UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Máster Universitario de Investigación en Ingeniería de Software
y Sistemas Informáticos

Trabajo Fin de Máster - Código 31105151

Mejoras de los Motores Físicos: ODE y Bullet

Autor: Javier de la Puente Alonso
Tutor: Juan José Escribano Ródenas
Año 2020/2021
Convocatoria de febrero

Máster Universitario de Investigación en Ingeniería de Software
y Sistemas Informáticos

Trabajo Fin de Máster - Código 31105151

Mejoras de los Motores Físicos: ODE y Bullet

Tipo Trabajo A

Autor: Javier de la Puente Alonso
Tutor: Juan José Escribano Ródenas

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MÁSTER

Fecha: 26/01/2021

Quién suscribe:

Autor: JAVIER DE LA PUENTE ALONSO
D.N.I.: 71938158T

Hace constar que es el autor del trabajo:

Título completo del trabajo.

Mejoras de los motores físicos: ODE y Bullet

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.

DE LA PUENTE ALONSO, JAVIER (FIRMA)

Digitally signed by DE LA PUENTE ALONSO, JAVIER (FIRMA)
Date: 2021.01.26 16:12:18 +01'00'

Impreso TFdM05_AutorPbl. Autorización de publicación y difusión del TFM para fines académicos

Autorización

Autorizo a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y el prototipo desarrollado.

Firma del Autor

DE LA PUENTE ALONSO, JAVIER (FIRMA)

Digitally signed by DE LA PUENTE ALONSO, JAVIER (FIRMA)
Date: 2021.01.26 16:13:02 +01'00'

Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09
www.issi.uned.es

Resumen

La simulación de la dinámica del cuerpo rígido constituye una simplificación de la realidad con múltiples aplicaciones que abarcan desde la animación y los videojuegos a la robótica y la simulación.

En este trabajo se presenta un análisis y estudio de los dos principales motores físicos de software abierto, ODE y Bullet. Este análisis se ha realizado de una forma sistemática teniendo en cuenta tanto sus características arquitectónicas como los algoritmos utilizados. Además, los dos motores se han comparado de forma práctica sobre cinco escenarios sintéticos que miden la plausibilidad, la precisión física de la simulación, la robustez y el rendimiento. Para la realización de esta comparativa se ha utilizado la herramienta Obugre, creada dentro del ámbito de este proyecto, que ofrece una interfaz común a ambos motores y permite la visualización de la simulación.

Derivadas del estudio del arte de los motores físicos y del análisis de ODE y Bullet se han identificado posibles mejoras, de entre las que se ha optado por implementar y evaluar el algoritmo V-Clip. Este algoritmo es capaz de encontrar el par de características más cercanas entre dos poliedros convexos y como se demuestra en este trabajo, es un firme contendiente de los algoritmos GJK y MPR utilizados en ODE y Bullet.

Palabras clave— Cuerpo Rígido, Motor Físico, ODE, Bullet, Detección de Colisiones, V-Clip

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Capítulo 1

Introducción

1.1. Introducción y motivación

El presente Trabajo Fin de Máster, de Tipo A y por lo tanto propuesto por un profesor, corresponde al título "Mejoras de los Motores Físicos: ODE y Bullet". En el enunciado del trabajo se solicita un estudio de las posibilidades de estos dos motores y el planteamiento y evaluación de posibles mejoras, dentro del ámbito de la simulación dinámica del cuerpo rígido.

La simulación del cuerpo rígido pertenece al vasto campo de la simulación física, que ha sido desde el inicio uno de los principales usos de los primeros ordenadores modernos. Su base teórica se sustenta en la mecánica clásica no relativista, siendo una simplificación de la realidad, ya que todos los cuerpos reales son deformables. La mecánica de cuerpos rígidos es, además, dentro de la física, una extensión de los sistemas de partículas.

Una simulación es una imitación aproximada de la operación de un proceso o sistema (Banks, Carson & Nelson, 2005). Para analizar el comportamiento del sistema según avanza en el tiempo es necesario desarrollar un modelo del mismo. Este modelo incluirá unas asunciones con respecto al funcionamiento del sistema, que serán expresadas de forma matemática, lógica o simbólica entre las entidades de interés. El objetivo de la simulación determinará la complejidad del modelo, y qué aspectos de la realidad modela.

En el caso de la simulación del cuerpo rígido, el punto de partida son las ecuaciones de Newton-Euler. Simular la trayectoria balística de un único

cuerpo en el vacío es relativamente sencillo, comenzando las dificultades al considerar aspectos como restricciones de movimientos, fricción y colisiones entre ellos. Para ello los motores físicos hacen uso de multitud de ramas de la ciencia tales como la mecánica analítica, geometría computacional, métodos numéricos, álgebra lineal, programación matemática...

A la hora de modelar cualquier sistema será necesario realizar un compromiso entre sus propiedades, de forma que se puedan conseguir los objetivos deseados. Una primera división de este tipo puede ser entre sistemas con alta precisión y sistemas en tiempo real. Sin embargo, no es la única, y así podrían preferirse modelos que sean estables a condiciones iniciales mal formuladas, aunque no se ajusten tan fielmente a la realidad, o modelos más simples de implementar. Es desde este prisma, los compromisos tomados a la hora de crear las librerías y aplicaciones para la simulación física (los motores físicos), desde el que se estudiarán las posibilidades de ODE y Bullet.

Tanto ODE (https://www.ode.org) como Bullet (https://pybullet.org) son motores físicos complejos, estables y consolidados que se han utilizado para multitud de aplicaciones prácticas, desde la animación y los videojuegos hasta la robótica y la investigación. Es difícil encontrar posibilidades de mejora que no sean triviales o excesivamente complicadas, que sean relevantes y que se puedan aplicar a ambos motores.

Como se verá más adelante, la implementación del algoritmo de detección de colisiones entre poliedros convexos V-Clip (Mirtich, 1998) es la opción adoptada, que trata de satisfacer los compromisos entre el interés teórico y práctico.

1.2. Simulación interactiva del cuerpo rígido.

Un cuerpo rígido es una noción idealizada en la que la distancia entre cualquier par de puntos es constante. Aunque no exenta de dificultades, esta simplificación de la realidad ha facilitado las simulaciones de escenas complejas y realistas (de forma física o aparente), capaces de ser ejecutadas en tiempo real, permitiendo incluso la interacción en bucle cerrado entre el simulador y una persona (Bender, Erleben, Trinkle & Coumans, 2012).

La simulación interactiva del cuerpo rígido es una disciplina madura, en la que las últimas tendencias se centran en crear simulaciones más rápidas o

escenas más grandes (Bender y col., 2012). Igualmente, se puede apreciar una tendencia a limitar la restricción de cuerpo rígido, permitiendo la interacción de cuerpos rígidos y deformables, como de hecho implementa Bullet. Aunque no se trata en este trabajo, la inclusión de cuerpos deformables presenta numerosas trabas. Así, los cuerpos deformables se modelan con derivadas parciales en vez de derivadas ordinarias, la detección de colisiones es más complicada y aparecen fenómenos como ondas.

Además de Bullet y Ode existen multitud de motores físicos que simulan el cuerpo rígido. Algunos de los más relevantes son los siguientes:

- Havok
- Physx
- Newton
- Box2D
- Mujoco
- Dart
- Simbody

1.3. Aplicaciones de los motores físicos

Las aplicaciones de los motores físicos son muy amplias. En el espectro se pueden encontrar desde aquellas que requieren una elevada calidad en la simulación y son "off-line" de las que requieren un mayor rendimiento para poder ofrecer resultados en tiempo real o incluso más rápido. Los motores ODE y Bullet tratados en este trabajo se orientan más hacia este último caso.

Un listado no exhaustivo de aplicaciones es el siguiente (Bender y col., 2012; Couman, s.f.; Russell Smith, 2004):

Animación por ordenador. Gracias a los motores físicos se pueden realizar animaciones muy realistas en escenarios complejos. Inicialmente, uno de los primeros usos de los motores físicos fue la animación de caídas o muertes de personajes (ragdoll).

Videojuegos. Al igual que en animación por ordenador, se persigue la plausibilidad (credibilidad) frente a la fidelidad física, pero en este caso es necesario un rendimiento en tiempo real. Con el uso de un motor físico no solo se consigue un mayor realismo, también se puede generar un "comportamiento emergente", en el que el usuario puede experimentar un videojuego de forma distinta a la prevista por el autor.

Robótica. Un motor físico permite realizar y desarrollar prototipos y pruebas en entornos peligrosos, no disponibles o reducir el coste. Además, permite entrenar algoritmos tanto de alto nivel como de bajo nivel en elementos de control. Aplicaciones en los campos de telecirugía, vehículos autónomos y simulación háptica son cada vez más comunes.

Realidad virtual, simuladores y entrenamiento. Los simuladores permiten entrenar a personas en situaciones de riesgo y que de otra forma sería imposible, como por ejemplo en la industria aeroespacial.

Diseño por ordenador y prototipado. Además del prototipado y análisis de riesgos digital de plantas y mecanismos, un motor físico permite el diseño y desarrollo de sensores como giroscopios y el de aplicaciones para mejorar la estabilidad en vehículos y maquinaria.

Experimentos físicos para formación e instrucción. Además de para formación e instrucción, un motor físico permite el estudio de movimientos extraños o difíciles de explicar como podría ser, por ejemplo, el caso de la peonza celta.

1.4. Ámbito y Objetivos

Este trabajo explorará únicamente los motores físicos desde la perspectiva del cuerpo rígido. No se hará ninguna referencia a otros aspectos como deformaciones, ondas o fluidos. Debido a que aun así el ámbito de la simulación del cuerpo rígido es demasiado amplio, en el motor físico Bullet no se tratarán de forma práctica aspectos relativos a la simulación con GPGPU ni el uso de coordenadas reducidas (algoritmo de Featherstone), a pesar de su clara utilidad en robótica.

En el enunciado del trabajo se solicitan dos objetivos claramente diferenciados. El primer objetivo del trabajo es el estudio de las posibilidades de los motores físicos ODE y Bullet. Para ello se estudiarán ambos motores y

se realizará una batería de casos de uso sobre ellos. Aunque no se plantea un análisis exhaustivo de todas las características de los motores, este estudio permitirá comparar ambos motores y obtener conclusiones acerca de sus posibilidades.

El segundo objetivo del trabajo es plantear y evaluar posibles mejoras de los mismos. Se expondrán algunas de las posibles mejoras encontradas en el estudio de la literatura y de los motores ODE y Bullet. Se implementará una de ellas, el algoritmo V-Clip, en ambos motores y se analizarán los resultados.

Tanto para el estudio de los motores físicos como del algoritmo implementado será necesario una herramienta gráfica, no solo para la mejor comprensión del problema sino también para evaluar subjetivamente la plausibilidad de las simulaciones.

1.5. Contenido del proyecto

El contenido del presente Trabajo Fin de Máster es el siguiente:

Introducción.

Estado del arte. Se introducen en este capítulo los principales algoritmos y paradigmas utilizados en la creación de un motor físico del cuerpo rígido.

ODE. Resumen de las principales características, algoritmos y arquitectura del motor físico ODE.

Bullet. Resumen de las principales características, algoritmos y arquitectura del motor físico Bullet.

Posibilidades y comparativa. En este capítulo se comparan los datos obtenidos de los dos capítulos anteriores y se realizan pruebas sobre escenarios de simulación en ambos motores, dando como resultado una evaluación y estudio de posibilidades de ODE y Bullet.

Mejora a los motores físicos. Algoritmo V-Clip. De toda la información obtenida hasta este punto, se proponen posibles ideas de mejora para ODE y Bullet. Entre ellas se explica la seleccionada, el algoritmo V-Clip, y su integración con los motores.

V-Clip. Análisis y resultados de la mejora. Evaluación tanto del algoritmo V-Clip como de la implementación realizada. Se compararán los motores con su algoritmo original y con la implementación de V-Clip realizada en este trabajo.

Conclusiones y líneas futuras de investigación.

Apéndice A. Entorno de pruebas. Descripción del entorno hardware y software utilizado para los escenarios y comparativas.

Anexo I. Guía de instalación y uso del software. Requisitos, instalación y explicación del funcionamiento del software desarrollado para la realización de este proyecto.

Capítulo 2

Estado del arte

En la comunidad de informática gráfica, especialmente aplicado al caso de la animación y los videojuegos, se suele hacer referencia al término animación basada en física (physics-based animation), en el que en ocasiones se prima la "plausibilidad", o si las personas perciben los movimientos como realistas, sobre la realidad física.

En este apartado se hace un resumen del estado del arte en la simulación del cuerpo rígido, en el que en especial se hará incidencia sobre la animación basada en física. Aunque es un campo de conocimiento activo y con nuevas técnicas y aplicaciones en la literatura, su base es madura y se considera desarrollada. Parte de la exposición en este capítulo ya se puede encontrar en el artículo «Non-penetrating Rigid Body Simulation» (Baraff, 1993).

De forma muy genérica se pueden diferenciar dos partes fundamentales en un motor físico. La primera, puramente geométrica, consiste en la detección de las colisiones. La segunda consiste en la simulación de los cuerpos teniendo en cuenta las restricciones oportunas, entre ellas las colisiones encontradas.

El objetivo por lo tanto es enumerar, introducir y comentar los fundamentos, prácticas y principales algoritmos, tanto de detección de colisiones como de simulación. La información recopilada en este capítulo será la base tanto del estudio teórico de ODE y Bullet como de la selección de mejoras. Para una descripción más completa y detallada de la simulación del cuerpo rígido se recomiendan las siguientes referencias: Eberly (2003), Witkin y Baraff (1997), Erleben, Sporring, Henriksen y Dohlman (2005), y Baraff (1993).

2.1. Simulación del cuerpo rígido

En física, un cuerpo rígido es una abstracción de la realidad en la que se considera que no existen deformaciones o que estas son despreciables. Es una evolución de los sistemas de partículas en que las distancias entre todas las partículas del mismo cuerpo son constantes.

La utilización de la asunción del cuerpo rígido ofrece muchas ventajas, tanto de simplicidad como de rendimiento en la simulación. Así las ecuaciones se modelan como ecuaciones diferenciales ordinarias en vez de ecuaciones en derivadas parciales como ocurre en los cuerpos deformables o en dinámica de fluidos. La dinámica del cuerpo rígido es un campo de la mecánica y de la mecánica analítica que ha sido estudiado en profundidad y para el cual hay multitud de formulaciones, siendo las más conocidas la lagrangiana y la hamiltoniana. Además, en el caso de colisiones entre cuerpos rígidos, estas se deben resolver en tiempo cero al no haber deformaciones (Erleben y col., 2005).

Se enumerarán primero los métodos matemáticos para la representación de la posición y la orientación. Para la simulación mecánica, la base teórica está basada en la ecuación de Newton-Euler, en sus distintas formulaciones. Se introducirá después el problema de las restricciones, entre las cuales están las colisiones, y las principales formas de resolverlo, para comentar posteriormente cómo se integran las ecuaciones diferenciales.

2.1.1. Sistemas de coordenadas

Un cuerpo rígido no restringido en el espacio tiene seis grados de libertad, correspondiendo tres de ellos a la traslación y tres de ellos a la orientación. A la hora de representar la orientación, esta se puede realizar por distintas técnicas:

- Ángulos de Euler.
- Pares de rotación y mapas exponenciales (Grassia, 1998).
- Matrices de rotación.
- Cuaternios unitarios.

Siendo los dos últimos los más utilizados en simulación del cuerpo rígido, al ser los que no presentan singularidades con respecto a las rotaciones. En el caso de la simulación dinámica, los cuaternios presentan ventajas al utilizar solo cuatro elementos y ser más fáciles de normalizar que las matrices de rotación.

Es habitual disponer de un sistema de coordenadas global y un sistema de coordenadas en el centro de gravedad de cada cuerpo alineado con sus momentos principales de inercia. A partir de ahora se utilizará el término centro de masas indistintamente al centro de gravedad, aunque no son equivalentes si la gravedad no es constante. Las transformaciones de un sistema a otro son transformaciones isométricas, fáciles de combinar mediante matrices de transformación homogéneas, o utilizando vectores de traslación y matrices de rotación o cuaternios (Featherstone, 2007).

2.1.2. Ecuaciones del movimiento

La base de las ecuaciones del movimiento son las leyes de Newton. Se recuerda aquí que un sistema inercial es aquel en que las leyes físicas se escriben en su forma estándar y en donde hay conservación del momento y la energía.

Para el caso del cuerpo rígido, y con respecto a un sistema de coordinadas con centro el centro de masas, las ecuaciones de Newton-Euler, que determinan su estado, son las siguientes:

$$\boldsymbol{F} = m\boldsymbol{a} \tag{2.1}$$

$$\boldsymbol{\tau} = \boldsymbol{I}\frac{d\boldsymbol{\omega}}{dt} + \boldsymbol{\omega} \times (\boldsymbol{I}\boldsymbol{\omega}) \tag{2.2}$$

De forma matricial:

$$\begin{pmatrix} \mathbf{F} \\ \boldsymbol{\tau} \end{pmatrix} = \begin{pmatrix} m\mathbf{I}_3 & 0 \\ 0 & \mathbf{I}_{\mathrm{cm}} \end{pmatrix} \begin{pmatrix} \mathbf{a}_{\mathrm{cm}} \\ \boldsymbol{\alpha} \end{pmatrix} + \begin{pmatrix} 0 \\ \boldsymbol{\omega} \times \mathbf{I}_{\mathrm{cm}}\, \boldsymbol{\omega} \end{pmatrix} \tag{2.3}$$

Esta no es la única formulación, y se puede realizar con otros sistemas,

ya sea no inerciales o con otro punto de referencia, siendo esta formulación la más simple y la que se suele utilizar en las simulaciones de cuerpos rígidos (Ploen, Hadaegh & Scharf, 2004). Más adelante en este trabajo la formulación utilizada se deberá tener en cuenta para calcular la energía, el momento lineal y el momento angular del sistema, de forma que se calculen en un sistema inercial.

Dentro de las ecuaciones del movimiento, el término $\boldsymbol{\omega} \times \mathbf{I}_{\mathrm{cm}}\, \boldsymbol{\omega}$ se corresponde a lo que se denomina el término centrífugo giroscópico (Lacoursière, 2006). En esta formulación, este término es debido a que el tensor de inercia es fijo con respecto al sistema de coordenadas no inercial fijo con el cuerpo rígido, pero que al utilizar un sistema inercial, el tensor de inercia varía. Este término a veces se ignora en la animación basada en física, aunque desde luego esto resulta en un comportamiento no físico.

Utilizando cuaternios, la derivada de la posición con respecto al tiempo es:

$$\frac{d\mathbf{q}(t)}{dt} = \frac{1}{2}\boldsymbol{\omega}(t)\mathbf{q}(t) \tag{2.4}$$

Utilizando matrices de rotación:

$$\frac{d\mathbf{R}(t)}{dt} = \boldsymbol{\omega}(t)^{\times}\mathbf{R}(t) \tag{2.5}$$

utilizándose el símbolo $^{\times}$ en $\boldsymbol{\omega}^{\times}$ para denotar la matriz antisimétrica:

$$\begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \tag{2.6}$$

Combinando las ecuaciones, cada cuerpo rígido se resuelve utilizando las ecuaciones:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \tag{2.7}$$

$$\frac{d\mathbf{q}}{dt} = \frac{1}{2}\boldsymbol{\omega}(t)\mathbf{q}(t) \tag{2.8}$$

$$\frac{d\mathbf{v}}{dt} = m^{-1}\mathbf{f} \tag{2.9}$$

$$\frac{d\boldsymbol{\omega}}{dt} = -I^{-1}\boldsymbol{\omega} \times I\boldsymbol{\omega} + \boldsymbol{\tau} \tag{2.10}$$

Para simplificar la terminología se suele utilizar el vector generalizado de posición y orientación, que incluye los tres componentes de la posición y cuatro de orientación con cuaternios, y el vector generalizado de velocidad, que incluye los tres de velocidad lineal y tres de velocidad angular.

En cuanto al tensor de inercia, que es constante en un cuerpo rígido en su sistema de coordenadas, se suele dar alineado con los momentos principales de inercia (ejes principales del cuerpo), de modo que se puede representar con solo tres coordenadas. Para ejemplos de su cálculo se puede utilizar Taylor (2005), y en el caso de poliedros sólidos es especialmente útil Eberly (2003).

La formulación comentada hasta ahora corresponde con la también denominada en la literatura como coordenadas maximalistas. En la Sección 2.1.3 se enumeran otras formulaciones basadas en coordenadas reducidas.

Colisión en un punto

Cuando se produce un contacto entre dos cuerpos rígidos, A y B, en el instante $t = 0$ se puede investigar si los cuerpos están separándose, en contacto relativo o colisionando, analizando la velocidad relativa entre los puntos de contacto $\boldsymbol{p}_a$ y $\boldsymbol{p}_b$.

Así, las velocidades en los puntos $\boldsymbol{p}_a$ y $\boldsymbol{p}_b$, siendo $\boldsymbol{r}$ el punto entre el centro de masas y el punto de contacto relativo al centro de masas del cuerpo, son las siguientes:

$$\dot{\boldsymbol{p}}_a = \boldsymbol{v}_a + \boldsymbol{\omega}_a \times \boldsymbol{r}_a \tag{2.11}$$

$$\dot{\boldsymbol{p}}_b = \boldsymbol{v}_b + \boldsymbol{\omega}_b \times \boldsymbol{r}_b \tag{2.12}$$

Y siendo $\boldsymbol{N}$ la dirección del plano normal al contacto entre los dos cuerpos calculado por el sistema de colisiones, la distancia relativa entre los cuerpos es:

$$d = \boldsymbol{N} \cdot (\boldsymbol{p}_a - \boldsymbol{p}_b) \tag{2.13}$$

y la velocidad relativa:

$$v_{rel} = \boldsymbol{N} \cdot (\dot{\boldsymbol{p}}_a - \dot{\boldsymbol{p}}_b) + \dot{\boldsymbol{N}} \cdot (\boldsymbol{p}_a - \boldsymbol{p}_b) \tag{2.14}$$

Utilizándose solo $\boldsymbol{N} \cdot (\dot{\boldsymbol{p}}_a - \dot{\boldsymbol{p}}_b)$ en el instante de la colisión.

En el caso de que se desee evitar la interpenetración en los objetos, se deberá aplicar un impulso, de forma que habrá una discontinuidad en la velocidad de los cuerpos.

Siendo $\boldsymbol{J}$ el impulso definido como:

$$\boldsymbol{J} = \int_{t_0}^{t_1} \boldsymbol{F} dt = m\boldsymbol{v}(t_1) - m\boldsymbol{v}(t_0) \tag{2.15}$$

el problema se resuelve aplicando un impulso que evite la interpenetración.

En el caso simple sin fricción en una colisión elástica, el resultado será la reflexión de la velocidad relativa entre los cuerpos sobre la normal al contacto. Este impulso aplicado producirá tanto un cambio en la velocidad angular del cuerpo como en la velocidad del centro de masas, pero no un cambio en las posiciones de los cuerpos.

Para calcular un impulso es necesaria una ley de colisión, habiendo fundamentalmente cuatro tipos (Erleben y col., 2005, pp. 130-131):

- Leyes de colisión algebraicas.
- Leyes de colisión incrementales.
- Leyes de colisión con deformaciones completas.
- Leyes de colisión conforme al modelo de contacto.

Si la colisión no es perfectamente elástica se perderá energía. Para parametrizar la elasticidad de la colisión se utilizará el correspondiente coeficiente de restitución. Las principales leyes utilizadas, todas ellas aproximaciones de la realidad, son la ley de impacto de Newton, la hipótesis de Poisson y la hipótesis de Stronge. Estas se pueden consultar en Erleben y col. (2005), Mirtich (1996), y Eberly (2003).

En el caso de contactos en los que en la velocidad relativa entre los cuerpos sea cero, y que por lo tanto son "contactos en reposo" (resting contacts), la restricción es que aceleración relativa no puede ser negativa, ya que habría interpenetración. Una resolución simple de este caso se puede ver en Witkin y Baraff (1997).

2.1.3. Resolución de las restricciones

Se introducen en este apartado los principales paradigmas de simulación de cuerpos rígidos. Estos se diferencian en la forma de tratar las restricciones. Las restricciones pueden ser bilaterales, como son las articulaciones o unilaterales, como son las colisiones.

Métodos basados en penalizaciones

En este paradigma las colisiones se modelan como osciladores armónicos amortiguados (resorte con atenuador o sistema resorte-amortiguador). Es una de las aproximaciones más simples y se utiliza de forma muy similar para modelar sistemas deformables basados en resortes y amortiguadores. Una introducción a esta formulación se puede encontrar en Erleben y col. (2005).

La idea fundamental es que en el momento en que se detecte una interpenetración entre objetos, se inserta un oscilador armónico amortiguado entre los puntos de mayor penetración, se calculan las fuerzas resultantes y se incluyen en la integración de las ecuaciones del movimiento.

$$\boldsymbol{F}_{resorte} = -k_1\boldsymbol{x} \tag{2.16}$$

$$\boldsymbol{F}_{atenuador} = -k_2\dot{\boldsymbol{x}} \tag{2.17}$$

Aunque conceptualmente simple y capaz de generar fuerzas de reacción válidas físicamente, los principales problemas inherentes a este método son los siguientes:

- La integración de las ecuaciones diferenciales es problemática cuando las constantes de los resortes son elevadas, al ser ecuaciones rígidas (stiff equations). Si en cambio no son rígidas, la interpenetración será elevada.
- No es simple encontrar normales a las superficies de contacto y profundidades de las penetraciones, pudiéndose producir configuraciones erróneas.
- Determinar las constantes del oscilador es una tarea de prueba y error.

- Es necesario que haya una penetración para que haya una fuerza de reacción, por lo que se producen oscilaciones.

Métodos basados en impulsos

También denominados métodos basados en colisiones, consisten en aplicar secuencias de impulsos para resolver las colisiones. Aunque la separación entre métodos no siempre es clara (Stepie, 2013), se puede considerar que un método está basado en impulsos si:

- En cada momento solo una pareja de cuerpos se trata para resolver las colisiones.
- Las colisiones se tratan con una ley de impacto (Newton, Poisson, Stronge...) y no es necesario introducir restricciones adicionales.

Aunque estos métodos cuentan con ventajas ya que las colisiones se modelan de una forma natural y de forma local entre dos cuerpos, sus principales desventajas son las siguientes:

- Cuando los contactos son permanentes (resting contacts), el resultado es una secuencia de colisiones. En una pila de objetos esto implica inestabilidades.
- Las articulaciones determinan restricciones que difícilmente se van a cumplir de una forma estable con un método basado en impulsos.
- Es difícil integrar la fricción, y puede dar lugar a que no se mantenga correctamente la fricción estática. Esto puede ocurrir por ejemplo con un cubo que debería estar inmóvil en una rampa y que, sin embargo, debido a la secuencia de colisiones, va bajando dando pequeños saltos.

Existen muchas variantes de los métodos basados en impulsos, siendo muy recurrente en la literatura la aproximación de Mirtich (1996), basada en CA, (Conservative Advancement). En CA, el siguiente momento en el que se va a producir un impacto (TOI, Time Of Impact) se calcula de forma conservadora. Se integra hasta el TOI y si realmente existe la colisión se resuelve. Es habitual utilizar CA combinado con otros métodos (métodos híbridos) para tratar de solucionar algunos de sus problemas, ya sea con

métodos basados en restricciones o con métodos basados en coordenadas reducidas.

Métodos basados en restricciones

Tanto las colisiones como las articulaciones se pueden modelar como restricciones, ya sean geométricas o cinemáticas de los cuerpos rígidos. Existen muchas clasificaciones de métodos basados en restricciones, y se suelen clasificar como (Erleben y col., 2005):

- Basadas en fuerzas.
- Basadas en velocidad.
- Basadas en energía cinética.
- Basadas en la posición (movimiento en el espacio).

En los modelos basados en fuerzas se conoce la configuración del sistema en cada momento, mientras que en los basados en velocidad se observan los cambios el momento lineal y angular en un intervalo de tiempo como impulsos. El problema de los métodos basados en fuerzas es que sufren de problemas de indeterminación y de inconsistencias al tratar las colisiones y la fricción, al haber discontinuidades (Erleben y col., 2005).

Los métodos basados en velocidad y posición, resueltos por métodos tomados de la programación matemática (como LCP y MLCP), aunque resuelven muchos de los problemas de los métodos basados en fuerzas tienen el problema de que no son necesariamente correctos desde un punto de vista físico (Chatterjee, 1999; Deul, Charrier & Bender, 2016), además de poder tener varias soluciones. Aunque los métodos basados en posición parecen estar ganando relevancia (Deul y col., 2016), los métodos basados en velocidad son los más utilizados, siendo la base de ODE y Bullet y a los que se refiere el resto de este apartado.

Las restricciones pueden ser bilaterales o unilaterales según según se formulen con ecuaciones o inecuaciones.

$$C(q_1, q_2, ..., u_1, u_2, ...\dot{u}_1, \dot{u}_2..., t) = 0 \tag{2.18}$$

$$C(q_1, q_2, ..., u_1, u_2, ...\dot{u}_1, \dot{u}_2..., t) \geq 0 \tag{2.19}$$

Siendo q_i las coordenadas que describen en el sistema, u las velocidades y $\dot{u}$ las aceleraciones. Las colisiones y los límites de las articulaciones no son restricciones holonómicas, en cuanto son unilaterales (al evitar la penetración entre dos cuerpos).

Los sistemas de articulaciones se pueden representar con restricciones holonómicas escleronómicas (restricciones bilaterales solo con dependencia de las coordenadas y no de velocidades o aceleraciones). Derivando con respecto al tiempo:

$$\frac{dC}{dt} = \frac{\partial C}{\partial q}\frac{dq}{dt} = Ju \tag{2.20}$$

Siendo J el término denominado Jacobiano que restringe los grados de libertad.

Volviendo a derivar con respecto al tiempo se puede comprobar (ver Erleben y col., 2005) que la fuerza generalizada ejercida por una restricción holonómica es:

$$F = J\lambda \tag{2.21}$$

siendo λ el multiplicador de Lagrange.

En el caso de una restricción unilateral, además se deberá cumplir que el multiplicador de Lagrange λ sea positivo, y que sea nulo si la restricción no está activa.

Utilizando el vector de todas las posiciones generalizadas y todas las restricciones el sistema se puede modelar como el problema del método complementario no lineal, pudiendo ser discretizado al problema del método lineal (LCP y MLCP). No se incluyen aquí los resultados que se pueden encontrar por ejemplo en Erleben y col. (2005).

Una vez obtenidas todas las fuerzas generalizadas, se puede calcular la aceleración del sistema e integrar hasta obtener la posición. Así mismo, se deberán incluir en la formulación métodos para corregir los errores que se hayan acumulado durante la simulación, y que resultan en que las restricciones no se cumplan. En el caso de restricciones basadas en velocidad los errores serán de posición.

Una vez caracterizadas las restricciones, el problema del LCP se puede resolver de una forma iterativa o exacta, siendo la última más lenta y con posibles problemas numéricos, aunque más precisa. El problema del LCP se extiende al problema de MLCP al introducir variables libres, necesarias para resolver conjuntamente restricciones unilaterales y bilaterales. Para encajar el cono de fricción dentro del sistema LCP/MLCP es necesario linealizarlo. En caso contrario nos enfrentamos a un problema NCP (Nonlinear Complementarity Problem), de resolución mucho más difícil.

Algunos métodos habituales de resolver los problemas de LCP/MLCP son los siguientes:

- Dantzig.
- Lemke-Howson.
- Método de Gauss-Seidel.
- Successive over-relaxation (SOR).

Los métodos de Dantzig y Lemke-Howson resuelven el problema de forma exacta, sin embargo, son problemáticos por su complejidad algorítmica ($O(n^3)$) y en la práctica son utilizados los métodos iterativos. Es importante, especialmente para escenas grandes, dividir la escena en grupos de cuerpos que solo tienen restricciones entre ellos y resolverlos independientemente. Estos grupos de cuerpos se suelen denominar islas.

Formulaciones en coordinadas reducidas

Además de las formulaciones basadas en multiplicadores de Lagrange (coordenadas maximalistas), que calculan las fuerzas de las restricciones, el problema de las restricciones holonómicas se puede plantear en coordenadas generalizadas. Estas coordinadas describen unívocamente la configuración del sistema y su base está en la mecánica analítica

Los algoritmos más conocidos son el algoritmo recursivo de Newton-Euler (RNEA, Recursive Newton-Euler Algorithm) para dinámica inversa, el algoritmo compuesto del cuerpo rígido (CRBA, The Composite-Rigid-Body Algorithm) y el algoritmo de Featherstone (ABA, Articulated Body Algorithm) (Featherstone, 2007; Stepie, 2013).

Estos algoritmos tienen ventajas como que suelen ser generalmente más rápidos de calcular y con menores errores numéricos y de deriva (drifting). Como contrapartida, son más complicados y menos generales y muy difíciles de realizar en el caso de bucles cerrados y contactos.

Métodos híbridos

Ya que cada paradigma tiene unas ventajas y desventajas, es posible utilizar varios paradigmas, tratando de obtener lo mejor de cada uno (Mirtich, 1996; Stepie, 2013). En el caso del motor físico Bullet, se puede utilizar en la misma simulación el método basado en restricciones y el método basado en coordenadas reducidas.

2.1.4. Fricción

Además de las fuerzas de contacto, que se corresponden con fuerzas normales al plano de contacto, las otras fuerzas importantes a considerar en las colisiones son las de fricción, tangenciales al plano de contacto.

La fricción es una fuerza compleja para la que se utilizan modelos experimentales. El modelo más habitual es la ley de fricción de Coulomb, en el que la fuerza es independiente de la velocidad entre los cuerpos:

$$F_f \leq \mu F_n \tag{2.22}$$

Siendo F_f la fuerza de fricción, μ el coeficiente de fricción, que depende de los materiales en contacto y F_n la fuerza normal entre las superficies de los objetos. La fuerza de fricción siempre se opone al movimiento que hubiera habido sin fricción entre las superficies. En el caso estático, la fuerza de fricción será necesaria para prevenir el movimiento. En el caso de que haya movimiento, la ecuación será una igualdad. El coeficiente en el caso estático puede ser igual o superior al del caso dinámico, habiendo en esta situación dos coeficientes.

La fuerza de fricción añade muchos problemas a la formulación de las colisiones, y además introduce discontinuidades en la aceleración e indeterminaciones. Unido a leyes de colisiones como la de Newton o Poisson puede resultar en incrementos de energía en la colisión (Mirtich, 1996).

La fuerza de fricción modelada con la ley de Coulomb da lugar a un cono de fricción que representa los valores posibles de fuerza dentro de los cuales un cuerpo permanecerá estático. En el caso dinámico, la fuerza de fricción se situará en la superficie del cono.

La forma más habitual de modelar el cono de fricción en los modelos basados en restricciones es linealizarlo e introducirlo en la formulación del LCP, aunque existen formulaciones alternativas, como de hecho se utiliza en btMultiBody en Bullet, que proyectan la fuerza de fricción contra la superficie del cono.

La fricción por lo tanto será una fuerza no conservativa en el caso dinámico. Sin embargo, en el caso estático existen otras configuraciones (como una esfera que rueda hacia abajo en un plano inclinado), en el que la fricción estática es conservativa, por lo que hay otros tipos de fricciones a considerar, como son la de rodante y la de rotación (rolling friction y spinning friction).

2.1.5. Integradores

Una vez obtenidas las ecuaciones diferenciales ordinarias del movimiento y sus condiciones iniciales es necesario resolverlas de una forma numérica al no haber solución analítica. El método más utilizado el de método de Euler semi-implícito que, aunque es un método de primer orden y por lo tanto con un error elevado, es estable para sistemas conservativos (es un integrador simpléctico) y apropiado para aplicaciones que requieren un rendimiento elevado. Su formulación es simple y no requiere del uso de fuerzas no conocidas en el futuro:

$$v_{n+1} = v_n + a_n \Delta(t) \tag{2.23}$$

$$x_{n+1} = x_n + v_{n+1} \Delta(t) \tag{2.24}$$

Existen otros muchos integradores de órdenes superiores utilizados en la práctica, estando su estudio y el de la convergencia y la estabilidad de los métodos fuera del ámbito de este trabajo. Algunos de ellos, como Runge Kutta o Verlet, se exponen en Eberly (2003). La elección del paso, el análisis

de los errores numéricos de truncado del algoritmo y de la representación de los números es un problema complejo del campo de los métodos numéricos.

En cuanto a las rotaciones, además del método habitual de rotaciones infinitesimales, en el caso de que la velocidad angular sea elevada con respecto al paso, es habitual realizar rotaciones finitas.

2.2. Detección de colisiones

El ámbito de la detección de colisiones excede a la simulación del cuerpo rígido, y junto con las consultas de proximidad es utilizado en multitud de ámbitos como el diseño, la fabricación o la simulación molecular.

A la hora de resolver las ecuaciones del movimiento la geometría del cuerpo no es relevante, siendo condensada en el centro de gravedad y el tensor de inercia. Son las restricciones de no deformación y la imposibilidad de que dos cuerpos ocupen el mismo espacio al mismo tiempo las que hacen necesario determinar el lugar geométrico en que los cuerpos colisionan. Aunque de por sí un problema fundamentalmente geométrico y matemático, debido a que se debe de implementar en un ordenador, se deben tener muy en cuenta los siguientes aspectos:

- Rendimiento. Especialmente en la simulación en tiempo real, en donde puede haber multitud de objetos y además estos ser complejos. Los algoritmos explotarán la coherencia temporal y espacial.
- Robustez numérica. Son muchos los problemas de estabilidad y robustez en la detección de colisiones. Algunos de ellos estarán derivados de la limitación de usar números en punto flotante en vez de números reales, que generan problemas de precisión numérica y mal acondicionamiento. Además, en muchos casos será imposible evitar la penetración entre cuerpos, lo que generará configuraciones que se deberán tratar correctamente.
- Uso de la memoria. Muchos de los simuladores de cuerpos rígidos son ya aplicaciones complejas que en muchos casos limitan la cantidad de memoria disponible para el módulo de detección de colisiones. En algunos casos será necesario un compromiso entre el rendimiento y el uso de la memoria.

El motor físico requerirá más información que la información de si ha habido o no una colisión entre dos cuerpos. Algunos de los datos que puede devolver el módulo de gestión de colisiones son los siguientes:

- El lugar geométrico en el que se ha producido la colisión. Esto podrá ser uno o varios puntos, una o varias curvas o una o varias superficies (parche). Generalmente y por motivos de rendimiento se utiliza solo uno o varios puntos.
- Una o varias direcciones normales a las superficies de contacto de los cuerpos.
- En el caso en que haya interpenetración entre los cuerpos, una estimación del valor de la penetración y la dirección en la que se resuelve esta penetración.
- En algunas de las modalidades de detección de colisiones continua (CCD, Continuous Collision Detection) como Continuous Advancement es necesaria una estimación del momento de la colisión (TOI, Time Of Impact).

En el caso en que la detección de colisiones se ejecute de forma discreta, habrá que tener en cuenta que raramente se detectará la colisión y que de una ejecución a otra se pasará de un estado sin penetración a uno con penetración. En el caso de que los cuerpos vayan a una velocidad relativa elevada con respecto al paso de la simulación, además se puede presentar el problema de "tunneling", en que un cuerpo atraviesa al otro cuerpo y no es detectado por el sistema de colisiones.

Parcialmente se puede solucionar el problema de la penetración utilizando un margen, de modo que cuando dos cuerpos estén a menos de una distancia prefijada se notifique una colisión. La opción más directa es gestionar la penetración, dando al motor físico una medida de la penetración.

La medida de penetración más habitual es la mínima distancia que es necesaria desplazar un objeto para que solo toque al otro objeto, y que utilizando la distancia euclídea corresponde con la distancia del origen al espacio de configuración de obstáculos (CSO, Configuration Space Obstacles). Otras medidas de distancia son posibles (Tóth, Goodman & O'Rourke, 2017).

El módulo de detección de colisiones es una parte crítica de un motor físico, que utilizará una buena parte del tiempo disponible para la simulación.

Sus algoritmos son complicados de implementar y probar y suelen requerir de ajustes manuales para mejorar el rendimiento y la robustez. Se recomiendan como introducción al problema las referencias van den Bergen (2003) y Ericson (2004) y en Tóth y col. (2017) se puede encontrar literatura de especial relevancia. En este capítulo se expondrán las principales técnicas, organizadas según cuales se aplican en los dos submódulos principales que se suelen encontrar en el módulo de detección de colisiones de un motor físico:

- Broadphase. Trata de eliminar de la prueba de colisión (collision culling) los pares de cuerpos que no pueden colisionar para evitar procesamiento innecesario.
- Narrowphase. La detección de colisiones como tal entre un par de cuerpos.

2.2.1. Broadphase

Muchos de los objetos de una escena son complejos, pudiendo estar formados por miles de polígonos. Comprobar las colisiones en todas las parejas de objetos es un algoritmo de orden $O(n^2)$ en el número de objetos, inasumible cuando el número de estos es elevado.

El objetivo de la fase Broadphase es realizar un primer filtrado de los objetos que potencialmente pueden colisionar y no comprobar los que no lo van a hacer. Las dos aproximaciones principales son las siguientes:

- División del espacio. En este caso el espacio se va dividiendo para tratar de obtener una ventaja por la coherencia espacial de los objetos.
- División por el modelo. En este caso los objetos se agrupan en jerarquías. Al igual que en el caso anterior, esta agrupación trata de obtener una ventaja por la coherencia espacial de los objetos.

Para no tener que utilizar todas las primitivas del objeto original es necesario primero utilizar un volumen delimitador (BV, Bounding Volume), que envuelva completamente al objeto inicial. Entre otras propiedades, este volumen deberá ser simple de calcular, ajustar bien al objeto, permitir comparaciones rápidas, ser simple de rotar y transformar y utilizar poca memoria (ver Ericson, 2004, p. 76). Algunos de los volúmenes delimitadores más

utilizados son esferas, cajas rectangulares (box), k-DOPs o envolturas convexas. En el caso de algunos de estos volúmenes como las cajas rectangulares, estas pueden estar alineadas con los ejes principales (AABB, Axis Aligned Bounding Box), o no (OBB, Oriented Bounding Box).

Generalmente la geometría que mejor ajusta a un cuerpo permite ser más selectivo en qué cuerpos pueden potencialmente colisionar, sin embargo, se debe de tener en cuenta la dificultad de construir el BV y de realizar las comprobaciones sobre él.

Se describen a continuación las técnicas de Broadphase, basadas en la coherencia espacial entre los objetos, dentro de sus dos grandes familias: división del espacio y división del modelo.

División del espacio

La división del espacio más simple consiste en subdividirlo en cuadrículas, donde dos objetos podrán colisionar si comparten al menos una cuadrícula (grids). Estas cuadrículas pueden ser uniformes o se pueden estructurar de una forma jerárquica.

Los árboles permiten otras formas de particionar el espacio. En cada nodo el espacio quedará dividido en dos o más subparticiones. Algunas de las variedades son QuadTrees, Octrees, kd-trees y los más flexibles, espacios de partición binarios (BSP, Binary Space Partitioning).

Más simple que las divisiones del espacio en regiones es el método "Sort and Sweep" (también denominado en la literatura "Sweep and Prune") que utiliza una sola dimensión para barrer el espacio y ordenar los objetos en zonas de posible colisión. Además de estos algoritmos, existen muchos otros relacionados con la división del espacio en la literatura como "Loose Quadtrees" o "Cells and Portals" (Ericson, 2004).

División del modelo

En el caso de subdivisión por el modelo, utilizando algún tipo de volumen delimitador, los objetos de van agrupando en jerarquías de volúmenes (BVH, Bounding Volume Hierarchy). En cada nivel de la jerarquía, los objetos que pertenecen a esta delimitación deben estar incluidos totalmente, y dos objetos

que en el mismo nivel de la jerarquía no están en la misma celda no requerirán un test de intersección.

A la hora de construir y utilizar una jerarquía de BV se deberán tomar muchas decisiones, así, se podrá construir con una estrategia insertando dinámicamente, Top-down o Buttom-up, según se vayan subdividiendo grupos de BV o agrupando al contruir el árbol. Igualmente son decisiones importantes el tipo de BV a utilizar o la forma de recorrer dos ramas de un árbol (o de dos árboles) para buscar potenciales colisiones.

Según Ericson (2004, pp. 236-237) las propiedades deseadas para un BVH son las siguientes:

- Los nodos dentro de un subárbol deberían estar cercanos entre sí.
- Cada nodo en la jerarquía debería ocupar el mínimo volumen.
- La suma de los volúmenes de todos los BV debería ser mínima.
- Se debería prestar especial atención a los nodos cerca de la raíz de la jerarquía.
- El volumen de la intersección de nodos hermanos debería ser mínimo.
- La jerarquía debería estar balanceada, tanto con respecto a la estructura de nodos como a su contenido.

Discusión

La creación de estructuras para la fase Broadphase es muy dependiente de la aplicación y de sus características. Las dimensiones y complejidad de los objetos, su distribución en el espacio, su coherencia temporal, los requisitos de rendimiento o la disponibilidad de memoria para esta fase determinarán que estructuras de datos son más apropiadas. Además, algunos de los algoritmos de la fase Broadphase son complejos de implementar y de probar, y en su diseño se toman muchas decisiones casi arbitrarias.

2.2.2. Narrowphase

Una vez filtrados los objetos que pueden tener colisiones potenciales, la fase Narrowphase permite determinar si esta intersección o colisión existe y obtener la información necesaria para el motor físico.

Existen distintas clasificaciones de los algoritmos existentes. En Erleben y col. (2005) se dividen en cuatro tipos:

- Estructuras espaciales y jerarquías de BV. Son las mismas que se han analizado en la fase Broadphase. Son frecuentemente usados en la simulación dinámica para los objetos que son difícilmente divisibles en partes convexas como "sopas de polígonos", especialmente en las partes estáticas de la escena. Para ello se realiza una división hasta las partes constituyentes de los objetos, de forma que se puede analizar la pieza del objeto que ha colisionado.
- Algoritmos basados en características (feature-based). En una malla de polígonos las características son las caras, los lados y los vértices, y el algoritmo realizará operaciones entre características de ambos cuerpos para buscar colisiones.
- Algoritmos basados en símplices (simplex-based). Basadas en la envoltura convexa de un conjunto independiente afín de puntos. Se comentarán brevemente algunos de estos algoritmos.
- Algoritmos basados en volumen. Se basan en la vista volumétrica de los objetos. Un ejemplo es el grid 3D de distancias con signo a un objeto (Distance Maps). Existen multitud de variaciones, muchos de ellos motivados por la capacidad computacional de las actuales GPUs.

Por sus propiedades singulares, los cuerpos convexos ofrecen numerosas ventajas para la detección de colisiones. Se recuerda que un cuerpo convexo es aquel que, dados dos puntos cualesquiera del cuerpo, contiene todos los puntos del segmento que los une. La mínima distancia entre dos cuerpos convexos es un mínimo tanto local como global, aspecto que no comparten los cuerpos cóncavos. Además, existen algoritmos que aprovechan la coherencia temporal, haciendo uso de testigos (witnesses), que permiten algoritmos que en ciertos casos se aproximan a una complejidad $O(1)$.

Para ciertas aplicaciones en que la restricción de cuerpo convexo no sea apropiada, se puede optar por dividir el cuerpo cóncavo en partes convexas.

La descomposición de un cuerpo cóncavo en varios convexos se ha demostrado ser un problema NP-duro. En la práctica, para la mayoría de los casos se puede conseguir una descomposición aproximada (ACD, Approximate Convex Decomposition). Un ejemplo de librería que implementa un algoritmo ACD es V-HACD (https://github.com/kmammou/v-hacd). Otra alternativa es obtener la envoltura convexa de un cuerpo cóncavo, para lo que se pueden emplear algoritmos como QuickHull (Barber, Dobkin & Huhdanpaa, 1996), empleado en la librería QHull (http://www.qhull.org/).

Por los motivos expuestos, en lo que resta de esta sección se tratará exclusivamente algoritmos que aplican a cuerpos convexos. En el caso de poliedros convexos (también llamados politopos, aunque la nomenclatura no es clara, ya que a veces politopo es simplemente definido como la extensión de polígono a dimensiones superiores y no necesariamente convexo), uno de los algoritmos más simples está basado en SAT (Separating Axes Theorem). Este teorema indica que si existe un hiperplano que separa dos politopos convexos, estos no colisionan. Para probar dos poliedros convexos es suficiente con probar todas las caras de ambos poliedros y todos los productos vectoriales entre los lados de ambos poliedros.

Siguiendo a Tóth y col. (2017, p. 1032), los algoritmos se pueden clasificar en algoritmos basados en programación lineal, basados en sumas de Minkowski y optimización convexa y basados en características utilizando localidad geométrica.

Algoritmos basados programación lineal

Basándose en el teorema SAT, se plantea la búsqueda del plano de separación como un problema de programación lineal. Utilizando un método iterativo con testigos se pueden conseguir un rendimiento cercano a tiempo constante (Tóth y col., 2017).

Algoritmos basados en espacios de Minkowski y optimización convexa

La suma de Minkowski de dos conjuntos A y B se define como el conjunto:

$$A \oplus B = \{\boldsymbol{a} + \boldsymbol{b} \,|\, \boldsymbol{a} \in A, \boldsymbol{b} \in B\} \tag{2.25}$$

Utilizando la suma de Minkowski, la operación $A \oplus (-B)$, también llamado CSO (Configuration Space Obstacles, C-Space Obstacles o TCSO, Translational C-Space Obstacles), tiene propiedades muy útiles:

- Si el CSO de dos cuerpos incluye el origen, los cuerpos tienen interpenetración, y la penetración corresponderá a la menor distancia entre el origen y la frontera del CSO.
- Si el CSO no incluye el origen, la menor distancia entre el origen y el CSO de dos cuerpos es igual a la mínima distancia entre dos objetos.
- El CSO de dos cuerpos convexos es convexo.

Utilizando directamente estas propiedades y calculando el CSO se puede determinar si dos cuerpos convexos colisionan o una distancia entre ellos. Sin embargo, el CSO de un poliedro convexo requiere de $O(n^2)$ vértices, por lo que no es práctico.

Se llama función de soporte (support mapping) a la función que, dada un conjunto convexo y una dirección, devuelve el punto de soporte, es decir, el punto más alejado del conjunto convexo en la dirección dada.

El algoritmo GJK (Gilbert–Johnson–Keerthi), descrito en profundidad en van den Bergen (2003), es un algoritmo iterativo que va recorriendo símplices del CSO, seleccionándolos mediante una función de soporte por lo que no es necesario construir el CSO inicialmente. La salida del algoritmo GJK será penetración si el símplice contiene el origen o una estimación de la distancia entre los objetos. Inicialmente fue propuesto para politopos convexos, habiendo extensiones para cuerpos representados por superficies implícitas y variaciones que toman ventaja de la coherencia temporal utilizando vértices de anteriores ejecuciones.

El algoritmo GJK no da medida de la penetración entre los objetos. Junto con GJK se suele utilizar el algoritmo EPA (Expanding Polytope Algorithm). El algoritmo EPA se puede inicializar con el símplice que devuelve GJK y que contiene el origen en el CSO. De forma iterativa, el símplice se va aumentando hasta encontrar la menor distancia del CSO al origen, dando una medida de la penetración. El algoritmo se encuentra detallado en van den Bergen (2003).

XenoCollide, variante de MPR (Minkowski Portal Refinement), descrito en Jacobs (2008), es otro algoritmo basado en CSO. Basado también en el

uso de funciones de soporte, es un algoritmo iterativo que proyecta rayos hacia la superficie del CSO. Aunque el algoritmo MPR no da una medida de separación sí indica la colisión/penetración y puede dar una medida de ella.

Aunque no está basado directamente en el CSO, otro algoritmo rápido y simple de implementar utilizando vértices para buscar planos de separación entre dos cuerpos es CW (Chung-Wang algorithm) (Ericson, 2004).

Algoritmos basados en características

Un poliedro en tres dimensiones tiene tres tipos de características, caras, lados y vértices. Utilizando las regiones de Voronoi de las características de un poliedro convexo externas al polígono, Lin y Canny propusieron un algoritmo, LC, que recorría las características hasta encontrar un mínimo (Lin, 1994). La base es que la distancia mínima entre dos poliedros convexos que no interpenetran se da en las características de cada poliedro que están contenidas en la región de Voronoi del otro poliedro.

Mirtich mejoró el algoritmo LC, proponiendo V-Clip (Mirtich, 1998), que se explicará en un capítulo posterior al ser implementado en este proyecto.

Existen muchas otras variaciones, por ejemplo, basadas en niveles de detalle o en jerarquías de Dobkin y Kirkpatrick (Dobkin & Kirkpatrick, 1990), que aunque requieren más memoria y preprocesamiento permiten mejorar el tiempo de procesamiento. Algunas de estas técnicas también se han propuesto para algoritmos como GJK.

2.2.3. Otros temas de detección de colisiones

Determinación de contactos

Un motor físico requiere de información de la colisión detectada para la simulación. Tanto por las características de los cuerpos simulados, muchas veces poliedros, como por la respuesta dada por el algoritmo, la precisión numérica y la posibilidad de interpenetración, es necesario un tratamiento, que en la literatura se denomina determinación de contactos.

Una aproximación habitual consiste en disponer de una zona de seguridad

de la colisión (collision envelope), de forma que se reportarán todos los puntos de contacto de esta zona, habitualmente representada por dos planos.

En Erleben y col. (2005, pp. 445-450) se presentan dos algoritmos, uno geométrico para la búsqueda de contactos dentro de la zona de seguridad de la colisión y otro que realiza "backtracking" en el módulo de simulación cuando se encuentra una colisión.

Continuous Collision Detection

Los algoritmos tratados hasta ahora resuelven el problema de la detección de colisiones en momentos discretos de tiempo. Como resultado de esto puede que no se reporten colisiones válidas o que se dé lo que se denomina efecto túnel (tunnelling), en que un cuerpo atraviesa a otro sin ser detectado.

Los algoritmos de detección de colisiones continua (CCD, Continuous Collision Detection) interpolan dos configuraciones en dos puntos de la simulación y comprueban las colisiones en esa trayectoria. Si la colisión existe podrá ser necesario informar del momento del impacto (TOI).

Al incluir la variable tiempo, el problema de CCD es un problema de hallar raíces en un espacio de cuatro dimensiones, que en algunos casos se trata sobre la proyección en las tres dimensiones del espacio. Los algoritmos se clasifican en métodos algebraicos, volúmenes barridos (swept-volume), métodos de bisección, estructuras de datos cinemáticas (KDS, Kinetic Data Structures), formulaciones basadas en sumas de Minkowski y avance conservativo (CA) (Coumans, 2005; Tóth y col., 2017).

Uno de los mayores problemas de los métodos CCD es su mayor tiempo de computación, por lo que muchas de estas técnicas no son apropiadas en simulación interactiva o en tiempo real.

Representaciones de cuerpos

Además de representaciones basadas en poliedros y mallas de polígonos, existen otras formas de representar o combinar cuerpos rígidos, como superficies implícitas, geometría sólida computacional (CSG, Computational Solid Geometry), representaciones de frontera (B-rep, Boundary Representation) o NURBS (Non-Uniform Rational Basis Spline). Algunas de estas represen-

taciones tienen ventajas para cierto tipo de simulación, sin embargo, en la simulación el cuerpo rígido las más habituales son los poliedros, mallas de polígonos y ciertas superficies implícitas como esferas y cilindros con tapa esférica (cápsulas o píldoras).

2.3. Arquitectura

Un simulador del cuerpo rígido está compuesto por multitud de módulos, cada uno encargado de una tarea específica. A muy alto nivel y de forma genérica un simulador se separa en dos componentes principales, detección de colisiones y simulación.

Erbelen propone un diseño modular, explicado en Erleben (2004), que trata de ser aplicable a todos los paradigmas de simulación, a las distintas variaciones entre detección de colisiones discretas o continuas (CCD) e incluso a algoritmos más elaborados de sincronización de colisiones como el algoritmo Timewarp de Mirtich (Mirtich, 2000).

El componente de simulación se divide en los siguientes módulos (Erleben, 2004, pp. 10-20) (ver Figura 2.1):

- Resolutor de colisiones (Collision Solver). Llamado desde el módulo resolutor de restricciones, calcula y aplica los impulsos necesarios a los cuerpos que intervienen en una colisión.
- Control del tiempo (Time Control). Parte central del simulador. Ejecuta la simulación desde un instante inicial a uno final y devuelve un estado de configuración. Puede utilizar un paso fijo de integración, uno variable, no permitir penetración (CA, Conservative Advancement) o utilizar algoritmos de backtracking en caso de penetración para volver a un estado correcto de la simulación.
- Resolutor del movimiento (Motion Solver). Responsable de la integración de los objetos de acuerdo con las ecuaciones del movimiento.
- Resolutor de restricciones (Constraing Solver). Este módulo calcula las fuerzas necesarias para que las restricciones deseadas se mantengan. Estas normalmente se dividen en restricciones unilaterales y bilaterales. Entre sus responsabilidades está la de invocar el módulo resolutor de colisiones.

El componente de detección de colisiones se divide en los siguientes módulos:

- Fase Broadphase.
- Fase Narrowphase.
- Determinación de contactos (Contact Determination). Con la información de la fase Narrowphase se determinarán las regiones de contacto entre objetos que tocan o que penetran. Este problema no es trivial y la representación no siempre es única. Dependiendo de las necesidades de rendimiento de la aplicación se puede optar por dar un solo punto de contacto junto con la normal a las superficies o la intersección completa de los objetos.
- Análisis de coherencia temporal (STC, Spacial-Temporal Coherente, analysis). Este módulo analiza la configuración, detecta contactos independientes (islas o grupos) y utiliza cachés para mejorar la eficiencia.

Para más información se recomienda Erleben, 2004, Capítulo 2, donde se explican de forma detallada las interrelaciones entre los módulos y su aplicación a distintos paradigmas de simulación.

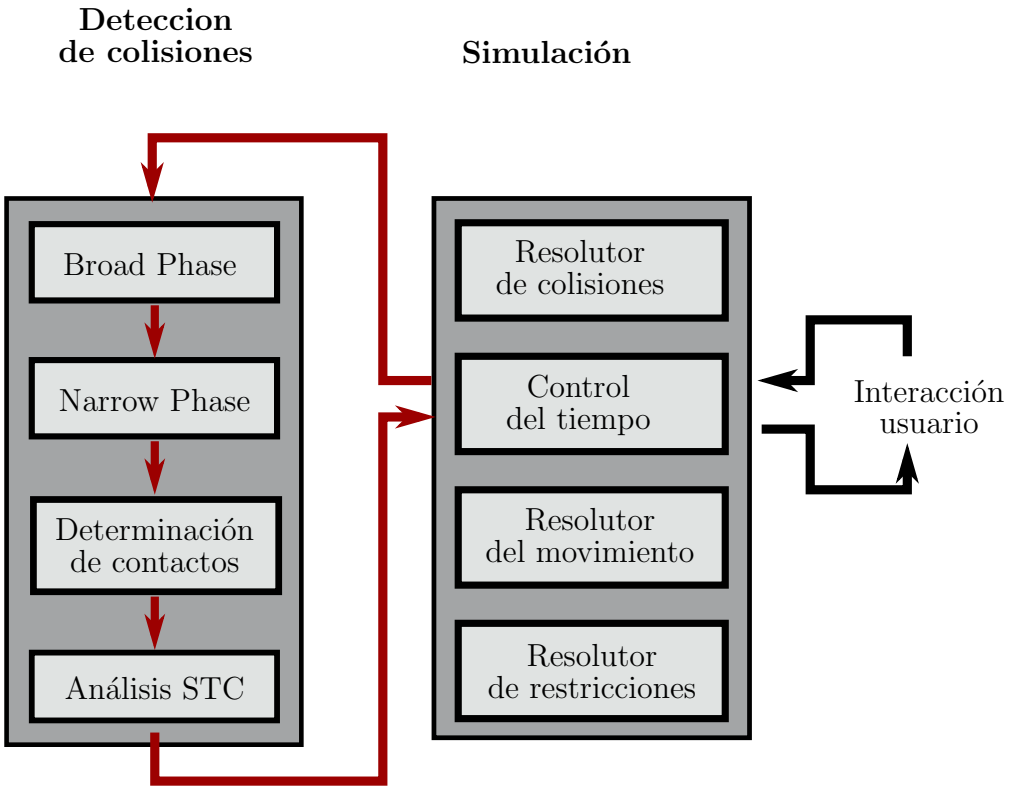


Figura 2.1: Arquitectura universal de un simulador, según se describe en Erleben (2004) y la versión simplificada en Stepie (2013). Adaptado de Stepie (2013).

Capítulo 3

ODE

3.1. Introducción

Figura 3.1: Logo de ODE. Obtenido de https://bitbucket.org/odedevs/ode

En este capítulo se describe y analiza el motor físico ODE. Para ello se utilizarán todas las referencias encontradas y el propio código fuente del proyecto. De este análisis se han seleccionado algunas de sus características más relevante y se han sintetizado en las siguientes tablas:

- Aspectos básicos, Tabla 3.4.
- Funcionalidades y arquitectura, Tabla 3.1.
- Principales algoritmos de detección de colisiones, Tabla 3.2
- Algoritmos de simulación, Tabla 3.3.

3.1.1. Descripción general

Open Dynamics Engine, ODE (http://www.ode.org/), es un motor físico para la simulación de cuerpos rígidos. La versión inicial data del año 2001, y su desarrollador principal es Russel Smith, siendo por lo tanto uno de los primeros simuladores de cuerpo rígido generalistas con el código fuente disponible.

Se distribuye bajo dos licencias, GNU LPGL y BSD, lo que permite su uso y modificación en aplicaciones comerciales. Estando enfocado a la simulación interactiva y es y ha sido utilizado por multitud de productos, tanto en el ámbito de la simulación y robótica como en videojuegos. ODE está desarrollado en C++ con una interfaz en C, y se incluyen instrucciones para compilarlo en sistemas basados en Linux, Windows y Mac.

ODE incluye tanto un módulo de detección de colisiones como uno de simulación basado en restricciones de velocidad. Además, se incluyen en el código fuente las siguientes librerías opcionales:

- GIMPACT. Librería para detección de colisiones, que implementa algoritmos de jerarquías de cajas rectangulares (el BV es una caja y es AABB) y mallas de triángulos (trimeshes).
- libccd. Librería para detección de colisiones que implementa los algoritmos GJK, EPA y MPR.
- OPCODE. Librería de árboles AABB.
- drawstuff. Pequeña librería gráfica para visualizar los ejemplos (ver Figura 3.2).
- Demos, otras contribuciones y APIs para utilizarlo con otros lenguajes de programación.

Aunque ODE sigue siendo muy utilizado, no se desarrolla de forma activa y en los últimos años no se ha añadido ninguna funcionalidad significativa.

3.1.2. Ámbito de aplicación y principales usos

En el manual (Russel Smith, s.f.), ODE se presenta como bueno para la simulación de rígidos articulados y estructuras, diseñado para ser utilizado en

simulaciones interactivas y de tiempo real, en donde se prioriza el rendimiento y la estabilidad frente a la precisión física.

Dentro de la wiki de ODE, en Russell Smith (s.f.-b), se encuentra un listado de aplicaciones que lo utilizan. Destacan los simuladores de robótica entre los que se encuentran muchos de los más referenciados en la literatura, como Gazebo, Webots y CoppeliaSim. También hay un catálogo de juegos, algunos de ellos de famosos como "Call of Juarez", aunque en el listado no hay juegos relevantes recientes.

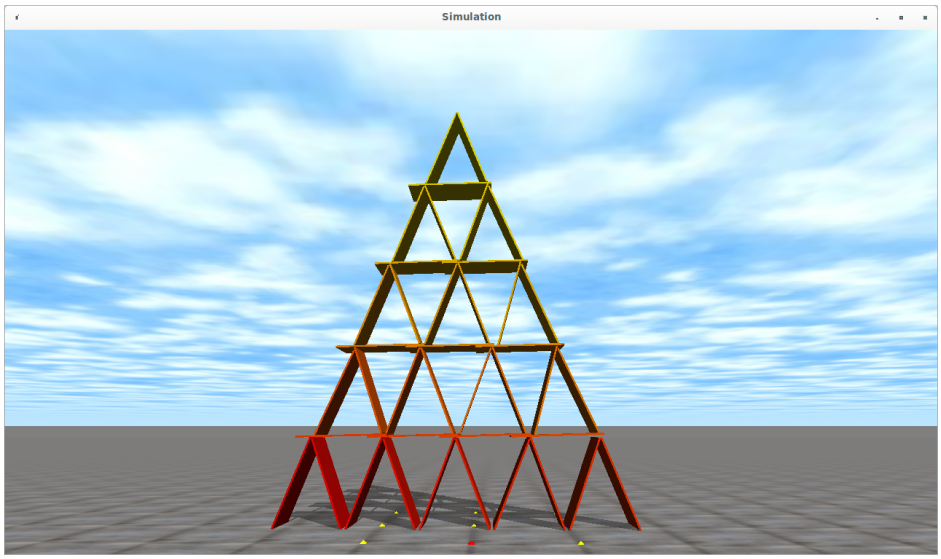


Figura 3.2: Demo incluida con ODE. ode_cards

3.1.3. Relación con otras herramientas

No hay herramientas de importación de sólidos, mayas de polígonos o cuerpos articulados desde formatos de fichero aceptados por herramientas como Blender o Autodesk 3ds Max. Sin embargo, algunas de las aplicaciones y simuladores que utilizan ODE disponen de estas integraciones. ODE dispone de alguna facilidad para calcular tensores de inercia y centros de gravedad, aunque la funcionalidad es limitada. Por ejemplo, es capaz de calcular el tensor de inercia de una malla de triángulos pero no de un poliedro convexo.

Motores de juegos como Irrlicth y motores físicos como Ogre3d disponen de plugins para facilitar el uso de ODE. La herramienta PAL (Boeing, s.f.), que define un API común a varios motores físicos está obsoleta y el autor no

ha sido capaz de utilizarla. Existen bindings con Python y .net, sin embargo, parece que no están activamente mantenidos.

3.2. Diseño y arquitectura

ODE está escrito en C++, utilizando una mezcla de orientación a objetos y programación estructurada. La interfaz principal para utilizar ODE está exclusivamente basada en C (aunque también existe una interfaz no mantenida en C++), prescindiendo de la orientación a objetos, y que expone funciones a las que se les pasan identificadores de objetos, que corresponden a punteros a las clases internas. En este apartado se hará referencia a estas clases internas, aunque no se utilizan directamente en la interfaz. Así `dBodyID`, el identificador de un cuerpo, está definido como "`typedef struct dxBody *dBodyID;`".

La delimitación entre el sistema de detección de colisiones y la simulación es muy clara, siendo las clases `dxGeom` y `dxWorld` la base de esta separación, pudiéndose reemplazar el módulo de detección de colisiones con facilidad (Russel Smith, s.f.).

Para inicializar ODE es necesario llamar a funciones globales, que por lo tanto no permiten crear varias instancias de ODE. Hay varias estructuras que se compartirán entre todos los escenarios de simulación creados con ODE, como por ejemplo las funciones escogidas para la detección de colisiones. Esto supone una dificultad al tratar de utilizar dos instancias de ODE en una aplicación, como de hecho ocurre en este trabajo cuando se desea utilizar simultáneamente dos simulaciones con distintas funciones de gestión de colisiones.

Una vez creado el mundo con los objetos `dxWorld` y las estructuras de colisiones `dxGeom` necesarias, con al menos un espacio `dxSpace`, se puede iniciar la simulación. La simulación consiste en básicamente un bucle con primero la detección de colisiones y después el paso de simulación, que se recomienda de tipo tiempo fijo. En el Listado 3.1 se muestran los pasos fundamentales de la simulación. Primero se llama al módulo de detección de colisiones con la función `dSpaceCollide`, que creará contactos dentro del grupo de contactos `contactgroup`. Con los contactos resueltos, se ejecutará la simulación con el algoritmo iterativo `dWorldQuickStep` o con el fijo `dWorldStep`. Por último y antes de volver a ejecutar el bucle será necesario borrar los contactos creados.

Tabla 3.1: ODE. Funcionalidades y arquitectura

Característica	Soporte	Detalles
Paradigmas simulación	• Restricciones basadas en velocidad.	
Detección de colisiones independiente de simulación	Sí	Ambos módulos son totalmente independientes.
Facilidad cambio sistema detección de colisiones	Alta	En el manual (Russel Smith, s.f.) están las instrucciones para utilizar otra librería de colisiones.
Espacios de colisiones recursivos	Sí	
Varias instancias del motor independientes	Parcial	ODE hace un uso elevado de variables globales. Aunque varias escenas se pueden ejecutar en el mismo proceso, hay parámetros que no se pueden variar por escena.
Posibilidad de geometrías compartidas por los cuerpos	No	

Listado 3.1: Pasos simulación ODE

```
dSpaceCollide(space, 0, &nearCallback);
dWorldQuickStep(world, 0.02) o dWorldStep(world, 0.02);
dJointGroupEmpty(contactgroup);
```

3.2.1. Gestión de colisiones

Los elementos de tipo `dxGeom` son las estructuras para la gestión de colisiones. Pueden ser de tipo espacio, que contiene otros `dxGeom` y sirve para facilitar la fase Broadphase o como forma del objeto final. Los espacios se pueden estructurar recursivamente y es necesario al menos uno para iniciar

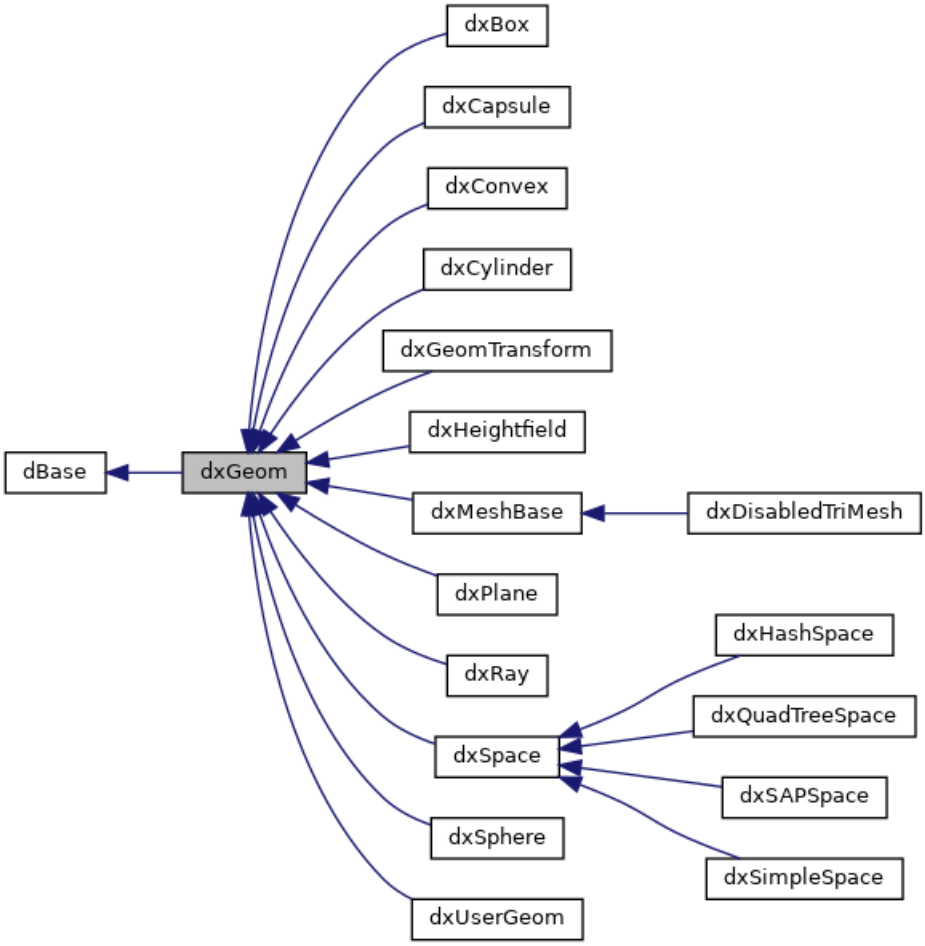


Figura 3.3: Jerarquía de clases dxGeom

la detección de colisiones.

Todos los tipos de `dxGeom` menos los espacios y los planos (`dxPlane`) se pueden asociar a un cuerpo rígido (`dxBody`), y se puede modificar su orientación y su posición (en terminología de ODE son "placeable"). Ejemplos de "placeable" `dxGeom` son cuerpos convexos `dConvexClass` y mallas de polígonos `dTriMeshClass` (ver Figura 3.3 para todos los tipos de geometrías disponibles).

El `dxGeom` "placeable" `dxRay` es un caso especial, que representa un rayo que no tiene una forma asociada, útil para cierto tipo de aplicaciones. Los "placeable" `dxGeom` se pueden asociar a cuerpos rígidos (`dBody`) o pueden representar partes estáticas del escenario si no se asocian a cuerpos rígidos.

El punto de entrada a la detección de colisiones es la función `dSpaceCollide`, a la que se pasará un espacio y una función callback. Es en esta función creada por el usuario donde se gestionará cómo se realiza la detección de colisiones entre espacios para la fase Broadphase y para cuerpos directamente para la fase Narrowphase. En esta función se generarán los contactos, la estructura `dContact` para el contacto `dxJointContact`. Además de la parte geométrica de la colisión, la estructura `dContactGeom`, es necesario informar de las propiedades de la superficie en la estructura `dSurfaceParameters`. Aspectos como fricción, coeficiente de restitución, ERP y CFM se definen en esta estructura. ERP y CFM se explican en la Sección 3.3.2.

Un cuerpo rígido podrá estar asociado a varios "placeable" `dxGeom`, que podrán estar desplazados con respecto a la posición inicial del cuerpo para construir cuerpos más complejos. No hay una estructura "placeable" `dxGeom` que pueda agrupar varios y crear grupos de transformación.

3.2.2. Simulación

Una vez inicializado ODE, se pueden crear varios mundos, que funcionan de forma independiente (`dxWorld`).

En un `dxWorld` los objetos principales que existen son los siguientes:

- `dxWorld`.
- `dxBody`.
- `dxJoint`.
- `dxJointGroup`.

El objecto central de ODE es `dxBody`, un cuerpo rígido con sus propiedades de centro de gravedad, tensor de inercia, `dxGeom` asociados, posición, orientación, velocidad lineal y velocidad angular y sobre el que se aplicarán las fuerzas y pares oportunos.

Dos `dxBody` se podrán asociar con una restricción `dxJoint` o un un solo `dxBody` y el espacio estático. Cada tipo de `dxJoint` establecerá unas restricciones, siendo `dxJointContact` un tipo especial que representa un contacto por colisión. Para facilitar el borrado de grupos de contactos se pueden agrupar

en la estructura `dxJointGroup` (ver Figura 3.4 para todos los tipos de restricciones existentes en ODE).

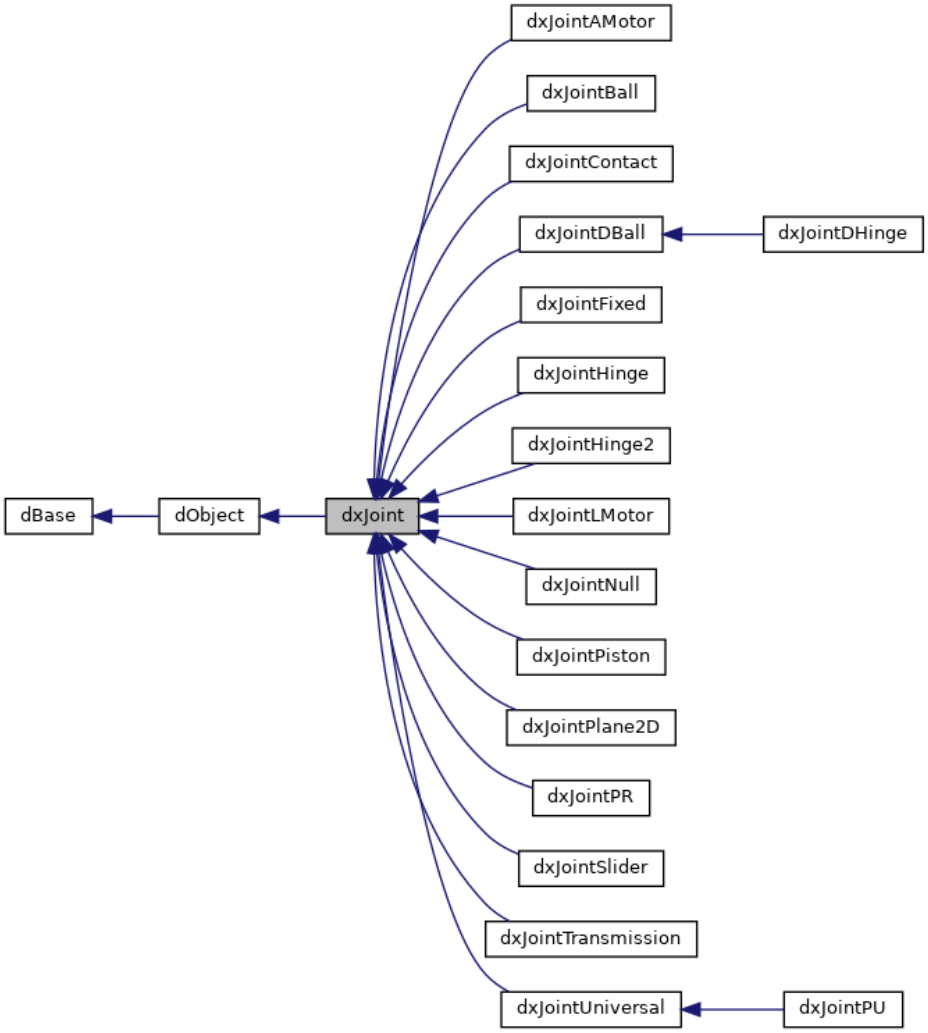


Figura 3.4: Jerarquía de clases dxJoint

Una vez obtenidas las colisiones, se procederá a la simulación con una de las siguientes funciones:

- `dWorldStep`. Método no iterativo basado en pivots utilizando el algoritmo Dantzig. Más exacto, más lento y con problemas si la matriz está mal acondicionada.
- `dWorldQuickStep`. Método iterativo para resolver el LCP basado en SOR (Successive Over Relaxation).

Aunque los dos procedimientos son bastante diferentes, en resumen, realizan las siguientes operaciones:

- Añaden la fuerza de gravedad a los cuerpos.
- Calculan las matrices de inercia y sus inversas en coordinadas globales.
- Se calculan las fuerzas giroscópicas (ver Lacoursière, 2006).
- Identifica islas, es decir, agrupaciones de cuerpos con restricciones `dxJoint` entre ellos. Esto determinará el tamaño de la matriz jacobiana. Cada isla se resolverá independientemente de las otras, e incluso el código está preparado para hacerlo en paralelo utilizando varios hilos.
- Una vez obtenidas las fuerzas y pares de las restricciones se integran de forma explícita las velocidades angulares y lineales.
- Se integran de forma implícita las posiciones y orientaciones. Las orientaciones se pueden integrar de forma infinitesimal o finita.
- Se borran los acumuladores de fuerzas y pares.
- Por último, es responsabilidad del usuario borrar los contactos. Con esto se termina el paso de la simulación.

3.3. Algoritmos

3.3.1. Detección de colisiones

ODE no realiza ningún tipo de CCD (Continuous Collision Detection) ni de predicción especulativa de contactos. La detección de colisiones es exclusivamente discreta.

Para la fase Broadphase hay cuatro algoritmos disponibles, que se pueden además utilizar de forma jerárquica entre ellos:

- Fuerza Bruta. Comprueba los AABB de todos los cuerpos contra todos los cuerpos. Clase `dxSimpleSpace`.

Tabla 3.2: ODE. Principales algoritmos de detección de colisiones

Tipo	Algoritmos	Detalles
Broadphase por división del espacio	Grid jerárquico QuadTree Sweep and Prune	
Broadphase por división del modelo	-	
Convexo-Convexo	• SAT • MPR	Para MPR es necesario utilizar la librería libccd. SAT es para poliedros, no es posible aplicarlo en cuerpos definidos implícitamente.
Trimesh-Trimesh Trimesh-Otros	• Top-Down AABB tree • Box-Pruning	Es necesario utilizar las librerías OPCODE y GIMPACT.
CCD	-	

- Grid jerárquico. Consiste en una tabla hash, en el que se pueden definir los tamaños mínimos y máximos de las celdas como potencias de 2, es decir, los valores de i en 2^i. Clase `dHashSpaceClass`.
- Árbol QuadTree en la que se definen el tamaño de los bloques. Clase `dxQuadTreeSpace`.
- Sweep and Prune sobre los ejes principales del espacio. Para ordenar se utiliza el algoritmo el algoritmo Radix Sort, por lo que no se utiliza la coherencia temporal del algoritmo original, en que se utilizaba el algoritmo Insertion Sort que permite ordenar en tiempo $O(n)$ conjuntos casi ordenados. El autor indica en el código que aun así gestiona eficazmente el movimiento. Clase `dxSAPSpace`.

Todos los algoritmos utilizados en ODE se basan en división del espacio y utilizan BV alineados con los ejes principales (AABB). Cada "placeable" `dxGeom` es encargado de generar un AABB, de modo que si es ineficiente al generarlo o no lo genera la fase Broadphase perderá su utilidad. Como curiosidad, para los cuerpos de tipo `dxConvex`, cada vez que es necesario generar un AABB la implementación actual recorre todos los vértices de la geometría, lo que es bastante ineficiente para cuerpos convexos complejos. En cambio, para las mallas de polígonos `dxMeshBase`, utilizadas en las librerías con las li-

brerías OPCODE y GIMPACT, se calcula a partir de un OBB bien orientado proyectándolo sobre los ejes de coordenadas.

Según se puede observar en la Figura 3.3, ODE tiene los siguientes tipos de geometrías básicas:

- Rayos (`dxRay`).
- Planos (`dxPlane`).
- Esferas (`dxSphere`).
- Cajas (`dxBox`).
- Cápsulas (`dxCapsula`).
- Cilindros (`dxCylinder`).
- Trimesh (`dxBaseTrimesh`).
- Objetos convexos (`dxConvex`).
- HeightField (`dxHeightField`).

Para realizar la detección de colisiones en la fase Narrowphase, ODE dispone de muchas funciones especializadas entre los objetos. En el manual (Russel Smith, s.f.) se pueden ver las combinaciones posibles de objetos para realizar la detección de colisiones, pudiéndose en algunos casos optar por las librerías externas OPCODE, GIMPACT y libccd. Se comentan aquí solo los algoritmos más relevantes.

Dentro del código de ODE, por defecto, para las colisiones entre poliedros convexos se utiliza directamente el algoritmo SAT, probando como planos separadores todas las caras y todos los productos vectoriales entre los lados de los poliedros convexos. En el caso de que se utilice la librería libccd, se hará uso de una implementación del algoritmo MPR, que no utiliza coherencia temporal. Además de entre poliedros convexos, gracias a libccd también se podrán detectar colisiones entre superficies implícitas convexas, como cilindros, cápsulas o esferas.

En el caso de mallas poligonales se puede utilizar la librería OPCODE o la libraría GIMPACT. OPCODE utiliza árboles AABB binarios, que se construyen de forma top-down, buscando ejes con la mayor varianza. Para

la detección de colisiones utiliza la coherencia temporal. Según el autor de OPCODE (Terdiman, s.f.-a), la implementación es más rápida que la librería SOLID con un uso menor de memoria. En el caso de la librería GIMPACT no se utilizan estructuras jerárquicas, y se utiliza el algoritmo Box-Pruning, variación de Sweep and Prune sobre todos los ejes. Las últimas versiones de GIMPACT utilizan árboles AABB, pero no están incluidas en el código de ODE.

En el manual se indica que no existe un algoritmo que sea capaz de detectar colisiones entre objetos convexos y Trimesh. En el código fuente de ODE se puede comprobar que esta combinación sí está gestionada. En vez de utilizar el objeto convexo utiliza su AABB.

3.3.2. Simulación física y restricciones

Tabla 3.3: ODE. Algoritmos de simulación

Tipo	Soporte	Detalles
Fricción	• Box-Friction • Pirámide linealizada	Box-Friction es más rápido y menos realista físicamente al no hace uso de la fuerza normal. También se llama "constant-force-limit approximation".
Algoritmo Integración	Euler semi-implícito	El paso de integración es fijo.
Algoritmos LCP/MLCP	• Dantzig • Succesive Over Relaxation	El algoritmo Dantzig en una variación del descrito por Baraff.
Término giroscópico	• Implícito en coordenadas globales.	Basado en Lacoursière (2006). En Joseph (2013) se explican las peculiaridades de la implementación en ODE.
Tratamiento errores en restricciones	• ERP y CFM	
Uso CCD	-	

ODE utiliza el paradigma basado en restricciones a nivel de velocidad, y su base teórica se encuentra en los artículos Anitescu y Potra (1997) y Stewart y Trinkle, 1996. En Stewart y Trinkle (1996) se utiliza una formulación para

la resolución de la fricción de Coulomb linealizando el cono de fricción, pero basándose en una formulación basada en restricciones a nivel de posición. No es la que utiliza ODE, que lo realiza basándose en la formulación de Anitescu y Potra (1997), pero con modificaciones para evitar matrices no simétricas e indefinidas (Thulesen, 2015). Por lo tanto, ODE no utiliza realmente el cono de fricción linealizado, sino una variación para mejorar la estabilidad y el rendimiento.

La otra formulación de la que dispone ODE para la fricción es la denominada caja de fricción (box-friction), en la que la fuerza de fricción es independiente de la fuerza normal y por lo tanto poco realista.

La fricción se aplica sobre dos direcciones definidas en la estructura `dSurfaceParameters`. Además, ODE dispone de la opción de aplicar fricción de rotación y rodante (rolling friction y spinning friction) y fricción dependiente del deslizamiento (FDS, Force-Dependent-Slip). ODE no dispone de opciones para modelar la fricción en las articulaciones. Como opción para la restitución, se utiliza un coeficiente de restitución de 0 a 1.

Para la resolución del problema del LCP/MLCP, ODE dispone de dos algoritmos. El primero de ellos está basado en el método de pivots de Dantzing modificado para poder tratar con la fricción, en el que las restricciones holonómicas y las colisiones se gestionan de forma simultánea. Estos algoritmos se aplican sobre las islas de cuerpos que no están conectados por restricciones. El segundo está basado en el algoritmo Succesive Over Relaxation, método iterativo que aunque no da un resultado exacto permite soslayar problemas numéricos y de acondicionamiento.

En cuanto al algoritmo de integración, se utiliza un integrador de primer orden basado en el método de Euler semi-implícito. Para la integración de las rotaciones se utilizan cuaternios, y exista la opción de utilizar rotaciones finitas o infinitesimales. En el caso de rotaciones finitas, se puede forzar un eje de giro. En cuanto a las fuerzas giroscópicas, su integración está basado en Lacoursière (2006). Una descripción aplicada a ODE se puede encontrar en Joseph (2013).

Para corregir los errores de las restricciones ODE añade a las ecuaciones de las restricciones dos parámetros, ERP y CFM. Estos son un caso particular de estabilización de Baumgarte. El primero, ERP, Error Reduction Parameter, determina la proporción del error que se resolverá durante el siguiente paso de la simulación. El segundo, CFM, Constraint Force Mixing, mezcla el resultado

de la fuerza de restricción con la restricción que lo produce, dando lugar a cierta "suavidad" en la restricción, permitiéndose su violación. Es decir, siendo $\boldsymbol{b}$ el error a corregir, las ecuaciones de la restricción resultan:

$$\boldsymbol{Ju} = \boldsymbol{b}K_{erp} - K_{cmf}\boldsymbol{\lambda} \tag{3.1}$$

$$\boldsymbol{F} = \boldsymbol{J^T}\boldsymbol{\lambda} \tag{3.2}$$

Combinando ambos parámetros, una restricción se comporta como un sistema de muelle y amortiguador. Se pueden encontrar información más detallada del funcionamiento de los parámetros CFM y ERP en el manual de ODE, Russel Smith (s.f.), y en Erleben y col. (2005).

Además de lo comentado, son muchas más las opciones que permiten modificar los algoritmos de ODE, como habilitar o deshabilitar cuerpos o amortiguar movimientos. Muchas de estas opciones están deficientemente documentadas, y es llamativo que se pueden encontrar en documentación de simuladores de robots o de formatos de ficheros, como por ejemplo en Foundation (s.f.).

3.4. Aspectos prácticos y resumen

La principal fuente de documentación de ODE es el manual de usuario, que describe la mayor parte de las funcionalidades de ODE. Además, tanto en la wiki de ODE como junto al código fuente hay varios ejemplos reales de su uso. En cuanto a documentación de diseño y algoritmos de ODE, es prácticamente inexistente y queda casi restringida al foro de ODE (Russell Smith, s.f.-a).

El proyecto ya no se desarrolla activamente, aunque se sigue manteniendo.

Tabla 3.4: ODE. Aspectos básicos

Nombre	**Soporte**	**Detalles**
Licencia	BSD y LGPL	Es posible utilizar cualquiera de las dos.
Estado del proyecto	Maduro	
Evolución del proyecto	Baja	
Calidad documentación	Media	
Principales usos	• Videojuegos • Robótica	
Plataformas soportadas	• Linux • Windows • macOS	Además, algunos de los productos que utilizan ODE parecen haberlo portado a otras plataformas como Xbox 360 o iPhone.

Capítulo 4

Bullet

4.1. Introducción

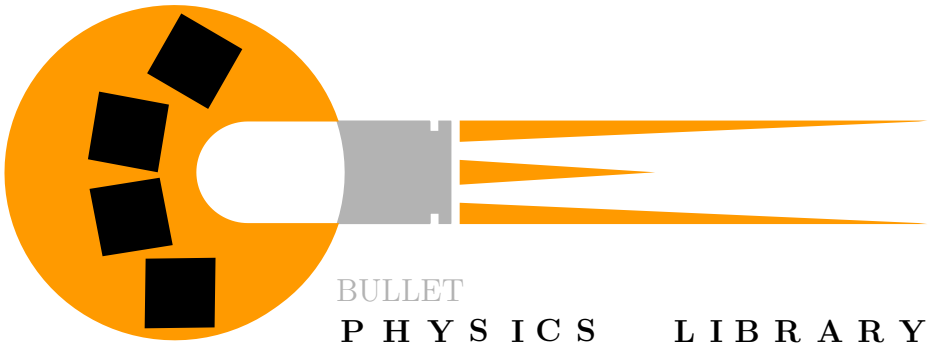


Figura 4.1: Logo de Bullet. Obtenido de https://github.com/bulletphysics/bullet3

En este capítulo se describe y analiza el motor físico Bullet. Al igual que en el capítulo anterior, del análisis, para el que se han utilizado todas las fuentes disponibles, se hace un resumen y síntesis en las siguientes tablas, con los mismos apartados que para el motor físico ODE:

- Aspectos básicos, Tabla 4.4.
- Funcionalidades y arquitectura, Tabla 4.1.
- Principales algoritmos de detección de colisiones, Tabla 4.2

- Algoritmos de simulación, Tabla 4.3.

4.1.1. Descripción general

El segundo motor físico por tratar, Bullet (https://pybullet.org/), cuyas primeras versiones datan del año 2005, es probablemente el motor físico con licencia libre más activo y utilizado. Su principal autor es Erwin Coumans, y la evolución del motor físico ha estado asociada a la trayectoria profesional del autor. Actualmente Bullet parece estar orientándose hacia la robótica impulsado por Google, especialmente con el módulo PyBullet.

Además de la dinámica del cuerpo rígido, Bullet también es capaz de simular cuerpos deformables. En el código fuente se encuentra también una versión que utiliza OpenCL que, aunque funciona, está en versión de pruebas y no está siendo mantenida activamente. Igualmente, en el repositorio se encuentran viejas versiones en las que se implementó una versión de prueba de CA, Continuous Advancement, `btContinuousDynamicsWorld` (Coumans, 2005). Aunque ha desaparecido el código de simulación de CA del código fuente, se siguen manteniendo los algoritmos de CCD.

Ya que este trabajo trata de la dinámica del cuerpo rígido, no se analizarán las capacidades de simulación de cuerpos deformables. Tampoco se analizará la versión de Bullet 3 que hace uso de OpenCL.

El núcleo de Bullet está escrito en C++ y se puede utilizar en las plataformas PlayStation 3, XBox 360, Wii, Windows, Linux, Mac OSX, Android y iPhone (Team [BPDT], 2015).

4.1.2. Ámbito de aplicación y principales usos.

Al igual que ODE su uso está orientado a la simulación interactiva y en tiempo real. En el foro de Bullet hay un listado de aplicaciones creadas con Bullet, destacando su uso en juegos como “Grand Theft Auto IV” y “Red Dead Redemption”. Al igual que ODE también se utiliza en multitud de simuladores de robótica como Gazebo, Webots y CoppeliaSim. Su uso en la comunidad científica es muy alto, tanto la librería como el entorno completo con ayuda de PyBullet. Además de en investigación en dinámica de cuerpos rígidos y deformables, se utiliza extensivamente para entrenar sistemas de

control e inteligencia artificial. La página principal, https://pybullet.org/, se actualiza periódicamente con los principales usos. Bullet también ha sido utilizado para peliculas de animación de primer nivel como Inception, Prince of Persia o X-Men: First Class.

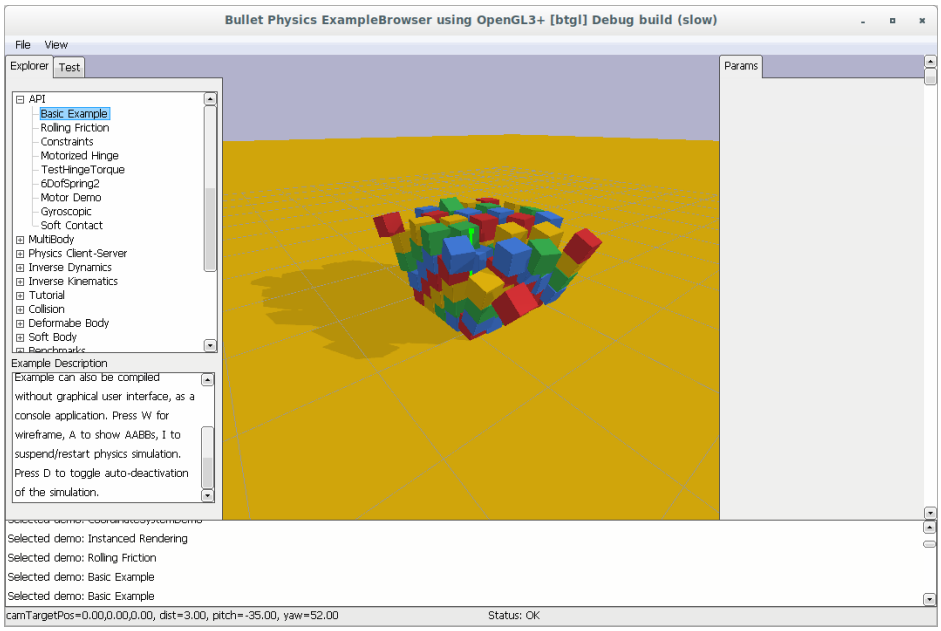


Figura 4.2: Demo de Bullet. ExampleBrowser

4.1.3. Relación con otras herramientas.

Bullet permite importar muchos ficheros de formatos, entre los que destacan Wavefront .obj y Quake .bsp. Además, importa formatos de fichero de cuerpos y robots articulados y con datos físicos como URDF, SDF, MJCF y COLLADA 1.4. Bullet dispone de integraciones con Blender y Maya y tiene un formato de fichero nativo, .bullet, para serializar las escenas.

PyBullet, que se incluye dentro del código de Bullet, es un módulo en Python para simulación de robots y machine learning. PyBullet utiliza Bullet siguiendo una arquitectura cliente-servidor, con una interfaz de usuario y la posibilidad de utilizar visores de realidad virtual. Además de simulación dinámica, existe la posibilidad de utilizar módulos de dinámica inversa y de cinemática directa e inversa para los robots. PyBullet utiliza la API de Bullet C-API, diseñada para independizar el desarrollo de PyBullet del de Bullet, e incluso para permitir en el futuro utilizar otro motor físico.

Como utilidades dentro del código de Bullet también se encuentran algoritmos como V-HACD para descomponer cuerpos cóncavos, interfaces como DebugDraw para facilitar la depuración de la simulación y MotionState para interpolar en puntos intermedios de la simulación. En la Figura 4.2 se muestra la aplicación demostrativa ExampleBrowser que se incluye con el código fuente de Bullet.

4.2. Diseño y arquitectura

Al igual que en ODE, es posible utilizar de forma independiente el módulo de detección de colisiones del módulo de simulación. Sin embargo, como se puede observar en las Figuras 4.3 y 4.4 , el módulo de simulación depende del módulo de detección de colisiones, por lo que no es simple de reemplazar.

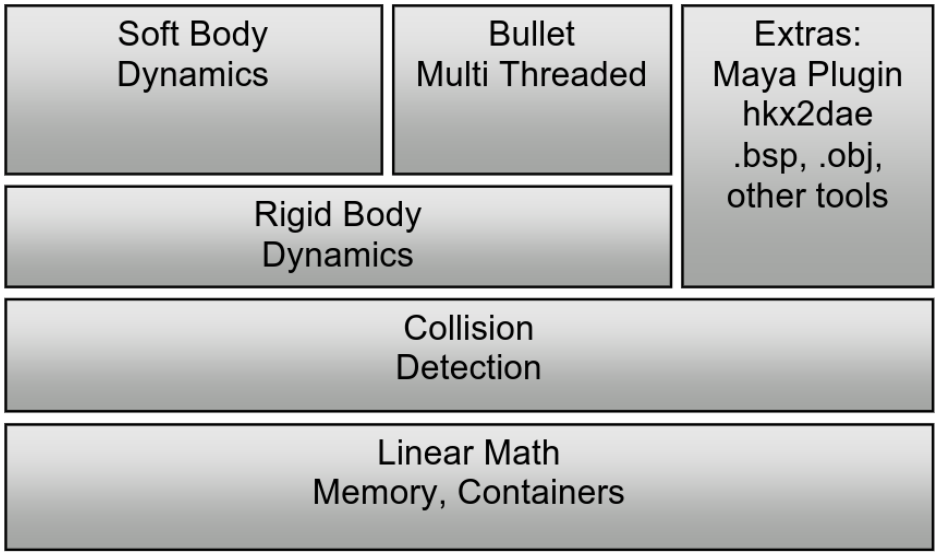


Figura 4.3: Arquitectura de Bullet. A diferencia de ODE, el módulo de simulación de dinámica del cuerpo rígido depende del módulo de detección de colisiones, por lo que es difícil de reemplazar. Imagen obtenida de BPDT (2015)

La principal entidad de Bullet es la clase mundo, `btDynamicsWorld`, que a su vez hereda de `btCollisionWorld`, de la que hay varias subclases según la aplicación que se desee hacer:

- `btDiscreteDynamicsWorld`. Principal mundo para cuerpos rígidos, el que

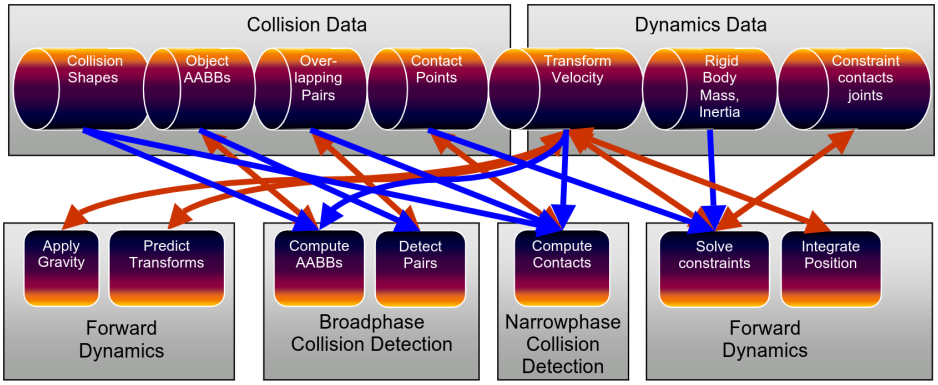


Figura 4.4: Bullet Pipeline. En la figura se puede observar cómo los datos de velocidad transformados, "Transform Velocity", se utilizan para la detección de colisiones al calcular los AABB debido a las funcionalidades de CCD de Bullet. Imagen obtenida de BPDT (2015)

se tratará en este trabajo.

- `btDiscreteDynamicsWorldMt`. Versión para cuerpos rígidos con multihilos.
- `btMultiBodyDynamicsWorld`. Mundo que además de cuerpos rígidos incluye cuerpos rídigos articulados, usando coordinadas generalizadas.
- `btSoftMultiBodyDynamicsWorld`. Mundo que añade cuerpos deformables al anterior.
- `btSoftRigidDynamicsWorld`. Mundo que incluye cuerpos rígidos y cuerpos deformables.
- `btSimpleDynamicsWorld`. Mundo de prueba.

Para crear un mundo son necesarios los siguientes objetos:

- `btDispatcher`. Encargado de crear pares de objetos desde la fase Broadphase, que además selecciona los algoritmos utilizados en la Narrowphase. Puede ser de tipo `btCollisionDispatcher` o la versión multihilo `btCollisionDispatcherMt`.
- `btBroadphaseInterface`. Interfaz para detectar AABB que se solapan. Es decir, la interfaz para la fase Broadphase.

Tabla 4.1: Bullet. Funcionalidades y arquitectura

Característica	Soporte	Detalles
Paradigmas simulación	• Restricciones basadas en velocidad. • Coordenadas generalizadas.	Utiliza el algoritmo de Featherstone para coordenadas generalizadas.
Detección de colisiones independiente de simulación	Sí	Existe un mundo, `btCollisionWorld`, que permite utilizar Bullet sin simulación física.
Facilidad cambio sistema detección de colisiones	Baja	El código de detección de colisiones está muy integrado dentro de Bullet con la simulación física.
Espacios de colisiones recursivos	No	
Varias instancias del motor independientes	Sí	
Posibilidad de geometrías compartidas por los cuerpos	Sí	

- `btConstraintSolver`. Interfaz para resolver las restricciones de articulaciones y contactos. Hay varios tipos según distintos algoritmos y según sean cuerpos rígidos, multicuerpo o deformables (ver Figura 4.7). Por defecto para cuerpos rígido se utiliza la clase `btSequentialImpulseConstraintSolver`.
- `btCollisionConfiguration`. Permite configurar la gestión de colisiones, aspectos como los algoritmos a utilizar o el allocator para la memoria dinámica. La configuración por defecto se define en `btDefaultCollisionConfiguration`.
- Una de las instancias de `btDynamicsWorld`, a la que se le añadirán los cuerpos a simular, su geometría y las restricciones deseadas.

El punto de entrada a la simulación es el método `stepSimulation`, al

que se la pasan dos medidas de paso de tiempo, `timeStep` y `fixedTimeStep`. `fixedTimeStep` es el paso de integración real, por defecto 60 hercios. `timeStep` es el paso que requiere la aplicación llamante, y que Bullet es capaz de interpolar utilizando el paso interno.

4.2.1. Gestión de colisiones

Un cuerpo rígido `btRigidBody` hereda de la clase `btCollisionObject`. A diferencia de ODE, la geometría se puede compartir entre varios cuerpos rígidos. La geometría de un cuerpo se define con las clases de la jerarquía de `btCollisionShape`, pudiéndose agrupar con `btCompoundShape` (patrón Composite). El punto de entrada a la gestión de colisiones es la función `performDiscreteCollisionDetection` de la clase `btCollisionWorld`.

Se describe aquí el funcionamiento general del proceso de detección de colisiones discreta, no teniendo en cuenta la parte de CCD. Se comentarán los algoritmos de detección de colisiones continuas, CCD, en la Sección 4.3.1.

Cuando se añade un nuevo objeto `btCollisionObject` al mundo (`btCollisionWorld`) se genera un objeto que lo recubre, `btBroadphaseProxy` (patrón Decorator). Cada uno de estos proxies dispone de información necesaria para el algoritmo Broadphase, como el AABB, los nodos de los árboles AABB, o información de orden para los test Sweep and Prune.

El algoritmo de Broadphase, que implementa la interfaz `btBroadphaseInterface`, genera pares de posibles colisiones, `btBroadphasePair`, que se retienen en una caché dentro del algoritmo Broadphase. Una vez están generados todos los pares, sobre la clase `btCollisionDispatcher` se ejecuta el método `dispatchAllCollisionPairs`, que por cada par llamará al callback, que por defecto es la función `btCollisionDispatcher::defaultNearCallback`. Esta función creará el algoritmo en el par y lo llamará, de forma que si no hay una colisión el par se puede eliminar de la caché. Estos pares de objetos son los contenedores de información que permite aprovechar la coherencia temporal y espacial en la fase Narrowphase.

En el procesamiento del algoritmo se generará por cada par un `btPersistentManifold`, al que se le añadirán puntos de tipo `btManifoldPoint` con toda la información necesaria para la colisión. Es destacable que los puntos de un `btPersistentManifold` se pueden utilizar más de una vez en la simulación, siendo borrados cuando no se cumplen ciertas condiciones.

Además de la colisión discreta, si está opcionalmente activa, y ya dentro de la simulación en la clase `btDiscreteDynamicsWorld`, la función `createPredictiveContactsInternal` realiza CCD para cada cuerpo que pasa un umbral de movimiento lineal (`getCcdSquareMotionThreshold`), utilizando una esfera corrida (sweep sphere). Esto se vuelve a hacer una vez resueltas las restricciones, para que no haya tunnelling, dejando el objeto en el punto del contacto (motion clamping) en vez de volver a resolver las restricciones.

4.2.2. Simulación

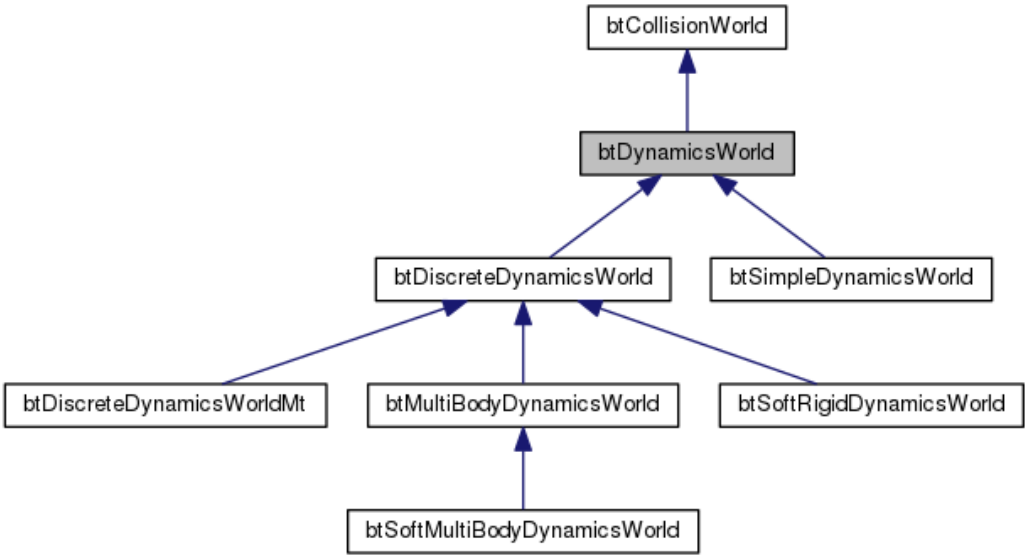


Figura 4.5: Jerarquía de btDynamicsWorld

Los mundos relevantes para la simulación de cuerpos rígidos son `btDiscreteDynamicsWorld` y `btMultiBodyDynamicsWorld` (ver Figura 4.5). Al utilizar `btMultiBodyDynamicsWorld` se pueden combinar cuerpos rígidos articulados formulados con coordenadas reducidas con la formulación maximalista de `btDiscreteDynamicsWorld`.

Para el caso de los cuerpos articulados, `btMultiBody` es la representación que permite articulaciones esféricas, prismáticas, de rotación y planares, y que se resuelven con el algoritmo de Featherstone. Aunque relevante para la simulación del cuerpo rígido y por lo tanto dentro del ámbito de este proyecto, en este apartado solo se va a analizar el funcionamiento de `btDiscreteDynamicsWorld`.

La clase principal que representa a un cuerpo rígido es `btRigidBody`, subclase de `btCollisionObject`. Cabe destacar que a diferencia de ODE no se puede utilizar un tensor de inercia no diagonal, y por tanto el cuerpo rígido y su geometría `btCollisionShape` se deberán alinear para que lo sean. Los cuerpos rígidos pueden ser estáticos, dinámicos o cinemáticos. Los cuerpos estáticos forman la parte estática de la escena (en ODE no eran cuerpos rígidos). A los cinemáticos, al igual que a los rígidos, no les afecta la simulación, pero están preparados para que el usuario de Bullet los pueda animar manualmente.

En cuanto a las restricciones estas heredan de `btTypedConstraint` que a su vez es una subclase de `btTypedObject`. `btTypedObject` además de las restricciones es una súper clase de `btPersistentManifold`, donde se definen los contactos para las colisiones. La jerarquía con todas las restricciones definidas en Bullet se puede ver en la Figura 4.6.

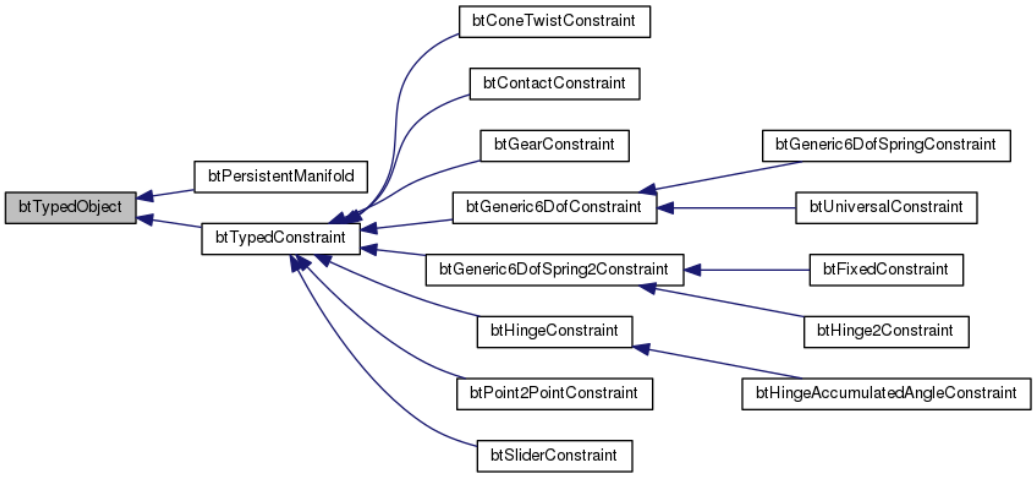


Figura 4.6: Jerarquía de btTypedObject

Bullet utiliza un paso fijo interno y no se recomienda su modificación durante la simulación. Bullet facilita la coordinación entre el paso de la aplicación llamante y el paso interno de Bullet interpolando las posiciones y las velocidades. El funcionamiento del bucle principal de simulación, fundamentalmente definido `btDiscreteDynamicsWorld::internalSingleStepSimulation`, es el siguiente:

- Se añade la gravedad a los cuerpos.
- Se calcula un movimiento predictivo del cuerpo en el paso, `predictUnconstraintMotion`.

- Se buscan contactos predictivos especulativos, `createPredictiveContacts`, según se indica en la Sección 4.2.1. Por cada posible colisión, utilizando el TOI se calcula un posible contacto.
- Se realiza la detección de colisiones de forma discreta `performDiscrete-CollisionDetection`.
- Se calculan islas de cuerpos que están unidos por alguna restricción, de forma que se resuelven las colisiones en cada una de las islas individualmente.
- Utilizando la interfaz `btConstraintSolver` que utilizan los métodos de resolución de restricciones, se resuelven.
- Integra la transformación, `integrateTransforms`. Antes de realizar la integración se vuelven a crear contactos predictivos especulativos. Si se encuentran se realiza "motion clamping", es decir, se detiene al cuerpo antes de la colisión. Una vez terminada la integración, se puede opcionalmente aplicar una fuerza restitutiva a estos cuerpos con la opción `applySpeculativeContactRestitution`.
- Se ejecutan las acciones, callbacks que utilizan los usuarios de Bullet, `updateActions`.
- Se actualizan los estados de actividad (sleep) `updateActivationState`.

4.3. Algoritmos

4.3.1. Detección de colisiones

Aunque ya no existe la implementación de Continuous Advancement, Bullet conserva parte de sus algoritmos CCD en el módulo de gestión de colisiones. Estos se comentarán muy brevemente en esta sección.

Broadphase

Bullet dispone de los siguientes algoritmos Broadphase que implementan la interfaz `btBroadphaseInterface`:

Tabla 4.2: Bullet. Principales algoritmos de detección de colisiones

Tipo	Algoritmos	Detalles
Broadphase por división del espacio	• Sweep and Prune	
Broadphase por división del modelo	• Árbol dinámico BVH	
Convexo-Convexo	• GJK/EPA • GJK/MPR	
TriMesh-trimesh Trimesh-otros	• Árbol dinámico BVH	Solo para objetos estáticos. Se utiliza la librería GIMPACT.
CCD	• Convex Cast	La librería dispone de muchos módulos de CCD, utilizándose actualmente solo para predicciones especulativas.

- Algoritmo de fuerza bruta que utiliza los AABB para un primer filtrado. `btSimpleBroadphaseProxy`.
- Sweep and Prune sobre los tres ejes principales. Estructura de datos optimizada con índices de 16 bits. `btAxisSweep3`.
- Sweep and Prune sobre los tres ejes principales. `bt32BitAxisSweep3`.
- Árbol dinámico de AABB (BVH), que se construye dinámicamente (método insertion, inserción dinámica). Realmente hay dos árboles, uno para objetos dinámicos y otro para objetos estáticos. Utiliza coherencia temporal al guardar pares de ejecuciones anteriores. `btDbvtBroadphase`.

Además de estos algoritmos están las versiones que utilizan OpenCL, en fase experimental. El principal algoritmo es una versión de BVH.

Narrowphase

Bullet dispone de multitud de formas geométricas que heredan de `btCollisionShape`, y que se clasifican en tipos con la estructura `enum LocalBroadphaseNativeTypes` a la hora de seleccionar los algoritmos Narrowphase.

En el manual de Bullet (BPDT, 2015) se incluyen tanto recomendaciones de las formas a utilizar según las necesidades como una tabla con algoritmos

utilizados. En general la recomendación es utilizar cuerpos convexos si estos se mueven, y alguna variación de mallas de triángulos para el escenario. Desafortunadamente el manual no incluye todos los algoritmos realmente utilizados. Se enumerarán en esta sección los algoritmos más relevantes.

Para detección de colisiones entre cuerpos convexos, además de algunos específicos se utiliza SAT o GJK con EPA para calcular una medida de penetración. Existe también una versión que calcula una estimación de penetración utilizando espacios de Minkowski por fuerza bruta. En los ejemplos está implementada una versión que utiliza MPR para calcular la penetración (`btConvexConvexMprAlgorithm`).

En el caso de cuerpos compuestos o de mallas de triángulos utiliza al igual que en la fase Broadphase árboles BVH. Además, y al igual que en ODE, es posible utilizar la librería GIMPACT.

Bullet dispone de implementaciones de algoritmos de CCD. Todos ellos están basados en Convex Cast, que consiste en barrer un cuerpo a lo largo de una línea y parar en la primera intersección con otro cuerpo. Cuando se habla de Ray Cast se refiere a que el cuerpo barrido es un punto. Las implementaciones de Bullet se pueden analizar en `btContinuousConvexCollision`, `btGjkConvexCast` y `btSubsimplexConvexCast`, descritos en Couman y kevlar (s.f.). En la simulación solo se está utilizando la versión que utiliza el barrido de un punto y de una esfera.

4.3.2. Simulación física y restricciones

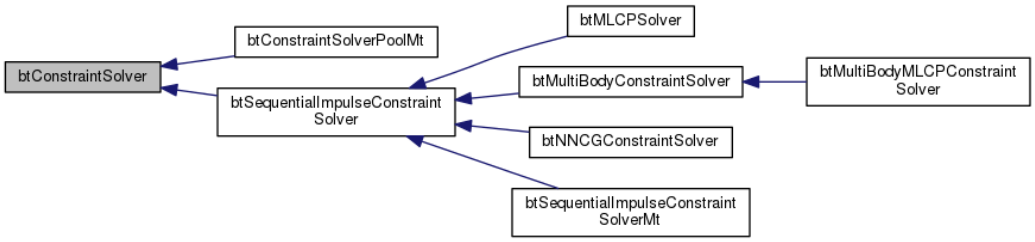


Figura 4.7: Jerarquía de btConstraintSolver

Bullet dispone de la posibilidad de utilizar cuerpos deformables junto a cuerpos rígidos. Además, dispone de una implementación no mantenida del motor físico utilizando GPGPU (OpenCL).

Tabla 4.3: Bullet. Algoritmos de simulación

Nombre	Soporte	Detalles
Fricción	• Pirámide linealizada • Proyección sobre el cono de fricción.	
Algoritmo Integración	Euler semi-implícito	El paso es fijo.
Algoritmos LCP/MLCP	• Sequential Impulse • PGS • NNCG • Dantzig • Lemke	
Término giroscópico	• Implícito en sistema global • Implícito en sistema del cuerpo • Explícito	Las opciones están en el enumerado `btRigidBodyFlags`.
Tratamiento errores en restricciones	• ERP y CFM • Stiffness y damping.	
Uso CCD	• Contactos especulativos • Motion Clamping	

Al igual que ODE, Bullet utiliza el paradigma basado en restricciones a nivel de velocidad (`btRidigBody`), pero también contiene una implementación del algoritmo de Featherstone basado en coordenadas generalizadas (`btMultiBody`), siendo posible utilizar ambos paradigmas de forma simultánea.

Bullet utiliza en el modo de restricciones a nivel de velocidad `btDiscrete-DynamicsWorld` una pirámide de fricción linealizada con dos direcciones definibles por el usuario. Además, al utilizar el mundo de coordenadas generalizadas, `btMultiBodyDynamicsWorld`, es posible utilizar el cono de fricción de forma implícita, de forma que la fricción lateral se proyecta contra el cono

de fricción (Couman & bram, s.f.).

Bullet dispone también de opciones para establecer fricción de rotación y rodante (rolling friction y spinning friction), fricción dependiente el deslizamiento (FDS, force-Dependent-Slip) y fricción en las articulaciones (joint friction). Para establecer el coeficiente de restitución se utiliza un valor entre 0 y 1.

En cuanto al algoritmo para resolver el MLCP, Bullet utiliza por defecto el algoritmo iterativo de impulsos secuenciales (sequential impulses), `btSequentialImpulseConstraintSolver`. La mayor ventaja de este algoritmo es que no es necesario preparar la estructura de datos de la matriz no densa. Este algoritmo se explica en Catto (2006).

De forma experimental, en el código fuente de Bullet se encuentra implementados los siguientes algoritmos (ver Figura 4.7):

- Iterativo. NNCG (Nonsmooth Nonlinear Conjugate Gradient). `btNNCGConstraintSolver`.
- Iterativo. Projected Gauss Seidel. `btSolveProjectedGaussSeidel`, usado por `btMLCPSolver`.
- No iterativo. Dantzig. `btDantzigSolver`, usado por `btMLCPSolver`.
- No iterativo. Lemke. `btLemkeAlgorithm`, usado por `btMLCPSolver`.

Además de tenerse en cuenta los contactos discretos, Bullet realizar Convex Cast (sweep volume) sobre el movimiento predicho sin tener en cuenta las restricciones de los cuerpos. Estas nuevas posibles colisiones, con una distancia de penetración positiva, se tendrán en cuenta en el algoritmo MLCP para cuerpos marcados como CCD. Además, una vez calculadas las fuerzas finales, se volverá a realizar un Convex Cast a los cuerpos marcados con CCD, y si se detecta penetración con el Convex Cast, se utilizará el algoritmo "CCD motion clamping", que sin tener en cuenta ni fricción ni coeficiente de resolución parará el movimiento del cuerpo en el primer instante de la colisión.

En cuanto a la estabilización de Baumgarte, además de poder utilizar los parámetros CFM y ERP como en ODE, se pueden definir directamente como parámetros de un sistema resorte amortiguador.

4.4. Aspectos prácticos y resumen

La documentación de Bullet se encuentra en Github junto con el código fuente. El manual de usuario es muy básico y es más una introducción a Bullet, no muy actualizada, que un manual de usuario. En cambio, en el manual de PyBullet se pueden encontrar las descripciones de muchos parámetros del motor físico.

No existe documentación de diseño ni de los algoritmos de Bullet. Sin embargo, el foro de Bullet contiene hilos con multitud de información que van desde temas generales de simulación a decisiones tomadas al crear Bullet.

El proyecto está desarrollado muy activamente, en muchos casos por personas e instituciones relacionadas con la investigación.

Tabla 4.4: Bullet. Aspectos básicos

Nombre	**Soporte**	**Detalles**
Licencia	zlib	
Estado del proyecto	Maduro	
Evolución del proyecto	Alta	
Calidad documentación	Baja	Es escasa y difícil de encontrar.
Principales usos	• Videojuegos • Robótica • Animación • Investigación	La página principal de Bullet https://pybullet.org/ es un escaparate de proyectos realizados con Bullet y PyBullet.
Plataformas soportadas	• PlayStation • XBox • Wii • Windows • Linux • macOS • Android • iPhone	

Capítulo 5

Posibilidades y comparativa

Este capítulo trata de resolver la primera parte de este proyecto, un análisis de las posibilidades de los motores físicos ODE y Bullet. La forma de plantearlo ha sido estudiando sus características y además realizando una comparación entre ambos.

Antes de abordar el análisis de las posibilidades, es necesario delimitar su ámbito, que generalmente va a ir asociado a la aplicación práctica de los motores físicos, habiéndose expuesto las principales aplicaciones en la Sección 1.3. Será la aplicación deseada la que determine los requisitos que deberá cumplir el motor físico, tanto de tipo funcional como de tipo no funcional. Así la licencia, la facilidad de uso o la documentación pueden ser tan importantes como los algoritmos utilizados.

No se debe olvidar que tanto Bullet como ODE están especialmente adaptados para aplicaciones interactivas y en tiempo real (soft real-time), y encajan en el campo de la animación basada en física. Parece obvio por lo tanto que ambos se pueden utilizar para crear algunos tipos de videojuegos, animaciones y simulaciones capaces de conseguir al menos apariencia de realismo.

Existen en la literatura numerosos análisis, evaluaciones, comparativas y pruebas de rendimiento entre motores físicos que persiguen objetivos similares a los de este capítulo. Algunos de las más relevantes son los siguientes: Boeing y Bräunl (2007), Erez, Tassa y Todorov (2015) Mondesire, Maxwell y Stevens (2016), Hummel, Wolff, Stein, Gerndt y Kuhlen (2012) Roennau, Sutter, Heppner, Oberlaender y Dillmann (2013) y, Kang y Hwangho (s.f.).

Cada uno de ellos tiene unos motivos últimos distintos, desde la pura comparativa hasta la presentación de un nuevo motor físico.

Este capítulo se compondrá de dos partes bien diferenciadas. Se enumerarán primero las características de cada motor gráfico, tomadas de la investigación realizada en los capítulos anteriores. Después se procederá a realizar una pequeña batería de pruebas prácticas (benchmark) con los dos motores para tratar medir algunas de sus propiedades. Posteriormente se expondrán las conclusiones encontradas, tanto en la comparación de propiedades, más teórico, como en las pruebas, más práctico.

5.1. Comparativa de características de los motores

Se vuelven a representar en este apartado las tablas de los dos capítulos anteriores. En este caso se comparan directamente las características de los motores ODE y Bullet. Son las siguientes tablas:

- Aspectos básicos, Tabla 5.4. Basada en las Tablas 3.4 y 4.4.
- Funcionalidades y arquitectura, Tabla 5.1. Basada en las Tablas 3.1 y 4.1.
- Principales algoritmos de detección de colisiones, Tabla 5.2. Basada en las tablas 3.2 y 4.2.
- Algoritmos de simulación, Tabla 5.3. Basada en las Tablas 3.3 y 4.3.

Estas tablas resumen las principales características de los motores. Para realizarlas ha sido necesario seleccionar unas características y excluir otras. El autor considera que, dentro de cierta arbitrariedad inevitable, las características seleccionadas permiten comparar los motores de forma fiel. A pesar de sus diferencias, hay muchas similitudes. Ambos utilizan de forma principal un modelo basado en restricciones de velocidad. Además, Bullet ha tomado ideas prestadas de ODE, como por ejemplo CFM y ERP (ver Tabla 5.3).

Tabla 5.1: Comparativa. Funcionalidades y arquitectura

Característica	**ODE**	**Bullet**
Paradigmas simulación	• Restricciones basadas en velocidad	• Restricciones basadas en velocidad. • Coordenadas generalizadas
Detección de colisiones independiente de simulación	Sí	Sí
Facilidad cambio sistema detección de colisiones	Alta	Baja
Espacios de colisiones recursivos	Sí	No
Varias instancias del motor independientes	Parcial	Sí
Posibilidad de geometrías compartidas por los cuerpos	No	Sí

Tabla 5.2: Comparativa. Principales algoritmos de detección de colisiones

Tipo	**ODE**	**Bullet**
Broadphase por división del espacio	• Grid jerárquico • QuadTree • Sweep and Prune	• Sweep and Prune
Broadphase por división del modelo	-	• Árbol dinámico BVH
Convexo-Convexo	• SAT • MPR	• GJK/EPA • GJK/MPR • SAT
Trimesh-Trimesh Trimesh-Otros	• Top-Down AABB tree • Box-Pruning	Árbol dinámico BVH
CCD	-	• Convex Cast

Tabla 5.3: Comparativa. Algoritmos de simulación

Tipo	**ODE**	**Bullet**
Fricción	• Box-Friction • Pirámide linealizada	• Pirámide linealizada • Proyección sobre el cono de fricción
Algoritmo Integración	Euler semi-implícito	Euler semi-implícito
Algoritmos LCP/MLCP	• Dantzig • Succesive Over Relaxation	• Sequential Impulse • PGS • NNCG • Dantzig • Lemke
Término giroscópico	• Implícito en coordenadas globales	• Implícito en sistema global • Implícito en sistema del cuerpo • Explícito
Tratamiento errores en restricciones	• ERP y CFM	• ERP y CFM • Stiffness y damping
Uso CCD	-	• Contactos especulativos • Motion Clamping

Tabla 5.4: Comparativa. Aspectos básicos

Aspecto	**ODE**	**Bullet**
Licencia	BSD y LGPL	zlib
Estado del proyecto	Maduro	Maduro
Evolución del proyecto	Baja	Alta
Calidad documentación	Media	Baja
Principales usos	• Videojuegos • Robótica	• Videojuegos • Robótica • Animación • Investigación
Plataformas soportadas	• Linux • Windows • Mac OSX	• PlayStation • XBox • Wii • Windows • Linux • Mac OSX • Android • iPhone

5.2. Benchmark. Pruebas prácticas

El autor ha considerado necesario realizar una pequeña batería de pruebas sobre los motores físicos ODE y Bullet para poder analizar con mayor profundidad sus posibilidades y la comparación entre ellos, además de permitir indagar en características de los motores no encontradas analizando el código ni la documentación.

El número de aspectos y configuraciones que se pueden evaluar excede las posibilidades de este trabajo. Se ha optado por realizar cinco escenarios simples y con un número de configuraciones muy limitado. Será el usuario del motor físico, el investigador o el desarrollador, según sus necesidades, el que determine qué pruebas y bajo qué configuraciones son necesarios. Para este trabajo se ha tratado de que estas pruebas sean representativas del motor físico con respecto a las propiedades a evaluar.

Las propiedades generales a evaluar escogidas por el autor son las siguientes: plausibilidad, precisión física, rendimiento y estabilidad/robustez. Se definen de la siguiente forma:

- Plausible, entendida esta palabra como que independientemente de seguir las leyes físicas, en general parece que las sigue. Es especialmente importante en animación y videojuegos.

  Se valorará como Bajo/Medio/Alto de forma subjetiva por el autor del trabajo al visualizar la animación.

- Precisión física de la simulación. No es crítico en videojuegos y animación, siendo más importante por ejemplo en simulación y robótica.

  Para evaluar la precisión física se analizará el momento lineal, el momento angular y la energía cinética del sistema, teniendo en cuenta que se conservan en un sistema cerrado conservativo y no inercial. También se evaluarán propiedades conocidas del sistema en cuestión. Debe de quedar claro que un cuerpo rígido es una simplificación de la realidad, y por lo tanto nunca es perfectamente realista. Igualmente, los paradigmas de simulación basados en restricciones de velocidad tampoco modelan la realidad de una forma fidedigna (Chatterjee, 1999).

  Se valorará como Bajo/Medio/Alto.

- Rendimiento. Aunque menos importante en animación que no es en tiempo real, es muy importante en videojuegos y sistemas de simula-

ción para entrenamiento y crítico en sistemas predictivos en robótica, háptica y cirugía.

Para cada prueba se dará la ratio entre el tiempo real de la simulación y el tiempo que ha requerido el motor físico para realizar la simulación. Así, un valor superior a uno indica que la simulación se ha realizado a una velocidad superior al tiempo real. Este valor será para el total de la escena simulada, lo que implica que el tiempo requerido para el paso de simulación podrá ser a veces más rápido que el tiempo real y a veces más lento, dentro de la misma simulación. Se ha optado por esta métrica por su simplicidad, aunque dependiendo de la aplicación podría ser necesaría otra como el mínimo, el máximo o una medida de dispersión como la varianza.

- Estabilidad/Robustez. La palabra estabilidad tiene muchos significados. En el ámbito de las ecuaciones diferenciales y el control suele referirse a que una pequeña perturbación en la entrada o en las condiciones iniciales no varía sustancialmente la solución final. En cuanto a robustez, generalmente se entiende de forma más amplia que la estabilidad, en el que el sistema sigue funcionando de forma aceptable ante perturbaciones internas o externas, incluso si son físicamente imposibles.

  En este trabajo, el concepto estabilidad/robustez se refiere a que el sistema se comporta de una forma "aceptable", independientemente de si es preciso físicamente o plausible. Implica, por ejemplo, que no hay ganancia de energía de forma inestable, que las restricciones no se incumplen de una forma indefinida en el tiempo...

  Se valorará como Bajo/Medio/Alto.

Se ha tratado de que los escenarios a probar, aun siendo simples, sean equilibrados en cuanto a su espectro. Un resumen de escenarios y configuraciones habitualmente testados se puede encontrar en Bender y col. (2012), y multitud de ideas se pueden encontrar en libros de mecánica clásica o en otras comparativas.

5.2.1. Herramienta de pruebas. Obugre

Tanto para el análisis de las pruebas como para su visualización se ha considerado necesario construir una herramienta con un API común para los motores Bullet y ODE. Esta herramienta permitirá tanto la creación de escenas como la visualización de la simulación.

Aproximaciones y frameworks similares en la literatura

En la literatura existen herramientas similares a Obugre. Las más relevantes son las siguientes:

- PEEL (Terdiman, s.f.-b). Herramienta para evaluar, comparar y probar motores físicos, especialmente en lo referente al rendimiento, uso de memoria y plausibilidad. No integra ODE.
- PAL (Boeing, s.f.). Interfaz común para multitud de motores incluyendo ODE y Bullet, utilizado para la comparativa descrita en Boeing y Bräunl (2007). No está mantenido actualmente y el autor no ha conseguido probarlo.
- SimBenchmark (Kang & Hwangho, s.f.). Creado por los desarrolladores del motor físico Raisim para probar aplicaciones robóticas. Incluye API tanto para ODE como para Bullet. Utilizado en Erez y col. (2015). El autor no lo ha probado.
- Gangsta (cos_xavier & sfmonster, s.f.). Interfaz común para motores físicos, anterior a PAL y sin mantener. No incluye Bullet.

Se ha optado por no utilizar ninguno de ellos por dos motivos. El primero la experiencia de comprobar de primera mano las diferencias y facilidad de uso de los motores físicos. El segundo es la dificultad de crear las pruebas deseadas con estas herramientas. PAL es una alternativa viable, sin embargo, está sin mantener desde 2013 y adaptarlo se ha considerado más complejo que crear una herramienta más simple de cero. En cuanto a PEEL y SimBenchmark, tienen objetivos distintos, estando el primero orientado a la plausibilidad y el segundo a la robótica. En PEEL sería necesario crear la interfaz con ODE y métodos para obtener parámetros físicos como energía o momentos. SimBenchmark cumple casi todos los requisitos excepto la visualización de la escena que es más básica y basada en Matlab, no pudiéndose comparar dos motores físicos simultáneamente como en PEEL.

Framework de pruebas Obugre

Para realizar las pruebas se ha creado un pequeño framework con el motor gráfico Ogre. Los requisitos para el framework son los siguientes:

- Crear un API mínimo que permita utilizar ODE y Bullet de forma homogénea.
- Posibilidad de visualizar cada prueba de forma simultánea en los dos motores. Con esta opción será más simple valorar la plausibilidad.
- Posibilidad de analizar el rendimiento.
- Posibilidad de obtener información de cada cuerpo y motor físico, tal como posiciones, propiedades físicas (momento angular, momento linear, energía...).

Inicialmente se ha tratado de realizar un API común que incluya todas las funcionalidades de los motores. Ante el enorme coste, finalmente, manteniendo los requisitos, se han creado solo las funcionalidades necesarias para cada prueba. En el Anexo Guía de instalación y uso del software se encuentra más información sobre la herramienta.

Configuración de los motores físicos

Es inviable para este trabajo probar todas las configuraciones de los motores físicos para cada prueba. En cualquier caso, el autor ha probado varias configuraciones, especialmente en las situaciones en que un motor físico se ha comportado de una forma contraria a las expectativas. Se indicarán en cada prueba los casos en los que se han probado distintas configuraciones de parámetros para una escena que sean relevantes.

Por defecto se utilizará la configuración inicial de cada motor con un paso de integración 1/60 segundos. Se deshabilitará la desactivación de cuerpos (sleeping) y se habilitarán las fuerzas giroscópicas. No se utilizarán coordenadas generalizadas en Bullet (`btRigidBody`) aunque para alguna de las pruebas hubiese sido óptimo.

5.2.2. Sobre las pruebas seleccionadas

Las pruebas realizadas son una selección que trata de poder valorar las cuatro propiedades a analizar, y que por lo tanto cubre buena parte del espectro de las aplicaciones de un motor físico. Junto con la descripción de la prueba se indicarán los motivos por los que ha sido seleccionada.

Algunas otras opciones que se han valorado y que no se han seleccionado finalmente son las siguientes:

- Teletransportación de cuerpos forzando penetraciones y configuraciones inválidas en las restricciones.
- Peonza celta, como movimiento complejo difícil de entender y modelar.
- Sistema Solar, como ejercicio útil en educación.
- Torre con cuerpos de masas y tamaños muy distintos, para probar la estabilidad del sistema.

En lo que sigue a este capítulo no se indicarán unidades físicas para medir propiedades físicas. Se puede sobreentender que se utilizan las unidades del Sistema Internacional. Una discusión de este aspecto se puede encontrar en el manual de ODE (Russel Smith, s.f.).

5.2.3. Prueba 1. Simulación cuerpo balístico

Esta prueba está inspirada en la sección 10.8 del libro Taylor (2005). El objetivo es probar la precisión física de la simulación en lo referente a las fuerzas giroscópicas en ausencia de colisiones.

Todo cuerpo rígido tiene tres momentos principales de inercia (el tensor de inercia es diagonalizable). En el caso de que estos sean distintos, el mayor y el menor serán estables, y el intermedio inestable (Taylor, 2005).

Se han creado tres cajas iguales, con lados de longitud 4, 2 y 6. Inicialmente cada una de ellas tiene una velocidad angular ligeramente desalineada con cada momento principal de inercia. La caja superior es la que gira inicialmente cerca del momento principal de inercia mayor. la central sobre el momento principal de inercia intermedio y la inferior la que gira sobre el momento principal de inercia menor.

En la Figura 5.1 se observa la situación inicial. Las flechas cian representan velocidad angular y las flechas violetas momento angular. La expectativa es que además de conservar los momentos y la energía cinética, los momentos principales mayor y menor sean estables, y el intermedio inestable.

A los 28 segundos (ver Figura 5.2a) se observa que la caja inferior en ODE no se estabiliza en el momento principal de inercia menor. Tanto en Bullet como en ODE, correctamente, la caja intermedia no se estabiliza sobre el momento principal de inercia intermedio.

A los 45 minutos (ver Figura 5.2b) en ODE las tres cajas giran sobre el momento principal de inercia mayor. En Bullet la superior y la intermedia giran sobre el momento principal de inercia mayor y la inferior sobre el momento principal de inercia menor. El caso de ODE es claramente erróneo y en el caso de Bullet el autor no ha comprobado que esta sea la situación correcta. Sin embargo, sí que se observa una elevada pérdida de energía en la caja intermedia en Bullet, según se puede observar en la Figura 5.3b. En ambos motores, alinearse con un momento de inercia principal distinto al inicial resulta en una pérdida de energía (ver Figura 5.3).

En el artículo Lacoursière (2006) se explica el algoritmo de ODE, y se demuestra el problema de inestabilidad en el eje alineado con el momento principal de inercia menor. El autor no ha encontrado ningún estudio similar para Bullet.

Tabla 5.5: Prueba 1. Resumen

Propiedad	**Motor**	**Valor**	**Comentarios**
Plausibilidad	ODE	Alta	El autor considera que el problema del momento principal de inercia menor parece realista.
	Bullet	Alta	
Precisión física	ODE	Baja	El momento principal de inercia menor es inestable, lo que no es físicamente preciso.
	Bullet	Media	Aunque el momento principal de inercia menor es estable, pierde energía.
Rendimiento	ODE	5831	
	Bullet	4918	
Estabilidad/ robustez	ODE	Alta	
	Bullet	Alta	

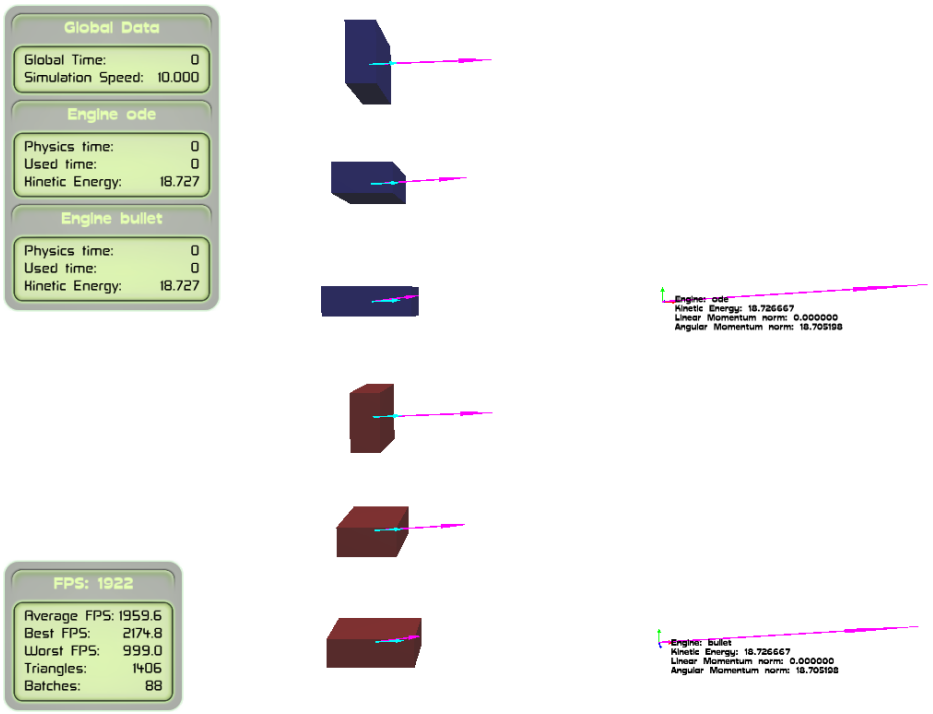
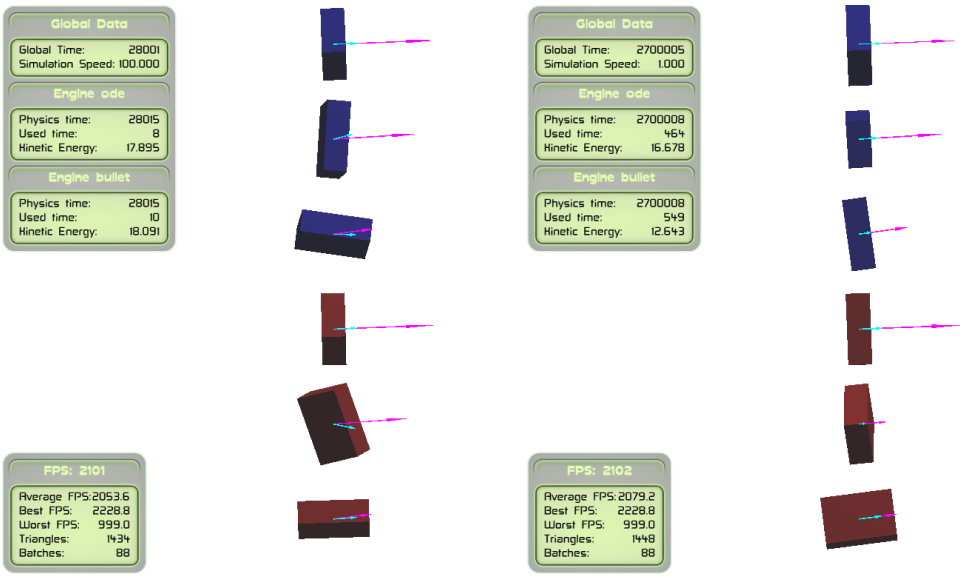
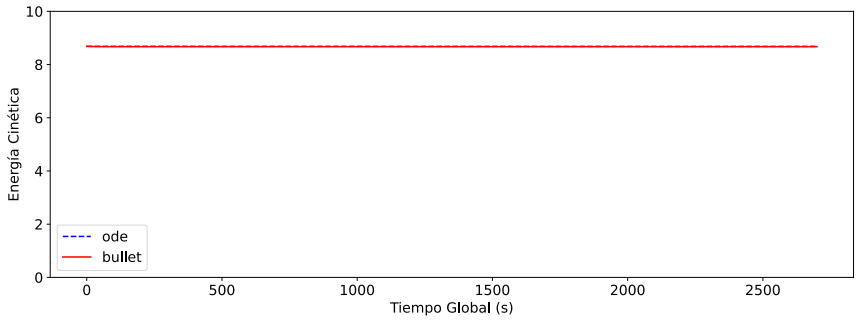


Figura 5.1: Prueba 1. Situación inicial. Azul ODE, rojo Bullet. El momento angular se representa en violeta y la velocidad angular en cian. Cada una de las cajas de cada motor tiene la velocidad angular ligeramente desalineada con uno de los momentos de inercia principales. La caja superior con el momento principal de inercia mayor, la intermedia con el momento principal de inercia intermedio y la inferior con el momento principal de inercia menor. La expectativa es que las cajas superior e inferior sean estables y la intermedia inestable.

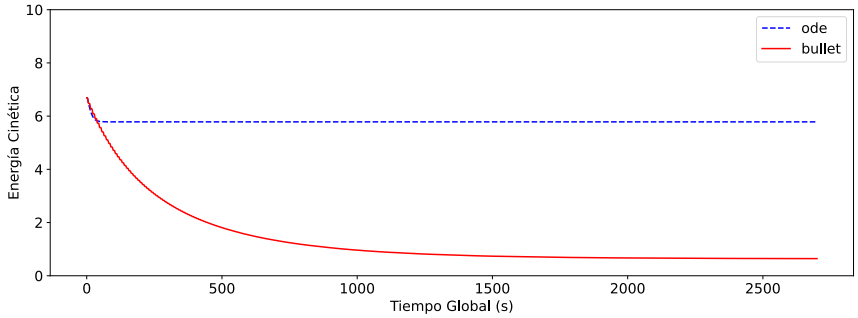


(a) 28 segundos (b) 45 minutos

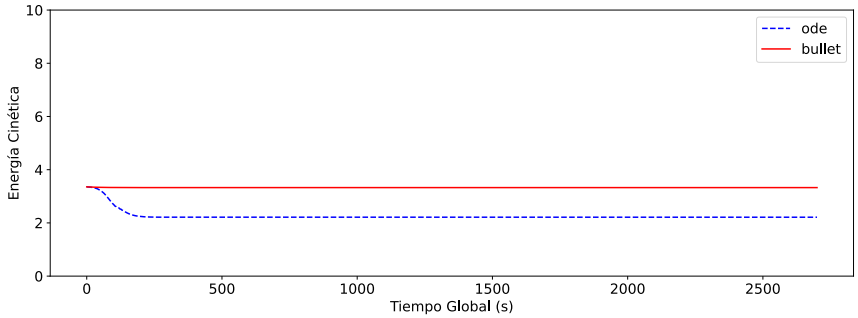
Figura 5.2: Prueba 1. A los 28 segundos y a los 45 minutos. En (a), se observa en ODE (azul) que la velocidad angular y el momento angular se desalinean en las cajas intermedia e inferior, mientras que en Bullet (rojo) solo se desalinea el intermedio. En (b) se puede observar cómo en ODE todas las cajas terminan girando sobre el momento principal de inercia mayor, mientras que en Bullet solo lo hacen la superior y la intermedia.



(a) Cuerpo con velocidad angular inicial alineada con momento mayor



(b) Cuerpo con velocidad angular inicial alineada con momento intermedio



(c) Cuerpo con velocidad angular inicial alineada con momento menor

Figura 5.3: Prueba 1. Pérdida de energía cinética en el tiempo en cada caja

5.2.4. Prueba 2. Esferas y plano inclinado

Este es un ejemplo sencillo de física básica, apropiado para la educación. Dos esferas ruedan por un plano inclinado desde la misma altura. Aunque ambas tienen la misma masa, una de ellas es maciza y la otra es una corona esférica (tiene un hueco, una cavidad esférica). Ambas tienen densidad constante.

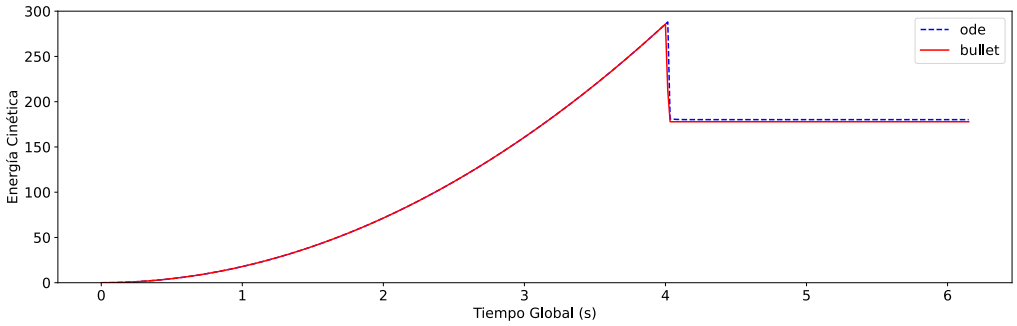
El coeficiente de fricción será una constante alta para que las esferas rueden y no se deslicen por la rampa. El coeficiente de restitución será cero, aunque no debería tener ningún efecto hasta que las esferas llegan al suelo. No hay fricción de rodadura (rolling friction).

La expectativa es que, al finalizar la rampa, la bola maciza llegue antes que la bola con una cavidad. La energía cinética de ambas al finalizar la rampa debería ser igual y equivalente a la pérdida de energía potencial.

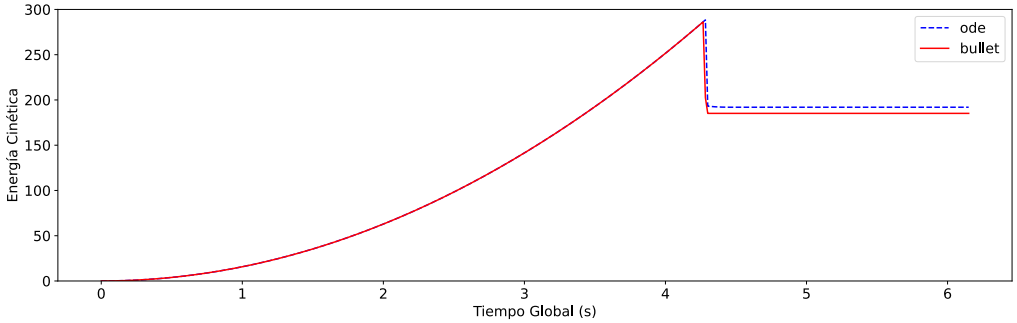
En la Figura 5.5 se puede comprobar como al bajar las esferas, la esfera maciza de cada motor baja más rápido y su momento angular es menor que el de la otra esfera, lo que es coherente con la física.

Analizando los registros de la prueba, la altura de las bolas es 28,83, la masa de las bolas 1 y la gravedad 10, por lo que la energía potencial con respecto al suelo es 288,3. La esfera maciza, en su máximo, alcanza una energía cinética de 288,1 a los 4,00 segundos en ODE y de 285.71 a los 3,99 segundos en Bullet. La esfera hueca en su máximo alcanza una energía cinética de 288,24 a los 4,28 segundos en ODE y de 286,18 a los 4,27 segundos en Bullet (ver Figura 5.4).

Ambos resultados son coherentes, sabiendo por ejemplo que Bullet tiene un margen en la colisión de 0,04 por el que se puede perder un poco de energía.



(a) Energía cinética esfera maciza



(b) Energía cinética esfera hueca

Figura 5.4: Prueba 2. Energía cinética

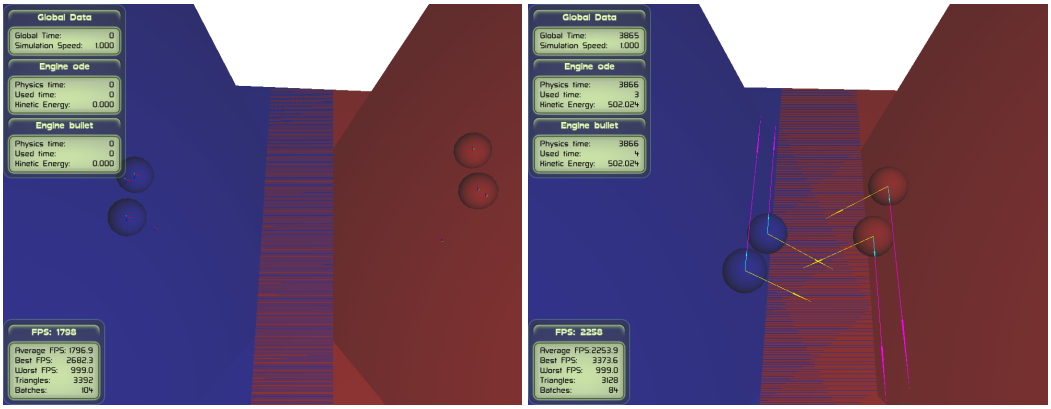


Figura 5.5: Prueba 2. Esferas bajando rampa

Tabla 5.6: Prueba 2. Resumen

Propiedad	**Motor**	**Valor**	**Comentarios**
Plausibilidad	ODE	Alta	
	Bullet	Alta	
Precisión física	ODE	Alta	
	Bullet	Alta	Hay una ligera pérdida de energía, que se puede achacar al margen de las colisiones en Bullet.
Rendimiento	ODE	1080	
	Bullet	832	
Estabilidad/ robustez	ODE	Alta	
	Bullet	Alta	

5.2.5. Prueba 3. Bandera creada con bisagras

El objetivo de esta prueba es comprobar la estabilidad y corrección en presencia de cadenas de restricciones (articulaciones). Este tipo de configuraciones es habitual en las pruebas de motores físicos.

Se ha creado una cadena formada por eslabones unidos por bisagras de masa 1 (hinge contraint) y al final de la cadena se ha colocado una esfera de masa 5. La cadena está anclada a un poste a cierta altura del suelo. La esfera tiene una velocidad lineal inicial perpendicular a la cadena (ver Figura 5.6). El coeficiente de restitución será uno y el de fricción cero.

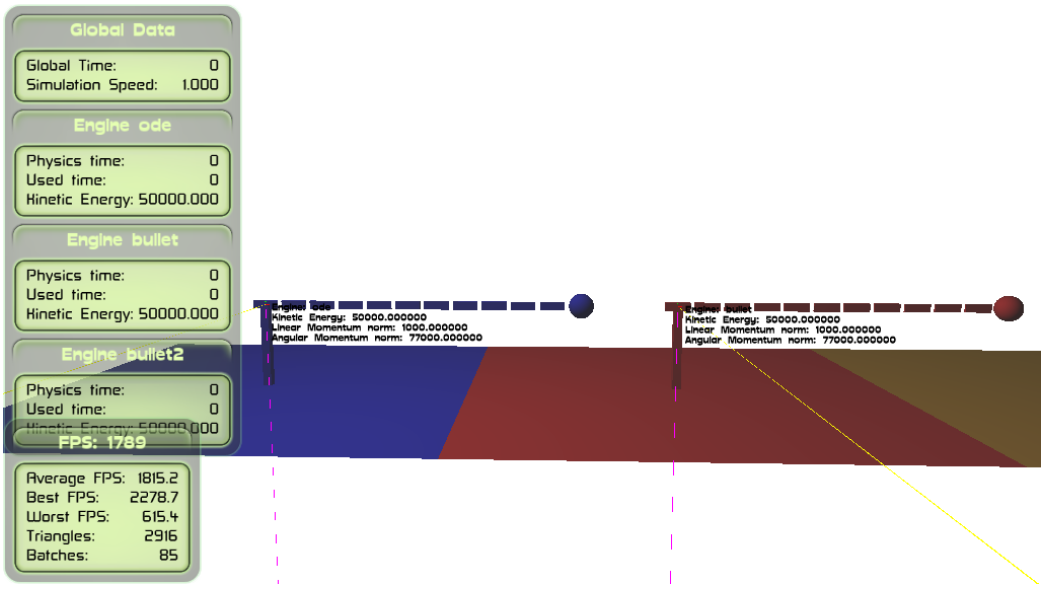
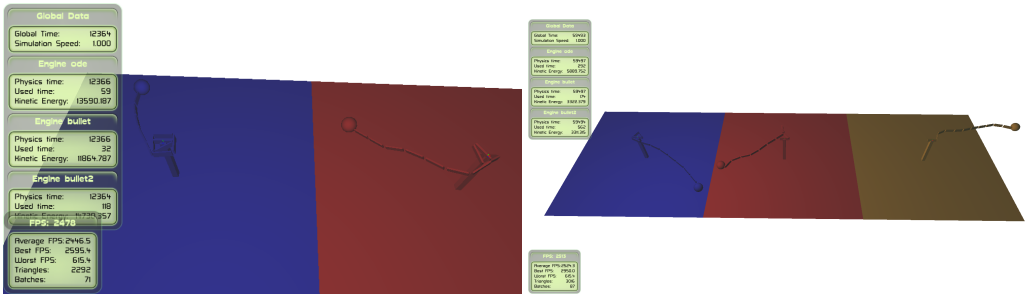


Figura 5.6: Prueba 3. Situación inicial. Azul ODE, rojo Bullet

Una vez arrancada la simulación, ODE es más estable que Bullet. En Bullet se producen situaciones imposibles como se puede comprobar en la Figura 5.7a, en la que por el mayor desvío de las restricciones los eslabones terminan colocándose unos encima de otros. El autor no ha conseguido solucionar este problema variando los coeficientes ERP y CFM en Bullet, y la única solución encontrada ha sido disminuir el tamaño del paso de integración. En la Figura 5.7b se reproduce una ejecución con Bullet con un paso de integración de 1/240 segundos, en la que se puede ver que ahora la escena es estable.

Se debe de tener en cuenta que para Bullet no se han usado coordinadas



(a) Situación imposible en Bullet (b) En naranja Bullet paso 1/240

Figura 5.7: Prueba 3. Evolución cadena. ODE en azul y Bullet en rojo con pasos de integración 1/60. Se puede comprobar como Bullet con el paso de integración menor no mantiene correctamente las restricciones, ya que hay eslabones superpuestos verticalmente.

Tabla 5.7: Prueba 3. Resumen

Propiedad	Motor	Valor	Comentarios
Plausibilidad	ODE	Media	Se observa que la bola ha bajado.
	Bullet	Baja	Los eslabones se superponen si no se cambia el paso.
Precisión física	ODE	Media	Se pierde energía. Además, se observan momentos con aumentos de energía.
	Bullet	Media	Ignorando el problema de las restricciones, al igual que en ODE se pierde energía.
Rendimiento	ODE	205	
	Bullet	342	Con el paso a 1/240 segundos, Bullet es casi tan rápido como ODE y mantiene las restricciones de forma aceptable.
Estabilidad/ robustez	ODE	Alta	
	Bullet	Media	Aunque el sistema no implosiona, las restricciones se incumplen de una manera notoria.

generalizadas, que sería la forma natural de resolver este problema, y donde es previsible que no se llegue a situaciones indeseadas con ningún paso de integración.

5.2.6. Prueba 4. Rendimiento de colisiones

Esta prueba trata de ver cómo se comportan los motores cuando hay muchas colisiones, y por lo tanto probará especialmente el rendimiento. Para ello el escenario consiste en cuatro planos inclinados 45 grados, colocados como si fuera al interior de una pirámide cuadrada con la cúspide apoyada en el suelo (ver Figura 5.8). Se irán lanzando diez cuerpos cada segundo, que serán esferas, cuerpos convexos y cajas (ver Figura 5.9). El coeficiente de restitución será 0,5 y el de fricción 0,5.

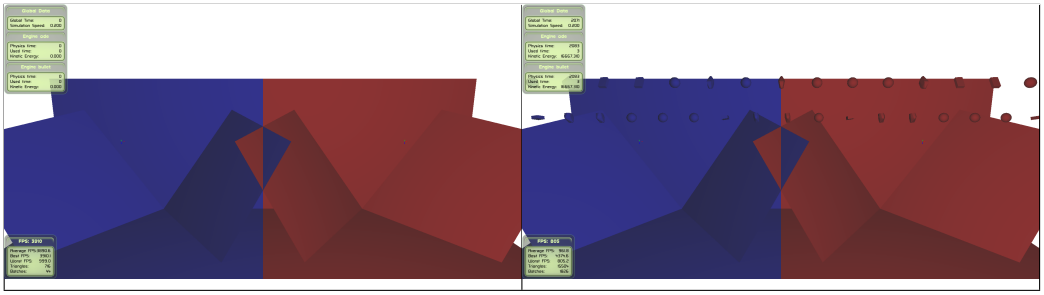


Figura 5.8: Prueba 4. Situación inicial y caída de cuerpos

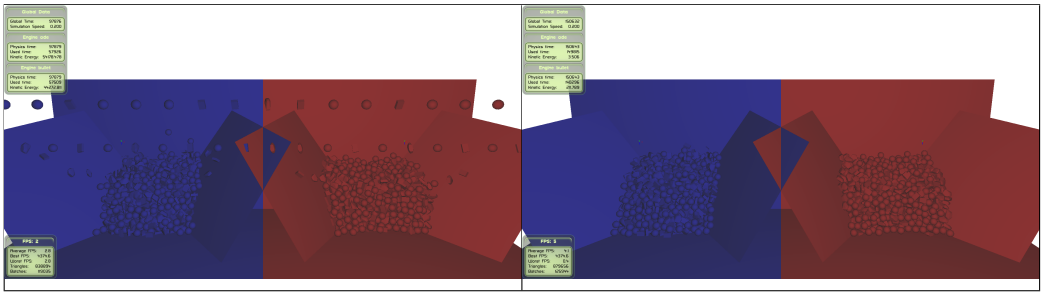


Figura 5.9: Prueba 4. Situación final

En ninguna de las ejecuciones realizadas se han observado comportamientos indeseados, y por lo tanto el único dato relevante es el rendimiento. Según se ve en la Figura 5.10 el rendimiento es similar en ambos motores, con una ligera ventaja para Bullet.

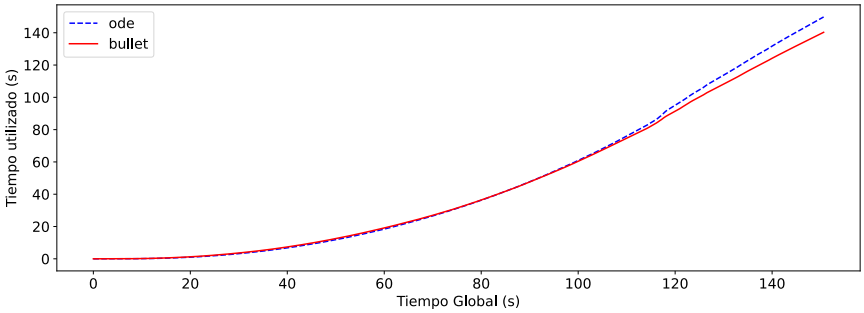


Figura 5.10: Prueba 4. Rendimiento

Tabla 5.8: Prueba 4. Resumen

Propiedad	**Motor**	**Valor**
Plausibilidad	ODE	Alta
	Bullet	Alta
Precisión física	ODE	-
	Bullet	-
Rendimiento	ODE	1,00
	Bullet	1,07
Estabilidad/robustez	ODE	Alta
	Bullet	Alta

5.2.7. Prueba 5. Torre

Para la última prueba se ha decidido un ejercicio de estrés clásico de motores físicos, que es una torre de varias alturas. En este caso se ha creado un edificio de 50 alturas, con seis ladrillos por altura, y además una base y un techo, según se puede observar en la Figura 5.11. Una vez hayan pasado diez segundos, se lanzará una esfera que demolerá la torre. Se utilizará un coeficiente de restitución de 0,3 y uno de fricción de 1,0.

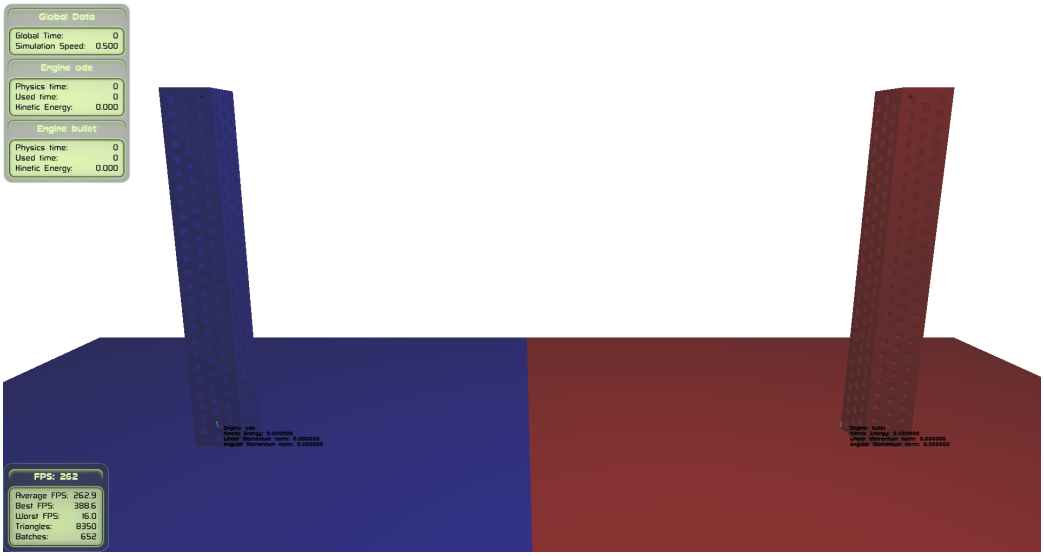


Figura 5.11: Situación inicial. Azul ODE, rojo Bullet

Se han realizado varias pruebas con distintas configuraciones de torres, que por espacio solo se comentarán superficialmente. Con sesenta ladrillos la torre de ODE es muy inestable. En el caso de Bullet, la torre presenta problemas visibles de vibración ya desde los diez ladrillos. En cuanto al coeficiente de restitución, si este tiene un valor alto (por ejemplo 0,9) la torre literalmente explota.

Se puede ver que con cincuenta ladrillos en la configuración básica la torre se termina cayendo tanto en ODE como en Bullet. En la Figura 5.12 se representa la energía cinética, en la que no se demuele la torre con una bola. En Bullet hay fuertes vibraciones desde el principio. En ODE es más estable, aunque al final cae. En el caso de Bullet no se consiguen eliminar totalmente las vibraciones ni disminuyendo el paso de integración ni probando otros métodos de resolución del MLCP, ya sean iterativos o no. En Couman y skocznymrocznyp (s.f.) el autor de Bullet comenta el problema.

Es interesante observar que las ejecuciones son no deterministas. Hay varios motivos para esto, además de variables no inicializadas o errores, que son debidos al orden de resolución de las restricciones y contactos obtenidos de forma aleatoria en Bullet.

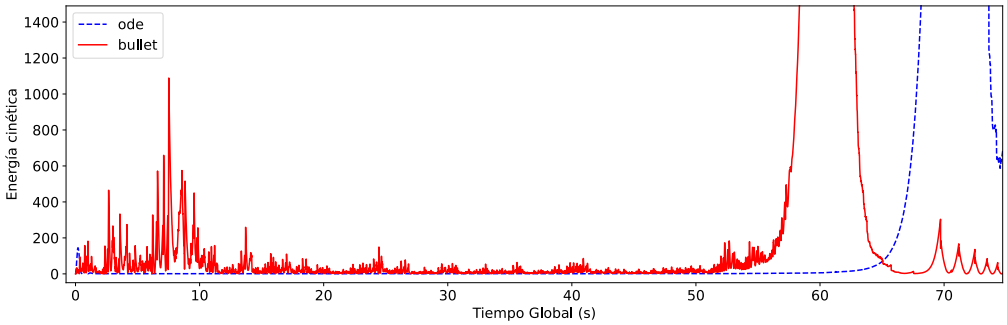


Figura 5.12: Prueba 5. Energía cinética de la torre sin bola de demolición. El mayor aumento de energía cinética corresponde con la caída de las torres.

En la Figura 5.13 se puede apreciar el efecto de la bola de demolición, con un efecto visual muy bueno en ambos motores. En la Figura 5.14 se comprueba que ODE es algo más lento mientras la torre está levantada, y un poco más rápido cuando está caída en el suelo.

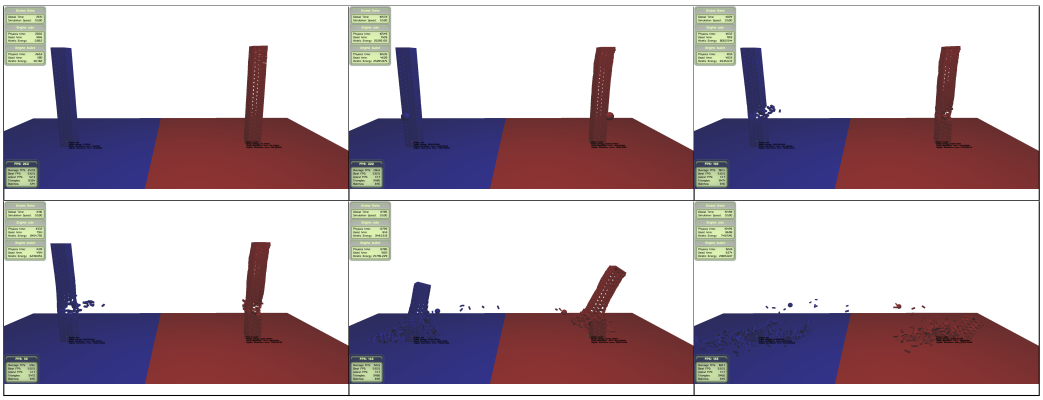


Figura 5.13: Prueba 5. Evolución

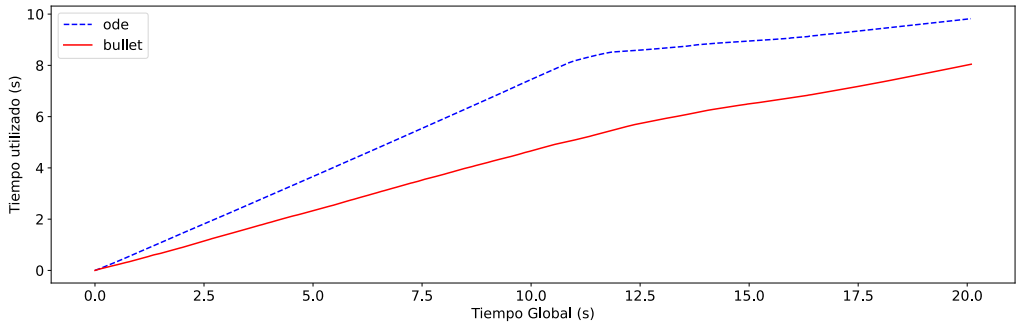


Figura 5.14: Prueba 5. Rendimiento. Se observa como una vez la bola de demolición golpea la torre el rendimiento de ODE aumenta, mientras que en Bullet se mantiene aproximadamente constante.

Tabla 5.9: Prueba 5. Resumen

Propiedad	**Motor**	**Valor**	**Comentarios**
Plausibilidad	ODE	Alta	
	Bullet	Media	La demolición de la torre es visualmente buena, pero la torre vibra
Precisión física	ODE	-	
	Bullet	-	
Rendimiento	ODE	0,51	ODE es más lento mientras la torre está levantada, y un poco más rápido cuando los ladrillos están en el suelo
	Bullet	0,39	
Estabilidad/ robustez	ODE	Media	Los ladrillos se descolocan poco a poco hasta que la torre cae
	Bullet	Baja	La torre vibra desde ya unas pocas alturas

5.2.8. Resultado benchmark

La Tabla 5.10 resume el resultado de las cinco pruebas prácticas realizadas. Los datos con comentarios se encuentran en las Tablas 5.5, 5.6, 5.7, 5.8 y 5.9. Se recuerda que un número más alto en rendimiento es positivo.

Tabla 5.10: Resumen benchmark

Prueba	**Plausibilidad**		**Precisión física**		**Rendimiento**		**Estabilidad/ robustez**	
	ODE	**Bullet**	**ODE**	**Bullet**	**ODE**	**Bullet**	**ODE**	**Bullet**
Prueba 1	Alta	Alta	Baja	Media	5831	4918	Alta	Alta
Prueba 2	Alta	Alta	Alta	Alta	1080	832	Alta	Alta
Prueba 3	Media	Baja	Media	Media	205	342	Alta	Media
Prueba 4	Alta	Alta	-	-	1,00	1,07	Alta	Alta
Prueba 5	Alta	Media	-	-	0,51	0,39	Media	Baja

5.3. Conclusiones

Se han comparado ODE y Bullet desde dos puntos de vista, uno basándose en el estudio de los motores y sus algoritmos y otro realizando pruebas prácticas.

La complejidad de la temática y la falta de documentación han supuesto un reto a la hora de estudiar el código y la documentación de los motores. El código fuente de ODE ha resultado muy complicado de entender y en el caso de Bullet la falta de documentación y la multitud de funcionalidades implementadas, algunas de ellas sin terminar, han dificultado hacer un resumen fidedigno.

En el caso de las pruebas prácticas, aunque han sido seleccionado siguiendo unos criterios, no dejan de caer en la subjetividad del autor y ser un número demasiado reducido para un análisis completo. Los motores se han probado en su configuración básica, no siendo en todos los casos la más apropiada. En el caso de Bullet esto es una clara desventaja, al no haber hecho la prueba 3 en coordenadas generalizadas.

Bullet y ODE implementan muchos de los principales algoritmos en el ámbito de la simulación del cuerpo rígido y la detección de colisiones. Sin

embargo, es Bullet por tener la posibilidad de usar dos paradigmas de simulación y por el uso de predicción especulativa de contactos y motion clamping el que abarca un espectro más grande de opciones.

En las pruebas prácticas no hay claro ganador, apreciándose que cada motor físico tiene escenarios en los que es superior al otro, y otros casos en los que es indiferente utilizar uno u otro. Destaca la prueba 5, donde la torre es inestable en Bullet con pocas alturas, mientras que en ODE se comporta de forma más razonable.

Ambos motores son apropiados para aplicaciones como animación, videojuegos o robótica y de hecho ambos son utilizados ampliamente estos tres ámbitos. Bullet, gracias a disponer de simulación de cuerpos deformables en el caso de la animación y los videojuegos y de coordenadas generalizadas en el caso de la robótica, parece estar un paso por delante de ODE.

Por lo anterior, y muy especialmente por estar desarrollado activamente, Bullet sería a criterio del autor la opción principal como motor físico en caso de duda. No obstante, y como demuestra la prueba 5, será el escenario y la aplicación lo que determine cuál es el motor más apropiado a utilizar en cada caso.

Capítulo 6

Mejora a los motores físicos. Algoritmo V-Clip.

En el enunciado del trabajo se solicita plantear y evaluar posibles mejoras en los motores físicos ODE y Bullet. Se plantean en este capítulo tanto algunas opciones de mejora como la elección finalmente implementada y que se evalúa en el capítulo siguiente.

6.1. Posibles opciones de mejora

El campo de la simulación del cuerpo rígido es muy amplio y la literatura extensa y especializada. ODE y Bullet son piezas software de gran tamaño y realizar un cambio de envergadura está fuera de las posibilidades de este trabajo. El repositorio de ODE contiene más de doscientas mil líneas de código y el de Bullet supera ampliamente el millón.

Una primera opción sería implementar o portar algoritmos de uno de los motores al otro. Así GJK/EPA no está implementado en ODE y sí que lo está en Bullet. Esta opción se ha descartado por dos motivos, la primera que no podría mejorar ambos motores a la vez y la segunda que en general su interés teórico sería limitado. Aun así se listarán algunas de estas opciones.

Por lo tanto, la mejora óptima sería aquella que tenga cierta complejidad pero que sea realizable en un tiempo razonable, que no esté implementada en ninguno de los motores y que realmente pueda mejorar una propiedad

medible de los motores.

Las posibles opciones se categorizarán en detección de colisiones, algoritmos de simulación y otras opciones. Este último apartado incluye opciones cercanas a la simulación pero que no forman parte del núcleo de un motor físico.

6.1.1. Detección de colisiones

- ODE y Bullet. ODE, para calcular el AABB de cuerpos convexos para la fase Broadphase, itera sobre todos los puntos del poliedro cada vez que este ha variado su posición. Este algoritmo es muy ineficiente. Se propone conseguir un mejor BV para el cuerpo y transformarlo según su posición y orientación. El cálculo de un buen BV para un cuerpo no es trivial y por ejemplo en Ericson (2004) se analizan distintas formas de obtenerlo.

  En el caso de Bullet, se calcula inicialmente el mejor AABB en coordenadas locales del cuerpo. Cuando es necesario el AABB en coordenadas globales se calcula a partir del AABB en coordenadas locales, siendo un cálculo muy rápido. La calidad del AABB en coordenadas globales dependerá, entre otros factores, del buen ajuste del AABB en coordinadas locales. Al igual que en el caso de ODE, se trataría de obtener un BV inicial que ajuste mejor al cuerpo que el AABB obtenido en el sistema de coordinadas locales del cuerpo.

  Aunque esta opción no ha sido la finalmente seleccionada es probablemente la mayor mejora que se puede hacer a ODE.

- ODE y Bullet. Crear nuevas geometrías implícitas como elipsoides y sus algoritmos para detección de colisiones.

- ODE. Implementar un algoritmo Broadphase por división del modelo y compararlo con los implementados por división del espacio.

- ODE y Bullet. Implementar estructura de datos KD-Trees para fase Broadphase.

- ODE y Bullet. Implementar el algoritmo V-Clip para la detección de colisiones Narrowphase entre poliedros convexos.

- ODE y Bullet. Crear un nuevo objeto cóncavo de forma implícita. Con este objeto sería posible realizar simulaciones como la curva de descenso

más rápido (curva braquistócrona). Sería también necesario incrementar el orden del algoritmo de integración.

Aunque esta opción no ha sido la finalmente seleccionada, es un campo de investigación sobre el que el autor no ha encontrado mucha literatura y que permitiría simular con mayor precisión muchos escenarios. Actualmente, una esfera descendiendo por una rampa cóncava presentará penetraciones de forma inevitable, ya que el orden de la geometría es mayor que el del paso de integración.

6.1.2. Algoritmos de simulación

- ODE. Implementar contactos predictivos especulativos para eliminar el problema de tunneling al igual que se hace en Bullet. Sería necesario implementar un algoritmo CCD al estilo Convex Cast de Bullet.
- ODE. Utilizar un cono de fricción no linealizada.
- ODE y Bullet. Modificar el paradigma de restricciones basadas en velocidad para que sea basado en restricciones de posición. Esta opción es muy compleja, siendo probablemente preferible escribir un motor físico de cero.
- ODE y Bullet. Crear nuevos tipos de restricciones (joints).
- ODE y Bullet. Implementar el algoritmo de «Linear-Time Dynamics Using Lagrange Multipliers» descrito en Baraff (1996). Idea tomada de https://pybullet.org/Bullet/phpBB3/viewtopic.php?t=9452.

6.1.3. Otras opciones

- ODE y Bullet. Examinar más algoritmos para la descomposición de cuerpos cóncavos a convexos.
- ODE y Bullet. Implementar algoritmo de Eberly para el cálculo de la matriz de inercia en poliedros convexos (Eberly, 2003).
- ODE y Bullet. Retomar la interfaz PAL (Boeing, s.f.) o crear una similar.

  Esta opción, aunque no es la seleccionada, se ha realizado de forma experimental e incompleta con la herramienta Obugre.

6.2. Opción seleccionada

La opción de mejora elegida ha sido el algoritmo V-Clip (Mirtich, 1998), que permite la detección de colisiones ofreciendo testigos, por lo que se puede obtener una estimación de distancias entre politopos convexos.

Hay varias razones por las que se ha escogido este algoritmo:

- La patente expiró en 2017-08-29, por lo que no hay ninguna restricción (ver https://patents.google.com/patent/US6054997).
- El autor no ha encontrado ninguna implementación de V-Clip utilizada en un motor físico, siendo un algoritmo basado en regiones de Voronoi en vez de los habituales basados en CSO.
- Desde luego, no está implementado ni en ODE ni en Bullet.
- Mirtich en la descripción del algoritmo indica que es más rápido y robusto que las versiones evaluadas del algoritmo GJK (Mirtich, 1998). Para ello, además de buscar mínimos locales utiliza coherencia temporal.
- Aunque no da la mínima interpenetración entre cuerpos, sí que es capaz de detectar un testigo de la penetración.
- En el foro de Bullet se encuentra como posible mejora (ver https://pybullet.org/Bullet/phpBB3/viewtopic.php?t=11034 y https://github.com/bulletphysics/bullet3/issues/563).

El autor de este trabajo ha encontrado dos implementaciones del algoritmo V-Clip. La primera la original de Mirtich junto con su versión en Java (Lloyd, s.f.). La otra versión se encontró una vez terminada la implementación de este proyecto, y se encuentra dentro del proyecto OpenTissue (Erleben, s.f.). Ninguna de estas implementaciones se ha consultado para la implementación de este proyecto, basándose exclusivamente en el artículo original, Mirtich (1998).

Además de las posibles ventajas, V-Clip también tiene una gran desventaja, y es que, en el caso de penetración, aunque el algoritmo ofrece un testigo de la penetración, este es arbitrario (generalmente un vértice y una cara o un lado y una cara). Con estos testigos, la medida de penetración no estará

bien definida y puede no ser continua al cambiar la pose de los objetos, a diferencia por ejemplo de los métodos basados en CSO. Se analizará el efecto de este problema en la simulación del cuerpo rígido. Otra desventaja frente a MPR y GJK/EPA es que el algoritmo no se puede aplicar a cuerpos definidos de forma implícita.

6.3. Algoritmo V-Clip

Se hará aquí un breve resumen de los puntos más destacables del algoritmo. Para su descripción completa se puede consultar el documento original en Mirtich (1998), en el que se basa este apartado.

El algoritmo aplica a poliedros convexos. En este trabajo se utilizarán indistintamente los términos politopo y poliedro, aunque politopo también aplica a más de tres dimensiones. En tres dimensiones un politopo tiene tres tipos de características, caras, lados y vértices (ver Figura 6.1).

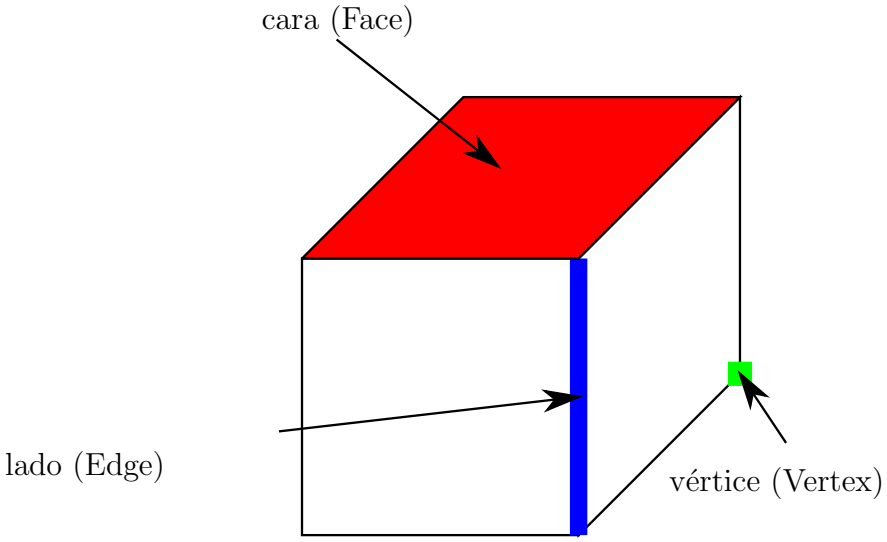
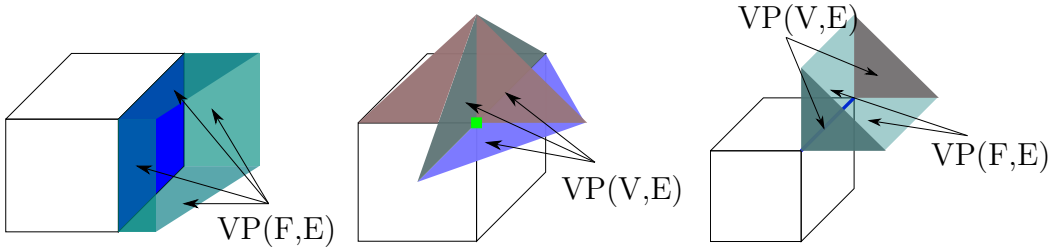


Figura 6.1: Características de un cubo

Definición 1. Se define la región de Voronoi de una característica X de un politopo convexo $VR(X)$ como el conjunto de puntos externos al politopo que están al menos tan cerca de X como de cualquier otra característica del politopo. El plano de Voronoi $VP(X, Y)$ entre las características vecinas X e Y es el plano formado por los puntos $VR(X) \cap VR(Y)$.



(a) Cara (b) Vértice (c) Lado

Figura 6.2: Regiones de Voronoi

La fundamentación del algoritmo es el concepto de regiones de Voronoi externas al poliedro, dados en la Definición 1. Como ejemplo, en la Figura 6.2 están representadas las regiones y planos de Voronoi para las tres características de un cubo.

Además, es necesario el siguiente teorema, demostrado en Mirtich (1998), y que es fundamental tanto para el algoritmo V-Clip como para el algoritmo Lin-Canny:

Teorema 1. *Sean X e Y un par de características de dos poliedros convexos disjuntos, y sean $x \in X$ y $y \in Y$ los puntos más cercanos entre X e Y. Si $x \in VR(Y)$ e $y \in VR(X)$, entonces x e y son globalmente los puntos más cercanos entre los poliedros.*

V-Clip realizará una búsqueda hacia las características que cumplan el Teorema 1, iterando por las características, generalmente vecinas, de los poliedros. En el caso de que la característica avanzada por V-Clip sea de dimensión menor (un vértice tiene dimensión cero, un lado dimensión uno y una cara dimensión dos), no se mejorará la distancia entre características de los dos poliedros, ya que se considerarán las características como conjuntos cerrados. En el caso de que V-Clip avance a una característica de dimensión superior, la distancia entre poliedros deberá mejorar.

Los estados posibles del algoritmo, que dependen de las características tratadas son vértice-vértice (V-V), vértice-lado (V-E), vértice-cara (V-F), lado-lado (E-E) y lado-cara (E-F). Las flechas continuas en la Figura 6.3 representan pasos a estados de mayor dimensión que encuentran características más próximas entre los politopos, mientras que las flechas discontinuas pasan a una dimensión menor y por lo tanto no se decrementa la distancia entre politopos. Al no haber bucles de flechas discontinuas, el algoritmo termina.

Además, el algoritmo V-Clip es capaz de detectar penetración entre cuerpos, tratando de forma especial casos de mínimos locales y dando un testigo de la penetración. El estado especial E-F solo podrá ser final si es un testigo de la penetración.

La coherencia se consigue recordando las características de la anterior invocación del algoritmo, de forma que se podría conseguir un orden algorítmico casi constante si la variación de las posiciones de los cuerpos es pequeña. Se deberá tener en cuenta que podría haber un coste de espacio de orden $O(n^2)$, siendo n el número de cuerpos, debido a la caché que almacena los testigos entre los pares de cuerpos.

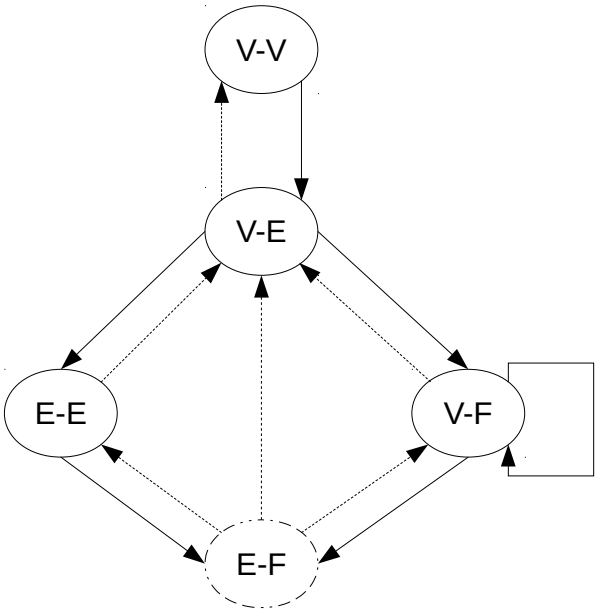
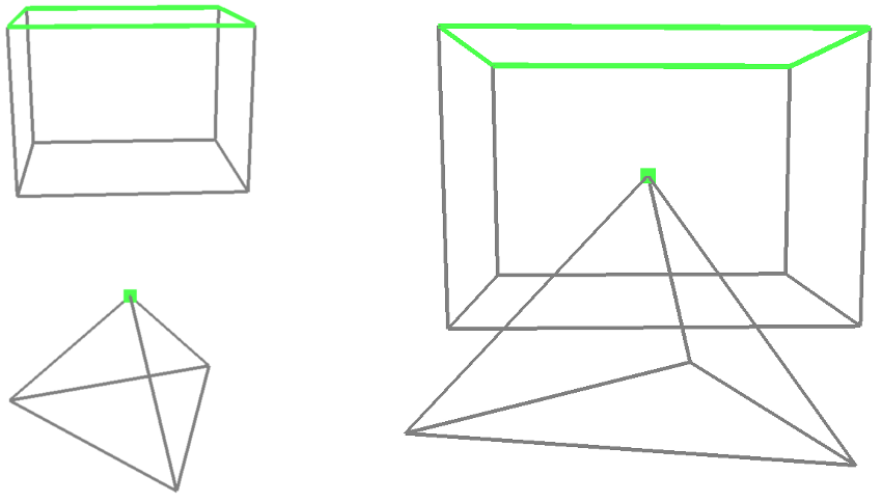


Figura 6.3: Estados del algoritmo V-Clip. Reproducido de Mirtich (1998)

La ventaja de V-Clip sobre Lin-Canny es que no realiza comparaciones de distancias entre características. En cada estado las operaciones realizadas consisten fundamentalmente en observar si se violan planos de Voronoi o en hacer edge-clipping (recorte de lados), que consiste en determinar si un lado interseca una región convexa delimitada por un conjunto de planos de Voronoi. Cuando hay intersección al realizar el edge-clipping, se puede determinar el nuevo estado analizando las derivadas en los puntos de corte, y si no hay intersección se trata según el tipo de exclusión.

Dentro de los estados de V-Clip hay un caso problemático de mínimo local en el estado V-F (ver Figura 6.4). En este caso, que se puede dar tanto con

penetración como sin ella, es imposible avanzar a una característica vecina ya que se incrementará la distancia entre las características de los poliedros. La solución dada por Mirtich consiste en buscar la cara de menor distancia al vértice, lo que implica iterar por todas las caras de un poliedro y supone un coste computacional $O(n)$, siendo n el número de caras del poliedro.



(a) Caso sin penetración (b) Caso con penetración

Figura 6.4: Mínimo local V-F

En la Figura 7.2 se puede ver un ejemplo completo de cómo V-Clip avanza por las características de los poliedros hasta encontrar las más cercanas. Esta ejecución es dependiente del orden de los poliedros y además la solución no es única en caso de penetración.

6.4. Implementación

Se comentan en este apartado las decisiones tomadas para la implementación del algoritmo junto a sus justificaciones y algunas de las experiencias obtenidas.

6.4.1. Estructura de datos de los politopos convexos

A diferencia de GJK, V-Clip requiere de una estructura de datos que represente la topología de los poliedros y permita avanzar con facilidad entre las características.

Existen dos estructuras de datos especialmente aptas para esta tarea que son winged-edge y half-edge, también llamada en la literatura DECL, Doubly Connected Edge List (Shirley & Marschner, 2009). Aunque conceptualmente similares, cada una de ellas tiene unas ventajas y desventajas. A la hora de implementar V-Clip la ventaja de winged-edge es que permite recorrer los lados de una cara en cualquier sentido con facilidad, mientras su mayor desventaja es que se debe comprobar continuamente el sentido recorrido de los lados.

En el caso de half-edge, recorrer los lados de una cara en sentido contrario a las agujas del reloj es trivial, mientras que es más costoso realizarlo en el sentido contrario, siendo necesario avanzar por todos los lados de cada vértice (ver Figura 6.5). En cuanto a los requisitos de memoria son iguales en ambas estructuras. En la estructura half-edge cada lado se representa dos veces (son lados dirigidos o half-edges), pero con la mitad de índices a otros lados dirigidos, caras o vértices.

El autor considera que cualquiera de las estructuras es apropiada, habiéndose escogido la estructura half-edge para evitar comprobaciones de sentido innecesarias. Como contrapartida, para implementar el apartado "3.3.3 Face Voronoi region exclusion" de Mirtich (1998) se han realizado variaciones en el algoritmo para no recorrer en sentido a las agujas del reloj los lados de una cara.

En el caso de half-edge la información necesaria (y por lo tanto los requisitos de memoria) es la siguiente:

- Cada vértice incluye las coordenadas del punto y un índice a un half-edge (el que está dirigido hacia este vértice).
- Cada half-edge (lado dirigido) dispone de un índice al vértice al que se dirige, dos índices a half-edges, uno a su par en la otra dirección y otra al siguiente half-edge en la misma cara siguiendo la dirección contraria a las agujas del reloj y otro índice a la cara que cubre.
- Cada cara contiene un índice a un half-edge de la cara.

Además, cada half-edge contendrá sus dos planos de Voronoi, uno el VP(F,E) con la cara asociada y el otro VP(V,E) con el vértice al que apunta. Los planos de Voronoi de los vértices y de las caras serán los mismos que los de los half-edges con las componentes invertidas. En el fichero `tfm_ode_bullet/obvclip/include/polytope.hpp` se encuentra la estructura de datos half-edge implementada. Se ha optado por utilizar valores numéricos en vez de punteros para representar los índices. Si el número de características es pequeño, se puede modificar el tipo `FeatureId` a 8 o 16 bits para conseguir un considerable ahorro en espacio.

6.4.2. Generación de politopos convexos a partir de puntos

Dados un conjunto de puntos no coplanares, generalmente obtenidos de una herramienta de diseño 3D, se pretende a partir de ellos generar la envoltura convexa. A partir de este poliedro convexo es necesario generar la estructura half-edge con la que funcionará el algoritmo V-Clip. Una opción intermedia que no se va a tratar en este trabajo es la posibilidad de descomponer un cuerpo cóncavo en cuerpos convexos.

Para obtener a partir de un conjunto de puntos de entrada su envoltura convexa se utilizará la herramienta QHull (http://www.qhull.org/), que utiliza el algoritmo Quickhull (Barber y col., 1996). La respuesta de QHull consiste en los planos de cada cara junto con los vértices no ordenados que componen esa cara.

En el fichero `obvclip/src/polytope.cpp` en la función `ConvexPolytope::generate_from_vertices`, los vértices se ordenan según el ángulo que forman en el plano de la cara sobre el centro de la cara y el primer vértice tratado. Con esta información es trivial generar la estructura half-edge, para posteriormente generar todos los planos de Voronoi sobre los half-edges.

Una vez se disponga de las caras, lados y vértices, se comprobará si cumplen con la característica de Euler, es decir:

$$V\acute{e}rtices - Lados + Caras = 2 \tag{6.1}$$

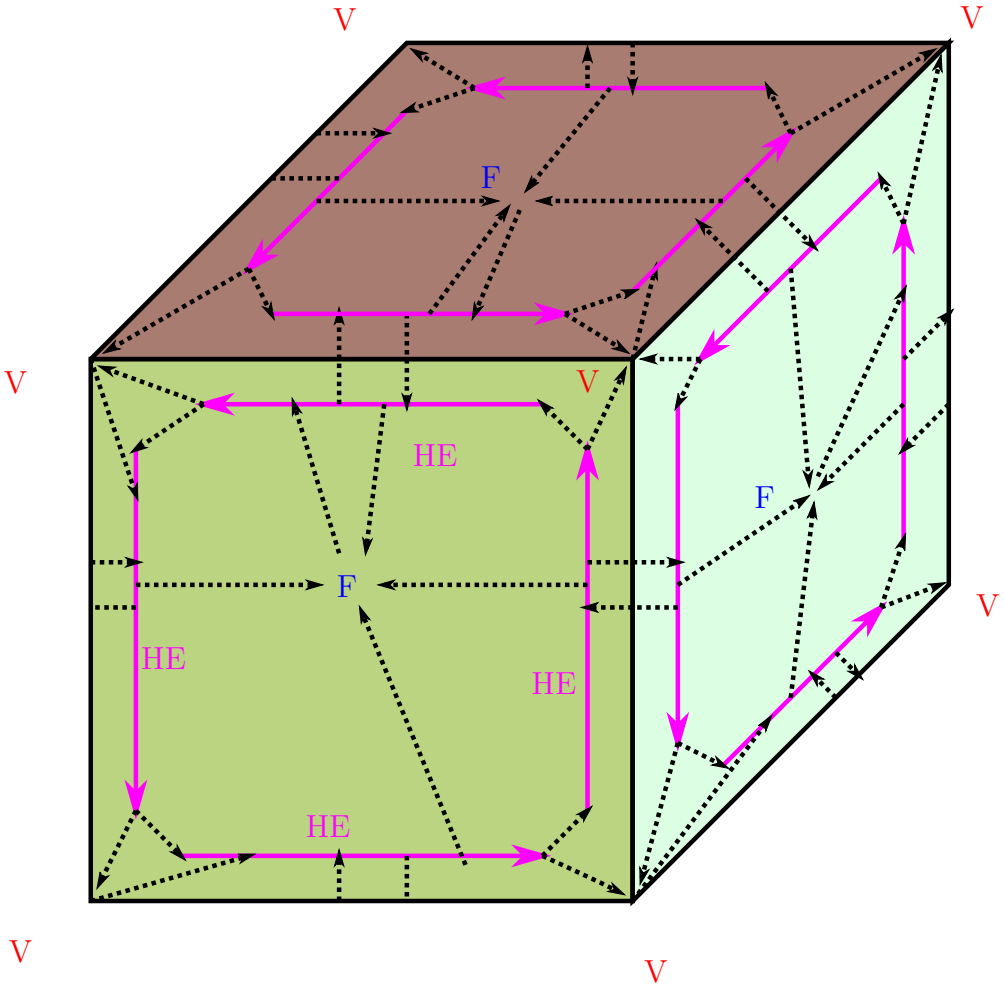


Figura 6.5: Estructura Half Edge. F son caras, V son vértices y HE half-edges. Cada F dispone de un puntero a uno de los HE, cada HE dispone de un puntero al siguiente HE, al HE de sentido contrario, a F y al V al que apunta el HE. Un V tiene un puntero a uno de los HE salientes. Recorrer los HE alrededor de una F en sentido contrario a las agujas del reloj es inmediato, mientras que recorrerlos en sentido contrario implica recorrer todos los HE que inciden en cada vértice de F.

6.4.3. Implementación del algoritmo

Inicialmente se optó por implementar una librería matemática desde cero. Para este proyecto es necesario trabajar con cuaternios, matrices de rotación, matrices homogéneas, y multitud de operaciones geométricas. Vistos

los problemas de depuración de la librería propia, se optó posteriormente por la librería Eigen (http://eigen.tuxfamily.org/). Aunque inicialmente el cambio aceleró considerablemente el desarrollo, posteriormente surgieron algunos problemas:

- La librería está basada en programación genérica y hace un uso intensivo de funciones y métodos inline. Sin optimización y en modo depuración la librería es extremadamente lenta, lo que el autor inicialmente achacó a la implementación del algoritmo.

- El autor no ha sido capaz de utilizar la representación `AffineCompact` para matrices homogéneas, lo que implica que la última fila de la matriz homogénea se guarda en memoria, lo que es un gasto innecesario de espacio, al ser todas las transformaciones isométricas.

Una vez creado un politopo convexo (clase `Polytope`), este podrá ser reutilizado en varios cuerpos con pose representados por la clase `WorldConvexPolytope`. El cuerpo con pose será la entrada al algoritmo junto con las características iniciales a partir de las cuales iterar.

Para no utilizar directamente la estructura half-edge y posibilitar el cambio de representación del politopo se optó por devolver vectores con los datos (por ejemplo los planos de Voronoi VP(V,E) de un vértice), y por lo tanto utilizando memoria dinámica. Posteriormente por los problemas de rendimiento, se cambió a un sistema basado en iteradores.

Debido a la utilización de una estructura half-edge, la implementación de la situación de un lado excluido de una región de Voronoi de una cara, apartado "3.3.3 Face Voronoi region exclusion" de Mirtich (1998), se ha cambiado. Se ha optado por recorrer el perímetro completo de la cara siempre en sentido contrario a las agujas del reloj, hasta encontrar una característica con derivada mínima, no estando el lado excluido de los VP(V,E) de la característica del perímetro y si es un lado que no esté excluido del VP(F,E) del lado probado contra la cara. No se han encontrado problemas de rendimiento por este caso.

En cuanto a la caché que permite almacenar las características que resultaron de la última ejecución de V-Clip para dos cuerpos rígidos (con pose), se ha implementado como un simple mapa en C++. Sería recomendable que la caché utilizara algún mecanismo de borrado para que no creciese demasiado.

Además de estas consideraciones, se han realizado modificaciones al algoritmo V-Clip que se han considerado erróneas o problemáticas. Se describen en la Sección 7.1.5.

6.4.4. Determinación de contactos

Una vez obtenido del algoritmo V-Clip las características más cercanas entre dos politopos o un testigo de la penetración es necesario obtener uno o varios posibles contactos que serán pasados al motor físico. Este problema se trata en detalle en Erleben y col. (2005, Capítulo 14).

Se han probado dos algoritmos, uno simple, que devuelve un solo punto de contacto y otro basado en "A Geometrical Algorithm" obtenido de Erleben y col. (2005, pp. 445-447). Para el algoritmo simple se ha optado por los siguientes métodos dependiendo del estado final del algoritmo V-Clip:

- En el caso V-V la normal tiene el sentido del segmento que los separa, y el punto de contacto el punto intermedio. En el caso de que los puntos coincidan, la normal calculada no será válida.

- En el caso V-E, la normal irá del punto más cercano del lado al vértice y el punto de contacto será el punto intermedio entre ellos. En el caso en que el vértice esté contenido en el lado, la normal será inválida.

- En el caso E-E se calcularán los puntos más cercanos de cada lado, y su diferencia será la normal y su punto intermedio el punto de contacto. Si los lados intersecan, la normal será inválida.

- En el caso V-F, la normal será la normal del plano de la cara y el punto de contacto el intermedio entre el vértice y la proyección del vértice en la cara y el vértice.

- En el caso E-F, que siempre corresponde a una penetración, se ha considerado indicar como normal la de la cara y el punto de contacto la intersección entre la cara y el lado. En el capítulo siguiente se comentarán todos los problemas derivados de este caso.

6.4.5. ODE

Por defecto ODE utiliza el algoritmo SAT para la detección de contactos en poliedros convexos, utilizando la función `dCollideConvexConvex`. Activando la librería libCCD, en la función `dCollideConvexConvexCCD`, se llama a una implementación del algoritmo MPR.

Las funciones por defecto para la gestión de colisiones según el tipo del objeto de geometría (se describen en un tipo enum en el fichero `collision.h`) se inicializan en la función `dInitColliders` en el fichero `collision_kernel.h`. Para sobrescribir una función de colisión se utiliza la función `dSetColliderOverride`, que en este caso se llamará como `dSetColliderOverride(dConvexClass, dConvexClass, &ODEdCollideConvexConvex);`.

La nueva función creada deberá devolver un array de contactos, de acuerdo con la estructura mostrada en el Listado 6.1. Siguiendo al manual de ODE la convención es que, si el primer cuerpo se mueve una distancia "depth" sobre la normal, entonces la distancia "depth" se reducirá a cero, es decir, que la normal apunta hacia el primer cuerpo. La variable "depth" siempre deberá ser positiva.

Listado 6.1: Estructura dContactGeom

```
struct dContactGeom {
    dVector3 pos;    // contact position
    dVector3 normal; // normal vector
    dReal depth;     // penetration depth.
    dGeomID g1,g2;   // the colliding geoms
};
```

Debido a los problemas para obtener una buena medida de penetración y un punto y una normal de máxima penetración, la utilización de V-Clip en este trabajo ha sido utilizando un margen. Para las geometrías convexas, ODE calcula el AABB en `dxConvex::computeAABB()`, y lo genera iterando sobre todos los puntos de la geometría. Además de muy lento e ineficiente, el filtrado por AABB no permitirá llamar al algoritmo V-Clip en ciertos casos en que, aunque no hay penetración sin el margen, con el margen sí que lo hay. Por lo tanto, esta función se ha modificado para tener en cuenta el margen y ampliar el AABB.

6.4.6. Bullet

La arquitectura de Bullet es considerablemente más complicada que la de ODE, y muy orientada al algoritmo GJK. Se recuerda aquí que una de las ventajas de GJK es su uso en cuerpos definidos implícitamente, ventaja que V-Clip no comparte.

La estructura `enum BroadphaseNativeTypes` define los tipos de elementos geométricos que pueden participar en la detección de colisiones. Al analizar la estructura, todos los primeros son formas geométricas poliédricas, a los que se les podría aplicar el algoritmo V-Clip.

Dentro del ejemplo `VoronoiFractureDemo.cpp` se incluye un ejemplo del uso de GJK+MPR que registra la creación de un nuevo algoritmo en la fase Narrowphase. La clase `btConvexConvexMprAlgorithm` se ha utilizado en este proyecto como prototipo para la utilización del algoritmo V-Clip en Bullet, y se ha realizado en la clase `btConvexConvexVclipAlgorithm`. La nueva clase se registrará en el `btCollisionDispatcher` con la función `registerCollisionCreateFunc` (ver Listado 6.2).

Dentro de `btConvexConvexVclipAlgorithm`, para resolver la colisión se buscará el politopo y se actualizará la pose, con lo que se podrá llamar al algoritmo V-Clip. No se ha incluido la estructura de datos del politopo en Bullet con su propio sistema de gestión de memoria, y en este trabajo se gestiona externamente en la librería Obugre. La inclusión de este código en Bullet sería el siguiente paso lógico que no se ha realizado en este trabajo.

Listado 6.2: Registro V-Clip en Bullet

```
create_func_vclip.reset(new btConvexConvexVclipAlgorithm::CreateFunc
                                {vclip_cache,
                                 shape_convex_hashmap}
                        );
btCollisionAlgorithmCreateFunc *cf = create_func_vclip.get();
dispatcher->registerCollisionCreateFunc(CONVEX_HULL_SHAPE_PROXYTYPE,
                                        CONVEX_HULL_SHAPE_PROXYTYPE, cf);
```

En cuanto a los contactos generados, el criterio de Bullet es ligeramente distinto al de ODE. La función `addContactPoint` permite añadir nuevos puntos de colisión. `normalOnBInWorld` es la normal que apunta del segundo objeto al primero. `pointInWorld` es el punto de colisión en el objeto B en coordenadas del mundo y el valor depth, de forma un poco confusa, es positivo cuando hay una colisión (ver Listado 6.3).

Listado 6.3: Bullet. Prototipo función addContactPoint

```
virtual void addContactPoint(const btVector3& normalOnBInWorld,
                             const btVector3& pointInWorld, btScalar depth);
```

Capítulo 7

V-Clip. Análisis y resultados de la mejora.

En esta sección se analizará y evaluará tanto el algoritmo V-Clip de una forma genérica como la implementación realizada, y posteriormente se evaluará al utilizarlo conjuntamente con ODE y Bullet. Será en su uso dentro de los motores físicos cuando se pueda comparar con los algoritmos implementados en ODE y Bullet, es decir, SAT, GJK/EPA y MPR.

Además de la herramienta creada para el Capítulo 5, se ha creado una herramienta gráfica para la visualización de las características y la evolución de V-Clip. Con la ayuda de ambas herramientas se han creado las figuras de este capítulo.

7.1. Análisis inicial

El autor considera que el algoritmo se ha implementado correctamente. No se ha encontrado ningún caso en que falle después de numerosas ejecuciones con multitud de cuerpos convexos, entre ellas muchas configuraciones singulares. Desde luego, al no ser un algoritmo trivial, no es posible garantizar la inexistencia de errores.

7.1.1. Problemas encontrados en la implementación.

La implementación robusta y con elevado rendimiento de algoritmos tales como Lin-Canny, V-Clip o GJK no es una tarea trivial. Los problemas más destacables que se han encontrado en la implementación de V-Clip son los siguientes:

- Ha sido necesario eliminar completamente todo el uso de memoria dinámica, y por lo tanto de la clase `std::vector`, ya que consumían la mayor parte del tiempo del algoritmo.
- Como se comentó en el capítulo anterior, la librería matemática utilizada Eigen funciona extremadamente lenta utilizando el compilador GCC con el flag "-01". Igualmente, el uso de algoritmos matemáticos no optimizados (por ejemplo, utilizando transformaciones afines generales en vez de isometrías) supone que el algoritmo se ralentice considerablemente.
- Ha sido necesario crear una herramienta gráfica para poder depurar el algoritmo.
- Además, se han encontrado dos problemas con la definición del algoritmo original (Mirtich, 1998), que se describen a continuación.

Problemas detectados en el algoritmo original.

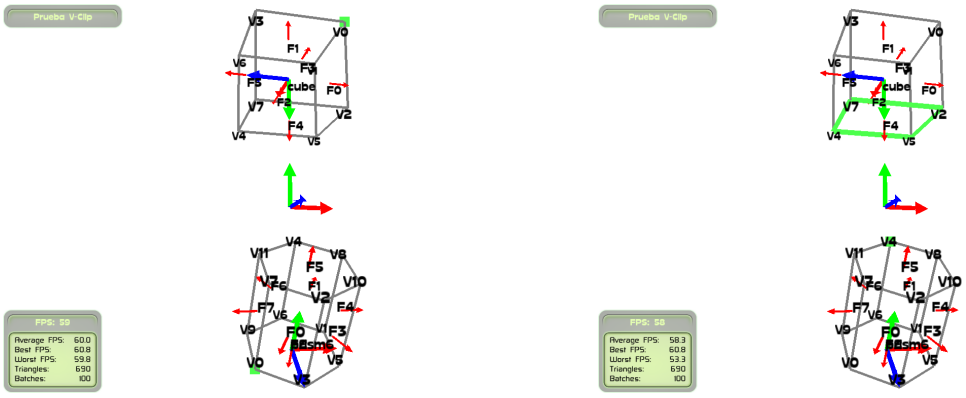
El primer problema se encuentra en "algorithm 6", línea 6, Mirtich (1998). Este caso corresponde al estado vértice-cara. En esta línea se busca un lado que parta del vértice y que mejore la distancia con respecto a la cara. Para ello utiliza la distancia del vértice inicial (V) a la cara y la compara con la distancia del otro vértice (V') del lado probado a la cara. En el caso de que el lado penetre la cara, la condición $|D_P(V)| > |D_P(V')|$ puede ser errónea, al poder estar más lejos en valor absoluto el vértice V' que el V y sin embargo mejorarse la distancia al pasar al lado (estado final por ser lado-cara con penetración).

El segundo problema se encuentra en "algorithm 3, handleLocalMin", líneas 8-9, que se invoca desde el "algorithm 6", línea 12 Mirtich (1998). Este caso también se da en el estado vértice-cara, cuando el vértice está a una distancia negativa de la cara. En el caso de que se determine que hay

penetración, y que por lo tanto el vértice está dentro del otro politopo, si no se actualiza la característica, en la próxima ejecución de V-Clip podemos caer en el mismo estado, y el orden del algoritmo en este caso es de $O(n)$. Si se actualiza la cara a la más cercana encontrada el problema se soluciona.

7.1.2. Ejemplo sin penetración

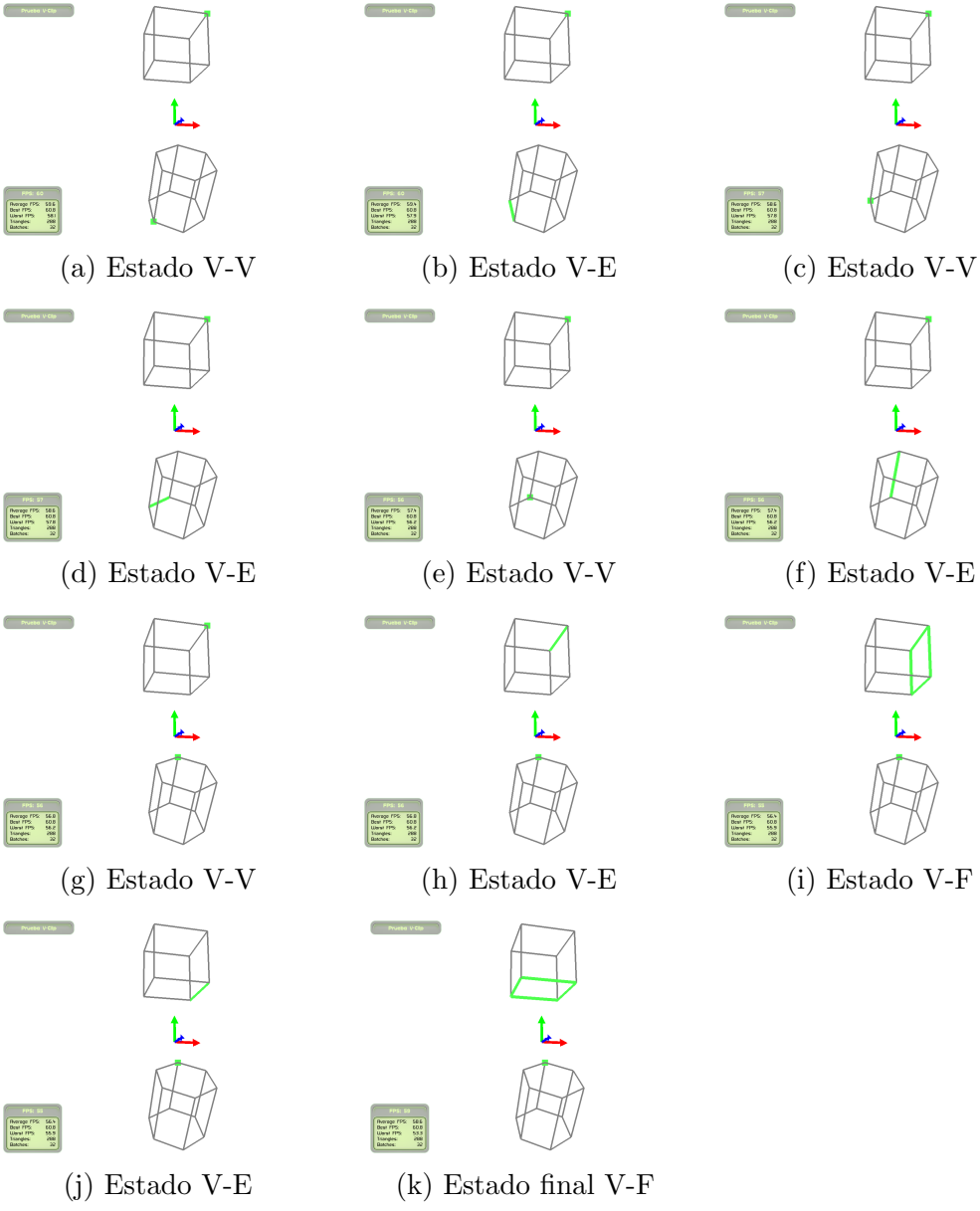
El ejemplo de la Figura 7.2 muestra como avanza el algoritmo V-Clip desde los vértices V0 de cada cuerpo (estado V-V) hasta el estado vértice-cara (V-F) que es la posición final. En las figuras de ejemplo de ejecuciones de V-Clip, las características del estado de V-Clip de cada cuerpo se pintarán con el color verde (no se debe confundir con el eje Y del cuerpo, que se representa con una flecha). Si la característica es un vértice, se representa como un punto sobre el vértice (en forma de cubo), si es un lado como una línea sobre el lado y si es una cara como líneas sobre todos los lados de esa cara. En la Figura 7.1 se muestran además en rojo para los estados inicial y final los nombres de los vértices y caras y las direcciones normales de las caras (no confundir con las flechas que representan los ejes X).



(a) Inicio (b) Fin

Figura 7.1: V-Clip. Posiciones inicial y final

Debido al funcionamiento de V-Clip, cada avance de estado disminuye la distancia entre características si se sube de dimensión o mantiene la distancia constante si se baja de dimensión, y el estado final representa claramente las características más cercanas (ver Figura 7.2).



(a) Estado V-V (b) Estado V-E (c) Estado V-V

(d) Estado V-E (e) Estado V-V (f) Estado V-E

(g) Estado V-V (h) Estado V-E (i) Estado V-F

(j) Estado V-E (k) Estado final V-F

Figura 7.2: V-Clip. Ejemplo sin penetración

7.1.3. Ejemplo con penetración

El siguiente ejemplo es uno con penetración, que parte desde un estado V-V y avanza hasta encontrar el estado E-F, que siempre indica penetración (ver Figura 7.3).

Además del estado E-F, todos los demás estados pueden indicar penetración. Los estados V-V, V-E, E-E solo podrán indicar penetración en el caso singular en que la distancia entre los cuerpos sea cero. El estado V-F, como de hecho ocurre en ejemplo de la siguiente sección, también puede indicar penetración sin distancia cero.

El problema de la penetración en V-Clip es que el estado final no es único y además puede depender del estado inicial. En este caso una pirámide cuadrada atraviesa un cubo en una de sus caras. Hay cuatro resultados posibles con estado final E-F, y en el caso de que el estado inicial fuera la cara superior del cuadrado y el ápex de la pirámide, acabaría en estado V-F.

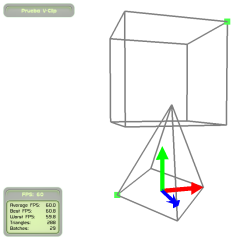
Además, no es posible dar una buena medida de penetración ni un punto de máxima penetración de una forma sencilla con el algoritmo V-Clip. En estado final E-F, la intersección de las características tendrá forzosamente distancia cero. Parte del lado en el lado negativo del plano de la cara podría incluso estar fuera del politopo con el que interseca.

7.1.4. Ejemplo con un cuerpo dentro de otro

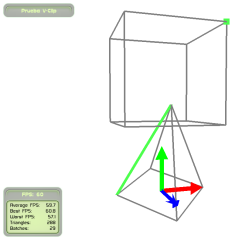
En el caso de que un poliedro esté totalmente dentro de otro poliedro V-Clip también es capaz de indicar la penetración, dando como resultado el estado V-F. Una vez en el estado final se deberán recorrer todas las caras del poliedro exterior, lo que implica un coste computacional $O(n)$ cada vez que se ejecute el algoritmo. Además, el estado final V-F será un mínimo local, no necesariamente un mínimo global (ver Figura 7.4).

7.1.5. Problemas en la detección de colisiones

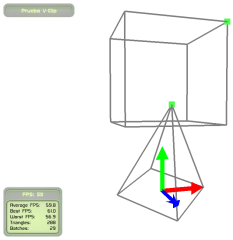
El mayor problema que se da en los contactos generados a partir de la salida del algoritmo V-Clip es en el caso de penetración. Al no disponer de una medida de penetración coherente (como por ejemplo la basada en CSO), y



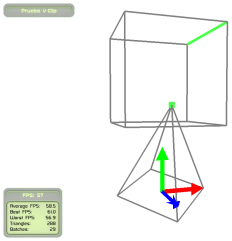
(a) Estado V-V



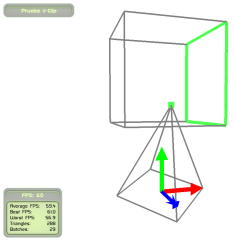
(b) Estado V-E



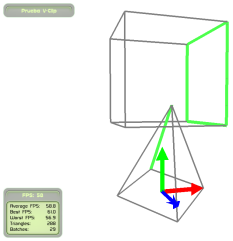
(c) Estado V-V



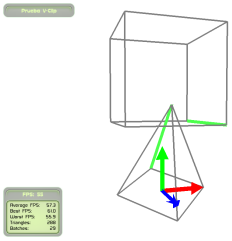
(d) Estado V-E



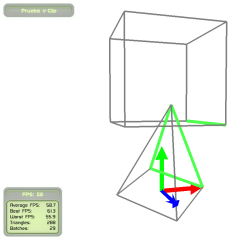
(e) Estado V-F



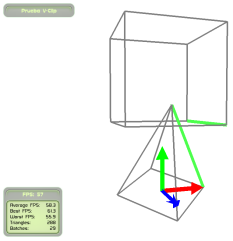
(f) Estado E-F



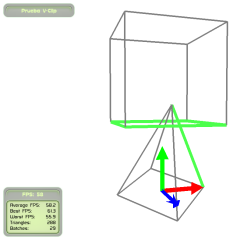
(g) Estado E-E



(h) Estado E-F

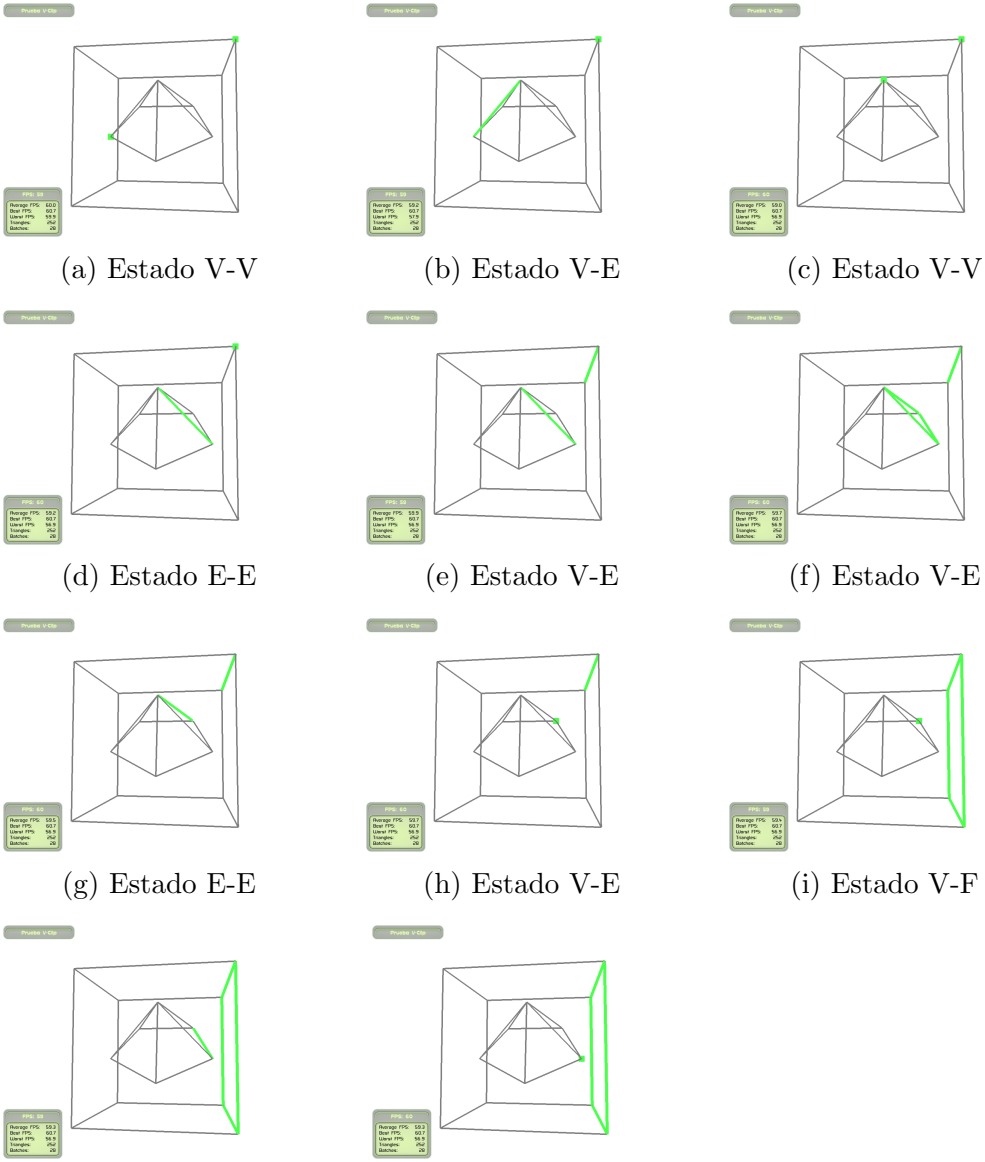


(i) Estado E-E



(j) Estado final E-F

Figura 7.3: V-Clip. Ejemplo con penetración



(a) Estado V-V (b) Estado V-E (c) Estado V-V

(d) Estado E-E (e) Estado V-E (f) Estado V-E

(g) Estado E-E (h) Estado V-E (i) Estado V-F

(j) Estado E-F (k) Estado final V-F

Figura 7.4: V-Clip. Ejemplo con cuerpo interior

disponer exclusivamente de un testigo de contacto, la normal generada puede no resultar la mejor para separar los cuerpos y la medida de interpenetración casi arbitraria. Un ejemplo especialmente problemático es cuando dos cuerpos penetran, uno de ellos girando en dirección perpendicular a la normal del contacto (ver Figura 7.5). En este caso los cuerpos siguen incrementando la penetración dando lugar a situaciones no deseadas. Además, tanto las normales como los puntos de contacto pueden variar de forma brusca.

La solución a estos problemas sería utilizar un algoritmo que calcule un mejor punto de contacto, normal y distancia, tal como EPA o MPR. Estos algoritmos podrían ser inicializado con el testigo de penetración obtenido con V-Clip.

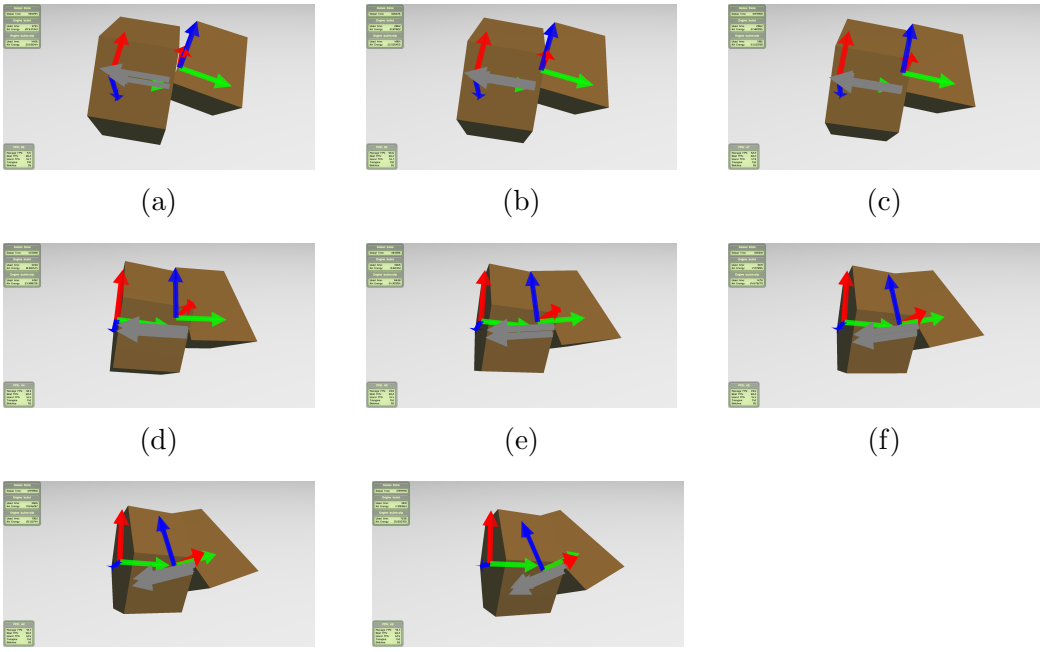


Figura 7.5: V-Clip. Problema interpenetración con cuerpos girando. El cuerpo de la derecha tiene una velocidad angular dirigida hacia afuera de la página. En (a) y (b), los contactos generados y las normales son coherentes con la configuración geométrica, pero en (c) y (d) se observa que los contactos y normales dibujados facilitan la interpenetración de los cuerpos en vez de su separación. La situación en (e), (f), (g) y (h) es ya patológica, y los contactos y sus normales, en vez de evitar la interpenetración, hacen de eje de giro de un cuerpo sobre otro.

En caso de no haber penetración y según se explica en Erleben y col.

(2005, Capítulo 14), las normales no siempre están claramente definidas. En este trabajo se ha optado para las normales utilizar la dirección de los puntos más cercanos.

Ya que se ha considerado que hay un contacto cuando los cuerpos se encuentran a una distancia menor de un margen, en realidad lo que se ha hecho es "redondear" los cuerpos, o más técnicamente, añadir en el espacio de Minkowski de cada cuerpo una esfera de radio la mitad del margen. De esta forma se decrementa considerablemente el número de penetraciones y se evitan problemas numéricos que no permiten detectar la colisión justo en el momento de contacto.

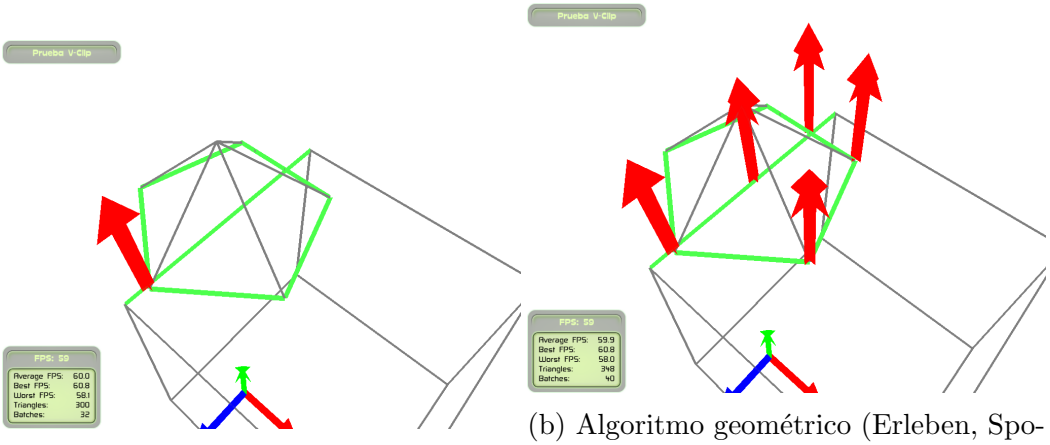
En el caso de que la distancia entre cuerpos sea cero, con esta forma de cálculo la normal estará indefinida en los estados V-V, V-E y E-E. En la práctica el autor no ha encontrado ningún problema en este caso.

7.1.6. Generación de contactos

El algoritmo geométrico de Erleben y col. (2005) se ha probado y descartado, al no suponer, al menos visualmente, una mejora sobre el método más simple de cálculo de normales y puntos de contacto. En la Figura 7.6 se pueden apreciar los dos métodos comentados de cálculo de puntos de contacto. El primero de ellos, el algoritmo simple, dará como normal la del plano en los casos V-F y E-F y en los demás casos la dirección de los puntos más cercanos. El punto de contacto será el más cercano a los dos politopos y la penetración estará basada en la distancia entre cuerpos.

7.2. Análisis de rendimiento. ODE y Bullet

El análisis que se va a realizar de V-Clip va a ser fundamentalmente de rendimiento. En el caso sin penetración V-Clip da como resultado las características más cercanas entre dos cuerpos, de una forma exacta, por lo que no es posible que dé un peor resultado que GJK o MPR, que al ser iterativos y tener un límite de iteraciones pueden no dar el mejor resultado. El algoritmo implementado no ha dado ningún problema de estabilidad ni numérico. La única dificultad en el caso sin penetración es la determinación de contactos y normales, algo que no es inherente a V-Clip. El caso con penetración es



(a) Método simple

(b) Algoritmo geométrico (Erleben, Sporring, Henriksen & Dohlman, 2005)

Figura 7.6: V-Clip. Obtención de contactos

más complicado y aquí V-Clip sí genera situaciones indeseadas. Como se ha comentado, queda pendiente realizar una extensión en la que se utilice un algoritmo distinto para este caso.

Se va a utilizar el mismo escenario para el resto de las pruebas de este capítulo, basadas en el de la Sección 5.2.6. Se va a determinar que ha habido una colisión cuando la distancia entre cuerpos sea menor a 0,1. Esto es equivalente a un margen en Bullet de 0,05, siendo por defecto en Bullet 0,04 y por lo tanto muy similar.

Se lanzarán diez cuerpos convexos por segundo, de los siguientes tipos:

- La envoltura convexa de un conejo de 105 vértices. Esta se ha tomado del código fuente de ODE:
- Un cuerpo de 1000 puntos al azar en una esfera 3D generados con la herramienta RBox de QHull.
- Un prisma pentagonal.

Una vez lanzados 500 objetos, se esperará 10 segundos y se terminará la prueba. En las Figuras 7.7 y 7.8 se puede apreciar la configuración de la escena.

Un aspecto importante del funcionamiento del entorno es que ejecutar dos veces seguidas un mismo motor físico permite aprovechar las memorias

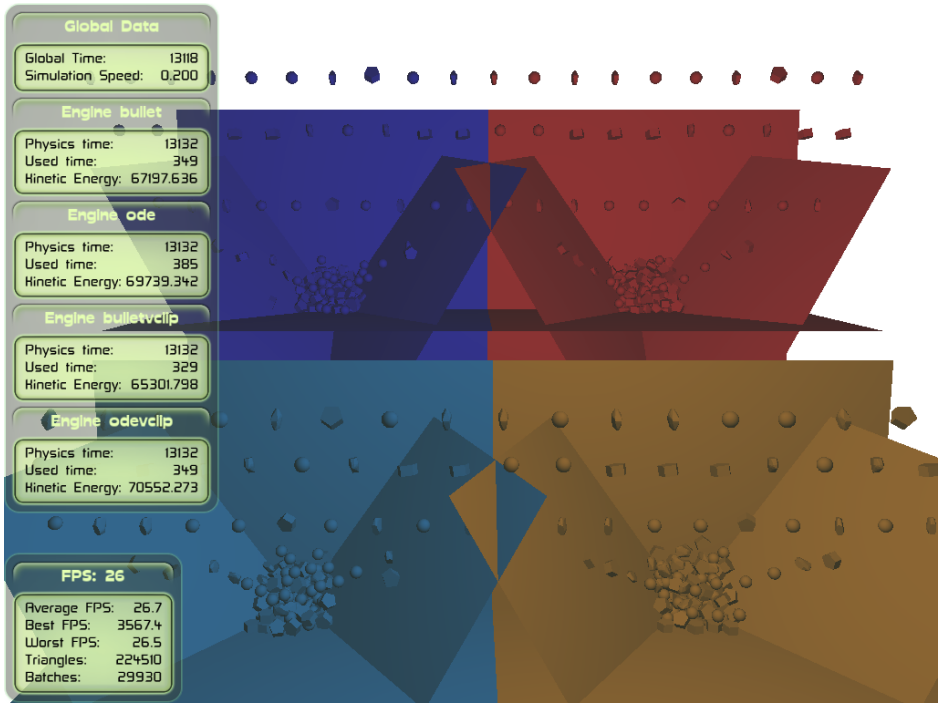


Figura 7.7: Cuerpos cayendo en prueba de rendimiento. Vista motores físicos

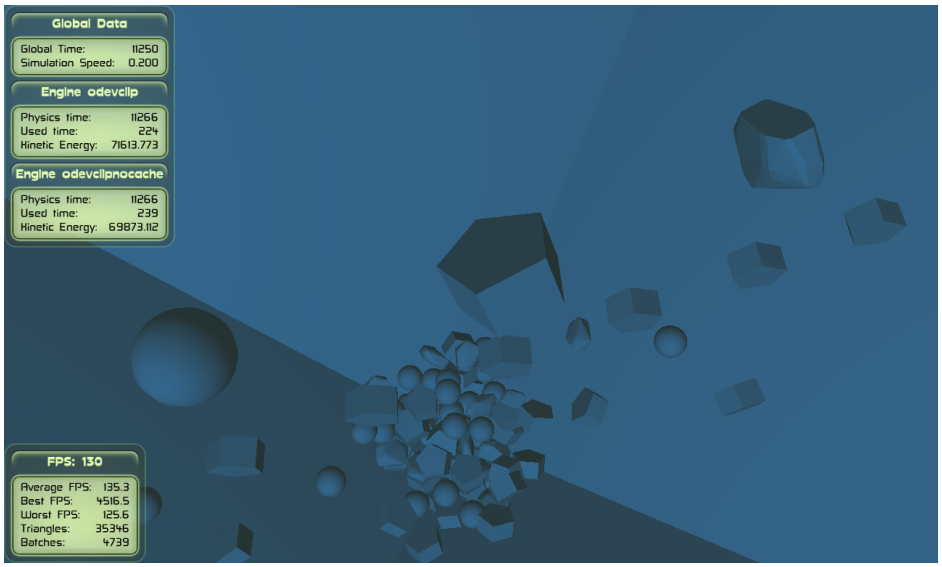


Figura 7.8: Cuerpos cayendo en prueba de rendimiento

caché de la CPU. En todas las pruebas se ha tenido en cuenta y se ha tratado de no favorecer ninguna configuración.

7.2.1. Rendimiento V-Clip con y sin caché en ODE.

En esta primera prueba se comparará la diferencia de utilizar V-Clip en ODE primero sin caché que guarde los testigos de la anterior ejecución y después con caché, es decir, haciendo uso de la coherencia temporal.

En la Figura 7.9 se muestra el tiempo medio necesario por cada ejecución de V-Clip en cada paso. Se observa como al hacer uso de caché este baja rápidamente a partir de las primeras ejecuciones, según va habiendo más cuerpos en la parte inferior del escenario. En el caso de sin caché el tiempo se mantiene estable, y de media más de cuatro veces superior a con el uso de caché.

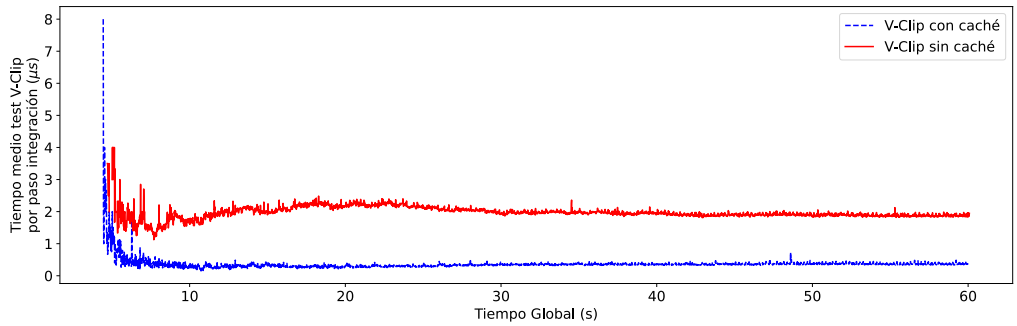


Figura 7.9: Tiempo medio ejecución V-Clip en cada paso con y sin caché en ODE

Es interesante observar como el número de ejecuciones de V-Clip por paso varía según se utilice o no la caché. Además de por motivos internos a ODE, la utilización de V-Clip con y sin caché dará lugar a distintos testigos en caso de penetración y por lo tanto a una evolución distinta del sistema (ver Figura 7.10).

El tiempo total es muy similar en los primeros segundos, a partir del cual se observa una clara ganancia en rendimiento al utilizar la caché (ver Figuras 7.11 y 7.12). No se ha analizado el uso de memoria, aunque su complejidad de espacio en el peor caso es de orden cuadrático con el número de cuerpos.

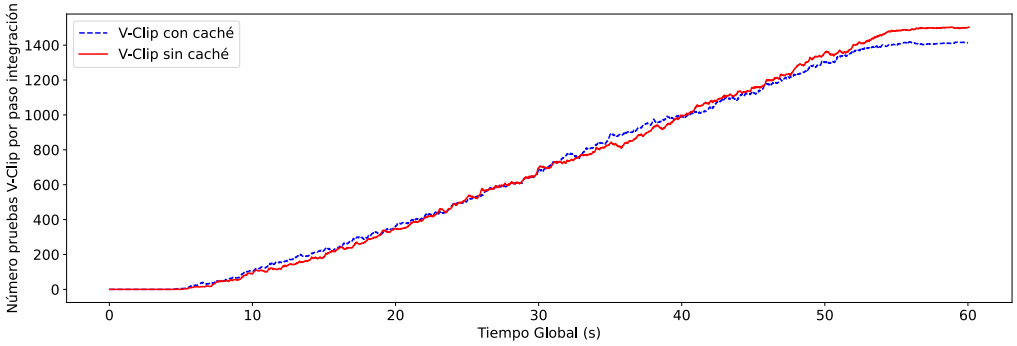


Figura 7.10: Número de ejecuciones de V-Clip por paso con y sin caché en ODE

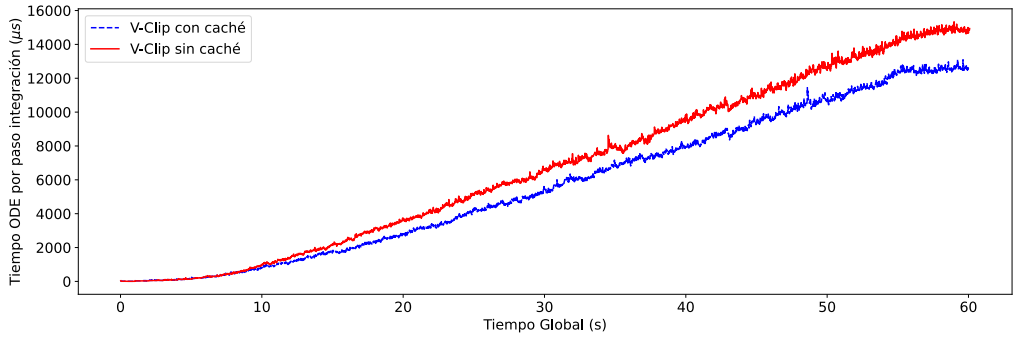


Figura 7.11: V-Clip. Tiempo usado por paso de integración con y sin caché en ODE

7.2.2. ODE

En este apartado se compara ODE utilizando la librería libCCD y por lo tanto con el algoritmo MPR, contra V-Clip utilizando caché. Es destacable que en la implementación utilizada por ODE no se utiliza ningún tipo de coherencia temporal.

Se ha probado también ODE utilizando el algoritmo SAT para cuerpos convexos, y como era previsible su rendimiento es muy inferior en el momento que el número de cuerpos crece al tener una complejidad algorítmica de $O(n^2)$. En concreto para esta prueba, ODE con SAT requirió más de 75 segundos, frente a menos de 25 segundos con MPR o V-Clip.

En la Figura 7.13 se observa como V-Clip es más rápido que MPR cuando el número de cuerpos aumenta y puede hacer uso de la coherencia temporal.

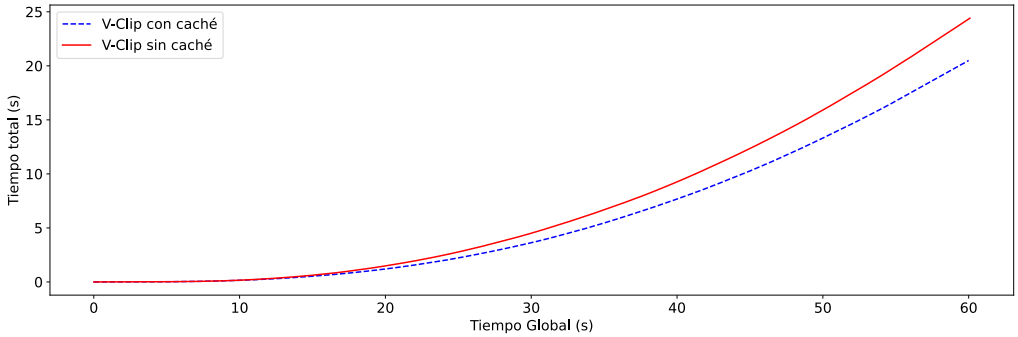


Figura 7.12: V-Clip. Tiempo total ODE con y sin caché

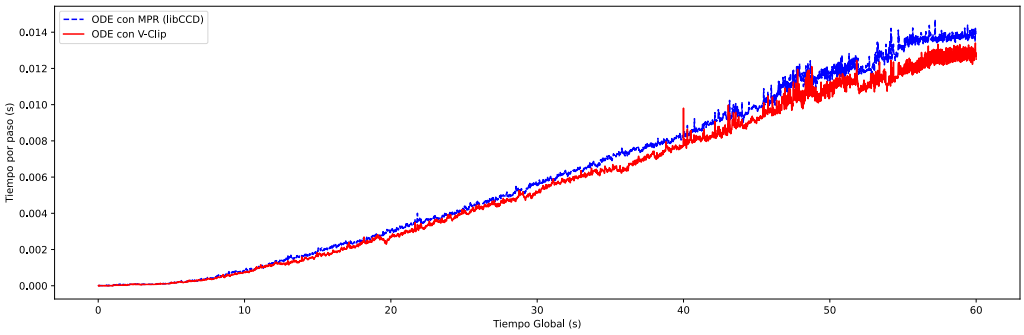


Figura 7.13: ODE V-Clip vs MPR. Tiempo por paso de integración

No se incluyen comparativas con la caché de V-Clip desactivada, en cuyo caso el tiempo de ejecución de V-Clip es ligeramente mayor que con MPR. Cuando termina la simulación, después de 60 segundos, V-Clip ha permitido reducir considerablemente el tiempo total de simulación (ver Figura 7.14).

Se debe de tener en cuenta que MPR utiliza un número fijo de iteraciones y por lo tanto no da un resultado exacto, aunque como ventaja es un algoritmo que trata de forma coherente el caso de la penetración, dando una medida coherente con CSO.

La simulación resultante es estable y los cuerpos tienen a parar una vez han dejado de caer cuerpos. Sin embargo, se observan casos de penetración en la simulación, como se puede apreciar en la Figura 7.15. Según se va aumentando el margen, el número de penetraciones disminuye.

Con el margen utilizado de 0,1, el número de penetraciones al final de la prueba se mantiene estable en torno a 5 por paso de integración, y a lo largo de toda la prueba menos del 1 % de las ejecuciones de V-Clip resulta

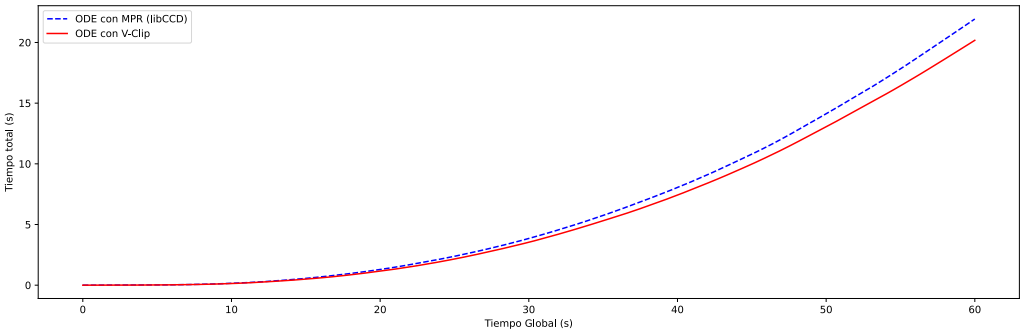


Figura 7.14: ODE V-Clip vs MPR. Tiempo total

en penetración. Aún sin haber realizado la prueba, es razonable concluir que incluso utilizando un algoritmo a mayores de V-Clip en caso de penetración, se conseguiría mejorar el rendimiento original de ODE.

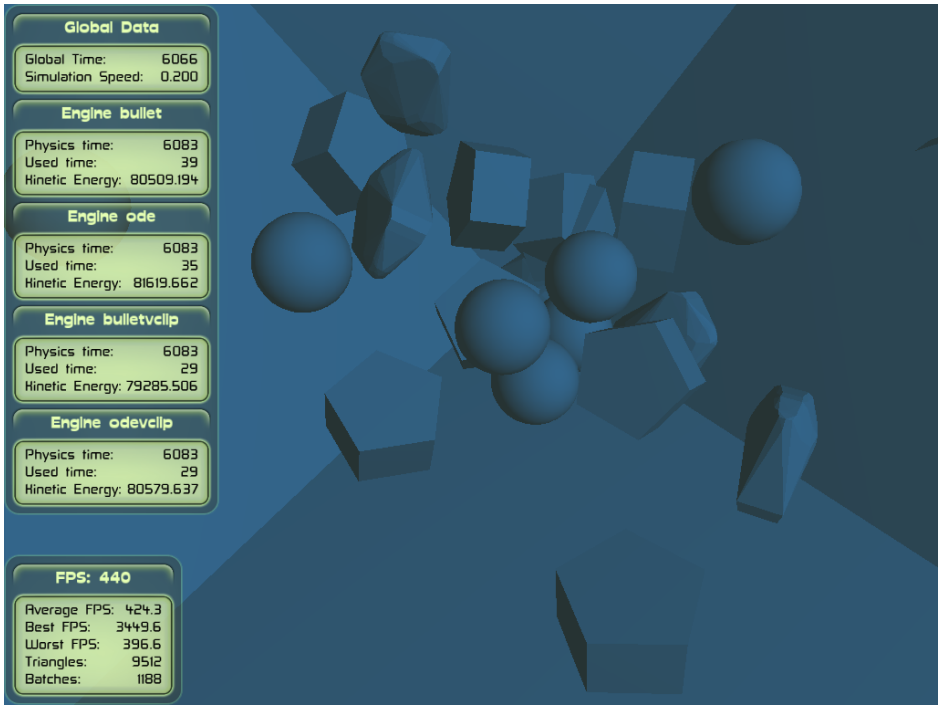


Figura 7.15: V-Clip ODE. Caso de penetración no deseable. Cerca del ápex de la pirámide invertida, siguiendo la arista superior derecha, se observan dos prismas interpenetrando en una configuración incorrecta. Según se aumenta el margen para la colisión el número de casos problemáticos disminuye, pero no llega a desaparecer.

7.2.3. Bullet

El algoritmo por defecto de Bullet está basado en GJK y en el caso de penetración utiliza el algoritmo EPA. Bullet utiliza un margen predefinido de 0,04 por cuerpo, por lo que de facto hay una colisión cuando hay una distancia de 0,08 entre dos cuerpos. De esta forma se evita utilizar en exceso el algoritmo EPA, que numéricamente es más inestable que GJK.

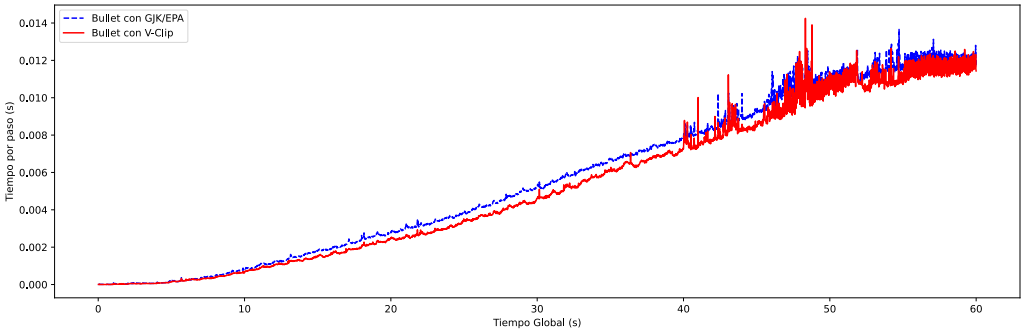


Figura 7.16: Bullet V-Clip vs GJK/EPA. Tiempo por paso de integración

Al igual que en el caso de ODE, V-Clip tiene un rendimiento superior al algoritmo original de Bullet (ver Figura 7.16) casi desde el inicio de la simulación. Una vez terminada la ejecución se consigue una mejora de rendimiento gracias al algoritmo V-Clip, aunque más moderada que en ODE (ver Figura 7.17).

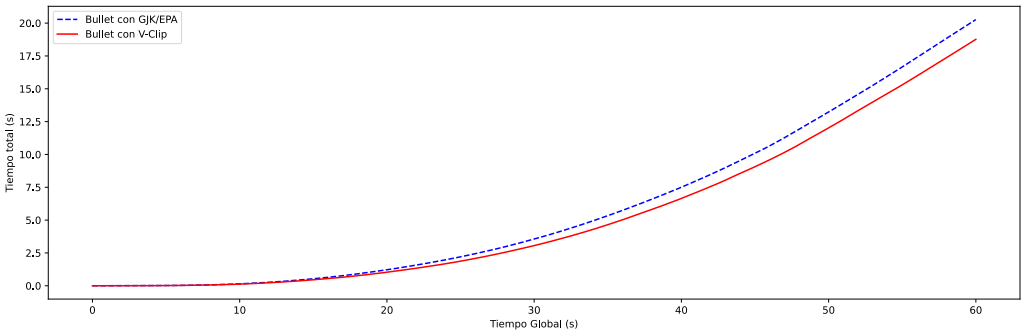


Figura 7.17: Bullet V-Clip vs GJK/EPA. Tiempo total

En cuanto a la estabilidad, los cuerpos vibran en la situación final con el margen de 0,10. Es necesario incrementarlo al menos otro 0,02 para evitar las vibraciones. Cabe destacar que Bullet, incluso en la versión sin V-Clip, tiene

en la situación final una energía cinética más elevada que en ODE, cuando esta debería ser cero, aunque no se perciban visualmente las vibraciones.

El número de penetraciones al final de la simulación está en torno a 5 por paso de integración. En la Figura 7.18 se puede observar una penetración en la simulación. El número total de invocaciones al algoritmo V-Clip es de aproximadamente seis millones, considerablemente superior al realizado en ODE, probablemente por el menor ajuste del AABB.

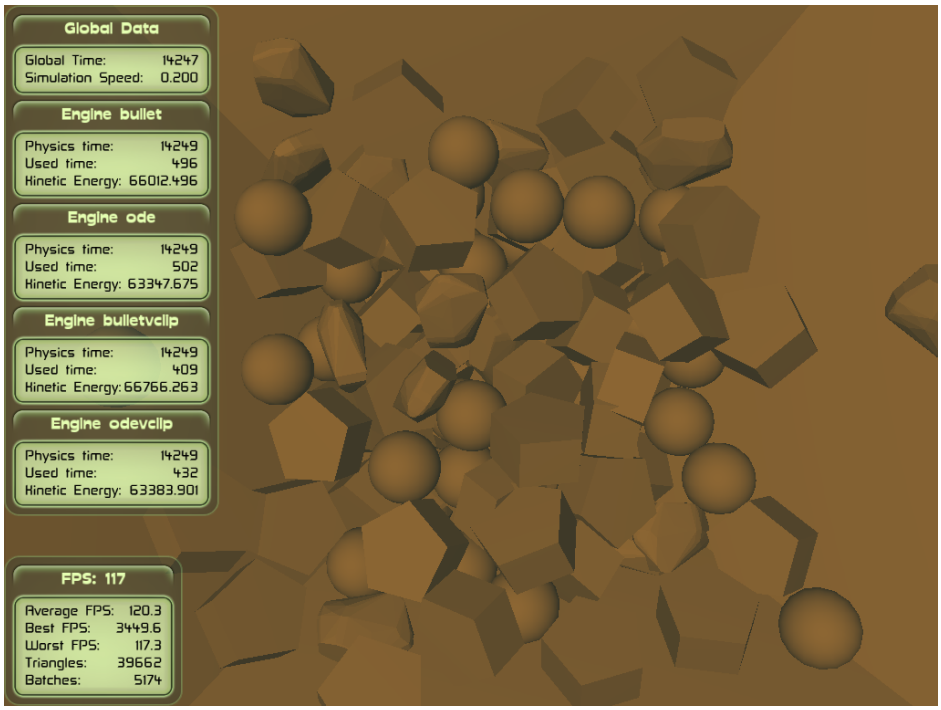


Figura 7.18: V-Clip Bullet. Caso de penetración no deseable. En la figura se puede apreciar un caso de dos prismas interpenetrando. Bullet mantiene contactos de una ejecución a la siguiente y en general ha resultado más inestable que ODE. En el caso de utilizar un margen menor de 0,12 el escenario no se estabiliza y vibra indefinidamente.

7.3. Resumen

En este capítulo se ha realizado analizado y evaluado el algoritmo V-Clip. Aunque el escenario de las pruebas es solo un caso de los muchos que se pueden probar, queda patente que la afirmación de Mirtich en la descripción

de su algoritmo original Mirtich (1998) es consistente con la realidad. V-Clip se compara favorablemente en rendimiento con GJK y con MPR y además no presenta problemas de estabilidad. Esta ventaja dependerá del uso de la coherencia temporal, del número de penetraciones y de la topología de los cuerpos.

El mayor problema es el caso de penetración, apartado que no se ha tratado y que supondría el uso de otro algoritmo. Debido a que gracias al margen el caso de penetración es infrecuente, es muy probable que este añadido no aumentaría significativamente el tiempo de ejecución.

Tanto ODE como Bullet disponen de implementaciones del algoritmo MPR que se pueden utilizar para obtener una medida de penetración. Este algoritmo requiere de un testigo, que se puede obtener a partir de la salida de V-Clip. En el caso de Bullet, el algoritmo EPA también permite obtener una medida de la penetración.

Concluimos en este trabajo que V-Clip, una vez complementado con MPR o EPA para el caso de penetración, es una mejora capaz de aumentar el rendimiento de los motores físicos sin introducir ningún problema de estabilidad.

Capítulo 8

Conclusiones y líneas futuras de investigación

8.1. Conclusiones

ODE y Bullet son los dos principales motores físicos de código abierto que simulan la dinámica del cuerpo rígido. Su estudio en este trabajo ha demostrado que son piezas de software intrincadas, capaces de simular escenarios complejos en tiempo real, con resultados plausibles y en algunos casos imitando con aceptable fidelidad la física newtoniana.

No existe un claro ganador en la comparativa entre ODE y Bullet, y cada uno de ellos exhibe unas fortalezas y debilidades. Sin embargo, Bullet tiene dos ventajas que le declaran como una mejor alternativa en caso de indecisión: está desarrollado activamente e implementa más paradigmas de simulación.

La mejora a ODE y Bullet analizada en este trabajo, el algoritmo V-Clip, ha demostrado ser un firme contendiente de GJK y MPR para la detección de colisiones entre poliedros convexos. El autor considera un éxito la implementación realizada de V-Clip. Se debe de tener en cuenta que los algoritmos de ODE y Bullet están muy optimizados y aun así la implementación de V-Clip ha conseguido, en el escenario probado, superar su rendimiento.

Como residuo del estudio de los motores físicos y de la implementación del algoritmo V-Clip se creado una herramienta, Obugre, que al estilo de PAL

trata de homogeneizar las interfaces de los motores físicos y al estilo de PEEL permite la visualización simultánea de varias instancias de motores físicos. Aunque aún está en un estado muy primitivo, ha simplificado notablemente la creación y análisis de escenarios de simulación en este proyecto.

8.2. Líneas futuras de investigación

Son muchas las líneas de investigación que amplían o extienden el análisis realizado en este trabajo.

En cuanto al análisis de posibilidades de motores físicos:

- Utilización de coordenadas generalizadas en Bullet.
- Creación sistemática de pruebas con variación de parámetros controlados por la herramienta, como paso y algoritmos de integración, algoritmos de resolución de LCP...
- Análisis del uso de memoria.
- Análisis de otros motores físicos, especialmente con distintos paradigmas de simulación.

En cuanto a mejoras a ODE y Bullet, implementar y evaluar las propuestas sugeridas en la Sección 6.1.

En cuanto a la implementación del algoritmo V-Clip:

- Utilización de otro algoritmo, como MPR o EPA, en caso de penetración. Incluso se podría analizar si es viable la creación de un nuevo algoritmo geométrico que extienda V-Clip haciendo uso de las regiones de Voronoi.
- Integración con el código de Bullet y su sistema de gestión de memoria.
- Cambio de la estructura de datos half-edge por winged-edge para comparar su rendimiento.
- Cambio de la librería Eigen por una librería matemática propia para comprobar si se puede mejorar el rendimiento.

- Estudio estadístico pormenorizado del tiempo y memoria utilizado por V-Clip y sus algoritmos rivales en distintas configuraciones y escenarios. Para ello serían apropiados estadísticos como mínimo, máximo, medio y desviación típica en distintos entornos de pruebas. Igualmente, para algoritmos iterativos sería apropiado medir el posible error en la selección de características más cercanas.

En cuanto a la herramienta Obugre:

- Inclusión de la opción de medir la memoria utilizada por cada motor físico.
- Creación de una API que incluya todas las características de los motores.
- Inclusión de más motores físicos.
- Posibilitar el uso de Obugre exclusivamente como API de los motores físicos.
- Lectura de formatos de fichero como COLLADA y URDF.

Referencias

Anitescu, M. & Potra, F. A. (1997). Formulating Dynamic Multi-Rigid-Body Contact Problems with Friction as Solvable Linear Complementarity Problems. *Nonlinear Dynamics*, *14*, 231-247. doi:10.1023/A:1008292328909

Banks, J., Carson, J. & Nelson, B. (2005). *Discrete-event System Simulation.* Pearson Prentice Hall.

Baraff, D. (1993). Non-penetrating Rigid Body Simulation. En *Eurographics 1993 State of the Art Reports*, Eurographics Association.

Baraff, D. (1996). Linear-Time Dynamics Using Lagrange Multipliers. En *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (pp. 137-146). SIGGRAPH '96. doi:10.1145/237170.237226

Barber, C. B., Dobkin, D. P. & Huhdanpaa, H. (1996). The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw. 22*(4), 469-483. doi:10.1145/235815.235821

Bender, J., Erleben, K., Trinkle, J. & Coumans, E. (2012). Interactive Simulation of Rigid Body Dynamics in Computer Graphics. En *Eurographics 2012 State of the Art Reports*, Eurographics Association.

Boeing, A. (s.f.). PAL. Physics Abstraction Layer. Recuperado el 26 de octubre de 2020, desde http://www.adrianboeing.com/pal/index.html

Boeing, A. & Bräunl, T. (2007). Evaluation of Real-Time Physics Simulation System. En *Proceedings of the 5th International Conference on Com-*

puter Graphics and Interactive Techniques in Australia and Southeast Asia (pp. 281-288). doi:10.1145/1321261.1321312

Catto, E. (2006). Fast and Simple Physics using Sequential Impulses. En *Game Developers Conference (GDC:06)*. Recuperado desde https://box2d.org/files/ErinCatto_SequentialImpulses_GDC2006.pdf

Chatterjee, A. (1999). On the realism of complementarity conditions in rigid body collisions. *Nonlinear Dynamics*, *20*, 159-168. doi:10.1023/A:1008397905242

cos_xavier & sfmonster. (s.f.). The Gangsta Wrapper. Recuperado el 26 de octubre de 2020, desde https://sourceforge.net/p/gangsta/code/

Couman, E. (s.f.). Foro Bullet. Applications, Games, Demos or Movies using Bullet. Recuperado el 10 de noviembre de 2020, desde https://pybullet.org/Bullet/phpBB3/viewforum.php?f=17

Couman, E. & bram. (s.f.). Foro Bullet. Different types of friction. Recuperado el 10 de noviembre de 2020, desde https://pybullet.org/Bullet/phpBB3/viewtopic.php?t=12500

Couman, E. & kevlar. (s.f.). Foro Bullet. GJK Ray cast. Recuperado el 10 de noviembre de 2020, desde https://pybullet.org/Bullet/phpBB3/viewtopic.php?t=565

Couman, E. & skocznymrocznyp. (s.f.). Foro Bullet. Tower of stacked boxes? Recuperado el 10 de noviembre de 2020, desde https://pybullet.org/Bullet/phpBB3/viewtopic.php?t=4323

Coumans, E. (2005). *Continuous Collision Detection and Physics*. Sony Computer Entertainment. Recuperado el 26 de octubre de 2020, desde https://www.gamedevs.org/uploads/continuous-collision-detection-and-physics.pdf

Deul, C., Charrier, P. & Bender, J. (2016). Position-Based Rigid-Body Dynamics. *Comput. Animat. Virtual Worlds*, *27*(2), 103-112. doi:10.1002/cav.1614

Dobkin, D. P. & Kirkpatrick, D. G. (1990). Determining the separation of preprocessed polyhedra — A unified approach. En M. S. Paterson (Ed.),

Automata, Languages and Programming (pp. 400-413). doi:10.1007/BFb0032047

Eberly, D. H. (2003). *Game Physics*. Elsevier Science Inc.

Erez, T., Tassa, Y. & Todorov, E. (2015). Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. En *2015 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 4397-4404). doi:10.1109/ICRA.2015.7139807

Ericson, C. (2004). *Real-Time Collision Detection*. CRC Press, Inc.

Erleben, K. (s.f.). OpenTissue. Recuperado el 26 de octubre de 2020, desde https://github.com/erleben/OpenTissue

Erleben, K. (2004). *Stable, Robust, and Versatile Multibody Dynamics Animation* (Tesis doctoral, Department of Computer Science, University of Copenhagen).

Erleben, K., Sporring, J., Henriksen, K. & Dohlman, K. (2005). *Physics-Based Animation (Graphics Series)*. Charles River Media.

Featherstone, R. (2007). *Rigid Body Dynamics Algorithms*. Springer-Verlag.

Foundation, O. S. R. (s.f.). SDFormat specification. Recuperado el 25 de octubre de 2020, desde http://sdformat.org/spec

Grassia, F. S. (1998). Practical Parameterization of Rotations Using the Exponential Map. *Journal of Graphics Tools*, *3*(3), 29-48. doi:10.1080/10867651.1998.10487493

Hummel, J., Wolff, R., Stein, T., Gerndt, A. & Kuhlen, T. (2012). An Evaluation of Open Source Physics Engines for Use in Virtual Reality Assembly Simulations. En *Advances in Visual Computing* (pp. 346-357). doi:10.1007/978-3-642-33191-6_34

Jacobs, S. (2008). *Game Programming Gems 7*. Charles River Media/Course Technology.

Joseph. (2013). Gyroscopic forces in ODE. Recuperado el 7 de diciembre de 2020, desde http://lost-found-wandering.blogspot.com/2013/01/gyroscopic-forces-in-ode.html

Kang, D. & Hwangho, J. (s.f.). SimBenchmark. Physics engine benchmark for robotics applications: RaiSim vs. Bullet vs. ODE vs. MuJoCo vs. DartSim. Recuperado el 10 de octubre de 2020, desde https://leggedrobotics.github.io/SimBenchmark/

Lacoursière, C. (2006). *Stabilizing Gyroscopic Forces in Rigid Multibody Simulations*. Department of Computing Science, Umeå University.

Lin, M. C. (1994). *Efficient Collision Detection for Animation and Robotics* (Tesis doctoral, EECS Department, University of California, Berkeley).

Lloyd, J. (s.f.). V-Clip Java implementation. Recuperado el 26 de octubre de 2020, desde https://www.cs.ubc.ca/~lloyd/java/vclip.html

Mirtich, B. (1996). *Impulse-Based Dynamic Simulation of Rigid Body Systems* (Tesis doctoral, University of California, Berkeley).

Mirtich, B. (1998). V-Clip: Fast and Robust Polyhedral Collision Detection. *ACM Trans. Graph. 17*(3), 177-208. doi:10.1145/285857.285860

Mirtich, B. (2000). Timewarp Rigid Body Simulation. En *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (pp. 193-200). SIGGRAPH '00. doi:10.1145/344779.344866

Mondesire, S. C., Maxwell, D. B. & Stevens, J. (2016). Physics Engine Threading Design and Object-Scalability in Virtual Simulation. En *2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (pp. 136-141). doi:10.1109/WETICE.2016.37

Ploen, S. R., Hadaegh, F. Y. & Scharf, D. P. (2004). Rigid body equations of motion for modeling and control of spacecraft formations. Part 1: Absolute equations of motion. En *Proceedings of the 2004 American Control Conference* (Vol. 4, 3646-3653 vol.4).

Roennau, A., Sutter, F., Heppner, G., Oberlaender, J. & Dillmann, R. (2013). Evaluation of physics engines for robotic simulations with a special

focus on the dynamics of walking robots. En *2013 16th International Conference on Advanced Robotics (ICAR)* (pp. 1-7). doi:10.1109/ICAR.2013.6766527

Shirley, P. & Marschner, S. (2009). *Fundamentals of Computer Graphics* (3rd). A. K. Peters, Ltd.

Smith, R. [Russel]. (s.f.). ODE - Wiki Manual. Recuperado el 15 de diciembre de 2020, desde http://ode.org/wiki/index.php?title=Manual

Smith, R. [Russell]. (s.f.-a). Foro ODE. Recuperado el 10 de noviembre de 2020, desde https://groups.google.com/g/ode-users

Smith, R. [Russell]. (s.f.-b). Products that use ODE. Recuperado el 15 de diciembre de 2020, desde https://www.ode.org/

Smith, R. [Russell]. (2004). Dynamics Simulation. A Whirlwind Tour. Presentación a PARC. PARC. Recuperado el 1 de noviembre de 2020, desde http://ode.org/slides/parc/dynamics.pdf

Stepie, J. (2013). *Physics-Based Animation of Articulated Rigid Body Systems for Virtual Environments* (Tesis doctoral, Silesian University of Technology).

Stewart, D. & Trinkle, J. C. (1996). An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Coulomb Friction. *International Journal of Numerical Methods in Engineering*, *39*, 2673-2691. doi:10.1002/(SICI)1097-0207(19960815)39:15%3C2673::AID-NME972%3E3.0.CO;2-I

Taylor, J. (2005). *Classical Mechanics*. University Science Books.

Team, B. P. D. (2015). Bullet 2.83 Physics SDK Manual. Recuperado el 11 de enero de 2021, desde https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf

Terdiman, P. (s.f.-a). Library OPCODE, Optimized Collision Detection. Recuperado el 10 de noviembre de 2020, desde http://www.codercorner.com/Opcode.htm

Terdiman, P. (s.f.-b). PEEL. Physics Engine Evaluation Lab. Recuperado el 26 de octubre de 2020, desde https://github.com/Pierre-Terdiman/PEEL

Thulesen, T. N. (2015). *Dynamic Simulation of Manipulation & Assembly Actions* (Tesis doctoral, Syddansk Universitet. Det Tekniske Fakultet).

Tóth, C., Goodman, J. & O'Rourke, J. (2017). *Handbook of Discrete and Computational Geometry*. CRC Press.

van den Bergen, G. (2003). *Collision Detection in Interactive 3D Environments*. CRC Press.

Witkin, A. & Baraff, D. (1997). Physically Based Modeling: Principles and Practice. Recuperado el 7 de diciembre de 2020, desde https://www.cs.cmu.edu/~baraff/sigcourse/

Glosario

AABB Axis-Aligned Bounded Box.

Broadphase Fase de la detección de colisiones orientada a filtrar pares de cuerpos que no pueden colisionar.

BV Bounding Volume.

BVH Bounding Volume Hierarchy.

CA Conservative Advancement (Mirtich, 1996).

CCD Continuous Collision Detection.

CFM Constraint Force Mixing.

CSO Configuration Space Obstacles, $A \oplus -B$ en espacio de Minkowski.

EOM Equations of Motion, ecuaciones del movimiento.

EPA Expanding Polytope Algorithm.

ERP Error Reduction Parameter.

GJK Algoritmo Gilbert–Johnson–Keerthi.

LCP Linear Complementary Problem, problema complementario lineal.

MLCP Mixed Linear Complementary Problem, problema complementario lineal mixto.

MRP Minkowski Portal Refinement.

Narrowphase Fase de la detección de colisiones que detecta y resuelve colisiones.

OBB Oriented Bounded Box.

PAL Physics Abstraction Layer (Boeing & Bräunl, 2007).

SAT Separating Axis Theorem.

TOI Time of Impact.

Apéndice A

Entorno pruebas

Para la realización de este proyecto se ha utilizado un ordenador con las siguientes características:

- Procesador: Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
- Memoria: 8GB RAM DDR4 2400 2400MHz.
- Disco: Seagate Barracuda 1TB. 7200RPM.
- GPU: NVIDIA GeForce GTX 1650.

El sistema operativo utilizado en las pruebas ha sido Ubuntu 20.04 LTS.

El proyecto ha sido compilado en modo Release, y se ha utilizado doble precisión tanto para V-Clip como para ODE y Bullet.

Anexo I

Guía de instalación y uso del software.

I.1. Requisitos

El proyecto ha sido compilado y ejecutado con éxito en los sistemas operativos Ubuntu 20.04 LTS y Windows 10 con Visual Studio, ambos sobre arquitectura x64.

Además de CMake, es necesario un compilador con soporte para C++11 y la posibilidad de compilar los proyectos QHull, Ogre, ODE y Bullet. Ya que ODE solo está soportado en Linux, Windows y macOS, no es posible utilizar este proyecto en otras plataformas.

I.2. Compilación y ejecución

Se dan las instrucciones para compilar el proyecto en el sistema operativo Ubuntu 20.04 LTS.

- Se instalarán las dependencias con el comando: "apt install cmake build-essential libxaw7-dev libxrandr-dev libsdl2-dev libzzip-dev".
- Se deberá descomprimir el fichero que contiene el código fuente o clo-

narse desde el repositorio de git correspondiente.

- Se deberá acceder al directorio principal del proyecto.
- En el caso de que el proyecto no se haya obtenido de git, será necesario ejecutar el comando "bash clone_external.sh" para obtener las dependencias Eigen, QHull, ODE, Google Test, Bullet y Ogre. Algunas de estas dependencias han sido ligeramente modificadas para este proyecto, por lo que se descargan de la cuenta de GitHub del autor.
- Se creará el directorio de compilación. "mkdir build && cd build"
- Se invocará CMake "cmake .."
- Opcionalmente se cambiarán las opciones de compilación con cmake o con una herramienta gráfica como "cmake-gui".
- Con el comando "make" se compilará el código fuente.

Una vez compilado correctamente el proyecto, en el directorio "obvclip" se encuentran los ejemplos de V-Clip sin utilizar motores físicos. Como ejemplo, desde el directorio "build" se puede ejecutar el comando "./obvclip/analisis1" que permite iterar por los estados de V-Clip entre dos cuerpos.

En el directorio "obugre" se encuentran los ejecutables que utilizan la aplicación Obugre para simular los motores físicos. Como ejemplo, el comando "./obugre/comparativa1" ejecutará la prueba 1 de este proyecto. En la Sección I.3 se indica como se utilizan las herramientas de cada directorio.

I.3. Instrucciones de uso

I.3.1. Herramienta visualización V-Clip. Directorio obvclip.

- Tecla '1'. Muestra el wireframe.
- Tecla '2'. Muestra cuerpos sólidos.
- Tecla '3'. Muestra nombres de cuerpos, vértices y caras, los sistemas de coordenadas y las normales de las caras.

- Tecla '4'. Muestra los planos de Voronoi "V-E" del estado actual.
- Tecla '5'. Muestra los planos de Voronoi "E-F" del estado actual.
- Tecla '6'. Muestra los contactos. No funciona si no es un estado final de V-Clip.
- Tecla '7'. Muestra los contactos en modo simple o con algoritmo geométrico.
- Tecla 'p'. Imprime en pantalla la estructura de datos half-edge.
- Tecla 'c'. Avanza al siguiente estado de V-Clip.
- Tecla 'a'. Imprime en pantalla el estado actual de V-Clip.
- Tecla 'n'. Si hay más de dos cuerpos en la escena, avanza al siguiente par de cuerpos.
- Tecla 'd'. Guarda un fichero png con la captura de pantalla.

Para algunos de estos comandos será necesario estar en un estado de V-Clip válido, para lo que es necesario haber invocado al comando 'c'.

I.3.2. Obugre. Directorio obugre.

- Tecla Escape. Sale de la animación.
- Tecla Espacio. Para/Arranca la animación.
- Tecla 'c'. Para la animación en caso de colisión.
- Tecla '1'. Muestra los sistemas de coordenadas de los cuerpos.
- Tecla '2'. Muestra información textual de los cuerpos, nombre del cuerpo, energía cinética, norma del momento lineal y norma del momento angular.
- Tecla '3'. Muestra los vectores de momento angular y momento linear de los cuerpos.
- Tecla '4'. Muestra los sistemas de coordenadas de los motores.

- Tecla '5'. Muestra información textual de los motores, nombre del motor, energía cinética, norma del momento lineal y norma del momento angular.

- Tecla '6'. Muestra los vectores de momento angular y momento linear de los motores, con respecto al origen.

- Tecla '7'. Muestra los contactos.

- Tecla '8'. Cambia entre proyección en perspectiva y proyección ortográfica.

- Tecla '9'. En proyección ortográfica, amplia el frustrum de la cámara.

- Tecla '0'. En proyección ortográfica, disminuye el frustrum de la cámara.

- Tecla 's'. Guarda un fichero png con la captura de pantalla.

I.4. Uso algoritmo V-Clip

El código que implementa el algoritmo V-Clip está en los directorios `obvclip/include` y `obvclip/src`.

La clase `ConvexPolytope` representa la geometría de un poliedro convexo. La clase `WorldConvexPolytope` representa un `ConvexPolytope` con una pose.

En el fichero de cabecera `polytope_examples.hpp` se encuentran funciones de ayuda para crear algunos tipos de poliedros. Por ejemplo, y de forma extensa, para crear una pirámide con base pentagonal de altura 1 y radio de la base 2 a partir de puntos, y darle una pose se puede consultar el Listado I.1.

Listado I.1: Creación de pirámide con pose

```
std::vector<OB::Point> puntos = get_pyramid_points(5,1,2);
std::shared_ptr<OB::ConvexPolytope> piramide =
    std::make_shared<OB::ConvexPolytope>();
piramide->generate_from_vertices(puntos);
OB::WorldConvexPolytope wcp{"piramide", piramide};
OB::Transform t = OB::Translation(0, -3 ,0);
t *= OB::AngleAxis(OB::PI * 0.2, OB::Vector::UnitX());
wcp.set_pose(t);
```

Una vez creados dos `WorldConvexPolytope`, se puede invocar sobre ellos el algoritmo V-Clip. Para ello será necesario o invocar a la función `VClip::solve` con un testigo al azar o al operador `VClip::operator` con una instancia de caché `VClipCache` (ver Listado I.2).

Listado I.2: Uso V-Clip

```
OB::WorldConvexPolytope wcp0 = ...
OB::WorldConvexPolytope wcp1 = ...
OB::VClip vclip{wcp0, wcp1};
VClipWitness witness{VClipWitness::FirstVertices()};
VClipResult result = vclip.solve(witness);
```

La estructura `VClipResult` contiene el resultado del algoritmo V-Clip. Entre otros campos son los testigos de menor distancia o de penetración, si hay o no penetración y la distancia entre los testigos. Con la salida de V-Clip se pueden invocar las funciones del fichero de cabecera `contacts.hpp` que calculan puntos de contactos, normales y distancias entre los poliedros.

Además, en el fichero `polytope.hpp` se encuentran multitud de funciones que pueden resultar útiles para el cálculo de distancias entre características. Algunas de estas son `absdist_edge_face`, `absdist_between_features`, `dist_edge_edge`, `closest_point_to_line`...

I.5. Uso Herramienta visualización V-Clip

El código fuente se encuentra en `obvclip/tools` y ejemplos de su uso en `obvclip/apps` y `obvclip/memoria`. Para ejecutar la herramienta, una vez creados los objectos `WorldConvexPolytope`, se puede utilizar como referencia el código

del Listado I.3.

Listado I.3: Uso herramienta visualización V-Clip

```
OB::OgreVClipApp app;
app.initApp();
app.addWorldConvexPolytope(...);
...
app.addWorldConvexPolytope(...);
app.getRoot()->startRendering();
app.closeApp();
```

I.6. Uso Obugre

Obugre está planteado como un API común a los motores físicos al estilo PAL, junto con la posibilidad de visualización simultánea de varios motores.

Obugre no es un proyecto finalizado, al haberse implementado solo las características necesarias para la realización de este proyecto. Igualmente, aspectos como rendimiento no han sido considerados en su diseño ya que es una herramienta para comparar motores y no para su uso en productos finales. Se describe en este apartado los principios de su funcionamiento.

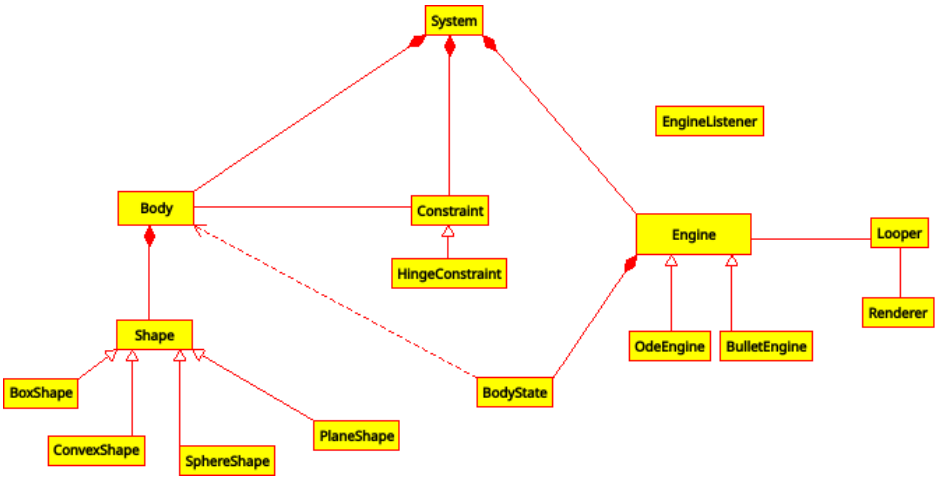


Figura I.1: Principales clases de Obugre

La clase principal de Obugre es la clase `System`. Una vez creada una instancia, será necesario añadir motores físicos, cuerpos con su geometría y

restricciones. Se podrá optar qué cuerpos y que restricciones ser desean utilizar en qué motores. Una vez configurado el escenario y los motores físicos, la clase `Looper` ejecutará la simulación con los motores deseados.

Para poder interactuar con los motores físicos durante la simulación es posible utilizar los callbacks definidos en `EngineListener`, que permiten a la aplicación obtener información o modificar aspectos del escenario. Un ejemplo de este uso es la clase `EngineLogger`, utilizada para obtener información de tiempos de ejecución o de invariantes físicos en los motores (ver Figura I.1 para las principales clases de Obugre).

Listado I.4: Uso herramienta visualización Obugre

```
System system{};
system.set_default_gravity(Vector(0.0, -9.8, 0.0));

system.create_engine("ode", EngineType::Ode);
system.create_engine("bullet", EngineType::Bullet);

Plane ground_plane{Vector(0, 1, 0).normalized(), 0};
Body& ground =  system.create_body("ground");
ground.get_initial_state().set_position(Vector(0,0,0));
ground.set_static(true);

Body& box = system.create_body("box");
BoxShape& box_shape = box1.add_box_shape(Vector(3,4,5));
box.get_initial_state().set_position(Vector(-10,0,0));
box.get_initial_state().set_angular_velocity(Vector(0, -10, 0));


Body& sphere = system.create_body("sphere");
SphereShape& sphere_shape = sphere.add_sphere_shape(5);
sphere.get_initial_state().set_position(Vector(-10,0,0));
sphere.get_initial_state().set_angular_velocity(Vector(0, -10, 0));

HingeConstraint& constraint =
        system.create_hinge_constraint("constraint", "box", "sphere",
                                       Vector(0, 1, 0), Point(2, 0, 0));

system.insert_all_bodies_in_all_engines();
system.insert_all_constraints_in_all_engines();

Looper looper{system};
looper.add_engine("ode", Vector{-40, 0, 0});
looper.add_engine("bullet", Vector{-15, 0, 0});
looper.loop();
```

Algunas de las funciones más útiles a la hora de utilizar Obugre son las siguientes:

- `Engine::add_listener`. Añade un listener al motor deseado.

- `Engine::set_use_vclip`. Utiliza el algoritmo V-Clip para colisiones entre poliedros convexos.
- `Looper::set_simulation_speed`. Cambia la velocidad de la visualización de la simulación.
- `Looper::set_pause_on_collisions`. La simulación se detiene cuando en algún motor hay una colisión.

Para ejemplos completos de uso se pueden consultar los ficheros en los directorios `obugrep/apps` y `obugre/memoria`. Un ejemplo simple de caso de uso se puede encontrar en el Listado I.4.