorio en esfera es que este no delimita bien el objeto, ya que todo depende de la forma de este. Por ejemplo, si el objeto es muy alargado la esfera contenedora ocuparía mucho volumen en relación al objeto que envuelve.

2.3.1.2.2. Caja envolvente alineada a los ejes de coordenadas (AABB).

Esta técnica consiste en envolver los objetos en una caja rectangular alineada a los ejes de coordenadas AABB (Axis-Alignment Bounding Box) (Ver la *Figura 15*). Las normales de los planos que forma la caja rectangular son paralelas a los ejes de coordenadas.

Las AABB se representan con dos puntos en el espacio. Estos dos puntos se calculan buscando el valor máximo y mínimo de cada uno de los vértices en las tres coordenadas (ver *Figura 17*). El orden de eficiencia del algoritmo que calcula la caja envolvente es $O(n)$ siendo n el número de vértices del objeto.

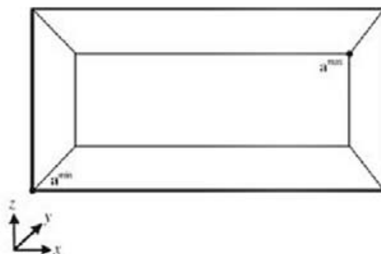


Figura 17. Caja AABB con los puntos máximo y mínimo que la representan.

Cuando los objetos se mueven por la escena, la caja envolvente AABB se tiene que recalcular para adaptarse a la nueva posición. Para ello se recorren de nuevo todos los vértices trasformados tras el movimiento, o bien se aplica la transformación del movimiento a los seis vértices de la caja envolvente, calculando después la nueva AABB a partir de los seis vértices trasformados. La segunda solución se calcula de forma muy fácil y rápida pero presenta los siguientes inconveniente:

- La nueva caja AABB calculada puede llegar a tener un volumen que duplique al de la caja original.
- Las cajas AABB no siempre se adaptan muy bien al objeto que envuelven. Por ejemplo si el objeto es muy largo, delgado y se encuenta orientado en una de las diagonales, la caja AABB ocupara un gran volumen.

Por el contrario la gran ventaja que presenta este método es la facilidad y la eficiencia de los cálculos para detectar la colisión entre dos cajas rectangulares alineadas a los ejes.

Cuando en una escena se tiene un gran número de objetos Cohen y otros (J. D. Cohen et al. 1995) proponen un método que consiste en proyectar las cajas AABB en los ejes de coordenadas. Cada una de las proyecciones se evaluan por separado para ver si existe colisión.

2.3.1.2.3. Caja envolvente orientada al objeto (OBB).

La técnica de envolver a los objetos en una caja orientada al objeto OBB (Oriented Bounding Box), consiste en definir una caja rectangular con una orientación arbitraria. La orientación de la caja que se escoge es la que mejor se adapte a la forma del objeto y que minimice el volumen de esta (Ver *Figura 15*).

La información necesaria para guardar la representación de la caja OBB es mayor que en el caso de las AABB. Por ejemplo, se pueden utilizar los ocho puntos de los

vértices de la caja o se puede utilizar el punto central de la caja, tres longitudes de los lados y una matriz de rotación (Ver *Figura 18*).

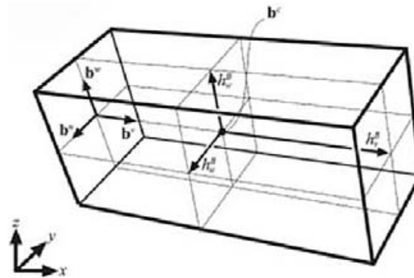


Figura 18. Punto central, tamaño de los lados y las nuevas coordenadas.

La detección de colisiones entre dos cajas OBB no se puede hacer directamente como en el caso de las AABB, si no hay que utilizar alguna técnica. Gottschalk y otros en 1996 (Gottschalk 1996) proponen una técnica basada en el teorema del eje separador, en la cual se detalla que con 200 operaciones se logra realizar este test.

En el test anterior, para objetos en dos dimensiones, el proceso consiste en tomar un eje del espacio sobre el cual se proyectan todas las cajas OBB de un intervalo. Si todos los intervalos proyectados no intersectan entonces se puede afirmar que los objetos no colisionan. Si por el contrario intersectan entonces hay que tomar otro eje y repetir el proceso (ver *Figura 19*). En tres dimensiones en vez de tomar ejes separadores, se toman planos separadores: los que son paralelos a alguna de las caras o de las aristas de cada uno de los poliedros. Como consecuencia de lo anterior dos poliedros convexos no colisionan si existe un eje separador perpendicular a alguna cara o arista de cada poliedro. Por esto, en el caso de dos cajas OBB, como mucho se tendrán quince ejes separadores.

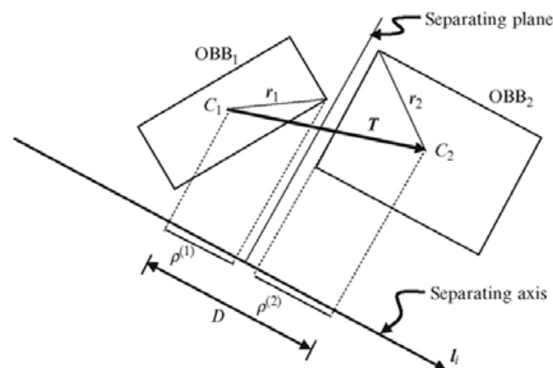


Figura 19. Proyección de dos OBB sobre un eje separador("Collision Detection (Advanced Methods in Computer Graphics)," n.d.).

A pesar de la complicación en los cálculos para detectar la colisión entre dos objetos envueltos con OBB, su uso se puede justificar porque la caja envolvente se adapta mejor al objeto y porque la actualización de la caja cuando se produce una rotación es muy rápida.

2.3.1.2.4. Poliedros discretos orientados (K-DOP).

Otra técnica para envolver los objetos es la utilización de poliedros discretos orientados k-dop (Discrete Orintation Polytopes) (Klosowski et al. 1998) (Ver ejemplo en la *Figura 15*). El cálculo del volumen envolvente se realiza utilizando slabs (región infinita del espacio delimitada por dos planos paralelos). Un ejemplo se puede ver en la *Figura 20*.

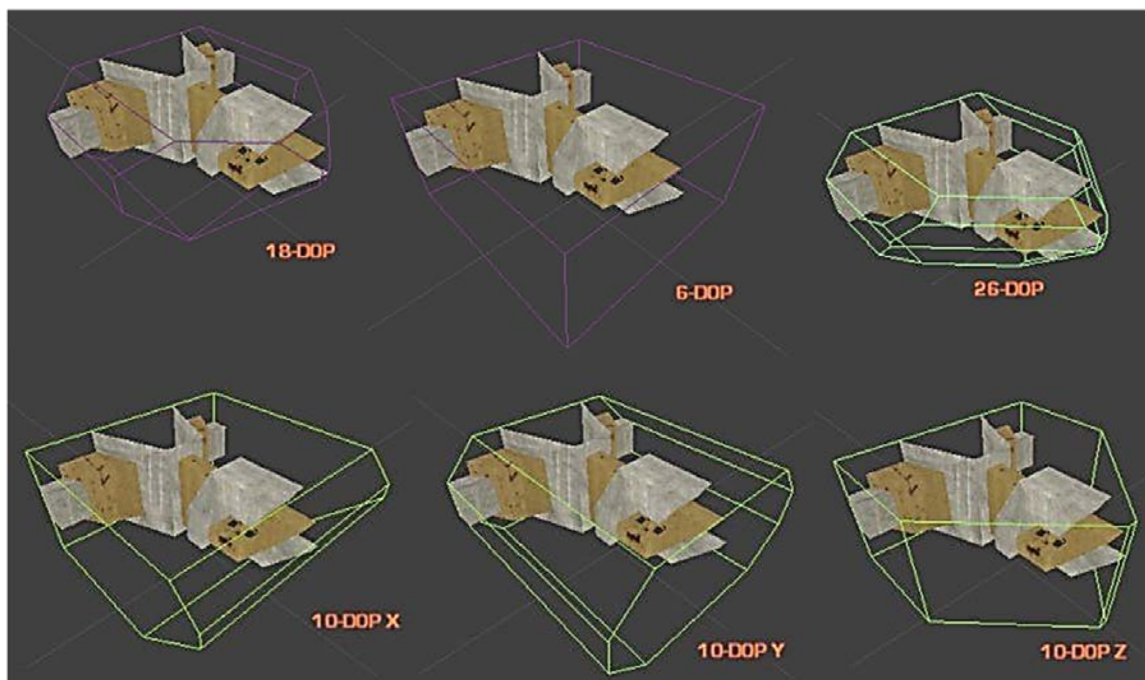


Figura 20. Ejemplos de K-dop de un objeto ("Add a K-DOP Collision Hull to a Static Mesh," n.d.).

Un 6-dop estará formado por tres slab y podría ser equivalente a un AABB. El 14-dop es uno de los más utilizados. Para obtenerlo se toman los seis planos del 6-dop y se le añade los planos definidos por las normales (1,1,-1), (1,-1,1), (1,1,1) y (1,-1,-1) y los planos opuestos a los definidos por las normales anteriores.

La construcción de un *k-dop* se realiza buscando, para cada una de las k direcciones, las posiciones extremas del objeto. Este proceso tiene un orden de eficiencia de O (*n*).

Una ventaja que presenta el envolver los objetos con una *k-dop* es que es muy fácil, rápido y sencillo realizar los test para detectar la colisión entre dos objetos. Para ello solo hay que realizar k/2 test de solapamiento. Otra ventaja es que desde el punto de vista geométrico los *k-dop* son las más eficientes de todas las anteriores.

Una desventaja que presentan los *k-dop* es el coste empleado en recalcularlos cuando el objeto se mueve por la escena. Como el coste de recalcular el *k-dop* es muy grande lo que se hace es trasformar el *k-dop* de acuerdo con el movimiento del objeto.

2.3.1.2.5. R-trees.

Los *R-tree* (Guttman 1984; Brown 2004) recogen información de un objeto, dividiendo este en subconjuntos de elementos más simples. La subdivisión del objeto se suele realizar utilizando un criterio. Normalmente el criterio utilizado se suele basar en una heurística geométrica.

Los nodos intermedios guardan una referencia a cada uno de los hijos y la información geométrica de la rama, como puede ser la caja envolvente. Los nodos hoja almacenan los componentes geométricos en los que se descompone el objeto. Los *R-tree* se crean de arriba hacia abajo. Las cajas envolventes de dos ramas del *R-tree* que representan una subdivisión del espacio suelen ser no disjuntas. En la *Figura 21* se puede ver un ejemplo de un *R-tree* en la que se divide un objeto en dos áreas.

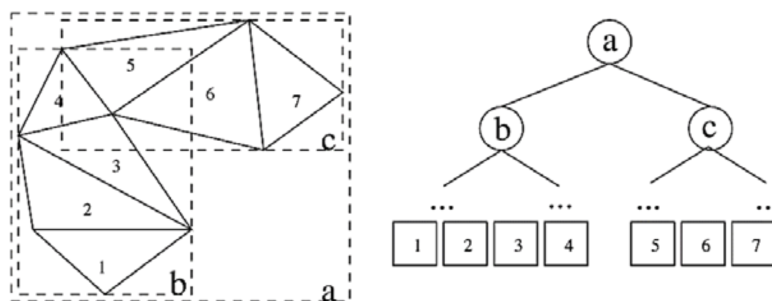


Figura 21. Ejemplo de un R-tree de un objeto (Brunet et al. 1999).

Los *R-tree* no suelen tener un buen rendimiento en el peor de los casos, pero con datos del mundo real se suelen comportar muy bien. El coste de insertar un nodo o buscar un nodo hoja es de O ($\log_2 n$).

2.3.1.3. Estructuras jerárquicas de volúmenes envolventes

Los métodos que utilizan jerarquía de volúmenes trabajan particionando los objetos tal cual, es decir, no particionan al espacio que ocupan estos, como hacen las técnicas

detalladas anteriormente.

Esta técnica se basa en ir construyendo la estructura jerárquica comprobando si el volumen contenedor de un nodo intersecta con el objeto. Si el nodo intersecta, entonces se tienen que añadir los nodos hijos y se repite el proceso de forma recursiva para cada nodo hijo. Si el nodo no intersecta con el objeto entonces sus nodos hijos tampoco lo harán con lo cual no se tienen que añadir a la jerarquía.

El proceso para detectar la colisión entre dos objetos consiste en comparar los dos nodos iniciales de las jerarquías de volúmenes. Si estos dos nodos intersectan se desciende por la jerarquía comparando los nodos hijos. Este proceso se repite hasta que se llegue a los nodos hoja en los cuales se realiza el test de polígonos para ver si los objetos interseccionan.

El problema fundamental con el que se encuentran estos métodos es la construcción de la jerarquía de volúmenes de forma eficiente. La construcción de la estructura jerárquica se puede realizar siguiendo tres métodos: los métodos *top-down* y *bottom-up*, que se utilizan para construir la estructura jerárquica off-line y el método de inserción, en el que se construye la estructura en tiempo real. El proceso de construcción para cada uno de los métodos será:

- Método *top-down*: Este método incluye todo el espacio ocupado por todos los objetos en el nodo raíz de la estructura. Este nodo inicial se irá subdividiendo en volúmenes hasta que se llega a los nodos hoja. Cada uno de los volúmenes que se generan en la estructura contendrá una primitiva o un conjunto de estas. Un ejemplo de construcción de la estructura siguiendo este método se puede ver en la *Figura 22*.

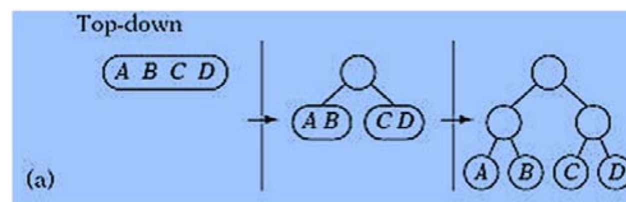


Figura 22. Construcción top-down de la jerarquía de datos.

- Método *bottom-up*: Este método asigna un conjunto de entradas a los nodos hoja. Partiendo de estos nodos se van agrupando las entradas, creando nodos intermedios. Este proceso se repite recursivamente hasta que toda la escena esté contenida en un solo nodo, llamado nodo raíz. Un ejemplo de

construcción de la jerarquía se puede ver en la *Figura 23*.

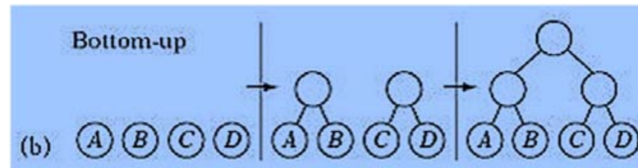


Figura 23 . Construcción bottom-up de la jerarquía de datos.

- Método de inserción: Este método parte de una estructura jerárquica vacía sobre la cual se van insertando uno a uno los elementos que forman parte de la escena. La posición en la que se crea el nuevo nodo debe hacer que la estructura jerárquica crezca lo mínimo posible. Para ello se utiliza una métrica de coste. Un ejemplo de este proceso de construcción se puede ver en la *Figura 24*.

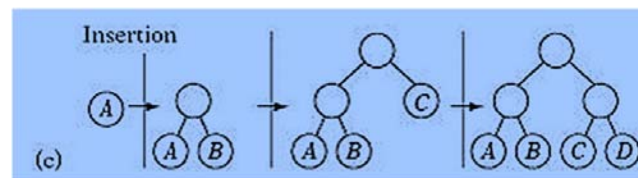


Figura 24. Construcción por Inserción de la jerarquía de datos.

El método *top-down* es el más utilizado y el más fácil de implementar, aunque la estructura jerárquica que se obtiene no es la que presenta mejor contención. El método *bottom-up* es mucho más difícil de implementar, pero la estructura jerárquica que se obtienen es la que presenta mejor contención. El método de inserción permite realizar la estructura jerárquica en tiempo de ejecución online y no necesita tener todas las primitivas antes de la construcción de la estructura, con lo cual se puede actualizar en tiempo de ejecución.

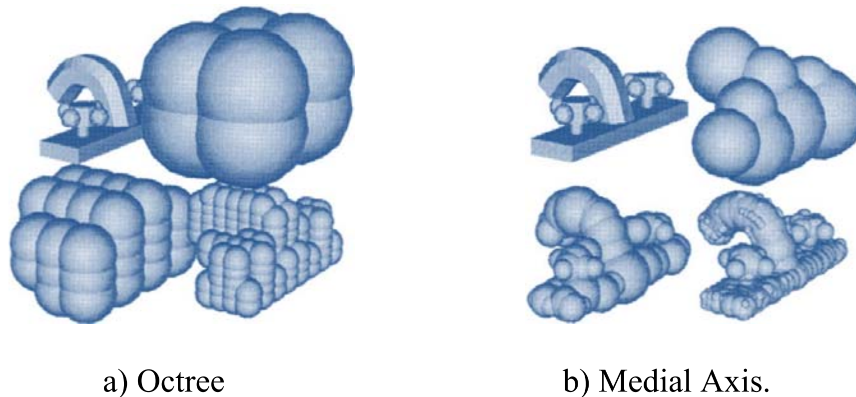
A continuación se va a hacer un resumen de las principales técnicas basadas en jerarquías de volúmenes.

2.3.1.3.1. Árbol de esferas SphereTree.

Estos árboles trabajan con una estructura jerárquica que utiliza como primitiva la esfera. Pero a diferencia de la que se ha visto en apartados anteriores, en este método las esferas se adaptan a la jerarquía del objeto (del Pobil and Ferre 1994; Hubbard 1995a; Dingliana and O'Sullivan 2000), mientras que los vistos anteriormente seguían un método de partición espacial basado en los octree.

En la literatura específica se puede encontrar la construcción del *SphereTree* siguiendo dos algoritmos basados ambos en un método *bottom-up*:

- En (Hubbard 1996) se comienza creando un esqueleto del objeto, que los autores llaman el Medial Axis. A continuación, se recubre este Medial Axis con esferas. De esta forma el recubrimiento del objeto se ajusta mucho mejor al mismo. En la *Figura 25* se puede ver un ejemplo en el que se compara la estructura jerárquica creada para un mismo objeto basada en octree y en Medial Axis.



a) Octree b) Medial Axis.

Figura 25. Ejemplo de Sphere Tree basado en Octree y Sphere Tree basado en Medial Axis. (Hubbard 1996)

- En (Quinlan 1994) se comienza creando una malla regular de esferas, todas del mismo tamaño, las cuales cubren a cada uno de los polígonos que forman parte del objeto. El centro de la esfera estará situado en el plano del polígono que envuelve dicha esfera. Cada una de estas esferas se asigna a un nodo hoja. Si el *SphereTree* es un árbol binario, entonces los nodos hoja obtenidos en el paso anterior se dividen en dos conjuntos, creando para cada uno de ellos un nodo intermedio que será padre de los anteriores. El proceso anterior se ira repitiendo de manera recursiva hasta llegar al nodo raíz. Se puede ver un ejemplo de dos objetos y los tres primeros niveles del sphere tree en la *Figura 26*.

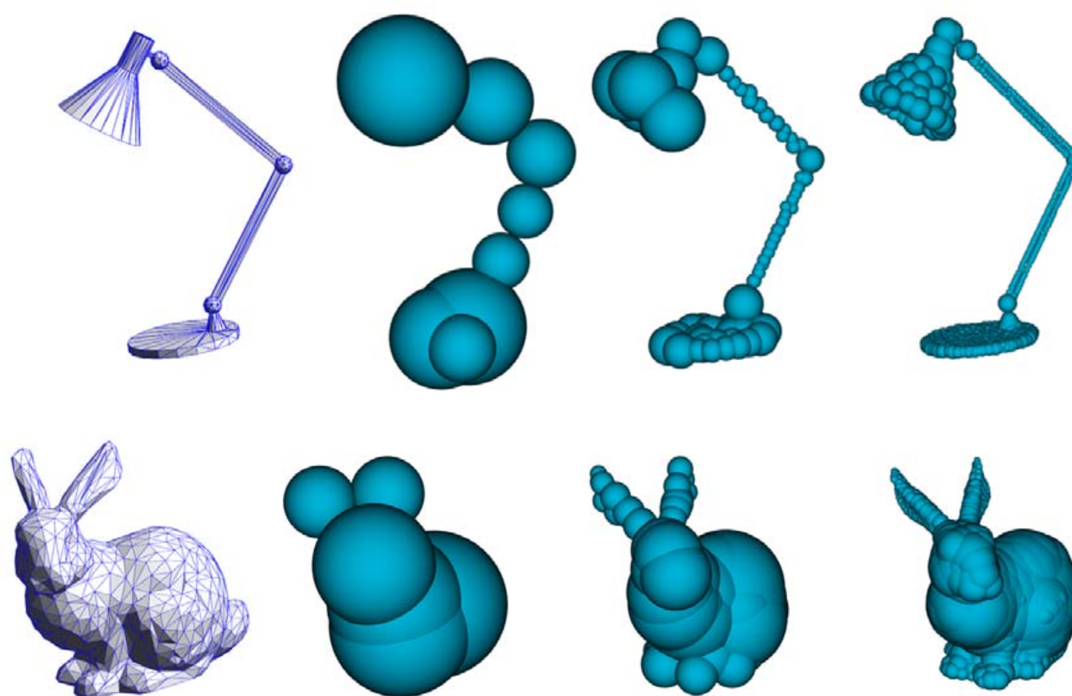


Figura 26. Modelo original y los tres primeros niveles del Sphere tree basados en la división planteada por Hubbard (Hubbard 1996).

2.3.1.3.2. Árbol de cajas alineadas a los ejes de coordenadas AABBTree.

Las estructuras jerárquicas basadas en cajas alineadas a los ejes de coordenadas (*AABBTree*) son muy utilizadas ya que tienen una ventaja fundamental: el coste para comprobar si dos objetos colisionan es muy bajo, ya que sólo hay que chequear cajas alineadas. Y esta tarea se puede realizar con unas sencillas operaciones.

La construcción del *AABBTree*, siguiendo el método de *top-down*, parte de un volumen contenedor con forma de caja rectangular alineada a los ejes de coordenadas, que engloba todo el objeto, asignando este volumen al nodo raíz. Para saber por dónde se tiene que dividir la escena se buscan todos los centroides de todos los triángulos incluidos en el nodo. El plano de corte seleccionado para dividir la escena debe pasar por la mediana de los centroides calculados anteriormente y además debe ser un plano ortogonal a uno de los ejes de coordenadas. Este proceso se va repitiendo de forma recursiva hasta que el número de triángulos que forman parte de un nodo sea inferior a un valor umbral dado, en cuyo caso se para el proceso y el nodo pasa a ser un nodo hoja. Un ejemplo de construcción de un *AABBTree* se

puede ver en la *Figura 27*.

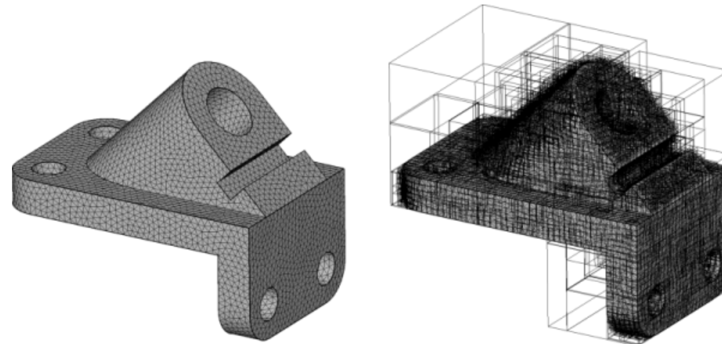


Figura 27. Malla triangular de una pieza y el AABBTree construido (Alliez, Tayeb, and Camille, n.d.).

Para comprobar si un objeto colisiona con alguno de los objetos incluidos en la escena se realizan los siguientes pasos:

1. Se calcula la caja de volumen envolvente para el nuevo objeto.
2. Se comprueba si colisiona con la caja envolvente de toda la escena. Si es así, entonces se comprueba para cada una de las cajas envolventes de los nodos hijos.
3. Este proceso se va repitiendo de manera recursiva hasta llegar a uno nodo hoja en el cual se comprueba si los triángulos de los objetos colisionan.

En la literatura especializada podemos encontrar diferentes artículos donde se utiliza este tipo de estructura para detectar la colisión entre objetos. En (Held, Klosowski, and Mitchell 1995) se hace una comparación con otros métodos basados en la partición espacial. También Van Den Bergen (Bergen 1997) utiliza un método basado en este tipo de estructuras jerárquicas para implementar un algoritmo que detecta la colisión entre objetos deformables. Van den Bergen demuestra que el algoritmo implementado se puede utilizar y es eficiente con objetos deformables, ya que los cálculos para la detección de colisiones son más rápidos que los obtenidos con otras estructuras de división jerárquica del espacio, como por ejemplo los *OBBTree*.

2.3.1.3.3. Árbol de cajas orientadas al objeto OBBTree.

Los métodos basados en construir una estructura jerárquica de cajas envolventes orientados al objeto OBB (Gottschalk et al. 1996), trabajan clasificando el espacio ocupado por los objetos, de manera que la detección de colisiones comienza

comparando los OBB que engloban a los dos objetos. Si estos colisionan entonces se baja por las estructuras jerárquicas comparando los OBB. Este proceso se repite de forma recursiva hasta llegar al último nivel de la estructura en donde se comprueba si los triángulos reales de los objetos interseccionan.

La construcción de la estructura jerárquica de cajas envolventes orientadas al objeto, se suele realizar siguiendo una metodología *top-down*. Los polígonos se irán asignando a un lado u otro de la estructura tomando el centro del polígono como referencia y comprobando si este se sitúa a un lado u otro del plano de corte. Un ejemplo de construcción del *OBBTree* se puede ver en la *Figura 28*.

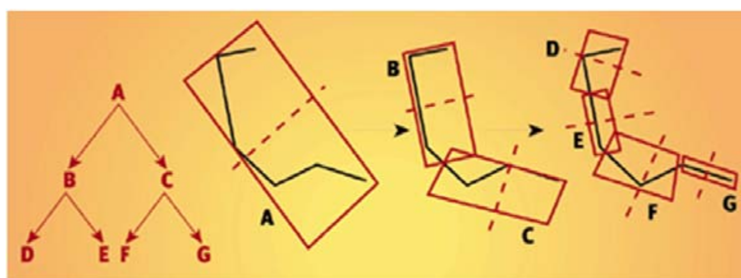


Figura 28. Proceso de construcción del OBBTree. (Gottschalk et al. 1996).

En la literatura especializada se encuentran artículos que utilizan *OBBTrees* para resolver la colisión entre objetos. Cabe destacar el trabajo de Barequet y otros (Barequet et al. 1996) en el cual se crea el *OBBTree* utilizando la técnica de *bottom-up*. Otro trabajo que utiliza los *OBBTree*s es el realizado por Hudson y otros (Hudson et al. 1997) en el cual la detección de colisiones entre objetos se hace en dos fases. En la primera se utiliza la técnica de Sweep and Prune para clasificar los posibles objetos que son potenciales de colisión y en una segunda fase utiliza los *OBBTree* para detectar la colisión sólo entre los objetos seleccionados en la fase anterior.

2.3.1.3.4. Árbol de poliedros discretos orientados K-Dop Tree.

En la literatura se pueden encontrar diferentes artículos que presentan varias técnicas que utilizan una jerarquía de volúmenes envolventes basadas en los *K-Dop Tree*. Entre ellos se puede destacar el trabajo realizado por Klosowski y otros (Klosowski et al. 1998). Estos autores crean una jerarquía de k-Dops utilizando una técnica de *top-down*, de forma que el plano de corte seleccionado para dividir el k-Dop tiene en cuenta los volúmenes de los hijos que resultarían tras la división. Otro artículo en que se utilizan los *k-Dop Tree* es el presentado por Zachmann (Zachmann 1998) en el cual se demuestra que utilizando un *k-Dop Tree* se obtienen mejores

resultados que utilizado un *OBBTree*.

2.3.2. Conclusiones sobre las estructuras para la detección de colisiones.

Tras el análisis y estudio de estas técnicas se puede hacer el siguiente resumen:

Para cada uno de los volúmenes contenedores vistos anteriormente, en la bibliografía se pueden encontrar trabajos que montan la estructura jerárquica de diferentes formas. Los test para detectar la intersección entre las dos estructuras jerárquicas de los objetos dependen del volumen contenedor elegido. De esta forma, si el volumen contenedor elegido es una esfera, los test de intersección son más eficientes que si se escoge el *k-dop*.

Los métodos que utilizan una estructura jerárquica de volúmenes envolventes han sido muy utilizados. El problema que presentan estas estructuras es la gran cantidad de datos que se tienen que manejar, del orden de varios Gigas para objetos cuyo tamaño no es mayor de varios cientos de miles de polígonos, haciendo que sea imposible trabajar con ellos en memoria principal. Wilson y otros (Wilson et al. 1999) proponen un método que permite trabajar con objetos de un tamaño máximo de quince millones de polígonos. Las estructuras jerárquicas para estos grandes modelos pueden llegar a ocupar hasta 1,3 GigaBytes, las cuales se almacenan en memoria secundaria, cargando sólo en memoria principal una pequeña parte de la jerarquía, más concretamente las zonas que se encuentran próximas y son más probables que sufran colisión.

En la bibliografía se pueden encontrar varios artículos que comparan las estructuras jerárquicas de volúmenes envolventes para identificar cuáles de ellas son las mejores. Caben destacar dos artículos: el propuesto por Gottschalk (Gottschalk et al. 1996) donde se describe una función para poder calcular el coste de estas estructuras, y el artículo de Klosowski (Klosowski et al. 1998) que modifica la función dada por Gottschalk.

2.4. Estructuras de datos y algoritmos para detección de colisiones entre objetos convexos.

Un buen trabajo en el que se trata este tema con profundidad es el de Teschner y otros (Teschner et al. 2005).

La detección de colisiones se presenta en una gran variedad de situaciones, lo que lleva asociado un gran número de problemas que se resuelven utilizando un conjunto de técnicas.
En la sección 2.3.1 se analizaron las diferentes técnicas que abordan el problema de la detección de colisiones. Todas intentan acelerar las operaciones, minimizando el número de cálculos y filtrando el número de comparaciones entre las primitivas utilizadas para modelar el escenario o los objetos.
Entre los artículos estudiados se pueden destacar una serie que describen taxonomías sobre la detección de colisiones (Hubbard 1996; M. C. Lin et al. 1996; O'Sullivan et al. 2001; Jimenez, Thomas, and Torras 2001).
Dentro de las técnicas que realizan una descomposición espacial podemos destacar las presentadas en los siguientes trabajos:

- Teschner y otros (Teschner et al. 2003) diseñan una técnica de hashing optimizada para objetos deformables.
- Le Grand (Nguyen 2007) describe cómo se puede implementar un enfoque de hashing espacial optimizado para GPUs.
- Pabst y otros (Pabst, Koch, and Straßer 2010) presentan una técnica de detección de colisiones para objetos rígidos y deformables que utiliza tanto la CPU como la GPU que permite trabajar con objetos compuestos por decenas de millares de triángulos obteniendo tiempos de respuesta en pocos milisegundos.

Dentro de las técnicas que hacen uso de jerarquías de volúmenes envolventes podemos destacar las presentadas en los siguientes trabajos:

- Thomaszewski y otros (Thomaszewski, Pabst, and Blochinger 2008) utilizan una descomposición dinámica de tareas de árboles de test de jerarquías de volúmenes envolventes en arquitecturas de memoria compartida.
- Kim y otros (D. Kim et al. 2009) usan un enfoque de jerarquía de volúmenes envolventes de descomposición de tareas similar a (Thomaszewski, Pabst, and Blochinger 2008) usando la CPU y en la GPU realizan test básicos de vértice-triángulo y arista-arista. Los cálculos en la CPU y en la GPU se

pueden superponer en un cierto grado lo que consigue una aceleración adicional.

- Lauterbach y otros (Lauterbach, Mo, and Manocha 2010) presentan un enfoque de jerarquías de volúmenes envolventes lineal basado en al GPU que produce unos resultados muy buenos.

Otros autores introducen técnicas para reducir el número de test elementales redundantes durante la detección de colisiones entre ellos podemos destacar (Wong and Baciu 2006; Tang et al. 2009).

Tang y otros (Tang, Manocha, and Tong 2010) proporcionan una técnica para reducir el número de test necesarios para saber si varios puntos están dentro del mismo plano.

En (Govindaraju et al. 2005; Sud et al. 2006) se hace uso de la GPU mediante el diseño de técnicas basadas en rasterización que utilizan el test de buffer de depth/stencil o funciones de distancia.

En (Argudo et al. 2016) se presenta una estructura de árbol multirresolución que permite seleccionar y editar, para su estudio individual, determinadas piezas que forman parte de un modelo CAD complejo.

En el libro (Weller 2013) se recogen una serie de estructuras de representación de modelos enfocadas a la detección de colisiones además de un grupo de algoritmos que permiten trabajar con los nuevos dispositivos hápticos.

En un escenario se pueden incluir objetos cóncavos y convexos. Los objetos cóncavos presentan una serie de propiedades que hacen que el cálculo de colisiones sea mucho más complejo. Para simplificar estos cálculos se suele utilizar una técnica que consiste en la descomposición de un objeto cóncavo en varios convexos. Si se revisa la bibliografía se pueden encontrar una gran cantidad de métodos para calcular la colisión entre objetos convexos (Van Den Bergen 1999a; Hubbard 1996; Jimenez, Thomas, and Torras 2001). La proliferación de estos métodos se basa, por un lado, en las propiedades que presentan los objetos convexos que hacen que se puedan desarrollar métodos que sean muy eficientes para la detección de colisiones, y por otro lado en que todos los objetos cóncavos se pueden descomponer en partes que sean convexos. Algunas de las propiedades que presentan los objetos convexos que los hacen tan adecuados para poder detectar la colisión de una manera eficiente son:

- La propiedad de que la distancia entre dos puntos presenta un mínimo

local y global (*Figura 29*). Esta propiedad consiste en que la distancia a la que se encuentra un punto *a* de un objeto *A* frente a otro objeto *B* es un mínimo local y global, con lo cual el cálculo de la distancia entre dos objetos convexos se puede realizar encontrando esa distancia global. El cálculo de la distancia global se puede realizar de una manera fácil y eficiente.

- La propiedad de la existencia de un plano de separación entre objetos. Esta propiedad se basa en el teorema que dice que si dos objetos convexos no están en contacto entonces se puede encontrar un plano de separación de manera que cada objeto quede en una zona del plano. De esta forma los algoritmos que calculan si dos objetos colisionan o no lo único que tienen que hacer es buscar dicho plano de separación.

A continuación, se presentan algunos de los algoritmos que más se han utilizado para calcular la detección de colisiones y que utilizan las propiedades que se han expuesto anteriormente.

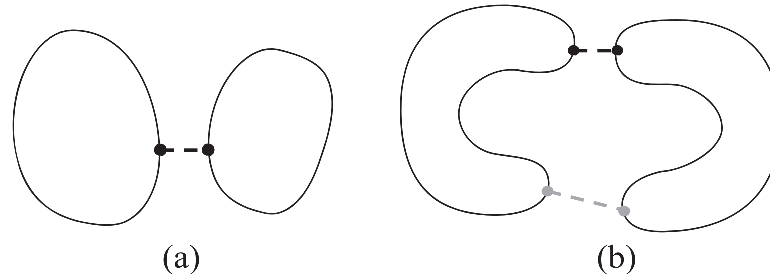


Figura 29. (a) Dos objetos convexos: la distancia mínima local entre dos puntos es siempre un mínimo global. (b) Dos objetos cóncavos: la distancia mínima local entre dos puntos (en gris) no es necesariamente un mínimo global (en negro). (Ericson 2005)

2.4.1. Detección de colisiones en objetos B-rep

Moore (Moore and Wilhelms 1988) en 1988 propone un algoritmo que permite calcular la colisión entre dos objetos (P y Q) que se encuentren dentro del mismo escenario. Los pasos de dicho algoritmo son:

1. Se comprueba, para toda arista del objeto P, si intersecciona con el poliedro

Q. Si la arista de P se encuentra parcial o totalmente dentro de Q, entonces los poliedros colisionan y se detiene el proceso.

2. Recíprocamente se comprueba si las aristas de Q interseccionan con el poliedro P. Si alguna arista de Q se encuentra parcial o totalmente dentro de P, entonces se para el proceso y los poliedros colisionan.
3. Por último se contempla el caso de poliedros degenerados, los cuales tienen sus caras alineadas. Aquí se compara el centroide de cada cara de Q con todas las caras de P. Si uno de los centroides de Q se encuentra contenido en P, entonces se detiene el proceso y se devuelve que los poliedros colisionan.
4. Se devuelve que los poliedros no colisionan.

El algoritmo anterior presenta un problema a la hora de realizar los dos primeros pasos y es que estos conllevan un gran coste computacional. Para solucionarlo se presentó en (Ericson 2005) el siguiente algoritmo:

1. Se compara cada vértice de Q con todas las caras de P. Si se encuentra un vértice dentro de todas las caras de P, se devuelven los poliedros como intersección y se detiene el proceso.
2. Se repite el paso anterior, pero ahora comprobando cada vértice de P con las caras de Q.
3. Se comprueban si las aristas de Q interseccionan con el poliedro P. Si alguna arista de Q se encuentra parcial o totalmente dentro de P, entonces se detiene el proceso y los poliedros colisionan.
4. Por último, se comprueba el caso de poliedros degenerados, los que tienen sus caras alineadas. Para estos casos se comprueba el centroide de cada cara de Q con todas las caras de P. Si uno de los centroides de Q se encuentra contenido en P, entonces se para el proceso y se devuelve que los poliedros colisionan.
5. Se devuelve los poliedros que no colisionan.

Estos dos algoritmos tienen un orden de eficiencia en el peor de los caos de O (n^2) tomando n como el número de vértices y las aristas de los objetos.

2.4.2. Algoritmos para detección de colisiones con objetos convexos

A continuación, se presentan algunos algoritmos o métodos que han optimizado el cálculo de la detección de colisiones:

- Algoritmo de Gilbert, Johnson y Keerthi: GJK (1988)

  El algoritmo diseñado por Gilbert, Johnson y Keerthi (GJK) (Gilbert, Johnson, and Keerthi 1988) es uno de los más rápidos y eficientes para resolver el problema de conocer si dos poliedros interseccionan entre si. Se basa en la utilización de simplejos, que son puntos, aristas, triángulos o tetraedros. Este algoritmo tomo como entrada dos conjuntos de vértices (A y B) y busca la distancia euclidea entre las envolventes convexas de los conjuntos anteriores. Para ello utiliza un método iterativo calculando en cada iteración el simplejo contenido en (A – B) que se encuentre más próximo al origen que el simplejo calculado en la iteración anterior. En la k-esima iteración tiene calculado un simplejo con un conjunto de vértices representado por V_k y el punto v_k que se encuentra más cercano al origen. Por ejemplo, si se quiere calcular la colisión de dos poliedros A y B, se calcularía la distancia más corta al origen por el simplejo formado por los vértices del poliedro resultante de la diferencia de Minkoski (A – B) (Larsen et al. 2000). Esta distancia se corresponde con la distancia más corta entre los dos poliedros. Si el origen está dentro del poliedro resultante es porque los poliedros iniciales int ersecciona n. La distancia del simplejo más cercano al origen da el grado de interpenetración que presentan los objetos.

  Como el cálculo de la diferencia de Minkowski entre dos objetos es muy costoso, el algoritmo GJK utiliza funciones de soporte que permiten obtener puntos de la diferencia de Minkowski.

- Algoritmo del vector de separación de Chung Wang (1996)

  El algoritmo del vector de separación de Chung Wang (Chung and Wang 1996; Chung 1996) se puede descomponer en dos algoritmos. El primero de ellos toma un vector como posible candidato que proporciona una separación entre los objetos. En el caso de que no se produzca intersección entonces iterativamente se irá calculando un vector más cercano al vector de separación. En la *Figura 30* se puede ver gráficamente el proceso para buscar el vector separador. El segundo algoritmo sirve para detectar cuándo dos

objetos colisionan, ya que en este caso el primer algoritmo estaría iterando de manera indefinida.

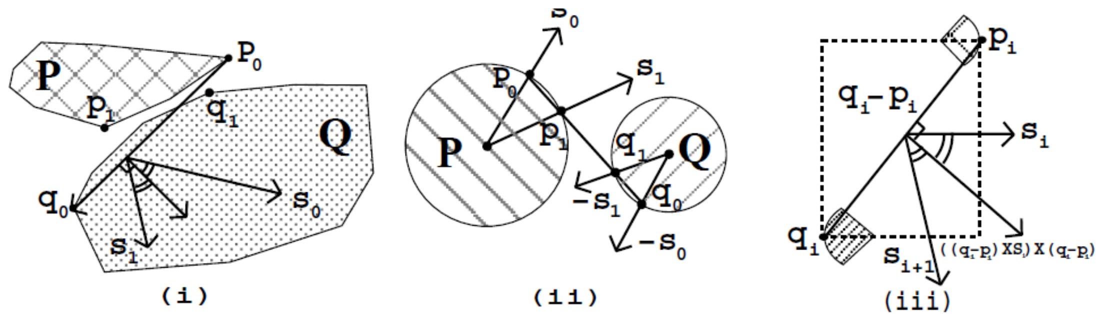


Figura 30. (i) Búsqueda de vector separador, (ii) la idea, (iii) en el caso de círculos eligiendo S_{i+1} (Chung and Wang 1996)

- Algoritmo de Mirtich: V-Clip (1998)

El algoritmo desarrollado por Mirtich, denominado Voronoi-Clip o V-Clip (Mirtich and June 1997) resuelve algunos de los problemas que presenta el algoritmo de Lin y Canny (M. C. Lin and Canny 1991). El V-Clip es capaz de tratar casos en los que los objetos penetran uno frente al otro. Es un algoritmo robusto, que puede trabajar con casos degradados y es más fácil de implementar que el de Lin y Canny.

Este algoritmo busca entre los vértices, las aristas y las caras cuáles son las más cercanas. Ya no utiliza los puntos más cercanos, ya que con estos, un pequeño cambio de orientación, podría hacer que un punto cercano pasase de un extremo a otro de una cara.

El algoritmo trabaja con las características de un poliedro. Una característica se define como un vértice, una arista o una cara del poliedro.

Para calcular si dos poliedros colisionan el algoritmo toma una característica de cada uno y comprueba si son las más cercanas globalmente. Si detecta que son cercanas el algoritmo para indicando que no hay colisión. Pero si no, el algoritmo actualiza una de las características con una vecina y vuelve a comprobar la cercanía de estas. Las características vecinas a un vértice son las aristas que inciden sobre él, la de una arista son los vértices y las dos caras que inciden sobre la arista y la de una cara son las aristas que la forman. Un ejemplo se puede ver en la *Figura 31*.

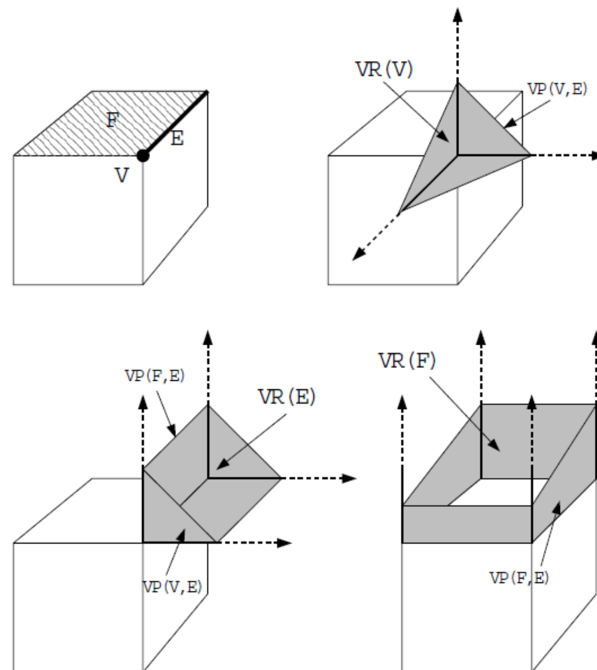


Figura 31. Arriba a la izquierda: Un poliedro cúbico. Entre sus características se encuentran la cara F, la arista E y el vértice V. Arriba a la derecha: la región Voronoi VR (V). Uno de los planos de Voronoi que delimitan esta región es VP (V, E), correspondiente a la arista vecina E. Abajo izquierda: la región de Voronoi VR (E). Dos de los planos de Voronoi que delimitan esta región son VP (V, E) y VP (F, E), correspondientes respectivamente a las características vecinas de E, V y F. Abajo a la derecha: la región de Voronoi VR (F). Uno de los planos de Voronoi que delimitan esta región es VP (F, E), correspondiente a la arista E vecina de F. El plano de soporte de F mismo también limita VR (F). (Mirtich and June 1997)

- Algoritmo de Ehmann y Lin (2000)

  El algoritmo de Ehmann y Lin utiliza una serie de volúmenes envolventes de diferente nivel de detalle para calcular la colisión entre dos objetos. Para cada objeto se calculan una serie de volúmenes envolventes con diferente nivel de detalle, acotados por un error. Con la serie de volúmenes envolventes se obtiene una jerarquía de Dobkin-Kirkpatrick (Dobkin and Kirkpatrick 1990) (Ver ejemplo de construcción en la *Figura 32*) la cual está formada por una secuencia de politopos convexos, siendo el primer politopo de la serie el politopo de entrada y cada uno de los politopos de la serie tiene un menor número de características que el anterior. Gracias a esta jerarquía el cálculo de un el punto más cercano a otro se puede realizar muy rápido.

Los tiempos para calcular si dos polígonos convexos intersecionan pueden ser del orden de O (*log n*), mientras que para comprobar dos poliedros puede ser de O ($log^2 n$).

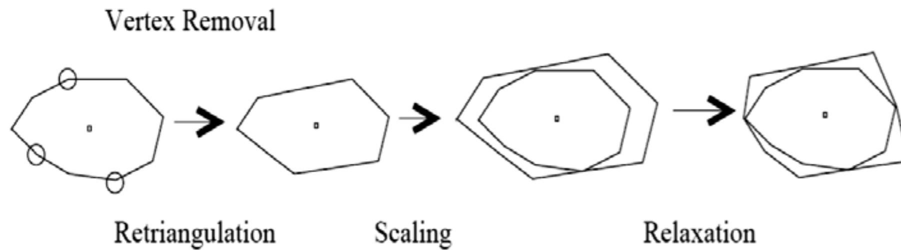


Figura 32. Ejemplo de construcción de una jerarquía de Dobkin-Kirkpatrick. (Dobkin and Kirkpatrick 1990)

- Algoritmo Pivot (Proximity Information From Voronoi Techniques) (2001)

Este algoritmo utiliza la aceleración hardware que proporciona la tarjeta gráfica para hallar la colisión entre modelos poligonales arbitrarios en 2D (Hoff III et al. 2001) y 3D (Hoff, Zaferakis, and Lin 2002). El algoritmo utiliza una combinación de la localización del objeto-espacio, técnicas de dibujado multipasada y cálculo de la distancia para hacer consultas de proximidad entre objetos de manera interactiva. Este algoritmo se puede utilizar con objetos poligonales cerrados no convexos, pudiendo ser estos objetos rígidos o deformables y de una complejidad alta.

Las consultas para calcular la proximidad entre objetos incluyen, no sólo la detección de colisiones, sino también el cálculo de intersecciones, distancia mínima de separación, puntos más cercanos, profundidad y dirección de penetración, y puntos de contacto y normales.

La carga de trabajo se equilibra entre los subsistemas CPU y gráficos (GPU) a través de una geometría híbrida y un enfoque basado en imágenes.

Este algoritmo realiza el cálculo de la colisión en dos pasadas. En la primera utiliza técnicas geométricas de espacio-objeto para localizar a grandes rasgos las interacciones potenciales de colisión entre los dos objetos. Y en segunda pasada se utiliza la aceleración del hardware gráfico para aplicar técnicas de espacio de imágenes, proporcionando la información de proximidad a bajo nivel.

2.5. Librerias y aplicaciones de código abierto para la Detección de Colisiones

Revisando la bibliografía sobre el problema de la detección de colisiones se pueden encontrar una serie de librerías y aplicaciones basadas en los métodos detallados anteriormente, que dan respuesta a las preguntas (distancia entre objetos, puntos de contacto, etc) sobre la detección de colisiones. A continuación, se describen las librerías más referenciadas, cuyo código esta accesible en internet y es de libre acceso.

- BSC: Reliable continuous collision detection (Tang et al. 2014). Implementa un algoritmo rápido para realizar consultas de detección de colisiones continuas (CCD) precisas entre modelos triangulares. El algoritmo utiliza una fórmula basada en las propiedades de la base de Bernstein y de las curvas Bezier, reduciendo el problema a la evaluación de los signos de polinomios. El algoritmo de CCD es geométricamente exacto y se basa en el paradigma de cómputo geométrico exacto para realizar consultas de colisión. El artículo donde se presenta este algoritmo y el código se pueden descargar en la dirección:

  http://gamma.cs.unc.edu/BSC/

- CULLIDE (Govindaraju et al. 2003), R-CULLIDE (Govindaraju, Lin, and Manocha 2006), y Q-CULLIDE (Govindaraju, Lin, and Manocha 2005): Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware. Este algoritmo presenta un nuevo enfoque para la detección de colisiones entre varios objetos deformables y frágiles en un gran entorno utilizando hardware gráfico. El algoritmo calcula un conjunto de zonas que pueden colisionar mediante consultas de visibilidad. No necesita precomputación y trabaja en varias etapas: primero calcula el conjunto de posibles colisiones a nivel del objeto. A continuación calcula el conjunto de posibles colisiones a nivel de sub-object, seguidamente calcula la detección de colisiones exactas. El cálculo de las posibles zonas de colisión se realiza con un algoritmo de procesamiento que

trabaja en dos pasadas, y tiene un orden de eficiencia lineal O(n). El enfoque global no hace ninguna asunción sobre las primitivas de entrada o el objeto del movimiento y es directamente aplicable a todos los modelos triangulares. Los artículos y el código se pueden descargar respectivamente de las siguientes direcciones de internet:

http://gamma.cs.unc.edu/CULLIDE/

http://gamma.cs.unc.edu/RCULLIDE/

http://gamma.cs.unc.edu/QCULLIDE/

- DEEP: Dual-space Expansion for Estimating Penetration Depth (Y. J. Kim, Lin, and Manocha 2002). Aquí se propone un algoritmo incremental para estimar la profundidad de penetración entre los politopos convexos en 3D. El algoritmo incremental busca una "solución local óptima" trabajando sobre la superficie de la suma de Minkowski, que se calcula implícitamente mediante la construcción de un mapa local de Gauss. En la práctica, el algoritmo funciona bien cuando hay mucha coherencia espacial de los objetos que se encuentran en movimiento por el escenario, siendo capaz de calcular la solución óptima en la mayoría de los casos. La dirección de internet donde se puede encontrar el artículo y el código es:

  http://gamma.cs.unc.edu/DEEP/

- DEFORMCD: Collision Detection for Deforming Objects (Tang et al. 2009). Se diseña un algoritmo interactivo para la detección continua de colisiones entre modelos deformables. Utiliza dos técnicas para mejorar la eficiencia de la selección y reducir el número de pares de triángulos candidatos potencialmente a la colisión. En primer lugar, se presenta una fórmula novedosa que usa conos continuos de normales y utiliza estos conos normales para eliminar eficientemente grandes regiones de la malla a partir de pruebas de auto-colisión. En segundo lugar, se explota la conectividad de la malla e introduce el concepto de "conjuntos de huérfanos" para

eliminar casi todos los test elementales redundantes entre triángulos adyacentes. Estas técnicas de selección se han combinado con las jerarquías de volumen obteniéndose unos resultados muy buenos en comparación con los algoritmos anteriores para modelos deformables. La dirección de internet donde se puede encontrar documentación y el código fuente es:

http://gamma.cs.unc.edu/CBC/

- DVD: Fast Proximity Computation Among Deformable Models Using Discrete Voronoi Diagrams (Sud et al. 2006). Son una serie de algoritmos que permiten realizar consultas de colisión y distancia entre múltiples modelos deformables en entornos dinámicos. Estas incluyen consultas inter-objeto entre objetos diferentes, así como consultas intra-objeto. Utilizan un método único para realizar las consultas, que se basa en el cálculo de distancias en sistemas n-body y en un segundo paso usan las propiedades del diagrama de Voronoi discreto (DVD) para realizar una selección sobre los N-body. Los algoritmos no necesitan preprocesamiento y también funcionan bien en modelos con topologías cambiantes. Más información sobre estos algoritmos se puede encontrar en la siguiente página web:

  http://gamma.cs.unc.edu/DVD/

- FCL: A Fast Collision Library and its integration with ROS for use with robotic systems (Pan, Chitta, and Manocha 2012). FCL es un proyecto de código abierto que incluye varias técnicas de detección de colisiones eficientes y cálculo de proximidad entre objetos. La página web donde se puede consultar más información y descargar el código fuentes es:

  http://gamma.cs.unc.edu/FCL/fcl_docs/webpage/generated/index.html

- GJK: algoritmo de la distancia de Gilbert–Johnson–Keerthi (Cameron 1997). El algoritmo de la distancia de Gilbert-Johnson-Keerthi se utiliza para determinar la distancia mínima entre dos conjuntos convexos. A diferencia

de muchos otros algoritmos que calculan la distancia, en este los datos no tienen que estar almacenados en un formato específico. El algoritmo se basa únicamente en una función que genera iterativamente simplices más cercanos a la respuesta correcta mediante la suma de Minkowski (CSO) de dos formas convexas. El código de este algoritmo se puede encontrar en la siguiente dirección:

http://web.comlab.ox.ac.uk/oucl/work/stephen.cameron/distances/index.html

- H-COLLIDE: A Framework for Fast and Accurate Collision Detection for Haptic Interaction (Gregory et al. 1999). Consiste en una serie de algoritmos y un sistema especializado de detección de colisiones rápidas y precisas que se utilizan en la interacción háptica con los objetos en un entorno virtual. Para cumplir los requisitos rigurosos de rendimiento de la interacción háptica, se utilizan las siguientes técnicas para calcular la colisión:

    - Descomposición espacial: se descompone el espacio en cuadrículas uniformes o células, como una tabla hash, para ocuparse eficientemente de requisitos de almacenamiento grande. En tiempo de ejecución, el algoritmo puede encontrar rápidamente la celda que contiene la ruta de barrido hacia fuera por la sonda.

    - Jerarquía de volúmenes envolventes basada en *OBBTrees*: un *OBBTree* es una jerarquía de volumen envolvente y cada nodo de la jerarquía corresponde a un cubo orientado a los ejes de coordenadas (OBB).

    - Coherencia entre frames: normalmente hay poco movimiento en la posición de los objetos entre pasos sucesivos. El algoritmo utiliza esta coherencia para mantener una caché con la información de contacto del paso anterior para realizar cálculos incrementales.

En la siguiente dirección de internet se puede encontrar más información de dichos algoritmos:

http://gamma.cs.unc.edu/H-COLLIDE/

- I-COLLIDE: An Interactive and Exact Collision Detection System for Large-scaled Environments (J. D. Cohen et al. 1994). I-COLLIDE es una librería que contiene una serie de utilidades que permiten calcular la detección de colisiones de manera interactiva y exacta para entornos de gran tamaño compuestos de poliedros convexos. Muchos poliedros no convexos pueden ser descompuestos en un conjunto de poliedros convexos, que luego pueden ser utilizados en esta librería. I-COLLIDE utiliza la coherencia espacial y las propiedades de convexidad para lograr una detección de colisiones muy rápida. En la siguiente página web se puede encontrar más información:

  http://gamma.cs.unc.edu/I-COLLIDE/

- IMMPACT: A System for Interactive Proximity Queries On Massive (Wilson et al., n.d.).

  Se describe un sistema de consultas de proximidad interactiva en modelos masivos que se componen de decenas de millones de primitivas geométricas. El conjunto de consultas incluye detección de colisiones, cálculo de distancia y verificación de la tolerancia. Estas son esenciales para la interacción con los modelos masivos. Se presenta un nuevo algoritmo usando gráficos de superposición para localizar las "regiones de interés" dentro de un modelo masivo, reduciendo requisitos de memoria de tiempo de ejecución. Para realizar consultas de proximidad interactiva utiliza las jerarquías de volúmenes envolventes y la coherencia espacial y temporal. En la siguiente dirección de internet se puede encontrar la librería y más información de la misma:

  http://gamma.cs.unc.edu/MMC/

- MCCD: A library for Multi-core Collision Detection (Tang, Manocha, and Tong 2009). Se presenta un algoritmo paralelo rápido para la detección de colisiones continuas (CCD) entre modelos deformables con procesadores multi-core. Utiliza una representación jerárquica para acelerar las consultas y presenta un algoritmo incremental que explota la coherencia temporal entre frames sucesivos. Los cálculos se distribuyen entre varios núcleos. Se presentan también unas técnicas eficientes para reducir el número de test elementales y utiliza un enfoque basado en la escalabilidad. En la siguiente dirección web se puede encontrar más información:

  http://gamma.cs.unc.edu/PCD/

- gProximity: Hierarchical GPU-based operations for collision and distance queries (Lauterbach, Mo, and Manocha 2010). Implementa un algoritmo paralelo que permite construir, actualizar y recorrer jerarquías de volúmenes envolvente utilizando para ello los núcleos de la GPU. El algoritmo sigue un enfoque general basándose en un esquema de distribución, que se puede adaptar fácil y dinámicamente a las cambiantes cargas de trabajo que suelen surgir en las aplicaciones interactivas. Esto permite aprovechar todos los núcleos para realizar operaciones jerárquicas usando ajustados OBBs en comparación con los AABBs. La estructura jerárquica que utiliza se crea rápidamente y se utiliza para la detección de la colisión, emplea una técnica de trazado de rayos. Esto permite trabajar con modelos complejos y deformables de manera interactiva para resolver el problema de la colisión. En la siguiente dirección se puede encontrar más información del algoritmo, así como el código fuente:

  http://gamma.cs.unc.edu/GPUCOL/

- PIVOT2D y PIVOT3D: Proximity information from Voronoi techniques (Hoff III et al. 2001)(Hoff, Zaferakis, and Lin 2002). Proporciona un nuevo enfoque para calcular la proximidad en objetos 2D, utilizando hardware gráfico. Implementa técnicas de procesamiento multi-paso. Estos

algoritmos presentan un enfoque unificado para realizar una variedad de consultas de proximidad entre objetos. No incluyen solamente la detección de colisiones, sino también el cálculo de las intersecciones, la distancia de separación, la profundidad de penetración y los puntos de contacto con sus normales. Utiliza una aplicación híbrida basada en geometría e imágenes, que equilibra el cálculo entre CPU y los subsistemas gráficos. Las técnicas geométricas de espacio-objeto localizan de forma aproximada las posibles regiones de intersección o características más cercanas entre dos objetos y las técnicas de espacio de imagen calculan la información de proximidad de bajo nivel en estas regiones localizadas. La mayor parte de la información de proximidad se obtiene del cálculo de la distancia, para lo cual se han usado hardware gráficos. Permite el cálculo de la respuesta a la colisión pudiéndose utilizar en simulaciones de dinámica de cuerpo rígidos y deformables. En la siguiente página web se puede encontrar más información y el código para el PIVOT2D:

http://gamma.cs.unc.edu/PIVOT/

- A Proximity Query Package (PQP): Fast proximity queries with swept sphere volumes (Gottschalk et al. 1996; Larsen et al. 1999). PQP proporciona soporte para cálculos de distancia, consultas de verificación de tolerancia y de detección de colisiones. Su API es similar a la de RAPID. Se puede aplicar a modelos poligonales generales y no necesita información topológica. Hace uso del volumen calculado por una esfera barrido para consultas de distancia entre objetos. Además, es flexible y al usuario utilizar más de un volumen envolvente para una consulta determinada. Más detalles y el código fuente están disponibles en la siguiente página web:

  http://gamma.cs.unc.edu/SSV/

- QuickCD: Efficient collision detection using bounding volume hierarchies of k-DOPs (Krishnan et al. 1998). El QuickCD es una librería donde se desarrolla y analiza un método, basado en las jerarquías del volumen

envolvente, que detecta eficientemente la colisión entre los objetos que se mueven en ambientes altamente complejos. El tipo de volumen envolvente que se utiliza es un politopo de orientación discreta (k-DOP), un politopo convexo cuyas caras están determinadas por semiespacios cuyas normales externas proceden de un pequeño conjunto fijo de k-orientaciones. Más información se puede encontrar en las siguientes direcciones de internet:

http://www.cosy.sbg.ac.at/~held/projects/collision/collision.html

http://www.computational-geometry.org/mailing-lists/compgeom-announce/1999-May/000224.html

- RAPID: Robust and Accurate Polygon Interference Detection (Gottschalk et al. 1996). RAPID es una librería de paquetes pequeña y fácil de usar. Funciona con sopas poligonales, que son sólo modelos poligonales que no requieren ninguna estructura topológica particular, como formar una malla o incluso un objeto cerrado. RAPID acepta una nube de triángulos desconectados como modelo. Dado dos modelos y su colocación dentro de un sistema de coordenadas mundial, RAPID devuelve una lista de pares de contactos entre triángulos, donde cada par de contactos es un triángulo tomado de cada modelo. Si la lista que devuelve es una lista vacía, los modelos no se tocan. Para procesar un par de modelos el programa cliente debe llamar explícitamente a un procedimiento de colisión, pasándole esos dos modelos y sus ubicaciones. Más información y el código fuente de esta librería se puede encontrar es la siguiente dirección de internet:

  http://gamma.cs.unc.edu/OBB/

- SOLID: Software Library for Interference Detection (Bergen 1997; Van Den Bergen 1999a; Van Den Bergen 2004). SOLID es una librería de código abierto que sirve para calcular la detección de colisiones entre objetos tridimensionales rígidos y deformables en movimiento. SOLID está diseñado para ser utilizado en aplicaciones interactivas de gráficos 3D y es

especialmente adecuado para la detección de colisiones de objetos y mundos descritos en VRML. Algunas de sus características son:

- Las formas de objeto están representadas por formas primitivas (caja, cono, cilindro, esfera) y complejos de politopos (segmentos de línea, polígonos convexos, poliedros convexos). Se puede utilizar una sola forma para instanciar varios objetos.
- El movimiento se especifica mediante traslaciones, rotaciones y escalas no uniformes del sistema de coordenadas local de cada objeto en movimiento.
- Las deformaciones de objetos complejos se pueden especificar usando matrices de vértices definidas por el usuario.
- La respuesta de colisión es definida por el usuario mediante la llamada a una función.
- Las llamadas de respuesta pueden utilizar datos de colisión que describen la configuración de un par de objetos que colisionan.
- La coherencia entre frames se explota manteniendo un conjunto de pares de objetos próximos (barrido incremental y poda de cajas delimitadoras alineadas con el eje) y colocando en caché los ejes de separación para estos pares.

El código fuente y más información de esta librería se puede encontrar en la siguiente página web:

http://solid.sourceforge.net/

- SELF-CCD: Continuous Collision Detection for deforming objects (Tang et al. 2009; Tang, Manocha, and Tong 2010). Esta es una librería para la detección colisiones entre objetos deformables. Se puede realizar la detección de colisiones inter e intra-objeto. Además, realiza una detección

de colisión continua entre las instancias discretas. Más información y el código fuente se puede consultar en la siguiente página web:

http://gamma.cs.unc.edu/SELFCD/

- SWIFT: Speedy Walking via Improved Feature Testing (S. A. Ehmann and Lin 2000). Esta es una librería para la detección de colisiones, cálculo de distancia y determinación de la zona de contacto, sobre objetos poligonales tridimensionales rígidos sometidos a movimiento (rotación y traslación). Se proporciona una API sencilla pero potente. Además, SWIFT es un banco de pruebas para la investigación activa en la detección de colisiones, por lo que es probable que se realicen mejoras en el futuro. SWIFT puede manejar modelos geométricos que cumplan las siguientes propiedades:

    - Cerrado. No hay límites.

    - Representación poliédrica. Los objetos deben estar compuestos por un conjunto de polígonos que describen el límite de un sólido en 3D.

    - Convexo o compuesto de piezas convexas. Los objetos deben ser convexos o ser un conjunto de piezas convexas (en una versión futura, esto puede ser relajado para permitir poliedros arbitrarios no convexos).

  SWIFT puede realizar cuatro tipos de consultas de proximidad:

    - Detección de intersección (colisión): detecta si dos objetos se cruzan (penetran).

    - Cálculo aproximado de la distancia: calcula la distancia mínima con un límite de error entre un par de objetos.

    - Cálculo de distancia exacta: calcula la distancia mínima entre un par de objetos.

  - Determinación de contactos: calcula las características más cercanas (vértice, borde, cara) de un par de objetos.

Más información y el código fuente de esta librería se pueden encontrar en la siguiente página web:

http://gamma.cs.unc.edu/SWIFT/

- SWIFT++: Speedy Walking via Improved Feature Testing for non-convex objects (S. a. Ehmann and Lin 2001). SWIFT ++ es un paquete de detección de colisiones capaz de detectar la intersección, realizar la verificación de tolerancia, calcular la distancia aproximada y exacta, o determinar los contactos entre pares de objetos en una escena compuesta por modelos poliédricos rígidos. Es una ampliación importante del sistema SWIFT. Ofrece mejoras sobre algoritmos e implementaciones anteriores. Proporciona una API sencilla pero potente. SWIFT ++ puede manejar modelos geométricos que cumplan:

  - Representación poliédrica. Los objetos deben estar compuestos por un conjunto de polígonos.

  - Cerrado o con Límite. Cualquier modelo poliédrico.

  - Forma general.

SWIFT ++ puede realizar cinco tipos de preguntas de proximidad:

  - Detección de intersección (colisión): detecta si dos objetos se cruzan (penetran).

  - Verificación de tolerancia: detecta si dos objetos están más cerca de una tolerancia dada.

- Cálculo de distancia aproximada: calcula la distancia mínima dentro de una aplicación especificada (relativa y / o absoluta) de error entre un par de objetos.

- Cálculo de la distancia exacta: calcula la distancia mínima entre un par de objetos.

- Determinación de contactos: calcula el conjunto de características más cercanas (vértice, borde, cara) para un par de objetos cuando están disjuntos.

Más información y el código fuente se puede encontrar en la siguiente página web:

http://gamma.cs.unc.edu/SWIFT++/

- V-CLIP: Collision detection algorithm for polyhedral objects (Mirtich and June 1997). El algoritmo Voronoi Clip, o V-Clip, es un algoritmo de detección de colisiones de bajo nivel para objetos poliédricos. Los principales objetivos en el diseño de esta librería fueron la robustez y la eficiencia. La implementación funciona bien incluso en presencia de degeneraciones geométricas. V-Clip calcula los puntos más cercanos entre objetos en un tiempo casi constante. La librería opera sobre objetos poliédricos que pueden ser no convexos o incluso desconectados. Devuelve los puntos más cercanos entre los objetos y las distancias entre ellos. Si los objetos penetran, devuelve la profundidad de penetración.

En la siguiente página web se puede descargar el código fuente y consultar más información:

http://www3.cs.stonybrook.edu/~algorith/implement/V-CLIP/implement.shtml

- V-COLLIDE: Accelerated collision detection for VRML (Hudson et al. 1997). Esta librería permite la detección de colisiones en grandes entornos. Está diseñada para operar con un gran número de objetos poligonales. No hace suposiciones sobre la estructura de entrada y trabaja con modelos arbitrarios, también conocidos como sopas de polígono. V-COLLIDE utiliza una arquitectura de detección de colisión en tres etapas:
    - Un test de los n-body para encontrar posibles pares de objetos que chocan.
    - Un test sobre la jerárquica de cajas envolventes orientadas (OBB) para encontrar posibles pares de triángulos que chocan.
    - Un test exacto que determina si un par de triángulos realmente se superponen.

  La rutina de n-body utiliza la coherencia espacial y temporal de las animaciones y simulaciones en movimiento. Las jerarquías de cajas envolventes orientadas (OBB) y las rutinas de colisión exactas utilizan las de la librería RAPID.

  Más información y el código fuente se pueden descargar se la siguiente dirección de internet:

  http://gamma.cs.unc.edu/V-COLLIDE/

- KCCD: Kinematic Continuous Collision Detection (Täubig and Frese 2012). Esta librería está diseñada para la detección continua de colisiones en tiempo real dentro del campo industrial y de la robótica humanoide. La librería esta implementada en C++ y utiliza SSCH para representar el volumen de los modelos con los que trabaja. Calcula volúmenes de barrido y distancias para todos los pares de elementos de un robot, y proporciona un conjunto de operaciones que permite un equilibrio entre la exactitud y el tiempo de cálculo.

http://www.dfki.de/cps/3d-collision-avoidance

2.6. Dispositivos hápticos

En los últimos tiempos, los dispositivos hardware que permiten emular entornos virtuales han sufrido una gran evolución, lo que permite que cada vez se puedan simular los entornos virtuales de una manera mucho más real. A pesar del grado de realismo que están alcanzado estos dispositivos, presentan una gran limitación y es que sólo pueden proporcionar información al usuario por dos canales: la vista y el oído. Debido a esto el usuario se puede ver inmerso dentro de un mundo virtual muy similar al real, pero presenta la limitación de que no puede interactuar con el entorno. Para paliar este problema surge una nueva disciplina que intenta añadir a este mundo virtual la percepción táctil, de forma que se puedan sentir los objetos como, por ejemplo, su superficie (textura), su peso, su temperatura, etc.

Por todo lo anterior el tacto es un sentido fundamental que, unido a la vista y al oído, posibilitan una simulación virtual mucho más realista, proporcionando al usuario la sensación de inmersión dentro del entorno. Por este motivo la industria ha comenzado a desarrollar en los últimos años una serie de dispositivos hápticos (“Tecnología Háptica,” n.d.), los cuales permiten estimular en el usuario la sensación táctil.

2.6.1. Interfaces hápticos

Teniendo en cuenta las definiciones anteriores de sentido del tacto y de háptico, se puede definir el término “Interfaces Hápticos” como el conjunto de dispositivos que permiten al usuario establecer una comunicación bidireccional en tiempo real entre el usuario y el medio virtual. Más concretamente se pueden definir los interfaces hápticos como aquellos dispositivos que permiten a una persona sentir, manipular o tocar objetos simulados en sistemas teleoperados y entornos virtuales, proporcionándole con ello sensación de inmersión y la posibilidad de interactuar con el medio virtual (Martínez et al. 2009).

2.6.2. Sistema háptico

La información que se percibe a través del sentido del tacto llega por tres canales diferenciados que son:

- Percepción táctil, es la que le llega a la persona a través de un estímulo producido en la piel.
- Percepción Kinestésica, es la que recibe la persona a través de los tendones y los músculos.
- Percepción háptica, se produce cuando se combinan las dos percepciones anteriores, siendo la manera habitual en la que una persona percibe los objetos de forma voluntaria y activa cuando utiliza el sentido del tacto.

Por todo lo anterior se puede definir un sistema háptico como aquel a través del cual a una persona adquiere información del entorno a partir de la percepción táctil y la percepción Kinestésica. Cualquier tarea que realiza una persona con sus manos lleva asociada la percepción de información por estos dos sentidos.

2.6.3. Percepción háptica en las personas

La percepción háptica es bidireccional ya que cuando una persona toca un objeto es capaz de percibir información sobre el mismo, así como de ejercer fuerza sobre este. Por esta razón para poder diseñar un dispositivo háptico lo más real posible hay que entender cómo una persona es capaz de sentir y ejercer dichos estímulos.

El sentido del tacto se localiza bajo la piel de las personas. Bajo la piel se encuentran una serie de receptores nerviosos, que son capaces de transformar una sucesión de estímulos que se producen sobre ellos, en información que llega al cerebro, el cual puede interpretar y tratar ("Tacto," n.d.).

El sentido kinestésico es el sentido que tienen las personas para detectar y localizar las posiciones y movimientos que realizan sus articulaciones y músculos. Gracias a este sentido una persona puede detectar la posición y el equilibrio de diferentes partes de su cuerpo (Katherin.memi, n.d.), permitiendo por ejemplo caminar, mover las manos, hablar, etc.

En la piel se encuentran una serie de receptores táctiles o corpúsculos, que se pueden agrupar en los siguientes apartados:

- *Corpúsculo de Meissner*. Permite detectar cambios en las texturas de la superficie de los objetos.
- *Corpúsculos de Krause*. Proporcionan la sensación de frio a bajas temperaturas.
- *Celdas de Merkel*. Permiten detectar presión, texturas y patrones en la superficie de los objetos.
- *Terminaciones de Ruffini*. Permiten detectar cambios de temperatura asociados a la sensación de calor.
- *Corpúsculos de Pacini*. Proporcionan la posibilidad de dar respuesta a presiones mecánicas y a vibraciones intensas.

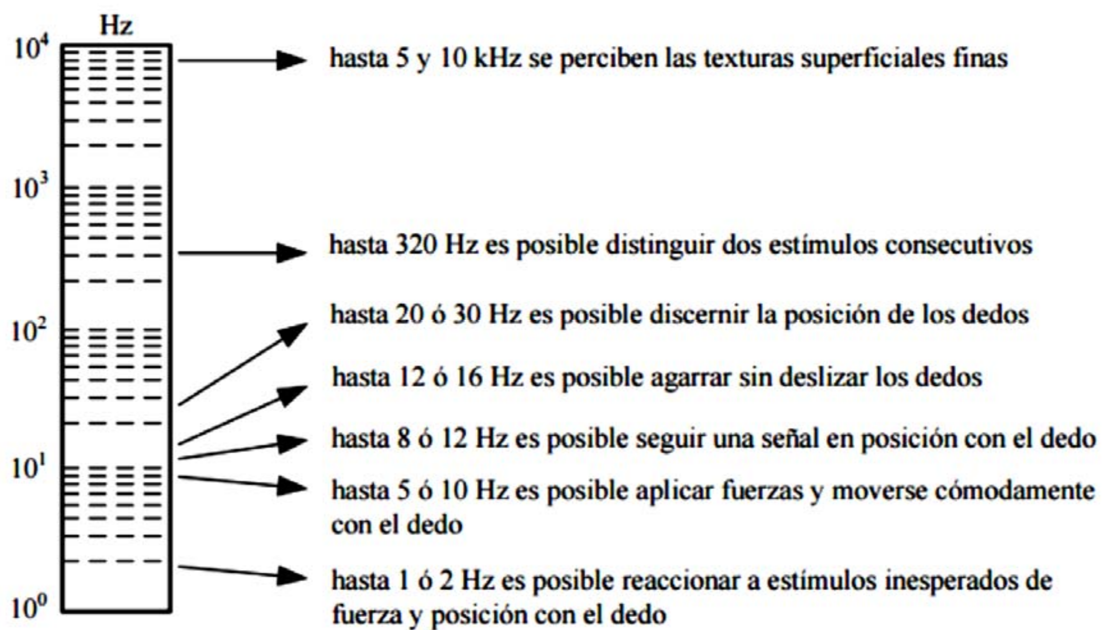


Figura 33. Clasificación frecuencial frente a los estímulos (Shimoga, n.d.).

Las personas reciben los estímulos táctiles y cinestéticos a una velocidad muy grande en comparación con la velocidad de respuesta que puede proporcionar a los mismos. Por esto Shimoga en 1992 (Shimoga, n.d.) realizó una clasificación frecuencial frente a los estímulos táctiles que se puede ver en la *Figura 33*.

Los fabricantes de dispositivos hápticos se basan en esta clasificación para diseñarlos dotándolos a estos dispositivos con una frecuencia de muestreo de, al menos, 1KHz, ya que con dicha frecuencia se garantizan que sus dispositivos pueden simular la gran mayoría de los estímulos táctiles.

2.6.4. Aspectos algorítmicos del renderizado háptico.

Se han desarrollado algoritmos para proporcionar renderizado háptico de objetos tridimensionales (3D) en entornos virtuales, es decir, simulados computacionalmente. El objetivo de la renderización háptica es generar visualizaciones táctiles de las formas, durezas, texturas superficiales y propiedades friccionales de los objetos 3D en tiempo real.

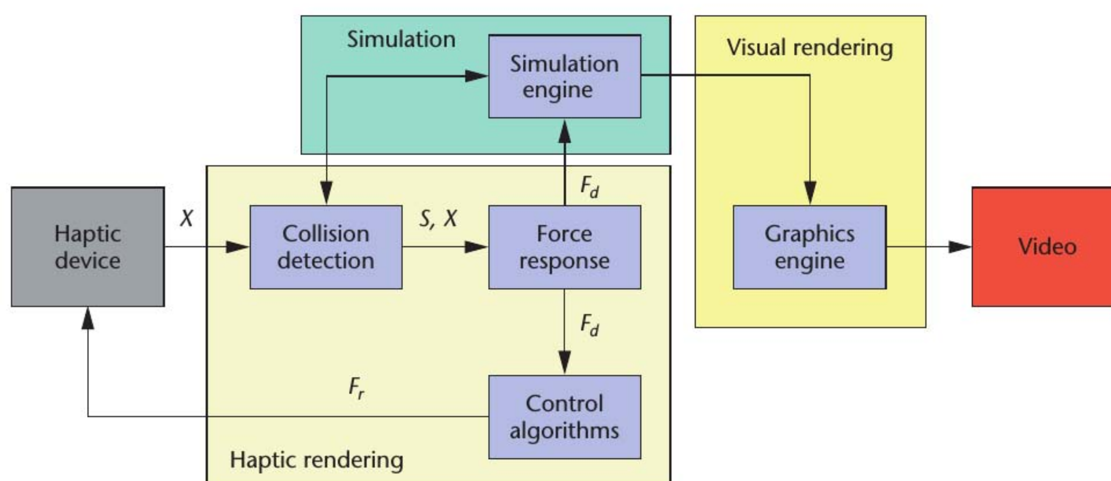


Figura 34. Esquema de la división en bloques del renderizado háptico y de la visualización (Salisbury, Conti, and Barbagli 2004).

Los algoritmos de renderizado háptico permiten a los usuarios tocar, sentir y manipular objetos 3D en entornos virtuales a través de dispositivos de retroalimentación de fuerza, también conocidos como interfaces hápticas. Típicamente un algoritmo de representación háptica consta de una etapa de detección de colisión y otra de respuesta a colisión (ver *Figura 34*). A medida que el usuario manipula el dispositivo de retroalimentación de fuerza (por ejemplo, una sonda en la punta del dedo) se adquieren la posición actual o reciente y la orientación de la sonda y el algoritmo de detección detecta colisiones entre la punta del dedo y

los objetos virtuales cercanos a su yema. Cuando esto se produce, el algoritmo de colisión-respuesta calcula las fuerzas de interacción entre la yema del dedo y los objetos virtuales y ordena al dispositivo de retroalimentación de fuerza para generar la representación táctica del objeto. La fricción del contacto dedo/objeto virtual, la textura del objeto y su dureza se pueden simular a través de perturbaciones espaciales y temporales apropiadas de la fuerza generada por el dispositivo de fuerza-retroalimentación.

Los algoritmos desarrollados para el renderizado háptico deben seguir un modelo cliente-servidor para proporcionar sincronización entre la visualización y la respuesta háptica con el fin de hacer que las tasas de actualización sean aceptablemente altas. Por ejemplo, mediante el uso de técnicas de multihilo, se pueden calcular las fuerzas de contacto a una velocidad de 1 kHz en un hilo mientras se actualizan las imágenes visuales a 30 Hz en otro hilo.

Los aspectos más relevantes que se deben de tener en cuenta a la hora de diseñar un algoritmo de renderizado háptico son:

- La frecuencia mínima del módulo del dispositivo de respuesta a la fuerza debe ser de 1 KHz.
- El módulo de detección de colisión debe proporcionar una respuesta exacta de dónde se ha producido esta.
- En la respuesta a la colisión se debe tener en cuenta la orientación del objeto.

En la *Figura 34* se observa que dentro de la fase de renderizado háptico se encuentra la etapa de detección de colisiones. Su funcionamiento es crucial dentro de los sistemas hápticos ya que debe proporcionar una respuesta lo más rápida posible para obtener una mejor sensación de realismo.

2.6.5. Principales dispositivos hápticos en el mercado

Como ya se comentó en los apartados anteriores, la inclusión de la percepción táctil a un entorno de realidad virtual hace que la sensación de inmersión sea mucho más realista e intuitiva. La comunicación en entornos de realidad virtual es unidireccional, ya que al usuario sólo le llega información visual o sonora, pero la

incorporación de dispositivos hápticos hace que la comunicación sea bidireccional (ver la *Figura 35*).

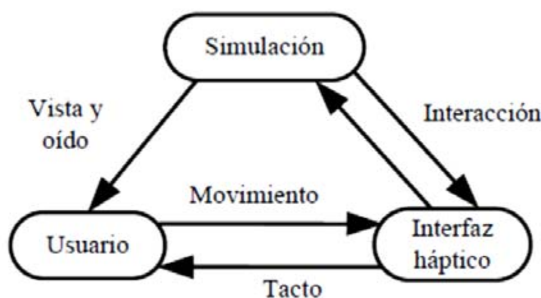


Figura 35. Sistema bidireccional de un dispositivo háptico.

Los dispositivos hápticos se pueden clasificar en función del tipo de respuesta que proporcionen o en función de su finalidad. A continuación, se presenta una clasificación de los dispositivos hápticos atendiendo a la funcionalidad.

2.6.6. Clasificación de los dispositivos hápticos atendiendo a la funcionalidad de los mismos

Si se hace un estudio de los principales dispositivos hápticos que se pueden encontrar en el mercado se observa que estos se pueden clasificar en dos grandes grupos:

- *Dispositivos que trabajan con un puntero tipo ratón o lápiz*. Permiten a la persona que los maneja tener sentido de las texturas sobre los objetos en 2D con los que trabaja.

- *Los dispositivos tipo lápiz (stylus) o guante*. Permiten a la persona que los utiliza poder interactuar con objetos virtuales, escaneados en tres dimensiones, pudiendo manipularlos, desplazándolos o rotándolos por la escena, proporcionándole la sensación de como si los tocara.

En la *Figura 36* se pueden ver algunos ejemplos de estos dispositivos.

Como parte del objetivo de esta tesis es realizar una simulación con una aplicación informática que utilice la estructura de datos y los algoritmos propuestos en un dispositivo háptico, dentro de todos los del mercado, nos hemos centrado en estudiar los de tipo lápiz.

2.6.6.1. Dispositivos hápticos de tipo lápiz (stylus).

Estos dispositivos proporcionan a la persona que los utiliza la posibilidad de poder interactuar con objetos virtuales escaneados en tres dimensiones, pudiendo desplazar o rotar objetos que se encuentren dentro de la escena, proporcionándole la sensación de como si se estuvieran tocando dichos objetos. Con los dispositivos tipo lápiz, la persona va a poder recorrer el contorno del objeto como si se pasase un puntero por encina de este.

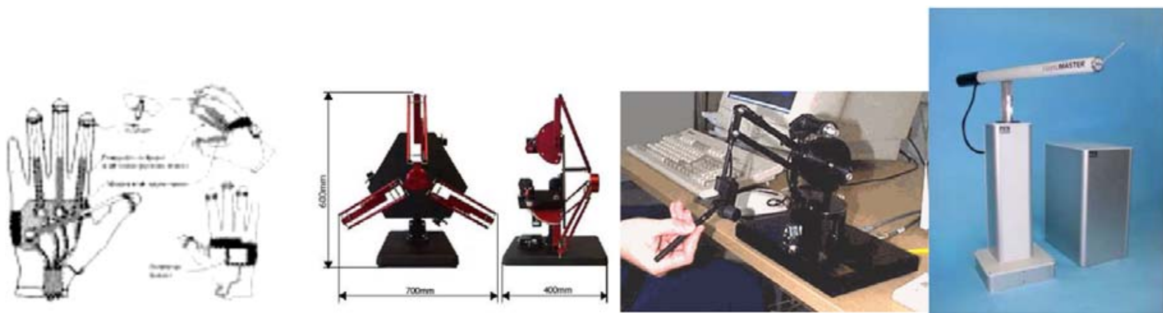


Figura 36. Distintos dispositivos clasificados según su funcionalidad.

Dentro de este grupo se pueden encontrar en el mercado los PHANToM ("Geomagic®," n.d.) fabricados por la empresa Geomagic. Son los dispositivos hápticos más populares. Su interface proporciona una realimentación muy buena. Esta empresa construye toda una familia de brazos hápticos que se clasifican en las siguientes familias: los Touch 3D Stylus, los Geomagic® Touch™ (anteriormente conocidos como Sensable Omni), los Geomagic® Touch™ X (anteriormente conocidos como Sensable Phantom Desktop) y la familia de los modelos 3D Systems Phantom Premium™. Cada uno de estos dispositivos se ha diseñado para cubrir una serie de requisitos y la amplia gama de modelos varía según el tamaño del espacio de trabajo, fuerza, número de fuerzas, el rango de movimiento, la posición y precisión de los sensores, la inercia y la fidelidad.

- **3D Touch™ Haptic 3D Stylus** (*Figura 37*): Permite esculpir con el lápiz

modelos en 3D. Presenta una forma muy intuitiva y natural de interactuar con modelos en 3D. En lugar de utilizar un cursor que se mueva en el espacio 2D, utiliza un conjunto de herramientas que permiten esculpir en la pantalla. Estas herramientas se mueven con la mano a través del lápiz táctil que incorpora. Las características más destacadas de estos dispositivos son: tiene seis grados de libertad, presenta fuerza de realimentación lo que permite al usuario sentir lo que se toca en la pantalla, interfaz USB, altura del movimiento de 7 pulgadas (17,8 cm) y un diámetro de movimiento de 5.5 pulgadas (14cm).

Figura 37. 3D Touch™ Haptic 3D Stylus

- **Geomagic® Touch™ Haptic Device** (anteriormente conocido como PHANToM OMNi). Estos dispositivos hápticos presentan un diseño sencillo o básico, con lo cual la funcionalidad es también reducida. El tamaño de los mismos no es muy grande, lo que permite que sean portables y compactos. Su instalación suele ser fácil, ya que están pensados bajo la filosofía de plug and play. Por su diseño, el espacio de trabajo que proporcionan es reducido, emulando el movimiento de una muñeca. (*Figura 38*).

Figura 38. Geomagic® Touch™ Haptic Device.

- **Geomagic® Touch™ X Haptic Device** (anteriormente conocido como PHANToM Desktop). Este dispositivo háptico es una solución de escritorio asequible y sofisticada. Sus características más destacadas son: proporciona una entrada de posicionamiento con cierta precisión, la salida es una respuesta de fuerza de alta fidelidad y por último, presenta unas reacciones de fuerza mayores y de baja fricción. (*Figura 39*).

Figura 39. Geomagic® Touch™ X Haptic Device

- **Phantom Premium Haptic Devices** (anteriormente conocido como PHANToM Premium). Pretende simular el brazo hasta el codo, con lo cual proporciona un espacio de trabajo medio. El precio de este dispositivo es más alto que los que se han mencionado en los apartados anteriores pero tiene la ventaja de que presenta una mayor calidad. La realimentación de fuerza que proporciona permite trabajar con tres grados de libertad. Existe

una versión que multiplica por cinco la fuerza máxima. (*Figura 40*).

Figura 40. Phantom Premium Haptic Devices

- **3D Systems Phantom® Premium™ 6DOF** (anteriormente conocidos como PHANToM 6DOF). Estos, a diferencia de los anteriores, presentan realimentación de fuerza en seis grados de libertad. También permiten simular realimentación de momentos de fuerzas. Emulan un brazo completo desde el hombro hasta los dedos. Su coste y su calidad son altos. El espacio de trabajo en el que pueden funcionar es mucho mayor que cualquiera de los otros dispositivos vistos anteriormente. (*Figura 41*).

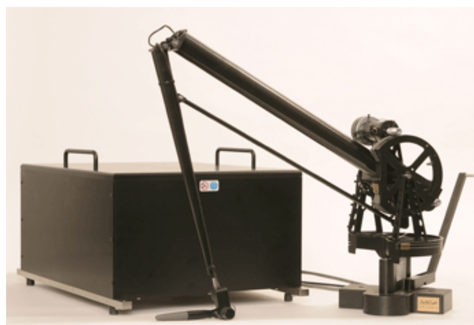


Figura 41. 3D Systems Phantom® Premium™ 6DOF

De todos estos dispositivos, para realizar la aplicación que utiliza la estructura de datos y los algoritmos propuestos en esta tesis, se ha seleccionado el Geomagic® Touch™ Haptic Device.

3. LA ESTRUCTURA EBP-OCTREE

En este capítulo se presenta un esquema de representación de sólidos: el EBP-Octree, que está diseñado para poder trabajar con modelos de datos formados por varias decenas de polígonos. Se define esta estructura de datos y se especifican los pasos que se tienen que dar para construir el EBO-Octree a partir de los datos de un modelo.

Se define la estructura de representación de sólidos Extended BP-Octree (EBP-Octree) (Aguilera, Feito, and Melero 2013), la cual cumple todas las propiedades, posibilitando representar sólidos de gran tamaño formados por varias decenas de millones de polígonos.

El EBP-Octree parte de la información de la frontera de un sólido B-Rep para montar una estructura jerárquica de envolventes convexas sobre el sólido que quiere representar. Para ello se basa en la estructura propuesta por Melero, el BP-Octree (Melero 2008), que trabaja con modelos pequeños formados por varios millones de polígonos. El EBP-Octree utiliza una estructura con forma de árbol, la cual permite tener en cada nodo intermedio como máximo ocho nodos hijos, un Octree. Los nodos hoja o nodos terminales del EBP-Octree recogen información de la frontera del sólido. Los nodos intermedios almacenan una secuencia de envolventes convexas, que son construidas a partir de la información guardada de la frontera del sólido en

los nodos hoja.

Al igual que en la estructura BP-Octree en el diseño y construcción de la estructura EBP-Octree, se han tenido en cuenta una serie de aspectos para su correcto funcionamiento que garanticen que las soluciones obtenidas a partir de ella son soluciones completas y correctas. Estos aspectos son:

- El cálculo de las envolventes se realiza haciendo un recorrido en el octree de abajo hacia arriba, garantizando que el volumen de la envolvente calculada en un nivel n del octree contiene al volumen de las envolventes generadas en el nivel $n+1$. Como es lógico, las envolventes de niveles inferiores del EBP-Octree tienen que aproximarse más a la frontera del sólido. Para garantizar esta propiedad, el cálculo de la envolvente en los nodos hoja se hace utilizando los polígonos reales de la superficie del sólido. Para el cálculo de las envolventes de los nodos interiores se utiliza la envolvente calculada para los nodos hijos, es decir los que están un nivel inferior en el octree.
- Garantizar que toda la información geométrica contenida en un nodo se utiliza para calcular la envolvente del mismo. Para ello se tiene que hacer es clasificar todos los polígonos que forman la frontera del sólido en los nodos hoja del EBP-Octree. Como cada polígono puede atravesar varios nodos hoja del Octree se utiliza el algoritmo 3DDDA (Three-dimensional digital difference analyzer), que recorre todos los polígonos que forman la frontera del sólido, calculando para cada uno los nodos hoja que corta o atraviesa.
- Determinación del exterior y del interior de sólidos cerrados y 2-variedad. Para determinar qué nodos del octree son interiores y cuáles son exteriores al sólido se parte de los nodos hoja, que son los que están en la frontera del sólido. Se utiliza la posición de las esquinas (dentro o fuera del sólido) y se hace un recorrido de abajo hacia arriba, propagando sus valores.

3.1. Fundamentos del EBP-Octree

Los EBP-Octree se basan en los BP-Octree los a su vez se fundamentan en la idea utilizada en los BSP-Trees (Fuchs, Kedem, and Naylor 1980; Thibault and Naylor 1987; Naylor, Amanatides, and Thibault 1990). Estos últimos son una estructura de

datos jerárquica que particionan recursivamente el espacio de la escena o los objetos en dos partes que, a partir de un plano, obteniendo celdas convexas. Por contra los EBP-Octree particionan el espacio o el objeto en ocho partes, utilizando como estructura para almacenar la información un octree, de manera que se puedan utilizar como índices espaciales.

Las deficiencias fundamentales de los BP-Octree son:

- Sólo pueden trabajar con modelos cuyo número de polígonos no supere los dos millones.
- Trabajan con una longitud de palabra de memoria de 32 Bits, lo cual limita el número máximo de niveles del árbol a once.
- No tiene definidos algoritmos que calculen la detección de colisiones entre modelos.

Las características nuevas que aportan los EBP-Octree son:

- Trabajan con una longitud de palabra de memoria de 64 Bits, lo cual permite tener árboles con hasta 20 niveles de profundidad.
- Optimizan el algoritmo 3DDDA en el cálculo de los nodos que son cortados por un polígono.
- Proporcionan diferentes métodos para la selección de planos relevantes en la creación de las envolventes.
- Calculan el EBP-Octree *out-of-core*, lo que permite construir solamente una vez el EBP-Octree en ordenadores que tengan un tamaño de memoria limitado, almacenándose en disco según se va calculando.
- Calculan progresivamente las envolventes, almacenándolas en disco de forma gradual conforme se van obteniendo.
- Calculan previamente cuántos niveles del EBP-Octree se tienen que mantener en memoria principal, antes de la carga de este en memoria principal. El número de niveles dependerá de la memoria disponible del ordenador en el que se está trabajando.
- Utilizan memoria caché para optimizar las operaciones de cálculos de inclusión punto en sólido y de distancias de un punto a un modelo.

- Incorporan la implementación de algoritmos para el cálculo de la colisión entre modelos.

La información necesaria para la construcción del EBP-Octree es diferente a la que se necesita para cargarlo en memoria. Esto da lugar a dos tipos diferentes de EBP-Octree: el EBP-Octree de construcción y el EBP-Octree definitivo.

Los EBP-Octree de construcción van a tener sólo cuatro tipos de nodos:

- **Nodo blanco**, cuando el nodo está totalmente fuera del sólido.
- **Nodo negro**, cuando el nodo está por completo dentro del sólido.
- **Nodo gris**, cuando el nodo tiene una parte dentro y otra parte fuera del sólido.
- **Nodo hoja**. Son nodos terminales del árbol que son cortados por la frontera del sólido.

Los EBP-Octree definitivos además de los cuatro tipos de nodos anteriores utilizan un tipo de nodo más:

- **Nodo gris separador**. Pertenecen todos a un mismo nivel del Octree, el cual se selecciona previamente a la carga en memoria principal del EBP-Octree, y es el que se utiliza de límite en el EBP-Octree entre los nodos que están en memoria principal y los que están guardados en disco (ver *Figura 42*).

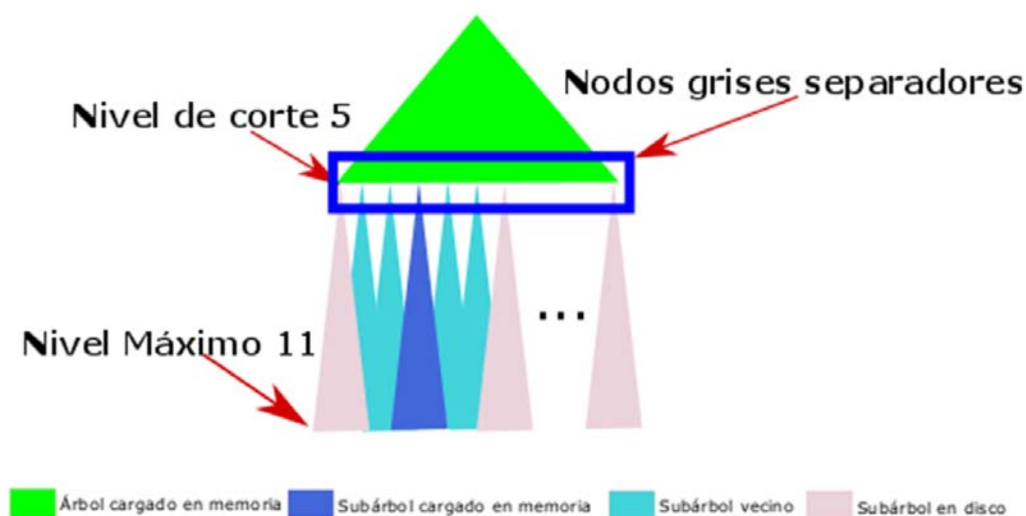


Figura 42. Posición de los nodos grises separadores en el EBP-Octree.

Los nodos *blancos* y *negros* no se tienen que almacenar en la estructura, ya que no aportan ninguna información relevante para el calculo de la inclusión punto en sólido o el de la colisión, pues sólo se utilizan para la visualización del modelo. Para los nodos *grises*, *grises separadores* y *hoja* se guardan una serie de planos cuyos semiespacios interiores son utilizados para intersecarlos con el volumen del nodo, obteniéndose un volumen convexo que engloba a la parte del sólido contenido en ese nodo.

Además, para los nodos *grises separadores* se necesita almacenar un vector de tamaño ocho en el que se guardan las posiciones del archivo donde se encuentran almacenados los subárboles de cada uno de los hijos. En la *Figura 43* se puede ver un nodo *hoja* sobre el que se ha calculado la envolvente convexa.

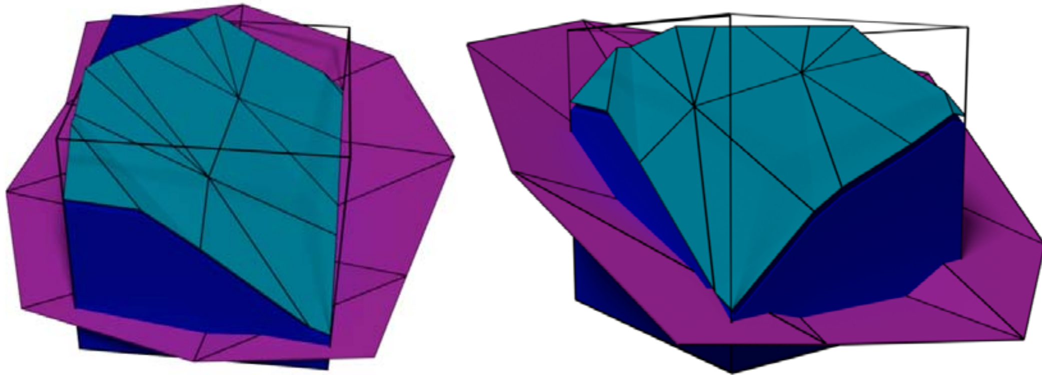


Figura 43. Nodo hoja. Los triángulos rosas se corresponden con los polígonos del sólido, los polígonos de color cian forman la envolvente convexa calculada y los de azul determinan los polígonos añadidos de las paredes del nodo.

En los nodos *hoja* también se tiene que guardar una lista de referencias a los polígonos de la frontera del sólido que cortan o atraviesan a cada nodo, algo similar a como se hace en los Polytree (Carlbom, Chakravarty, and Vanderschel 1985).

Al igual que se hace en los SP-Octrees (Cano, Torres, and Velasco 2003), los planos utilizados para calcular las convexidades de los nodos se extraen de los planos de los polígonos que forman el sólido B-Rep.

Los EBP-Octree utilizan tres vectores para guarda la información de los planos seleccionados para crear su envolvente, el esquema gráfico de esta estructura se puede ver en *Figura 44*.

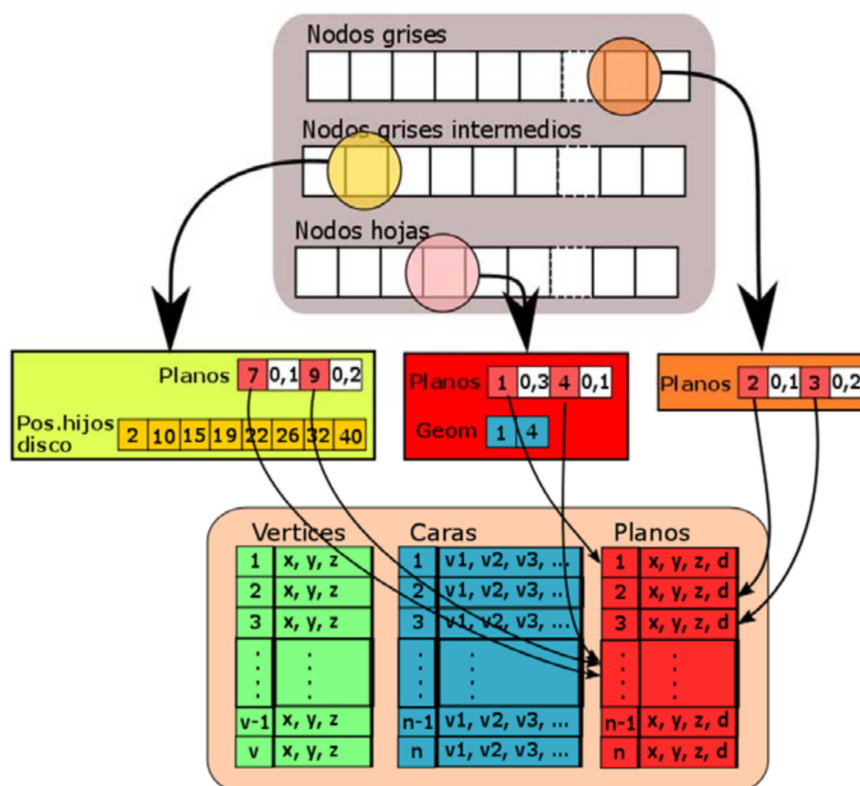


Figura 44. Esquema gráfico de la estructura de un EBP-Octree.

Cada nodo tiene que guardar una lista de índices de planos así como el desplazamiento realizado para poder ser utilizado en el cálculo de la envolvente convexa del nodo.

Además de los datos anteriores, los nodos grises separadores tienen que guardar un vector de tamaño ocho (uno para cada hijo del nodo) en el que se almacena la posición en el archivo donde se encuentra guardada la rama del subárbol del hijo correspondiente.

Los nodos *hoja,* además de los datos de los nodos *grises,* deben guardar una lista de índices a los planos que forman la geometría del sólido y que cortan o atraviesan dicho nodo.

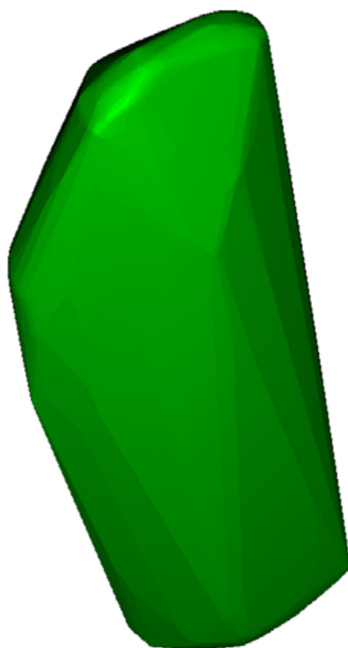
El algoritmo utilizado para construir el EBP-Octree consta de las siguientes etapas:

- Elección del modo de selección de los planos que van a formar la envolvente de los nodos y el porcentaje de planos del octree seleccionados entre niveles.

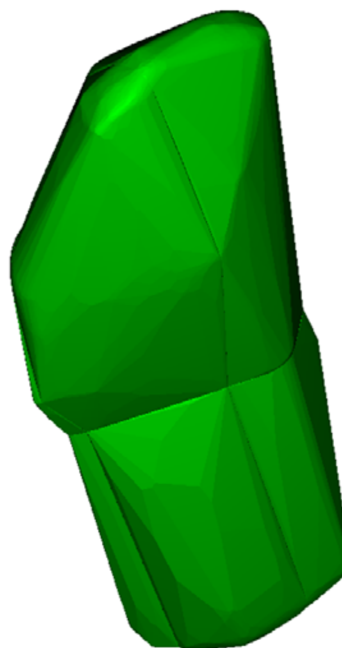
- Cálculo de la caja envolvente del objeto.
- Cálculo del número de niveles que va a tener el octree y el tamaño del lado que tendrá el nodo en cada uno de ellos.
- Indexación de los polígonos que forman la frontera del modelo en un grid 3D. La resolución de este grid vendrá dada por el número de niveles que tenga el octree.
- Construcción del octree. Para ello, en primer lugar se tendrán que calcular los nodos hoja. Estos nodos se obtienen teniendo en cuenta las celdas del grid anterior que son cortadas o atravesadas por algún polinomio de los que forman la frontera del objeto. Partiendo de los nodos hoja se van creando los nodos grises de arriba hacia abajo.
- Creación de las envolventes convexas de los nodos hoja. Se parte del cubo del nodo y se recorta con los planos seleccionados de los polígonos que lo atraviesan.
- Creación de las envolventes convexas de los nodos grises. Para ello se toman los planos seleccionados para calcular la envolvente de los nodos hijos, seleccionando un subconjunto de estos. Este proceso comienza con los nodos hoja calculando las envolventes de abajo hacia arriba.
- Almacenamiento del EBP-Octree en una serie de archivos para que pueda ser recuperado en cualquier momento.

En la *Figura 45* se puede observar un ejemplo de la jerarquía de volúmenes envolventes que se crean en un EBP-Octree para un modelo concreto: el de la Amazona Herida.

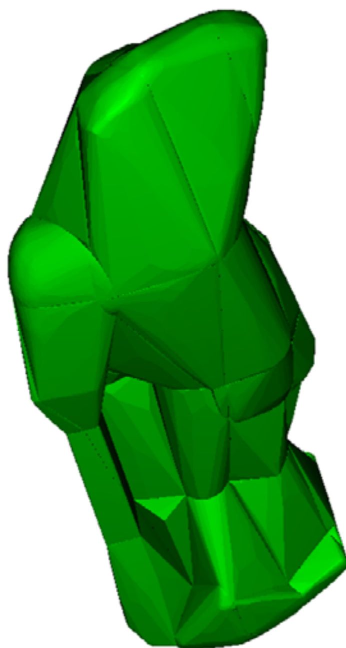
a) Real.



b) Nivel 1.

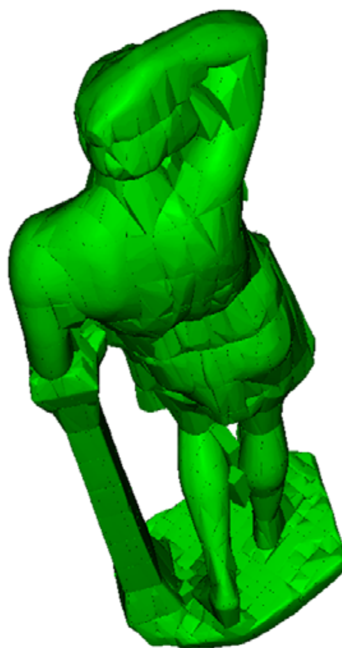


c) Nivel 2.

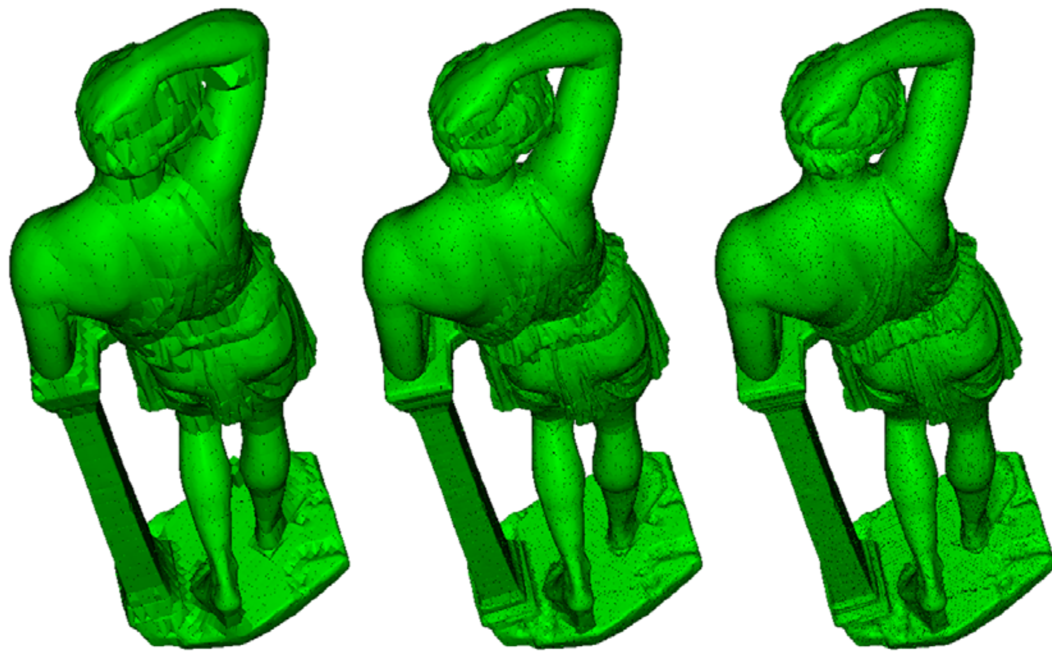


d) Nivel 3.

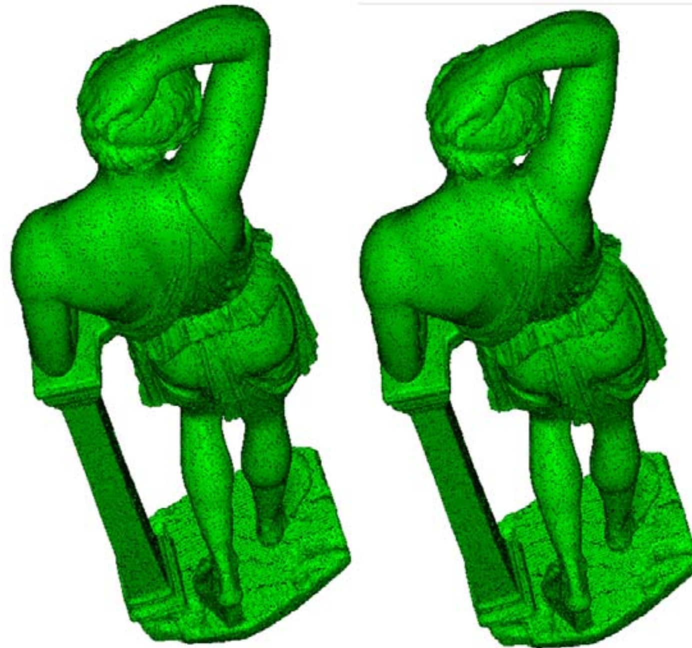
e) Nivel 4.



f) Nivel 5.



g) Nivel 6. h) Nivel 7. i) Nivel 8.



j) Nivel 9. k) Nivel 10.

Figura 45. Ejemplo de las envolventes generadas para los primeros 10 niveles del modelo de la Amazona Herida de 28 millones de polígonos.

A continuación se detallan cada uno de los pasos expuestos anteriormente para la construcción del EBP-Octree.

3.2. Cálculos preliminares para la construcción del EBP-Octree.

Antes de construir el EBP-Octree se necesitan conocer una serie de datos sobre el modelo, como son el tamaño de la caja que lo envuelve y la longitud media de los lados de los polígonos que los forman. A partir de ellos se calculan el número de niveles que va a tener el EBP-Octree, así como el tamaño de los lados de todos los nodos que forman parte de él. En las subsecciones siguientes se detalla su proceso de obtención.

3.2.1. Cálculo de la caja envolvente del modelo.

En primer lugar lo que se hace es calcular la caja envolvente del modelo que se encuentra alineada a los ejes de coordenadas (Axis-Aligned Bounding Box (AABB), (Bergen 1997)). La caja envolvente se representa con dos puntos significativos, el máximo y el mínimo (ver *Figura 46*):

$$CajaEnvolvente = (PuntoMin(X_{min}, Y_{min}, Z_{min}), PuntoMax(X_{max}, Y_{max}, Z_{max}));$$

El cálculo de la caja envolvente se realiza recorriendo el vector donde se almacenan todos los vértices de los polígonos que forman la frontera del sólido. En este recorrido se van calculando, para cada una de las coordenadas X, Y y Z, los valores mínimos y máximos.

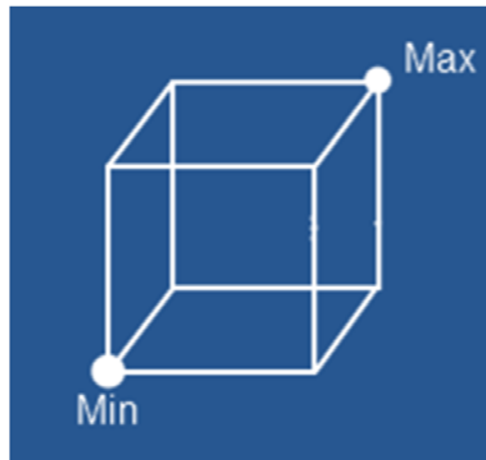


Figura 46. Puntos significativos para representar una AABB.

3.2.2. Creación del índice espacial.

La creación del índice espacial se puede realizar siguiendo una de las dos estrategias clásicas para la construcción de árboles: la *bottom-up* o la *top-down*. En nuestro caso se ha elegido la primera de ellas, es decir la de abajo hacia arriba. Para ello lo primero que se tiene que hacer es calcular todos los nodos hoja. Este proceso comienza calculando una rejilla espacial en 3D en la que están incluidos todos los polígonos que forman la frontera de sólido. El tamaño de esta rejilla viene dado por el número máximo de niveles con los que se desea calcular el octree, que se corresponde con el tamaño de sus nodos del último nivel. Partiendo de la rejilla anterior se calculan todos los nodos que son cortados o atravesados por los polígonos del sólido. Estos nodos se corresponden con los nodos hoja del octree. Una vez identificados todos los nodos hoja ya se pueden calcular todos los nodos intermedios del octree. Este proceso comienza calculando, para cada nodo hoja, un valor en octal (el *octcode*) el cual proporciona el camino que se tiene que seguir desde el nodo raíz hasta el nodo en cuestión. Siguiendo este camino de abajo hacia arriba se van creando de forma progresiva todos los nodos intermedios.

Debido a la naturaleza del proceso de construcción del octree descrito en el párrafo anterior, la primera tarea se tiene que realizar es la indexación de todos los polígonos que forman el sólido, asignando estos a todos los nodos hoja que atraviesan o cortan. Este proceso se realiza utilizando el *octcode* de cada uno de los polígonos. El *octcode*, se calcula como un código Morton tradicional (Morton 1966).

Para poder calcular la celda de la rejilla a un nivel determinado donde cae un punto se utilizar la siguiente fórmula:

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

donde N_d es la coordenada discreta, l es el nivel de profundidad, d la dimensión, w_d es la longitud del nodo raíz de dimensión d, d_{min} es el valor mínimo de la coordenada correspondiente a la dimensión d y p_d es el valor de la coordenada d del punto p.

La fórmula anterior tiene un inconveniente cuando los puntos que se desean clasificar están en el borde máximo de la caja envolvente, ya que son asignados al nodo vecino siguiente, que está fuera de la caja. Esto se produce por tener intervalos abiertos por la derecha. Gráficamente el problema, para dos dimensiones, se puede ver en la *Figura 47*, donde el punto V_3, que está en la parte de la izquierda, se clasifica bien, pero el punto V_1 que está en la frontera derecha se clasifica en el nodo vecino que está fuera de la caja envolvente del sólido.

Este problema se puede solucionar utilizando diferentes estrategias como:

- Filtrar los puntos que se clasifiquen en el voxel vecino fuera de la frontera, asignándolos al nodo vecino por la izquierda, el cual si está dentro.
- Ampliar un poco el tamaño del nodo raíz.

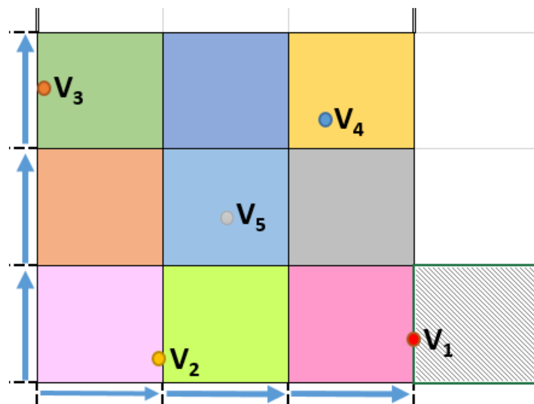


Figura 47. Clasificación de puntos en nodos.

En este trabajo se ha utilizado la segunda de las estrategias.

3.2.3. Código Morton.

Cada uno de los nodos que se incluyen en la estructura del EBP-Octree se identifica con un número entero largo llamado *octcode*, el cual depende de la posición que ocupe el nodo en el espacio. Este valor es único para cada nodo y se calcula siguiendo una codificación de Morton (Morton 1966). El *octcode* sirve para poder saber cuál es el camino que se debe seguir por el árbol para alcanzar un nodo determinado desde el nodo raíz. Los dígitos del *octcode* se leen en octal, indicando el número del nodo hijo al que hay que desplazarse para alcanzar al nodo buscado. Esta codificación en octal es parecida a la utilizada por Gargantini en (Gargantini 1982).

El código Morton se utiliza para calcular los *octcodes* de los puntos que se quieren clasificar, es decir, para ver en qué nodo del octree están.

Este código entrelaza el valor de las coordenadas discretas de un punto en binario, obteniéndose de esta forma un valor único para cada punto. Este valor se corresponde con el nodo en el que está incluido.

El intercalado del valor en binario de las coordenadas discretas consiste en combinar ordenadamente los bits de cada coordenada. Por ejemplo, si tenemos un punto en el espacio entonces se tienen tres coordenadas X, Y y Z y el entrelazado lo que hace es tomar el bit más significativo de la coordenada X, a continuación el bit más significativo de la coordenada Y, después el más significativo de la coordenada Z, a continuación se entrelazan en el mismo orden de coordenadas indicado los segundos bit más significativos. Este proceso se repite hasta incluir todos los bits de las tres coordenadas.

El número que se obtiene es único para cada uno de los nodos de un nivel del octree, pero se repite en los diferentes niveles.

Para solucionar el problema de que dos nodos tengan el mismo *octcode* estando en diferentes niveles se tiene que añadir al *octcode* información del nivel al que pertenece el nodo. Para ello se pueden seguir alguno de estos dos criterios:

- La información del nivel se añade al *octcode* por la parte de la izquierda (*Figura 48.a*), lo cual produce que los nodos del octree se guarden por niveles, empezando por los del primer nivel y acabando por los nodos hoja. Los nodos se guardan siguiendo un orden en anchura.
- Si la información se añade al *octcode* por la derecha (*Figura 48.b*), produce

que los nodos del octree se guarden siguiendo un orden en profundidad.

Nivel	Código Nortom

Nivel	Nodo	Valor Binario	Valor Decimal
1	0	01 000000	64
1	1	01 000001	65
2	0	10 000000	128
2	1	10 000001	129

a) Orden en anchura

Código Morton	Nivel

Nivel	Nodo	Valor Binario	Valor Decimal
1	0	000000 01	1
2	0	000000 10	2
1	1	000001 01	5
2	1	000001 10	6

b) Orden en profundidad

Figura 48. Código Morton con información del nivel.

Al añadir al código Morton la información del nivel, se transforma en un localizador del nodo, ya que con este número se pueden conocer las coordenadas exactas que ocupa un nodo dentro de la caja envolvente. En la *Figura 49* se puede ver un ejemplo de código Morton en el que a los nodos del primer nivel de un octree se les ha añadido la información del nivel por la izquierda.

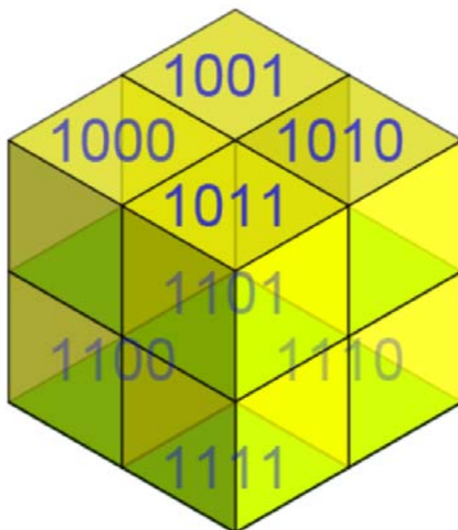


Figura 49. Ejemplo de código Morton con información del nivel a la izquierda.

La opción elegida ha sido la de orden en anchura, ya que la estructura propuesta necesita trabajar con nodos de un mismo nivel, posibilitando esta ordenación el almacenamiento y la transmisión de los nodos del octree por niveles.

3.2.4. Cálculo del número de niveles del EBP-Octree.

Tal y como se ha visto en el apartado anterior, el código Morton para un octree necesita tres bits por nivel más los bits en los que se guarda la información del nivel al que pertenece el nodo. El número de bits que se necesitan para codificar en binario la información del nivel viene dado por la fórmula:

$$Número\ bits\ nivel = log_2(profundidad_nivel_máximo)$$

Ecuación 2. Numero de bits necesarios para codificar hasta un nivel.

Como los modelos con los que se va a trabajar son modelos muy grandes se busca poder tener un octree con el mayor número de niveles. Para maximizar el número de niveles del octree la estructura se ha diseñado para trabajar con procesadores cuya longitud de palabra sea de 64 bits. De esta manera el *octcode* se implementa como un número entero largo que se almacena en memoria principal utilizando 64 bits. Con un *octcode* de 64 bits se podrían codificar octrees de veintiún niveles, pero como ya se ha comentado se tienen que quitar los bits con la información del nivel al que pertenece el nodo. Así, si se aplica la *Ecuación 2* para veintiún niveles se necesitan cinco bits. Al quitar al *octcode* los bits con la información del nivel sólo quedan cincuenta y nueve bits para codificar los niveles. Con este número de bits se pueden codificar octrees con diecinueve niveles, ya que 19 * 3 = 57 bits más los 5 del nivel hacen un total de 62 bits. Como el nodo raíz es la caja envolvente del modelo no hace falta que se incluya en el *octcode* ya que a él se accede directamente. Por todo lo anterior, con un *octcode* de 64 bits se podrían codificar localizadores de nodos con un **nivel máximo de veinte**. En la *Figura 50* se puede apreciar gráficamente cómo se distribuyen los bits en un *octcode*. En color verde los bits del nivel, en amarillo dos bits que no se utilizan y en azul los bits donde se almacena la información del *octcode*. Los dos bits que se han quedado libres se podrán utilizar más adelante para codificar cualquier información de los nodos.

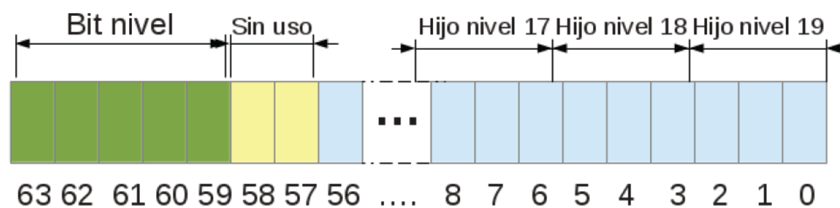


Figura 50. Octcode de 64 bit.

El número de niveles con los que se va a calcular el EBP-Octree es un dato fundamental, ya que de él va a depender la resolución máxima con la que se va a calcular la estructura jerárquica de volúmenes envolventes. Un octree de 20 niveles permite tener modelos cuyo número máximo de nodos hoja sea de 8^{19}, siendo estos más que suficientes para representar modelos formados por varias decenas de millones de polígonos con una muy buena calidad o resolución.

Como se ha visto anteriormente el tamaño máximo de niveles que puede tener el octree utilizando 64 bits para guardar los datos es de 20. No siempre es necesario calcular el EBP-Octree con el tamaño máximo de niveles, ya que esto implica, por un lado, un tiempo de cálculo muy grande y por otro que se tenga que utilizar mucha memoria para poder almacenar el EBP-Octree, obteniéndose una resolución tan grande que gráficamente no pueda ser apreciada por el ojo humano. Por esto se hace necesario utilizar un método de cálculo que se adapte al modelo, por ejemplo, comparando la resolución o el tamaño de los polígonos que forman la frontera del sólido con el tamaño de la caja que lo envuelve. Utilizando este criterio se consigue que la obtención del nivel máximo del octree se adapte al modelo sobre el que se desea calcular el EBP-Octree.

Para el cálculo del nivel máximo del EBP-Octree se han planteado varios métodos con distintas estrategias y se han comparado los resultados obtenidos. Estas estrategias son:

1. La que toma el lado del polígono mayor y lo compara con el lado de la caja envolvente mayor.
2. La que coge el lado del polígono menor y lo compara con el lado de la caja envolvente menor.
3. La que toma la media mayor de los lados de los polígonos y lo compara con el lado mayor de la caja envolvente.
4. La que calcula la media mayor de los lados de los polígonos y lo compara con el lado menor de la caja envolvente.
5. La que toma la media menor de los lados de los polígonos y lo compara con el lado mayor de la caja envolvente.
6. La que calcula la media menor de los lados de los polígonos y lo compara con el lado menor de la caja envolvente.

7. La que toma la media de los tamaños de los lados de los polígonos que forman la frontera del sólido y la compara con el lado de mayor tamaño de la caja envolvente del modelo.

Todas estas técnicas producen resultados muy similares en el cálculo del nivel máximo del octree, aunque la que mejores resultados proporciona es la propuesta en último lugar.

El nivel máximo del EBP-Octree se obtiene como el número de veces que se puede dividir entre dos el lado mayor de la caja envolvente y cuyo valor sea mayor que el tamaño medio de los lados de los polígonos que forman la frontera del sólido (ver *Ecuación 3*). El procedimiento que calcula el nivel máximo del octree se puede ver en el *Algoritmo 1*.

$$L = \max(X_{max} - X_{min}, Y_{max} - Y_{min}, Z_{max} - Z_{min},)$$

$$a = avg\left(longitud_{arista_i}\right) / i = 0..n_{aristas}$$

$$\boldsymbol{Nivel\ m}\text{á}\boldsymbol{ximo} = \min i \in (1..20) / \frac{L}{2^i} < a$$

Ecuación 3. Calculo del nivel máximo del EBP-Octree.

```
void calculaNivelMaximo() {
    double suma = 0.0,  media = 0.0;
    for (int i = 0, fin = this->flist.size(); i < fin; i++) {
        suma=distancia(this->vlist.at(this->flist.at(i).verts[0]),
                           this->vlist.at(this->flist.at(i).verts
                          [this->flist.at(i).nverts - 1]));
        for (int j = 0; j<this->flist.at(i).nverts - 1; ++j) {
            suma+=distancia(this->vlist.at(this->flist.at(i).verts[j]),
                            this->vlist.at(this->flist.at(i).verts[j + 1]));
        }
        suma /= this->flist.at(i).nverts;
        media += suma;
    }
    media /= this->flist.size();
    int nivel = 0;
    double lado_ma = this->calculaMediaLadoCuboEnvolvente(this->bbox);
    while (lado_ma >= media) {
        nivel++;
        lado_ma /= 2;
    }
    this->MAX_NIVEL = nivel;
}
```

Algoritmo 1. Procedimiento para calcular el número de niveles del octree.

3.3. Cálculo de los nodos hoja.

La construcción del EBP-Octree se hace de abajo hacia arriba. Debido a esto en primer lugar se tienen que calcular los nodos hoja. Para ello se recorren todos los polígonos que forman la frontera del sólido obteniendo todos los nodos hoja que son cortados, incluyendo en una lista, para cada uno de ellos, el *octcode* del nodo hoja y el número de polígono. La lista resultante se ordena por *octcode* y a partir de ella se crean los nodos hoja, uno por cada *octcode* diferente. En cada nodo hoja se incluye una lista con todos los polígonos que lo cortan. Por último se calculan las envolventes de cada uno de estos nodos, seleccionando un porcentaje de los planos que incluyen a los polígonos que cortan al nodo hoja. En las siguientes subsecciones se detalla este proceso.

3.3.1. Clasificación de los polígonos en nodos hoja.

Un paso fundamental para la construcción del EBP-Octree es el cálculo de los nodos hoja del octree que son cortados o atravesados por los polígonos que forman la frontera del sólido.

Los polígonos que forman la frontera del sólido pueden tener tamaño muy dispar. Por este motivo se pueden tener polígonos que estén por completo dentro de un nodo hoja y otros polígonos que sean tan grandes que atraviesen varios nodos hoja del octree.

Antes de comenzar el proceso de indexación de los polígonos se calcula el *octcode* al máximo nivel para cada uno de los vértices que forman parte de la frontera del modelo. Estos *octcodes* se almacenan en un vector, ordenados de igual forma que el vector de vértices. De esta forma se evita tener que calcular para cada polígono que lo incluya el *octcode* del vértice.

El cálculo de todos los nodos hoja que son atravesados o cortados por un polígono se realiza siguiendo los siguientes pasos:

1. Se toma un polígono y para cada uno de los vértices que lo forman se calcula el *octcode* al nivel máximo del octree.
2. Se calcula el nodo padre común en el que se incluyen los vértices del polígono.

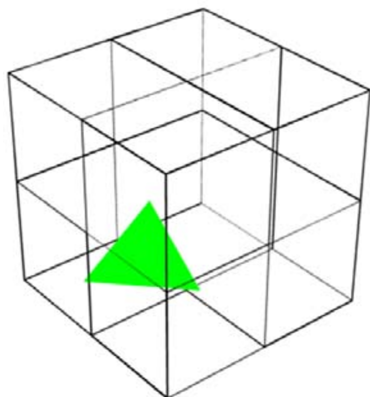
3. Se van comprobando los nodos hijos del nodo común para ver si son cortados por el polígono. Los nodos cortados son incluidos en una lista.
4. Se toman de la lista los nodos del siguiente nivel, comprobando si los nodos hijos son cortados por el polígono. En este caso se incluyen en la lista.
5. Este proceso se repite hasta que se llega al nivel máximo del octree.
6. Cada nodo hoja atravesado o cortado por un polígono se almacena en una lista en forma de tupla con dos valores: <*octcode* del nodo hoja, identificador del polígono >.

El siguiente algoritmo calcula el nodo padre común a todos los vértices del polígono (*Algoritmo* 2).

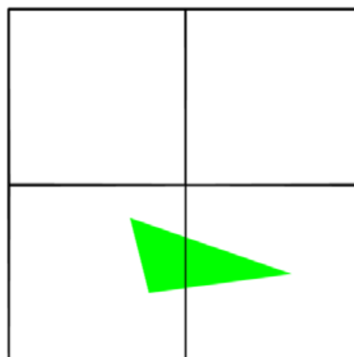
```
int desplaza;
desplaza = MAX_DEPTH - MAX_NIVEL;
face_oct.hi = fi;
face_oct.octcode  =  vérticesSPI->at(this->flist.at(fi).verts[hi]).octcode  >>
(desplaza * 3);
++hi;
while (hi < nv) {
   oct = vérticesSPI->at(this->flist.at(fi).verts[hi]).octcode >> (desplaza *
3);
   if(face_oct.octcode^oct){// Si es distinto con XOR (face_oct.octcode!=oct){
      face_oct.octcode = face_oct.octcode >> 3;
      desplaza++;
   } else ++hi;
}
//Se le pone el nivel adecuado de la caja que engloba a todos los puntos del
//polígono
if (desplaza) {
  //Se desplaza el octcode hasta su posición inicial.
  face_oct.octcode = face_oct.octcode << (desplaza * 3);
  //Se quita los bits del nivel
  face_oct.octcode = face_oct.octcode & OCTCODE_MASK;
  //Se le pone el nivel de la caja que engloba a todos los puntos del polígono
  face_oct.octcode = face_oct.octcode | levelBits[MAX_DEPTH - desplaza];
}
```

Algoritmo 2. Cálculo del nodo padre común que engloba un polígono.

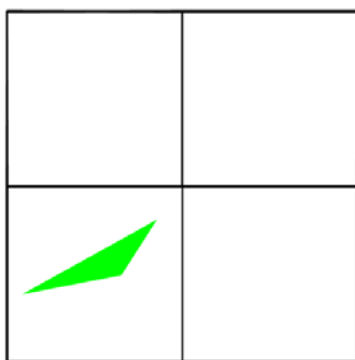
Gráficamente el proceso para calcular el nodo padre que contiene a todos los vértices de un polígono se puede ver en la *Figura 51*. En la *Figura 52* se puede ver un ejemplo de cómo se calculan todos los nodos hoja que son cortados por un polígono.



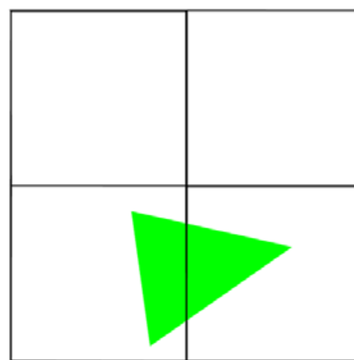
a) Polígono corta dos nodos hoja



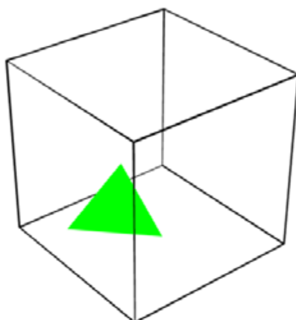
b) Vista planta



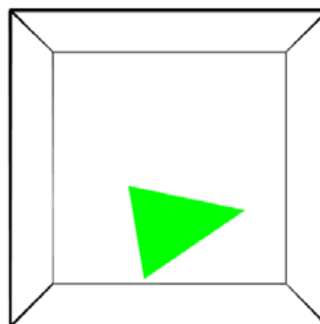
c) Vista alzado



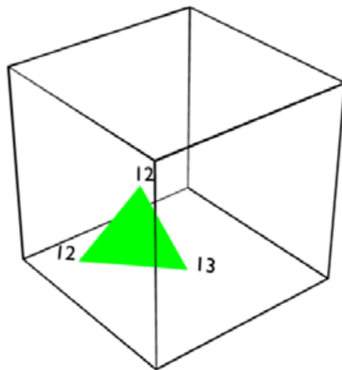
d) Vista perfil



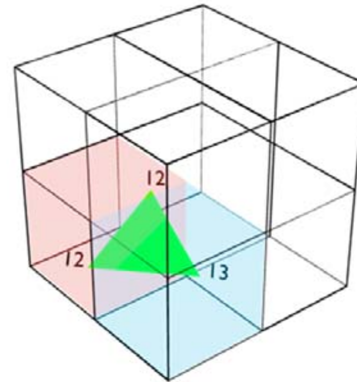
e) Nodo padre incluye polígono



f) Nodo padre vista planta



g) Octcode de los vértices polígonos



h) Nodos hoja 2 y 3 cortados polígono

Figura 51. Proceso de calculo de nodos hoja cortados por un polígono.

Polígono 1	Octcode	
Vértices	**Nivel**	**hijos nivel**
V1	3	132
V2	3	133
V3	3	132

a) Octcodes de los vértices que forman el polígono 1.

Octcode nodo padre	
Nivel	***Hijos nivel***
2	*13*

b) Octcode del nodo padre que engloba al polígono 1.

Octcode Nodos hoja		*Lista de Polígonos que cortan al nodo hoja*
Nivel	***hijos nivel***	
3	*132*	*1*
3	*133*	*1*

c) Lista de nodos hoja que son cortados por el polígono 1.

Figura 52. Ejemplo del proceso de cálculo de todos los nodos hoja que son cortados por un polígono.

Para saber si un polígono corta o atraviesa un cubo se ha utilizado el teorema de los ejes separados (SAT) optimizado por Möller (Akenine-Möller 2001) para la detección de la intersección entre un cubo y un triángulo. El algoritmo que calcula todos los nodos hoja que son cortados por un polígono es el siguiente (*Algoritmo 3*).

```
 std::vector<OCTCODE_T> nodosCorte;
 OCTCODE_T octCode;
 Coord boxcenter[3], triverts[3][3];
 nodosCorte.push_back(face_oct.octcode);
 for (int j = 0; j < 3; ++j) {
     triverts[j][0] = vlist.at(this->flist.at(fi).verts[j]).x();
     triverts[j][1] = vlist.at(this->flist.at(fi).verts[j]).y();
     triverts[j][2] = vlist.at(this->flist.at(fi).verts[j]).z();
 }
 //Se mira hijo por hijo y el que se corta al polígono lo añado a una lista.
 //Se recorre esta lista hasta que quede vacía.
 //Para ver si el polígono corta al nodo se utiliza el teorema de los ejes
separados.
 while (nodosCorte.size() != 0) {
     oct = nodosCorte.front();
     nodosCorte.erase(nodosCorte.begin());
     int nivel = oct>>BITS_CODIGO;
     if (nivel<this->MAX_NIVEL) {
         nivel++; //Ya que los hijos tienen un nivel más
         oct = oct >> (MAX_DEPTH - nivel)*3;
         for (int i = 0; i < 8; i++) {
             octCode = oct | i;
             octCode = octCode << (MAX_DEPTH - nivel)*3;
             octCode = octCode& OCTCODE_MASK;
             octCode = octCode | levelBits[nivel];
             MyBBox curBBox = this->getBBox(octCode, dmin, f[nivel]);
             boxcenter[0] = (curBBox.max.x() + curBBox.min.x()) / 2;
             boxcenter[1] = (curBBox.max.y() + curBBox.min.y()) / 2;
             boxcenter[2] = (curBBox.max.z() + curBBox.min.z()) / 2;
             //triBoxOverLap=1 Entonces se superponen el triangulo y el box
             //triBoxOverLab=0 Entonces no se superponen el triangulo y el box
             if (triBoxOverlap(boxcenter, boxhalfsize[nivel], triverts)) {
                 nodosCorte.push_back(octCode);
             }
         }
     } else {
         face_oct.octcode = oct;
         _tr->push_back(face_oct);
     }
 }
```

Algoritmo 3. Cálculo de los nodos hoja cortados por un polígono.

Una vez que se han recorrido todos los polígonos que forman la frontera del sólido se tiene una lista de tuplas con los *octcodes* y el identificador del polígono que lo corta.

Esta lista se utiliza para generar los nodos hoja.

3.3.2. Gestión out of core de la lista de nodos hoja.

Como puede ser muy grande y ocupar mucho espacio en memoria, para su generación se utiliza una estrategia basada en la técnica de Divide y Vencerás con gestión *out-of-core* en memoria externa.

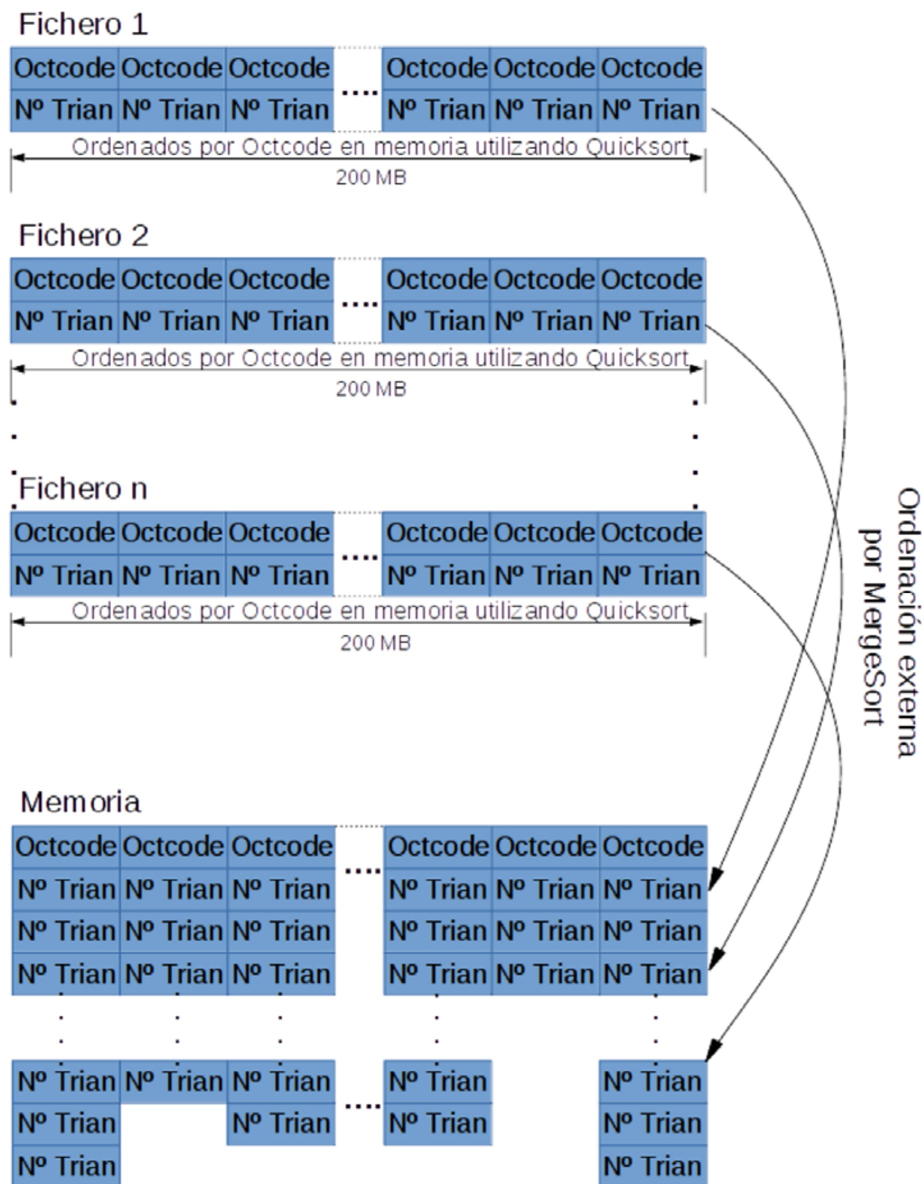


Figura 53. Estructura de archivos temporales para la indexación espacial.

Dicha estrategia consiste en ir añadiendo tuplas a la lista hasta que alcanza un tamaño máximo indicado. Dicho valor se obtiene en función del tamaño de la memoria principal de la máquina en la que se está generando el EBP-Octree. Una vez alcanzado el tamaño máximo la lista se ordena por *octcode*, ya que esta se genera ordenada por identificador de polígono. Este proceso se repite para todos los polígonos que forman parte de la frontera del sólido, almacenado todas las listas en diferentes archivos que se van numerando correlativamente.

El proceso anterior genera una serie de archivos que están formados por una lista de tuplas ordenadas por *octcode*. Estos archivos se ordenan utilizando un algoritmo clásico de ordenación externa por mezcla (*Mergesort*), obteniéndose una única lista en memoria que contiene en cada celda un *octcode* y una lista de identificadores de polígonos que cortan a ese nodo. Todo este proceso se puede ver en la *Figura 53* para un tamaño máximo de memoria de 200 MB.

3.3.3. Construcción de los nodos hoja.

Los nodos hoja se construyen a partir de la lista comentada en el apartado anterior. Como en cada una de las celdas de la lista se tiene un *octcode* y una lista de identificadores de polígonos, para cada una de las celdas de la lista se crea un nodo hoja que ocupa un volumen dentro de la caja envolvente del modelo que viene dada por su *octcode*.

Además, en cada uno de los nodos hoja del *octcode* se añade la lista con todos los identificadores de polígonos que cortan o cruzan el nodo, la cual ya está en la lista de la celda. Adicionalmente a cada nodo hoja se le añaden dos coordenadas que dan la posición y el tamaño de la caja del nodo. Para guardar la información de la caja sólo se necesitan sus coordenadas máxima y mínima. Este último dato, que sólo se utiliza en la fase de creación, se almacena con el objetivo de agilizar los cálculos en una etapa posterior.

Los nodos hoja creados se almacenan en un vector en memoria principal.

3.3.4. Creación de las envolventes convexas de los nodos hoja.

Para la creación de las envolventes convexas de los nodos hoja se sigue un proceso similar al descrito en los BP-Octree (Melero 2008). En este trabajo se realizan una serie

de mejoras en algunos de los pasos con el objetivo de poder trabajar con modelos formados por varias decenas de millones de polígonos. Las etapas que se modifican son:

3.3.4.1. Cálculo de puntos de corte sobre las aristas de la caja envolvente del nodo hoja.

Si una de las caras de un nodo hoja es cortada o atravesada por uno de los polígonos que forman parte de la frontera del sólido implica que esa cara atraviesa también un nodo hoja vecino al primero. Los nodos hoja vecinos comparten vértices y aristas entre sí. De esta forma si una arista es cortada por un polígono de la frontera implica que los cuatro nodos hoja vecinos que comparten esta arista se encuentran dentro de la lista de nodos hoja creados. Teniendo en cuenta todo lo anterior, para optimizar el cálculo de los puntos de corte sobre las aristas de las cajas envolventes de los nodos hoja, las aristas se han indexado para sólo tener que realizar los cálculos una única vez sobre cada una de ellas. En la *Figura 54* se puede observar una vista superior de dos nodos vecinos que comparten una arista que es cortada por dos polígonos de la superficie del sólido. Como se puede apreciar los puntos de corte son comunes para todos los nodos que comparten esa arista.

Figura 54. Arista común a dos nodos. Sólo se calculan los puntos de corte una vez.

Dado el ingente número de nodos en último nivel, para poder indexar las aristas de las cajas de los nodos se han dividido en tres grupos: las aristas de la coordenada X, las de la coordenada Y y las de la Z. En la *Figura 55* se pueden ver marcados con colores los tres grupos de aristas de la caja.

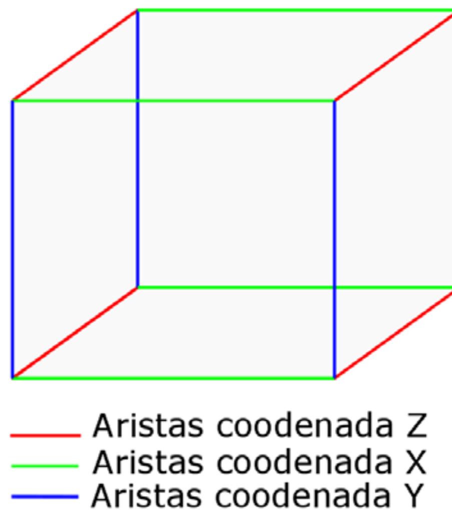


Figura 55. Distribución de las aristas en tres grupos.

El cálculo de los puntos de corte sobre las aristas de la caja se puede realizar de forma independiente para cada uno de los grupos. El proceso para calcular los puntos de corte sobre una arista de la caja se realiza siguiendo los pasos descritos a continuación:

- En primer lugar se toman todas las cajas envolventes de los nodos hoja y se crean tres listas de aristas, una para cada grupo. En estas listas se almacena el vértice inicial, el vértice final y el identificador del nodo hoja por el que se ha insertado. Los valores de los vértices se toman directamente del nodo hoja ya que estos se han calculado y almacenado en la etapa anterior. El vértice inicial y final se introducen, para todas las aristas, en el mismo orden. Por ejemplo, para una arista del grupo de coordenadas Y el vértice inicial es el del plano inferior de la caja y el final el del superior. Este orden es necesario para poder aplicar el siguiente paso.

- Una vez creadas las tres listas con todas las aristas de las cajas de los nodos hoja se ordenan por la posición de los vértices iniciales y, si estos coinciden, entonces por la posición de los vértices finales. Esta ordenación obtiene una lista de aristas comunes agrupadas una tras otra, diferenciándose sólo en el indicador del nodo hoja.

- Con las aristas ordenadas ya se puede empezar a calcular los puntos de corte sobre la arista. Para ello se toma la primera arista de una de las listas y se calculan dichos puntos tomando los polígonos que atraviesan el nodo hoja

al que referencia. Para calcular si una arista es cortada por un polígono se utiliza el algoritmo de Möller (Möller and Trumbore 2005) el cual calcula la intersección de un rayo sobre un triángulo de manera óptima. En cada celda de la nueva lista a la vez que se van guardando los puntos de corte se van calculando y almacenando la referencia al punto de corte mayor en esa arista y una referencia al polígono con el que se ha obtenido ese punto mayor.

- Una vez que se ha calculado todos los puntos de corte se añade la referencia a la posición en la lista de la arista en el nodo hoja.
- Se toma la siguiente arista de la lista ordenada y se compara con la última a la que se le han calculado los puntos de corte. Si el vértice inicial coincide entonces sólo hay que añadir la referencia de última arista calculada en el nuevo nodo hoja. Para comparar dos aristas sólo hay que tener en cuenta el vértice inicial, ya que todas las aristas tienen el mismo tamaño por ser de cajas envolventes de nodos que están al mismo nivel. Este paso se repite mientras la arista siguiente coincida con la última arista calculada. Si la arista no coincide con la última calculada entonces se repiten los dos pasos anteriores.
- El proceso se repite mientras queden aristas de un grupo. Cuando se han chequeado todas las aristas de un grupo se libera la memoria ocupada por la lista de aristas y se chequean las del siguiente grupo.

El seudocódigo donde se recoge todo este proceso se puede ver en el Algoritmo 4.

3.3.4.2. Puntos de corte sobre las caras de la caja envolvente de los nodos hoja con los polígonos que forman la frontera del sólido.

Una vez que se han calculado todos los puntos de corte sobre las aristas de la caja envolvente de los nodos hoja, puede darse el caso de que se tengan nodos hoja que son atravesados por los polígonos de la frontera del sólido y que ninguna de sus aristas sea cortada. Se puede ver un ejemplo de un nodo hoja de este tipo en la *Figura 56*.

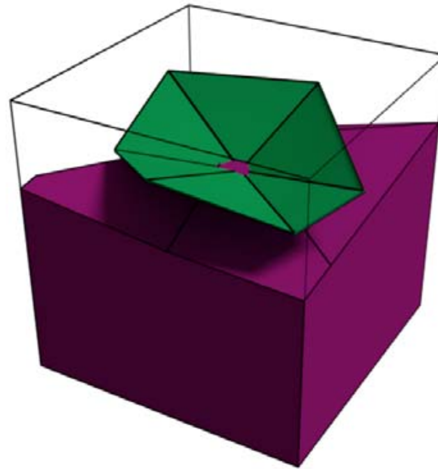


Figura 56. Nodo hoja cuyas aristas no son cortadas por los polígonos de la frontera del sólido.

```
for (int i = 0; i < 3; i++) {
   //Se crea la lista de aristas para una de las coordenadas del cubo
   for (Index idx = 0; idx<this->hojas.size(); idx++) {
            for (int k = i; k < 12; k = k + 3) {
                IdxAC = nuevaAristaOrde2(aristasOrde);
                aristasOrde.at(IdxAC).nodoHoja = idx;
            }
   }
   //Se ordenan las aristas por el valor del primer vértice.
   sort(aristasOrde.begin(), aristasOrde.end(), comparaArista2);
   for (j = 0; j < aristasOrde.size() - 1; j++) {
     /*calcula los puntos de corte de cada arista con los triángulos que cortan
      el nodo hoja*/
     calPuntosCorte(aristasOrde, j);
     //Se saltan las aristas que son iguales por no tener que calcularlas
     while(comparaPuntos(aristasOrde.at(j).R.P0, aristasOrde.at(j + 1).R.P0))
     {
        j=j+1;
     }
  }
}
```

Algoritmo 4. Cálculo puntos de corte de las aristas de los nodos hoja.

Para este tipo de nodos el proceso de cálculo de los puntos de corte de los polígonos sobre las caras de la caja es similar al proceso anterior aunque ahora se trabaja con caras en vez de con aristas. Uno de los aspectos que cambia es que ahora para calcular los puntos de corte sobre las caras aunque se utiliza el mismo método que en el caso anterior (la intersección rayo triángulo de Möller (Möller and Trumbore

2005)) ahora las caras de la caja se dividen en dos triángulos y como rayos se toman las aristas de los polígonos.

3.3.4.3. Elección del modo de selección de los planos relevantes para el cálculo de las envolventes convexas.

El cálculo de las envolventes convexas para todos los nodos del octree se realiza a partir de un porcentaje de los planos de los polígonos más relevantes que cortan o atraviesan un nodo. Con el objetivo de simplificar las envolventes resultantes para cada nodo y así acelerar los cálculos de inclusión punto en sólido y de detección de colisiones entre dos modelos, se selecciona un porcentaje de estos planos. Si se tuvieran en cuenta todos se generarían envolventes más próximas al volumen del modelo, aumentando considerablemente el tiempo de cálculo y la complejidad de las mismas. Además si el objeto tiene mucha superficie extremadamente convexa la envolvente se ajusta muy bien al objeto, pero no se consigue el objetivo del EBP-Octree que es el de simplificar la geometría del objeto. Al seleccionar un porcentaje de planos, los resultados obtenidos, tanto visuales como de volúmenes de las envolventes, son muy parecidos a si se seleccionan todos los planos, pero con un tiempo de cálculo considerablemente menor.

Además del porcentaje de planos seleccionados se debe elegir el modo por el cual se van a seleccionar unos planos envolventes frente a otros. Para ello se tienen en cuenta los siguientes criterios:

- Menor offset. En este modo se seleccionan los planos que menos se han tenido que desplazar a la hora de calcular la envolvente de un nodo.
- De forma aleatoria. Aquí los planos seleccionados se toman al azar. Para ello se genera un número aleatorio con la posición del primer plano y a partir de aquí se escogen tantos planos como indica el porcentaje.
- Agrupando en clúster (K-mediana). En este modo los planos se agrupan en clústeres dependiendo de su normal y se elige para cada clúster la mediana de las normales de todos los planos que lo forman.

Las dos primeras formas de seleccionar los planos relevantes para el cálculo de la envolvente son triviales, ya que en una sólo hay que ordenar los planos por desplazamiento seleccionando un porcentaje de los que menos desplazamiento tengan y en la segunda se escogen de manera aleatoria. La tercera forma es un poco más compleja por esto se explica de manera más detallada a continuación.

3.3.4.3.1. Método de selección de planos relevantes K-mediana.

Este método permite seleccionar de manera eficiente los planos relevantes para el cálculo de la envolvente eliminando redundancias de los mismos, agrupando los planos en función de su similitud. El criterio de similitud que se utiliza es el de la normal del plano. De esta manera, si se tiene una superficie casi llana (ver *Figura 57*), todos los planos que la forman tienen la normal muy parecida y todos ellos se pueden sustituir por uno solo. Si la superficie que se tiene es muy curva, por ejemplo, una esfera, los planos que se seleccionan están distribuidos homogéneamente por toda la superficie.

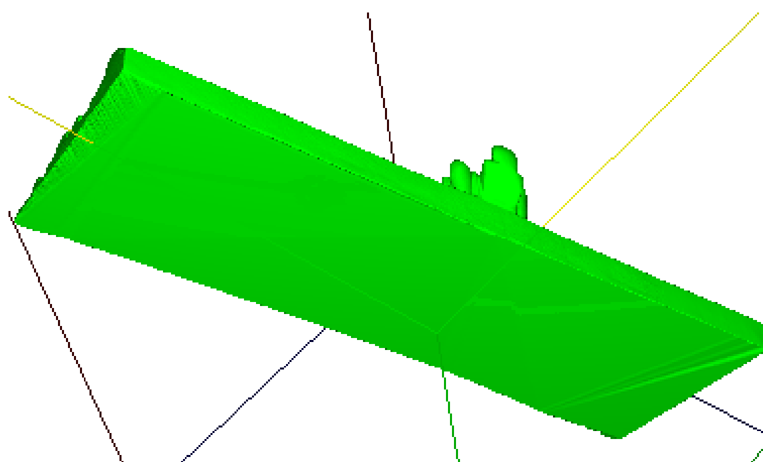


Figura 57. Parte de atrás de la Moldura 26 M.P.

Para la clasificación de los planos se utiliza un algoritmo k-mediana, siendo una variante del clásico k-media (Han, Kamber, and Tung 2001). El algoritmo k-mediana toma el dato más centrado del grupo en vez de tomar el dato del centro del grupo que es lo que hace el k-medias (Han, Kamber, and Tung 2001). El algoritmo K-mediana se comporta mucho mejor que el k-medias, ya que permite filtrar los valores anormales y las interferencias o el ruido que pueden tener los datos. En la bibliografía se pueden encontrar varias implementaciones del algoritmo k-mediana. La primera implementación del k-mediana se encuentra en el libro publicado por Kaufman y Rousseeuw (Kaufman and Rousseeuw 1990) donde se propone el PAM (Particionamiento Alrededor de las Medianas). De todas las variantes de este algoritmo en esta tesis se ha utilizado la propuesta de Raymond T. Ng y Jiawei Han (Ng and Han 1994). Este seudocódigo se puede ver en el *Algoritmo 5*.

1. Se seleccionan k planos representativos de forma aleatoria.
2. Se calcula 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. Se selecciona el par $< P_i, P_h >$ que tiene el valor mínimo para TC_{ih}. Si éste valor es negativo, se sustituye P_i por P_h y se vuelve al paso 2.
4. En caso contrario, para cada plano no seleccionado, se encuentra su representante más cercano y se finaliza el algoritmo.

Algoritmo 5. Seudocódigo del algoritmo utilizado para calcular el k-mediana.

El *Algoritmo 5* utiliza la distancia euclídea en 2 dimensiones como valor de coste entre dos planos. Esta distancia se calcula pasando el vector normal de cada uno de los planos a un sistema de coordenadas esféricas. Como los vectores están normalizados y la esfera utilizada tiene un radio igual a uno, entonces se pueden comparar las normales sólo con los ángulos que forman θ y ρ. La fórmula para comparar las normales se puede ver en la *Ecuación 4*.

$$TC_{hi} = (\theta_i - \theta_j)^2 + (\rho_i - \rho_j)^2$$

Ecuación 4. Comparación de normales.

El *Algoritmo 5* tiene el inconveniente de que necesita conocer a priόri el número de agrupaciones que se tienen que hacer. El tiempo de construcción del octree y los resultados visuales obtenidos varían bastante dependiendo de este valor. En la sección 3.7 se realizará un estudio para conocer la mejor opción del valor del porcentaje. El número total de agrupaciones se obtiene como un porcentaje sobre el total de los planos (ver *Ecuación 5*).

La selección del número de agrupaciones se realiza mediante un estudio detallado, comparando los resultados visuales obtenidos con el tiempo total de construcción del octree. Los resultados de dicho estudio, para diferentes valores de este parámetro, se pueden ver en la sección 3.1.13.

$$Número\ de\ clústeres = \left\lceil \left(\frac{Número\ de\ planos * valor}{100} \right) \right\rceil$$

Ecuación 5.- Número de clústeres para la selección de planos relevantes.

En la bibliografía se pueden encontrar otros algoritmos que utilizan técnicas evolutivas o que seleccionan el número de agrupaciones de manera automática. Uno de ellos es el QT clustering (Quality Treshold clustering) (Heyer, Kruglyak, and

Yooseph 1999). Su uso queda descartado debido al gran coste computacional que tienen asociado.

3.3.4.4. Recorte de los paralelepípedos de los nodos hoja con los planos relevantes seleccionados.

El proceso de recorte de la caja envolvente de los nodos hoja con los planos envolventes seleccionados es crítico a la hora de la creación del EBP-Octree, por ser muy costoso y por depender proporcionalmente del número de planos seleccionados. El coste de este proceso se puede medir en tiempo de construcción y en memoria principal ocupada por la envolvente resultante.

El algoritmo descrito en (Melero 2008) está diseñado para trabajar solamente con modelos de hasta dos millones de polígonos. Como en la mayoría de los modelos existentes en la actualidad el número de estos es mayor, necesitan manejar un gran número de nodos hoja y mantenerlos en memoria principal puede provocar un desbordamiento de esta. Para solucionar este problema lo que se hace es ir guardando las envolventes de los nodos hoja según se van calculando. Esto se puede hacer sin ningún problema ya que son totalmente independientes unas de las otras. La forma de guardar esta información se explica en la sección 3.5.

3.4. Construcción del EBP-octree completo.

Una vez que se han construido los nodos hoja, se han calculado las envolventes y se han guardado en disco, se pasa a construir el octree. Se puede hacer de dos formas:

- Generando un octree completo en memoria principal.
- Generando un octree por niveles, el cual se va creando nivel a nivel y se va guardando por niveles en archivos.

La elección de la manera de construir el octree se realiza en tiempo de ejecución, valorando para ello una serie de parámetros como son la cantidad de memoria principal del ordenador donde se calcula el octree, el tamaño en número de polígonos, la resolución del modelo, el tamaño máximo de niveles que tiene el octree, etc. Si el modelo es pequeño y el octree cabe en memoria principal entonces el programa elige la primera opción, pero si el modelo es muy grande o la memora principal es muy pequeña entonces el programa lo calcula por niveles de abajo hacia

arriba, generando un octree partido. Esta elección se hace porque el mantener el octree en memoria principal para modelos muy grandes formados por varias decenas de millones de polígonos es inviable, ya que en la mayoría de ordenadores se desbordaría la memoria principal. Como posteriormente se verá, el crear el octree por niveles conlleva que cuando este se cargue posteriormente en memoria dentro de un escenario también se cargará partido. En la *Figura 58* se puede ver gráficamente cómo se crea el octree. La parte en rosa es la que se mantiene en memoria y la parte en azul claro es la que se mantiene en disco.

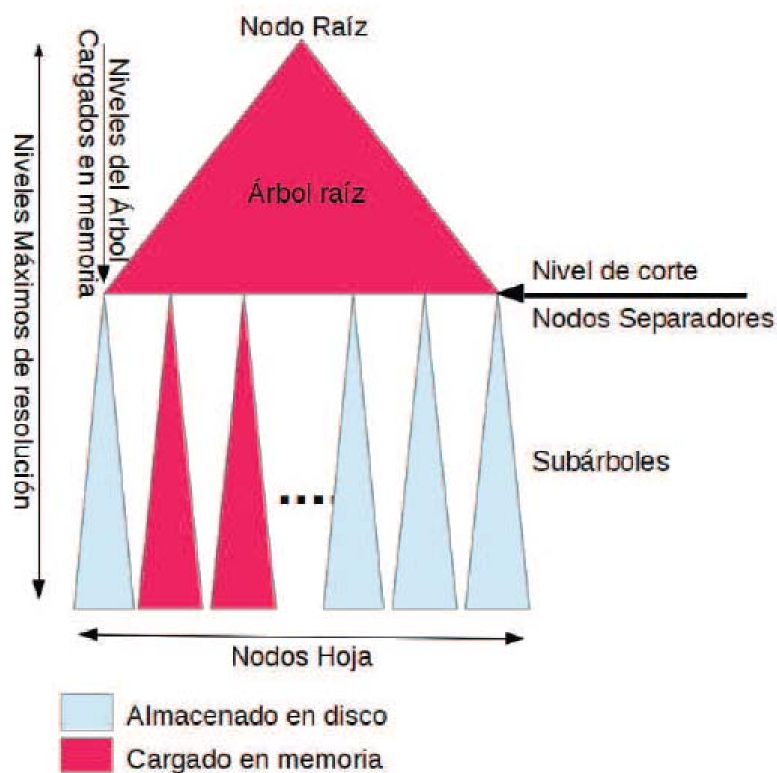


Figura 58. Octree que no cabe en memoria y ha sido generado partido.

La construcción del octree se realiza utilizando la técnica de *top-down*. Para ello se toman los *octcodes* de los nodos hoja, los cuales marcan el camino que se debe seguir desde el nodo raíz hasta el nodo hoja en cuestión. Con el *octcode* y basándose en el trabajo de Gargantini (Gargantini 1982) se puede saber, en cada nivel del octree, cuál es el hijo por el que se tiene que seguir para llegar al nodo hoja.

El proceso de construcción del octree comienza tomando el primer *octcode* del vector de nodo hoja. Como todavía no hay ningún nodo intermedio se crea el nodo raíz del

octree. A continuación se calcula con el *octcode* el número del nodo hijo por donde se tiene que seguir. Como tampoco está creado se crea. Estos pasos se van repitiendo hasta llegar al nodo hoja. Seguidamente se toma el segundo *octcode* y se repite el proceso anterior. En este caso puede ser que no se tengan que crear todos los nodos ya que han podido ser creados con el *octcode* anterior, por ejemplo, el nodo raíz. El proceso anterior se repite para todos nos nodos hoja que se tienen en el vector.

Un ejemplo gráfico se puede ver en la *Figura 59*, donde se representa la construcción del octree completo en memoria a partir del *octcode* 37|34256543456)$_8$. Lo primero que hace es calcular el nivel máximo del octree, siendo para este caso once (37_8= nivel 11). A continuación, se va calculando el camino que se tiene que seguir, creándose para cada nivel los nodos intermedios.

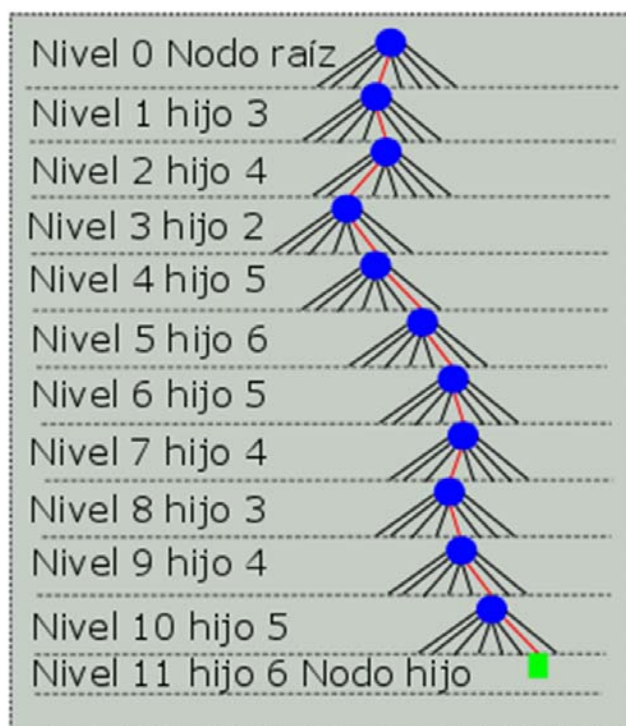


Figura 59. Creación de nodos para el octcode 37|34256543456)$_8$

Cuando la construcción del octree se hace de con memoria externa el proceso es similar. Lo único que cambia es que las envolventes de los nodos intermedios se guardan en archivos según se van calculando, y una vez guardadas se liberan de la memoria.

3.4.1. Cálculo de la envolvente de los nodos grises por niveles del EBP-Octree.

Como ya se ha visto, la construcción del EBP-Octree se puede hacer de dos formas distintas. Cuando el octree es pequeño y cabe en memoria o cuando es más grande y no cabe en memoria. Si el EBP-Octree se crea en memoria el proceso de construcción sigue los pasos que se han detallado anteriormente. Pero si este se va a construir partido por niveles el proceso cambia para no desbordar la memoria principal. Aquí el cálculo de las envolventes se hace de forma progresiva destruyéndose las que ya no se van a utilizar más.

Más detalladamente: el proceso de cálculo de las envolventes de los nodos grises consiste en tomar todos los datos de los nodos hijos (planos envolventes y vértices de los polígonos de las envolventes) y se calcula la envolvente siguiendo los pasos detallados para el cálculo de la envolvente del nodo hoja. Una vez obtenida la envolvente del nodo gris se guarda en los archivos, se propaga el valor de las etiquetas de las esquinas de los nodos hijos sobre el padre y se libera el espacio que ocupaban los nodos hijos, ya que esto no van a ser utilizados más en el proceso de construcción del EBP-Octree. El seudocódigo donde se detallan todos estos pasos se puede ver en el *Algoritmo 6*.

```
CalculoEnvolventeNodoGris(NumNodo)
   lista listaPuntosEnvolventesHijos;
   lista numPlanosEnvolventesHijos;
   Para cada uno de los hijos del nodo gris
        Añadir puntos envolventes sobre la lista listaPuntosEnvolventesHijos
        Añadir planos envolventes sobre la lista numPlanosEnvolventesHijos
   calcularEnvolventeNodoGris(listaPuntosEnvolventesHijos,
                              numPlanosEnvolventesHijos)
   Guardar envolvente en archivos
   Propagar posición esquinas hijos sobre nodo padre
   Calcular posición (dentro o fuera del sólido) de los nodos hijos terminales
   Liberar de memoria la información de los nodos hijos
```

Algoritmo 6. Seudocódigo para el cálculo de la envolvente de un nodo gris en un octree partido.

3.4.2. Cálculo de los nodos blancos y negros.

En esta etapa la diferencia con los BP-Octree es la siguiente. En el caso de que se

construya el EBP-Octree partido y almacenado por niveles, el etiquetado de los nodos blancos y negros se realiza recorriendo los nodos grises del nivel inferior. Se comprueban las posiciones de las esquinas de los nodos hijos y si están dentro del modelo se etiquetan como nodos negros. Si están fuera son nodos blancos. Todo este proceso se realiza antes de liberar la información de los nodos hijos. Este paso se puede ver en el *Algoritmo 6*.

3.5. Gestión del EBP-Octree con memoria externa.

El cálculo del EBP-Octree se realiza una sola vez y la jerarquía de envolventes se almacena en una serie de archivos. Así cada vez que se necesite insertar en una escena un modelo calculado previamente solo se tiene que acceder a memoria secundaria. Para poder trabajar con modelos formados por varias decenas de millones de polígonos, se crea una estructura de archivos que permite acceder a los datos deseados de forma eficiente. El número de archivos necesarios para guardar la información necesaria del EBP-Octree es de siete. Esta división se realiza así para poder acceder a la información que se desee directamente, ya que los datos que se necesitan para visualizar el modelo son diferentes a los que se necesitan para calcular test de inclusión o de colisiones. Las extensiones de los archivos que se utilizan son:

- *".vtx"*: Almacenan la información de los vértices que forman los polígonos del modelo original. Para cada vértice se guarda el valor de las coordenadas X, Y y Z.
- *".tgl"*: En este archivo se almacena información de todos los polígonos del modelo. Para ello se guarda la posición del vértice dentro del archivo *".vtx"*, ordenados por número de vértice. De esta forma, como se conoce el tamaño en bytes que ocupan en disco los números enteros largos, para acceder a uno solo hay que multiplicar el tamaño por la posición del vértice que se quiere leer. De esta forma se ahorra espacio por no tener que guardar las tres coordenadas de cada vértice para cada uno de los polígonos que lo comparten.
- *".fcn"*: En este archivo se almacenan las normales para todos los polígonos que forman parte del modelo. De cada normal se almacenan los cuatro coeficientes de la ecuación del plano (A, B, C, D). Al igual que en el caso de los vértices se guardan ordenadas por posición.

- *".geo"*: se almacena, para cada nodo hoja, el número de triángulos que lo cortan y la posición de cada triángulo en el archivo *".tgl"*.
- *".bvp"*: aquí se almacena la información de los vértices que forman la envolvente calculada de los nodos. Es decir, se guarda, para cada polígono, un puntero al archivo *".fcn"* con la posición de la normal del plano, a continuación el número de vértices que forman el polígono y por último las coordenadas de los vértices.
- *".bpl"*: se guarda la información de los planos seleccionados para recortar el nodo: número de planos seleccionados y, para cada plano, la posición de la normal en el archivo *".fcn"* y el desplazamiento que se le ha aplicado al plano para recortar el nodo (offset). También se guarda el número de planos que forman la envolvente recortada del nodo y un puntero hacia el archivo *".bvp"* donde están almacenados los vértices que forman los planos que se han obtenido tras recortar el nodo.
- *".oct"*: Y por último aquí se almacena el octree, guardando la información necesaria para poder montar el árbol. La información de los nodos que forman el árbol se almacena por niveles, empezando por el nodo raíz y acabando con los nodos hoja. La información que se almacena para los nodos interiores es diferente a la que se almacena para los nodos hoja.
    - Información que se guarda para un nodo interior:
        - dos bytes para saber cómo son sus nodos hijos: si son negros, blancos, grises o nodos hoja.
        - Un puntero hacia este mismo archivo, donde está almacenado el primer nodo hijo.
        - Un puntero hacia el archivo *".bpl"* donde se almacenan los planos que forman la envolvente recortada del nodo.
    - Para los nodos hoja la información que se guarda es:
        - Un puntero hacia el archivo *".geo"*, con los triángulos reales del modelo que cortan a ese nodo hoja.
        - Un puntero hacia el archivo *".bpl"* con la información de los planos envolventes obtenidos tras el recorte del nodo.

En la *Figura 60* se puede ver gráficamente como se relacionan todos estos archivos unos con otro.

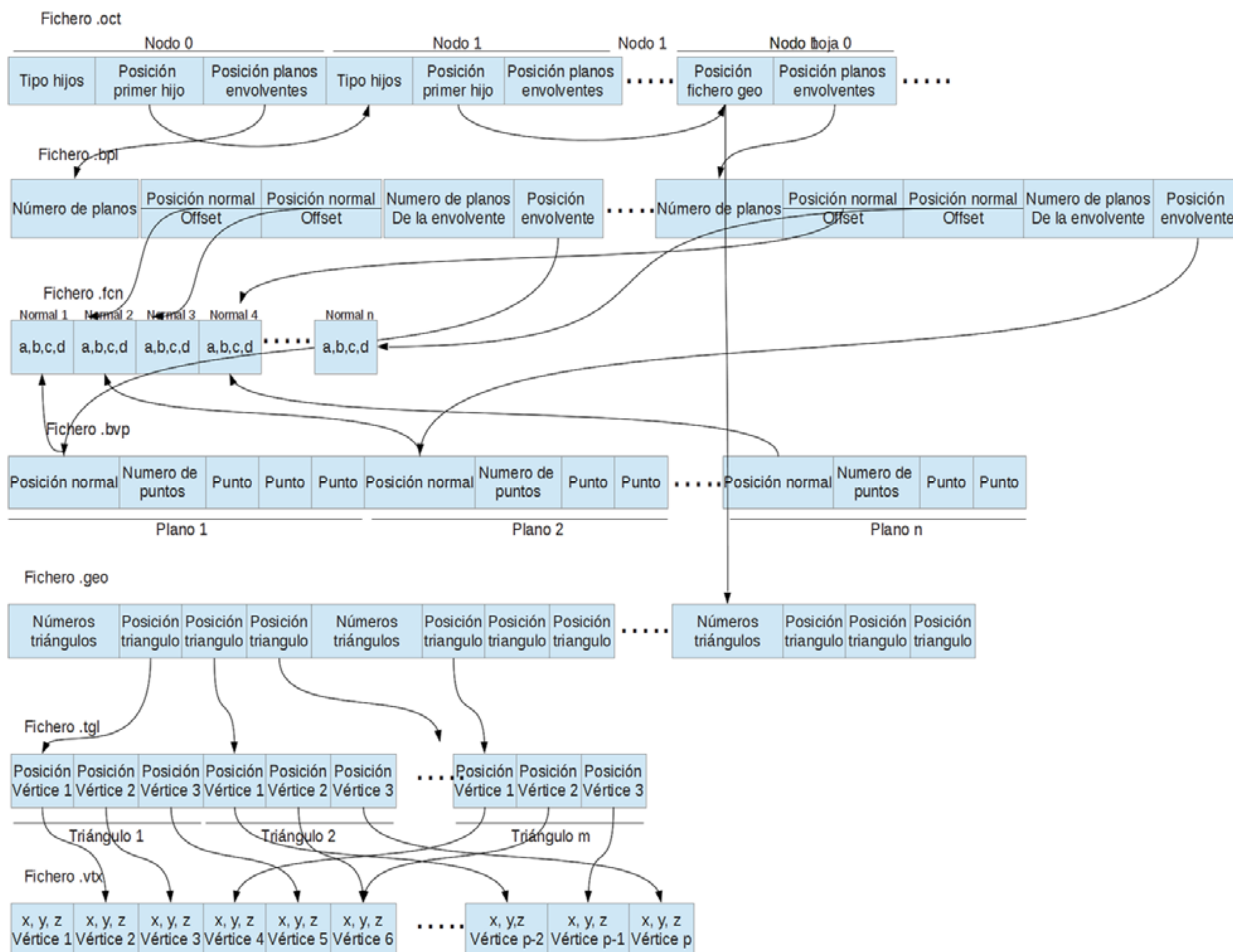


Figura 60. Estructura de archivos para almacenar un EBP-Octree en disco.

3.5.1. Inversión del EBP-Octree calculado out-of-core.

Si el EBP-Octree se ha calculado de manera partida, es decir guardando nivel por nivel, el almacenamiento se ha hecho al revés, ya que primero se guardaron los nodos hoja, después los nodos del penúltimo nivel y así sucesivamente hasta llegar al nodo raíz. Para trabajar con el EBP-Octree es más cómodo tenerlo almacenado al revés, es decir, primero el nodo raíz a continuación los nodos del primer nivel, y así sucesivamente hasta llegar a los nodos hoja. Por tanto es recomendable invertir el EBP-Octree. Para ello se utiliza un archivo auxiliar donde se van guardando los

nodos, empezando por el nodo raíz y se va cambiando la referencia a la posición en el archivo de los nodos hijos, hasta llegar a los nodos hoja.

El tener el EBP-Octree almacenado por niveles hace que se pueda transmitir hasta un cierto nivel y que esa planificación dependa, por ejemplo, del número de frames que se necesiten por segundo.

3.5.2. Carga del EBP-Octree en memoria.

El proceso descrito en el apartado anterior sólo se tiene que realizar una sola vez para cada modelo (*out-of-core*). Una vez realizado ya se puede cargar el modelo tantas veces como se quiera sin tener que realizar los cálculos previos. Basta con leer la información de los archivos que necesitemos según se vaya a visualizar, o se vaya a realizar test de inclusión punto en sólido, o se vaya a hacer la detección de colisiones entre modelos.

Al trabajar con modelos formados por varias decenas de millones de polígonos, los datos que se necesitarían, por ejemplo, para poder visualizar el modelo a su máxima resolución, desbordarían la memoria principal. Para evitar este problema y teniendo en cuenta que en un instante dado sólo se utiliza una pequeña parte del modelo, el octree no se carga entero en memoria, sino que se divide por niveles en varias capas, de manera que en un instante dado sólo se mantienen en memoria los nodos de los primeros niveles. En la *Figura 58* se puede ver gráficamente como se distribuye el EBP-Octree entre memoria principal y memoria masiva.

Para entender mejor la forma de cagar el EBP-Octree en memoria se introduce el término *nivel de corte*: Se define como el nivel máximo del octree que se mantiene en memoria principal.

El nivel de corte del octree se calcula dependiendo del tamaño de la memoria principal del ordenador donde se va a cargar el modelo, de forma que en un ordenador con mayor capacidad de memoria se cargarán más niveles del octree que en otro con menos memoria.

3.5.2.1. Estructura de datos del EBP-Octree.

La estructura de datos propuesta divide el octree por niveles, manteniendo solo en memoria la parte que se necesita en un instante dado. Para poder almacenar en memoria esta estructura se necesita tener una lista de nodos interiores y otra lista de

nodos de corte o separadores.

Los nodos interiores almacenan información de la envolvente y el tipo o posición de los nodos hijos a este. El tipo de nodo hijo se codifica con cuatro valores: un valor para los nodos blancos, otro valor para los nodos negros, un valor positivo indicando la posición dentro del vector del nodo hijo y, por último, un valor negativo para saber que se referencia a un nodo separador.

Los nodos separadores tienen información de la envolvente del nodo y la posición de comienzo del primer nodo hijo en el archivo ".oct". De esta forma cuando se necesita tener más detalle de una zona del modelo, se puede acceder al archivo y cargarla en memoria.

Para poder cargar y descargar los suboctree de un nodo separador, se mantiene en cada nodo separador una lista de nodos interiores y otra lista de nodos hoja, por cada uno de sus hijos.

3.5.2.2. Carga de la información básica del EBP-Octree en memoria.

En memoria principal siempre se mantiene la parte raíz del octree hasta el nivel de corte y una serie de suboctree que se irán cargando y descargando según se necesiten de forma que no se sature la memoria principal del ordenador. El número de suboctrees que se mantienen en memoria puede variar dependiendo del tamaño de la memoria disponible y del tamaño de estos. Los subárboles que se van a necesitar en un momento dado, tanto para visualizar el modelo, como para realizar un test de inclusión punto en solido o de colisión, en un momento dado, son casi los mismos que en un instante anterior, ya que los movimientos son continuos y siguen una trayectoria lineal. Por esto se mantiene en memoria una caché de subárboles, poniendo y quitando subárboles según se vaya desplazando la zona de interés por el modelo.

Kbyte	Moldura 30%	Moldura 40%	Lucy 30%	Lucy 40%	Amazona 30%	Amazona 40%
Tamaño árbol raíz en memoria	1517,7100	6864,1700	2393,0400	11526,480 0	2731,8700	13154,940 0
Tamaño medio nodos disco	1,4028	1,7516	1,1573	1,4285	1,0564	1,2833
Tamaño medio de los subárboles	4465,5087	5575,7666	3676,2550	4537,6986	3132,9087	3805,9623

Tabla 1. Tamaño en Kbyte del árbol raíz, los nodos en disco y los subárboles.

En la *Tabla 1* se muestran los tamaños de los árboles raíz, una media del tamaño de

los subárboles y el tamaño medio de los nodos guardados en disco, para los tres modelos estudiados, los cuales están formados por más de veintiséis millones de polígonos, para el método de selección de planos de corte relevantes k-mediana.

3.6. Elección del método de selección de los planos relevantes para el cálculo de las envolventes.

En el apartado 3.3.4.3 se plantearon tres métodos para la selección de los planos que se van a utilizar en el cálculo de las envolventes de los nodos del EBP-Octree, se hace un estudio comparativo para ver cuál es el que mejores resultados obtiene en un tiempo razonable.

La elección del método de selección de planos relevantes de la envolvente así como el porcentaje de planos seleccionados o número de agrupaciones creadas se ha hecho mediante un estudio empírico de los resultados obtenidos en un modelo de tamaño grande.

Para este estudio se ha tomado el modelo de la Amazona Herida, formado por 28 millones de polígonos. Se ha hecho un estudio empírico y detallado, generando el EBP-Octree para los tres métodos de selección de planos relevantes propuestos y tomando un porcentaje de planos que varían entre un 10 y un 50 por ciento en intervalos de 10. Dicho estudio se ha basado en la observación y comparación de los siguientes parámetros:

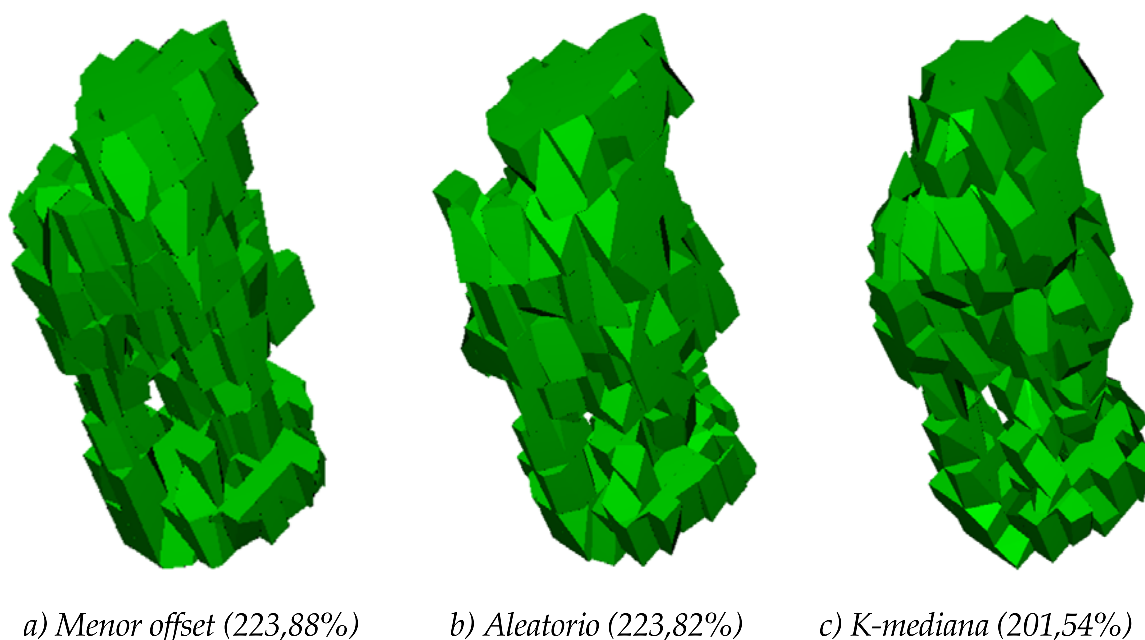
- Percepción visual de los resultados gráficos de las envolventes generadas para cada nivel.
- Tamaño de los archivos generados.
- Tamaño de EBP-Octree por niveles.
- Tiempo de creación del EBP-Octree.
- Tiempo de carga del EBP-Octree en memoria.
- Volumen por nivel de las envolventes calculadas.

A continuación, se presenta los datos obtenidos en este estudio empírico para cada uno de estos parámetros cuando se construyen los diferentes EBP-Octree del modelo de la Amazona Herida.

3.6.1. Resultados visuales por niveles de las envolventes obtenidas.

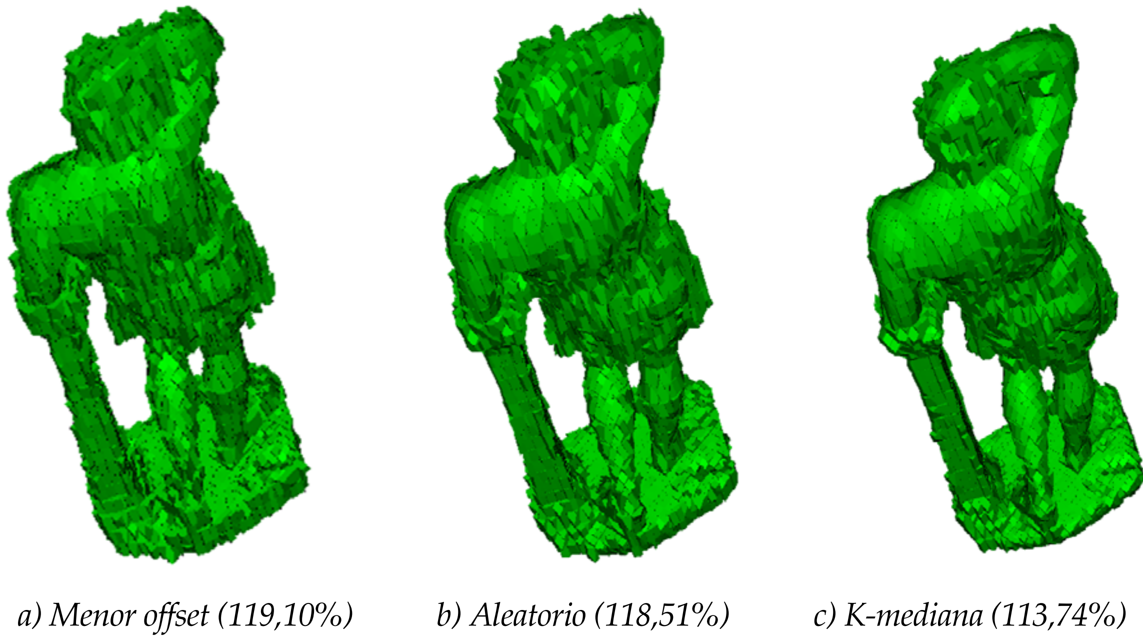
En las siguientes ilustraciones (desde la *Figura 61* hasta la *Figura 75*) se pueden visualizar las imágenes de las envolventes de nivel 4, 6 y 9 obtenidas para la Amazona Herida de 28 millones de polígonos (M.P.) para cada uno de los tres métodos de selección de planos variando el porcentaje de los mismos.

Haciendo un estudio visual de las envolventes obtenidas se puede concluir que, conforme aumenta el porcentaje de planos seleccionados, los resultados visuales que se obtienen mejoran. Otro dato a destacar es que para igual porcentaje de planos seleccionadas el método de k-mediana obtiene unos resultados visuales de las envolventes más continuas, sin grandes saltos entre nodos hermanos y por tanto más próximos al modelo.



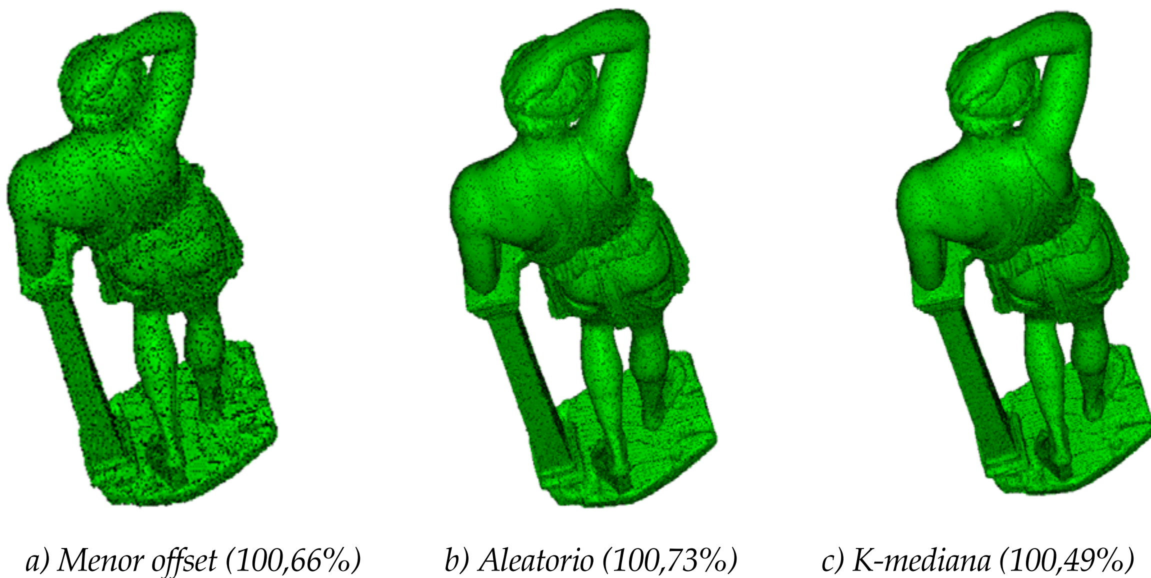
a) Menor offset (223,88%) *b) Aleatorio (223,82%)* *c) K-mediana (201,54%)*

Figura 61. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 10 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



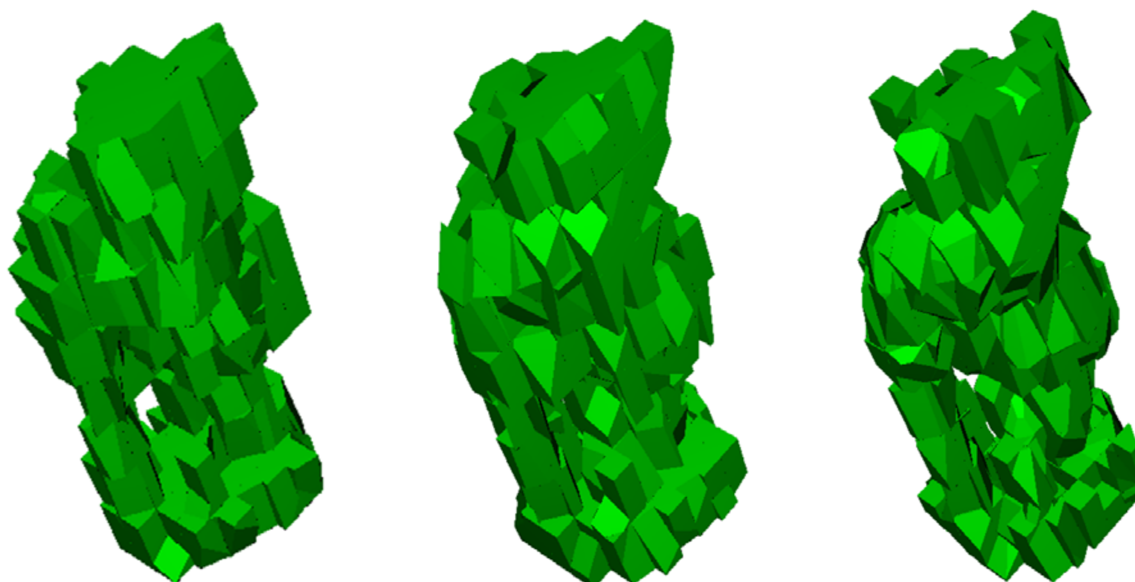
a) Menor offset (119,10%) *b) Aleatorio (118,51%)* *c) K-mediana (113,74%)*

Figura 62. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 10 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



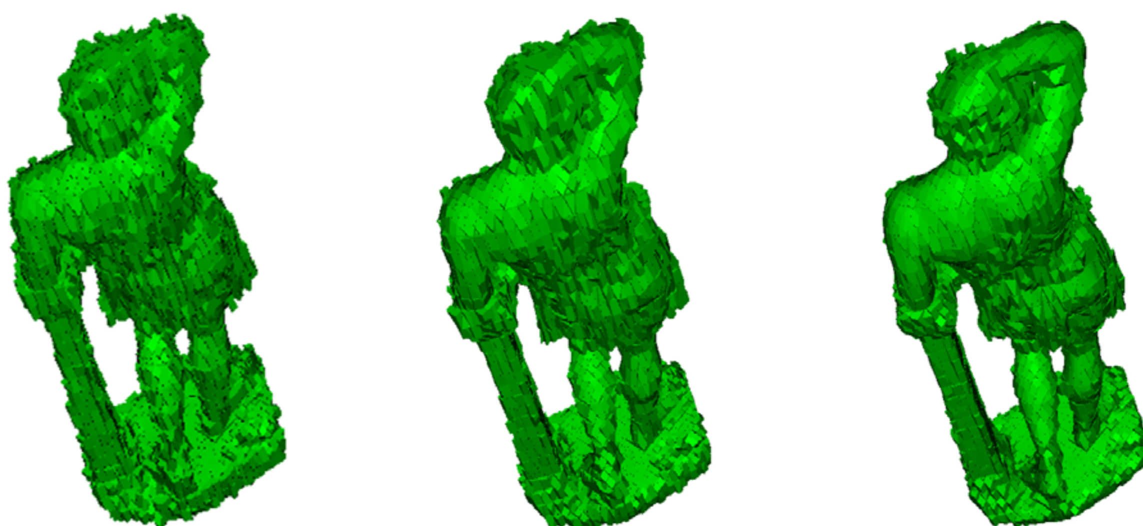
a) Menor offset (100,66%) *b) Aleatorio (100,73%)* *c) K-mediana (100,49%)*

Figura 63. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 10 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



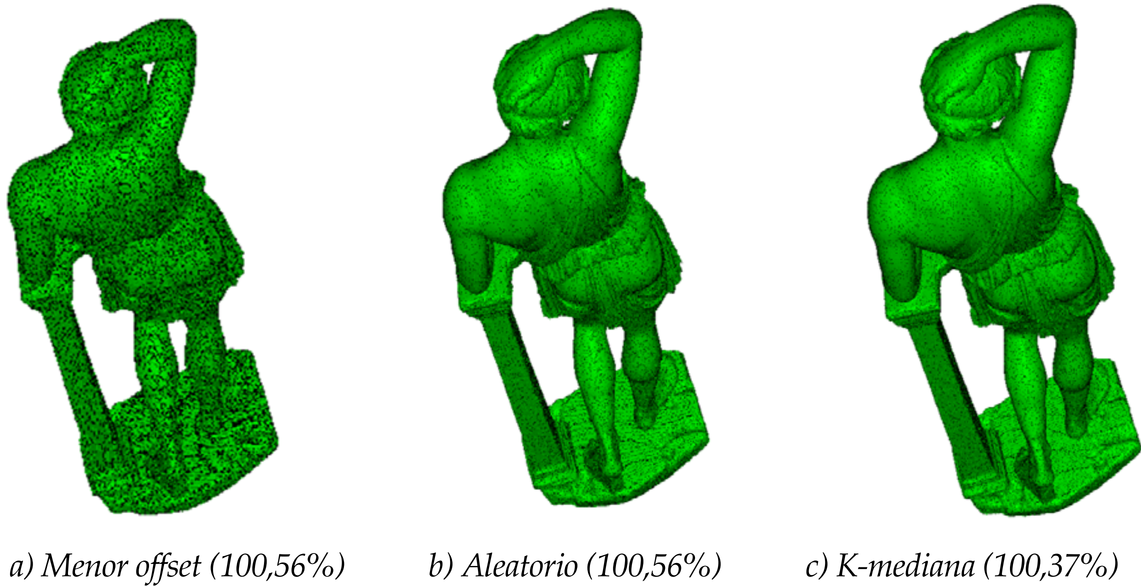
a) Menor offset (225,31%) *b) Aleatorio (224,61%)* *c) K-mediana (194,53%)*

Figura 64. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 20 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



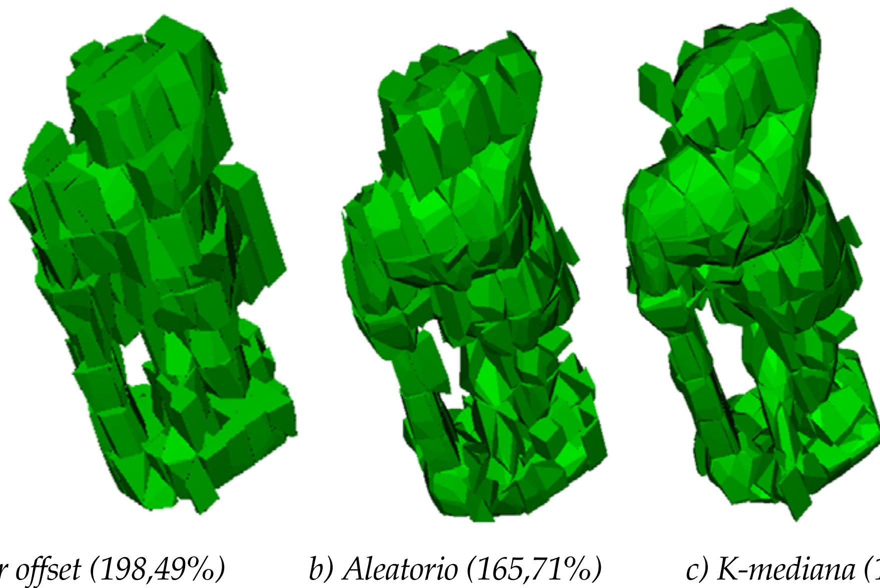
a) Menor offset (118,91%) *b) Aleatorio (118,31%)* *c) K-mediana (112,80%)*

Figura 65. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 20 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



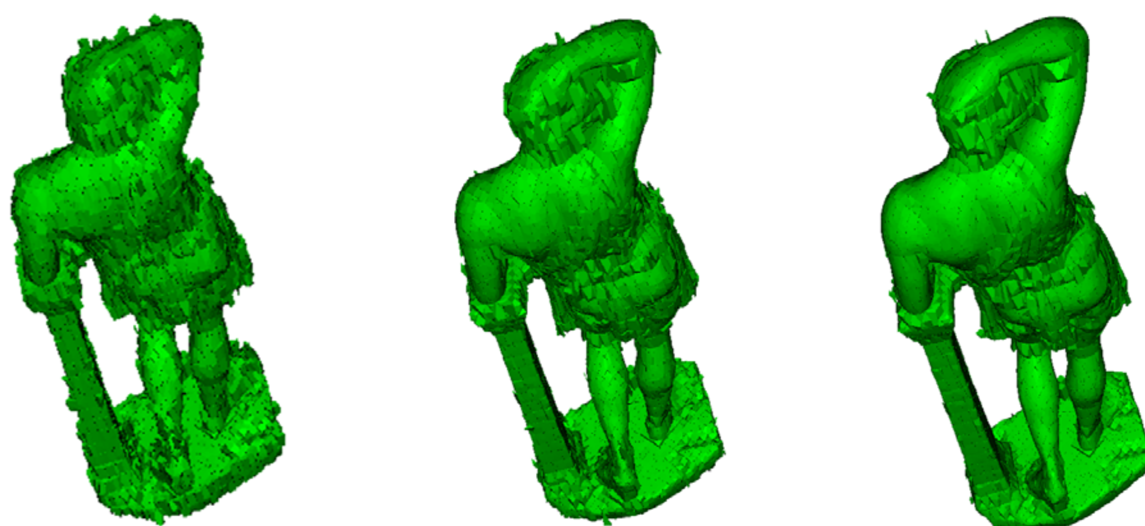
a) Menor offset (100,56%) *b) Aleatorio (100,56%)* *c) K-mediana (100,37%)*

Figura 66. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 20 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



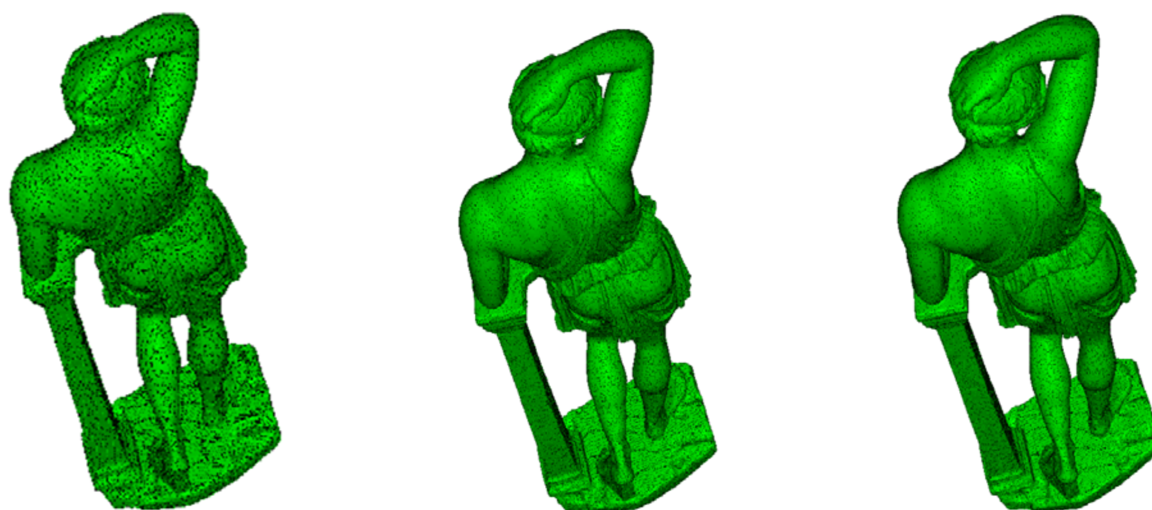
a) Menor offset (198,49%) *b) Aleatorio (165,71%)* *c) K-mediana (143,17%)*

Figura 67. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 30 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



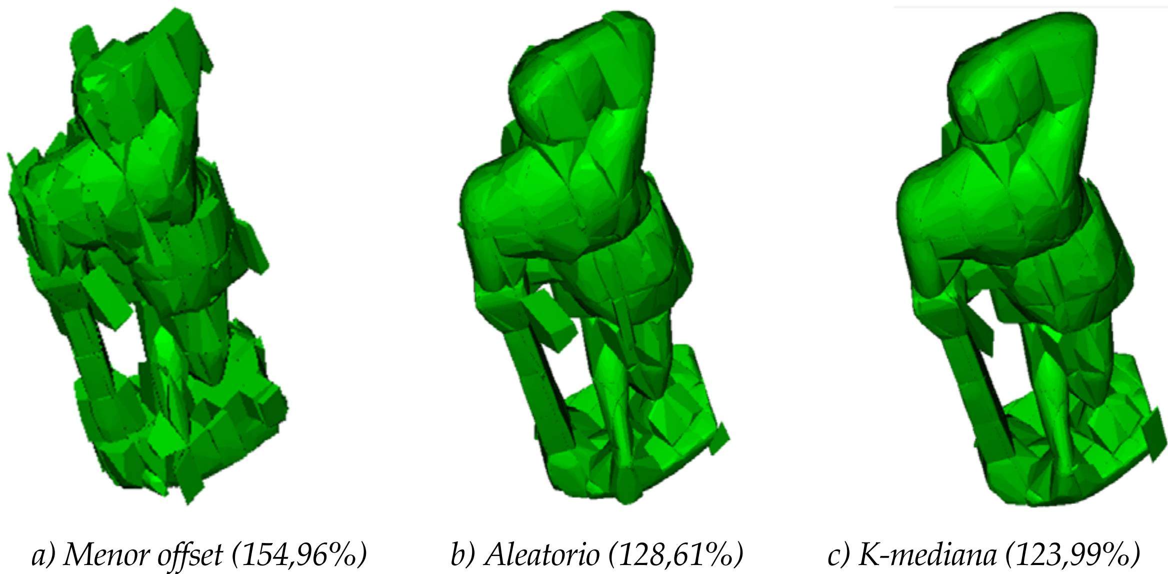
a) Menor offset (112,87%) *b) Aleatorio (109,14%)* *c) K-mediana (106,39%)*

Figura 68. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 30 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



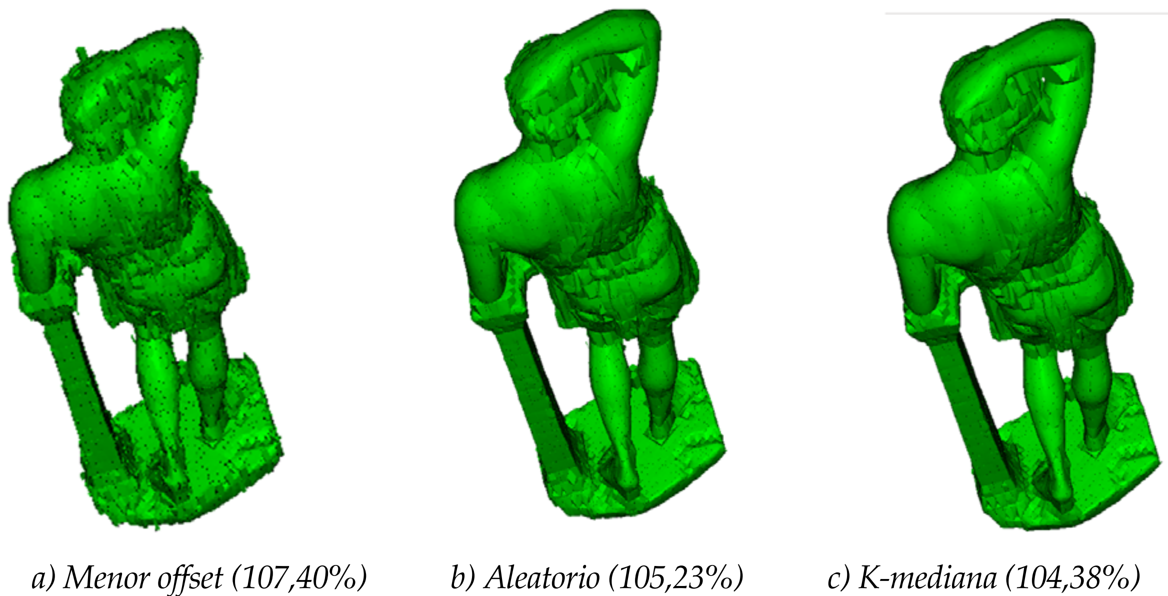
a) Menor offset (100,32%) *b) Aleatorio (100,30%)* *c) K-mediana (100,23%)*

Figura 69. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 30 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



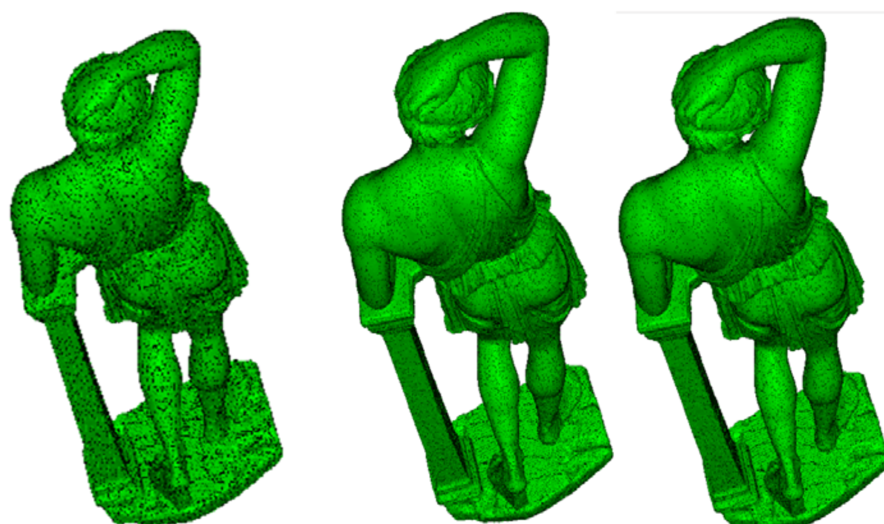
a) Menor offset (154,96%) *b) Aleatorio (128,61%)* *c) K-mediana (123,99%)*

Figura 70. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 40 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



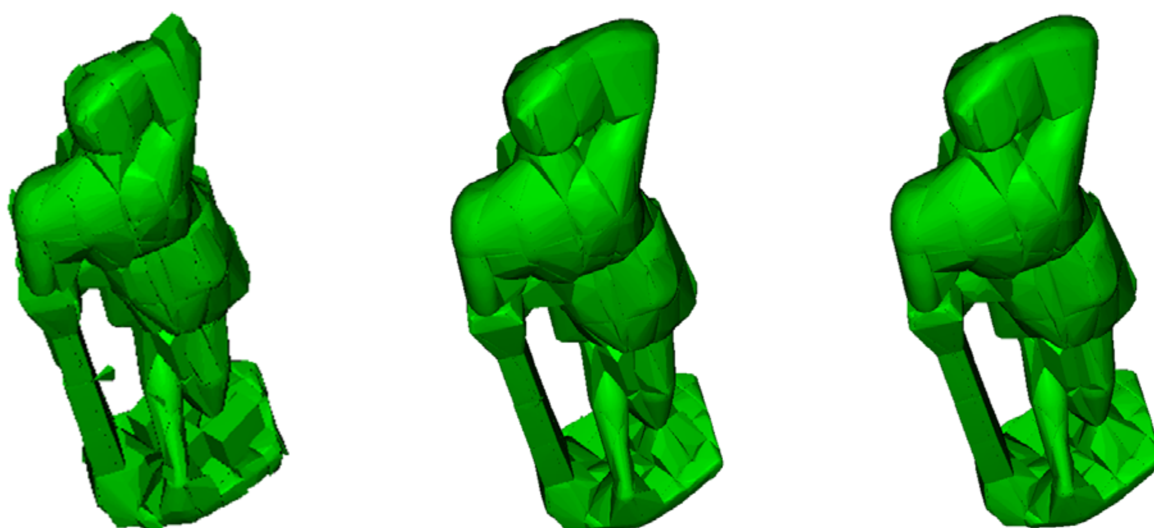
a) Menor offset (107,40%) *b) Aleatorio (105,23%)* *c) K-mediana (104,38%)*

Figura 71. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 40 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



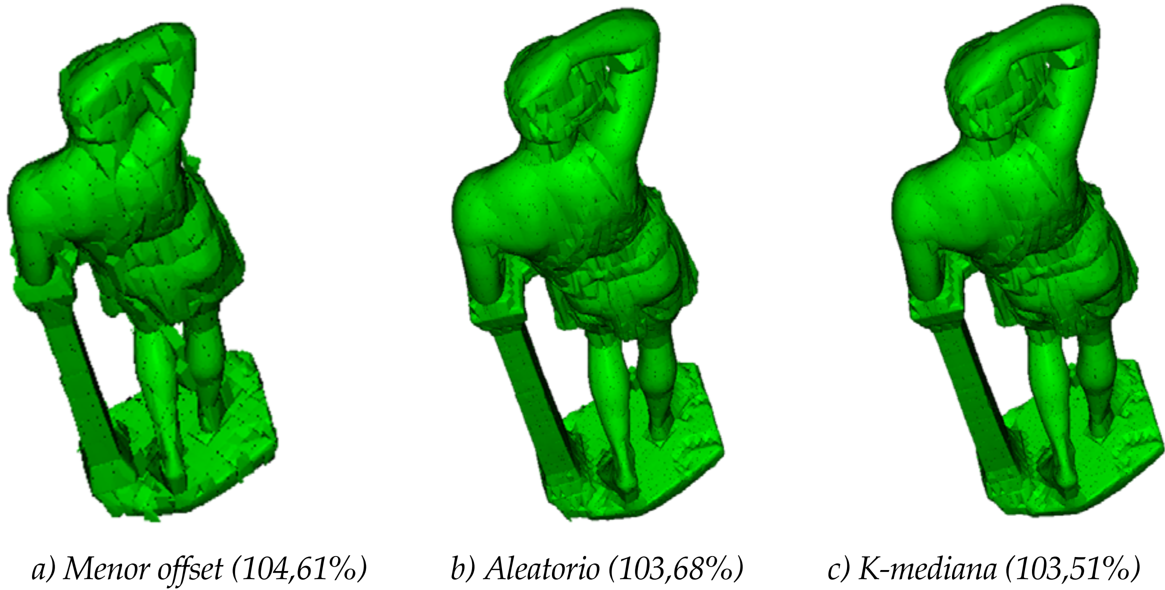
a) Menor offset (100,22%) *b) Aleatorio (100,21%)* *c) K-mediana (100,18%)*

Figura 72. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 40 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



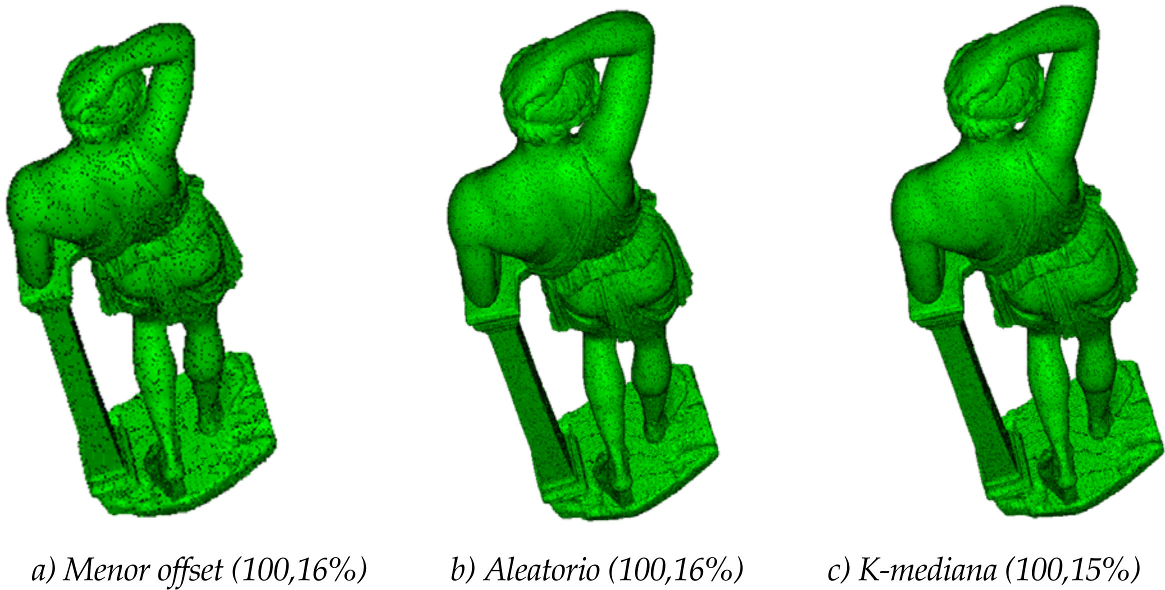
a) Menor offset (129,78%) *b) Aleatorio (117,62%)* *c) K-mediana (116,78%)*

Figura 73. Amazona Herida de 28 M.P. obtenida a nivel 4 y seleccionando el 50 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



a) Menor offset (104,61%) *b) Aleatorio (103,68%)* *c) K-mediana (103,51%)*

Figura 74. Amazona Herida de 28 M.P. obtenida a nivel 6 y seleccionando el 50 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.



a) Menor offset (100,16%) *b) Aleatorio (100,16%)* *c) K-mediana (100,15%)*

Figura 75. Amazona Herida de 28 M.P. obtenida a nivel 9 y seleccionando el 50 por ciento de los planos. El valor entre paréntesis corresponde al porcentaje del volumen de la envolvente con respecto al volumen real del modelo.

3.6.2. El EBP-Octree en disco.

En la *Tabla 2*, *Tabla 3* y *Tabla 4* se muestran una serie de datos en los que se pueden ver los tamaños de los archivos generados. En el *Gráfico 1*, *Gráfico 2* y *Gráfico 3* se puede apreciar una comparativa entre dichos tamaños. Se puede observar que el archivo con mayor tamaño es el ".bvp", que es en el que se almacenan los vértices de los planos que forman las envolventes de todos los nodos del EBP-Octree. Cabe destacar que este archivo no es necesario para los cálculos posteriores de detección de colisiones y solamente se utiliza para la visualización de las envolventes.

Como se puede observar en la *Tabla 2*, *Tabla 3* y *Tabla 4* los únicos archivos que varían en su tamaño son los "*.bpl*" y los "*.bvp*". Los archivos "*.bpl*" almacenan los planos seleccionados para realizar el recorte de los nodos, por esto varían con el porcentaje de planos seleccionados. Los archivos "*.bvp*" almacenan la información de los vértices que forman las envolventes del EBP-Octree, siendo estos mayores conforme se aumenta el porcentaje de planos seleccionados para calcular la envolvente, ya que se aproximan más al modelo cuanto mayor número de planos se eligen.

Menor offset	10%	20%	30%	40%	50%
Amazona Herida28.bpl	267.016.464	362.579.872	531.866.432	742.609.168	1.090.792.544
AmazonaHerida28.bpo	899.185.040	899.185.040	899.185.040	899.185.040	899.185.040
AmazonaHerida28.geo	607.737.280	607.737.280	607.737.280	607.737.280	607.737.280
AmazonaHerida28.nv	168.597.100	168.597.100	168.597.100	168.597.100	168.597.100
AmazonaHerida28.oct	127.479.154	127.479.154	127.479.154	127.479.154	127.479.154
AmazonaHerida28.tgl	224.796.240	224.796.240	224.796.240	224.796.240	224.796.240
AmazonaHerida28.vtx	2.023.166.160	2.023.166.160	2.023.166.160	2.023.166.160	2.023.166.160
Tamaño colisiones	**4.317.977.438**	**4.413.540.846**	**4.582.827.406**	**4.793.570.142**	**5.141.753.518**
AmazonaHerida28.bvp	5.227.077.348	6.064.463.328	7.471.158.780	9.056.868.396	11.414.966.868
Tamaño rendering	**9.545.054.786**	**10.478.004.174**	**12.053.986.186**	**13.850.438.538**	**16.556.720.386**

Tabla 2. Tamaño, en Bytes, de los archivos obtenidos para la Amazona Herida de 28 M.P. seleccionando los planos con menor offset.

Aleatorio	10%	20%	30%	40%	50%
AmazonaHerida28.bpl	267.012.288	363.468.864	534.347.872	746.242.688	1.093.709.488
AmazonaHerida28.bpo	899.185.040	899.185.040	899.185.040	899.185.040	899.185.040
AmazonaHerida28.geo	607.737.280	607.737.280	607.737.280	607.737.280	607.737.280
AmazonaHerida28.nv	168.597.100	168.597.100	168.597.100	168.597.100	168.597.100
AmazonaHerida28.oct	127.479.154	127.479.154	127.479.154	127.479.154	127.479.154
AmazonaHerida28.tgl	224.796.240	224.796.240	224.796.240	224.796.240	224.796.240
AmazonaHerida28.vtx	2.023.166.160	2.023.166.160	2.023.166.160	2.023.166.160	2.023.166.160
Tamaño colisiones	**4.317.973.262**	**4.414.429.838**	**4.585.308.846**	**4.797.203.662**	**5.144.670.462**
AmazonaHerida28.bvp	5.225.244.600	6.047.933.472	7.449.141.384	9.045.937.404	11.425.989.480
Tamaño rendering	**9.543.217.862**	**10.462.363.310**	**12.034.450.230**	**13.843.141.066**	**16.570.659.942**

Tabla 3. Tamaño, en Bytes, de los archivos obtenidos para la Amazona Herida de 28 M.P. seleccionando los planos aleatoriamente.

k-Mediana	10%	20%	30%	40%	50%
Amazona Herida_cerrada28.bpl	316.152.624	405.736.000	572.177.440	783.562.656	1.121.494.016
AmazonaHerida28.bpo	899.185.040	899.185.040	899.185.040	899.185.040	899.185.040
AmazonaHerida28.geo	607.737.280	607.737.280	607.737.280	607.737.280	607.737.280
AmazonaHerida28.nv	168.597.100	168.597.100	168.597.100	168.597.100	168.597.100
AmazonaHerida28.oct	127.479.154	127.479.154	127.479.154	127.479.154	127.479.154
AmazonaHerida28.tgl	224.796.240	224.796.240	224.796.240	224.796.240	224.796.240
AmazonaHerida28.vtx	2.023.166.160	2.023.166.160	2.023.166.160	2.023.166.160	2.023.166.160
Tamaño colisiones	**4.367.113.598**	**4.456.696.974**	**4.623.138.414**	**4.834.523.630**	**5.172.454.990**
AmazonaHerida28.bvp	5.635.242.936	6.425.022.468	7.831.076.292	9.475.811.652	11.862.296.592
Tamaño rendering	**10.002.356.534**	**10.881.719.442**	**12.454.214.706**	**14.310.335.282**	**17.034.751.582**

Tabla 4. Tamaño, en Bytes, de los archivos obtenidos para la Amazona Herida de 28 M.P. seleccionando los planos con el método de K-mediana.

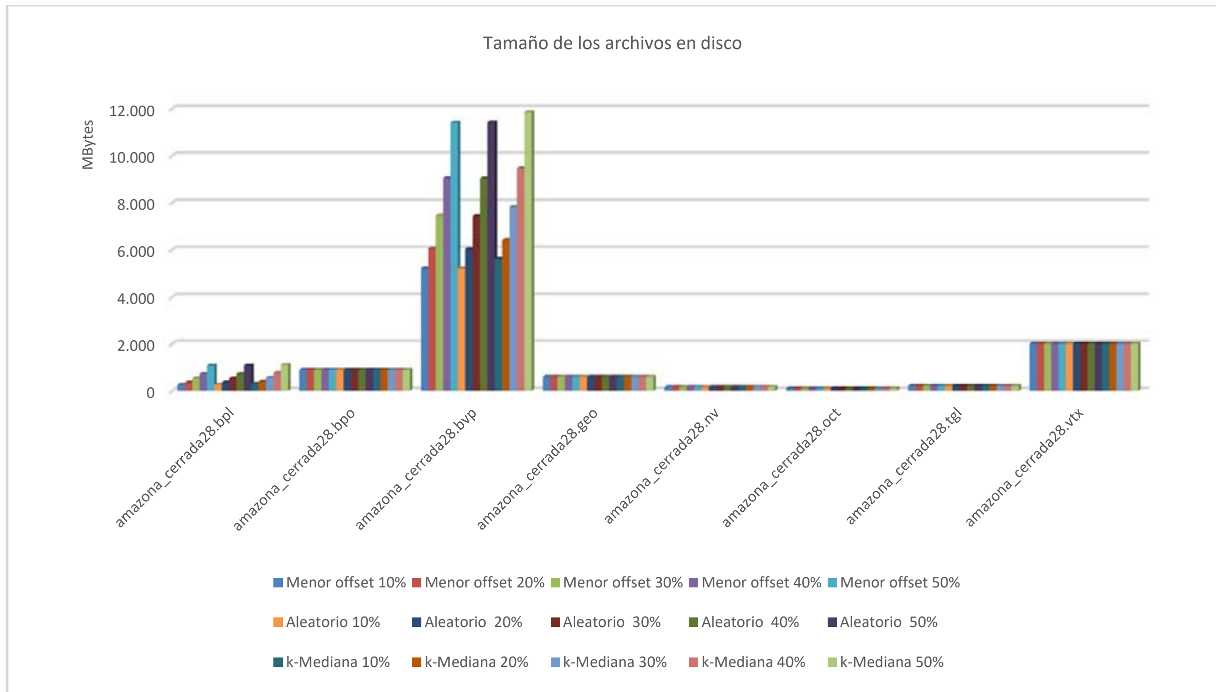


Gráfico 1. Comparación del tamaño de los archivos obtenidos para la Amazona Herida de 28 M. P. por los tres métodos de cálculo.

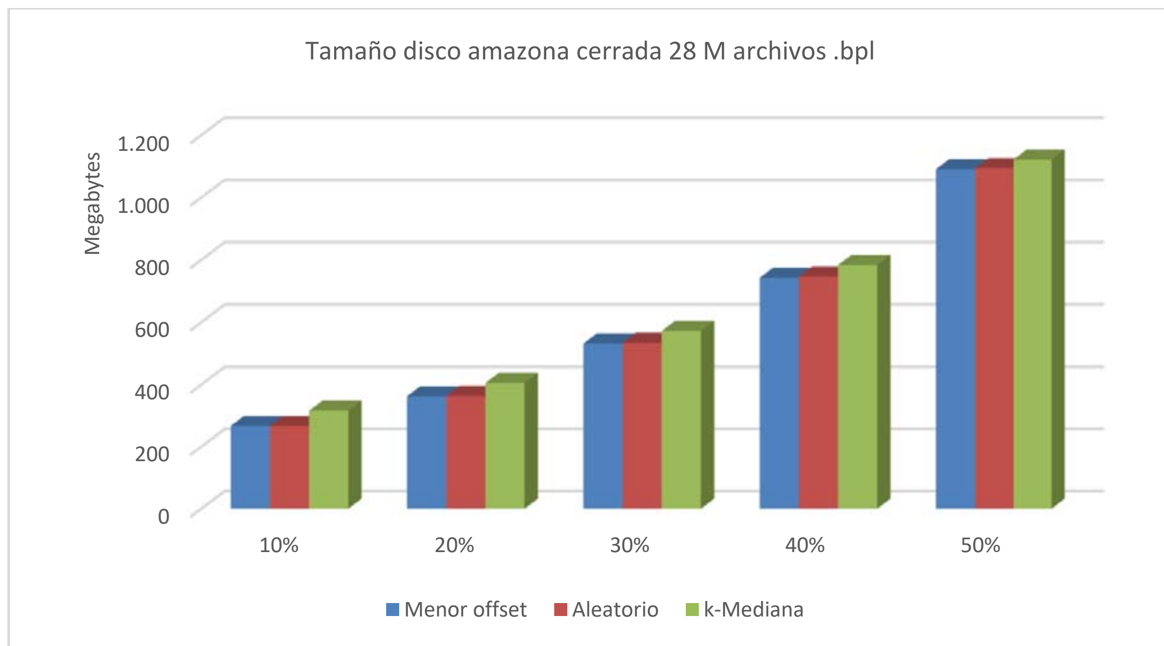


Gráfico 2. Tamaño de los archivos ".bpl", en Megabytes, para la Amazona Herida de 28 M.P.

En el *Gráfico 2* y *Gráfico 3* se puede ver con más detalle el tamaño ocupado en disco de estos dos tipos de archivos.

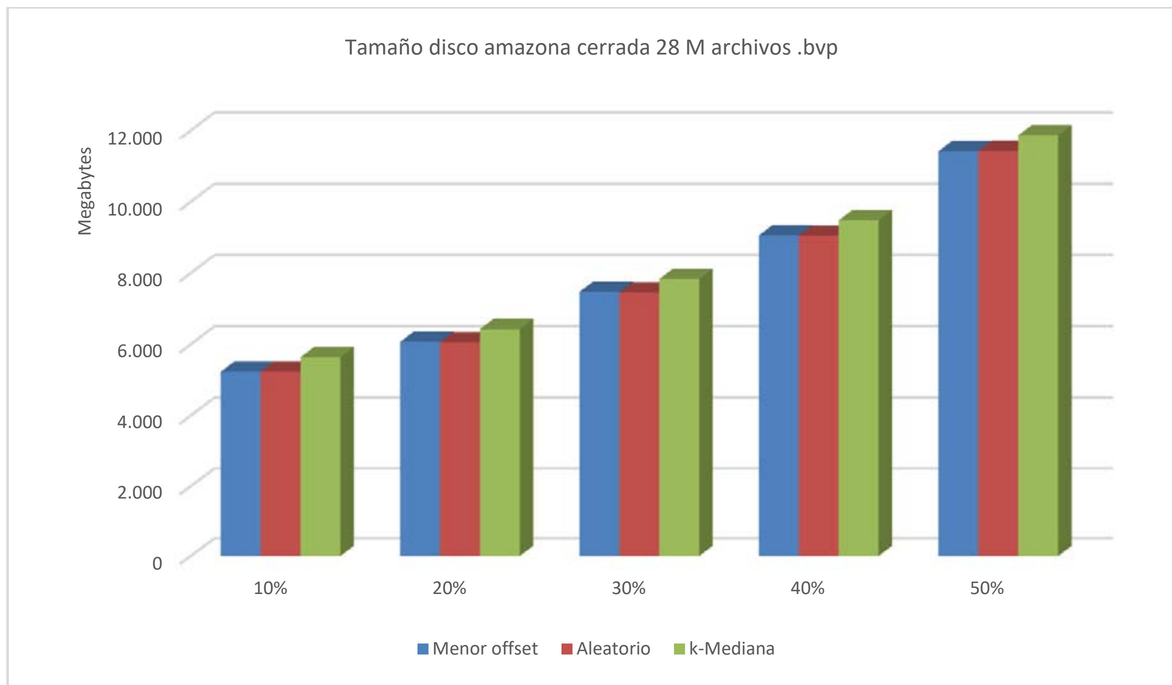


Gráfico 3. Tamaño de los archivos ".bvp", en Megabytes, para la Amazona Herida de 28 M.P.

3.6.3. Tamaño del EBP-Octree por niveles.

Otro dato fundamental que se ha estudiado ha sido el tamaño en bytes del EBP-Octree por niveles. Este dato es muy importante ya que de él se puede obtener el espacio que se necesita en memoria para cargar un modelo. En la *Tabla 5*, *Tabla 6* y *Tabla 7* se presentan los datos de tamaño del EBP-Octree por niveles obtenidos para el modelo de la Amazona Herida utilizando los tres métodos de selección de planos y variando el porcentaje.

Tamaño de memoria que ocupa el árbol en Kbyte	Menor Offset 10%	Menor Offset 20%	Menor Offset 30%	Menor Offset 40%	Menor Offset 50%
nivel 1	0,92	0,92	5,52	64,71	591,34
nivel 2	8,29	8,29	25,00	212,81	1602,78
nivel 3	42,36	42,56	95,61	591,10	3936,42
nivel 4	199,85	199,71	386,99	1953,46	10576,50
nivel 5	934,03	932,64	1590,13	6210,69	25960,82
nivel 6	4322,28	4323,90	6564,24	19278,96	63257,79
nivel 7	35046,03	35060,31	51194,18	118256,28	306608,12
nivel 8	141706,85	142336,23	200963,81	372055,35	756648,62
nivel 9	567958,34	581826,14	790126,29	1202991,23	1956341,43
nivel 10	2264805,57	2440222,17	3136589,21	4064775,98	5459097,73
nivel 11	7757765,62	9348038,90	11369776,20	13180318,64	15471729,75
Total	**10772790,14**	**12552991,77**	**15557317,18**	**18966709,21**	**24056351,30**

Tabla 5. Ocupación en memoria, en Kbyte, del EBP-Octree por niveles utilizando el método de selección de planos de menor offset.

Tamaño de memoria que ocupa el árbol en Kbyte	Aleatorio 10%	Aleatorio 20%	Aleatorio 30%	Aleatorio 40%	Aleatorio 50%
nivel 1	0,92	0,92	8,33	154,21	1760,17
nivel 2	7,85	8,00	31,29	342,02	2863,16
nivel 3	41,65	42,81	105,39	865,07	5334,12
nivel 4	201,75	200,42	416,36	2587,14	13089,38
nivel 5	930,27	923,76	1681,92	7489,46	30193,16
nivel 6	4305,54	4293,99	6890,56	21695,71	69554,52
nivel 7	34984,73	34991,10	53310,56	128469,62	326204,87
nivel 8	141712,82	142461,96	207702,23	391513,18	786713,43
nivel 9	568644,92	584735,82	807881,92	1235656,46	1994244,29
nivel 10	2267060,90	2454554,57	3165840,76	4097393,00	5486614,90
nivel 11	7751422,48	9301107,12	11275985,01	13062108,65	15349706,29
Total	**10769313,83**	**12523320,47**	**15519854,33**	**18948274,52**	**14066278,29**

Tabla 6. Ocupación en memoria, en Kbyte, del EBP-Octree por niveles utilizando el método de selección de planos aleatorio.

Tamaño de memoria que ocupa el árbol en Kbyte	k-mediana 10%	k-mediana 20%	k-mediana 30%	k-mediana 40%	k-mediana 50%
nivel 1	0,92	1,08	11,81	194,14	1965,83
nivel 2	8,34	8,80	41,03	406,42	3031,70
nivel 3	43,61	46,04	137,07	999,70	5916,21
nivel 4	212,88	218,07	519,64	2972,22	14357,88
nivel 5	978,41	1011,87	2022,32	8582,46	32930,84
nivel 6	4840,03	4748,28	8077,88	24631,32	75066,17
nivel 7	37479,70	39512,10	61026,01	142787,95	347489,32
nivel 8	163865,85	158902,98	231637,23	426717,03	827981,37
nivel 9	617360,71	657422,32	878114,82	1319299,56	2079976,92
nivel 10	2655060,43	2705228,50	3354094,46	4296901,23	5679018,04
nivel 11	8165034,01	9755806,70	11783237,93	13611114,60	15872668,90
Total	**11644884,89**	**13322906,74**	**16318920,20**	**19834606,63**	**24940403,18**

Tabla 7. Ocupación en memoria, en Kbyte, del EBP-Octree por niveles utilizando el método de selección de planos k-mediana.

De los datos anteriores podemos deducir que lo más relevante es la suma de los primeros cinco niveles, ya que este valor representa la ocupación real del modelo en memoria principal. En la *Gráfico 4* se muestra una comparación de la ocupación de memoria que tiene la Amazona Herida cuando se calcula para cada uno de los métodos y cuando se varía el porcentaje. Se puede apreciar como el tamaño de los primeros 5 niveles del EBP-Octree aumenta considerablemente en el caso en el que se seleccionan el cincuenta por ciento de los planos. También se puede observar que el tamaño de los cinco primeros niveles del EBP-Octree es muy similar para los tres métodos aunque siempre es un poco mayor en los obtenidos mediante el método de k-mediana.

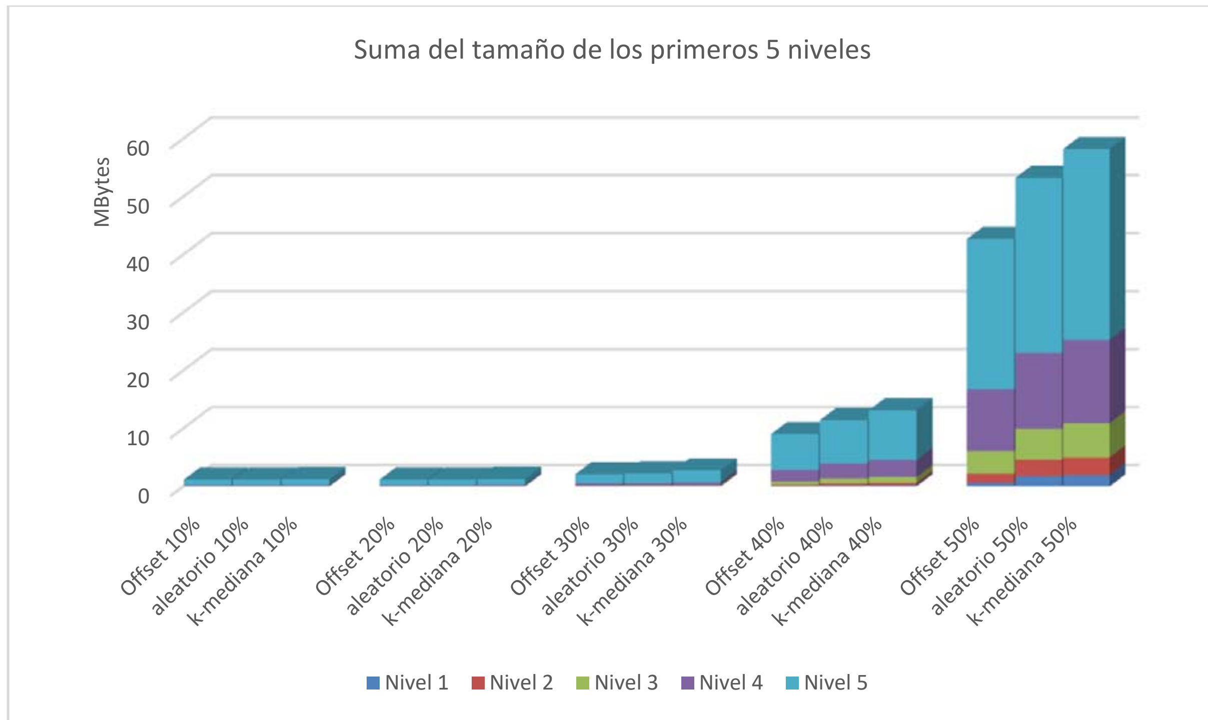


Gráfico 4. Suma del tamaño de los 5 primeros niveles del EBP-Octree.

3.6.4. Tiempo de creación del EBP-Octree.

A continuación, se hace un estudio detallado de tiempos de creación del EBP-Octree. Para ello, el tiempo se ha dividido en las siguientes siete etapas:

- Tiempo para crear nodos hoja.
- Tiempo en calcular la posición de las esquinas.
- Tiempo en calcular los planos envolventes de los nodos hoja.
- Tiempo en crear los caminos del árbol.
- Tiempo en calcular el recorte de los nodos hoja.

- Tiempo en guardar en archivos y propagar los valores por el árbol.
- Tiempo total en calcular los archivos y las envolventes del objeto.

De esta manera se puede ver cómo se ha distribuido el tiempo de creación del EBP-Octree, apreciándose las etapas que han consumido más tiempo con respecto a las que se han realizado más rápidamente. En la *Tabla 8*, *Tabla 9* y *Tabla 10* se pueden ver los tiempos obtenidos en cada una de las etapas para la construcción del EBP-Octree por cada uno de los tres métodos de selección de planos. En el *Gráfico 5*, *Gráfico 6* y *Gráfico 7* se puede observar la distribución de tiempos de construcción del EBP-Octree para cada una de las etapas mencionadas anteriormente. En ellas se puede deducir que la etapa a la que más tiempo se le dedica es a la de calcular los planos envolventes de los nodos hoja, y a la que menos a la creación de los nodos intermedios del octree. Otra conclusión lógica que se puede observar en los tres métodos de selección de planos es que el tiempo en realizar cada una de las etapas es mayor cuanto mayor sea el porcentaje de planos seleccionados.

Menor Offset	10%	20%	30%	40%	50%
Tiempo para crear nodos hoja	210,361	253,647	248,478	266,960	245,011
Tiempo en calcular la posición de las esquinas	47,419	45,754	46,719	47,046	51,191
Tiempo en calcular los planos envolventes de los nodos hoja	2341,630	2347,550	2352,970	2355,970	2371,650
Tiempo en crear los caminos del árbol	4,533	5,050	6,973	6,658	5,442
Tiempo en calcular recorte de los nodos hoja	118,664	164,600	171,855	215,646	276,851
Tiempo en guardar en archivos y propagar los valores por el árbol	159,992	179,702	217,113	283,022	848,317
Tiempo total en calcular los archivos y las envolventes del objeto	**2883,530**	**2997,600**	**3047,130**	**3177,390**	**3800,730**

Tabla 8. Distribución de tiempos, en segundos, para construir el EBP-Octree utilizando la selección de planos con menor offset.

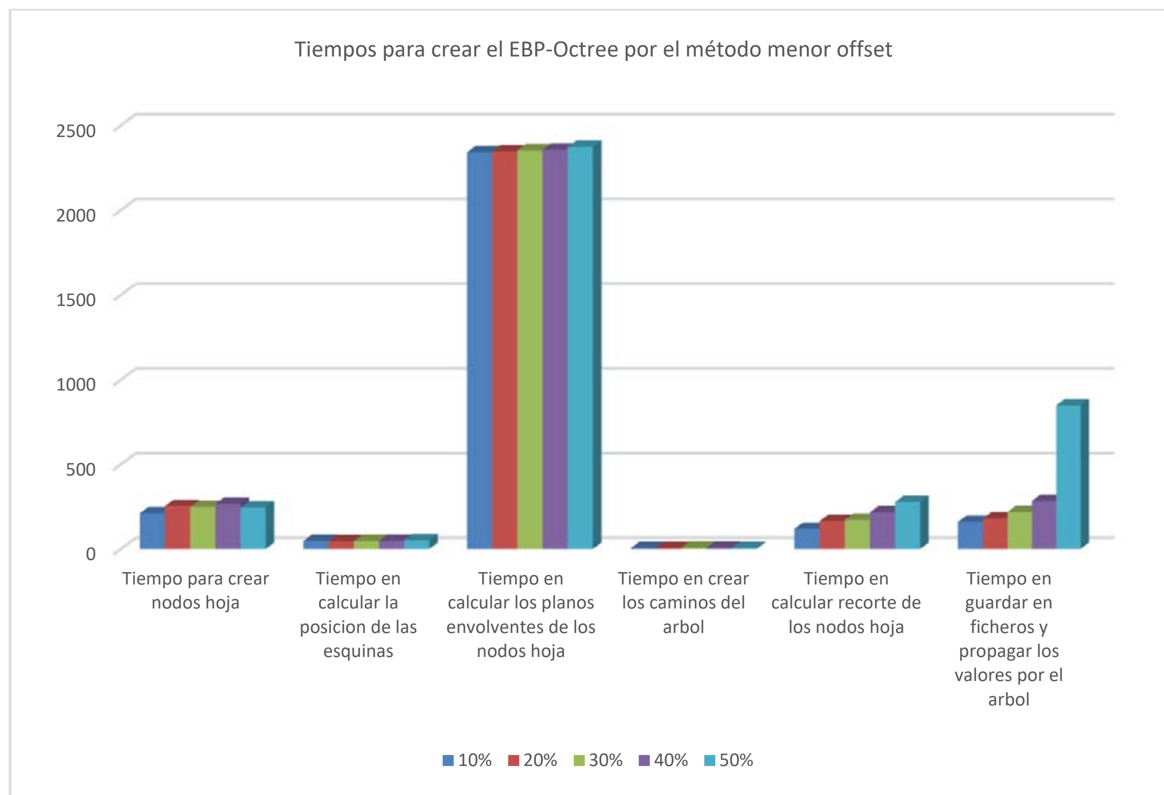


Gráfico 5. Tiempos por etapas, en segundos, para construir el EBP-Octree de la Amazona Herida de 28 M. P. seleccionando los planos con menor offset.

Aleatorio	10%	20%	30%	40%	50%
Tiempo para crear nodos hoja	199,328	203,267	194,125	209,331	192,707
Tiempo en calcular la posición de las esquinas	25,171	25,232	24,953	24,578	25,192
Tiempo en calcular los planos envolventes de los nodos hoja	2307,330	2306,920	2319,440	2323,640	2342,920
Tiempo en crear los caminos del árbol	1,130	1,165	1,400	2,316	3,936
Tiempo en calcular recorte de los nodos hoja	104,791	123,168	140,337	166,305	212,726
Tiempo en guardar en archivos y propagar los valores por el árbol	148,536	165,041	206,122	269,487	858,673
Tiempo total en calcular los archivos y las envolventes del objeto	**2786,740**	**2825,250**	**2886,860**	**2996,150**	**3636,720**

Tabla 9. Distribución de tiempos, en segundos, para construir el EBP-Octree utilizando la selección de planos aleatorios.

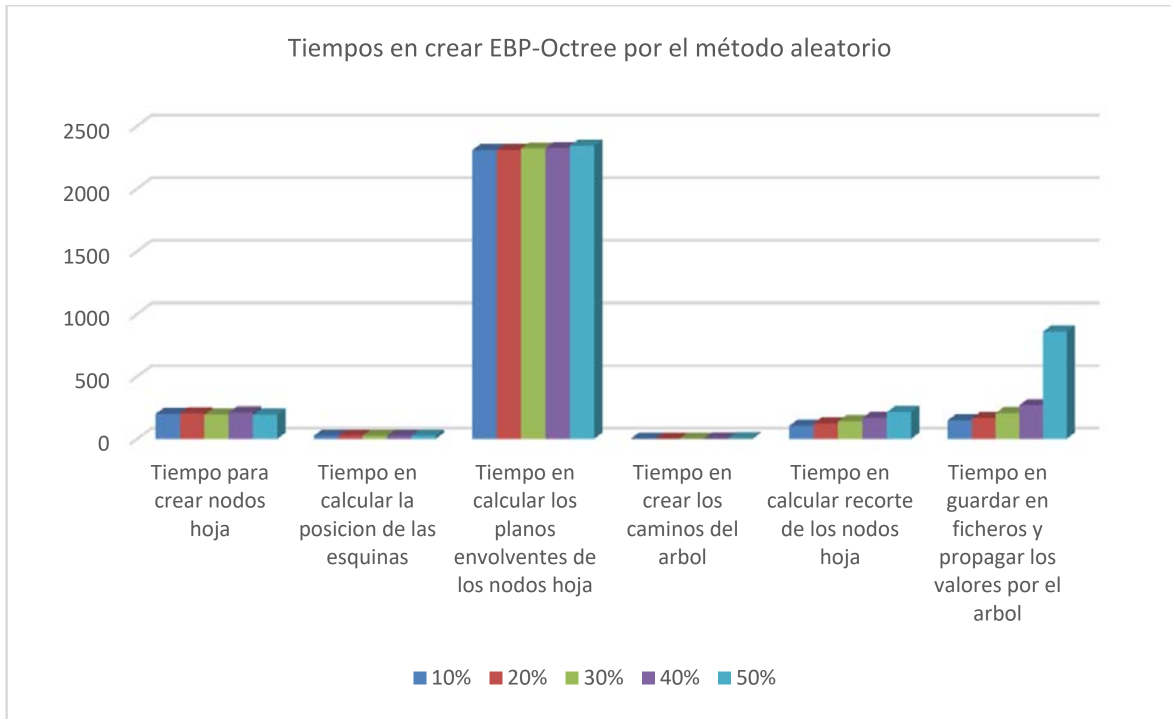


Gráfico 6. Tiempos por etapas, en segundos, para la construcción del EBP-Octree de la Amazona Herida de 28 M. P. seleccionando los planos de manera aleatoria.

k-mediana	10%	20%	30%	40%	50%
Tiempo para crear nodos hoja	202,056	170,137	168,136	163,688	217,310
Tiempo en calcular la posición de las esquinas	25,165	25,418	25,538	25,260	37,000
Tiempo en calcular los planos envolventes de los nodos hoja	2694,960	2891,470	2988,900	3086,610	3113,040
Tiempo en crear los caminos del árbol	1,501	1,748	2,456	3,023	5,320
Tiempo en calcular recorte de los nodos hoja	100,693	119,423	142,124	174,571	192,600
Tiempo en guardar en archivos y propagar los valores por el árbol	169,456	259,136	471,336	907,123	5364,970
Tiempo total en calcular los archivos y las envolventes del objeto	**3194,350**	**3467,900**	**3799,080**	**4360,890**	**8932,200**

Tabla 10. Distribución de tiempos, en segundos, para construir el EBP-Octree utilizando la selección de planos con el método de k-mediana.

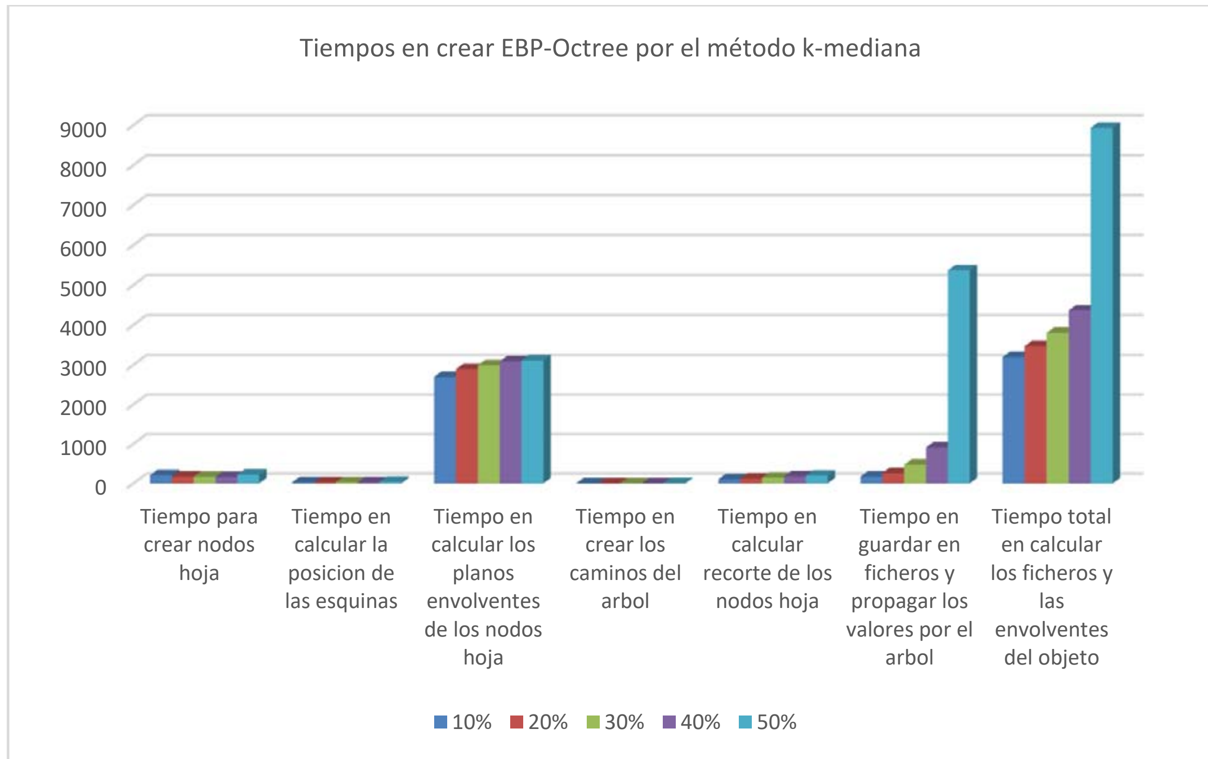


Gráfico 7. Tiempos por etapas, en segundos, para la construcción del EBP-Octree de la Amazona Herida de 28 M. P. seleccionando los planos utilizando el método de k-mediana.

En el *Gráfico 8* se puede apreciar un estudio comparativo de tiempos totales para la construcción de la Amazona Herida de 28 M. P. en cada uno de los métodos y variando el porcentaje en la selección de planos relevantes para el cálculo de la envolvente entre 10 y un 50 por ciento, con un intervalo de 10. De dicho gráfico se puede observar que los tiempos totales de construcción del EBP-Octree en cada uno de los métodos van creciendo conforme se aumenta el porcentaje de selección de planos, relevantes para el cálculo de la envolvente. Un dato relevante es que el tiempo total de cálculo aumenta considerablemente en los tres métodos cuando el porcentaje de planos es del 50%.

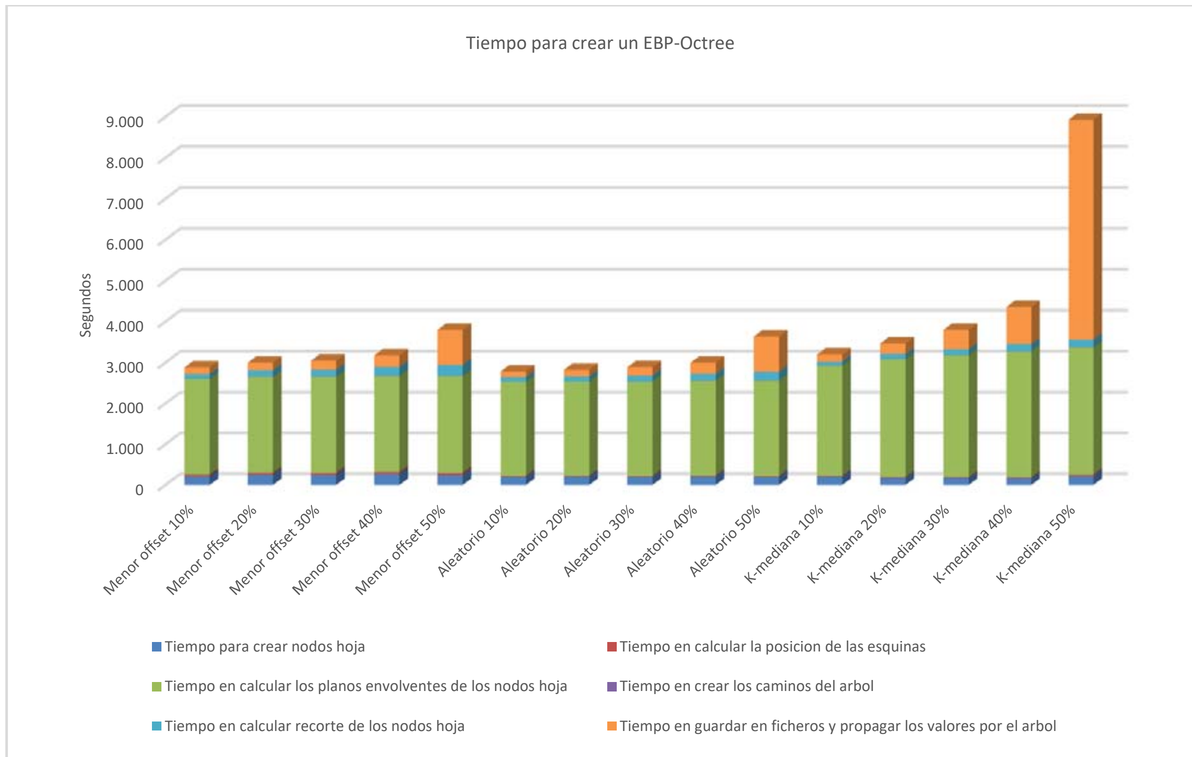


Gráfico 8. Comparación de tiempos totales, en segundos, en construir el EBP-Octree para la Amazona Herida de 28 M. P.

3.6.5. Tiempo de carga del EBP-Octree en memoria principal.

Como se puede apreciar en la *Tabla 11* y en el *Gráfico 9* los tiempos que tardan en cargase los EBP-Octree en memoria principal son muy parecidos, variando estos en unos cuantos milisegundos, aumentando en unas décimas de segundo conforme se aumenta el porcentaje de planos seleccionados.

	10%	20%	30%	40%	50%
Menor offset	3,22233	3,33180	3,53939	3,71713	3,80228
Aleatorio	3,31208	3,45966	3,54790	3,63529	3,75913
K-mediana	3,38389	3,40522	3,66873	3,66060	4,12781

Tabla 11. Tiempos de carga, en segundos, del EBP-Octree en memoria principal.

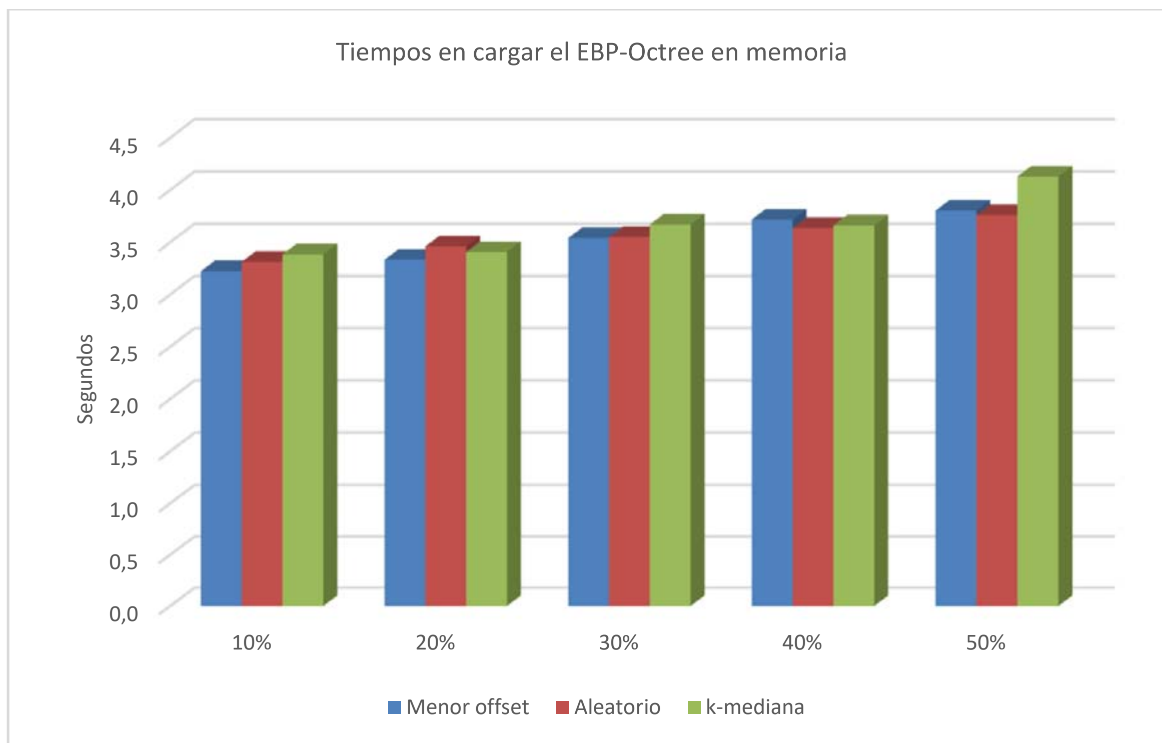


Gráfico 9. Tiempos de carga, en segundos, del EBP-Octree en memoria principal.

3.6.6. Volumen por nivel de las envolventes calculadas.

El estudio realizado en este caso consistió en calcular el volumen de la envolvente para cada uno de los tres métodos, en cada uno de los niveles del EBP-Octree y variando el porcentaje de selección de planos. Todos los cálculos se hicieron sobre el modelo de la Amazona Herida de 28 millones de polígonos. Los datos obtenidos se muestran en porcentaje de ocupación de la envolvente de un nivel con respecto al volumen de ocupación del modelo real. De esta manera se muestran los datos de tal forma que permiten comparar unos con otros, siendo los datos mejores cuando más se aproximen al 100%. En la *Tabla 12* se pueden ver los datos obtenidos cuando se seleccionan los planos con menor offset, en la *Tabla 13* cuando se hace una selección de planos aleatorio y en la *Tabla 14* cuando la selección de planos se realiza utilizando el método de k-mediana. En el *Gráfico 10*, *Gráfico 11* y *Gráfico 12* se puede ver gráficamente una comparación de los volúmenes envolventes obtenidos para cada uno de los métodos de selección de planos, observándose que, para todos los niveles, conforme se aumenta el porcentaje de planos seleccionados el volumen de la

envolvente se aproxima mejor al volumen real del modelo.

Volumen envolvente %	10%	20%	30%	40%	50%
nivel 1	659,137%	607,017%	623,215%	422,045%	323,866%
nivel 2	648,229%	612,117%	542,991%	337,943%	261,385%
nivel 3	374,227%	375,700%	323,506%	228,542%	173,958%
nivel 4	223,876%	225,309%	198,493%	154,962%	129,777%
nivel 5	150,981%	151,582%	137,099%	119,474%	111,360%
nivel 6	119,096%	118,908%	112,874%	107,397%	104,611%
nivel 7	106,678%	106,544%	104,155%	102,520%	101,698%
nivel 8	102,204%	102,065%	101,230%	100,778%	100,545%
nivel 9	100,663%	100,559%	100,324%	100,218%	100,157%
nivel 10	100,174%	100,123%	100,075%	100,055%	100,041%
nivel 11	100,031%	100,020%	100,015%	100,013%	100,012%

Tabla 12. Tanto por ciento de ocupación de las envolventes con respecto al volumen real del modelo. Datos obtenidos utilizando el método de selección de los planos con menor offset.

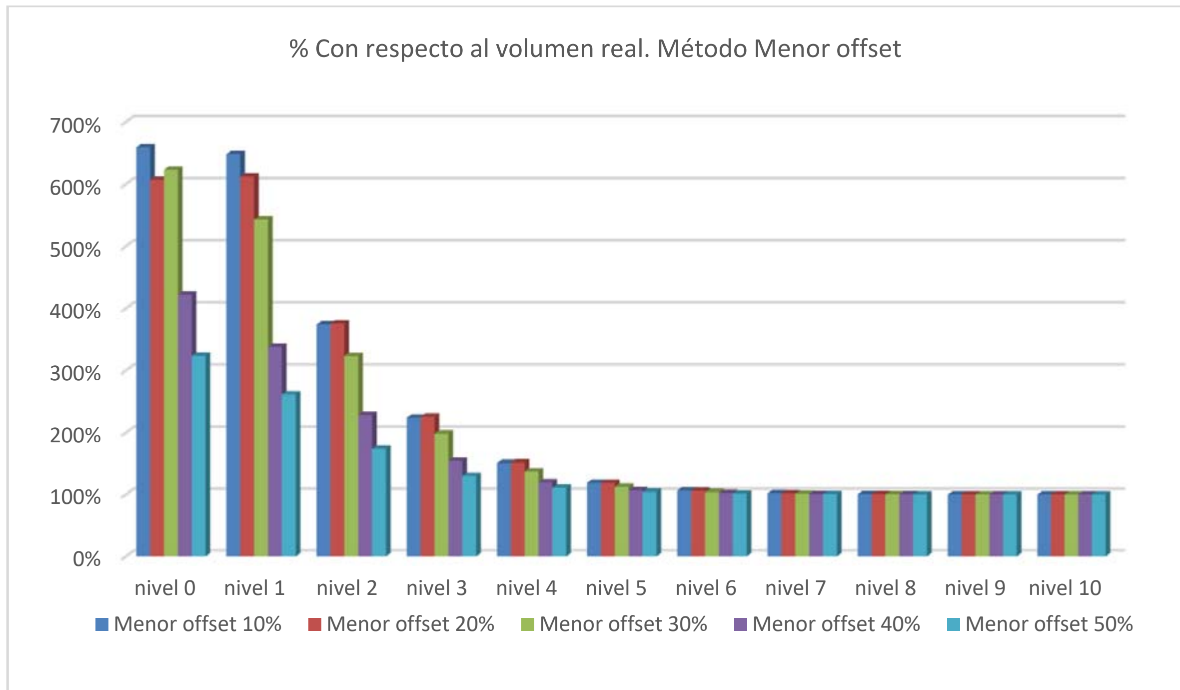


Gráfico 10. Porcentaje de volumen de la envolvente con respecto al volumen real de la Amazona Herida de 28 M. P. seleccionando los planos con menor offset.

Volumen envolvente %	10%	20%	30%	40%	50%
nivel 1	659,890%	653,335%	383,790%	321,109%	306,970%
nivel 2	615,540%	588,945%	424,922%	278,184%	234,506%
nivel 3	385,375%	377,095%	266,533%	174,758%	138,257%
nivel 4	223,816%	224,605%	165,712%	128,606%	117,620%
nivel 5	149,560%	148,442%	125,544%	111,529%	107,373%
nivel 6	118,511%	118,305%	109,138%	105,236%	103,678%
nivel 7	106,780%	106,577%	103,178%	101,996%	101,460%
nivel 8	102,312%	102,097%	101,015%	100,679%	100,500%
nivel 9	100,727%	100,556%	100,297%	100,210%	100,156%
nivel 10	100,197%	100,125%	100,078%	100,060%	100,046%
nivel 11	100,036%	100,023%	100,018%	100,015%	100,013%

Tabla 13. Tanto por ciento de ocupación de las envolventes con respecto al volumen real del modelo. Datos obtenidos utilizando el método de selección de los planos de manera aleatoria.

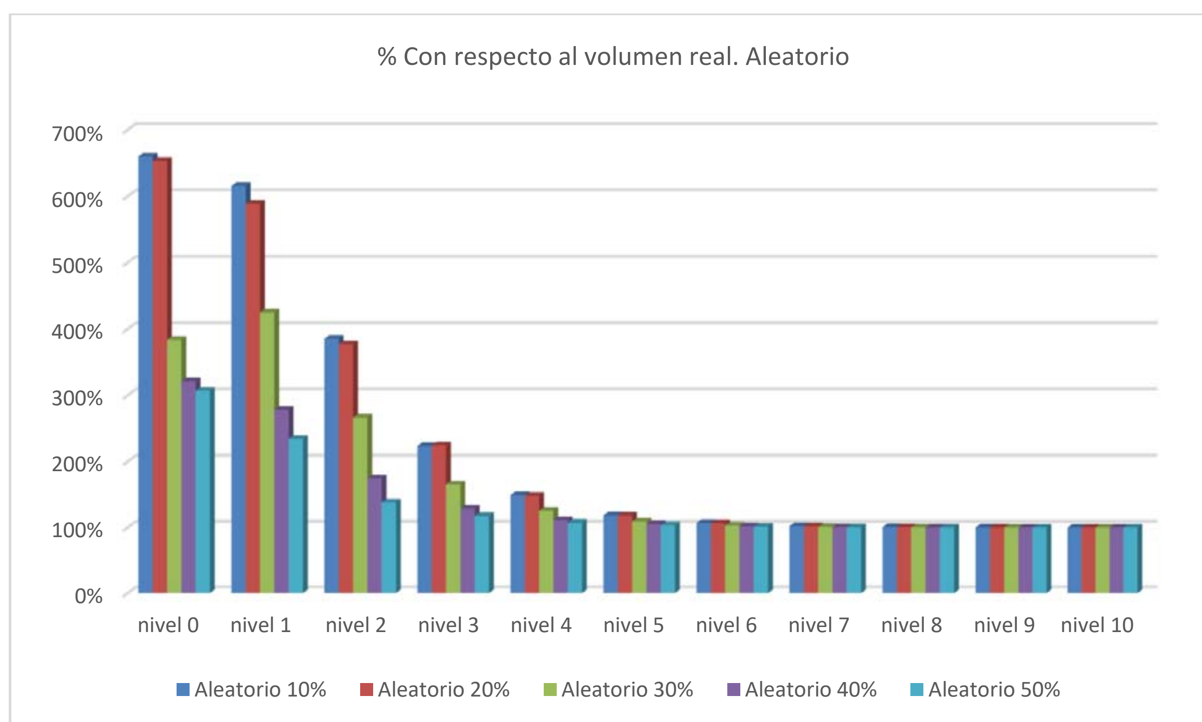


Gráfico 11. Porcentaje de volumen de la envolvente con respecto al volumen real de la Amazona Herida de 28 M. P. seleccionando los planos de manera aleatoria.

Volumen envolvente %	10%	20%	30%	40%	50%
nivel 1	617,006%	656,256%	517,152%	316,623%	291,294%
nivel 2	550,897%	576,428%	375,257%	256,464%	237,828%
nivel 3	337,620%	332,450%	221,586%	167,970%	134,476%
nivel 4	201,537%	194,528%	143,174%	123,991%	116,783%
nivel 5	137,076%	135,024%	116,283%	109,782%	107,299%
nivel 6	113,740%	112,803%	106,387%	104,376%	103,514%
nivel 7	105,055%	104,456%	102,309%	101,683%	101,359%
nivel 8	101,654%	101,361%	100,763%	100,575%	100,464%
nivel 9	100,493%	100,367%	100,232%	100,181%	100,144%
nivel 10	100,124%	100,088%	100,064%	100,052%	100,042%
nivel 11	100,025%	100,019%	100,015%	100,013%	100,012%

Tabla 14. Tanto por ciento de ocupación de las envolventes con respecto al volumen real del modelo. Datos obtenidos utilizando el método de selección de los planos k-mediana.

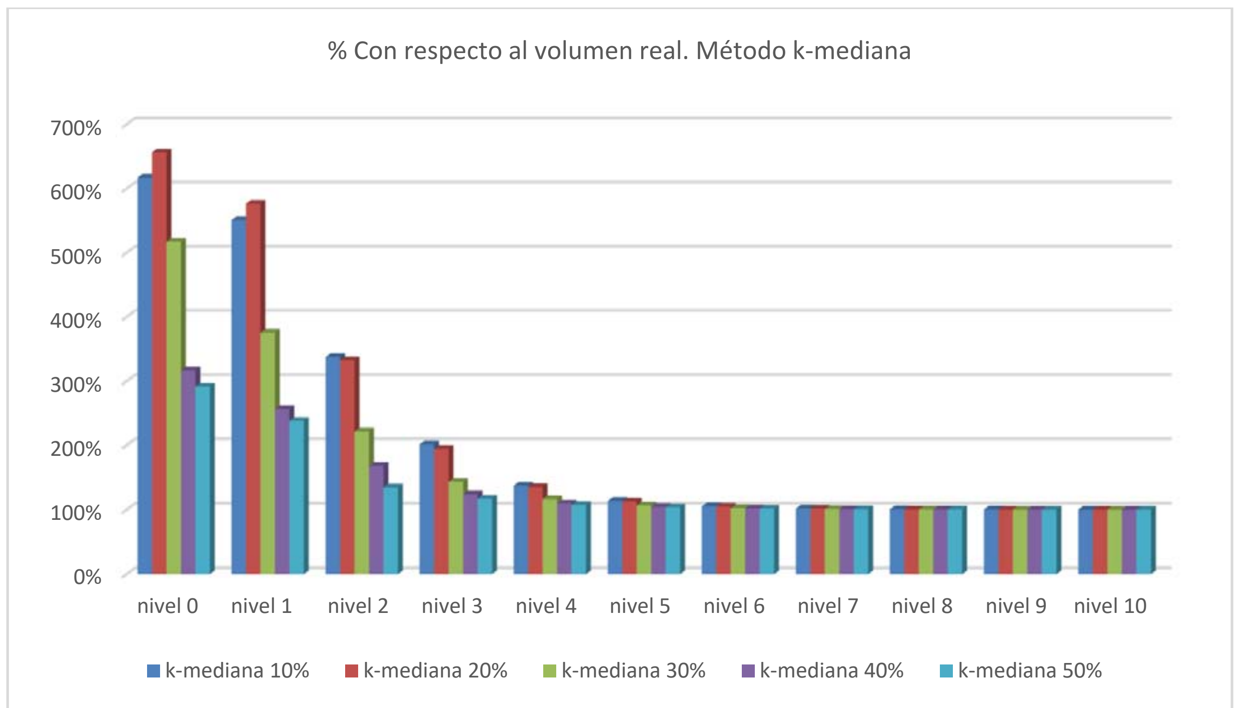


Gráfico 12. Porcentaje de volumen de la envolvente con respecto al volumen real de la Amazona Herida de 28 M. P. seleccionando los planos con el método de k-mediana.

En el *Gráfico 13* y *Gráfico 14* se muestra una comparación de los volúmenes envolventes obtenidos en cada uno de los niveles para los tres métodos de selección de planos. En cada uno de los gráficos se representan los volúmenes cuando se seleccionan el mismo porcentaje de planos. En estos gráficos se puede observar que el método de selección de planos k-mediana es el que mejores resultados proporciona, ya que los volúmenes envolventes obtenidos con este método son, en casi todos los casos, más pequeños que los volúmenes obtenidos con los otros dos métodos. Otro factor que se aprecia es que los volúmenes envolventes para los métodos de selección de planos con menor offset y aleatorio son muy parecidos en la mayoría de los casos.

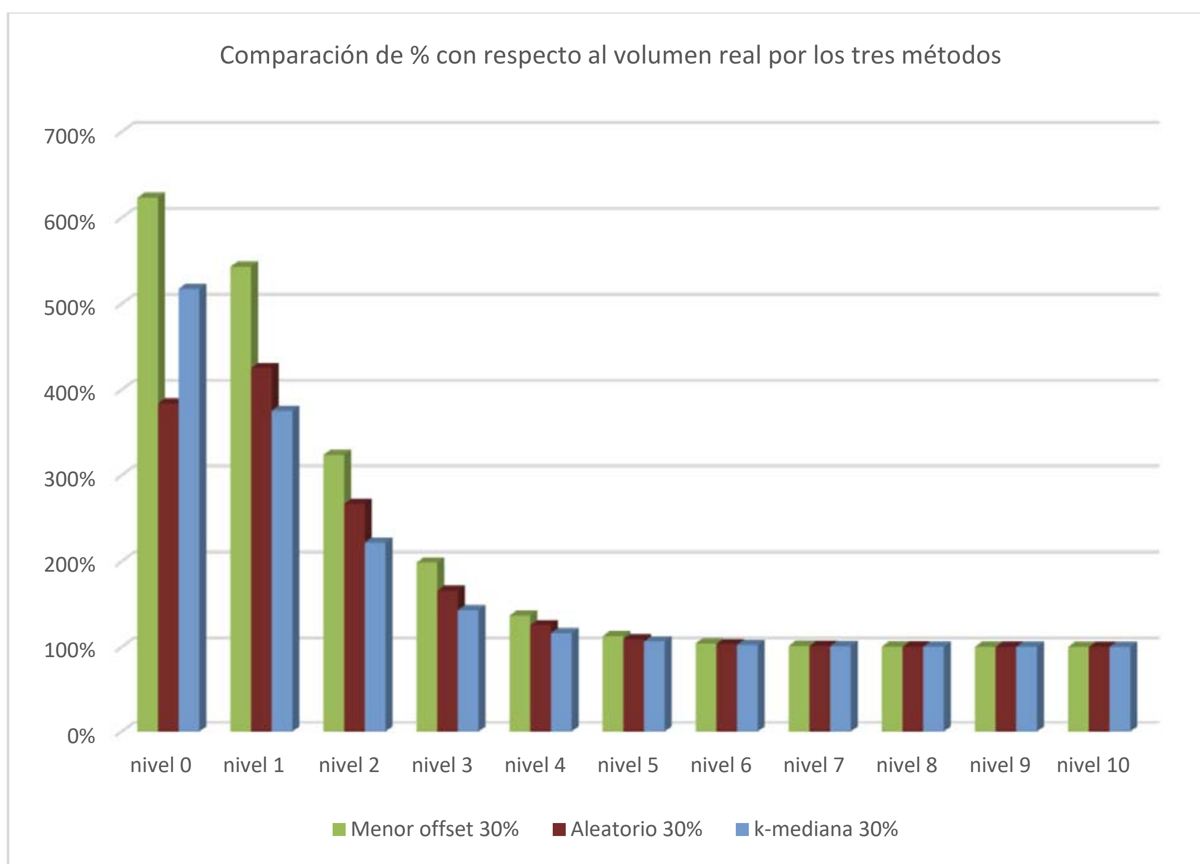


Gráfico 13. Comparación del porcentaje de ocupación del volumen de la envolvente con respecto al volumen del modelo, para los tres métodos, seleccionando un 30% de planos.

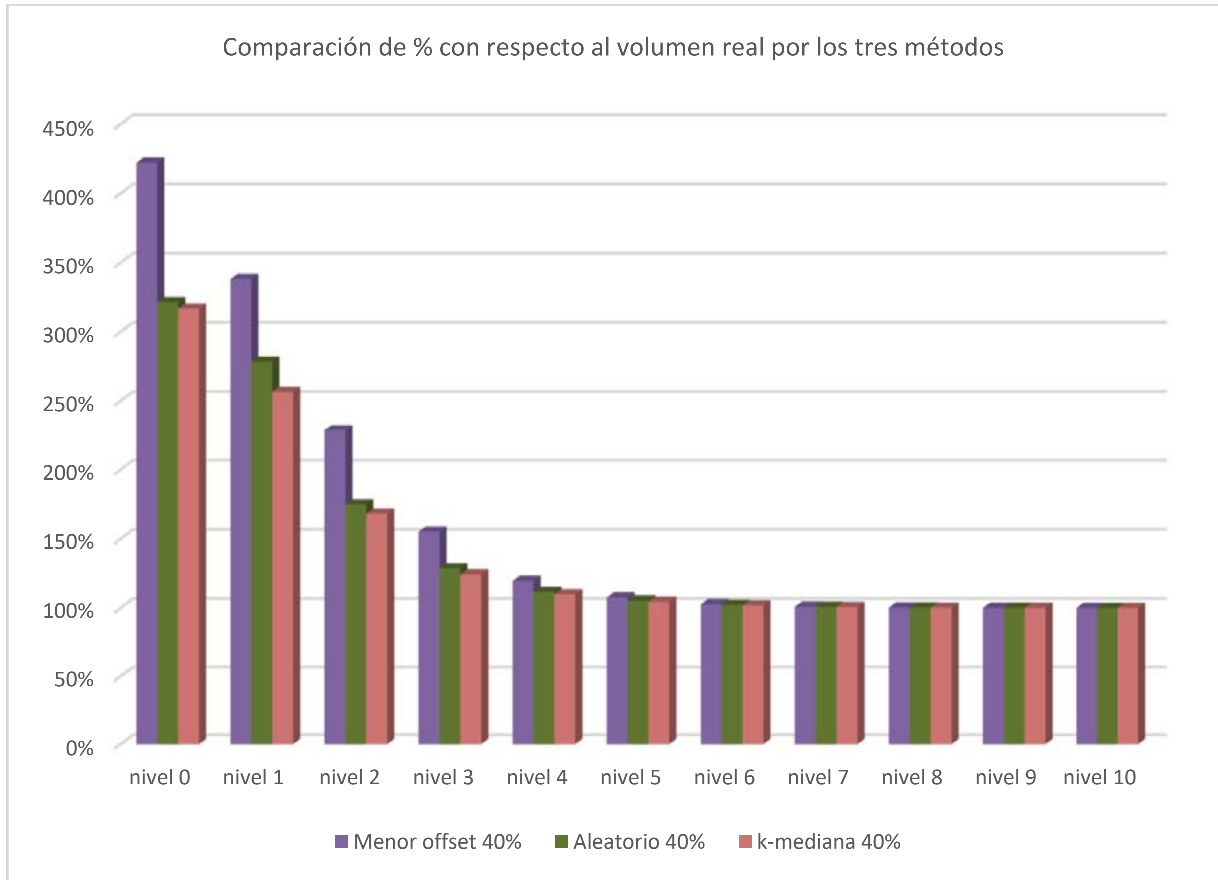


Gráfico 14. Comparación del porcentaje de ocupación del volumen de la envolvente con respecto al volumen del modelo, para los tres métodos, seleccionando un 40% de planos.

En la *Gráfico 15* se muestran las sumas de todas las envolventes generadas con un método y para un porcentaje de planos seleccionados. Aquí se agrupan los resultados por porcentaje de planos seleccionados, observándose que en todos los casos la suma de las envolventes obtenidas por el método de k-mediana para los diferentes porcentajes de planos seleccionados es menor lo que significa que mejor se aproxima al volumen real del modelo.

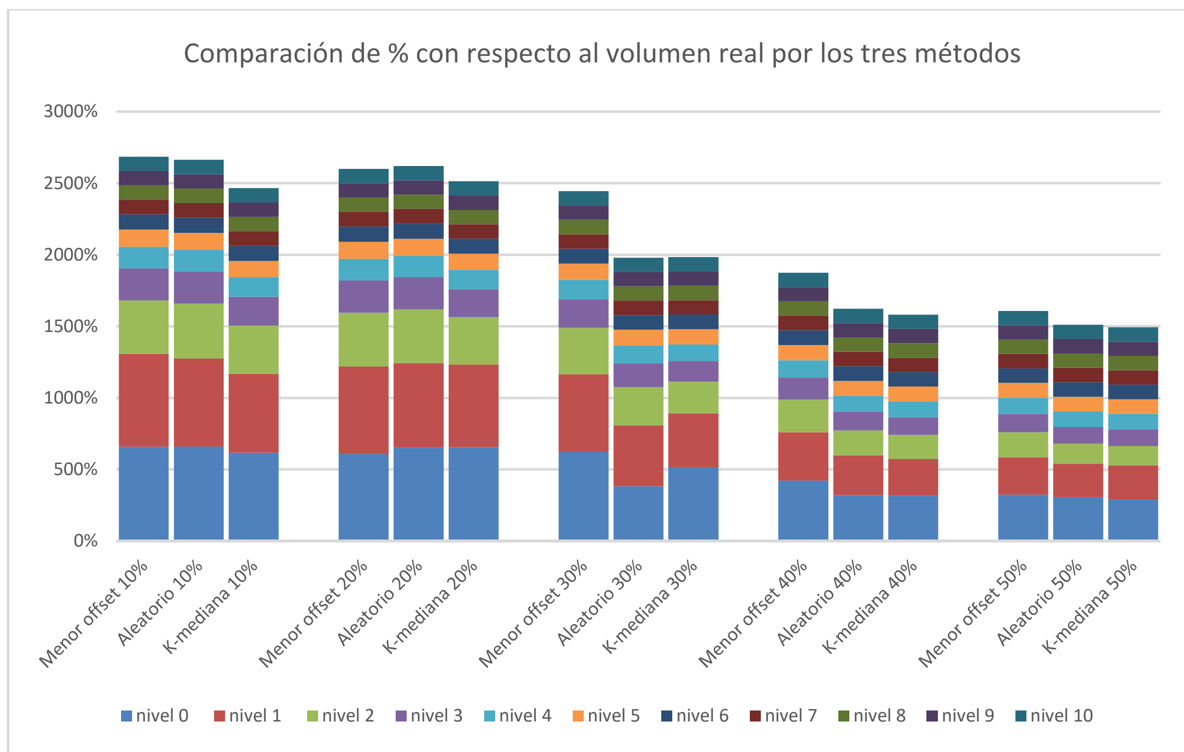


Gráfico 15. Suma de volúmenes envolventes obtenidos para cada uno de los métodos, agrupados por porcentaje de planos seleccionados.

3.6.7. Estudio de los resultados obtenidos.

La elección del método de selección de planos relevantes de la envolvente se ha hecho mediante un estudio empírico de los resultados obtenidos en el modelo de la Amazona Herida formado por 28 millones de polígonos.

Tras realizar las pruebas con los tres métodos de selección de planos relevantes para la construcción de las envolventes de los nodos y variando el porcentaje de planos seleccionados, se pueden observar los siguientes resultados para los tres métodos:

- El método del k-mediana es el que genera una envolvente de volumen menor y visualmente la envolvente generada es muy regular. En este método los tiempos para crear y para cargar en memoria la envolvente son mayores que los obtenidos en los otros métodos.
- En el método del menor offset los resultados que se obtienen se encuentran

entre los otros dos métodos tanto para el volumen de las envolventes como para los tiempos de creación y carga del octree en memoria.

- Por último, el método aleatorio es el más rápido a la hora de seleccionar los planos, pero los resultados en cuestión de volumen y de visualización de la envolvente varían en cada ejecución, no siento esto los más óptimos.

Haciendo un estudio comparativo sobre los datos obtenidos para los cinco parámetros vistos anteriormente, se puede concluir que, de los tres métodos, el que mejores resultados proporciona es el que utiliza una selección de planos mediante el algoritmo de k-mediana. Si bien cabe destacar que este método es el que tiene unos tiempos de construcción del EBP-Octree mayores, pero estos quedan compensados porque con él se obtienen mejores envolventes, lo cual es fundamental en fases posteriores, en las que cuanto mejores sean las envolventes, los resultados proporcionados serán mucho mejores ya que evitarán muchos falsos positivos. Con respecto al tiempo de carga del EBP-Octree en memoria la variación entre los tres métodos es de décimas de segundo. Estos costes se pueden asumir sin problema, ya que la ganancia en los resultados posteriores será mucho mayor.

Con respecto al porcentaje de planos que se deben seleccionar, viendo los resultados obtenidos queda claro que deben de estar comprendidos entre el 30 o el 40 por ciento, ya que superar estos valores hace que los tiempos de construcción y carga en memoria del EBP-Octree aumenten considerablemente y sin embargo la variación en los volúmenes de las envolventes no sea muy significativa.

Para decidir el porcentaje de planos que se deben seleccionar se realizó un estudio cuyos datos se pueden ver en el siguiente apartado.

3.7. Elección del porcentaje de planos relevantes para el cálculo de las envolventes.

De los resultados obtenidos en el apartado anterior se deduce que de los tres métodos estudiados el mejor es el que utiliza el algoritmo del k-mediana.

Lo que falta por decidir es el porcentaje de planos óptimo que se debe seleccionar para el cálculo del EBP-Octree. Para ello se realizo un estudio empírico comparativo utilizando tres modelos de gran tamaño, que se van a calcular con el método de selección de planos que utiliza el algoritmo de k-mediana, tomando como porcentaje

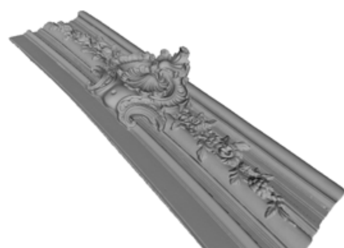
un 30% y un 40% de estos. Los tres modelos elegidos para este estudio son: Amazona Herida formada por 28 millones de polígonos, Moldura formada por 26 millones de polígonos y Lucy formada por 28 millones de polígonos.

En este estudio se han tenido en cuenta los mismos parámetros que se utilizaron en la selección del método de elección de planos relevantes. Estos parámetros son:

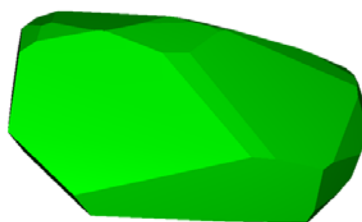
- Percepción visual de los resultados gráficos de las envolventes generadas para cada nivel.
- Tamaño en disco de los archivos generados.
- Tamaño del EBP-Octree por niveles.
- Tiempo de construcción del EBP-Octree.
- Tiempo de carga en memoria principal del EBP-Octree.
- Volumen de las envolventes generadas por niveles.

3.7.1. Resultados visuales por niveles de las envolventes obtenidas.

Entre las ilustraciones *Figura 61* y la *Figura 75* se pueden ver las envolventes obtenidas en los niveles 4, 6 y 9 para la Amazona Herida de 28 M. P., calculadas con la selección de planos utilizando el algoritmo k-mediana y tomando el 30% y el 40% de los planos. Entre las ilustraciones *Figura 76* y la *Figura 81* se pueden ver las envolventes obtenidas y de un detalle para los modelos de la Moldura de 26 M. P. y el de Lucy de 28 M. P., calculadas utilizando el método del k-mediana y tomando el 30% y el 40% de los planos.

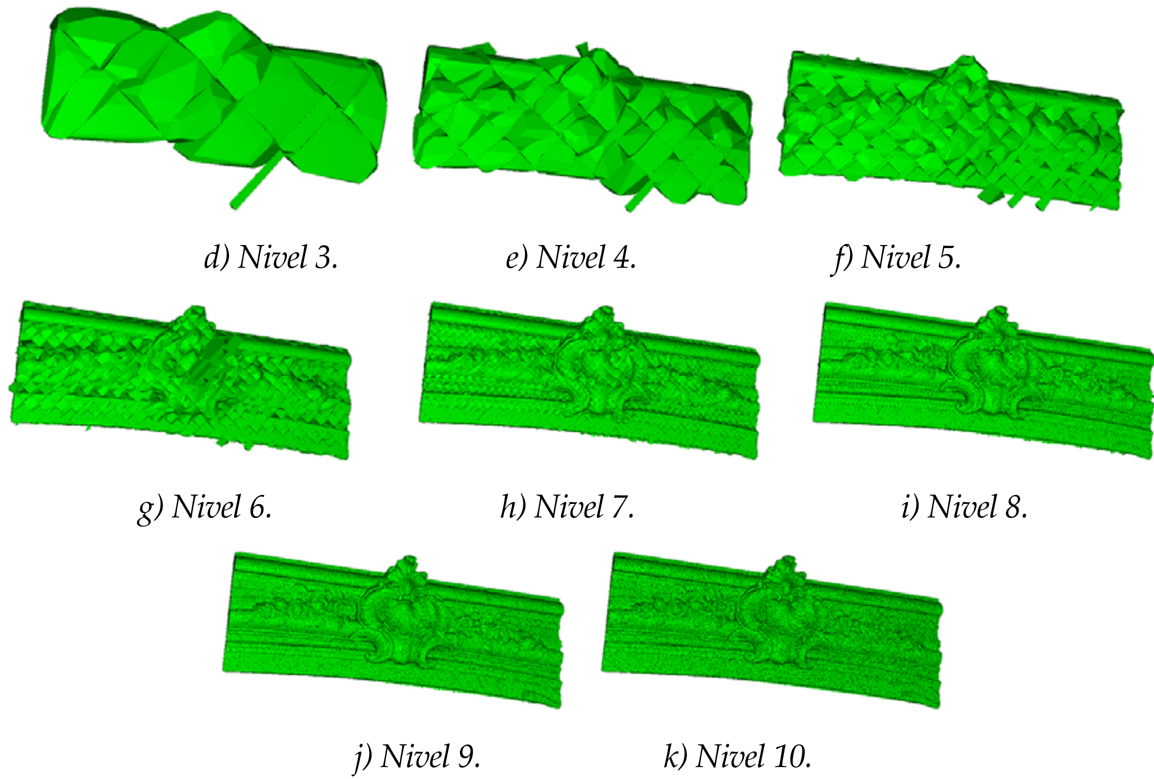


a) Real.



b) Nivel 1.

c) Nivel 2.

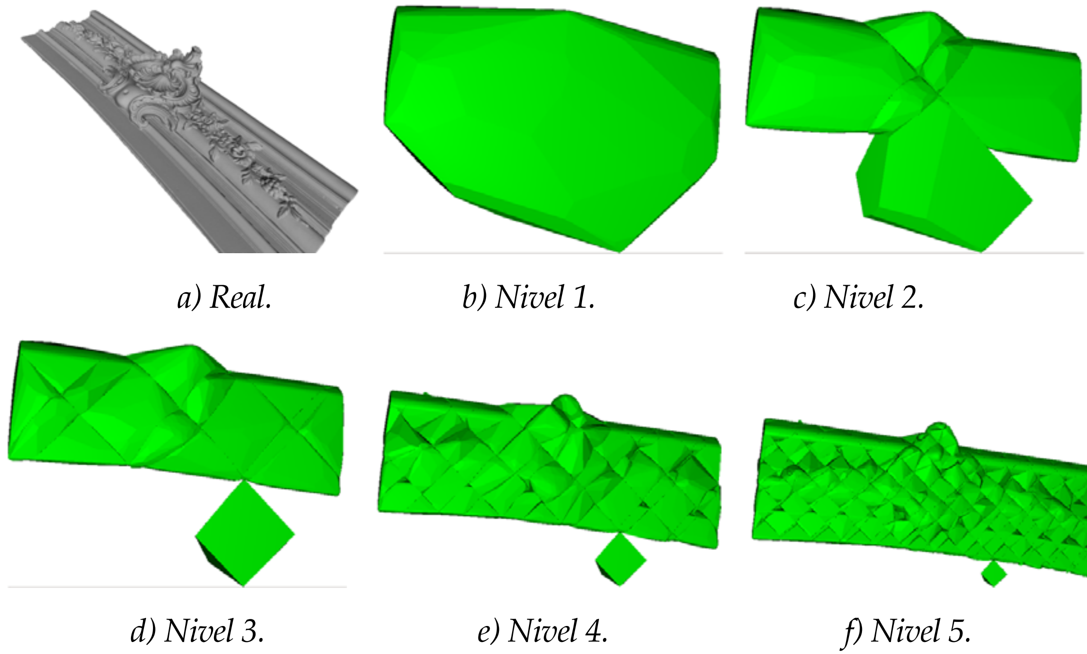


d) Nivel 3. *e) Nivel 4.* *f) Nivel 5.*

g) Nivel 6. *h) Nivel 7.* *i) Nivel 8.*

j) Nivel 9. *k) Nivel 10.*

Figura 76. Moldura de 26 M.P. obtenida con el método de selección de planos k-mediana y tomando el 30% de los planos.



a) Real. *b) Nivel 1.* *c) Nivel 2.*

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

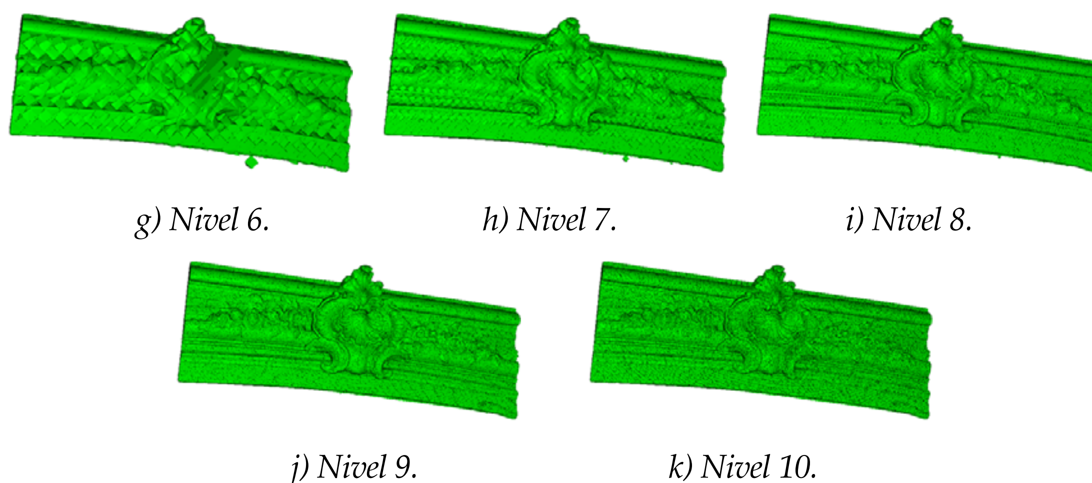


Figura 77. Moldura de 26 M.P. obtenida con el método de selección de planos k-mediana y tomando el 40% de los planos.

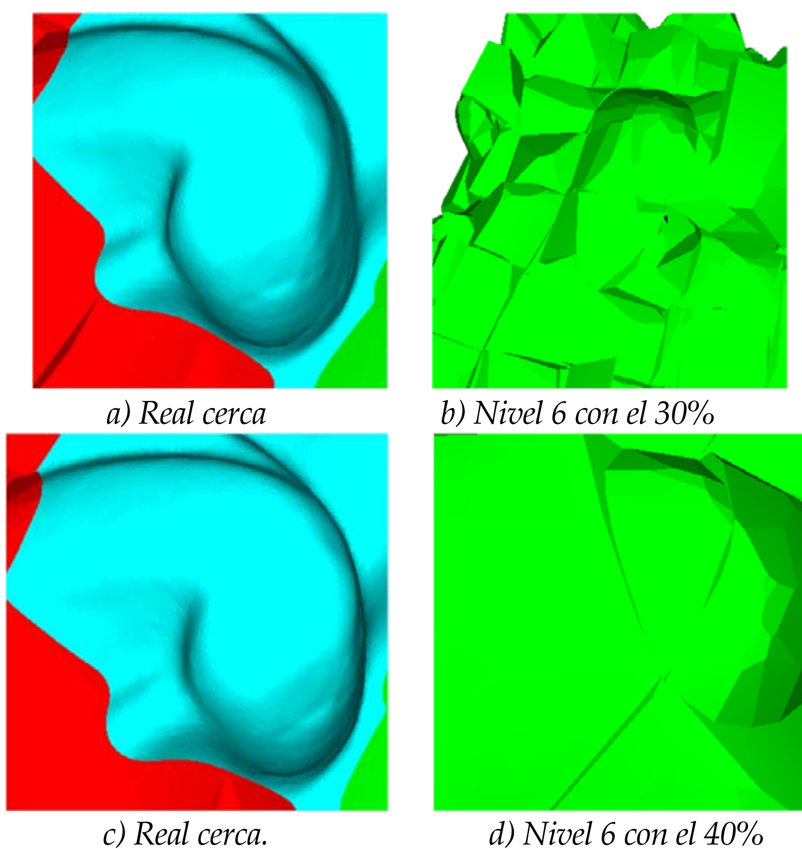
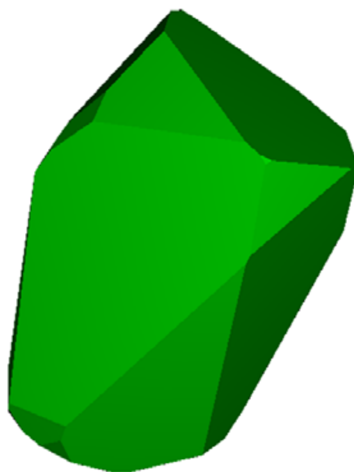
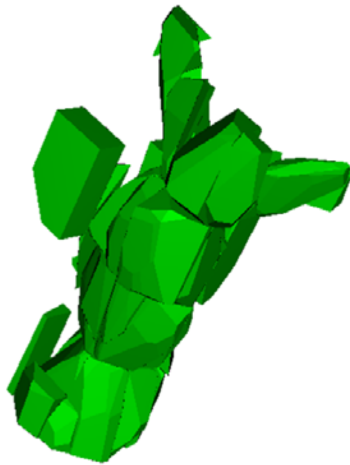


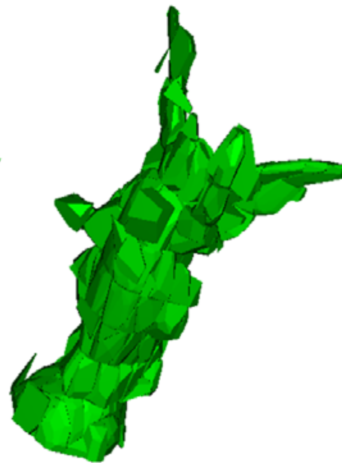
Figura 78. Detalle de la moldura de 26 M.P. a nivel 6 obtenido con el método de selección de planos k-mediana y tomando el 30% de los planos.

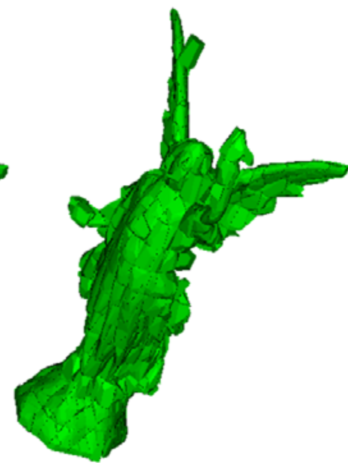
a) Real.

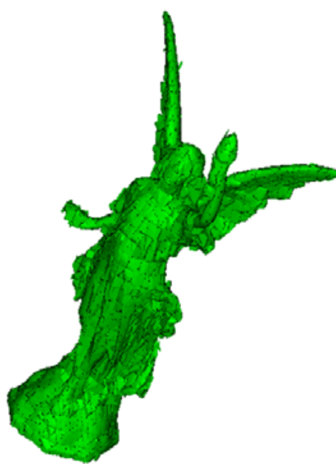

b) Nivel 1.

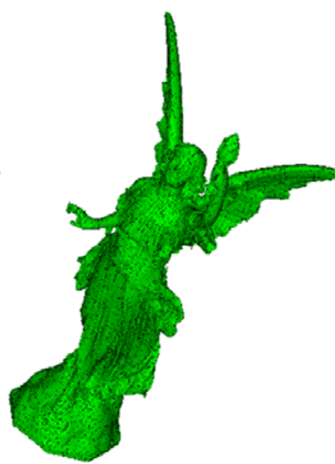
c) Nivel 2.

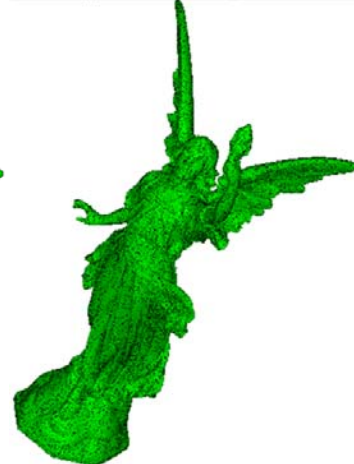

d) Nivel 3.

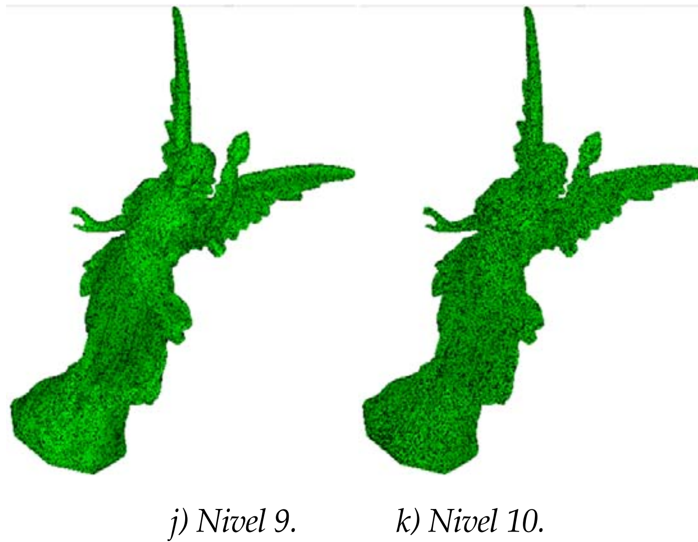

e) Nivel 4.


f) Nivel 5.


g) Nivel 6.

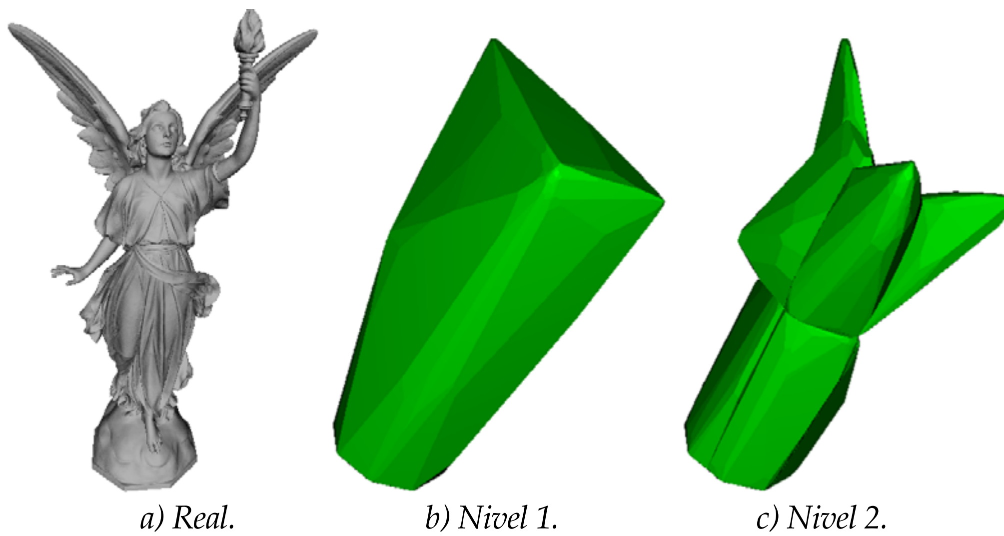

h) Nivel 7.


i) Nivel 8.



j) Nivel 9. *k) Nivel 10.*

Figura 79. Lucy de 28 M.P. obtenida con el método de selección de planos k-mediana y tomando el 30% de los planos.



a) Real. *b) Nivel 1.* *c) Nivel 2.*

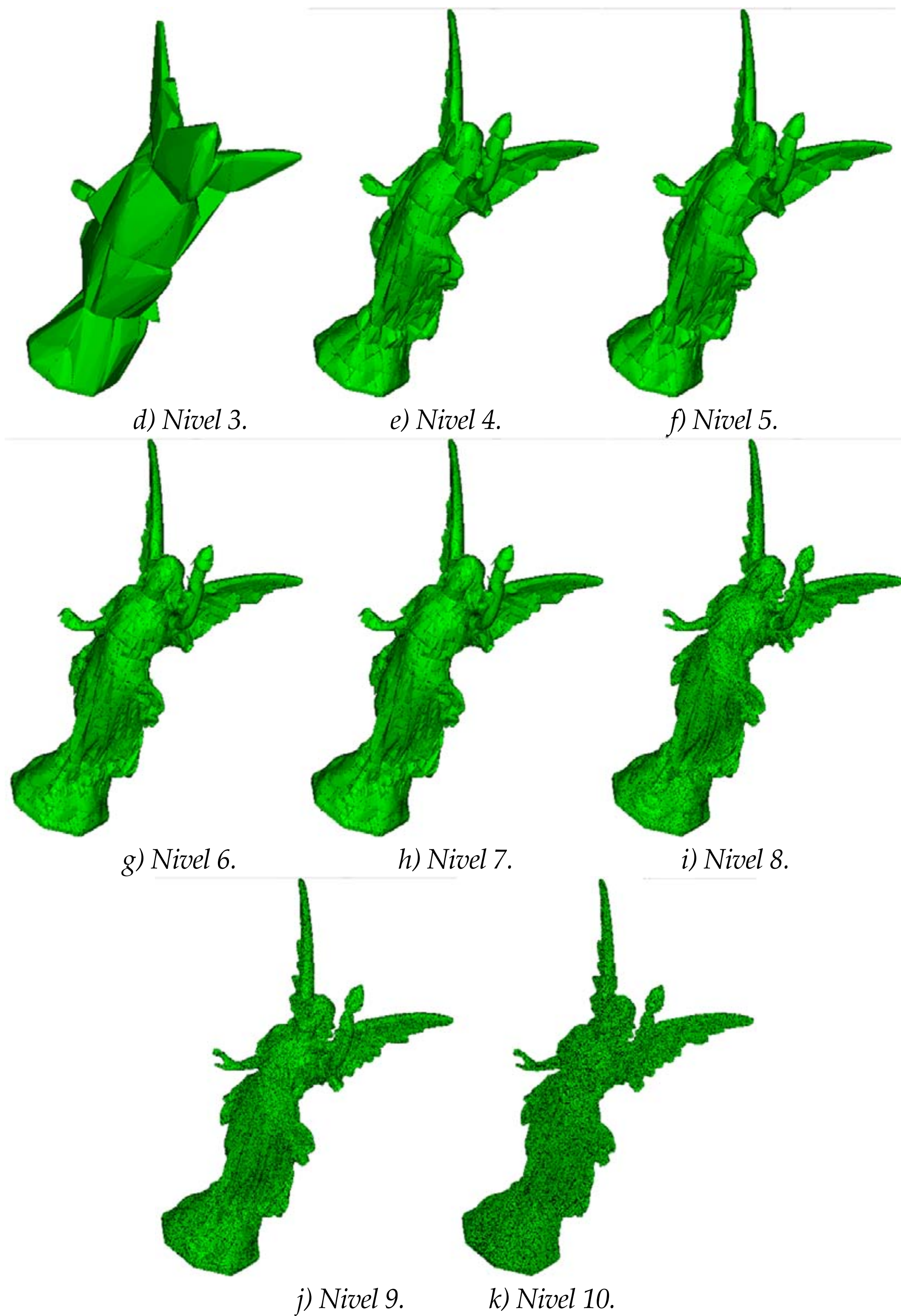
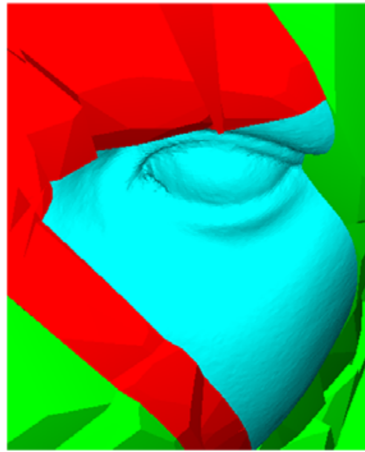
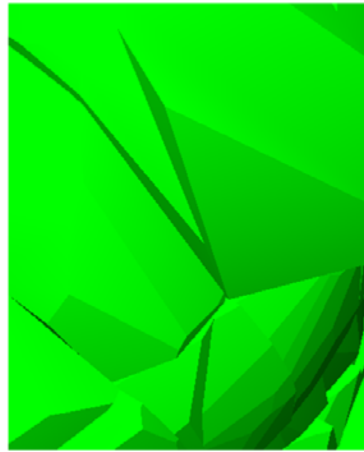


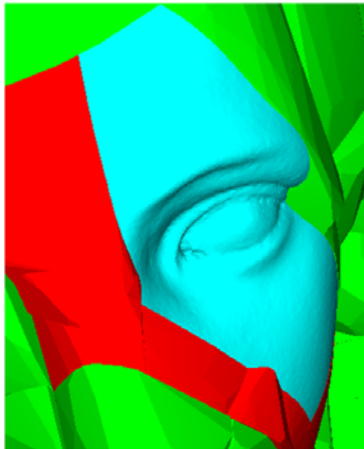
Figura 80. Lucy de 28 M.P. obtenida con el método de selección de planos k-mediana y tomando el 40% de los planos.



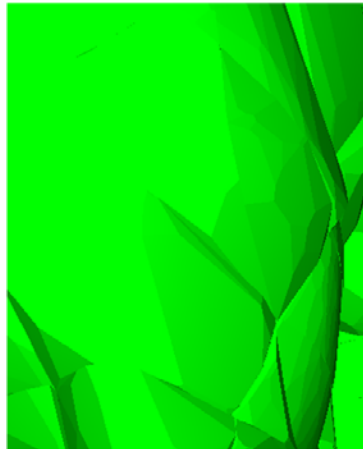
a) Real cerca.



b) Nivel 6 con el 30%.



c) Real cerca.



d) Nivel 6 con el 40%.

Figura 81. Detalle del ojo de Lucy de 28 M.P. a nivel 6 obtenido con el método de selección de planos k-mediana y tomando el 30% y 40% de los planos.

Visualmente se puede observar que las envolventes generadas utilizando un porcentaje mayor de los planos se aproximan más al modelo real, tal y como se podría de esperar.

3.7.2. El EBP-Octree en disco.

En la *Tabla 15* se muestran los resultados obtenidos para los tamaños de los ocho archivos generados para cada modelo, utilizando el método de k-mediana y seleccionando el 30% y el 40% de los planos.

Bytes	Moldura 30%	Moldura 40%	Lucy 30%	Lucy 40%	Amazona 30%	Amazona 40%
.ply	514.617.344		533.061.632		576.040.960	
.bpl	356.478.048	503.975.296	524.051.648	725.289.792	572.177.440	783.562.656
.bpo	851.817.936	851.817.936	897.783.824	897.783.824	899.185.040	899.185.040
.geo	415.843.744	415.843.744	561.096.868	561.096.868	607.737.280	607.737.280
.nv	168.562.084	168.562.084	168.334.468	168.334.468	168.597.100	168.597.100
.oct	50.050.198	50.050.198	104.140.558	104.140.558	127.479.154	127.479.154
.tgl	212.954.464	212.954.464	224.445.936	224.445.936	224.796.240	224.796.240
.vtx	1.916.590.176	1.916.590.176	2.020.013.424	2.020.013.424	2.023.166.160	2.023.166.160
Tamaño colisiones	**3.972.296.650**	**4.119.793.898**	**4.499.866.726**	**4.701.104.870**	**4.623.138.414**	**4.834.523.630**
.bvp	4.043.661.804	5.012.208.948	7.009.761.492	8.618.757.732	7.831.076.292	9.475.811.652
Tamaño rendering	**8.015.958.454**	**9.132.002.846**	**11.509.628.218**	**13.319.862.602**	**12.454.214.706**	**14.310.335.282**

Tabla 15. Tamaño en Bytes de los archivos generados.

Se puede observar que los únicos que modifican su tamaño son los ".bpl" y los ".bvp". En estos dos tipos de archivos se puede observar que, para los tres modelos, el aumento de los archivos ".bpl" obtenidos seleccionando el 40% de los planos, con respecto al tamaño de los obtenidos seleccionando el 30% de los planos ronda más o menos el 140%, mientras que para los archivos ".bvp" el aumento de los archivos es de aproximadamente del 120%. Cabe destacar que los archivos ".bvp" no se utilizan para el cálculo de las colisiones. Solamente son necesarios cuando se quieren visualizar las envolventes de los modelos.

En la *Gráfico 16* se muestran las sumas de todos los tamaños en miles de millones de Bytes, de los archivos obtenidos con los tres modelos seleccionando el 30% y el 40% de los planos. De este gráfico se puede deducir que el aumento del tamaño total de los archivos está entre un 110 y un 115 %, dependiendo del modelo.

De estos resultados se puede concluir que el coste de seleccionar el 40% de los planos es asumible, ya que el incremento total de los archivos no llega al 115%. Pero el utilizar un porcentaje mayor de planos hace que en los cálculos se aumente considerablemente el rendimiento aumentando así la eficiencia de los algoritmos utilizados.

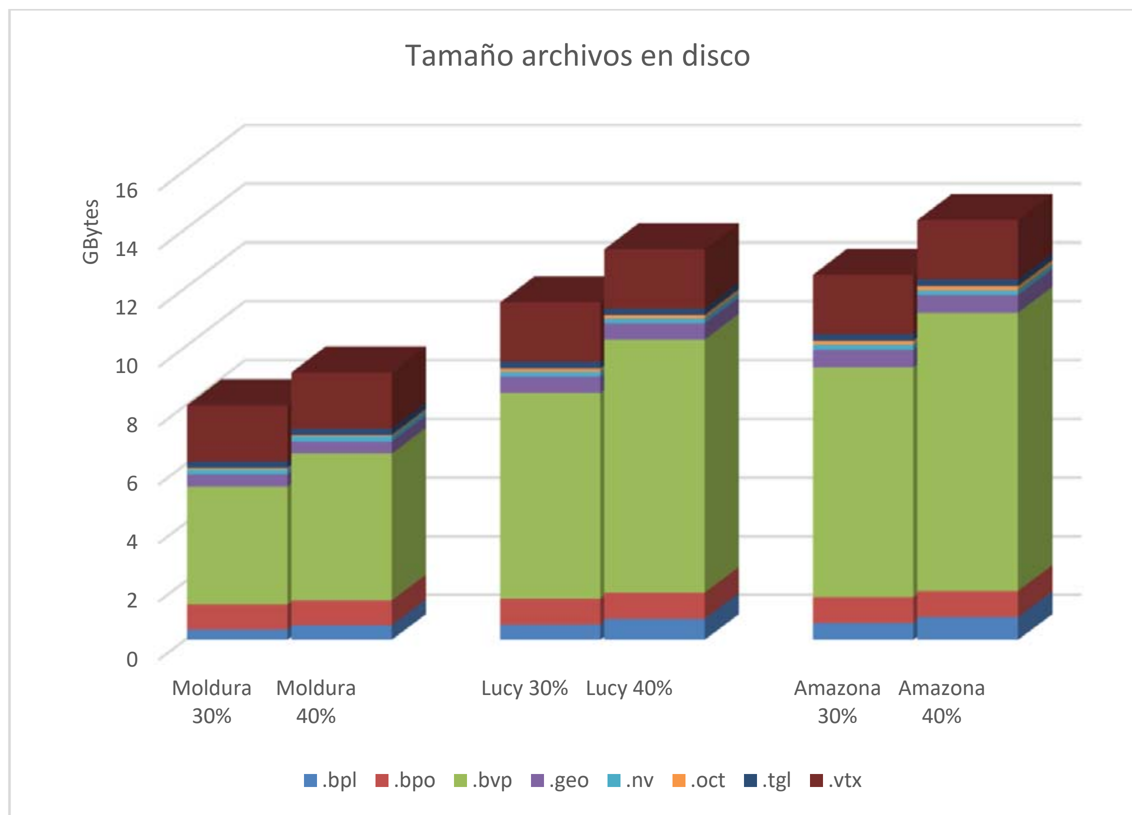


Gráfico 16. Suma de la ocupación de los archivos en disco para los tres modelos calculados por el método de k-mediana y seleccionado el 30% y el 40% de los planos.

3.7.3. Tamaño del EBP-Octree por niveles.

Un dato fundamental es el parámetro que da el tamaño de ocupación del EBP-Octree en memoria, ya que gracias a él es posible conocer cuántos modelos se pueden incluir en una escena sin que se desborde la memoria del ordenador con el que se está trabajando. Los datos de ocupación, por niveles, para los tres modelos estudiados y variando el porcentaje de planos seleccionados se pueden ver en la *Tabla 16*.

Mbyte	Moldura 30%	Moldura 40%	Lucy 30%	Lucy 40%	Amazona 30%	Amazona 40%
Nivel 1	0,0083	0,1059	0,0074	0,1233	0,0115	0,1896
Nivel 2	0,0289	0,2687	0,0392	0,3749	0,0401	0,3969
Nivel 3	0,0837	0,5947	0,1284	0,9540	0,1339	0,9763
Nivel 4	0,2811	1,4857	0,4565	2,6511	0,5075	2,9026
Nivel 5	1,0801	4,2483	1,7055	7,1530	1,9749	8,3813
Nivel 6	4,1034	12,4750	6,7139	20,7789	7,8886	24,0540
Nivel 7	31,2393	75,7122	52,5689	127,1625	59,5957	139,4414
Nivel 8	118,6366	232,9418	203,3316	391,9255	226,2082	416,7158
Nivel 9	444,0675	713,5514	773,1411	1215,5654	857,5340	1288,3785
Nivel 10	1670,3163	2244,8251	2935,8252	3877,1204	3275,4829	4196,1926
Nivel 11	5890,7841	6908,2478	10277,4975	11955,4609	11507,0683	13292,1041

Tabla 16. Ocupación en Mbyte del EBP-Octree por niveles.

Analizando los datos de la *Tabla 16* se observa que el tamaño de ocupación en memoria principal de todo el EBP-Octree supera la capacidad de memoria principal de la mayoría de ordenadores actuales. Por este motivo solamente se almacena en memoria hasta el nivel de corte, haciendo que se pueda trabajar con los EBP-Octree sin ningún problema.

Otros datos relevantes que se pueden obtener de la *Tabla 16* son la ocupación en memoria de la parte cargada del EBP-Octree, el tamaño medio de los nodos que se mantienen en disco y el tamaño medio de los subárboles. Todos estos datos se resumen en la *Tabla 17*.

Kbyte	Moldura 30%	Moldura 40%	Lucy 30%	Lucy 40%	Amazona 30%	Amazona 40%
Ocupación árbol raíz en memoria	1517,7100	6864,1700	2393,0400	11526,4800	2731,8700	13154,9400
Tamaño medio nodos disco	1,4028	1,7516	1,1573	1,4285	1,0564	1,2833
Tamaño medio de los subárboles	4465,5087	5575,7666	3676,2550	4537,6986	3132,9087	3805,9623

Tabla 17. Datos relevantes con respecto a la ocupación de memoria del EBP-Octree.

En la *Gráfico 17* se puede ver una comparativa de la ocupación en memoria de los EBP-Octree, observándose que cuanto mayor es el porcentaje de planos seleccionados la ocupación en memoria aumenta en los tres modelos cerca del 500%.

Este aumento es muy considerable, pero si se observan los tamaños de los EBP-Octree calculados con un porcentaje mayor de planos seleccionados, en el peor de los casos es de 12 Megabytes y en el mejor de los casos de 8 Megabytes. Estos valores no son muy elevados con respecto a la memoria principal de la mayoría de los ordenadores actuales.

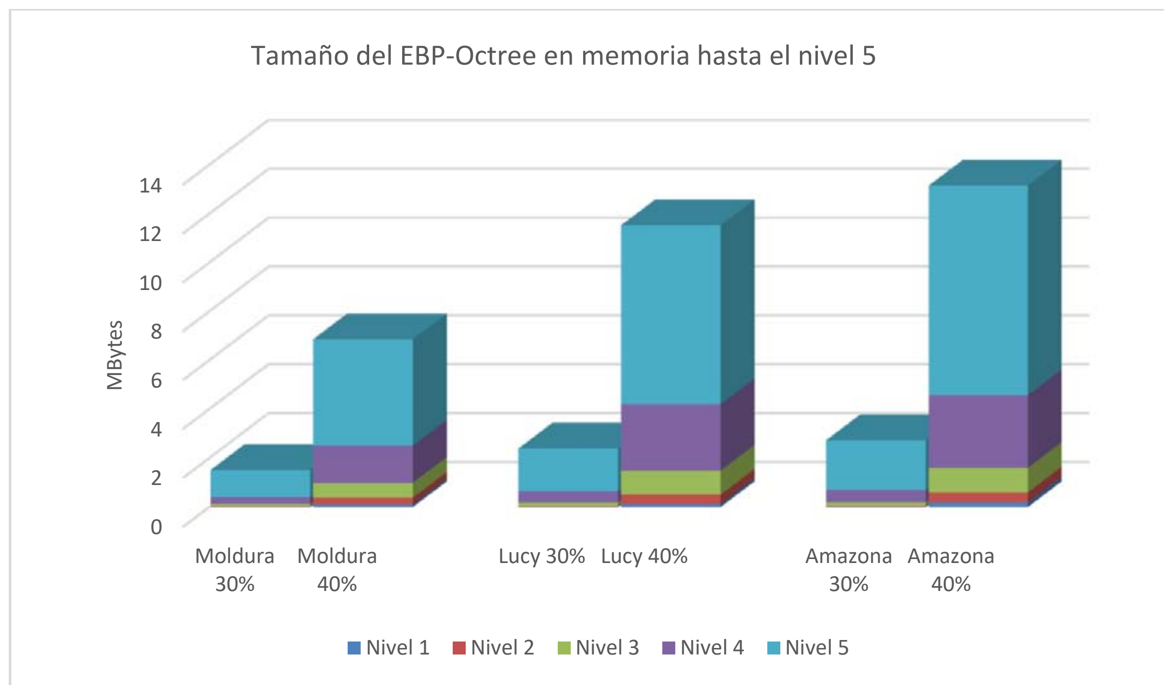


Gráfico 17. Tamaño de los EBP-Octree en memoria.

Los tamaños medios de los nodos y de los subárboles que se mantienen en memoria secundaria, para los tres modelos, el incremento de tamaño para el caso en el que se seleccionan más planos está sobre el 120%. Este incremento no es muy elevado y, por tanto, el coste de trabajar con los modelos más grande es asumible por la mayoría de ordenadores actuales.

3.7.4. Tiempo de creación del EBP-Octree.

En la *Tabla 18* se pueden ver los tiempos, en segundos, de creación del EBP-Octree para los tres modelos utilizando el método de selección de planos del k-mediana y tomando como porcentaje el 30% y el 40% de los planos. Los tiempos de construcción se dividen en etapas, observándose que la etapa que más tiempo consume es la construcción de las envolventes en los nodos hoja. Las etapas que calculan las

envolventes para los nodos hoja y para los nodos intermedios son las que más tiempo consumen.

Segundos	Moldura 30%	Moldura 40%	Lucy 30%	Lucy 40%	Amazona 30%	Amazona 40%
Tiempo para crear nodos hoja	49,100	52,720	115,750	123,160	168,136	163,688
Tiempo en calcular la posición de las esquinas	21,990	22,140	27,990	27,610	25,538	25,260
Tiempo en calcular los planos envolventes nodos hojas	2560,190	2601,240	2414,820	2451,630	2988,900	3086,610
Tiempo en crear los caminos del árbol	0,480	0,520	0,940	1,170	2,456	3,023
Tiempo en calcular recorte de los nodos hoja	73,020	89,680	115,960	131,080	142,124	174,571
Tiempo en guardar en archivos y propagar los valores por el árbol	315,380	745,040	374,800	789,370	471,336	907,123
Tiempo total en calcular los archivos y las envolventes del objeto	**3020,380**	**3511,570**	**3050,690**	**3524,470**	**3799,080**	**4360,890**

Tabla 18. Tiempos, en segundos, en construir el EBP-Octree para los tres modelos utilizando el método del k-mediana y seleccionando el 30% y el 40% de los planos.

Haciendo una comparación de los tiempos totales de construcción de los EBP-Octree para los tres modelos, se puede destacar que el incremento de tiempo para cada uno de los modelos cuando se seleccionan el 40% de los planos en vez del 30% ronda aproximadamente el 15%. El coste en el incremento de tiempo en el cálculo es asumible, ya que no es un coste muy excesivo y más teniendo en cuenta que los EBP-Octree sólo se construyen una única vez.

3.7.5. Tiempo de carga del EBP-Octree en memoria.

Como se puede ver en la *Tabla 19* los tiempos que tardan en cargarse en memoria principal los EBP-Octree son muy parecidos, apreciándose un ligero incremento en el tiempo para los modelos que han sido generados con un porcentaje mayor.

Segundos	30%	40%
Moldura	3,10919	3,14941
Lucy	3,19383	3,36974
Amazona	3,67873	3,76060

Tabla 19. Tiempo, en segundos, en cargar el EBP-Octree en memoria.

3.7.6. Volumen por nivel de las envolventes calculadas.

En la *Tabla 20* se muestra el incremento de volumen de las envolventes con respecto al volumen real de los modelos. Si se observan los datos, tal y como es de esperar, se puede apreciar que las envolventes de los niveles mayores se aproximan más al volumen real del modelo, mientras que las envolventes de los primeros niveles se alejan bastante de este valor. También se puede apreciar que, para los tres modelos y en todos los niveles, las envolventes generadas tomando un porcentaje mayor del número de planos se aproximan mejor al volumen del modelo.

% Volumen de la envolvente	Moldura 30%	Moldura 40%	Lucy 30%	Lucy 40%	Amazona 30%	Amazona 40%
Nivel 1	529,30%	447,16%	821,05%	466,70%	517,15%	316,62%
Nivel 2	285,54%	223,79%	421,70%	231,79%	375,26%	256,46%
Nivel 3	196,68%	167,53%	236,93%	169,64%	221,59%	167,97%
Nivel 4	150,06%	129,76%	154,42%	127,32%	143,17%	123,99%
Nivel 5	122,80%	111,29%	124,67%	115,50%	116,28%	109,78%
Nivel 6	111,13%	107,46%	110,53%	107,67%	106,39%	104,38%
Nivel 7	104,87%	103,43%	103,60%	102,69%	102,31%	101,68%
Nivel 8	101,72%	101,30%	101,13%	100,84%	100,76%	100,57%
Nivel 9	100,49%	100,37%	100,36%	100,28%	100,23%	100,18%
Nivel 10	100,12%	100,09%	100,13%	100,10%	100,06%	100,05%
Nivel 11	100,04%	100,03%	100,04%	100,04%	100,02%	100,01%

Tabla 20. Porcentajes del volumen de las envolventes con respecto al volumen real del modelo.

En la *Gráfico 18* se puede ver una comparativa de la suma de los porcentajes de los volúmenes de las envolventes de los tres modelos. Se puede apreciar, tal y como era de esperar, que cuando mayor es el porcentaje de planos seleccionados, los volúmenes de las envolventes para todos los niveles del EBP-Octree son mucho mejores. De aquí se puede concluir que es mucho mejor utilizar un porcentaje mayor de planos seleccionados, ya que los volúmenes de las envolventes generadas se aproximan más al volumen del modelo. Cuanto más se aproxime el volumen de la envolvente al volumen del modelo los cálculos futuros para la detección de colisiones serán más precisos y eficientes, ya que producirán menos falso positivos.

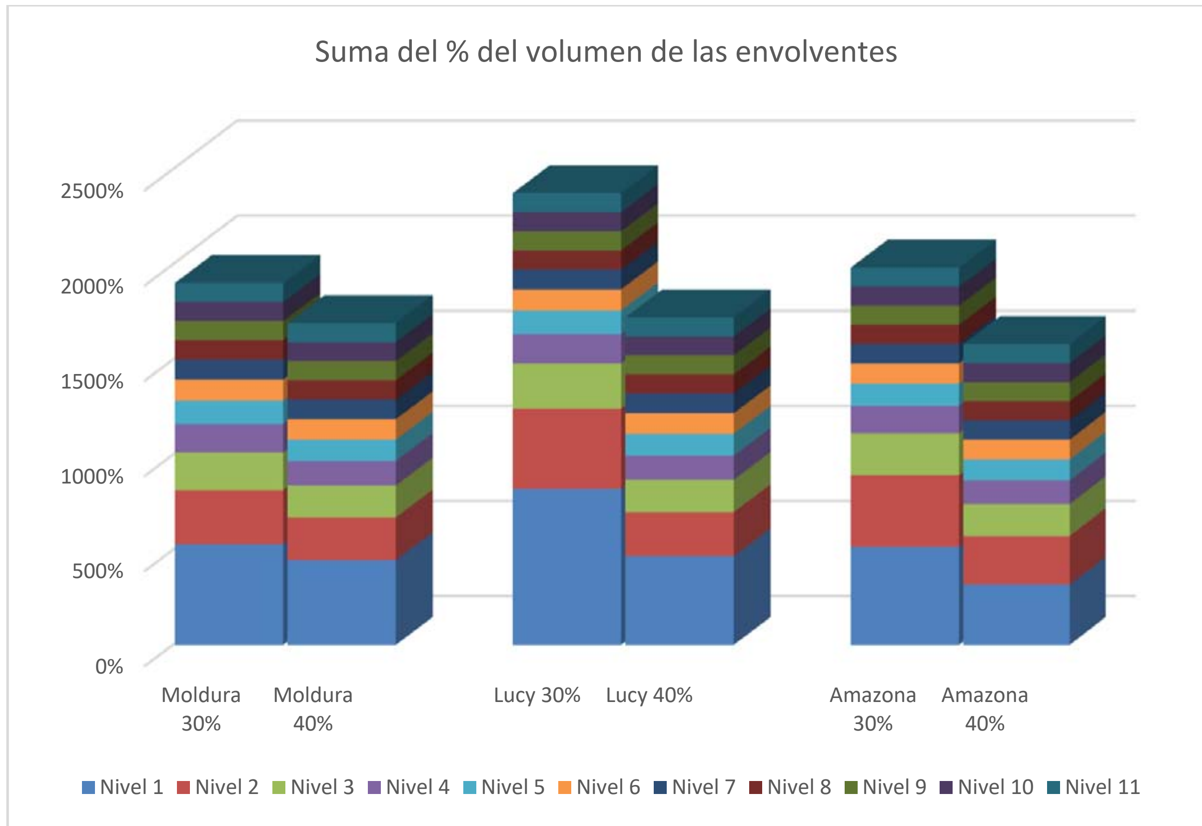


Gráfico 18. Suma de los porcentajes de volúmenes de las envolventes para los tres modelos calculados por el método de k-mediana y seleccionando un 30% y un 40% de los planos.

3.7.7. Conclusión obtenida tras el estudio de los resultados.

La elección del porcentaje de selección de planos se ha hecho mediante un estudio empírico de los resultados obtenidos en tres modelos de datos de un tamaño grande.

De todos los parámetros estudiados anteriormente se puede concluir que el porcentaje óptimo para la selección de planos es del 40%, ya que los costes de construcción, el tamaño del EBP-Octree que se mantiene en memoria y en el tamaño de los archivos en disco son sobradamente compensables con las mejoras que aporta la selección de este mayor número en el porcentaje de planos: las envolventes de los nodos se aproximan más al modelo, garantizando que los futuros cálculos para la detección de colisiones entre modelos sean mucho más precisos y eficientes, ya que evitan muchos falsos positivos.

Por todo lo anterior, a partir de ahora en todos los modelos con los que se va a trabajar, los EBP-Octree se generarán utilizando el método de selección de planos del k-mediana y tomando un porcentaje del 40% de los planos.

3.8. Resumen.

La estructura de datos presentada permite:

- Representar modelos con una alta resolución que pueden estar formados por varias decenas de millones de polígonos.
- Gestionar en disco algunos de los procesos implicados en la construcción del EBP-Octree para no saturar la memoria principal del ordenador.
- Utilizar poca memoria principal tanto en la fase de construcción como en la de carga en memoria del modelo.
- Obtener volúmenes envolventes ajustados al modelo, lo cual permite, ocupando unos 50 Megabytes de memoria, que se tengan unas envolventes que se ajustan alrededor del 105% del volumen original del modelo.
- El tener volúmenes envolventes tan ajustados permite que el cálculo de las colisiones dé como resultado menos falsos positivos en la detección.

4. INCLUSIÓN PUNTO EN SÓLIDO

En este capítulo se presentan los algoritmos diseñados para el cálculo de la inclusión punto en sólido utilizando el EBP-Octree. Estos se pueden clasificar en: los que realizan un test de inclusión punto en sólido y los que calculan la distancia mínima y orientación entre un punto y un sólido.

Los objetos se suelen representar mediante una colección de polígonos que forman su B-Rep. Dos objetos modelados con polígonos colisionan si y sólo si alguno de los polígonos de un objeto interseca con alguno de los polígonos del otro objeto. La aparición de escáneres 3D cada vez más precisos ha posibilitado el obtener objetos con una gran resolución, formados por varios cientos de millones de polígonos. Utilizar el método de la fuerza bruta para detectar la colisión entre dos objetos formados por un gran número de polígonos se hace casi inviable. Para solucionar este problema se plantea la utilización de una representación diferente del objeto para la detección de colisiones.

4.1. Inclusión punto en sólido utilizando el EBP-Octree.

Los EBP-Octrees son una estructura jerárquica de volúmenes envolventes que utilizan la subdivisión del espacio para clasificar las zonas de los objetos que pueden colisionar. Estos pueden trabajar con objetos formados por varias decenas de millones de polígonos.

La detección de colisiones con los EBP-Octree se realiza distinguiendo dos casos. El primero comprobará si un punto está dentro o fuera del objeto, test inclusión punto en sólido y en el segundo caso se comprobará si dos objetos interseccionan en un momento dado. Para el primero de los test se implementan dos algoritmos diferentes, el primero que calcula solamente si el punto está dentro o fuera del modelo y el segundo que calcula y devuelve la distancia y la normal del polígono de la superficie real del objeto que se encuentre más cercano al punto que se está testeando.

4.2. Test de inclusión punto en sólido.

En la bibliografía especifica se pueden encontrar numerosos trabajos que calculan el test de inclusión punto en sólido, ya que es uno de los problemas clásicos dentro de la informática gráfica. Un test punto en sólido consiste en que, dado un punto en el espacio y un objeto, se detecte si este punto está dentro, fuera o en la frontera del objeto. En los trabajos encontrados se puede apreciar que cada uno de ellos lo resuelve de una manera diferente, dependiendo del esquema de representación elegido para modelar el objeto sobre el que trabajan. De todos estos trabajos se puede apreciar que algunos de ellos, los que utilizan grid o estructuras espaciales, no proporcionan una respuesta exacta, sino que dan una aproximación de la posición del punto con respecto al modelo. Si lo que se necesita es una respuesta exacta de dónde se encuentra el punto con respecto al objeto se tienen los métodos que utilizan el teorema de la curva de Jordan (Jordan 1887; Akenine-Möller, Haines, and Hoffman 2008) o los que utilizan simplices (Feito and Torres 1997; Ogayar, Segura, and Feito 2005).

Los EBP-Octrees solucionan el problema que tienen los métodos que utilizan estructuras espaciales, ya que permiten dar una respuesta exacta de la posición del punto con respecto al objeto. Para ello se aprovechan de la jerarquía de volúmenes

envolventes que se tienen en cada uno de los nodos y de la clasificación de los polígonos reales del objeto que cortan a cada nodo hoja. Así van filtrando las comparaciones que se deben hacer hasta llegar a los nodos hoja, donde se utilizan los polígonos reales del objeto que cortan a ese nodo hoja junto con un método basado en el teorema de la curva de Jordan (O'Rourke 1994), para calcular la posición exacta del punto con respecto al objeto. Por todo lo anterior se puede afirmar que los EBP-Octrees tienen la ventaja de los métodos que utilizan la clasificación espacial unida con la mejora que presentan los métodos que dan una respuesta exacta de la posición de un punto.

Para determinar la inclusión de un punto en sólido se han probado diversas aproximaciones al problema:

- Las que utilizan las envolventes de los nodos intermedios para ir filtrando las comprobaciones, ya que si el punto se encuentra fuera de la envolvente se puede afirmar que el punto está fuera del objeto.
- Las que no utilizan las envolventes, llegando hasta el nodo hoja y comprobando directamente la posición del punto con los polígonos reales del modelo que lo cortan.
- Las que usan una caché de subárboles cargados en memoria.
- Las que sólo cargan los nodos que se encuentran en el camino del octree para llegar al nodo hoja que incluye al punto que se pretende comprobar.
- Las que cargan todo el subárbol que contiene al punto.
- Las que recorren el octree hasta un cierto nivel.

A continuación, se detallan los criterios en los que se basan cada uno de los métodos expuestos anteriormente, indicando la combinación de los mismos utilizados en cada uno de los test de inclusión punto en sólido que se han realizado.

4.2.1. Test de inclusión punto en sólido usando las envolventes de los nodos.

Este test tiene la ventaja de que aprovecha las envolventes de los nodos intermedios para ver la posición del punto con respecto a estas envolventes. El problema que presenta es que las envolventes producen falsos positivos, ya que el volumen de

estas es mayor que el del objeto. Este motivo hace que este test funcione muy bien cuando el punto está alejado de la superficie del objeto, pero cuando el punto se encuentra próximo a la superficie del objeto los cálculos aumentan considerablemente, ya que se tiene que comprobar la posición del punto con respecto a todas las envolventes de los nodos intermedios, hasta llegar a los nodos hoja, donde se comprueba con la superficie real del objeto.

```
Variables
  resultado Logica
  nivel Entero
  Nodo EnteroLargo
  Octcode EnteroLargo
Inicio
  Si el punto está fuera de la caja envolvente del objeto
    Entonces resultado = falso
    Sino
      Si el punto está fuera de la envolvente del nodo raíz
         Entonces resultado = falso
         Sino
           octcode = calculaOctcode(punto)
           nodo = nodo raíz
           nivel = 1
           Mientras nivel < Nivel_Máximo
             Si nodo es negro
                Entonces resultado = verdad
                Sino
                  Si nodo es blanco
                     Entonces resultado = falso
                     Sino
                       Si el punto está fuera de la envolvente
                         Entonces resultado = falso
                         Sino
                           nivel = nivel + 1
                           nodo = calculaNodoHijo(octcode, nivel)
                       FinSi
                  FinSi
             FinSi
           FinMientras
      FinSi
  FinSi
  Si el punto está dentro con respecto los triángulos que cortan al nodo hoja
    entonces resultado = verdad
    sino resultado = falso.
  FinSi
  Devolver resultado
Fin
```

Algoritmo 7. Seudocódigo para el cálculo del test inclusión punto en sólido utilizando las envolventes.

En este test lo primero que se comprueba es si el punto está dentro de la caja envolvente del objeto, (si está fuera queda claro que el punto se encuentra fuera del objeto). Si el punto está dentro de la caja envolvente del objeto, entonces se comprueba si el punto está dentro o fuera con respecto a la envolvente de primer nivel. Si el punto está fuera el proceso acaba, pero si está dentro se calcula el *octcode* del punto, lo que equivale al camino que se tiene que recorrer por el árbol para llegar al nodo hoja que incluiría a ese punto. Con el *octcode* se comprueba la envolvente del nodo hijo en la que se encuentra el punto. Este proceso se repite hasta que se llega a los nodos hoja en los cuales, la comprobación de si el punto está dentro o fuera se realiza con los polígonos reales del objeto que cortan a ese nodo hoja. Como lo que se desea es saber si el punto está dentro o fuera del objeto el proceso anterior también se detendría en el caso de que el punto estuviera dentro de un nodo negro o de uno blanco.

En el *Algoritmo 7* se detalla en seudocódigo el proceso descrito anteriormente.

4.2.2. Test de inclusión punto en solido sin usar las envolventes de los nodos.

Para intentar solucionar el problema que presenta el test de inclusión punto en sólido se pensó en utilizar una variante del mismo que consiste en no utilizar las envolventes intermedias, sino en utilizar solamente los triángulos que forman parte de la superficie del sólido. Esto es equivalente a usar un octree como índice espacial. Esta nueva solución funciona muy bien en el caso de que el punto esté cerca de la superficie del objeto, ya que no se tienen que hacer todas las comprobaciones con respecto a las envolventes de los nodos intermedios. Pero funciona peor cuando el punto se encuentra fuera de la superficie del sólido y más concretamente cuando está fuera de la primera envolvente. En este test lo primero que se comprueba es si el punto está dentro de la caja envolvente del objeto, si está fuera de esta, está claro que el punto se encuentra fuera del objeto. Si el punto está dentro de la caja envolvente del objeto, entonces se calcula el octcode del punto, lo que equivale al camino que se tiene que recorrer por el árbol para llegar al nodo hoja que incluiría a ese punto. Con este octcode se recorre todo el camino del árbol que lleva hasta la posición de ese punto. Si uno de los nodos intermedios es negro o blanco, este proceso se detiene devolviendo que el punto está dentro o fuera del sólido respectivamente. Si se llega al nodo hoja entonces se comprueban con los polígonos reales del objeto que cortan

a ese nodo hoja.

En el *Algoritmo 8* se detalla en seudocódigo el proceso descrito anteriormente.

```
Variables
  resultado Logica
  nivel Entero
  Nodo EnteroLargo
  Octcode EnteroLargo
Inicio
  Si el punto está fuera de la caja envolvente del objeto
    Entonces resultado = falso
    Sino
       octcode = calculaOctcode(punto)
       nodo = nodo raíz
       nivel = 1
       Mientras nivel < Nivel_Máximo
         Si nodo es negro
           Entonces se devuelve verdad
           Sino
             Si nodo es blanco
               Entonces se devuelve falso
               Sino
                 nivel = nivel + 1
                 nodo = calculaNodoHijo(octcode, nivel)
             FinSi
         FinSi
       FinMientras
  FinSi
  Si el punto está dentro con respecto los polígonos que cortan al nodo hoja
    Entonces
      resultado = verdad
    Sino
      resultado = falso
  FinSi
  Devolver resultado
Fin
```

Algoritmo 8. Seudocódigo para el cálculo del test de inclusión punto en sólido sin utilizar las envolventes.

4.2.3. Test de inclusión punto en sólido usando una caché de subárboles.

Una optimización que se puede hacer es la utilización de una caché de subárboles cargados. Este tipo de estructura está pensada para ser utilizada por dispositivos, como pueden ser los hápticos, que utilizan un movimiento continuo en el puntero

para el que hay que comprobar si se encuentra dentro o fuera del objeto.

La caché consiste en cargar en memoria los veintisiete subárboles que rodean al subárbol en el que en un momento dado se encuentra incluido el punto para el que se está comprobando la inclusión en el objeto (ver *Figura 82*). La idea de este método está basada en que conforme se mueve el puntero por el espacio, cuando este sale del subárbol que lo incluye, entonces se irían cargando y descargando de manera dinámica los subárboles que se encuentran alrededor del mismo. El esquema de carga de los subárboles en un instante dado para el EBP-Octree se puede ver en la *Figura 83*. La carga y descarga de los mismo se puede realizar siguiendo una serie de criterios como la velocidad de movimiento del puntero o comprobando cómo de sobrecargado está en un instante dado el ordenador en el que se ejecutan los test. Si la trayectoria del puntero se conoce con anterioridad también se puede programar de antemano cuándo se producirían la carga y la descarga en memoria de dichos subárboles.

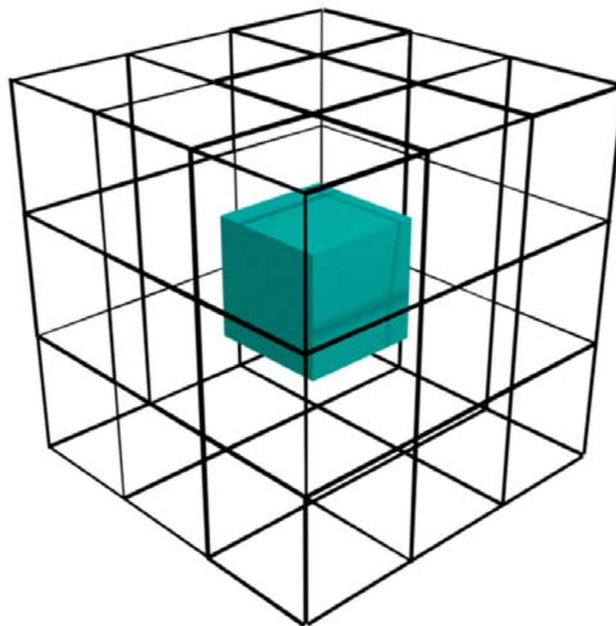


Figura 82. Nodo central donde se encuentra el punto y los veintisiete nodos de alrededor.

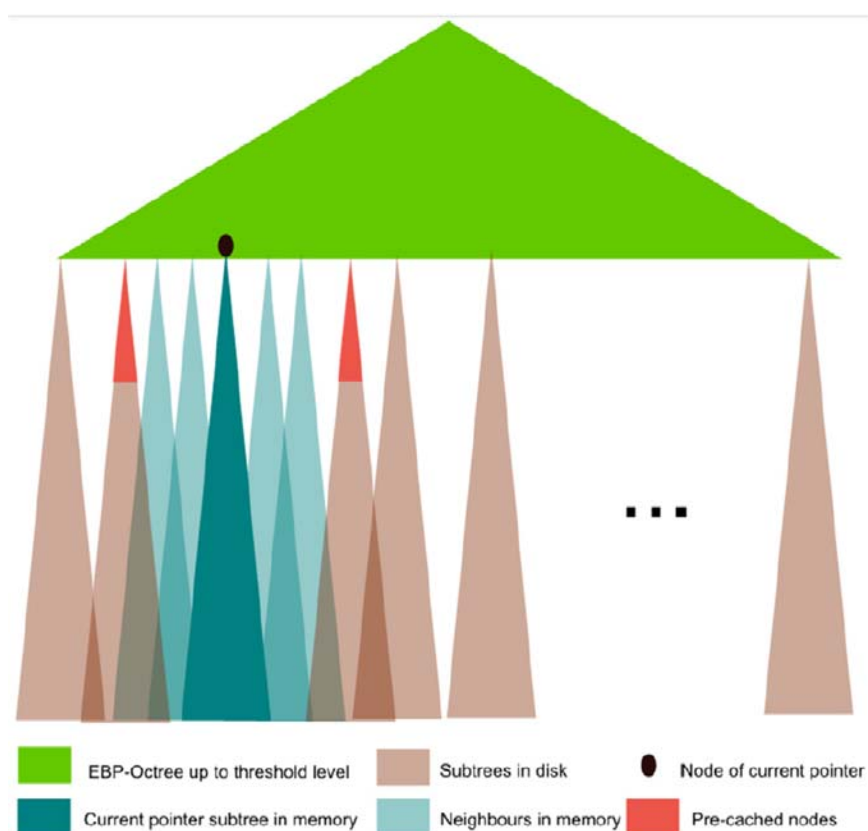


Figura 83. Planificación de la caché en un EBP-Octree.

4.2.4. Test inclusión punto en sólido con carga de sólo los nodos que se encuentran en el camino para llegar al nodo que contiene al punto.

Los test que se basan en este método van cargando en memoria sólo los nodos del EBP-Octree conforme se van utilizando. Una vez utilizados se descargan de la memoria, sin tener en cuenta si el próximo punto lo va a utilizar. Este método empieza a comprobar si el punto está dentro o fuera del objeto empezando por el nodo raíz y baja por el árbol siguiendo el camino marcado por la posición del punto. Si en el camino se llega a un nodo *blanco* o *negro* entonces se para el proceso indicando que el punto está fuera o dentro del objeto respectivamente. En la *Figura 84* se pueden ver los nodos que son almacenados en memoria principal cuando se comprueba si un punto incluido en el nodo hoja está dentro o fuera del objeto.

Figura 84. Almacenamiento en memoria principal de los nodos necesarios para saber si un punto está incluido o no en un objeto.

4.2.5. Test de inclusión punto en sólido con carga del subárbol que contiene al punto que se desea comprobar.

Los test que utilizan este método se basan en cargar el subárbol completo que contiene el camino hasta el punto a chequear, de manera que se mantiene en memoria el subárbol cargado hasta que otro punto necesite otro subárbol diferente. Este método se basa en la idea que utilizaba el método de la caché, pero en vez de mantener los veintisiete subárboles sólo se mantiene un subárbol (ver *Figura 85*).

Figura 85. Almacenamiento en memoria principal del subárbol que contiene al punto.

4.2.6. Test inclusión punto en sólido que llega hasta un nivel fijo del EBP-Octree.

Los test que utilizan esta técnica recorren el octree bajando hasta un nivel dado del mismo, de manera que permiten dar una respuesta más o menos precisa dependiendo del tiempo que se disponga para calcularla (ver *Figura 86*). Los test basados en esta técnica están pensados para ser utilizados por aplicaciones que necesiten una respuesta rápida y no muy precisa, ya que permiten adaptar los cálculos con el tiempo disponible para llevarlos a cabo. Cuantos más niveles del octree se consigan bajar, mayor precisión tendrá la respuesta. Estos test también se pueden utilizar en dispositivos de bajo rendimiento, ya que permiten devolver una respuesta aproximada en un tiempo máximo.

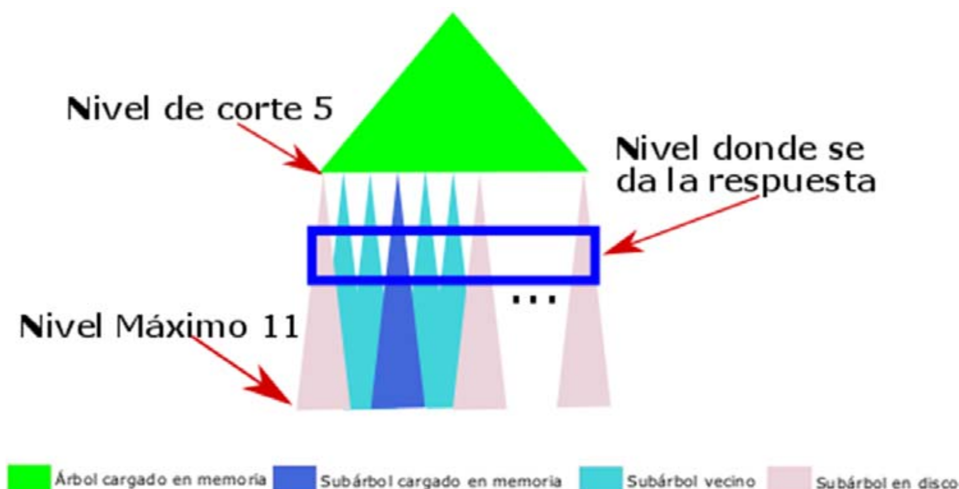


Figura 86. Nivel del árbol en el que se da la respuesta a la inclusión o no de un punto en un objeto.

4.3. Tipos de test de inclusión punto en sólido realizados.

Los tipos de test que se han elegido para la realización de las pruebas combinan uno o varios de los métodos vistos en el apartado 4.1.1. La idea fundamental de la elección de los mismos se basa en que intenten abarcar un abanico amplio que recojan todos los posibles casos que se pueden encontrar en las diferentes aplicaciones que los utilizan. Los diferentes test ejecutados sobre los tres modelos de trabajo son:

1. *Con caché y con envolventes hasta los nodos hoja.* En estos se cargan los

veintisiete subárboles de alrededor y se utilizan las envolventes de los nodos que se encuentran por el camino hasta llegar al nodo hoja que incluye el punto que se quiere chequear. Utilizan las envolventes de los nodos intermedios para ver si el punto está fuera de la envolvente. Si lo está el proceso se detiene y ya que se puede afirmar que dicho punto está fuera del objeto.

2. *Sin caché, con nodos y con envolventes hasta nodos hoja.* Estos test cargan sólo el nodo que se encuentra en el camino del octree que lleva hasta el punto a chequear, de manera que en cada nodo intermedio se comprueba con la envolvente si el punto está dentro o fuera.

3. *Sin caché con el subárbol y con envolventes hasta los nodos hoja.* Los test basados en esta combinación de técnicas funcionan cargando sólo el subárbol que se encuentra en el camino que lleva hasta el punto a chequear, de manera que el subárbol se mantiene en memoria hasta que otro punto necesite otro subárbol. Estos test utilizan la envolvente de los nodos intermedios para ver si el punto se encuentra fuera de la misma.

4. *Con caché y sin envolventes.* Estos test utilizan la caché de subárboles pero no utilizan las envolventes de los nodos intermedios para comprobar si el punto está fuera del objeto. Debido a esto, los test tienen que recorrer todo el camino que lleva al punto y sólo se detiene en caso de que el punto se encuentre dentro de un nodo negro o blanco, lo cual indicaría que el punto está dentro o fuera del objeto respectivamente. En el caso de que el punto no caiga dentro de uno nodo blanco o negro se llega al nodo hoja, comprobándose la posición del punto con respecto a los polígonos reales del objeto que cortan a ese nodo hoja.

5. *Sin caché con el subárbol y sin envolventes.* Estos test no utilizan la caché de subárboles. Solo cargan un subárbol y lo mantienen en memoria mientras este sea utilizado. Para realizar el cálculo no utilizan las envolventes de los nodos intermedios. Recorren el camino marcado por la posición del punto de manera que, o bien se encuentran un nodo negro o blanco o llegan al nodo hoja que lo contiene, comprobando la posición con los polígonos reales del punto que cortan a dicho nodo hoja.

6. *Con caché y con envolventes hasta un nivel.* Este tipo de test permiten adaptar la precisión de los cálculos con el tiempo disponible para realizarlos. Así de

manera que cuando se ha agotado el tiempo se devuelve la solución parcial que se tiene calculada. Estos test utilizan la caché de subárboles y las envolventes de los nodos intermedios para calcular la solución.

7. *Sin caché, con nodos y con envolventes hasta un nivel.* Estos test se basan en la idea de los test anteriores, pero estos no cargan la caché de subárboles, sino que solo cargan los nodos intermedios que van necesitando. Para realizar los cálculos utilizan las envolventes de los nodos intermedios. El proceso se detiene en el caso de que el punto se encuentre fuera.

8. *Sin caché con subárbol y con envolventes hasta un nivel.* Estos test se basan en la idea de los test anteriores, pero cargan solo un subárbol y utilizan las envolventes de los nodos intermedios para ver si el punto se encuentra fuera del objeto. El subárbol permanece en memoria mientras que no se necesite cargar otro subárbol.

4.3.1. Elección del conjunto de puntos para la realización del test de inclusión punto en sólido.

Para la elección del tipo de test de inclusión punto en sólido se ha hecho un estudio de tiempos obtenidos tras la ejecución de los test sobre un conjunto de puntos en los tres modelos seleccionados. Los conjuntos de puntos que se seleccionaron siguieron dos tipos de distribuciones diferentes, todos ellos dentro de la caja envolvente:

- *Aleatoria*: Se genera una nube de puntos aleatorios por toda la caja envolvente del objeto que no tienen ninguna relación en el espacio-tiempo.
- *Movimiento continuo*: Se genera un conjunto de puntos que se encuentran muy próximos o en la superficie del objeto, siguiendo un recorrido casi continuo. Se intenta emular el comportamiento que tendría un dispositivo háptico tipo puntero que se desplaza siguiendo una trayectoria continua en el espacio-tiempo.

4.3.2. Resultados obtenidos por los diferentes tipos de test de inclusión punto en sólido.

Para evaluar el comportamiento del EBP-Octree a la hora de comprobar si un punto

está dentro o fuera del modelo se ha hecho un estudio empírico. Para ello se han utilizado los tres grandes modelos con los que se ha trabajado a lo largo de esta tesis: Amazona Herida de 28 M.P., Moldura de 26 M.P. y Lucy de 28 M.P. Para cada uno se ha creado su EBP-Octree utilizando el método de selección de planos k-mediana y tomando el 40% de los planos. A cada uno de estos tres EBP-Octree se le han pasado los nueve tipos de test indicados en los apartados anteriores y cada uno se han ejecutado sobre los dos tipos de conjuntos de puntos indicados en el apartado anterior. Los resultados se muestran agrupándolos en dos tipos, los que hacen la detección sin tener en cuenta un nivel límite y los que si lo tienen. En la *Tabla 21* y *Tabla 22* se pueden ver para cada modelo los tiempos medios que ha tardado en ejecutarse cada test sin tener en cuanta un nivel límite, y en *Tabla 23* y *Tabla 24* las que si lo tienen.

	Tiempo medio en segundos por test		
Test 1 Millón de puntos aleatorios	**Amazona**	**Moldura**	**Lucy**
Con cache y con envolventes hasta los nodos hoja.	0,00215761	0,00116299	0,00198673
Sin cache, con nodos y con envolventes hasta nodos hoja.	0,00121396	0,00095363	0,00129153
Sin cache, con subárbol y con envolventes hasta los nodo hoja.	0,00145000	0,00098787	0,00135521
Con cache y sin envolventes.	0,00225272	0,00130701	0,00239608
Sin cache se lee solo nodo y sin envolventes.	0,00289495	0,00248190	0,00179104
Sin cache se lee el subárbol y sin envolventes.	0,00292825	0,00256220	0,00233267

Tabla 21. Tiempo en segundos de los tests sin nivel límite y distribución puntos aleatoria.

	Tiempo medio en segundos por test		
Test 1 Millón de puntos movimiento continuo	**Amazona**	**Moldura**	**Lucy**
Con cache y con envolventes hasta los nodos hoja.	0,00032035	0,00015784	0,00029082
Sin cache, con nodos y con envolventes hasta nodos hoja.	0,00034005	0,00024478	0,00043685
Sin cache, con subárbol y con envolventes hasta los nodo hoja.	0,00033176	0,00026868	0,00037384
Con cache y sin envolventes.	0,00034321	0,00020556	0,00032014
Sin cache se lee solo nodo y sin envolventes.	0,00041719	0,00032623	0,00036422
Sin cache se lee el subárbol y sin envolventes.	0,00038020	0,00034214	0,00032140

Tabla 22. Tiempo en segundos de los tests sin nivel límite y distribución puntos movimiento continuo.

En la *Tabla 23* y *Tabla 24* se aprecia un gran incremento en los tiempos de cálculo entre los test que llegan hasta el nivel 5 con respecto a los que llegan hasta el nivel 6. Este salto se produce porque el nivel de corte de los EBP-Octrees es el 5, haciendo esto que para realizar los test del nivel 6 se tenga que acceder a disco.

	Test 1 Millón de puntos aleatorios								
	Tiempo medio en segundos por test								
	Con cache			Sin cache					
				Con nodo			Con subárbol		
Nivel	Amazona	Moldura	Lucy	Amazona	Moldura	Lucy	Amazona	Moldura	Lucy
1	0,00000416	0,0000006	0,00000385	0,00000427	0,00000059	0,00000419	0,00000422	0,00000058	0,00000381
2	0,00000616	0,00000073	0,00000447	0,00000621	0,00000071	0,00000449	0,00000626	0,00000073	0,00000445
3	0,00000691	0,00000084	0,00000485	0,00000697	0,00000082	0,00000486	0,00000705	0,00000084	0,00000484
4	0,00000718	0,0000009	0,00000503	0,0000072	0,00000091	0,00000501	0,0000073	0,0000009	0,000005
5	0,00000737	0,00000096	0,00000519	0,00000739	0,00000098	0,00000515	0,00000749	0,00000096	0,00000514
6	0,00200749	0,00104616	0,00256526	0,00047757	0,00010271	0,00045382	0,00040761	0,00011583	0,00052249
7	0,00203273	0,00117672	0,00258433	0,00049771	0,00011836	0,00046102	0,0004088	0,00011606	0,00055575
8	0,00279802	0,00100587	0,0025805	0,00050178	0,00012381	0,00046201	0,00041057	0,0001425	0,00060967
9	0,00278016	0,00100558	0,0025776	0,00051016	0,00012599	0,00046261	0,00045099	0,00021266	0,00075093
10	0,00278771	0,00100408	0,00257687	0,00052367	0,00013144	0,00046383	0,00048492	0,00044423	0,00081248
11	0,00307901	0,00132225	0,00300168	0,00052803	0,00036069	0,00051198	0,00055105	0,00091462	0,00096457

Tabla 23. Tiempo en segundos de los tests con nivel límite y distribución puntos aleatoria.

	Test 1 Millón de puntos movimiento continuo								
	Tiempo medio en segundos por test								
	Con cache			Sin cache					
				Con nodo			Con subárbol		
Nivel	Amazona	Moldura	Lucy	Amazona	Moldura	Lucy	Amazona	Moldura	Lucy
1	0,00001082	0,00000082	0,0000017	0,00001073	0,00000081	0,00000172	0,00001073	0,00000081	0,00000172
2	0,00001452	0,00000142	0,00000226	0,00001471	0,00000143	0,00000226	0,00001447	0,00000144	0,00000226
3	0,0000164	0,00000179	0,00000259	0,00001635	0,00000181	0,00000259	0,00001635	0,00000181	0,00000259
4	0,00001768	0,00000204	0,00000273	0,00001764	0,00000206	0,00000273	0,00001761	0,00000206	0,00000273
5	0,00001844	0,00000218	0,0000028	0,00001838	0,00000219	0,0000028	0,00001835	0,0000022	0,0000028
6	0,00018897	0,00022715	0,00018256	0,00018827	0,00022912	0,00028545	0,00018801	0,0002279	0,00028592
7	0,00019749	0,00024367	0,00019913	0,00023689	0,00029733	0,00034902	0,00019064	0,00023378	0,00029044
8	0,00019727	0,00025854	0,00019924	0,00026717	0,0003105	0,00039026	0,00019184	0,00023759	0,00029358
9	0,00019711	0,00026066	0,00019912	0,00029251	0,00032564	0,0004246	0,00019239	0,00024149	0,00029755
10	0,00019857	0,00026195	0,00019906	0,00031799	0,00034496	0,00045411	0,00019305	0,00024803	0,00030607
11	0,00016437	0,00031855	0,00019917	0,00036958	0,00035899	0,00054204	0,0001994	0,00030943	0,0004936

Tabla 24. Tiempo en segundos de los tests con nivel límite y distribución puntos movimiento continuo.

4.3.3. Estudio comparativo de tiempos para la elección del mejor test de inclusión punto en sólido.

Para seleccionar cuál de los test de inclusión punto en sólido se comporta mejor se ha calculado la media de tiempos en realizar cada uno de los test para los tres modelos y para cada una de las dos distribuciones de puntos. Los datos medios obtenidos se muestran en la *Tabla 25* para los tests que no tienen en cuanta un nivel límite y en la *Tabla 26* para los que si lo tienen.

Tiempo medio en segundos. Test 1 Millón de puntos.	Aleatorio	Movimiento continuo
Con cache y con envolventes hasta los nodos hoja.	0,00176911	0,00025633
Sin cache, con nodos y con envolventes hasta nodos hoja.	0,00115304	0,00034056
Sin cache, con subárbol y con envolventes hasta los nodo hoja.	0,00126436	0,00032476
Con cache y sin envolventes.	0,00198527	0,00028964
Sin cache se lee solo nodo y sin envolventes.	0,00238929	0,00036921
Sin cache se lee el subárbol y sin envolventes.	0,00260771	0,00034791

Tabla 25. Tiempo medio en segundos de los tests sin nivel límite.

	Test 1 Millón de puntos aleatorios					
	Tiempo medio en segundos por test					
	Con cache		Sin cache			
			Con nodo		Con subárbol	
Nivel	Aleatorio	Movimiento continuo	Aleatorio	Movimiento continuo	Aleatorio	Movimiento continuo
1	0,00000287	0,00000445	0,00000301	0,00000442	0,00000287	0,00000442
2	0,00000379	0,00000607	0,0000038	0,00000614	0,00000381	0,00000605
3	0,0000042	0,00000693	0,00000422	0,00000692	0,00000424	0,00000691
4	0,00000437	0,00000748	0,00000437	0,00000747	0,00000440	0,00000747
5	0,00000451	0,00000781	0,00000450	0,00000779	0,00000453	0,00000778
6	0,00187297	0,00019956	0,00034470	0,00023428	0,00034864	0,00023394
7	0,00193126	0,00021343	0,00035903	0,00029441	0,00036020	0,00023828
8	0,00212813	0,00021835	0,00036253	0,00032264	0,00038758	0,00024100
9	0,00212111	0,00021897	0,00036625	0,00034758	0,00047153	0,00024381
10	0,00212289	0,00021986	0,00037298	0,00037235	0,00058054	0,00024905
11	0,00246764	0,00022736	0,00046690	0,00042354	0,00081008	0,00033414

Tabla 26. Tiempo medio en segundos de los tests con nivel límite.

De los datos anteriores se puede deducir que los test realizados sobre el conjunto de puntos generados de manera aleatoria son los que peores resultados obtiene en todos los casos, lo cual es lógico ya que los test han sido pensados para optimizar su

utilización en puntos que sigan una distribución continua en el espacio-tiempo. Otro dato significativo es que los test que utilizan caché son los que mejor se compartan cuando los puntos siguen una ruta continua en el espacio-tiempo, similar a la que seguiría el puntero de un dispositivo háptico.

Otras conclusiones que se pueden sacar de los datos anteriores es que los test que se ejecutan llegando hasta un cierto nivel del EBP-Octree, obtienen unos tiempos muy bajos cuando el nivel máximo de comprobación está cargado en memoria aumentado cuando el nivel no se encuentra en esta. Por esto estos test son ideales para ejecutarlos en aplicaciones que necesiten dar una respuesta aproximada y muy rápida. En el *Gráfico 19* se puede ver la comparativa de tiempos observándose que la ejecución de los test sobre el conjunto de puntos aleatorios es la que peores tiempos obtiene en todos los casos y para todos los niveles. En el *Gráfico 20* se muestran los tiempos medios cuando los puntos siguen una distribución de movimiento continuo, observándose que los test que mejores resultados proporcionan para todos los niveles son los que utilizan la cache.

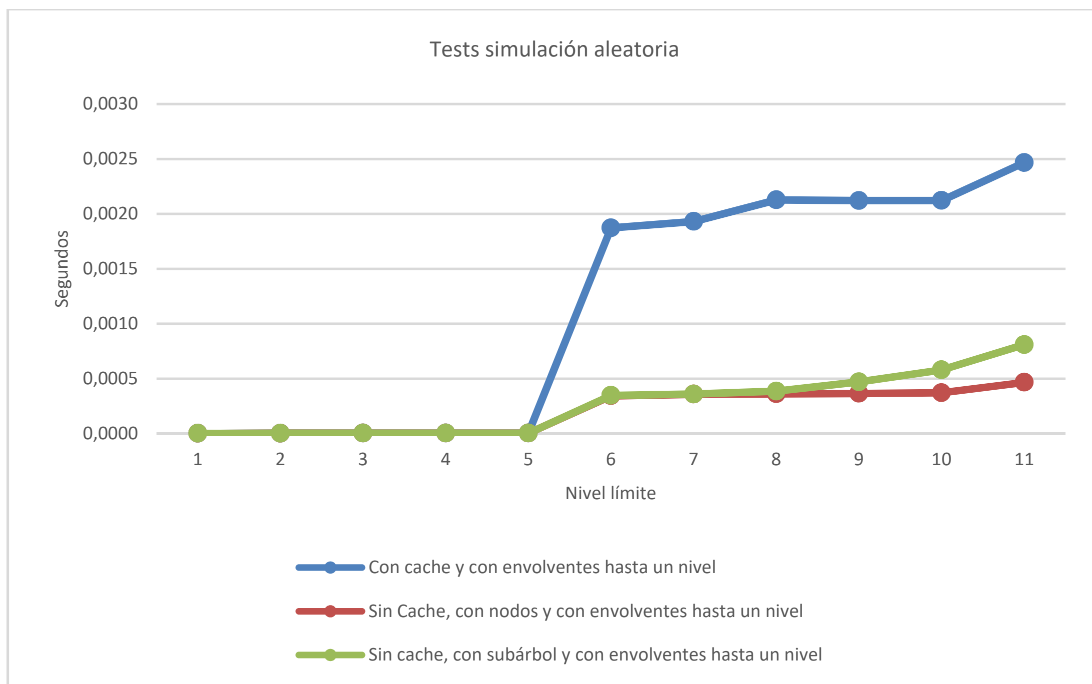


Gráfico 19. Tiempo medio, en segundos, de ejecutar los test siguiendo los puntos una distribución aleatoria.

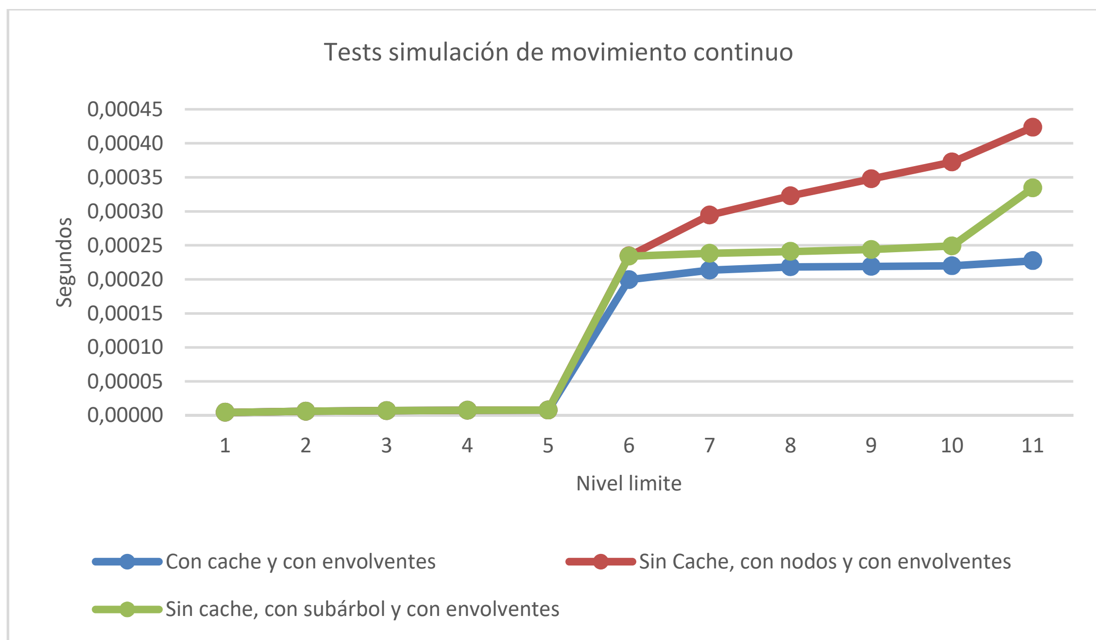


Gráfico 20. Tiempo medio, en segundos, de ejecutar los test siguiendo los puntos una distribución de movimiento continuo.

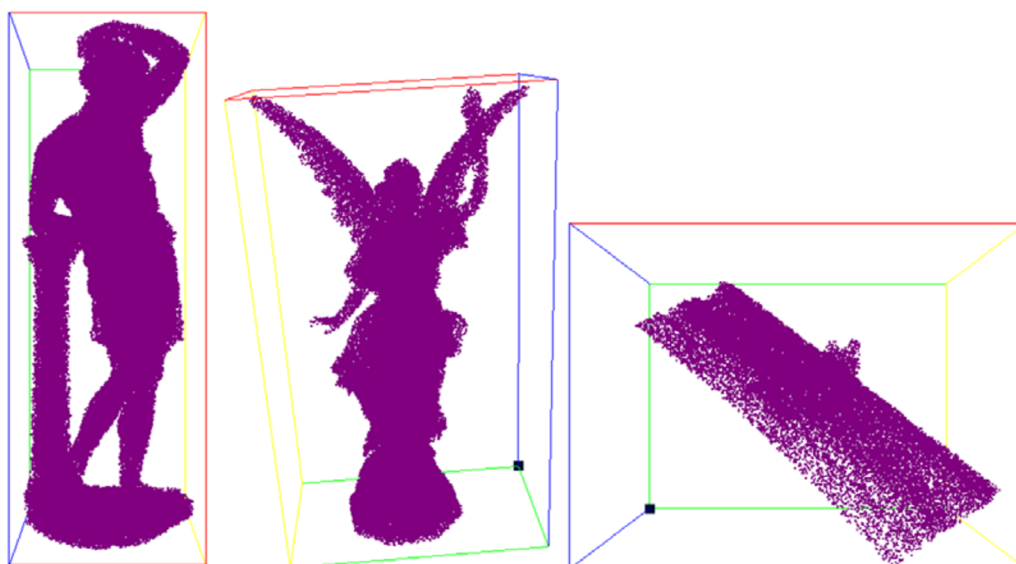
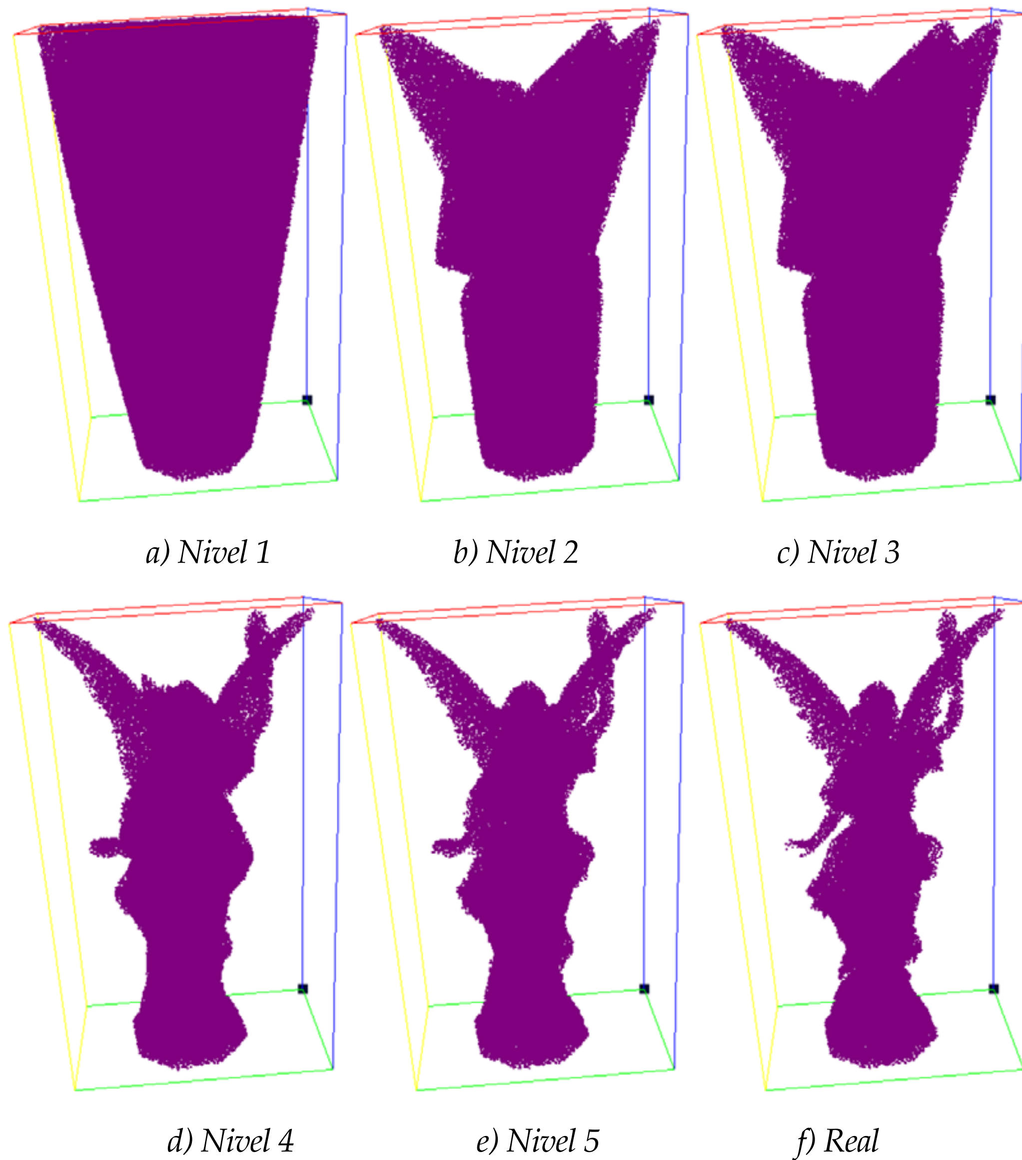


Figura 87. Resultado de los test de inclusión punto en sólido sobre un millón de puntos aleatorios.

En la *Figura 87*, *Figura 88* y *Figura 89* se pueden ver los puntos, en color morado, que caen dentro del sólido, tras ejecutar alguno de los anteriores test de inclusión punto

en sólido para un millón de puntos aleatorios o en superficie.



a) Nivel 1 b) Nivel 2 c) Nivel 3

d) Nivel 4 e) Nivel 5 f) Real

Figura 88. Resultado de ejecutar los test de inclusión punto en sólido por niveles sobre un millón de puntos aleatorios.

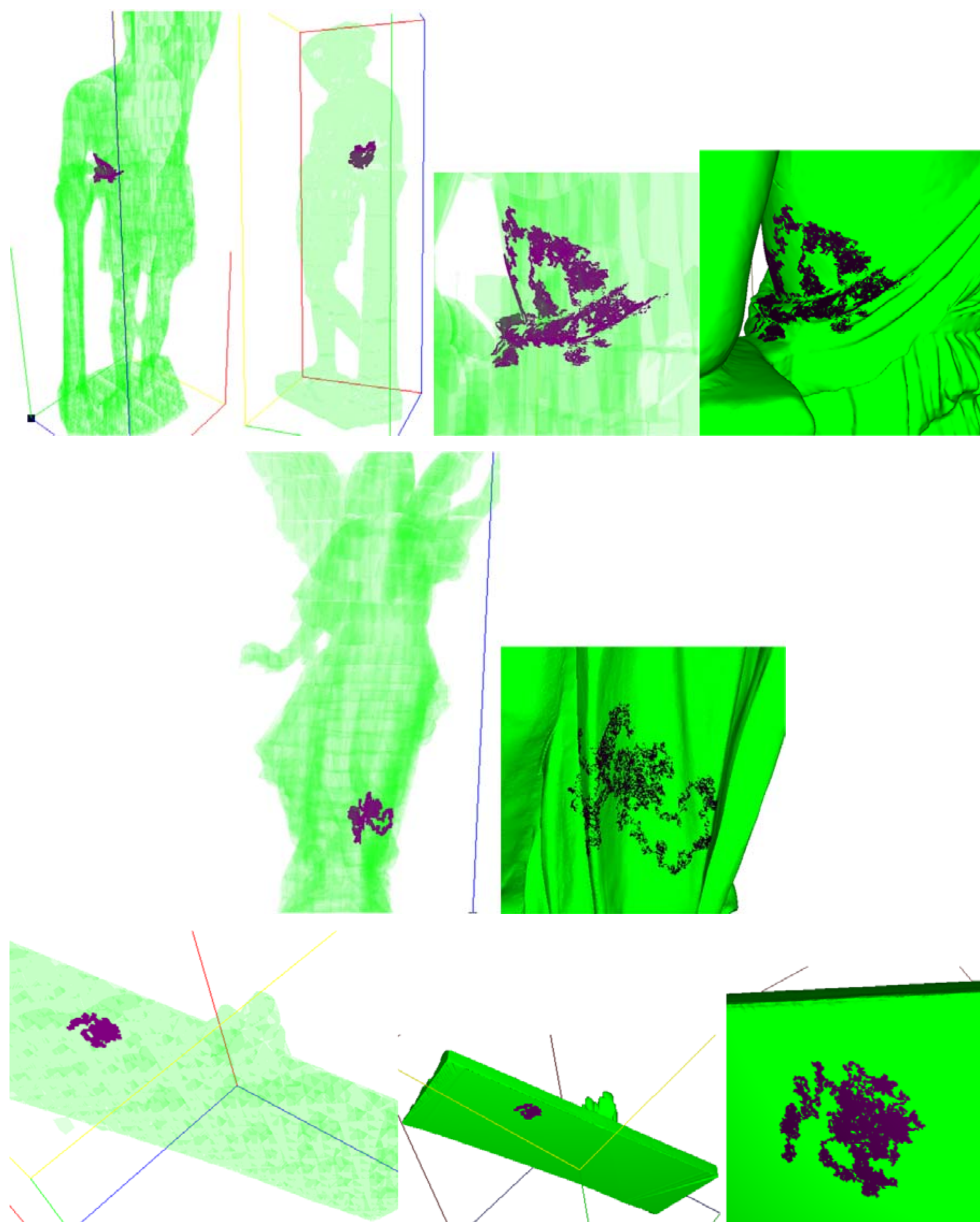


Figura 89. Resultado de ejecutar los test de inclusión punto en sólido sobre un millón de puntos de movimiento continuo.

4.3.4. Gestión de la memoria principal por parte del EBP-Octree durante el renderizado háptico.

La implementación del EBP-Octree utilizando el sistema de caché permite que ordenadores no muy potentes puedan trabajar con modelos muy grandes formados por varias decenas de millones de polígonos. Esto es así porque a la hora de almacenar el modelo en el ordenador se puede consultar la memoria principal disponible y de esta forma se carga el EBP-Octree raíz hasta un nivel que no sobrepase un porcentaje de la misma. El hecho de tener almacenado el EBP-Octree en archivos permite que cuando se necesite comprobar si un punto está dentro o fuera del objeto se cargue en memoria solo la parte necesaria del EBP-Octree.

Las aplicaciones que utilizan un renderizado háptico suelen ser aquellas en las que en un instante dado solo necesitan trabajar con una pequeña parte del modelo, con lo cual hacen que se adapten perfectamente a la estructura propuesta en los EBP-Octree.

Los EBP-Octree también permiten trabajar con una caché de subárboles, en la que se cargan un número de subárboles de manera simultánea en memoria principal. Como se puede ver en los resultados obtenidos en el apartado anterior el uso de esta caché de subárboles está indicado en aplicaciones en las que los puntos sobre los que se realiza la comprobación de inclusión siguen una trayectoria continua en el espacio tiempo, con el objetivo de que los subárboles se carguen el menor número de veces, ya que esta tarea conlleva un sobrecoste en el proceso de detección de la inclusión de un punto dentro de un objeto.

El uso de los EBP-Octree junto con la caché permite que ordenadores domésticos puedan cargar y trabajar con modelos de datos muy grandes y todo ello sin producir una sobrecarga del ordenador donde se esté trabajando.

Como ya se vio en el tema 2 los dispositivos hápticos de tipo puntero necesitan, como mínimo, una frecuencia de refresco de 1 kHz. **Esto es superado ampliamente con la estructura EBP-Octree y el uso de la caché.**

4.4. Test de cálculo de distancias y direcciones de colisión entre un punto y un modelo.

El cálculo de la distancia y la dirección de colisión entre un punto p y un modelo con superficie S, se lleva a cabo utilizando un mapa de distancias. El mapa de distancias de un punto a una superficie se define como una función escalar (ver *Ecuación 6*):

$$Dist: \mathbb{R}^3 \to \mathbb{R}, Dist(p) = \min_{q \in S}\{|p - q|\}, \forall\, p \in \mathbb{R}^3$$

Ecuación 6. Función escalar de un mapa de distancias.

Esta función devuelve la distancia mínima de un punto a una superficie. Si esta superficie es cerrada se puede afirmar que si la distancia es negativa entonces el punto está dentro del objeto y si es positiva lo que se tiene es la distancia a la que se encuentra dicho punto de la superficie.

El cálculo de la distancia entre un punto y un modelo sigue un proceso muy similar al llevado a cabo para calcular los test de inclusión punto en sólido. Este proceso consiste en calcular el *octcode* del punto para el que se quiere obtener la distancia al modelo. El *octcode* proporciona el camino que se debe seguir por el octree para llegar al nodo hoja que incluye a dicho punto. Si dicho nodo existe dentro del EBP-Octree, el cálculo de la distancia y la dirección de colisión se realiza calculando la distancia mínima entre el punto y cada uno de los polígonos reales del modelo que cortan al nodo hoja. El algoritmo devuelve la distancia mínima y la normal del polígono que se encuentra a menor distancia como dirección de colisión. El problema surge cuando el nodo hoja no existe en el octree, lo que significa que en el camino se ha encontrado un nodo blanco o negro. En estos casos, para calcular la distancia, se pueden seguir alguno de los siguientes criterios:

- *Cálculo recursivo*. Consiste en subir al primer nodo gris del camino y a partir de este bajar por todos los hijos, de manera recursiva, hasta llegar a todos los nodos hoja, calculando la distancia de los polígonos reales que cortan al nodo con respecto al punto y seleccionando la más pequeña. El Seudocódigo se puede ver en *Algoritmo 11* el cual llama a dos procedimientos recursivos *Algoritmo 9* y *Algoritmo 10* los cuales calculan la distancia mínima entre un modelo y un punto.

- *Uso de la envolvente de nodo gris*. Aquí lo que se hace es seleccionar el nodo gris más bajo en el camino que marca el *octcode* del punto dentro del octree. Se toma su envolvente convexa y se calcula la distancia de cada uno de los polígonos que la forman sobre el punto, devolviendo la distancia menor y la normal del polígono. El problema que presenta este método es que la distancia calculada no es real, sino que es una aproximación debido a que utiliza la envolvente del nodo gris y no los polígonos reales del modelo. Si el punto se encuentra fuera del modelo la distancia calculada será menor y si el punto está dentro entonces la distancia será mayor que la distancia real.
- *Subir hasta un nodo gris de un nivel determinado*. Este método está basado en el anterior, pero se acota el número de niveles subidos desde el nodo hoja en busca de un nodo gris. Así se limita el error de cálculo al tamaño del nodo intermedio hasta el que sube.
- *Búsqueda de nodos hoja vecinos*. Este método consiste en buscar en los nodos hojas vecinos del nodo que ocupa el punto, de manera que si existen uno o varios nodos hojas vecinos se calcula la distancia a cada uno de ellos, devolviendo la menor de estas y la normal del polígono que la proporcione. El problema de este método es que el EBP-Octree no está pensado ni optimizado para buscar nodos vecinos. Debido a esto, puede resultar muy costoso y, por tanto, su uso no es muy apropiado.

```
Procedimiento CalculaDistanciaMinimaSubArboles(posNodo, disPunTrian){
    punteroNodo= punteroNodos->at(posNodo);
    Para i=0 hasta 7 con incremento de 1
        Si (punteroNodo->hijos[i]>-1)
           CalculaDistanciaMinimaSubArboles(punteroNodo->hijos[i],
                                            puntoDisTri);
        Sino
            Si (punteroNodo->hijos[i]<-2){
                calculaDistanNodoHoja(hojas->at(-punteroNodo->hijos[i] - 3,
                                      puntoDisTri);
            FinSi
        FinSi
    FinPara
Fin
```

Algoritmo 9. Procedimiento Recursivo para el cálculo de la distancia mínima desde un nodo de un subárbol.

```
Procedimiento CalculaDistanciaMin (posNodo, dist){
    punteroNodo = nodos[posNodo];
    Para i=0 hasta 7 con incremento de 1
        Si (punteroNodo->hijos[i]>-1){
            CalculaDistanciaMin(punteroNodo->hijos[i],puntoDisTri);
        Sino
            Si (punteroNodo->hijos[i]<-2){
               CalculaDistanciaMinimaSubArboles(-punteroNodo->hijos[i]-3,
                                                puntoDisTri);
            FinSi
        FinSi
    FinPara
Fin
```

Algoritmo 10. Procedimiento Recursivo para el cálculo de la distancia mínima desde un nodo cargado en memoria.

4.4.1. Resultados obtenidos por los diferentes tipos de test de cálculo de distancias.

De las diferentes opciones planteadas en el apartado anterior para calcular la distancia mínima y la normal del polígono que se encuentra a menor distancia entre un punto y un modelo, se han implementado las dos primeras: la que busca por los hijos hasta encontrar el nodo más cercano y la que baja como máximo hasta un nivel del octree. Para hacer el estudio de tiempos se han utilizado, al igual que en los test de inclusión punto en sólido, las dos distribuciones de puntos: aleatoria y de movimiento continuo. Para cada una de estas distribuciones se ha generado una nube de puntos con un millón de puntos cada uno.

En la *Tabla 27* se muestran los tiempos medios en segundos, para los tres tipos de test, en calcular las distancias y la normal del triángulo más cercano al punto.

En la primera columna se indica el tipo de test que se ha utilizado. El test *Recursivo* se corresponde con la primera opción nombrada anteriormente, la cual busca la distancia mínima entre el punto y el modelo. Los restantes son del tipo de la segunda opción. El test *hojas* calcula la distancia de los puntos que sólo se encuentran dentro de algún nodo hoja. Por último los test denominados *nivel i* y un número calculan la distancia entre los puntos y la envolvente del nivel del EPB-Octree indicado.

```
nodo=0
padre=0
nivel=0
hijo= NodoHijo(nivel, octCode)
Repite mientras nodo>-1
  padre=nodo
  nodo=hijo
  nivel=nivel+1
  hijo= NodoHijo(nivel, octCode)
finRepite
Si nodo<-2
   compruebaCache(nodo, hijo)
   padre=nodo
   nodo=hijo
   nivel=nivel+1
   hijo= NodoHijo(nivel, octCode)
   Repite mientras nodo>-1
      padre=nodo
      nodo=hijo
      nivel=nivel+1
      hijo= NodoHijo(nivel, octCode)
   finRepite
   Si nodo> -3
     Para los 8 nodos hijos
        CalculaDistanciaMinimaDesdePadre(nodo, padre, hijo, dist)
     finPara
   Sino
     calculaDistanNodoHoja(nodo, dist)
   FinSi
Sino
   CalculaDistanciaMin(padre, dist)
FinSi
```

Algoritmo 11. Cálculo recursivo de la distancia mínima entre un punto y un modelo.

	Amazona		Moldura		Lucy	
	Aleatorio	Movimiento continuo	Aleatorio	Movimiento continuo	Aleatorio	Movimiento continuo
Recursivo	0,000831202	0,000000817	0,000330772	0,000007330	0,000878116	0,000014691
Hojas	0,000670450	0,000000115	0,000804068	0,000017791	0,000610465	0,000009901
nivel 1	0,000119280	0,000106559	0,000066915	0,000051660	0,000077537	0,000062787
nivel 2	0,000028657	0,000020381	0,000020776	0,000030863	0,000027440	0,000020440
nivel 3	0,000013861	0,000004199	0,000008924	0,000012602	0,000000000	0,000012549
nivel 4	0,000010514	0,000011224	0,000006580	0,000008158	0,000014312	0,000005373
nivel 5	0,000009346	0,000006947	0,000005961	0,000006109	0,000013030	0,000003517
nivel 6	0,000009346	0,000004722	0,000005835	0,000005292	0,000012699	0,000004406
nivel 7	0,006413401	0,000003001	0,003004701	0,000049817	0,006441030	0,000082120
nivel 8	0,006421768	0,000001907	0,002251486	0,000049186	0,005725785	0,000080736
nivel 9	0,006448134	0,000001622	0,002249638	0,000048691	0,005638451	0,000081751
nivel 10	0,006489810	0,000001650	0,002228513	0,000047527	0,005627434	0,000082357
nivel 11	0,006375349	0,000001645	0,002253691	0,000050826	0,005406070	0,000084955

Tabla 27. Tiempo medio, en segundos, de ejecutar 1 millón de test de cálculo de distancias sobre los tres modelos.

De los datos anteriores se puede observar como los test realizados sobre los puntos generados de manera aleatoria son lo que más tiempo tardan. Esto es debido a que no hacen uso de la caché, con lo cual para cada punto se tiene que cargar el subárbol correspondiente. Otro dato a destacar es que el algoritmo Recursivo en la mayoría de los casos se comporta mejor que los otros test. Un aspecto importante de remarcar es que el test que se queda en el nivel 0 tiene peor comportamiento que los de los niveles sucesivos que se encuentran cargados en memoria. Esto es debido a que la envolvente de nivel 0 está formada por un número mayor de planos que la de los niveles inferiores del árbol. Y como el coste de la función que calcula la distancia de un punto a un plano es elevado, hace que se aumente el tiempo total de ejecución. De hecho el tiempo va disminuyendo conforme se baja de nivel y siempre que los nodos estén por encima o en el nivel de corte del EBP-Octree.

Para hacer el estudio comparativo de resultados se han unificado los datos. Para ello se ha calculado la media de los datos obtenidos sobre los tres modelos, y se muestran en la *Tabla 28*.

Media	Aleatorio	Movimiento continuo
Recursivo	0,000680030	0,000007612
Hojas	0,000694994	0,000009269
nivel 0	0,000087911	0,000073668
nivel 1	0,000025624	0,000023895
nivel 2	0,000007595	0,000009783
nivel 3	0,000010469	0,000008252
nivel 4	0,000009446	0,000005524
nivel 5	0,000009293	0,000004807
nivel 6	0,005286377	0,000044979
nivel 7	0,004799680	0,000043943
nivel 8	0,004778741	0,000044022
nivel 9	0,004781919	0,000043845
nivel 10	0,004678370	0,000045809

Tabla 28. Media de tiempos obtenidos en el cálculo de la distancia entre un punto y un modelo.

Haciendo un estudio comparativo de los tiempos de la *Tabla 28* cuando se sigue una distribución de puntos aleatoria, se puede observar cómo el test *Recursivo* tiene un peor comportamiento que los test que llegan hasta el nivel de corte del EPB-Octree. Esto se debe a que no tienen que cargar de disco la información de los nodos. En los niveles en los que se tienen que cargar los subárboles, los tiempos aumentan considerablemente. Cabe recordar que la distribución de puntos usada se corresponde con el peor de los escenarios posibles, ya que los test no están pensados para trabajar con puntos que no sigan una ruta continua en el espacio-tiempo.

Cuando los puntos siguen una distribución de movimiento continuo se observa que el método *Recursivo* es el más eficiente. Con respecto a los métodos que calculan la distancia hasta un nivel, cabe destacar el alto valor del tiempo de cálculo del test que se queda en la envolvente 0. Esto ocurre porque su envolvente está formada por muchos más planos que las envolventes de los niveles inferiores, multiplicándose así el coste de calcular la distancia del punto a cada uno de estos planos. Otro dato significativo es que el tiempo que se tarda en calcular la distancia en los niveles inferiores que no se encuentran cargados en memoria principal es muy similar.

Si se comparan los resultados conseguidos para las dos distribuciones de puntos, se puede observar que los tiempos de respuesta obtenidos sobre las distribuciones de puntos que siguen una continuidad espacio-tiempo (la de movimiento continuo) superan con creces los 1KHz establecidos como estándar para poder trabajar con dispositivos hápticos.

En la *Figura 90*, *Figura 91* y *Figura 92* se pueden ver los mapas de distancias sobre la superficie obtenidos para los modelos de la Amazona Herida, Lucy y Moldura. La paleta de colores utilizada es:

- azul para los más lejanos,
- verde para los que se encuentran cerca de la superficie,
- rojo para los que están casi pegados a la superficie y
- negros para los que están sobre la superficie.

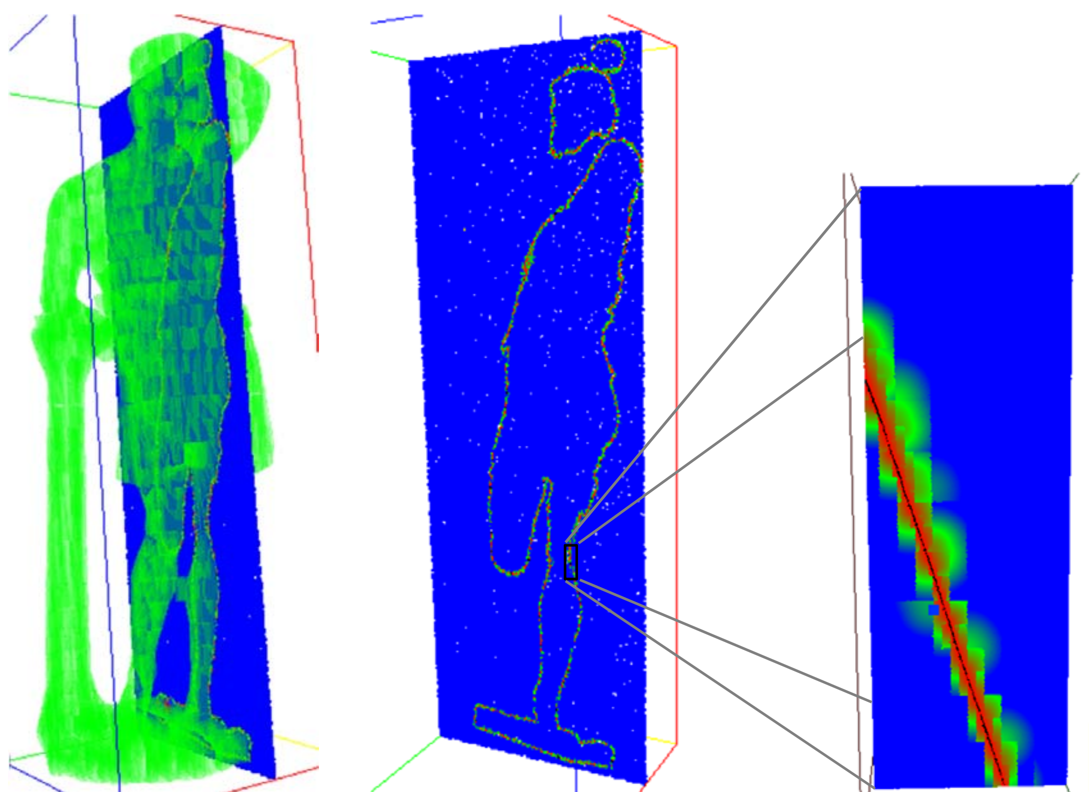


Figura 90. Mapa de distancias para el modelo Amazona Herida de 28 M.P.

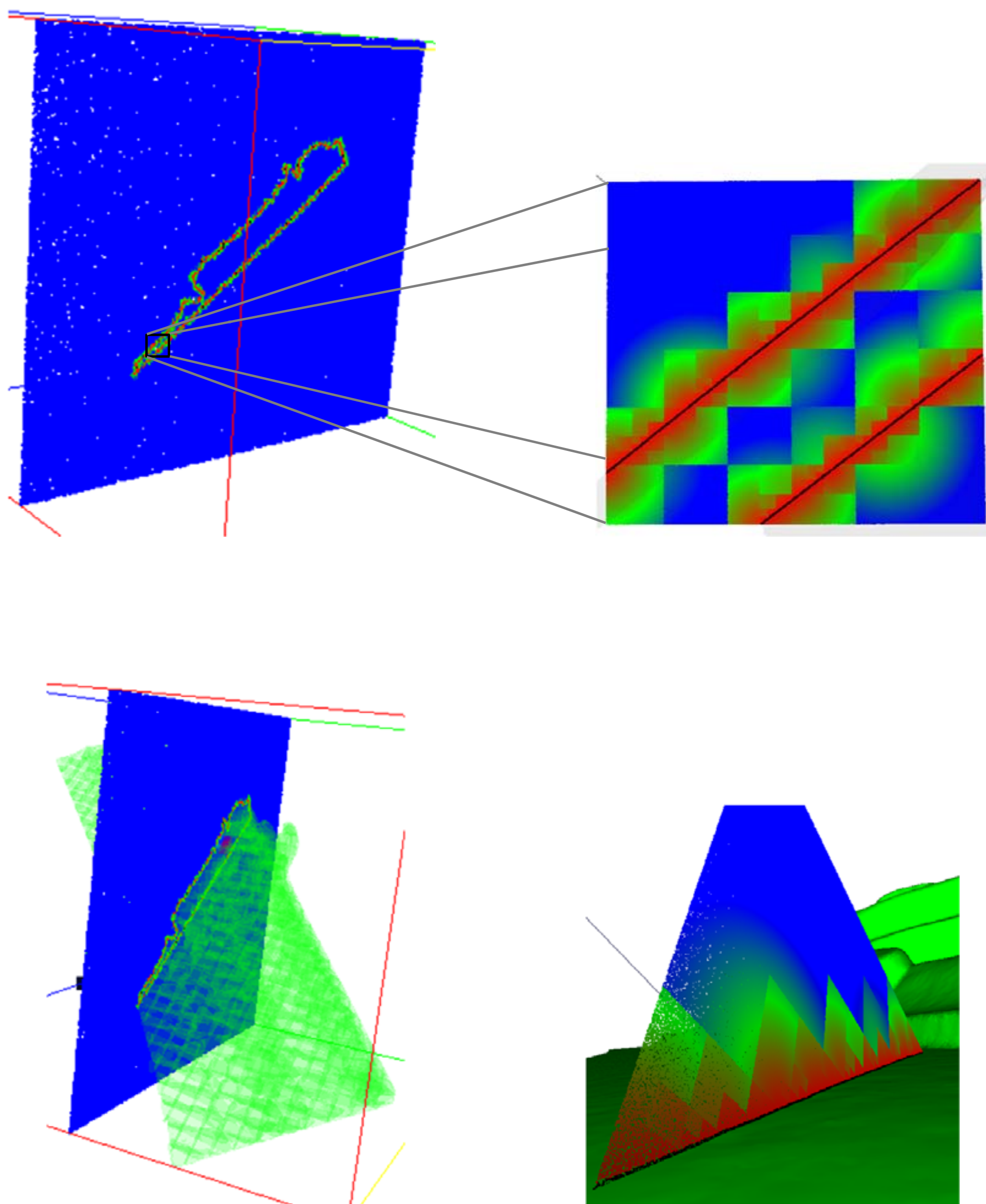


Figura 91. Mapa de distancias para el modelo Moldura de 26 M.P.

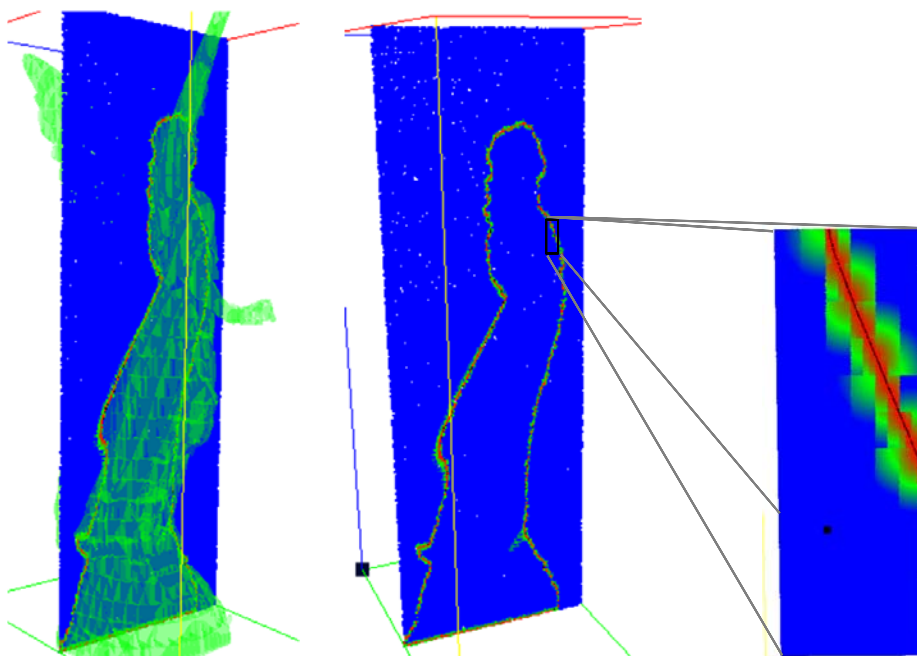


Figura 92. Mapa de distancias para el modelo, Lucy de 28 M.P.

4.5. Utilización de los EBP-Octree en aplicaciones con interacción háptica.

Como ya se ha visto en el apartado 4.1.3.1, los datos obtenidos en los test realizados para calcular la distancia y la normal entre un modelo y un punto son válidos para poder ser utilizados en un dispositivo háptico de tipo puntero, ya que su respuesta es mayor de los 1 kHz que es la frecuencia mínima de refresco que se recomienda para dichos dispositivos. Por ello se creó una pequeña aplicación para simular la utilización de modelos formados por varias decenas de millones de polígonos junto con un dispositivo háptico con 6 grados de libertad. Esta herramienta devolvía en todo momento la distancia mínima y el ángulo de contacto del puntero con respecto al modelo cargado.

La aplicación creada permite trabajar seleccionando uno de los dos modos programados de comportamiento del puntero:

- Con penetración del puntero sobre el modelo.

- Sin penetración del puntero sobre el modelo.

En el primero de los casos el puntero cambia de color conforme se acerca o se aleja del modelo. La paleta de colores utilizada es la misma que la vista en el apartado anterior. En este modo el puntero puede atravesar la frontera del modelo colocándose en el interior del mismo.

En el segundo de los modos de trabajo la aplicación intenta emular el funcionamiento de las presentadas por Arthur D. Gregory y otros (Gregory, Ehmann, and Lin 2000) y por Laehyun Kim y otros (L. Kim, Sukhatme, and Desbrun 2003), las cuales utilizan un rotulador para pintar en la superficie de los modelos. Para ello el puntero se comporta igual que un lápiz, de manera que cuando se toca la superficie del modelo se pintan una serie de puntos sobre los polígonos que toca. Así cuanto mayor sea la presión que se ejerce, mayor será el número de puntos que se pinten. La aplicación permite seleccionar el color de la tinta con la que se escribe. En este modo de trabajo el puntero no puede atravesar la frontera del modelo.

En la *Figura 93* se puede apreciar el resultado obtenido tras utilizar la aplicación de simulación háptica creada y pintar sobre el ojo de la Amazona Herida, cambiando el color de la tinta del puntero.

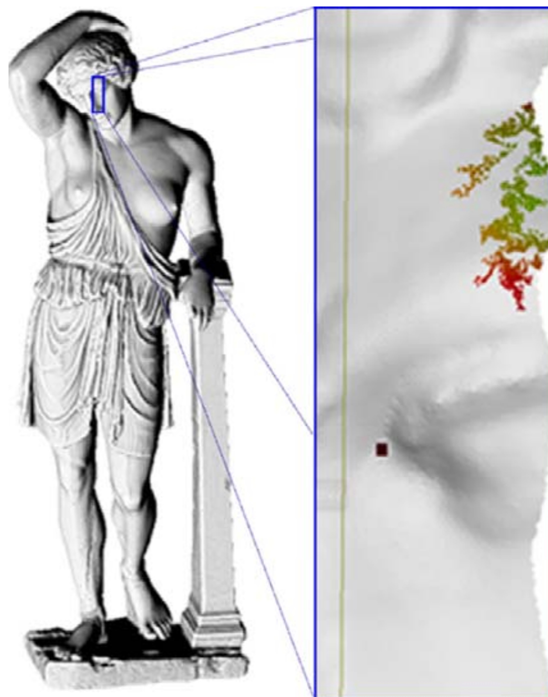


Figura 93. Amazona Herida con el ojo pintado por la aplicación de simulación háptica.

5. DETECCIÓN DE COLISIONES ENTRE DOS MODELOS

En este capítulo se presentan los algoritmos diseñados para el cálculo de la detección de colisiones entre dos modelos utilizando los EBP-Octree. Para demostrar la eficiencia de la estructura se hace un estudio de tiempos de cálculo, en el que se simulan varios escenarios sobre los que se mueven varios modelos.

Como ya se ha comentado en la sección 2.2, la detección de colisiones ha sido y es un tema de investigación muy importante en el campo de la informática gráfica. Existe una gran demanda de algoritmos para resolver este problema, que sean cada vez más rápidos y robustos.

En los trabajos encontrados en la literatura específica una gran parte de los algoritmos solamente detectan si se produce o no colisión entre modelos.

Uno de los objetivos principales de esta tesis ha sido el de diseñar algoritmos que cuando se produce colisión entre modelos nos devuelven exactamente en qué polígonos de los modelos se produce dicha colisión (ver *Figura 94*). En estos

algoritmos se hace un buen uso de la CPU de los ordenadores y permiten trabajar con modelos formados por un gran número de polígonos de manera eficiente.

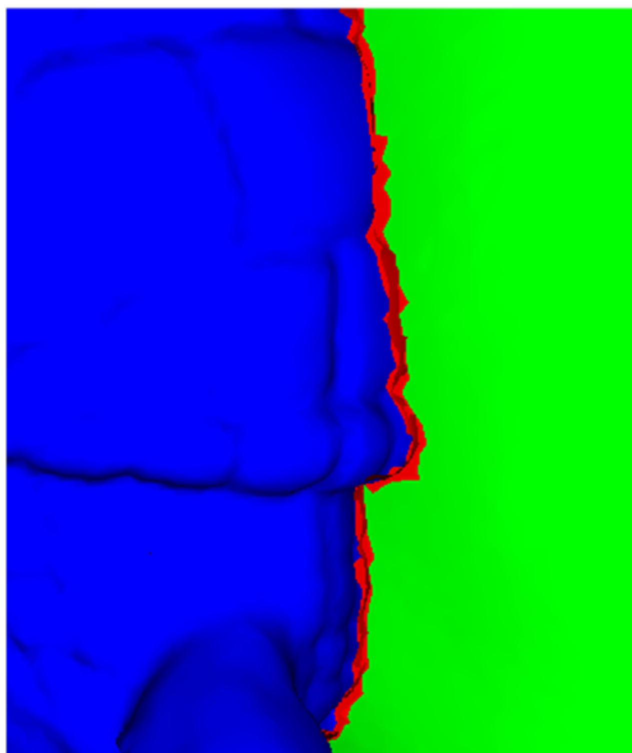


Figura 94. En verde primer modelo, en azul segundo modelo y en rojo los polígonos que intersecan.

5.1. Introducción.

La detección de colisiones entre un conjunto de objetos del mundo toma como entrada una geometría de colisión para cada uno de los objetos y calcula todos los puntos de contacto entre estos, devolviendo además, para cada punto, la normal y la profundidad. Debido a esto la detección de colisiones se convierte en un proceso complejo, ya que los objetos están formados por un gran número de polígonos con una forma compleja. Otro dato a tener en cuenta es que las comprobaciones que se tienen que realizar entre los objetos crecen de forma cuadrática con respecto al número de estos. Por todos motivos esto es fundamental optimizar el proceso de cálculo de colisiones entre objetos. La idea más utilizada para optimizar este proceso

es la de dividir los cálculos en dos etapas:

- Etapa amplia (Broad phase): En esta etapa se filtran los objetos seleccionando sólo aquellos que pueden colisionar. Para ello no se suele utilizar la forma real de los objetos, sino que se utilizan formas simplificadas que envuelven al objeto. Estas formas permiten comprobar rápidamente si dos objetos intersecciona o no.
- Etapa Reducida (Narrow phase): Esta etapa toma como entrada las parejas de objetos obtenidas en la etapa anterior y comprueba de manera exacta si los objetos colisionan, devolviendo los puntos exactos de contacto junto con la normal y la profundidad de cada uno de ellos.

En la etapa amplia las técnicas más utilizadas para filtrar los objetos son:

- Las que realizan un particionamiento espacial (ver sección 2.3.2.1.1).
- Las que simplifican el objeto (ver sección 2.3.2.3).
- Las que utilizan jerarquías de volúmenes envolventes (ver sección 2.3.2.4)

Para la etapa reducida, donde se calcula la zona exacta de contacto entre los objetos, las principales técnicas utilizadas son:

- Métodos basados en árboles y grids (Palmer and Grimsdale 1995; J. D. Cohen et al. 1995; Gottschalk et al. 1996; Gregory et al. 1999).
- Métodos que usan diagramas de Voronoi (Breen et al. 2001; Hoff et al. 1999).
- Métodos de propagación de distancias (Sethian 1996).

5.2. Detección de colisiones entre dos modelos utilizando EBP-Octree.

Para detectar la colisión entre modelos formados por varias decenas de millones de polígonos, se parte de una escena formada por varios modelos. Cada uno de los modelos que la componen se pueden encontrar en movimiento o en reposo. El tamaño de la escena se va recalculando de manera automática y progresiva conforme se van añadiendo más modelos a esta, de manera que no se sobrepasen los límites hardware del ordenador donde se ejecuta.

La estructura de los EBP-Octrees permite detectar la colisión entre modelos basándose en las dos etapas vistas en la sección 5.1. En primera etapa de ellas, la broad phase, se utiliza la caja envolvente de los modelos para filtrar todos aquellos en los que su caja envolvente no colisione. En la segunda etapa se pueden detectar todos los polígonos que colisionan exactamente. Para agilizar este proceso se utiliza tanto la jerarquía de volúmenes envolventes como la indexación espacial de los nodos y solamente se comprueban las colisiones entre los polígonos reales de los modelos incluídos en los nodos hoja cuyas envolventes colisionan.

Las escenas son almacenadas en archivos para poder cargarlas y reproducirlas cuando se desee. La información guardada de una escena está formada por los modelos que la forman y las rutas de movimiento que seguirían cada uno de los modelos por separado.

5.3. Test de detección de colisiones entre modelos utilizando EBP-Octree.

Para comprobar de la detección de colisiones dentro de una escena se diseña un algoritmo que compara la colisión entre dos modelos. Este algoritmo se ejecuta tantas veces como elementos tiene el conjunto formado por la combinación, sin repetición, de modelos tomados de dos en dos.

El algoritmo recibe seis parámetros:

- los dos modelos, tal y como se insertaron inicialmente en la escena (con la caja envolvente alineada con los ejes de coordenadas), es decir sin aplicar ninguna transformación,
- dos matrices de transformación, que se utilizan para calcular la posición del modelo dentro de la escena en un momento,
- y dos banderas, indicando si a los modelos hay que aplicarle la transformación o no.

Como los EBP-Octree calculados para cada modelo se encuentran alineados a los ejes de coordenadas, los modelos que están en movimiento o se han desplazado con respecto a la posición inicial, necesitan tener asociada una matriz de transformación. Para ello se tienen en cuenta las tres posibles combinaciones de casos entre los dos modelos:

- Si no están en movimiento o desplazados, entonces se detecta la colisión directamente sobre los dos EBP-Octree.
- Si uno de los modelos está en movimiento o desplazado y el otro no, entonces todos los puntos que forman las envolventes del modelo que está en movimiento o desplazado se tienen que multiplicar por la matriz de transformación de ese modelo, antes de detectar la colisión.
- Si los dos modelos están en movimiento o desplazados entonces se tiene que calcular la matriz inversa de transformación de uno de ellos y se multiplica por la matriz de transformación del otro modelo, obteniéndose una nueva matriz, que se utiliza para multiplicar todos los puntos de las envolventes que forman el EBP-Octree del primer modelo (ver *Ecuación 7*). Utilizar la matriz de transformación calculada anteriormente permite tratar este caso, en el que se tienen los dos modelos no alineados con los ejes, como el caso anterior en el que un modelo si está alineado en el estado inicial con los ejes y el otro no. Esto simplifica bastante las operaciones que hay que hacer para el cálculo de la colisión, ya que sólo se necesitan multiplicar las envolventes calculadas en el EBP-Octree de uno de los dos modelos por la matriz de transformación calculada.

$$M_1 = Matriz\ de\ transformación\ del\ modelo\ 1.$$

$$M_2 = Matriz\ de\ transformación\ del\ modelo\ 2.$$

$$M_{resultante} = M_2^{-1} * M_1$$

Ecuación 7. Cálculo de la matriz de posición relativa entre dos modelos.

Como los modelos que forman la escena pueden estar escaneados a diferentes resoluciones o tener diferentes tamaños, los EBP-Octree que representan a los modelos tienen diferentes números máximos de niveles y distintos niveles de corte. Esto implica a que se tengan dos posibles casos a la hora de detectar colisiones entre dos objetos:

- Los dos modelos tienen similar tamaño, es decir, coinciden sus niveles máximos y de corte (ver *Figura 95*).
- Uno de los modelos tiene sus niveles máximo y de corte mayores que el otro (ver *Figura 96*).

Para poder detectar la colisión entre dos modelos se necesitarían dos algoritmos

diferentes: uno cuando los modelos son similares y otro cuando no lo son. En el segundo de los casos, a la hora de llamar al algoritmo, se debe poner en primer lugar el que tenga un mayor número de niveles.

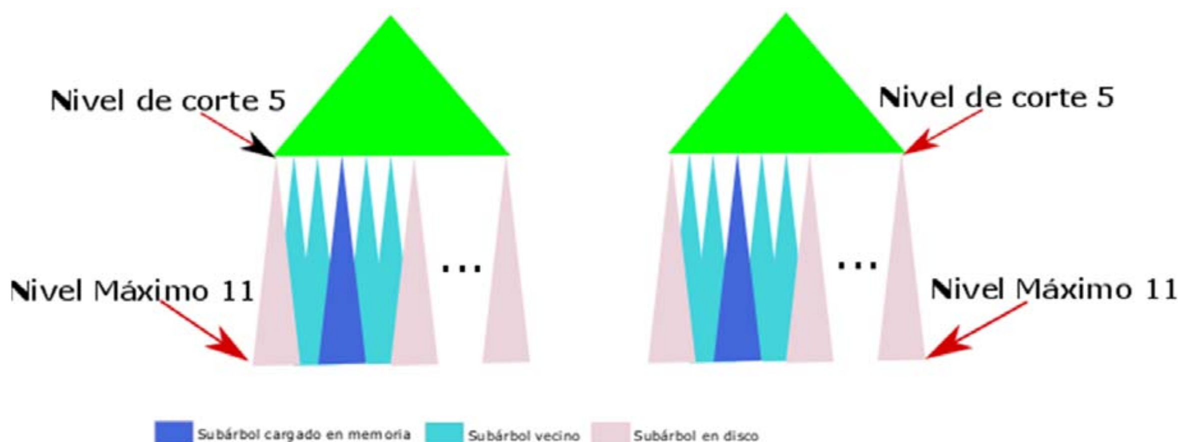


Figura 95. EBP-Octrees con similares niveles de corte y máximos.

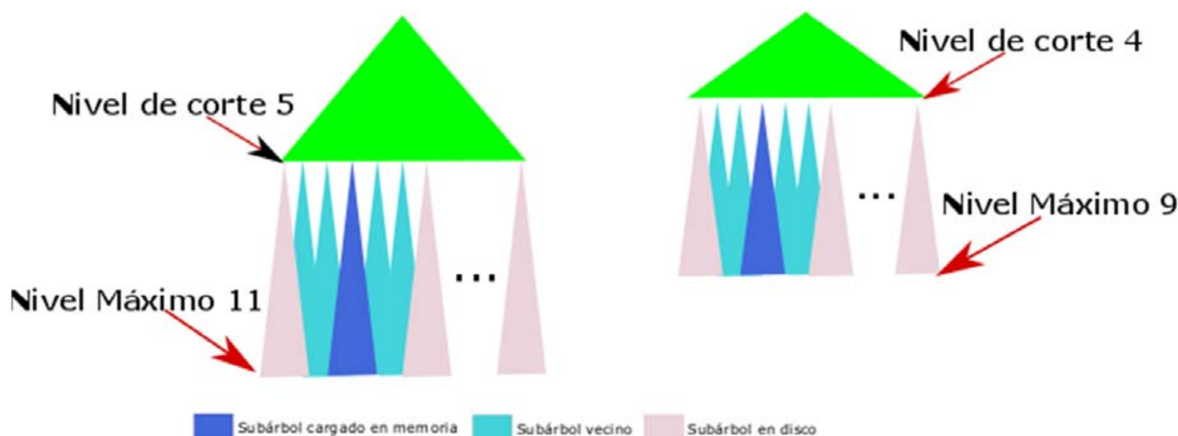


Figura 96. Primer EBP-Octree con nivel de corte 5 y máximo 11 y segundo con nivel de corte 4 y máximo 9.

5.4. Algoritmos de detección de colisiones entre modelos.

La detección de colisiones entre dos modelos se ha resuelto planteando el problema como si se tratase de una máquina de estados. En cada estado se miran las posibles colisiones entre las dos envolventes de los nodos de los modelos en el mismo nivel de profundidad del EBP-Octree.

Los tipos de nodos se clasifican dependiendo de cómo se acceda a ellos. Esto se tiene que realizar así porque los nodos se almacenan en el EBP-Octree de manera diferente. Por ejemplo, unos nodos pueden estar en memoria principal y otros en disco. Los diferentes tipos de nodos que se tienen son:

- A: Nodo del árbol que está cargado en memoria.
- NS: Nodo gris separador. Este nodo pertenece al último nivel de nodos que están cargados en memoria.
- SA: Nodo del subárbol. El que no está cargado en memoria.
- PNH: Nodo padre de un nodo hoja.
- NH: Nodo hoja.

En cada estado de la maquina se gestionan dos colas, una con los nodos del nivel actual en el que se está comprobando si hay o no colisión y otra en la que se van guardando los nodos del siguiente nivel que son susceptibles de tener colisión.

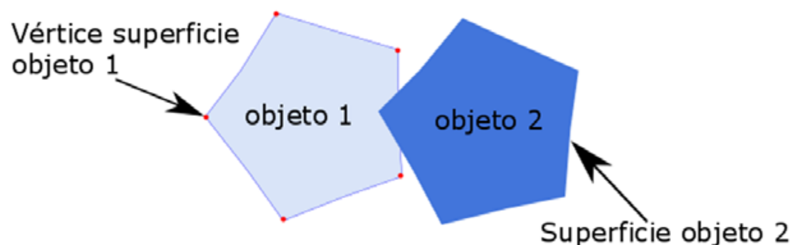
En cada una de las colas se guardan una serie de parejas de punteros, cada uno de los cuales hace referencia a un nodo de cada uno de los modelos en los que se está comprobando la colisión.

Entre estado y estado de la maquina se almacenan en una cola los posibles nodos que son susceptibles de tener colisión en el siguiente nivel de profundidad en el EBP-Octree.

Para detectar posibles colisiones entre dos modelos se comprueba si existe colisión entre las envolventes de los nodos raíz. Para ello las referencias de los nodos de nivel cero de los dos modelos se añaden a la cola de posible colisión. El algoritmo utiliza las dos envolventes en dicho nivel y comprueba si una envolvente colisiona con la otra, utilizando para ello mapas de distancias. Para ver si dos envolventes colisionan se calcula la distancia de todos los vértices que forman una envolvente con todos los planos que forman la envolvente del otro modelo. Para que dos envolventes colisionen se tiene que cumplir que, por lo menos, un vértice de la envolvente esté dentro de la envolvente del otro modelo. Un vértice está dentro de la envolvente del otro modelo si la distancia de ese punto a todos los planos que forman la segunda envolvente es negativa. Al comparar una envolvente con todos los puntos que forman la otra envolvente puede ser que devuelva que no hay colisión cuando si la hay. Para evitar estos falsos negativos, siempre que no hay colisión se tiene que hacer

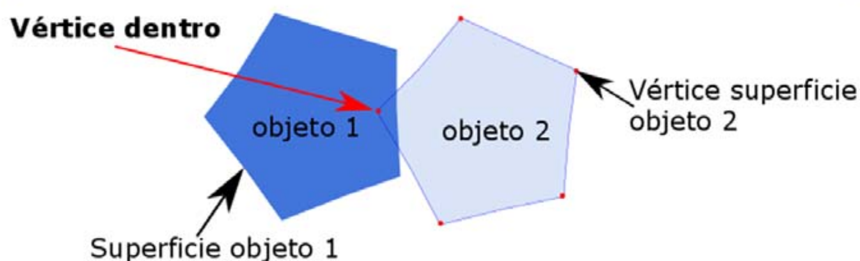
la comprobación al revés, es decir, hay que comprobar que todos los vértices de la envolvente de la que antes se tomaron los planos no están dentro de la envolvente de la que antes se tomaron los vértices. En la *Figura 97* se puede ver un ejemplo para un caso en 2 dimensiones.

Primera comprobación: Vertices objeto 1 dentro de superficie objeto 2



Ningún vértice objeto 1 dentro de objeto 2. **No colisionan.**

Segunda comprobación: Vertices objeto 2 dentro de superficie objeto 1



Un vértice objeto 2 dentro de objeto 1. **Colisionan**.

Figura 97. Detección de colisiones entre dos objetos en 2D.

Una vez hechas estas dos comprobaciones, en los dos sentidos, si todos los vértices están fuera del otro modelo, se puede decir que los dos modelos no colisionan.

Cuando hay colisión entre las dos envolventes de un nivel se añaden a la cola de comprobación de colisión del siguiente nivel todas las posibles combinaciones tomadas de dos en dos y sin repetición, entre los ocho nodos hijos del siguiente nivel con los ocho nodos hijos del siguiente nivel del otro modelo.

Este proceso se repite nivel a nivel hasta que se llega a los nodos del último nivel. Si continúa habiendo colisión entonces se toman los polígonos reales de los modelos que cortan a ese nodo hoja y se hace una comprobación para ver si intersectan con los polígonos del otro modelo. Si hay intersección entonces se puede decir que los dos modelos están colisionando.

El seudocódigo donde se detalla todo este proceso se puede ver en el *Algoritmo 12*.

```
Procedimiento calculoColisiones(bpo1, bpo2, M1, M2, nivel, nodo1, nodo2)
Inicio
   Si nivel < nivel_maximo
      Si colisionanEnvolventes(M1, M2, nodo1, nodo2)
         Entonces
              Para i=1 hasta 8
                 Para j=1 hasta 8
                     CalculoColisiones(bpo1, bpo2, M1, M2, nivel+1,
                                       nodo1.hijos(i), nodo2.hijos(j))
                FinPara
              FinPara
         FinSi
     Sino
         colisionanPoligonosReales(M1,M2,nodo1,nodo2)
  FinSi
Fin
```

Algoritmo 12. Seudocódigo recursivo del cálculo de colisiones entre dos modelos similares.

Tal y como se ha comentado en la sección 5.3, para poder hacer la detección de colisiones entre dos modelos se han utilizado dos algoritmos, cada uno de los cuales emula una máquina de estados. Cuando se comparan modelos que tienen el mismo número de niveles, esta máquina es mucho más simple que la que se utilizan cuando hay dos modelos con distinto número de niveles. Esto es debido a que no se accede a los nodos de igual manera. Por ejemplo, un nodo puede estar almacenado en memoria, por ser un nodo de los primeros niveles del EBP-Octree y el otro puede estar almacenado en los archivos, por ser un nodo que está en un subárbol, con lo cual antes de acceder a él se tiene que cargar en memoria principal. A continuación se describen estos algoritmos.

5.4.1. Algoritmo de detección de colisiones para modelos con igual número de niveles.

Para resolver este problema se utiliza una máquina de estados con cinco estados que se ejecutan secuencialmente, aunque dos de ellos se pueden repetir de una a *n* veces.

Los estados se clasifican en función del tipo de nodo que representen dentro del EBP-Octree. En este caso, como los dos modelos tienen el mismo número de niveles de resolución, la máquina de estados se simplifica mucho.

En el primer estado, en las celdas de la cola que almacena los nodos a chequear (los

que son susceptibles de tener colisión en el primer nivel), se almacenan dos referencias a dos nodos que se encuentran en la memoria principal del ordenador. Cuando se terminan de procesar todas las celdas de la cola se comprueba si los nodos de la cola de nodos del siguiente nivel son separadores del árbol. Si lo son, entonces se pasa al siguiente estado de la máquina, y si no se queda en este mismo estado intercambiando las colas mientras los nodos de nivel estén almacenados en memoria principal.

Para este algoritmo los colores de los nodos dentro de la máquina de estados, pueden ser de dos tipos: los de color verde que representan estados que se pueden repetir una o *n* veces y los de color azul los que solo se repiten una vez.

En la *Figura 98* se muestra la máquina de estados que utiliza el algoritmo de detección de colisiones cuando los dos modelos tienen el mismo número de niveles.

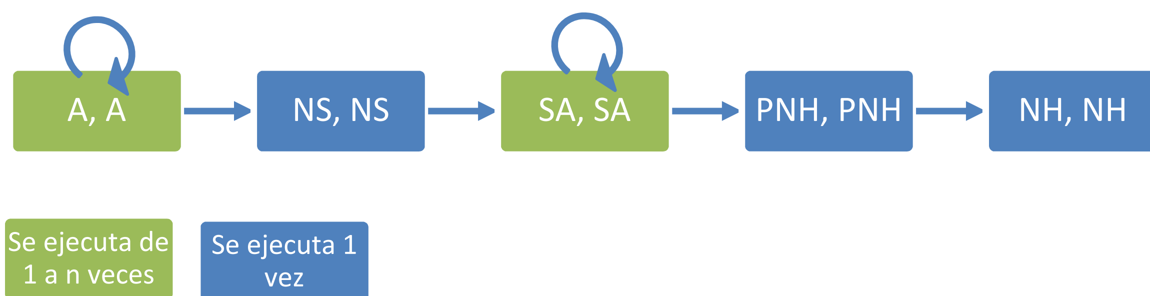


Figura 98. Máquina de estados para calcular colisión entre dos modelos con igual número de niveles en el EBP-Octree.

El proceso de ejecución de este algoritmo es el siguiente. Comienza añadiendo a la cola de comprobación de posibles colisiones las referencias a los nodos cero de los dos modelos, con lo cual se comparan las envolventes de nivel cero. Si hay colisión se añaden a la cola de posibles colisiones en el siguiente nivel las parejas de nodos hijos resultantes de la combinación sin repetición de los mismos. Este proceso acaba cuando se han procesado todos los elementos de la cola, es decir, cuando esta no contenga ningún elemento. Cuando la cola que almacena las parejas de nodos de un nivel que son susceptibles de colisión se queda vacía se produce un cambio de estado en la máquina, pasando al siguiente. El cambio de estado provoca un intercambio entre las dos colas, de manera que la que almacena las parejas de posibles colisiones

en un nivel pasa a ser la cola de posibles colisiones en el siguiente nivel y la cola de posibles colisiones pasa a ser la cola de chequeo de colisiones.

Este proceso se repite mientras la cola de posibles colisiones en el siguiente nivel tiene elementos, lo cual significa que no hay colisión, o cuando se llega al nivel máximo de profundidad de los EBP-Octree. En este caso se comprueba la envolvente de los nodos hoja y si sigue habiendo colisión, entonces se examina si hay colisión entre los polígonos reales de la superficie de los modelos que cortan a los nodos hoja.

En el *Algoritmo 18* se puede ver el seudocódigo que detalla todo este proceso.

```
Procedimiento EstadoMaquinaA_A(VAR nivel, VAR cola, VAR colaSepa)
    Mientras (nivel < NivelCorte)
            Mientras (cola no este vacia)
                Sacar de cola la pareja de nodos
                Si las envolventes de la pareja de nodos colisionan
                    Entonces añade a la colaSepa las parejas de nodos hijos
                finSi
            finMientras
            nivel++;
            colaAux = cola;
            cola = colaSepa;
            colaSepa = colaAux;
    finMientras
finProcedimiento
```

Algoritmo 13. Seudocódigo del estado de la máquina A, A.

```
Procedimiento EstadoMaquinaNS_NS(VAR cola, VAR colaSepa)
        Mientras (colaSepa no este vacia)
            Sacar de colaSepa la pareja de nodos
            Si las envolventes de la pareja de nodos colisionan
                Entonces añade a la colaSepa las parejas de nodos hijos
            finSi
        finMientras
finProcedimiento
```

Algoritmo 14. Seudocódigo del estado de la máquina NS, NS.

```
Procedimiento EstadoMaquinaSA_SA(VAR nivel, VAR cola, VAR colaSepa)
        Mientras (nivel<MAX_NIVEL-1)
            Mientras (cola no este vacia)
                Sacar de cola la pareja de nodos
                Cargar en memoria la pareja de nodos
                Si las envolventes de la pareja de nodos colisionan
                    Entonces añade a la colaSepa las parejas de nodos hijos
                finSi
            finMientras
            liberar de memoria la pareja de nodos
            nivel++;
            colaAux = cola;
            cola = colaSepa;
            colaSepa = colaAux;
        finMientras
finProcedimiento
```

Algoritmo 15. Seudocódigo del estado de la máquina SA, SA.

```
Procedimiento EstadoMaquinaPNH_PNH(VAR cola, VAR colaSepa)
         Mientras (cola no este vacia)
                Sacar de cola la pareja de nodos
                Cargar en memoria la pareja de nodos
                Si las envolventes de la pareja de nodos colisionan
                    Entonces añade a la colaSepa las parejas de nodos hijos
                finSi
         finMientras
finProcedimiento
```

Algoritmo 16. Seudocódigo del estado de la máquina PNH, PNH.

```
Procedimiento EstadoMaquinaNH_NH(VAR cola, VAR colaSepa)
        Mientras (cola no este vacia)
                Sacar de cola la pareja de nodos
                Cargar en memoria la pareja de nodos
                Calcular polígonos que se cortan entre la superficie real de
                 los dos nodos
        finMientras
        liberar de memoria la pareja de nodos
finProcedimiento
```

Algoritmo 17. Seudocódigo del estado de la máquina NH, NH.

```
Función ColisionModelosIgulaes (bpo1, bpo2, Var numColisiones) : Lógico
        TNumNodos numNodos;
        colaPrioridad cola, colaSepa, colaAux;
        numNodos.nodo1 = bpo.root;
        numNodos.nodo2 = bpo1.root;
        cola->push(numNodos);
        nivel = 0;
        EstadoMaquinaA_A(nivel, cola, colaSepa)
        colaAux = cola;
        cola = colaSepa;
        colaSepa = colaAux;
        EstadoMaquinaNS_NS(cola, colaSepa)
        nivel++;
        EstadoMaquinaSA_SA(nivel, cola, colaSepa)
        EstadoMaquinaPNH_PNH(cola, colaSepa)
        EstadoMaquinaNH_NH(cola,colaSepa)
        Si (numero de colisiones == 0 )
            devolver false;
        Sino
            devolver true;
        finSi
fin
```

Algoritmo 18. Cálculo de colisiones entre dos modelos con igual número de niveles en el EBP-Octree.

La búsqueda de colisiones entre dos modelos a través de este algoritmo se puede realizar de dos maneras diferentes. En una de ellas, el algoritmo se detiene cuando se encuentra que hay una colisión entre los dos modelos. En la otra, a la finalización del algoritmo, se devuelve una lista con todas las parejas de polígonos que producen colisión en ese momento dado.

Para este algoritmo se ha implementado una variante que funciona igual hasta que se llega al nivel de separación. En este momento, en vez de mirar todos los nodos del siguiente nivel, lo que hace es cargar los dos subárboles de los nodos que son susceptibles de colisión y se comprueba, si hay colisión o no, siguiendo el mismo procedimiento que en la parte inicial del árbol. Este algoritmo se detiene cuando se encuentra una colisión o puede seguir comprobando todas las posibles colisiones dando una lista de parejas con todos los polígonos que tienen colisión.

5.4.2. Algoritmo detección de colisiones para modelos con distinto número de niveles.

El algoritmo para la detección de colisiones entre dos modelos que tienen distinto número de niveles en el EBP-Octree también, se implementa con una máquina de estados. Al tener los modelos distinto número de niveles, las combinaciones de tipos de nodos son mayores, con lo cual el número de estados es mayor. Para simplificar la máquina de estados, los modelos con los que se trabajan se ordenan, poniendo en primer lugar el modelo con menor número de niveles. Teniendo en cuenta este criterio, la máquina de estados resultante se puede ver en la *Figura 99*.

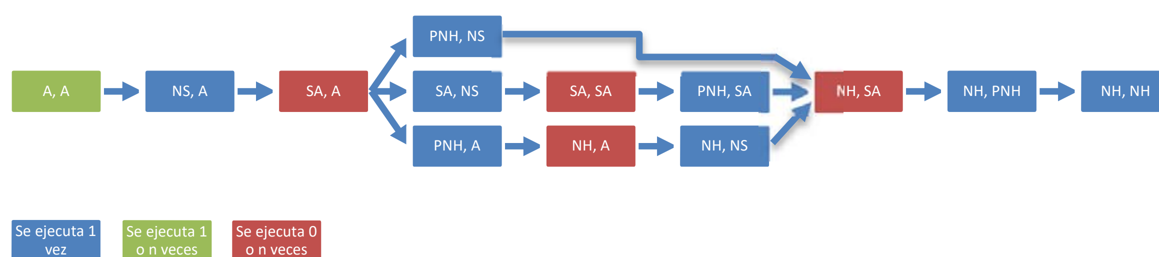


Figura 99. Máquina de estados para calcular colisión entre dos modelos con distinto número de niveles en el EBP-Octree.

Para poder emular todos los casos se ha tenido que añadir un nuevo tipo de estado: los estados representados por color rojo, son por los que no se pasa saltando al siguiente estado o se puede pasar *n* veces.

El funcionamiento es el mismo que en el caso anterior. Se tienen dos colas, una con los nodos con posibles colisiones en un nivel y otra con los nodos con posibles colisiones en el siguiente nivel. La cola se inicializa con los dos nodos de nivel 0. Las colas se van intercambiando en cada estado de la máquina, hasta no encontrar colisión o hasta llegar al nivel máximo de los dos modelos. Cuando se llega al nivel máximo del modelo más pequeño, las siguientes comparaciones se hacen entre los nodos de nivel hoja del primer modelo con los nodos de los niveles inferiores del otro modelo. Este proceso se repite hasta que se alcanza el nivel máximo del segundo modelo, el que tiene mayor número de niveles. Aquí se comparan las envolventes de los nodos que pertenecen a los dos últimos niveles. Si sigue habiendo colisión, entonces se comprueba si hay intersección entre los polígonos reales que forman las superficies de los modelos y cortan a esos nodos. Si se cortan es porque hay colisión.

Los criterios para finalizar la ejecución son los mismos que en el algoritmo anterior: calcula solamente si hay colisión o calcula una lista con todos los polígonos que colisionan.

5.5. Optimización sobre los algoritmos de detección de colisiones.

Tras hacer un estudio de distribución de tiempos en todas las funciones implicadas en el cálculo de las colisiones, se observó que las que se encargaban de cargar los nodos de disco a memoria eran las que consumían gran parte del tiempo total. Otro dato relevante del estudio fue que el número de veces que se cargaba un mismo nodo, de los que se mantenían en disco, rondaba una media de 6. Para evitar esto se realizó una mejora en los algoritmos anteriores que conseguía cargar una sola vez cada nodo. En primer lugar se analizó cómo se introducían las diferentes parejas de números de nodos en las distintas colas que utilizaba la máquina de estados. Se observó que estos no seguían ningún orden. De ahí surgió la idea de que si se insertaban en las colas siguiendo un orden, este se podía aprovechar a posteriori para evitar tener que cargar dos veces los nodos. El problema apareció cuando se tuvo que decidir cómo ordenar las diferentes celdas de las colas ya que estas estaban compuestas por una pareja de valores (ver *Figura 100*).

1	2	3	4	5	6	7	8
5 - 4	4 - 2	5 - 5	5 - 6	4 - 5	6 – 2	5 - 9	6 - 6

Figura 100. Cola sin sus celdas ordenadas.

La solución que se planteó fue el ordenar las celdas de la cola por el valor de número del primer nodo de cada pareja. La lista anterior ordenada se puede ver en la *Figura 101*.

1	2	3	4	5	6	7	8
4 - 2	4 - 5	5 - 4	5 - 5	5 - 6	5 – 9	6 - 2	6 - 6

Figura 101. Cola ordenada por el valor del primer número de cada pareja.

Con esta ordenación se tenía resuelto la mitad del problema, ya que los nodos a los que hacía referencia el primer número de cada pareja se cargaban en memoria y se mantenían en ella mientras eran utilizadas por las parejas posteriores. Por ejemplo, el nodo 4 se carga una sola vez en memoria y se utiliza dos veces una en la primera celda y otra en la segunda.

Para solucionar el problema de no tener que cargar más de una vez los nodos referenciados en segundo lugar dentro de cada celda se pensó en utilizar una estructura de datos que permitiera ordenarlos y acceder a ellos de una manera eficiente. La solución propuesta fue utilizar una tabla hash de direccionamiento abierto y con contador de ocurrencias. En esta tabla hash se insertan los nodos correspondientes al número segundo de cada pareja. Para cada elemento insertado en la tabla se tiene asociado un contador de ocurrencias. De esta forma cada nodo se lee una vez y se mantiene en memoria mientas el contador de ocurrencias no llegue a cero, en cuyo caso se elimina de memoria, ya que este no se tiene que volver a utilizar más. En la *Figura 102* se muestra la tabla hash obtenida del ejemplo anterior antes de eliminar ningún nodo.

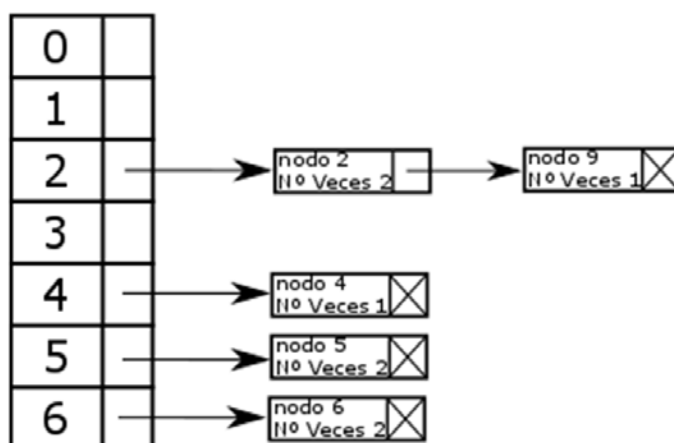


Figura 102. Tabla hash con los nodos del segundo modelo.

5.6. Estudio de tiempos obtenidos en el cálculo de la detección de colisiones.

El estudio de tiempos se ha llevado a cabo utilizando los algoritmos detallados en

los apartados anteriores, tanto para los optimizados como para los que no lo están. Para ello se ha creado una aplicación que monta una escena sobre la que se cargan los modelos. Esta herramienta permite asociar una ruta de movimiento a cada uno de ellos o bien generar diferentes tipos de rutas de movimiento. En las subsecciones siguientes se explican con más detalle los criterios que se han tenido en cuenta en su implementación.

Los tests utilizados para hacer el estudio de tiempos se han dividido en dos grupos:

- los que solamente calculan si dos modelos colisionan y
- los que además de detectar si dos modelos colisionan también devuelven una lista con todos los polígonos en los que se producen las colisiones.

Para cada uno de los dos grupos se han realizado una serie de test, agrupados en dos tipos: los que utilizan dos modelos iguales, es decir, con el mismo número de niveles en los EBP-Octree y los que utilizan dos modelos con diferente número de niveles. Se han creado cuatro escenarios diferentes, en los que en cada uno se han incluido dos modelos. Los modelos utilizados en cada escenario se pueden ver en la *Tabla 29*. Todos los test funcionan de la misma forma: animando los dos modelos de manera que uno se cruce sobre el otro. En la *Figura 103*, *Figura 104*, *Figura 105* y *Figura 106* se pueden ver las posiciones inicial y final de los modelos en los cuatro escenarios.

Escenario	Modelo 1	Modelo 2
1	Amazona Herida 28 M.P.	Amazona Herida 28 M.P.
2	Lucy 28 M.P.	Lucy 28 M.P.
3	Amazona Herida 28 M.P.	Armadillo 150K.P
4	Moldura 26M.P.	Armadillo 150K.P

Tabla 29. Escenarios creados para el estudio de tiempos de colisión entre modelos.

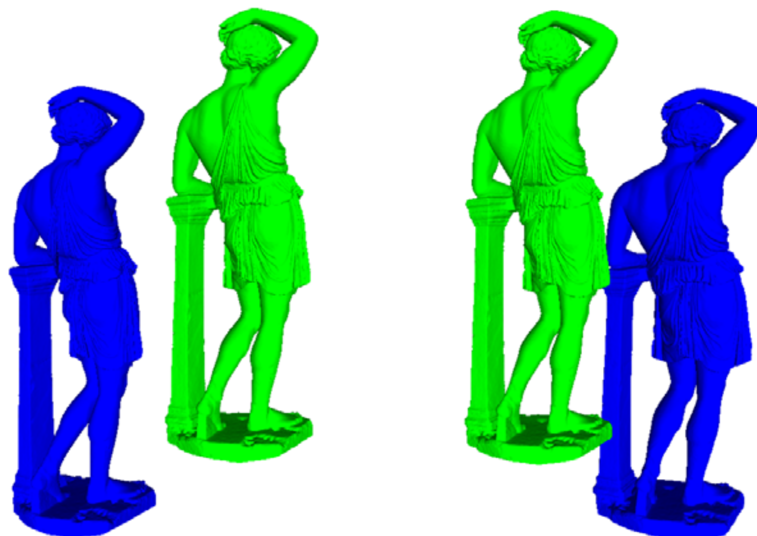


Figura 103. Posiciones inicial y final de las Amazonas Heridas en el escenario 1.

Figura 104. Posiciones inicial y final de las Lucys en el escenario 2.

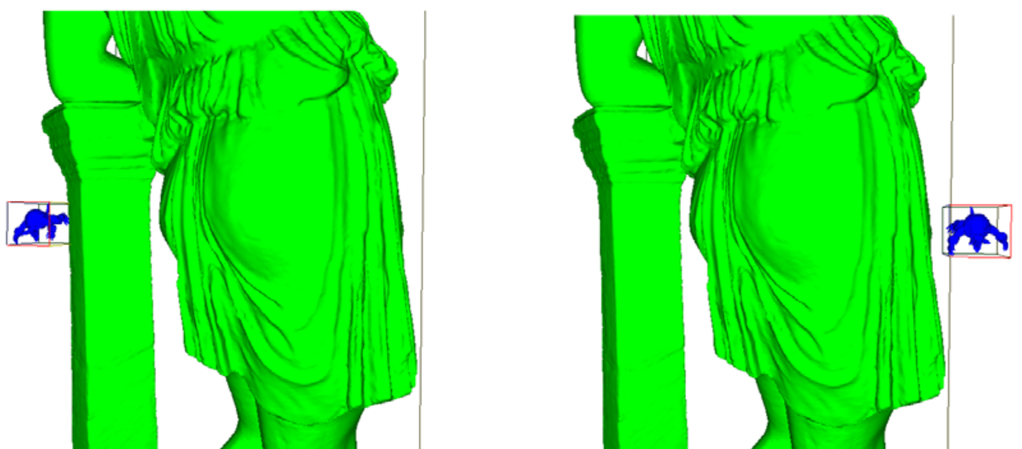


Figura 105. Posiciones inicial y final de Amazona Herida y Armadillo en el escenario 3.

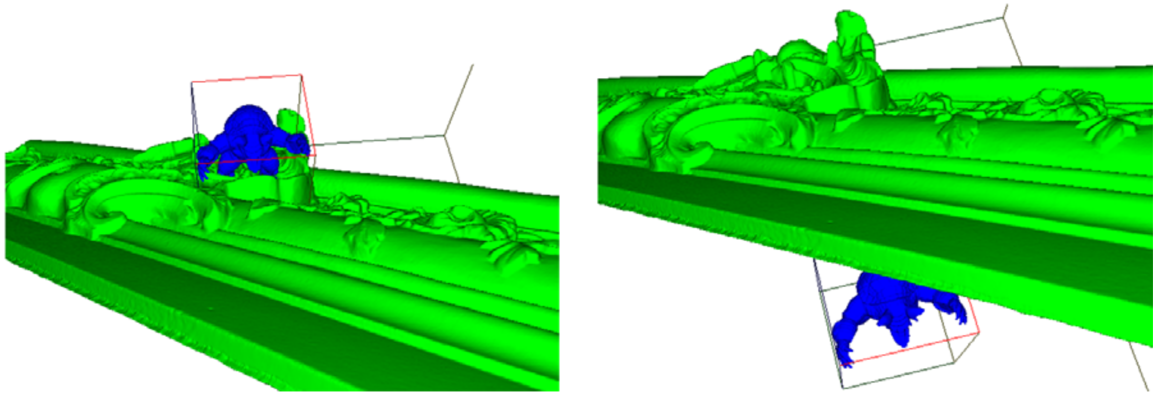


Figura 106. Posiciones inicial y final de Moldura y Armadillo en el escenario 4.

5.6.1. Cálculo de la escena

La escena es una caja alineada a los ejes de coodenadas en la que se introducen los modelos y marca los límites por los que se pueden mover los modelos (ver *Figura 107*). El algoritmo que obtiene el tamaño y la posición de la escena calcula, cada vez que se incorpora un nuevo modelo, el centro de masas de todos los modelos. Este valor será el centro de la escena. Para calcular el tamaño de la caja envolvente de la escena se determinan los tamaños máximos para cada uno de los ejes de coordenadas de las cajas envolventes de los modelos y la distancia entre el centro de masas de cada objeto con el centro de masas de la escena. Las distancias máximas en cada eje calculadas anteriormente se multiplican por cuatro, obteniéndose así la caja envolvente. Como las cajas envolventes de los modelos inicialmente están alineadas a los ejes de coordenadas la de la escena también lo está.

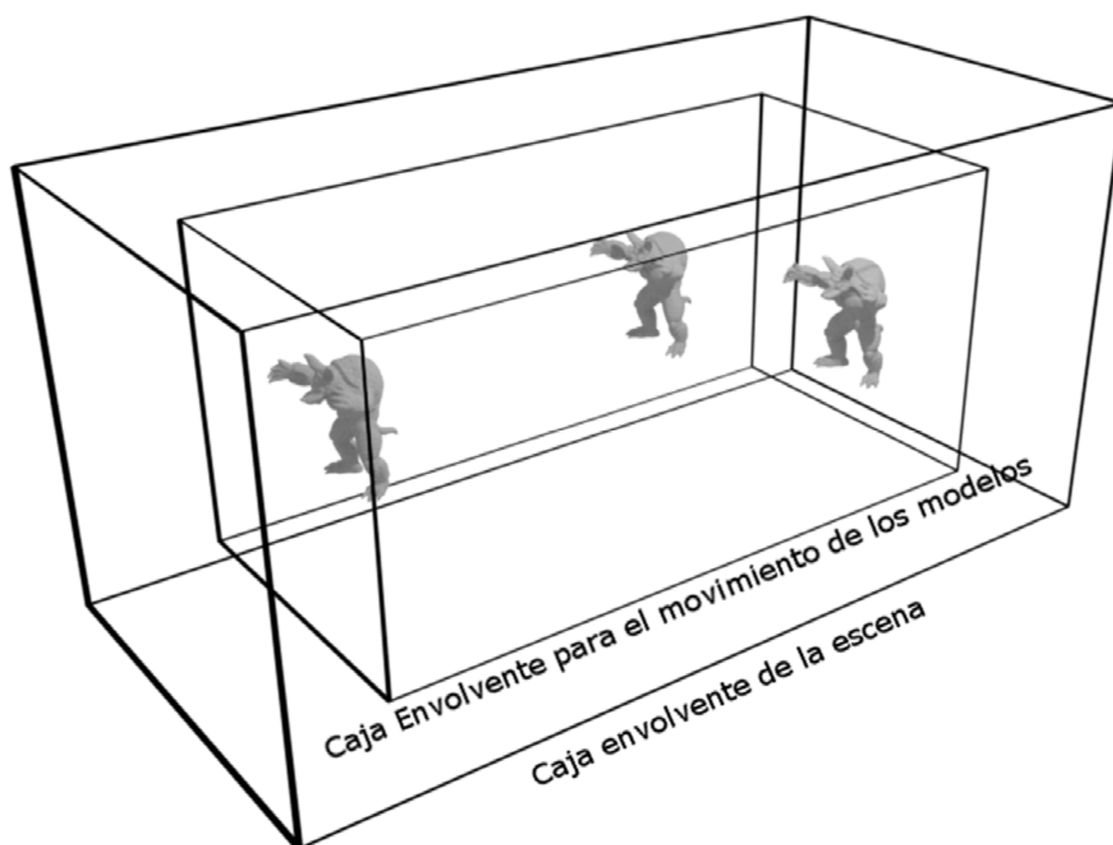


Figura 107. Cajas envolventes del escenario y de movimiento para los modelos.

5.6.2. Movimiento de los modelos por la escena

Los modelos que forman una escena se clasifican en dos tipos: los que están fijos en la misma y no se mueven a lo largo del tiempo y los que están en movimiento. Estos últimos tienen asignada una ruta de pasos y se mueven siguiéndola. Para poder animar una escena se necesitan tener cargadas en memoria los EBP-Octree de todos los modelos hasta el nivel de corte y las rutas de movimiento para cada uno. Cuando se da la orden de animar una escena, los modelos que son móviles se van moviendo paso a paso siguiendo su ruta precalculada. Si la ruta de un objeto tiene menos pasos que la de otro, el objeto con menos pasos se queda en la posición fijada por el último paso, pasando a ser un objeto de tipo inmóvil y se calculan las colisiones con los demás modelos, como si fuera un modelo fijo. Para cada paso dado, dentro de los movimientos de los objetos que forman una escena, se comprueban posibles

colisiones de todos los modelos móviles con todos, no distinguiéndose si están en movimiento o están parados.

5.6.3. Resultados de los test con solamente detección de colisiones.

Los algoritmos se modificaron para que pararan su ejecución cuando detectaran la primera colisión, devolviendo si se había producido colisión o no y el tiempo, en segundos, que había trascurrido para su cálculo.

En el *Gráfico 21*, *Gráfico 22*, *Gráfico 23* y *Gráfico 24* se pueden observar los tiempos, en segundos, de cálculo obtenidos para las rutas programadas sobre los cuatro escenarios (ver *Tabla 29*) comentados anteriormente. En estas gráficas se muestran los tiempos obtenidos con el algoritmo inicial y los obtenidos tras aplicarle a este la optimización vista en la sección 5.5.

Gráfico 21. Tiempo en segundos de la detección de colisión entre dos Amazonas Heridas de 28 M.P.

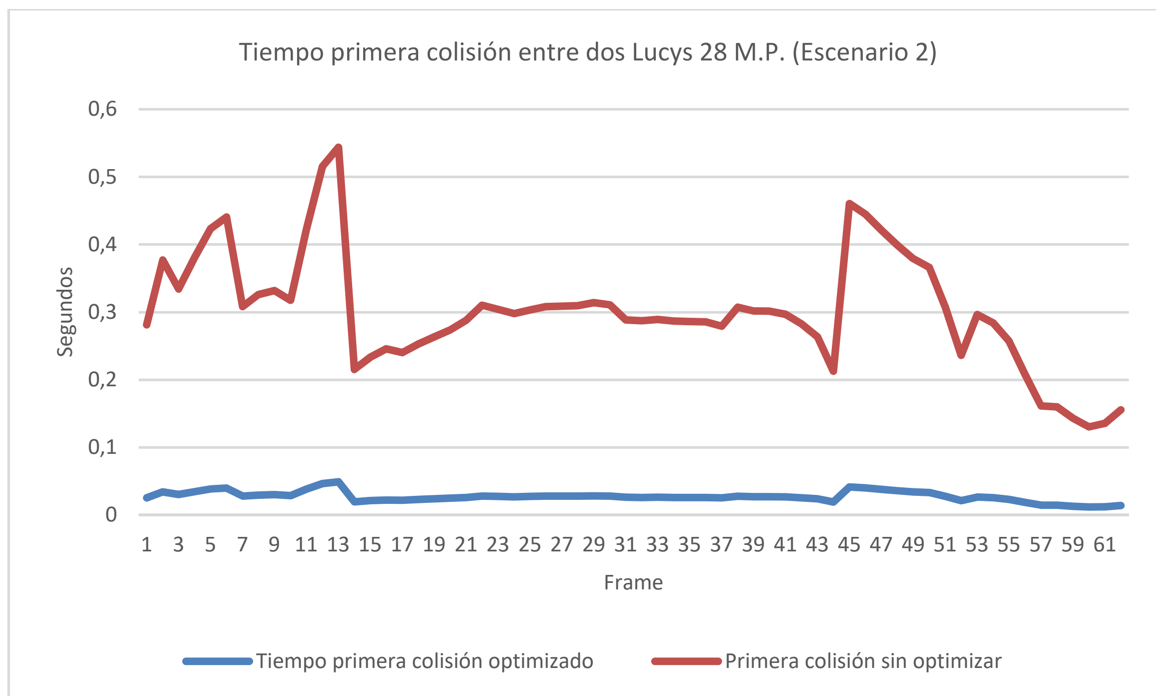


Gráfico 22. Tiempo en segundos de la detección de colisión entre dos Lucys de 28 M.P.

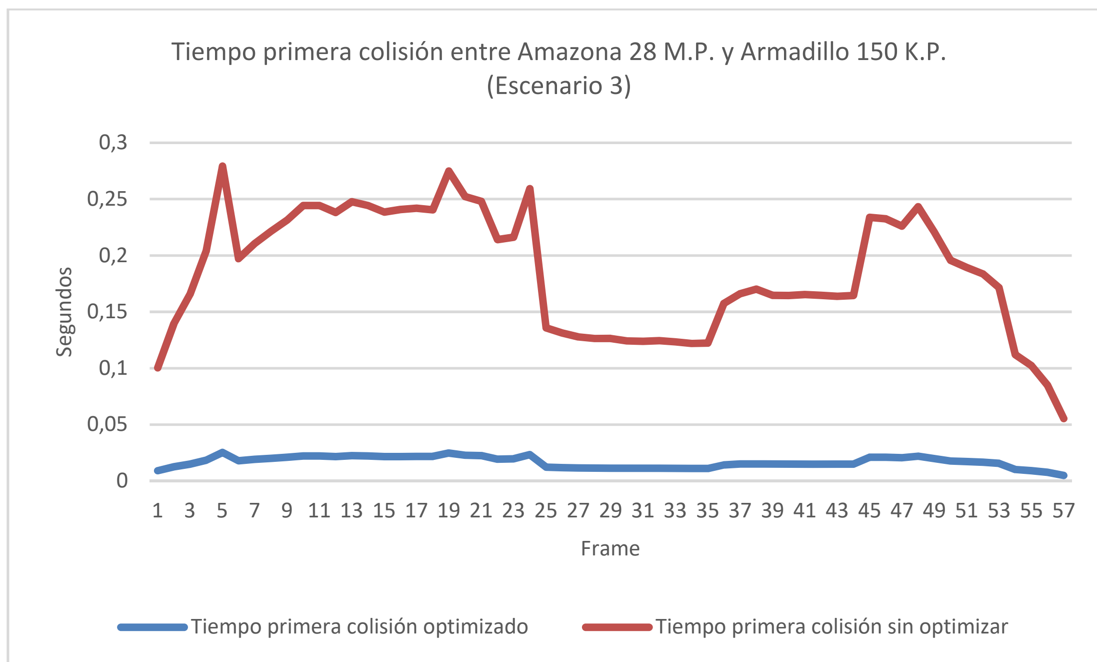


Gráfico 23. Tiempo en segundos de la detección de colisión entre Amazona Herida de 28 M.P. y Armadillo de 150 K.P.

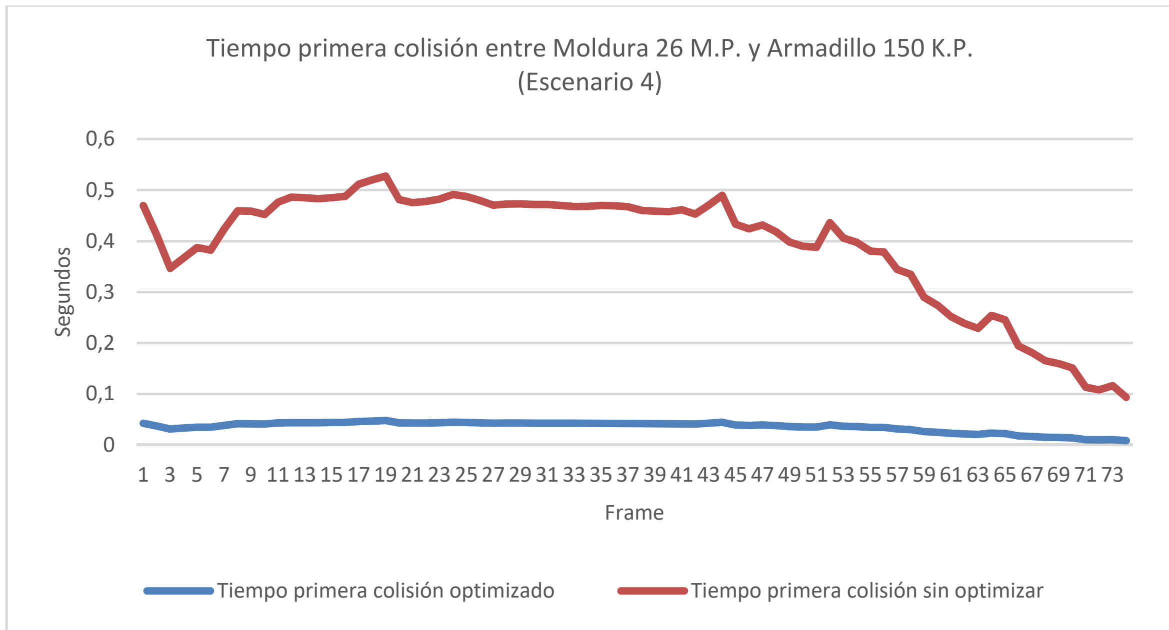


Gráfico 24. Tiempo en segundos de la detección de colisión entre Moldura de 26 M.P. y Armadillo de 150 K.P.

De los gráficos anteriores se puede deducir que la mejora en tiempo de los algoritmos optimizados frente a los que no lo están es superior al 10%. Esta mejora se produce tanto en los algoritmos que detectan la colisión entre dos modelos con igual número de niveles en sus EBP-Octree como en los que no.

5.6.4. Resultados de los test con detección e identificación de todas las colisiones

Cada test realizado a lo largo del movimiento de los modelos devuelve el tiempo que tarda en calcularse y el número de polígonos que colisionan.

En la *Tabla 30* se pueden ver los resultados obtenidos con los dos algoritmos sin optimizar.

Escenario	Modelo 1	Modelo 2	Número medio colisiones	Tiempo por colisión EBP-Octree
1	Amazona 28 M.P.	Amazona 28 M.P.	15790,694	0,000000146
2	Lucy 28 M.P.	Lucy 28 M.P.	38264,048	0,000000090
3	Amazona 28 M.P.	Armadillo 150K.P	971,052	0,000004271
4	Moldura 26M.P.	Armadillo 150K.P	2766,153	0,000000984

Tabla 30. Resultados obtenidos con los algoritmos de detección de colisiones sin optimizar.

En la *Tabla 31* se muestran los resultados obtenidos con los dos algoritmos optimizados.

Escenario	Modelo 1	Modelo 2	Número medio colisiones	Tiempo por colisión EBP-Octree optimizado
1	Amazona 28 M.P.	Amazona 28 M.P.	15790,694	0,000000016
2	Lucy 28 M.P.	Lucy 28 M.P.	38264,048	0,000000009
3	Amazona 28 M.P.	Armadillo 150K.P	971,052	0,000000434
4	Moldura 26M.P.	Armadillo 150K.P	2766,153	0,000000104

Tabla 31. Resultados obtenidos con los algoritmos de detección de colisiones optimizados.

En el *Gráfico 25*, *Gráfico 26*, *Gráfico 27* y *Gráfico 28* se muestran los resultados obtenidos tras ejecutar los algoritmos optimizados en los cuatro escenarios. En ellas de estos se puede observar que los tiempos de cálculo de cada paso aumentan conforme se incrementa el número de colisiones, como es lógico.

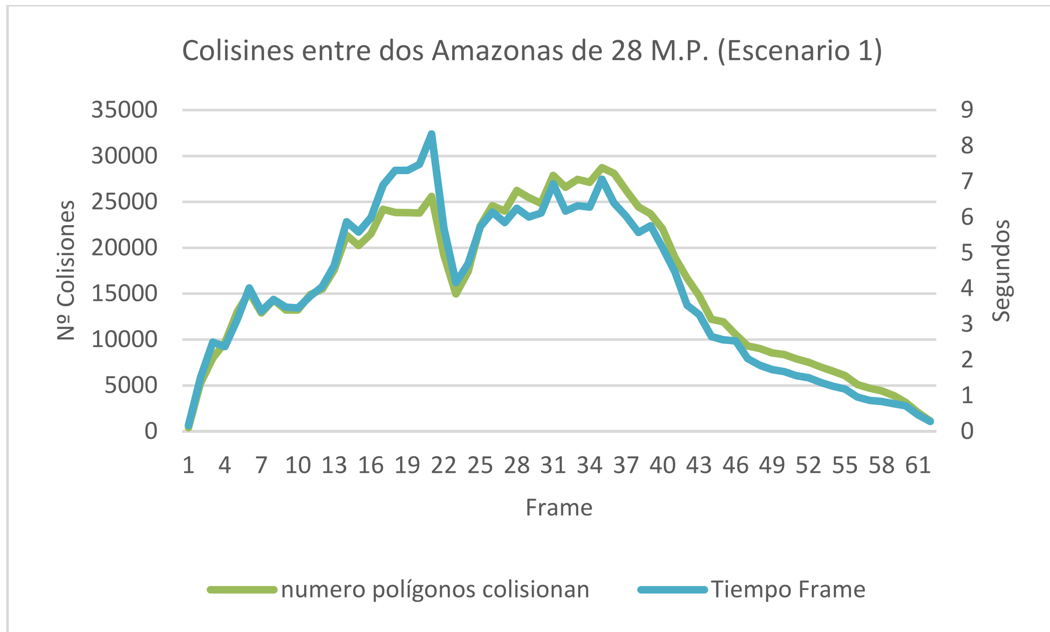


Gráfico 25. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 1 con el algoritmo optimizado.

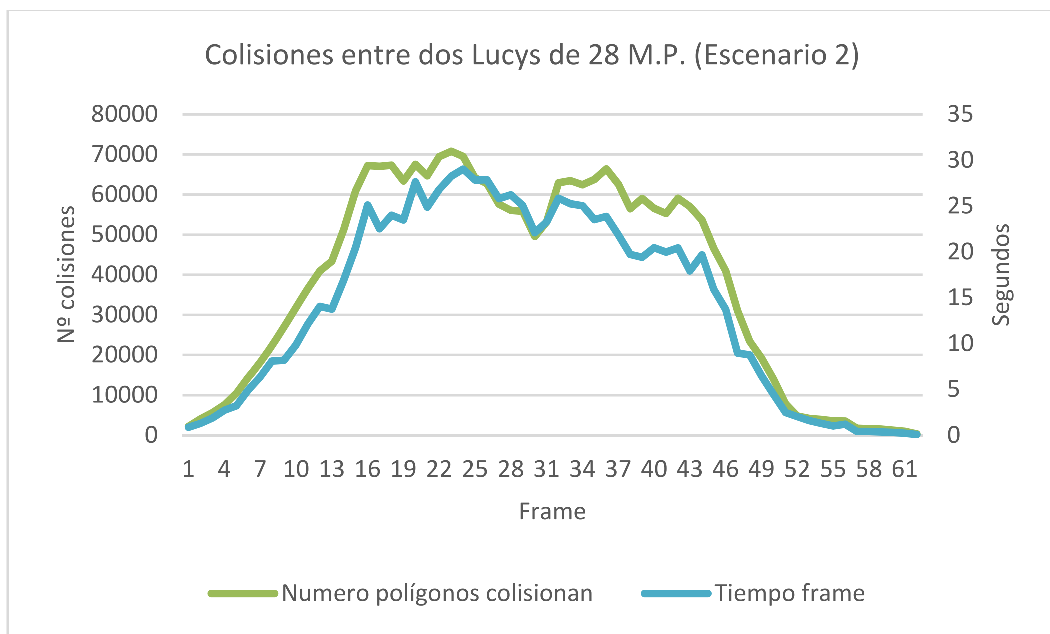


Gráfico 26. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 2 con el algoritmo optimizado.

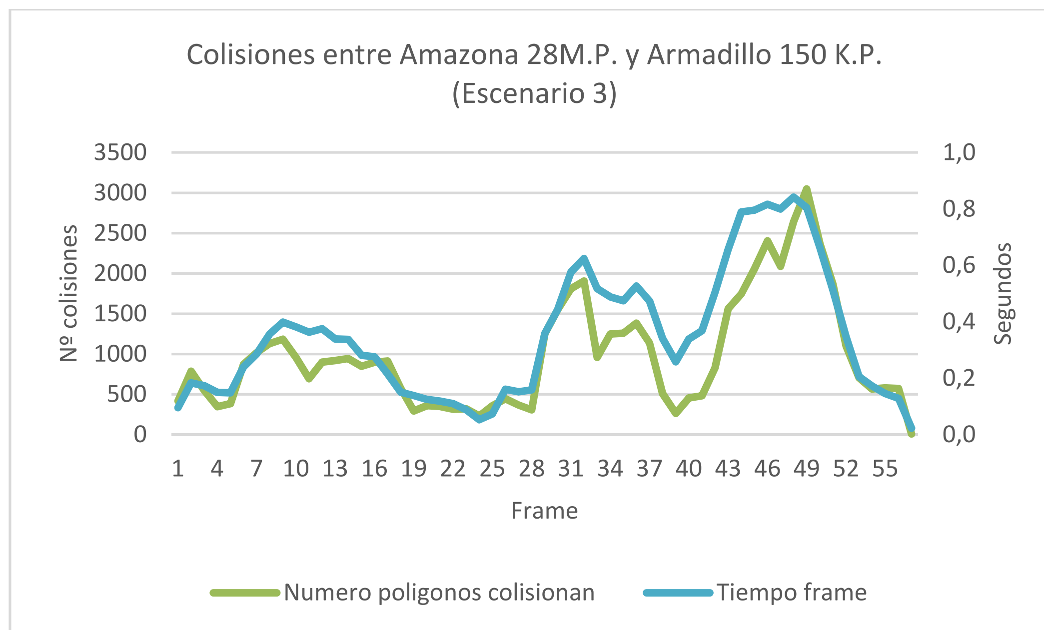


Gráfico 27. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 3 con el algoritmo optimizado.

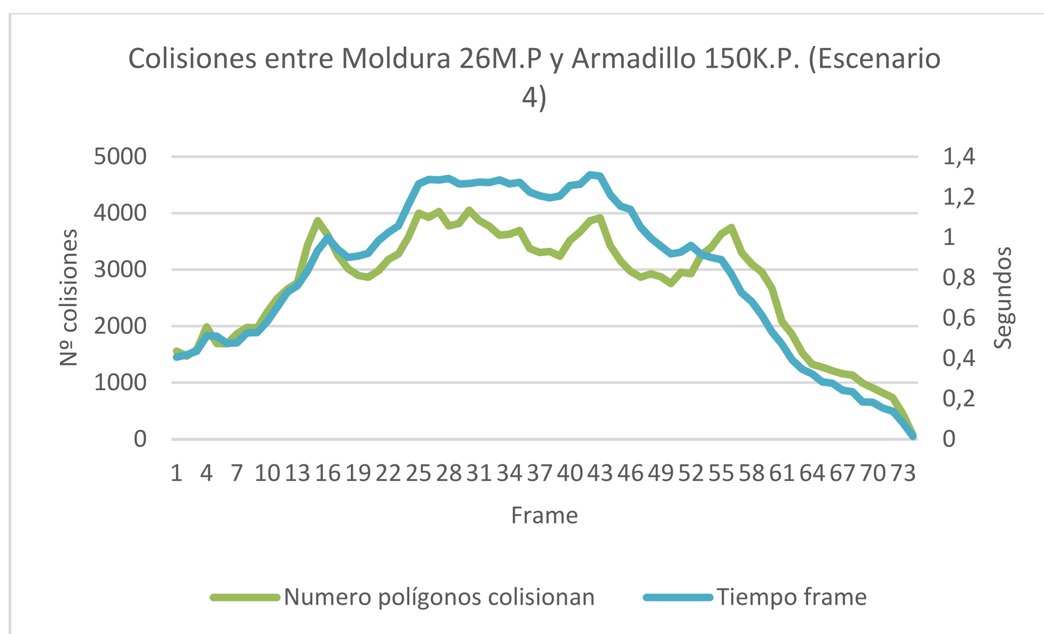


Gráfico 28. Número de colisiones y tiempo en segundos por paso tras realizar el escenario 4 con el algoritmo optimizado.

De los resultados de las tablas anteriores se aprecia que la ganancia en tiempos de los algoritmos optimizados ronda el 10% tanto para el tiempo medio por paso como para el tiempo medio por colisión detectada.

El hecho más destacable de los test de colisión entre modelos es la mejora en los tiempos que se obtienen por los algoritmos optimizados, tanto para el caso de comparar dos modelos con igual y con diferente número de niveles. Como media del cálculo de cada colisión los tiempos obtenidos son menores de 450 nanosegundos. Estos resultados permiten que los EBP-Octree se puedan utilizar en innumerables aplicaciones y dentro de una gran gama de dispositivos hápticos.

En la *Figura 108*, *Figura 109* y *Figura 110* se pueden ver los resultados visuales obtenidos tras ejecutar el test de inclusión entre dos modelos. Aquí se pueden apreciar, marcados de color rojo, los polígonos de los dos modelos resultantes del cálculo de la colisión.

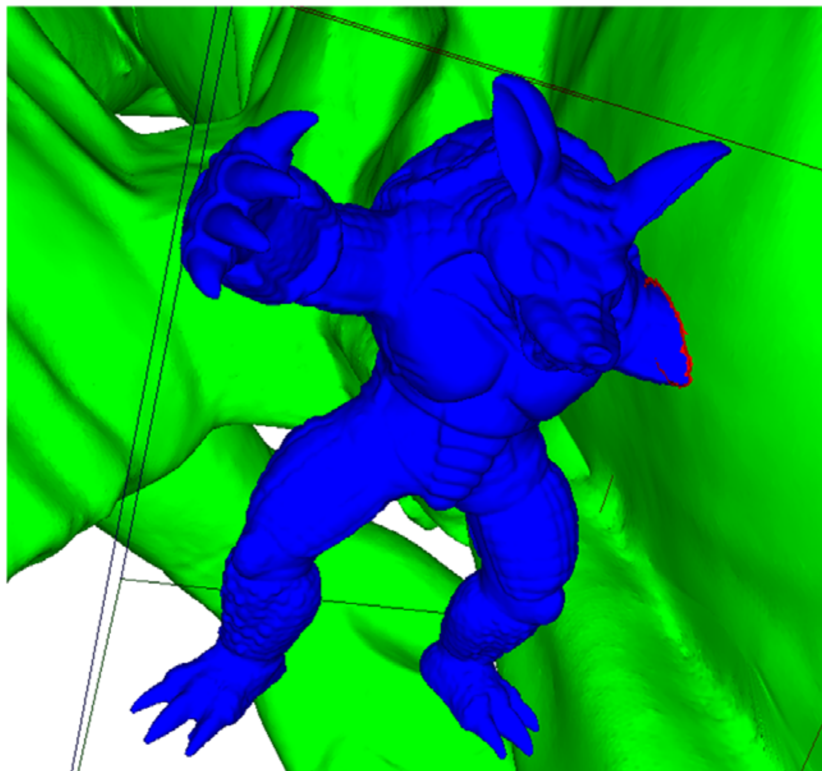


Figura 108. Armadillo con la mano metida dentro de Amazona Herida.

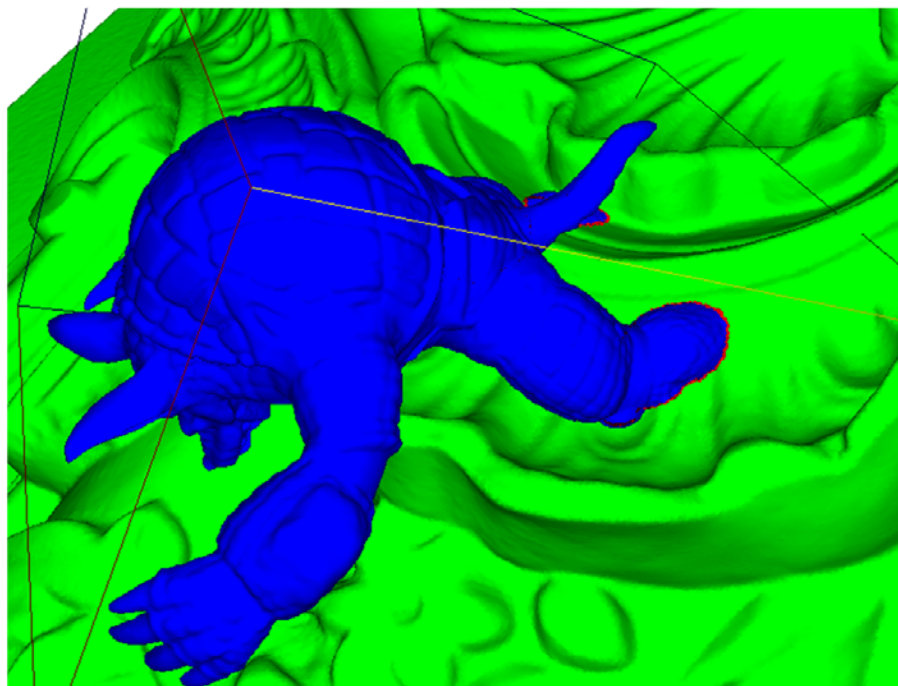


Figura 109. Armadillo con los pies dentro de la Moldura.

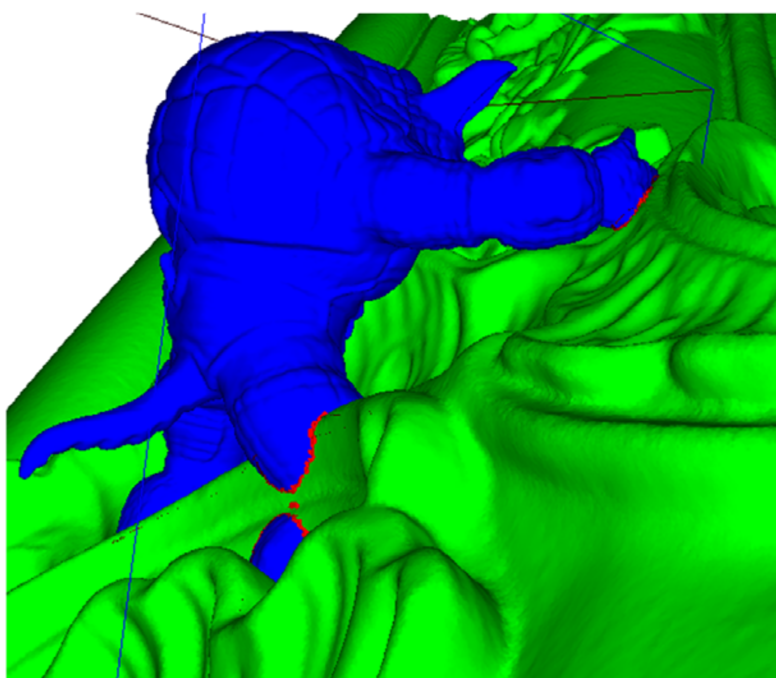


Figura 110. Armadillo con manos y piernas metidas dentro de Lucy.

5.7. Comparación de los resultados obtenidos con el EBP-Octree con otras aplicaciones o librerías de detección de colisiones.

Las aplicaciones o librerías utilizadas para hacer el estudio comparativo con el EBP-Octree se han seleccionado de los que se mostraron en la sección 2.5 y cuyo código está disponible y libre. Las tres aplicaciones o librerías utilizadas son: PQP, Swiff++ y Gproximity. Sus características principales son:

- PQP: no utiliza la GPU, devuelve todos los puntos y el lugar en el que se producen colisiones y no acepta modelos grandes.
- Swift++: no utiliza la GPU, presenta dos posibles formas para la detección de posibles colisiones. La primera devuelve verdadero o falso para la escena completa y la segunda devuelve una lista de todos los puntos de los objetos que colisionan. No permite trabajar con modelos grandes.
- Gproximity: utiliza la GPU, devuelve todas las colisiones que se producen y por motivos de GPU no está diseñado para trabajar con modelos grandes.

De estas características podemos deducir lo siguiente. En primer lugar el uso de la GPU es bueno porque permite paralelizar el proceso de cálculo pero tiene el inconveniente de que limita el tamaño de los modelos. Y en segundo lugar estas aplicaciones o librerías no permiten la carga de modelos como los vistos en el apartado anterior, por esto se ha hecho un estudio con un modelo generándolo a diferentes tamaños, empezando por medio millón de triángulos y aumentando en intervalos de medio millón de triángulos, comprobando que **el tamaño máximo soportado por estas aplicaciones es de dos millones y medio de triángulos**.

Una vez detectado el tamaño máximo de los modelos que soportan las librerías se tomaron dos modelos, la Amazona Herida reduciendo su tamaño a dos millones y medio de polígonos y el Armadillo de trescientos cuarenta y seis polígonos. Con estos dos modelos se crearon tres escenarios en los cuales dichos modelos se cruzaban pasando uno sobre el otro. El primer escenario cargaba dos armadillos, uno de ellos desplazado en el eje X la distancia de la caja envolvente en dicho eje, de manera que los dos modelos no se tocaran, a continuación uno de los modelos comienza a desplazarse por el eje X de manera que los dos modelos comienzan a tocarse, la animación termina cuando el modelo en movimiento atraviesa al otro modelo y lo deja de tocar. El escenario dos es igual que el primero, pero se toman

como modelos el Armadillo y la Amazona Herida. Y por último el tercer escenario toma como modelos dos Amazonas Heridas. En la *Tabla 32* se pueden ver los modelos que participan en cada escenario.

Escenario	Modelo 1	Modelo 2
1	Amazona 2,5 M.P.	Amazona 2,5 M.P.
2	Armadillo 350 mil P.	Armadillo 350 mil P.
3	Armadillo 350 mil P.	Amazona 2,5 M.P.

Tabla 32. Modelos incluidos en los distintos escenarios.

En la *Tabla 33* se muestran los tiempos medios obtenidos para detectar la colisión entre dos modelos para cada uno de los escenarios planteados en el apartado anterior. De estos datos cabe destacar que la librería Swift++ no soporta modelos muy grandes (máximo de 400 mil polígonos) por esto no se presentan datos en los escenarios en los cuales se cargue el modelo la Amazona Herida. El Gproximity en algunos casos se comporta mejor que el EBP-Octree pero hay que destacar que este método utiliza la GPU y su paralelismo para realizar los cálculos mientras que el EBP-Octree trabaja en CPU y sin utilizar técnicas de paralelismo.

Escenario	EBP-Octree	EBP-Octree optimizado	PQP	Swift++	Gproximity
Escenario 1	0,001627546	0,000029237	26,430600	no	0,000158289
Escenario 2	0,000615104	0,000035750	0,006206	0,018	0,000022534
Escenario 3	0,000072392	0,000020697	0,001530	no	0,000030559

Tabla 33. Tiempo medio en segundos para realizar el cálculo de la colisión entre dos modelos.

6. CONCLUSIONES Y TRABAJOS FUTUROS.

En este capítulo se exponen las conclusiones sobre el trabajo realizado, así como la novedad que aporta la estructura de datos diseñada: el EBP-Octree. También se presentan las líneas de trabajo que quedan abierta para su desarrollo como trabajos futuros, así como las posibles aplicaciones a desarrollar.

6.1. Conclusiones y principales aportaciones.

En este trabajo se ha presentado un nuevo esquema de representación de sólidos, EBP-Octree, que permite trabajar con modelos formados por varias decenas de millones de polígonos. Este esquema incorpora una jerarquía de volúmenes envolventes con una estructura de índices espaciales. El trabajo desarrollado en los

últimos años se encuentra resumido en los resultados obtenidos en (Aguilera, Feito, and Melero 2013; Aguilera, Melero, and Feito 2016).

Se han descrito la estructura de datos interna utilizada para el almacenamiento del modelo, los distintos nodos que la forman, así como el proceso detallado para la construcción de la misma a partir de un sólido que se encuentra representado por su frontera.

El esquema de representación de sólidos se ejecuta en ordenadores cuya longitud de palabra es de 64 bit, lo cual permite trabajar con octree de hasta veinte niveles de profundidad.

La estructura planteada se crea *out-of-core*, de manera que para cada sólido solo se calcula una vez y se almacena en archivos.

Con el objetivo de poder trabajar con modelos muy grandes la estructura de representación del sólido se ha dividido en dos partes, manteniendo una de ellas en memoria principal y la otra en memoria masiva, de manera que solo se carga en memoria principal la parte de la estructura que se necesita en cada instante.

Se han diseñado algoritmos que son capaces de adaptar, de manera dinámica, la estructura de representación del sólido a las características físicas del ordenador en el que se van a ejecutar. Para ello, a la hora de cargar un modelo en memoria, se hace un estudio de cuánta memoria libre dispone el ordenador en ese instante. Dependiendo del resultado obtenido solamente se almacena en memoria hasta el nivel de la estructura que haga que no se sobrepase el espacio libre.

La estructura presentada permite la visualización y transmisión progresiva del modelo, pudiendo mostrar o trasladar las envolventes hasta un nivel determinado.

Se han estudiado diferentes métodos para la selección de planos relevantes para el cálculo de las envolventes, demostrando que el método que utiliza una selección por clústeres es el que mejores resultados obtiene.

Además del esquema de representación también se ha presentado un algoritmo que utiliza los EBP-Octree para calcular la inclusión de un punto en un sólido, utilizando la jerarquía de envolventes y los índices espaciales que proporcionan la estructura para optimizar los cálculos, obteniendo tiempos de respuesta muy bajos.

Así mismo se ha utilizado dicho esquema de representación para diseñar un algoritmo que permite calcular la distancia mínima y el vector de colisión entre un

punto y un modelo.

Se han presentado también varios algoritmos que permiten el cálculo de la colisión entre dos modelos, así como la distancia mínima entre ellos y el vector de colisión de los mismos.

La estructura presentada ha permitido desarrollar una serie de algoritmos que se adaptan a las exigencias de diferentes aplicaciones, de manera que pueden dar una respuesta aproximada la detección de colisiones en un tiempo máximo, utilizando para ello las envolventes de los diferentes niveles.

Una de las utilidades de la estructura de datos ha sido su empleo en una pequeña aplicación que permite pintar sobre la superficie de los modelos utilizando un dispositivo háptico de seis grados de libertad. Esta aplicación ha podido trabajar con modelos formados por varias decenas de millones de polígonos en tiempo real.

Por último se ha realizado un estudio comparativo con otras aplicaciones o librerías de detección de colisiones con código abierto, demostrándose que estas no permiten trabajar con modelos muy grandes. En cambio, la estructura presentada, obtiene unos tiempos de respuesta sobre modelos pequeños mejores que las anteriores.

6.2. Trabajos futuros.

Como continuación del trabajo ya desarrollado y descrito en esta memoria, quedan abiertas y por ampliar una serie de posibles líneas de trabajo futuro. A continuación se enumeran las principales y futuras tareas de trabajo:

- Estudio de nuevos métodos de selección de planos relevantes para formar las envolventes.
- Estudio y prueba de diferentes algoritmos del tipo QT clústering que permitan la selección de los planos de manera automática.
- Mejorar el esquema de representación para conseguir mejorar los algoritmos desarrollados en este trabajo.
- Realizar un análisis, diseño, estudio de viabilidad e implementación de los EBP-Octrees con varios niveles de corte, de manera que se puedan representar modelos mucho más grandes que los actuales.

- Estudio de nuevos métodos para calcular el nivel de corte óptimo de los EPB-Octree.
- La paralelización de los algoritmos utilizados para el cálculo del EBP-Octree, para la detección de inclusión punto en sólido, para el cálculo de la distancia y del vector de colisión y para los algoritmos del cálculo de colisión entre dos modelos.
- Estudiar la viabilidad de utilizar el esquema de representación para implementar operaciones booleanas entre modelos.
- Adaptar y reimplementar todos los algoritmos desarrollados en este trabajo para que puedan utilizar toda la potencia de cálculo que proporcionan las nuevas tarjetas gráficas, las cuales incorporan GPU.
- Estudiar la viabilidad de utilizar el esquema de representación como herramienta para el esculpido de sólidos.
- Estudiar y adaptar el esquema de representación para que pueda recoger la representación de modelos deformables.
- Utilizar el esquema de representación en otros dispositivos hápticos.
- Buscar e implementar nuevas aplicaciones donde se pueda utilizar el esquema de representación propuesto en esta memoria.

BIBLIOGRAFÍA

"Add a K-DOP Collision Hull to a Static Mesh." n.d. https://docs.unrealengine.com/latest/INT/Engine/Physics/Collision/HowTo/AddDOP/.

Aguilera, Angel, Francisco R. Feito, and Francisco Javier Melero. 2013. "EBP-Octree : An Optimized Bounding Volume Hierarchy for Massive Polygonal Models." In *XXIII Congreso Español de Informática Gráfica (CEIG 2013)*, edited by Mª Carmen Juan and Diego Borro, 11–21. Madrid.

Aguilera, Angel, Francisco R. Feito, and Juan C. Torres. 2010. "Generalización Geográfica En Terrenos Representados Con Multi-Resolución Variable No Restrictiva." *Congreso Español de Informática Gráfica. (CEIG XX Valencia).*, 285–88.

Aguilera, Angel, Francisco Javier Melero, and Francisco R. Feito. 2016. "Out-of-Core Real-Time Haptic Interaction on Very Large Models." *CAD Computer Aided Design* 77. Elsevier Ltd: 98–106. doi:10.1016/j.cad.2016.04.002.

Aguilera, Angel, Juan C. Torres, and Francisco R. Feito. 2001. "Utilización de Índice Espacial Para Modelado Multiresolución de Terrenos." *Congreso Español de Informática Gráfica (CEIG XI GIRONA, ESPAÑA).*

———. 2003. "Multi-Resolution Modelling of Terrains by Using Non Restricted Quadtree Triangulation." *Eurographics 2003*. doi:10.2312/egp.20031007.

Akenine-Möller, Tomas. 2001. "Fast 3D Triangle-Box Overlap Testing." *Journal of Graphics Tools* 6 (1): 29–33. doi:10.1080/10867651.2001.10487535.

Akenine-Möller, Tomas, Eric Haines, and Naty Hoffman. 2008. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd.

Aliyu, Dahiru, and Khaled S. Al-Sultan. 1999. "Fast Collision Detection in Four-Dimensional Space." *European Journal of Operational Research* 114 (2): 437–45. doi:10.1016/S0377-2217(98)00030-7.

Alliez, Pierre, Stephane Tayeb, and Wormser Camille. n.d. "3D Fast Intersection and Distance Computation (AABB Tree)." http://doc.cgal.org/latest/AABB_tree/.

"Árbol Kd." n.d. https://es.wikipedia.org/wiki/Árbol_kd.

Argudo, Oscar, I. Besora, Pere Brunet, C. Creus, P. Hermosilla, Isabel Navazo, and Àlvaro Vinacua. 2016. "Interactive Inspection of Complex Multi-Object Industrial Assemblies." *CAD Computer Aided Design* 79. Elsevier Ltd: 48–59. doi:10.1016/j.cad.2016.06.005.

Ayala, Dolors, Pere Brunet, R. Juan, and Isabel Navazo. 1985. "Object Representation by Means of Nonminimal Division Quadtrees and Octrees." *ACM Transactions on Graphics* 4 (1): 41–59. doi:10.1145/3973.3975.

Baraff, David. 1992. "Dynamic Simulation of Non-Penetrating Rigid Body Simulation." Cornell University.

Barequet, Gill, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. 1996. "BOXTREE: A Hierarchical Representation for Surfaces in 3D." *Computer Graphics Forum* 15 (3): 387–96. doi:10.1111/1467-8659.1530387.

Barriuso, Jesús Ramón. 1998. "Cálculo de La Distancia Entre Objetos Representados Con Precisión Mediante Poliedros Cóncavos Durante Una Simulación Mecánica." Universidad de Navarra.

Bentley, Jon Louis. 1975. "Multidimensional Binary Search Trees Used for Associative Searching." *Communications of the ACM* 18 (9): 509–17. doi:10.1145/361002.361007.

Bergen, Gino Van Den. 1997. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees." *Journal of Graphics Tools* 2: 1–13. doi:10.1080/10867651.1997.10487480.

Bergen, Gino Van Den. 1999a. "A Fast and Robust GJK Implementation for Collision Detection of Convex Objects." *Journal of Graphics Tools* 4 (2): 7–25. doi:10.1080/10867651.1999.10487502.

———. 1999b. "Collision Detection in Interactive 3D Computer Animation." Universiteit Eindhoven.

———. 2004. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann publishers.

Blow, Jonathan. 2011. "Practical Collision Detection." *Navigation* 1 (42). http://www.gamedev.net/page/resources/_/reference/programming/game-programming/collision-detection/practical-collision-detection-r736.

Breen, David E., Sean Mauch, Ross T. Whitaker, and Jia Mao. 2001. "3D Metamorphosis between Different Types of Geometric Models." *Computer Graphics Forum* 20 (3): C/36-C/48. doi:10.1111/1467-8659.00496.

Brown, Amalia Duch. 2004. "Design and Analysis of Multidimensional Data Structures."

Brunet, Pere, and Isabel Navazo. 1992. *Geometric Modelling of Volumes*. Eurographics Tutorials'92, Eurographics Association, Cambridge ISSN 1017-4656.

Brunet, Pere, F.J. Santisteve, A. Vilanova, L. Chiarabini, G.A. Patow, E. Staffetti, and J. Surinyac. 1999. "Estructuras Geométricas Jerárquicas Para La Modelización de Escenas 3D."

Cameron, Stephen. 1997. "Enhancing GJK: Computing Minimum and Penetration Distances between Convex Polyhedra." In *Proceedings of International Conference on Robotics and Automation*, 4:3112–17. IEEE. doi:10.1109/ROBOT.1997.606761.

Cameron, Stephen, and Caigong Quin. 1998. "Motion Planning and Collision Avoidance with Complex Geometry." *IECON 98, 24th Annual Conference of the IEEE Industrial Electronics Society* 2: 2222–26.

Canny, John. 1986. "Collision Detection for Moving Polyhedra." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. doi:10.1109/TPAMI.1986.4767773.

Cano, Pedro, Juan Carlos Torres, and Francisco Velasco. 2003. "Progressive Transmission of Polyhedral Solids Using a Hierarchical Representation Scheme." *Journal of WSCG* 11 (1): 81–86. http://wscg.zcu.cz/WSCG2003/Papers_2003/I83.pdf.

Capens, Nicolas. 2001. "Smallest Enclosing Spheres." In *Flipcode Code of the Day Forum*. http://www.flipcode.com/archives/Smallest_Enclosing_Spheres.shtml.

Carlbom, Ingrid, Indranil Chakravarty, and David Vanderschel. 1985. "A Hierarchical Data Structure for Representing the Spatial Decomposition of 3D Objects." In *Frontiers in Computer Graphics*, 2–12. Tokyo: Springer Japan. doi:10.1007/978-4-431-68025-3_1.

Chung, Kelvin. 1996. "An Effcient Collision Detection Algorithm for Polytopes in Virtual Environments." The University of Hong Kong.

Chung, Kelvin, and Wenping Wang. 1996. "Quick Collision Detection of Polytopes in Virtual Environments." *ACM Symposium on Virtual Reality Software and Technology*, 1–4. doi:10.1.1.111.8336.

Cohen, Abbe, and Elaine Chen. 1999. "Six Degree-of-Freedom Haptic System as a Desktop Virtual Prototyping Interface." *Asme Dyn Syst Control Div Publ Dsc.* 67 (June 1999): 401–2.

Cohen, Jonathan D., Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. 1994. "Interactive and Exact Collision Detection for Multi-Body Environments." *University of North …*, no. 8920219. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.4429&rep=rep1&type=pdf.

Cohen, Jonathan D, Ming C Lin, Dinesh Manocha, and Madhav Ponamgi. 1995. "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments." *I3D '95 Proceedings of the 1995 Symposium on Interactive 3D Graphics*, 189–97. doi:10.1145/199404.199437.

"Collision Detection (Advanced Methods in Computer Graphics)." n.d. http://what-when-how.com/advanced-methods-in-computer-graphics/collision-detection-advanced-methods-in-computer-graphics-part-1/.

Culley, R., and Kilian Kempf. 1986. "A Collision Detection Algorithm Based on Velocity and Distance Bounds." In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, 3:1064–69. Institute of Electrical and Electronics Engineers. doi:10.1109/ROBOT.1986.1087575.

Dingliana, John, and Carol O'Sullivan. 2000. "Graceful Degradation of Collision Handling in Physically Based Animation." *Proc. of Eurographics* 2000 (February): 239–247. doi:10.1111/1467-8659.00416.

Dobkin, David P., and David G. Kirkpatrick. 1990. "Determining the Separation of Preprocessed Polyhedra — A Unified Approach." In *Automata, Languages and Programming,* 400–413. Berlin/Heidelberg: Springer-Verlag. doi:10.1007/BFb0032047.

Duff, Tom. 1992. "Interval Arithmetic Recursive Subdivision for Implicit Functions and Constructive Solid Geometry." *ACM SIGGRAPH Computer Graphics* 26 (2): 131–38. doi:10.1145/133994.134027.

Eberly, David H. 2012. *Game Physics (Second Edition)*. Transferred to Taylor & Francis as of 2012.

Ehmann, Stephen a., and Ming C. Lin. 2001. "Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition." *Computer Graphics Forum* 20 (3): 500–511. doi:10.1111/1467-8659.00543.

Ehmann, Stephen A, and Ming C Lin. 2000. "SWIFT: Accelerated Proximity Queries Using Multi-Level Voronoi Marching." *Chapel Hill, NC*, 1–20. doi:10.1109/IROS.2000.895281.

Ericson, Christer. 2005. "Real-Time Collision Detection." *Books.google.com*, 632. http://books.google.com/books?hl=en&lr=&id=0MvuykjoW_IC&oi=fnd&pg=PP2&dq=Real-Time+Collision+Detection&ots=CPuDYG1rHq&sig=xdpNvZ7vetSvhHuOXpUtJc-ZJIM%0Apapers2://publication/uuid/FD770B92-C0CA-4E3F-B7F7-230A55288387.

Feito, Francisco R., and Juan C. Torres. 1997. "Inclusion Test for General Polyhedra" 21 (1): 23–30. doi:15016870.

Fortune, Steven, and Christopher J. Van Wyk. 1993. "Efficient Exact Arithmetic for Computational Geometry." In *Proceedings of the Ninth Annual Symposium on Computational Geometry - SCG '93*, 163–72. New York, New York, USA: ACM Press. doi:10.1145/160985.161015.

Fuchs, Henry, Zvi M. Kedem, and Bruce F. Naylor. 1980. "On Visible Surface Generation by a Priori Tree Structures." In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '80*, 124–33. New York, New York, USA: ACM Press. doi:10.1145/800250.807481.

Garcia-Alonso, Alejandro. 1990. "Simulación Interactiva Y Análisis de Colisiones En Mecanismos Tridimensionales Con Gráficos Realistas." Universidad de Navarra.

Garcia-Alonso, Alejandro, Nicolás Serrano, and Juar Flaquer. 1994. "Solving the Collision Detection Problem." *IEEE Computer Graphics and Applications* 14 (3): 36–43. doi:10.1109/38.279041.

Gargantini, Irene. 1982. "Linear Octtrees for Fast Processing of Three-Dimensional Objects." *Computer Graphics and Image Processing* 20 (4): 365–74. doi:10.1016/0146-664X(82)90058-2.

"Geomagic®." n.d. http://www.geomagic.com/es/products/phantom-omni/overview/.

Gilbert, Elmer, Derek W. Johnson, and S. Sathiya Keerthi. 1988. "Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space." *IEEE Journal of Robotics and Automation* 4 (2): 193–203.

Gomes de Sá, Antonio, and Gabriel Zachmann. 1999. "Virtual Reality as a Tool Verification of Assembly and Maintenance Processes." *Computers & Graphics* 23 (3): 389–403.

Gottschalk, Stefan. 1996. "Separating Axis Theorem." doi:Technical Report TR96-024.

Gottschalk, Stefan, Ming C. Lin, Dinesh Manocha, and Chapel Hill. 1996. "OBB Tree: A Hierarchical Structure for Rapid Interference Detection." *Proceedings of SIGGRAPH 96*, no. 8920219: 171–80. doi:http://doi.acm.org/10.1145/237170.237244.

Govindaraju, Naga K., Ming C. Lin, and Dinesh Manocha. 2005. "Quick-CULLIDE: Fast Inter- and Intra-Object Collision Culling Using Graphics Hardware." *IEEE Proceedings VR 2005 Virtual Reality 2005* 2005: 59–66. doi:10.1109/VR.2005.1492754.

———. 2006. "Fast and Reliable Collision Culling Using Graphics Hardware." *IEEE Transactions on Visualization and Computer Graphics* 12 (2): 143–53. doi:10.1109/TVCG.2006.29.

Govindaraju, Naga K., Stephane Redon, Ming C. Lin, and Dinesh Manocha. 2003. "CULLIDE: Interactive Collision Detection between Complex Models in Large Environments Using Graphics Hardware." *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 25–32. http://portal.acm.org/citation.cfm?id=844178.

Govindaraju, Naga K, David Knott, Nitin Jain, Ilknur Kabul, Rasmus Tamstorf, Russell Gayle, Ming C Lin, and Dinesh Manocha. 2005. "Interactive Collision Detection between Deformable Models Using Chromatic Decomposition." In *ACM SIGGRAPH 2005 Papers on - SIGGRAPH '05*, 991. New York, New York, USA: ACM Press. doi:10.1145/1186822.1073301.

Gregory, Arthur D., Stephen A. Ehmann, and Ming C. Lin. 2000. "inTouch: Interactive Multiresolution Modeling and 3D Painting with a Haptic Interface." *Virtual Reality, 2000. Proceedings. IEEE*, 45–52. doi:10.1109/VR.2000.840362.

Gregory, Arthur D., Ming C. Lin, Stefan Gottschalk, and Russell Taylor. 1999. "A Framework for Fast and Accurate Collision Detection for Haptic Interaction." *Virtual Reality, 1999. Proceedings., IEEE*, 38–45. doi:10.1145/1198555.1198604.

Guttman, Antonin. 1984. "R-Trees: A Dynamic Index Structure for Spatial Searching." *ACM SIGMOD Record* 14 (2): 47. doi:10.1145/971697.602266.

Hahn, James K. 1988. "Realistic Animation of Rigid Bodies." *ACM SIGGRAPH Computer Graphics* 22 (4): 299–308. doi:10.1145/378456.378530.

Han, Jiawei, Micheline Kamber, and Anthony K H Tung. 2001. "Spatial Clustering Methods in Data Mining: A Survey." *Geographic Data Mining and Knowledge Discovery* 2: 188–217.

Hayward, Vincent. 1986. "Fast Collision Detection Scheme by Resursive Decomposition of A Manipulation Workspace." *Image (Rochester, N.Y.)*, 1044–49.

Held, Martin, James T. Klosowski, and Joseph S. B. Mitchell. 1995. "Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs." In *Canadian Conference on Computational Geometry*.

———. 1996. "Real-Time Collision Detection for Motion Simulation within Complex Environments." In *ACM SIGGRAPH 96 Visual Proceedings: The Art and Interdisciplinary Programs of SIGGRAPH '96 on - SIGGRAPH '96*, 151. New York, New York, USA: ACM Press. doi:10.1145/253607.253888.

Herman, Matthew A. 1986. "Fast, Three-Dimensional, Collision-Free Motion Planning." In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, 3:1056–63. Institute of Electrical and Electronics Engineers. doi:10.1109/ROBOT.1986.1087622.

Heyer, Laurie J, Semyon Kruglyak, and Shibu Yooseph. 1999. "Exploring Expression Data : Identification and Analysis of Coexpressed Genes Exploring Expression Data : Identification and Analysis of Coexpressed Genes," no. 213: 1106–15. doi:10.1101/gr.9.11.1106.

Hoff, Kenneth E., John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. 1999. "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware." In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '99*, 277–86. New York, New York, USA: ACM Press. doi:10.1145/311535.311567.

Hoff, Kenneth E., Andrew Zaferakis, and Ming Lin. 2002. "Fast 3d Geometric Proximity Queries between Rigid and Deformable Models Using Graphics Hardware Acceleration." *UNC-CH Technical Report*. http://gamma.cs.unc.edu/PIVOT/pivot3d.pdf.

Hoff III, Kenneth E, Andrew Zaferakis, Ming Lin, and Dinesh Manocha. 2001. "Fast and Simple 2D Geometric Proximity Queries Using Graphics Hardware." *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, 145–48. doi:10.1145/364338.364383.

Hubbard, Philip M. 1993a. "Interactive Collision Detection." *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, 24–31. doi:10.1109/VRAIS.1993.378267.

———. 1993b. "Space-Time Bounds for Collision Detection."

———. 1995a. "Collision Detection for Interactive Graphics Applications." *IEEE Transactions on Visualization and Computer Graphics* 1 (3): 218–30. doi:10.1109/2945.466717.

———. 1995b. "Real-Time Collision Detection and Time-Critical Computing." *Sive 95* (July): 92–96.

———. 1996. "Approximating Polyhedra with Spheres for Time-Critical Collision Detection." *ACM Trans. Graph.* 15 (3): 179–210. doi:10.1145/231731.231732.

Hudson, Thomas C, Lin Min C, Jonathan Cohen, Stefan Gottschalk, and Dinesh Manocha. 1997. "V-COLLIDE: Accelerated Collision Detection for VRML." *VRML '97 Proceedings of the Second Symposium on Virtual Reality Modeling Language*, 117–24. doi:10.1145/253437.253472.

Hughes, John F., Andries Van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, and Kurt Akeley. 2013. *Computer Graphics: Principles and Practice (3rd Edition)*. Third Edit.

Jiménez, Juan Jose, Francisco R. Feito, and Rafael Segura. 2003. "Algoritmos Para La Detección de Colisión 2D." In *XIII Congreso Español de Informática Gráfica (CEIG'03)*.

———. 2004. "Utilización de Tetra-Trees Y Recubrimientos Simpliciales Para La Detección de Colisión Entre Esfera Y Poliedro Complejo." In *XIV Congreso Español de Informática Gráfica (CEIG'04)*.

Jimenez, Pablo, F Thomas, and Carme Torras. 2001. "3D Collision Detection: A Survey" 25.

Jordan, Camille. 1887. *Course D ´Analyse*. Paris: École Polytchnique de Paris.

Katherin.memi. n.d. "Sinestesia, Kinestesia Y Cenestesia." https://www.clubensayos.com/Psicología/Sinestesia-Kinestesia-Y-Cenestesia/2629021.html.

Kaufman, Leonard, and Peter J. Rousseeuw. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons. Copyright. Publication.

Kim, Duksu, Jae Pil Heo, Jaehyuk Huh, John Kim, and Sung Eui Yoon. 2009. "HPCCD: Hybrid Parallel Continuous Collision Detection Using CPUs and GPUs." *Computer Graphics Forum* 28 (7): 1791–1800. doi:10.1111/j.1467-8659.2009.01556.x.

Kim, Laehyun, G. S. Sukhatme, and M. Desbrun. 2003. "Haptic Editing of Decoration and Material Properties." *Proceedings - 11th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, HAPTICS 2003*, 213–20. doi:10.1109/HAPTIC.2003.1191280.

Kim, Young J., Ming C. Lin, and Dinesh Manocha. 2002. "DEEP: Dual-Space Expansion for Estimating Penetration Depth between Convex Polytopes." *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on* 1: 921–26 o.1. doi:10.1109/ROBOT.2002.1013474.

Klosowski, James T., Martin Held, Joseph S B Mitchell, Henry Sowizral, and Karel Zikan. 1998. "Efficient Collision Detection Using Bounding Volume Hierarchies of K-DOPs." *IEEE Transactions on Visualization and Computer Graphics* 4 (1): 21–36. doi:10.1109/2945.675649.

Knuth, Donald. 1997. *The Art of Computer Programming.* Volume 2: Seminumerical Algorithms. 3d Edition. Addison-Wesley.

Krishnan, S. Hema, Meenakshisundaram Gopi, Ming Lin, Dinesh Manocha, and Amol Pattekar. 1998. "Rapid and Accurate Contact Determination between Spline Models Using ShellTrees." *Computer Graphics Forum* 17 (3): 315–26. doi:10.1111/1467-8659.00278.

Larsen, Eric, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. 2000. "Fast Distance Queries with Rectangular Swept Sphere Volumes." In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, 4:3719–26. IEEE. doi:10.1109/ROBOT.2000.845311.

Larsen, Eric, Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. 1999. "Fast Proximity Queries with Swept Sphere Volumes." *Technical Report of Department of Computer Science, UNC Chapel Hill*, 1–32. doi:10.1109/ROBOT.2000.845311.

Latombe, Jean-Claude. 1991. *Robot Motion Planning*. Book Robot Motion Planning Kluwer Academic Publishers Norwell, MA, USA ©1991.

Lauterbach, Christian, Qi Mo, and Dinesh Manocha. 2010. "GProximity: Hierarchical GPU-Based Operations for Collision and Distance Queries." *Computer Graphics Forum* 29 (2): 419–28. doi:10.1111/j.1467-8659.2009.01611.x.

Lawlor, Orion Sky, and Laxmikant V. Kalée. 2002. "A Voxel-Based Parallel Collision Detection Algorithm." *Proceedings of the 16th International Conference on Supercomputing - ICS '02*, 285. doi:10.1145/514230.514231.

Lin, M, and Stefan Gottschalk. 1998. “Collision Detection between Geometric Models: A Survey.” *Proc. of IMA Conference on Mathematics of Surfaces*, 1–20. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.506&rep=rep1&type=pdf.

Lin, Ming C. 1993. “Efficient Collision Detection for Animation and Robotics.” *Electrical Engineering*, 159. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.1507&rep=rep1&type=pdf.

Lin, Ming C., and J. F. Canny. 1991. “Efficient Algorithms for Incremental Distance Computation.” In *IEEE Conference on Robotics and Automation*, 1008–14.

Lin, Ming C., Dinesh Manocha, Jon Cohen, and Stefan Gottschalk. 1996. “Collision Detection : Algorithms and Applications.”

Lubachevsky, Boris D. 1991. “How to Simulate Billiards and Similar Systems.” *Journal of Computational Physics* 94 (2): 255–83. doi:10.1016/0021-9991(91)90222-7.

Mäntylä, Martti. 1988. *An Introduction to Solid Modeling*. Computer Science Press.

Martínez, Jonatan, Arturo S. García, Diego Martínez, and Pascual González. 2009. “Desarrollo de Un Guante de Datos Con Retorno Háptico Vibro-Táctil Basado En Arduino.” *Interacción 2009 - Jornadas de Realidad Virtual*, 1–10.

McNeely, William A., Kevin D. Puterbaugh, and James J. Troy. 1999. “Six Degree-of-Freedom Haptic Rendering Using Voxel Samping.” *Comput. Graph. (SIGGRAPH Proc.)*, 401–8. doi:10.1145/1198555.1198605.

Melero, Francisco Javier. 2008. “BP-Octree : Una Estructura Jerárquica de Volúmenes Envolventes.” Universidad de Granada.

Melero, Francisco Javier, Pedro Cano, and Juan Carlos Torres. 2008. “Bounding-Planes Octree: A New Volume-Based LOD Scheme.” *Computers and Graphics (Pergamon)* 32 (4): 385–92. doi:10.1016/j.cag.2008.04.008.

Mirtich, Brian. 2000. “Timewarp Rigid Body Simulation.” *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 193–200. doi:10.1145/344779.344866.

Mirtich, Brian, and Tr-- June. 1997. "V-Clip: Fast and Robust Polyhedral Collision Detection." doi:10.1145/285857.285860.

Mohd Suaib, Norhaida, Abdullah Bade, and Dzulkifli Mohamad. 2008. "Collision Detection Using Bounding Volume for Avatars in Virtual Environment Applications," 486–91.

Möller, Tomas, and Ben Trumbore. 2005. "Fast, Minimum Storage Ray/Triangle Intersection." *ACM SIGGRAPH 2005 Courses*, no. 1: 1–7. doi:10.1145/1198555.1198746.

Moore, Matthew, and Jane Wilhelms. 1988. "Collision Detection and Response for Computer Animation." *Computer Graphic* 22 (4): 289–98. doi:10.1145/378456.378528.

Morton, G.M. 1966. "A Computer Oriented Geodetic Data Base and a New Technique in the File Sequencing," 20.

Naylor, Bruce, John Amanatides, and William Thibault. 1990. "Merging BSP Trees Yields Polyhedral Set Operations." *ACM SIGGRAPH Computer Graphics* 24 (4): 115–24. doi:10.1145/97880.97892.

Ng, Raymond T, and Jiawei Han. 1994. "Efficient and Effective Clustering Data Mining Methods for Spatial." *Proceedings of the 20th VLDB Conference Santiago, Chile*, 144–55. http://www.vldb.org/conf/1994/P144.PDF.

Nguyen, Hubert. 2007. *Gpu Gems 3*. Addison-Wesley Professional ©2007.

O'Rourke, Joseph. 1994. *Computational Geometry in C. Journal of Chemical Information and Modeling*. Vol. 53. Cambridge University Press.

O'Sullivan, Carol, John Dingliana, Fabio Ganovelli, and Gareth Bradshaw. 2001. "Collision Handling for Virtual Environments." *Training and Education*. http://www.tara.tcd.ie/handle/2262/18959.

"Octree." n.d. https://es.wikipedia.org/wiki/Octree.

Ogayar, Carlos J., Rafael J. Segura, and Francisco R. Feito. 2005. "Point in Solid Strategies." *Computers and Graphics (Pergamon)* 29 (4): 616–24. doi:10.1016/j.cag.2005.05.012.

Ortega, Lidia, and Francisco R. Feito. 2005. "Collision Detection Using Polar Diagrams." *Computers & Graphics* 29 (5): 726–37. doi:10.1016/j.cag.2005.08.026.

Pabst, Simon, Artur Koch, and Wolfgang Straßer. 2010. "Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces." *Computer Graphics Forum* 29 (5): 1605–12. doi:10.1111/j.1467-8659.2010.01769.x.

Palmer, Ian John, and R. L. Grimsdale. 1995. "Collision Detection for Animation Using Sphere-Trees." *Computer Graphics Forum* 14 (2): 105–16. doi:10.1111/1467-8659.1420105.

Pan, Jia, Sachin Chitta, and Dinesh Manocha. 2012. "FCL: A General Purpose Library for Collision and Proximity Queries." *Proceedings - IEEE International Conference on Robotics and Automation*, 3859–66. doi:10.1109/ICRA.2012.6225337.

Pentland, Alex. 1990. "Computational Complexity Versus Simulated Environments." *Computer Graphics Forum* 22 (2): 185–92.

Pobil, Pascual Angel del, and M.A. Serna Ferre. 1994. "A New Object Representation for Robotics and Artificial Intelligence Applications." *INTERNATIONAL JOURNAL OF ROBOTICS AND AUTOMATION* 9 (1): 11–21.

Ponamgi, Madhav, Dinesh Manocha, and Ming C Lin. 1995. "Incremental Algorithms for Collision Detection between Solid Models." *Proceeding SMA '95 Proceedings of the Third ACM Symposium on Solid Modeling and Applications*, no. 8920219: 293–304. doi:10.1145/218013.218076.

Quinlan, Sean. 1994. "Efficient Distance Computation between Non-Convex Objects." *Icra*, 3324–29. doi:10.1109/ROBOT.1994.351059.

Requicha, Aristides G. 1980. "Representations for Rigid Solids: Theory, Methods, and Systems." *ACM Computing Surveys* 12 (4): 437–64. doi:10.1145/356827.356833.

Ritter, Jack. 1990. "An Efficient Bounding Sphere." In *Graphics Gems*, 301–3. Academic Press Professional, Inc. San Diego, CA, USA ©1990.

Rozas Merino, Oscar. 1997. "Detección de Colisiones En Simulación de Mecanismos Con Un Modelo de Representación Basado En Superficies NURBS."

Salisbury, Kenneth, Francois Conti, and Federico Barbagli. 2004. "Haptic Rendering : Introductory Concepts," no. April: 24–32.

Samet, Hanan, and Markku Tamminen. 1985. "Bintrees, CSGtrees, and Time." In *Proceedings of SIGGRAPH'85, Computer Graphics*, 121 – 130.

Samet, Hanan, and Robert E Webber. 1988. "Hierarchical Data Structures and Algorithms for Computer Graphics. II. Applications." *Computer Graphics and Applications, IEEE* 8 (4): 59–75. doi:10.1109/38.7750.

Savall, Joan, Diego Borro, Jorge Juan Gil, and Luis Matey. 2002. "Description of a Haptic System for Virtual Maintainability in Aeronautics." *2002 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2887–92. doi:10.1109/IRDS.2002.1041709.

Schroeder, Will, Ken Martin, and Bill Lorensen. 1996. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice-Hall, Inc.

Schulman, John, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. 2013. "Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization." *Robotics: Science and Systems* 9 (1): 1–10. doi:10.1.1.387.4642.

Selim, Shokri Z., and H. A. Almohamad. 1999. "Collision Computation of Moving Bodies." *European Journal of Operational Research* 119 (1): 121–29. doi:10.1016/S0377-2217(98)90356-3.

Sethian, James A. 1996. "A Fast Marching Level Set Method for Monotonically Advancing Fronts." *Pnas* 93 (4): 1591–95. doi:10.1073/pnas.93.4.1591.

Shaffer, Clifford A., and Gregory M. Herb. 1992. "A Real-Time Robot Arm Collision Avoidance System." *IEEE Transactions on Robotics and Automation* 8 (2): 149–60. doi:10.1109/70.134270.

Shimoga, K.B. n.d. "Finger Force and Touch Feedback Issues in Dexterous Telemanipulation." In *Proceedings. Fourth Annual Conference on Intelligent Robotic Systems for Space Exploration*, 159–78. IEEE. doi:10.1109/IRSSE.1992.671841.

Smith, Andrew, Yoshifumi Kitamura, Haruo Takemura, and Fumio Kishino. 1995. "A Simple and Efficient Method for Accurate Collision Detection among Deformable Polyhedral Objects in Arbitrary Motion." In *Proceedings Virtual Reality Annual International Symposium '95*, 136–45. IEEE Comput. Soc. Press. doi:10.1109/VRAIS.1995.512489.

Snyder, John M. 1992. "Interval Analysis for Computer Graphics." *ACM SIGGRAPH Computer Graphics* 26: 121–30. doi:10.1145/142920.134024.

Stroud, Ian. 2006. *Boundary Representation Modelling Techniques*. Springer Publishing Company, Incorporated ©2010.

Sud, Avneesh, Naga Govindaraju, Russell Gayle, Ilknur Kabul, and Dinesh Manocha. 2006. "Fast Proximity Computation among Deformable Models Using Discrete Voronoi Diagrams." *ACM Transactions on Graphics* 25 (3): 1144. doi:10.1145/1179352.1142006.

Suri, Subhash, Philip M Hubbard, and John F Hughest. 1998. "Collision Detection in Aspect and Scale Bounded Polyhedra." In *SODA '98 Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 127–36. http://dl.acm.org/citation.cfm?id=314692.

Swan, J. Edward. 1993. "Octree-Based Collision Detection with Fast Neighbor Finding." doi:OSU/ACCAD-12/93-TR7.

"Tacto." n.d. https://es.wikipedia.org/w/index.php?title=Tacto&oldid=83768675.

Tang, Min, Sean Curtis, Yoon Sung-Eui, and Dinesh Manocha. 2009. "ICCD: Interactive Continuous Collision Detection between Deformable Models Using Connectivity-Based Culling." *Visualization and Computer Graphics, IEEE Transactions on* 15 (4): 544–57. doi:10.1109/tvcg.2009.12.

Tang, Min, Dinesh Manocha, and Ruofeng Tong. 2009. "Multi-Core Collision Detection Between Deformable Models." *SIAM/ACM Joint Conference on Geometric and Physical Modeling*, 355–360. doi:10.1145/1629255.1629303.

———. 2010. "Fast Continuous Collision Detection Using Deforming Non-Penetration Filters." *Interactive 3D Graphics and Games* 1 (212): 7–13. doi:10.1145/1730804.1730806.

Tang, Min, Ruofeng Tong, Zhendong Wang, and Dinesh Manocha. 2014. "Fast and Exact Continuous Collision Detection with Bernstein Sign Classification." *ACM Transactions on Graphics* 33 (6): 186. doi:10.1145/2661229.2661237.

Täubig, Holger, and Udo Frese. 2012. "A New Library for Real-Time Continuous Collision Detection." *Robotik 2012* 1. http://www.vde-verlag.de/proceedings-en/453418020.html.

"Tecnología Háptica." n.d. *Ecured*. http://www.ecured.cu/index.php/Tecnología_háptica.

Teschner, Matthias, Bruno Heidelberger, Dinesh Manocha, Naga Govindaraju, Gabriel Zachmann, Stefan Kimmerle, Johannes Mezger, and Arnulph Fuhrmann. 2005. "[F]Collision Handling in Dynamic Simulation Environments." *Eurographics Tutorial # 2*, 1–4.

Teschner, Matthias, Bruno Hiedelberger, Matthias Müller, Danat Pomeranets, and Markus Gross. 2003. "Optimized Spatial Hashing for Collision Detection of Deformable Objects." *Vmv 2003*, 8. doi:10.1.1.4.5881.

Teschner, Matthias, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrman, Marie-paule Cani, et al. 2004. "Collision Detection for Deformable Objects." *Computer Graphics Forum* 23 (1): 61–81.

Thibault, William C., and Bruce F. Naylor. 1987. "Set Operations on Polyhedra Using Binary Space Partitioning Trees." *ACM SIGGRAPH Computer Graphics* 21 (4): 153–62. doi:10.1145/37402.37421.

Thomaszewski, Bernhard, Simon Pabst, and Wolfgang Blochinger. 2008. "Parallel Techniques for Physically Based Simulation on Multi-Core Processor Architectures." *Computers & Graphics* 32 (1): 25–40. doi:10.1016/j.cag.2007.11.003.

Traub, J. F. 1967. "Interval Analysis. Ramon E. Moore. Prentice-Hall, Englewood Cliffs, N.J., 1966. 159 Pp., Illus. $9." *Science* 158 (3799): 365–365. doi:10.1126/science.158.3799.365.

Vemuri, Baba C, Y. Cao, and Li Chen. 1998. "Fast Collision Detection Algorithms with Applications to Particle Flow." *Computer Graphics Forum* 17 (2): 121–34. doi:10.1111/1467-8659.00233.

Volino, Pascal, and Nadia Magnenat-Thalmann. 1997. "Interactive Cloth Simulation: Problems and Solutions." *Jws97-B* 94.

Weller, René. 2013. *New Geometric Data Structures for Collision Detection and Haptics*. *Journal of Chemical Information and Modeling*. Vol. 53. Springer Series on Touch and Haptic Systems. Heidelberg: Springer International Publishing. doi:10.1007/978-3-319-01020-5.

Welzl, Emo. 1991. "Smallest Enclosing Disks (Balls and Ellipsoids)." *New Results and New Trends in Computer Science* 11 (3): 359–70. doi:10.1007/BFb0038178.

"Wikipedia, 'Head-Mounted Display.'" n.d. https://en.wikipedia.org/wiki/Head-mounted_display.

Wilson, Andrew, Eric Larsen, Dinesh Manocha, and Ming C Lin. 1999. "Partitioning and Handling Massive Models for Interactive Collision Detection." *Computer Graphics Forum* 18 (3): 319–30. doi:10.1111/1467-8659.00352.

Wilson, Andrew, E Larsen D Manocha, M C Lin, and Chapel Hill. n.d. "IMMPACT : A System for Interactive Proximity Queries On Massive Models 1 Introduction." *Managing*, 1–26.

Wong, Wingo Sai-Keung, and George Baciu. 2006. "A Randomized Marking Scheme for Continuous Collision Detection in Simulation of Deformable Surfaces." In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications - VRCIA '06*, 181. New York, New York, USA: ACM Press. doi:10.1145/1128923.1128954.

Zachmann, Gabriel. 1997. "Real-Time and Exact Collision Detection for Interactive Virtual Prototyping." *Design Engineering*.

———. 1998. "Rapid Collision Detection by Dynamically Aligned DOP-Trees." *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium (Cat. No.98CB36180)*, 90–97. doi:10.1109/VRAIS.1998.658428.

Zhang, Dongliang, and Matthew M F Yuen. 2002. "A Coherence-Based Collision Detection Method for Dressed Human Simulation." *Computer Graphics Forum* 21 (1): 33–42. doi:10.1111/1467-8659.00564.