PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

Visualizador Tridimensional para la Simulación de Vehículos Portuarios

Autor: Gerardo Ismael Urbina Ampuero
Profesor Guía: Claudio Alonso Cubillos Figueroa

INFORME PARA OPTAR AL GRADO DE
MAGÍSTER EN INGENIERÍA INFORMÁTICA

JULIO 2016

Índice

Lista de Figuras

Lista de Tablas

Resumen

La simulación 3D corresponde a un área de estudio que ha permitido mejorar procesos de formación y aprendizaje en una amplia variedad de sectores, esto permite formar operadores prácticos capaces de controlar situaciones de riesgo, debido al entrenamiento seguro y la posibilidad de desenvolverse en actividades que mediante métodos convencionales resultarían inviables. El presente proyecto tiene como objetivo la creación de un motor gráfico tridimensional, el cual permita simular el funcionamiento de un vehículo grúa Reach-Stacker, basándose en modelos de simulación física de alta calidad.

Palabras claves: *Motor Gráfico 3D, Simulación, Reach-Stacker.*

Abstract

3D simulation corresponds to an area of study that has allowed the improvement of training and learning processes within a wide range of sectors, this allows the training of practical operators able to control risky situations, due to the safe training and the possibility to engage in activities that would be impossible to carry out through conventional methods. The end of this project is to create a three-dimensional graphics engine, which enables to simulate the operation of a Reach-Stacker crane vehicle, based on high quality physics simulation models.

Key words: *3D Graphic Engine, Simulation, Reach-Stacker.*

Capítulo 1: Descripción del Tema

1.1 Introducción

La simulación actualmente corresponde a un área de investigación de alto impacto en las empresas, a partir de esta se pueden simular ciertos modelos teóricos que permiten representar procesos de manera mucho más simple, en donde incluso se podrían llegar a simular procesos imposibles de realizar mediante métodos convencionales. Dentro de la simulación existen diversas áreas de investigación, tales como la simulación de eventos discretos, simulación de eventos continuos, simulación de sistemas dinámicos, simulación basada en agentes, simulación basada en la web, simulación de realidad virtual, entre otras [URL 1, 1969]. Una de estas áreas de simulación corresponde a la simulación tridimensional, a partir de la cual se basa el presente proyecto.

La simulación tridimensional o 3D corresponde a una representación o experimentación con un modelo de la realidad mediante la utilización de gráficos tridimensionales, esta tiene una gran influencia en empresas con maquinarias grandes y las cuales deben capacitar a sus trabajadores, a pesar de esto, en la actualidad muchas empresas no se plantean el implementar un simulador dentro de su proceso de formación, esto debido principalmente a los altos costos que suponía adquirir un simulador, es por esto que se piensa en una inversión inabordable para la empresa y sus planes de formación siguen funcionando mediante métodos convencionales. Si bien el costo en un comienzo puede ser en algunos casos elevado, la aplicación que puede llegar a tener es muy amplia, considerando además los diversos adelantos tecnológicos y su uso a largo plazo, se puede llegar a considerar la simulación 3D como una buena opción a tener en cuenta.

Algunas ventajas en la implementación de un simulador son:

- Reducción de costos asociados al aprendizaje: La dedicación de unidades de maquinaria al aprendizaje supone un coste muy elevado.
- Reducción de riesgos en el entrenamiento: Ciertos ejercicios beneficiosos desde el punto de vista del aprendizaje pueden conllevar un alto riesgo.
- Posibilidad de realizar acciones que no son viables con la máquina real, pero que pueden ser beneficiosas para el aprendizaje: Con el uso de un simulador se evita que en ocasiones las máquinas destinadas al aprendizaje se encuentren inutilizables por un determinado escenario (condiciones meteorológicas, trabajo con averías, etc.).
- Mejoramiento de habilidades: Permite mejorar habilidades de operadores experimentados de forma segura.

Un factor que juega un rol muy importante al momento de plantear el desarrollo de un simulador 3D es el comportamiento del entorno y cada uno de los elementos que interactúan con este, buscando principalmente una representación fiel de la realidad. Esto se logra principalmente mediante un modelado matemático y físico de cada uno de los elementos del sistema, otorgándole un conjunto de características representativas a cada uno de los

objetos del mundo 3D, permitiendo de esta forma una interacción que se adapte a los requerimientos del simulador acorde a un entorno físico real.

Un motor gráfico es la pieza fundamental de las aplicaciones que trabajan con gráficos bidimensionales o tridimensionales. Este se encarga de administrar y actualizar en tiempo real cada uno de los elementos que componen el entorno [Shalini, 2004]. Existe una gran cantidad de motores que no solo realizan el apartado gráfico visible por pantalla, sino que incorporan además una serie de otros módulos necesarios para otorgarle mayores capacidades y realismo a lo que se desea representar, estos pueden ser: la reproducción de sonidos y videos, manejo de físicas, uso de dispositivos de entrada y salida, detección de colisiones, entre otros.

En el presente proyecto, a pesar de la gran cantidad de formas de implementar un simulador, se ha optado por la utilización de una Interfaz de Programación de Aplicaciones (API) gráfica de bajo nivel, sin hacer uso de un motor gráfico ya existente. A pesar de las diversas prestaciones y ventajas que implica el uso de un motor gráfico comercial o de código abierto, la capacidad de extensión y modificación que podría implicar un proyecto de estas características dificulta la posibilidad de modificación del código fuente, debido principalmente a la complejidad que esto conlleva. Considerando además la importancia de la implementación de una librería física que permita simular los movimientos de los objetos en un escenario 3D, el desarrollo de un nuevo motor gráfico permitirá aplicar una librería física de manera mucho más fácil y que se ajuste rápidamente a las necesidades del proyecto.

La interfaz tridimensional a realizar corresponde a la creación de un simulador 3D orientado al vehículo grúa Reach-Stacker. La grúa Reach-Stacker es un vehículo utilizado para manipular contenedores en terminales portuarios de pequeño y mediano tamaño. Los Reach-Stacker son capaces de transportar contenedores en pequeñas distancias de manera muy rápida, permitiendo a su vez apilarlos en varias filas de acuerdo al tamaño que este posea.

La solución propuesta en este proyecto para la simulación y modelado del Reach-Stacker está basada en modelos de simulación de alta calidad correspondientes a los vehículos Reach-Stacker TCM [URL 2, 2014], en donde se incorporan tanto elementos visuales como físicos representativos de estos vehículos. El comportamiento de los objetos 3D incluye uno de los aspectos más importantes, el cual es la manipulación de la grúa y el desplazamiento de los contenedores a lo largo de una instalación portuaria, es en este escenario en donde podrá desenvolverse el usuario.

1.2 Definición de Objetivos

1.2.1 Objetivo General

Desarrollar una interfaz tridimensional para la simulación de vehículos portuarios, específicamente la simulación de un Reach-Stacker TCM, vehículo con una gran capacidad de carga capaz de movilizar contenedores de diferentes tamaños a través del terminal portuario, todo esto basándose en el desarrollo de un motor gráfico que se adapte a cada una de las necesidades específicas del proyecto.

1.2.2 Objetivos Específicos

- Investigar las diversas características del vehículo a partir del cual se desarrollará el simulador, su interacción con el entorno y las variables que determinan las acciones que se podrán llevar a cabo.
- Investigar el conjunto de herramientas que permitan elaborar los diversos recursos necesarios para el simulador, contemplando además la integración de estos con el motor gráfico.
- Realizar el diseño del motor gráfico contemplando en este la integración con el simulador y algunos framework o librerías que permitan simular otras características, como es la representación física de los diversos actores y su entorno.
- Desarrollar el simulador.
- Realizar pruebas del sistema.

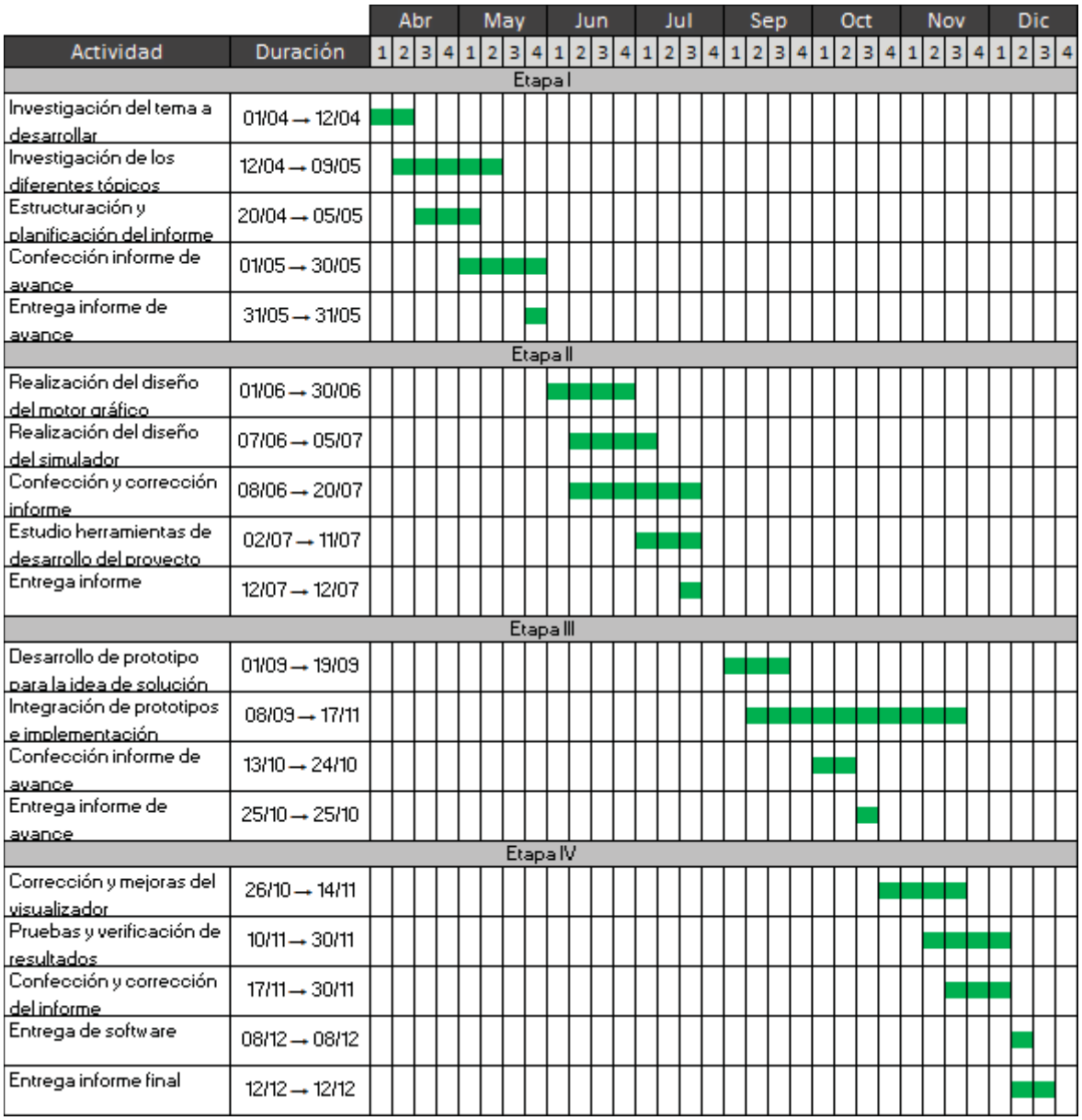
1.3 Planificación del Proyecto

La planificación del proyecto pretende estructurar las diferentes actividades e hitos que forman parte del proyecto. Una buena planificación puede significar el éxito y cumplimiento de cada uno de los objetivos propuestos, con los costos y tiempos establecidos. A partir de los objetivos planteados en el apartado 1.2.1 y 1.2.2, la planificación del proyecto fue dividida en cuatro etapas, las cuales son:

1) Recolección de información y análisis investigativo: En esta etapa, se realiza todo el proceso de investigación y recolección de información correspondiente a las diferentes áreas abordadas en este proyecto: Motores Gráficos, simulación de eventos físicos, funcionamiento del vehículo, ente otros.
2) Diseño y desarrollo de prototipos: Una vez finalizado el proceso de recolección e investigación, se procede a realizar el diseño del sistema en conjunto con el desarrollo de diversos prototipos incrementales. Esto permite validar tanto el diseño como el funcionamiento de las diversas tecnologías aplicadas en la resolución del problema.
3) Integración e implementación: Se integran e implementan los diferentes prototipos y diseños creados en la etapa anterior, permitiendo de esta forma, realizar en la etapa siguiente las pruebas y validaciones correspondientes.

4) Pruebas y validaciones de aplicación: Se realizan pruebas sobre la aplicación final, con el fin de validar los requerimientos iniciales del sistema y el correcto funcionamiento de la aplicación.

Tabla 1.1 Carga Gantt del proyecto [Elab. Propia]



La Tabla 1.1 detalla la forma en que se estructuran y planifican cada una de las actividades que conforman este proyecto, en donde se indican las fechas de inicio y de término correspondientes a los avances y entregas necesarios para cumplir con los objetivos y metas propuestas, abarcando de esta forma la totalidad del tiempo dispuesto.

1.4 Metodología de Trabajo

Para el desarrollo de este proyecto se ha decidido utilizar el modelo de proceso de software Agile UP (AUP). Esta metodología básicamente otorga un proceso simple y fácil de entender para el desarrollo de software mediante diversos conceptos y técnicas rescatadas de la Metodología Ágil y Rational UP (RUP) [URL 3, 2005]. RUP es considerado como una herramienta rígida y conducida por la metodología en cascada, por lo cual se ha flexibilizado con el fin de poder adaptarse a proyectos cambiantes y de tamaños moderados (respecto a tiempo y recursos), surgiendo de esta forma AUP [Prabhudas, 2008], [Canós *et al*., 2003]. En la Figura 1.1 se puede observar el ciclo de vida de esta metodología, en la que se han redefinido tanto las fases de RUP como sus diferentes disciplinas.

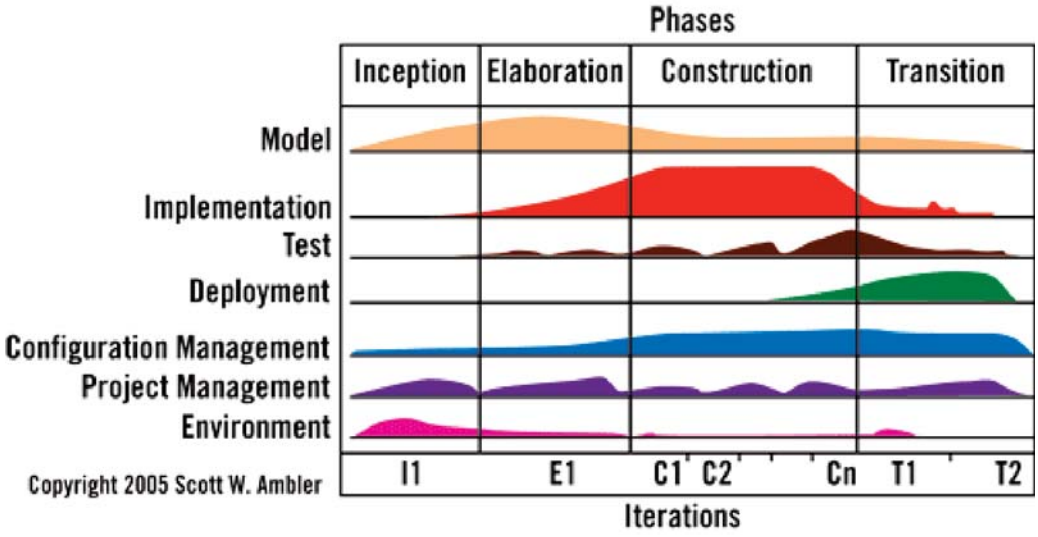


Figura 1.1 Ciclo de vida de Agile UP [URL 3, 2005]

1.4.1 Fases de Ágil UP

El desarrollo del proyecto mediante la metodología AUP implica descomponerse en fases seriales, esto quiere decir, que cada fase se realiza una tras otra, manteniendo siempre la relación que estas poseen. A continuación se describen las cuatro fases que lo componen [Prabhudas, 2008]:

1) Iniciación: Se establece el alcance inicial del proyecto, se priorizan los requerimientos, se identifica la visión de la arquitectura, y se consideran otros puntos, como el financiamiento y la aceptación por parte del involucrado.
2) Elaboración: Se analizan detalladamente los requerimientos, se mejora e inicia el modelamiento de la arquitectura inicial y se comienzan a realizar a su vez implementaciones del software.

3) Construcción: Se construye software funcional, considerando las pruebas y el desarrollo como actividades en conjunto.
4) Transición: En esta fase se consideran las validaciones y el despliegue final del proyecto. Los defectos que no pueden ser solucionados en versiones actuales se solucionan en versiones futuras.

1.4.2 Disciplinas e Iteraciones de Ágil UP

Además de las cuatro fases de AUP, existen una serie de disciplinas desarrolladas de forma iterativa, las cuales en algunos casos pueden realizarse de forma simultánea, estas disciplinas son [URL 3, 2005]:

a) Modelado: Se intenta comprender el negocio de la organización, el problema del dominio y se identifican soluciones viables.
b) Implementación: Se lleva a código ejecutable los modelos creados, para posteriormente poder realizar pruebas del sistema.
c) Pruebas: Se realiza con el fin de asegurar la calidad del software que se está desarrollando. En esta disciplina se considera la detección de defectos, validaciones del sistema y verificaciones de requerimientos.
d) Despliegue: Se planifica la entrega del proyecto y la ejecución del plan con que se realizará dicha entrega.
e) Administración de la configuración: Se verifican versiones del trabajo, el control y la administración de cambios.
f) Administración del proyecto: El objetivo de esta disciplina es dirigir cada una de las actividades a lo largo de todo el proyecto.
g) Entorno: Asegurar que las herramientas estén disponibles para cuando el equipo del proyecto las necesite.

Al igual que otras metodologías, en AUP se entregan diversas iteraciones a medida que se avanza en el proyecto (Figura 1.2), estas se pueden utilizar para realizar un aseguramiento de calidad, pruebas, procesos de despliegue, entre otros. Constituyendo un resultado de valor para el negocio.

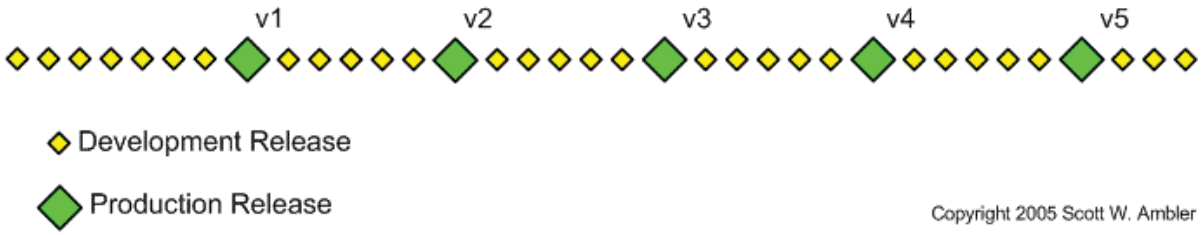


Figura 1.2 Versiones incrementales en el tiempo [URL 3, 2005]

Capítulo 2: Motores Gráficos

Al igual como ocurre con otras disciplinas en el campo de la informática, el desarrollo de aplicaciones gráficas se ha beneficiado de la aparición de herramientas que facilitan dicho desarrollo, automatizando tareas y otorgando un nivel de abstracción lo suficientemente elevado como para centrarse principalmente en la lógica de la aplicación. Un motor gráfico podría considerarse como un conjunto de herramientas y componentes que permiten generar entornos gráficos en tiempo real [Gregory, 2009]. La evolución de la tecnología relacionada en este ámbito ha permitido explotar considerablemente las capacidades que otorgan los diferentes dispositivos de procesamiento gráfico, permitiendo procesar una mayor cantidad de datos en menor tiempo, lo que conlleva a la generación de entornos mucho más realistas, si se considera además la incorporación de otro tipo de módulos, como la detección de colisiones, cálculo de fuerzas, incorporación de música y sonido, interfaces para la gestión de eventos, captura de dispositivos de E/S, entre muchos otros, se podrían llegar a simular procesos de la vida real con un gran nivel de detalle. Actualmente existe una gran cantidad de motores gráficos enfocados en las necesidades de desarrolladores independientes que no pueden crear su propio motor, ni las herramientas necesarias o adquirir una licencia. Esto ha tenido como consecuencia un desarrollo bastante importante en diversas áreas de estudio. Muchos usuarios han logrado plasmar sus ideas con pocos conocimientos respecto al funcionamiento interno del motor, o de las tecnologías que este maneja, permitiéndoles desarrollarse profesionalmente en un área de difícil acceso, por ejemplo, algunos diseñadores y animadores de gráficos 3D con altos conocimientos en herramientas de modelado 3D, han hecho uso de motores gráficos para presentar sus trabajos a través de un entorno 3D en tiempo real.

Si bien la mayoría de los motores gráficos son utilizados para la creación de videojuegos (los cuales se conocen en la literatura más comúnmente como Motores de Videojuegos), esto no impide que se puedan crear otro tipo de aplicaciones gráficas, ya que el conjunto de herramientas otorgadas por el motor no se centra en ningún área en específica, permitiendo generar entornos de acuerdo a las necesidades de cada usuario.

2.1 Arquitectura del Motor Gráfico

Un motor gráfico es un sistema de software bastante complejo, al igual que muchos sistemas de software, este se construye mediante capas. Por lo general las capas superiores dependen de las capas inferiores, pero no viceversa. Cuando una capa inferior depende de una superior, se le llama una dependencia circular. Las dependencias circulares deben evitarse, esto debido al acoplamiento indeseable que genera entre los sistemas, haciéndolo imposible de validar e impidiendo la reutilización del código. Esto se puede ver más claramente en sistemas de gran escala, como es un motor gráfico.

En la Figura 2.1 se presentan los componentes principales de ejecución que se encuentran típicamente en un motor gráfico 3D. A continuación se realizará una descripción general de cada una de las capas que componen el motor, desde el nivel más bajo al más complejo, describiendo sus componentes y enfocándose en los que se influyen directamente en el desarrollo del presente proyecto.

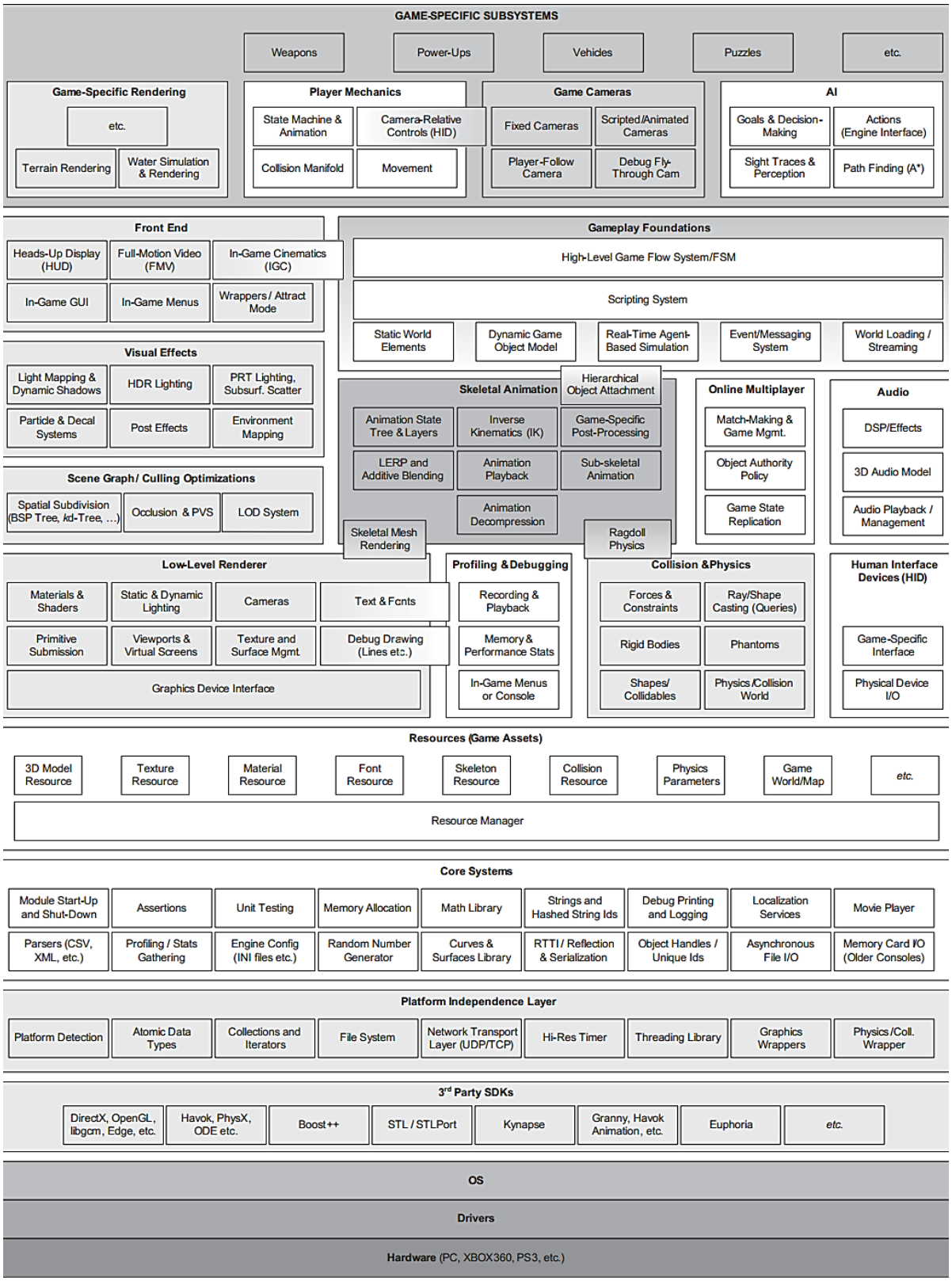


Figura 2.1 Arquitectura típica de un motor gráfico [Gregory, 2009]

2.1.1 Hardware

Esta capa representa el sistema operativo o la consola en la cual se ejecutará la aplicación a partir del motor gráfico. Algunas de estas plataformas pueden ser: Microsoft Windows, Linux, Apple iPhone, Macintosh, Microsoft Xbox One, PlayStation 4, PlayStation Vita (PS Vita), Navegadores Web, entre muchas otras.

2.1.2 Drivers

Corresponden a componentes de software de bajo nivel proveídos por el sistema operativo o directamente por el proveedor de hardware. Estos driver administran los recursos del hardware y otorgan una capa de abstracción tanto para el sistema operativo como para las demás capas en la arquitectura del motor gráfico, obviando detalles de comunicación con el conjunto de variantes que podría tener el hardware y los dispositivos disponibles.

2.1.3 Sistema Operativo

En el caso de un PC, el sistema operativo está ejecutándose todo el tiempo. Este orquesta la ejecución de múltiples programas en un único computador, en donde uno de estos programas corresponde a la aplicación gráfica. Sistemas operativos como Microsoft Windows emplean un mecanismo de multitarea preventiva, el cual busca compartir el hardware a lo largo de múltiples programas ejecutándose simultáneamente. Esto significa que la aplicación gráfica nunca asumirá que esta posee el control total del hardware, sino que se está ejecutando bien con otros programas en el mismo sistema.

En el caso de una consola, por lo general el sistema operativo es una pequeña capa de biblioteca que se compila directamente en el archivo ejecutable de la aplicación, por lo que normalmente cada aplicación *"posee"* toda la máquina. Sin embargo, en consolas como Microsoft Xbox One o PlayStation 4, el sistema operativo puede interrumpir la ejecución de la aplicación o tomar ciertos recursos del sistema, esto con el fin de mostrar ciertos tipos de mensajes, o permitir al usuario que pueda pausar la aplicación para acceder al menú principal de aplicaciones.

2.1.4 SDKs y Middleware de Terceros

La mayor parte de los motores gráficos implementan un gran número de Kits de Desarrollo de Software (SDKs) y middleware de terceros, como se puede observar en la Figura 2.1, las interfaces proveídas por un SDK son a menudo llamadas Interfaces de Programación de Aplicaciones (API), Las APIs básicamente otorgan una serie de funciones y procedimientos de uso general que permiten abstraerse de ciertas rutinas que por lo general deben programarse a bajo nivel. En esencia, una API define sentencias reutilizables que permiten modularizar funcionalidades que son finalmente incorporadas a las aplicaciones [Reddy, 2011].

2.1.4.1 APIs Gráficas

Las APIs gráficas están destinadas principalmente al manejo de gráficos representados por pantalla, estas interfaces se encuentran limitadas a las capacidades de hardware que posea el computador en el que se está desarrollando la aplicación.

Existen dos tipos de APIs gráficas, las de propósito específico y las de propósito general. Las APIs de propósito específico se utilizan principalmente para la creación de aplicaciones destinadas a quienes deseen mostrar imágenes sin preocuparse del procedimiento necesario para producir dicha imagen, por otro lado, las APIs de propósito general están destinadas principalmente a programadores que necesiten hacer uso de funciones a un nivel de abstracción menor [Hearn y Baker, 2005].

A continuación se describirán las dos interfaces de programación gráfica más importantes que existen actualmente en el mercado: OpenGL y Direct3D.

- **OpenGL**:

El sistema gráfico OpenGL (Open Graphics Library) es una interfaz de software para hardware gráfico originalmente creada por Silicon Graphics Incorporated (SGI), esta permite la creación de programas interactivos que requieren mostrar gráficos 2D y 3D en tiempo real [Shreiner, 2010], [Maroto, 2005]. Algunas de las principales características que hacen de OpenGL una fuerte alternativa a considerar al momento de escoger una API gráfica son: su gran potencial, su interfaz independiente del hardware, su simplicidad y su eficiencia.

Existe una gran cantidad de áreas en las cuales se ha utilizado el uso de esta API para el desarrollo de aplicaciones gráficas 3D, como son: los videojuegos, películas, simulaciones, hasta visualizaciones para uso científico, médico, comerciales, entre otros [Wright y Lipchak, 2004]. Actualmente es reconocido como estándar sobre gráficos computacionales, siendo además uno de los más utilizados en el mundo.

Ésta API posee una gran cantidad de funcionalidades, tales como el soporte de luces y sombras, mapeado de texturas, transparencia, animación de modelos, soporte de nieblas, texturas con profundidad, etc. [Wright et al., 2007].

Las partes que componen la escena gráfica en OpenGL son construidas a partir de primitivas geométricas básicas, como cubos, esferas, pirámides, etc., las cuales luego son convertidas mediante funciones matemáticas a un conjunto de pixeles (elemento visible más pequeño de una imagen). Finalmente estos pixeles se proyectan sobre un plano bidimensional, el cual puede ser por ejemplo el área de visualización de un monitor de computador [Hearn y Baker, 2005].

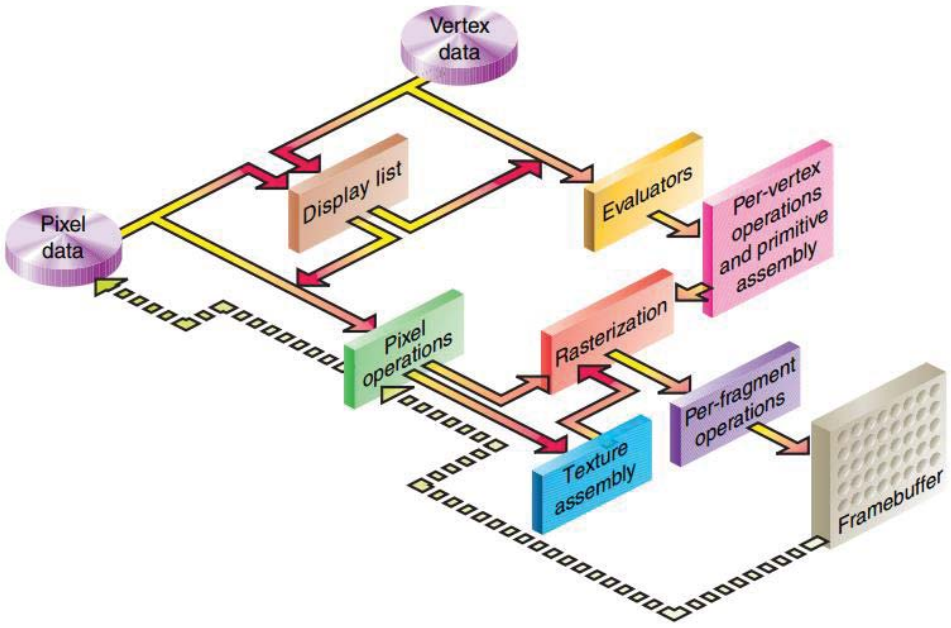


Figura 2.2 Fixed-function pipeline de OpenGL [Shreiner, 2010]

En la Figura 2.2 se puede observar el fixed-function pipeline de OpenGL, nombre que se le aplica a los pipeline de APIs gráficas anteriores a la llegada de los shaders. De manera general, el conjunto de elementos que conforman este pipeline son [Shreiner, 2010]:

- Lista de visualización: Todo el conjunto de datos que describe la geometría o pixeles se puede almacenar en una lista de visualización (la alternativa a almacenar en una lista de visualización es enviar inmediatamente el conjunto de datos, lo que se denomina como *modo inmediato*), la lista de visualización accede directamente al hardware gráfico, por lo que funciona de manera mucho más rápida y eficiente).
- Evaluadores: Proveen un método para derivar los vértices utilizados para representar superficies mediante los puntos de control. Este método puede producir normales, coordenadas de texturas, color, y valores de coordenadas espaciales desde los puntos de control.
- Operaciones por vértice y ensamblador de primitivas: En esta fase se convierten los vértices en primitivas, se generan y transforman las coordenadas de textura, se ejecutan cálculos de luces, entre otros. En el ensamblado de primitivas se realiza mayoritariamente funciones de *clipping*, en donde se eliminan las porciones de la geometría que se encuentra fuera del plano de proyección definido.
- Operaciones de pixel: Muchas de estas operaciones mueven grandes cantidades de datos de información de los pixeles en todo el sistema.
- Ensamblador de textura: Permite aplicar imágenes de texturas a objetos geométricos para crear objetos mucho más realistas.

- Rasterización: Corresponde a la conversión de tanto de los datos geométricos como de los pixeles en fragmentos correspondientes a ubicaciones de pixeles en la pantalla.
- Operaciones por fragmento: En esta fase, un conjunto de operaciones se aplica al conjunto de datos que se almacenan en el *framebuffer* (buffer en donde se realiza el volcado de los datos a la pantalla), una de estas operaciones corresponde al texturizado, en donde los texeles (unidad más pequeña de una textura), se aplica a cada uno de los fragmentos que les corresponde.

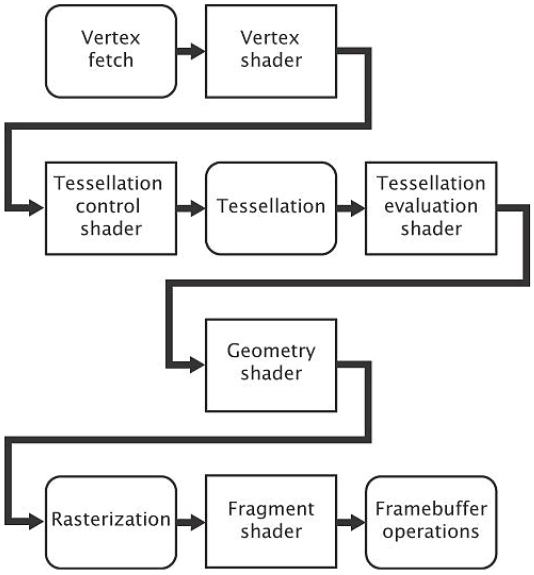


Figura 2.3 Pipeline OpenGL simplificado [Wright *et al.*, 2014]

En el presente proyecto se pretende hacer uso de las nuevas funcionalidades que otorgan tanto las tarjetas como APIs gráficas. En la Figura 2.3 se presenta el nuevo pipeline de OpenGL, el cual hace uso de shaders, los cuales son programados con el lenguaje de programación GLSL (OpenGL Shader Language), permitiendo que estos se ejecuten directamente en la Unidad de Procesamiento Gráfico (GPU). Más detalladamente, las actuales GPU consisten en un gran número de pequeños procesadores programables llamados *shader cores*, los cuales ejecutan pequeños programas llamados *shaders*.

Las figuras con bordes redondeados de la Figura 2.3 son consideradas como etapas *fixed-function*, lo que quiere decir que son programables, no así con las figuras con puntas cuadradas. A continuación se describen brevemente estas etapas [Wright *et al.*, 2014]:

- Vertex fetch: Prepara el conjunto de vértices que posteriormente será enviado a los vertex shader.
- Vertex shader: Programa en donde se trabaja con la estructura de vértices procesada previamente, este recibe un vértice a la vez y lo transforma a partir de otras estructuras de datos, como vectores o matrices. También permite recibir y enviar datos a otros shader.

- Shader de control de teselación: Toma la salida del vertex shader y es responsable en una primera instancia de determinar el nivel de teselación que será enviado al motor de teselación, mientras que la segunda es como se generarán los datos que serán enviados al shader de evaluación de teselación una vez ejecutado el motor de teselación.
- Teselación: Una vez que este motor de teselación es ejecutado, produce un número de salidas de vértices los cuales representan un conjunto de primitivas. Estas primitivas son enviadas posteriormente a los shader de evaluación de teselación.
- Shader de evaluación de teselación: Procesa la salida de vértices generados por la etapa de teselación y procesa las posiciones interpoladas de dichos vértices y otros tipos de cálculos.
- Geometry shader: Los geometry shader permiten crear primitivas básicas, tales como puntos, líneas o triángulos. A partir de estos shader se pueden crear modelos en tiempo de ejecución de manera muy simple (muy similar a la creación de modelos mediante el fixed-function pipeline). El tiempo que toma en generar un modelo es considerablemente mayor al tiempo que se demoraría si este es generado previamente y enviado a un vertex shader.
- Rasterización: Al igual que en el fixed-function pipeline, el conjunto de vértices procesados, cálculo de posiciones de textura, entre otros, deben ser convertidos a fragmentos ubicables en la pantalla, proceso que realiza la rasterización.
- Fragment shader: Trabaja en base al conjunto de fragmentos ya procesados. En este shader se procesa cada uno de los fragmentos por separado, y se le aplica finalmente el color que le corresponde.
- Framebuffer operators: Se establece el contenido visible de la pantalla a ser considerado, además de un número de regiones adicionales de memoria que son utilizadas para almacenar valores por pixel que no son de color.

- **Direct3D**:

Direct3D es una de las APIs que posee DirectX, utilizada principalmente en el sistema operativo Microsoft Windows y consolas como Xbox 360 y Xbox One. Esta API se encarga de todo el procesamiento gráfico necesario para poder visualizar imágenes 3D por pantalla. Una de las características de esta API, es que trabaja directamente con el hardware gráfico, a diferencia de otras interfaces de dispositivos gráficos, aumentando de esta forma el rendimiento considerablemente [Thomson, 2006].

Además esta API fue diseñada para tener fácil acceso a las características avanzadas 3D del hardware gráfico, creando una capa de abstracción la cual otorga una interfaz para el programador independiente del hardware [Engel y Geva, 2000].

Actualmente es utilizada por grandes motores de videojuegos como UDK (Unreal Development Kit) o CryEngine, esto debido principalmente a su capacidad de crear escenas realistas en tiempo real con una gran cantidad de detalles.

2.1.5 Capa de Independencia de la Plataforma

La mayor parte de los motores requieren ser capaces de ejecutar las aplicaciones en más de una plataforma de hardware, esta capa se asienta sobre las capas de hardware, drivers, sistema operativo y software de terceros. Esta independencia se puede realizar envolviendo o reemplazando las funciones estándar del lenguaje de programación, las llamadas a sistema operativo y el conjunto de Interfaces de Programación de Aplicaciones (API) utilizadas por el motor, de este modo la capa de independencia de la plataforma asegura un comportamiento coherente a todas las plataformas de hardware.

2.1.6 Sistemas Principales

Todo motor gráfico, requiere de un conjunto de herramientas de software que faciliten el desarrollo del motor, estas pueden ser implementaciones propias del desarrollador, como también librerías desarrolladas por terceros. Algunos ejemplos de esta capa se detallan a continuación:

- Aserciones: Corresponden a una serie de líneas de chequeo de errores, estas se encuentran insertas con la finalidad de capturar equivocaciones lógicas y accesos no permitidos por parte del programador. Las aserciones son por lo general eliminadas en versiones de producción de la aplicación.
- Librería Matemática: Los motores gráficos trabajan directamente con una gran cantidad de funciones matemáticas, es por eso que se hace casi indispensable el contar con un conjunto de funciones matemáticas y estructuras de datos que faciliten los diversos cálculos que finalmente permitirán generar el entorno 3D. Estas librerías por lo general ayudan en la creación de vectores y matrices, quaterniones, traslaciones, trigonometría, operaciones geométricas con líneas, esferas, generación de diversas proyecciones, y cualquier otra funcionalidad que el motor requiera [Lengyel, 2012].
- Estructuras de datos y algoritmos: A menos que los diseñadores del motor hayan decidido confiar completamente en librerías de terceros, como en la librería de plantillas estándar de C++ (STL), un conjunto de herramientas para el manejo de estructuras de datos y algoritmos es necesario. A menudo se programan a mano, permitiendo minimizar y eliminar la asignación de memoria dinámica y para garantizar un rendimiento óptimo.

2.1.7 Administrador de Recursos

En todo motor, el administrador de recursos provee una interfaz unificada para el acceso a muchos de los tipos de recursos que pueda necesitar la aplicación gráfica. En algunos motores, esto lo realizan de una manera consistente y altamente centralizado, mientras que en otros enfoques, ofrecen libertad al programador de acceder directamente a los tipos de archivos almacenados en disco con un sistema de comprensión de archivos especializado como los archivos .PAK o .BSP, los cuales contienen un conjunto de recursos de todo tipo (entidades, texturas, caras, nodos, planos, etc.).

2.1.8 Renderizador de Bajo Nivel

El motor de renderizado corresponde a uno de los elementos más complejos y extensos en cualquier tipo de motor gráfico. Los renderizadores pueden implementarse de muchas manera diferentes, a pesar de esto, la mayor parte de los renderizadores actuales comparten algunas características fundamentales. En las siguientes sub-secciones se presenta el motor de renderizado mediante una arquitectura dividida en capas. El renderizador de bajo nivel se encarga de establecer las bases a partir de las cuales se desarrollará el resto de los elementos de renderizado. En este nivel, el diseño se enfoca en la renderización de un conjunto de primitivas geométricas tan rápido como sea posible, sin enfocarse todavía en que partes de la pantalla serán visibles. A este nivel se pueden observar algunos elementos como:

- Cámaras: La cámara es la encargada de definir el campo de visión que tendrá el usuario a partir del tipo de proyección definido para la escena. En versiones simplificadas del pipeline de renderizado, estas además son las encargadas en la mayor parte de los casos de definir la matriz de MVP (proyección, vista y modelo) a partir de la cual se ubican los diferentes objetos en el escenario.
- Texto y Fuentes: Para la generación de texto y fuentes, así como también algunos elementos de la Interfaz Gráfica de Usuario (GUI), se deben establecer las bases para la proyección correspondiente al resultado que se espera, si este corresponde a texto o imágenes ubicadas en la pantalla, se debe optar por la utilización de una proyección ortogonal relativa al tamaño de la ventana, mientras que la matriz de proyección en perspectiva podría servir para colocar texto dentro de algún elemento 3D.
- Administrador de Texturas: Este permite mantener un control de la textura de cada uno de los elementos de la aplicación, en esta se almacena el conjunto de bits de cada imagen, además de su ancho y alto.
- Viewport y Ventanas Virtuales: Las ventanas virtuales corresponden al conjunto de ventanas creadas por el manipulador del dispositivo de ventanas a partir del cual se genera el canvas para dibujar los diferentes modelos 3D.

  El viewport se utiliza comúnmente para establecer diferentes áreas de dibujado en un mismo canvas, con esta se pueden reubicar las dimensiones de la ventana en relación al canvas que esta contiene en su interior, permitiendo por ejemplo, el poder dibujar toda la escena en la ventana en una relación de aspecto de 1:3. En la Figura 2.4 se puede observar una transformación del viewport en donde se ha reducido el área original de la ventana, haciendo que el resultado final de la geometría se pueda observar en un mayor tamaño.

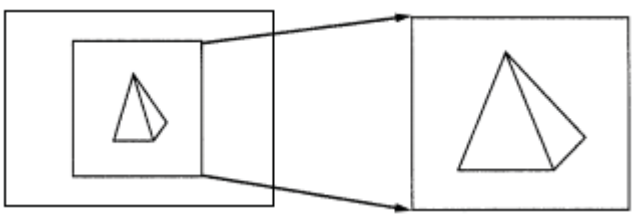


Figura 2.4 Transformación del viewport [Shalini, 2004]

2.1.9 Scene Graph / Optimización de Culling

Como se observó en la sección anterior, el renderizador de bajo nivel dibuja toda la geometría, sin calcular cuales de estas no son actualmente visibles (sin considerar el cálculo de ocultamiento realizado por el back-face culling o el clipping de triángulos). Para pequeñas aplicaciones gráficas, basta con una implementación de *frustrum cull* (algoritmo simple de eliminación de objetos que no son visibles por la cámara), pero cuando los mundos empiezan a convertirse cada vez más complejos, es necesario aplicar otro tipo de algoritmos, como pueden ser estructuras de datos de subdivisión del espacio.

2.1.9.1 Partición Binaria del Espacio

Una de estas técnicas utilizadas para subdividir el espacio se denomina partición binaria del espacio (BSP). La partición binaria del espacio es un método para subdividir recursivamente el espacio en elementos convexos utilizando planos. Esta subdivisión permite representar la escena mediante la estructuración de datos en un árbol, conocido como árbol BSP. Una de las funcionalidades de esta estructuración es la organización de los polígonos del modelo de forma jerárquica respecto a representación de las posiciones en el espacio [Gómez *et al*., 1999]. En la Figura 2.5 se puede observar como a partir de un polígono se va generando el árbol BSP, subdividiendo la superficie poco a poco mediante planos, obteniendo como resultado polígonos convexos.

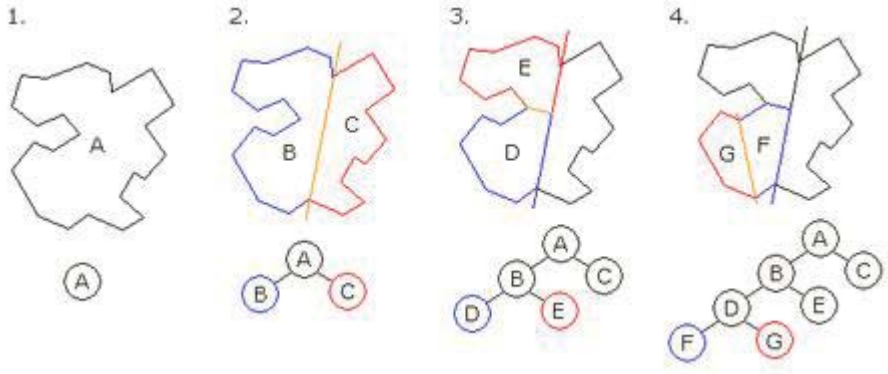


Figura 2.5 Ejemplo árbol BSP [URL 4, 2011]

2.1.9.2 Sistema de Nivel de Detalle

Otra forma de mejorar el rendimiento de la aplicación, puede ser la reducción del nivel de detalle de un determinado modelo 3D mediante el uso de LODs (sistema de nivel de detalle). Generalmente, una escena 3D está compuesta por una gran cantidad de polígonos, los cuales en muchas ocasiones no son percibidos completamente por el usuario [URL 5, 2009]. Para mejorar la eficiencia de las aplicaciones, se sugirió utilizar versiones más simples de geometría para elementos que no tienen una gran relevancia visual en determinados momentos, por ejemplo, geometrías que se encuentran muy lejos de la visión del usuario (ver Figura 2.6).

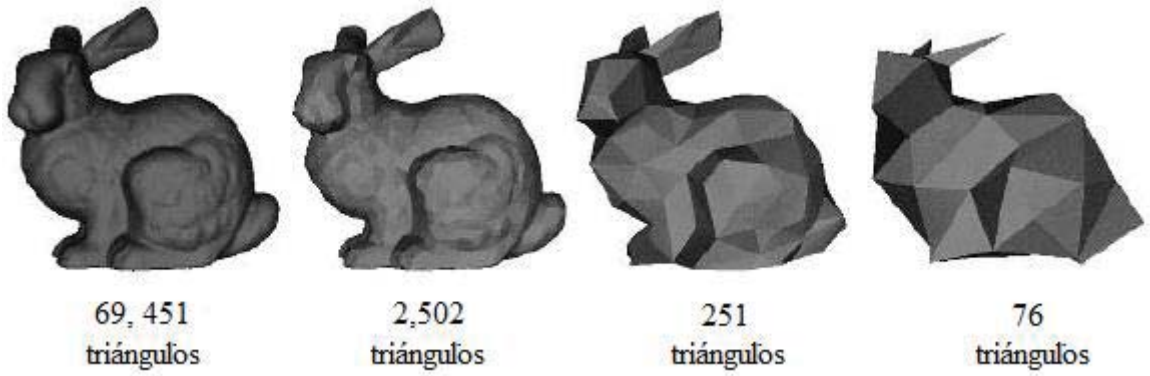


Figura 2.6 Ejemplo de diferentes tipos de LODs [URL 5, 2009]

2.1.10 Efectos Visuales

La mayor parte de los motores gráficos modernos implementan una can variedad de efectos visuales, los cuales pueden ser: sistemas de particular (humo, fuego, explosiones, marcas de huellas, etc.), sombras dinámicas, efectos post-procesamiento, sistemas de calcomanía (huellas en superficies 3D, hoyos de bala, etc.), anti-aliasing a pantalla completa (FSAA), entre muchos otros.

2.1.11 Front End

Una gran variedad de aplicaciones gráficas hacen uso de entornos 2D sobre la escena 3D por varios propósitos, algunos de estos pueden ser la creación de la interfaz gráfica de usuario (GUI), la visualización de elementos que se presentan en todo momento de la aplicación, denominado *heads-up display* (HUD), entre otros.

2.1.12 Perfiles y Herramientas de Depuración

Debido a que la mayor parte de las aplicaciones gráficas funcionan en tiempo real, es necesario el implementar un sistema que mida el rendimiento de esta con la finalidad de poder realizar mejoras. Esta capa engloba el conjunto de herramientas de análisis y facilidades para realizar la depuración de la aplicación, tales como el dibujado de la depuración, entrega de información a través del menú de la aplicación o la consola, y la posibilidad de grabar y re ejecutar la aplicación con fines de depuración y prueba.

2.1.13 Colisiones y Físicas

La colisión es un elemento muy importante para cualquier tipo de aplicación gráfica. Sin esta, los objetos podrían fusionarse y sería imposible interactuar con el mundo virtual de ninguna forma. El conjunto de simulaciones dinámicas realistas o semi-realistas aplicadas se les denomina en la industria de videojuegos como el *sistema de*

físicas. Para realizar esto, la mayor parte de los motores hacen uso de SDK de terceros, los cuales típicamente se integran en el motor.

Existen varios motores físicos, los cuales son tanto privados como de código abierto. Muchos de los motores físicos otorgan simulaciones bastante realistas de elementos que interactúan con el entorno, como simulación de fuerzas, colisiones, agua, viento, cuerpos rígidos, cuerpos blandos, cuerdas, etc.

En el presente proyecto se ha optado por la implementación de Bullet Physics como motor de físicas [URL 6, 2014], esto debido principalmente por su amplia aceptación en el desarrollo de aplicaciones independientes, su extensa documentación y una comunidad de desarrolladores en constante crecimiento.

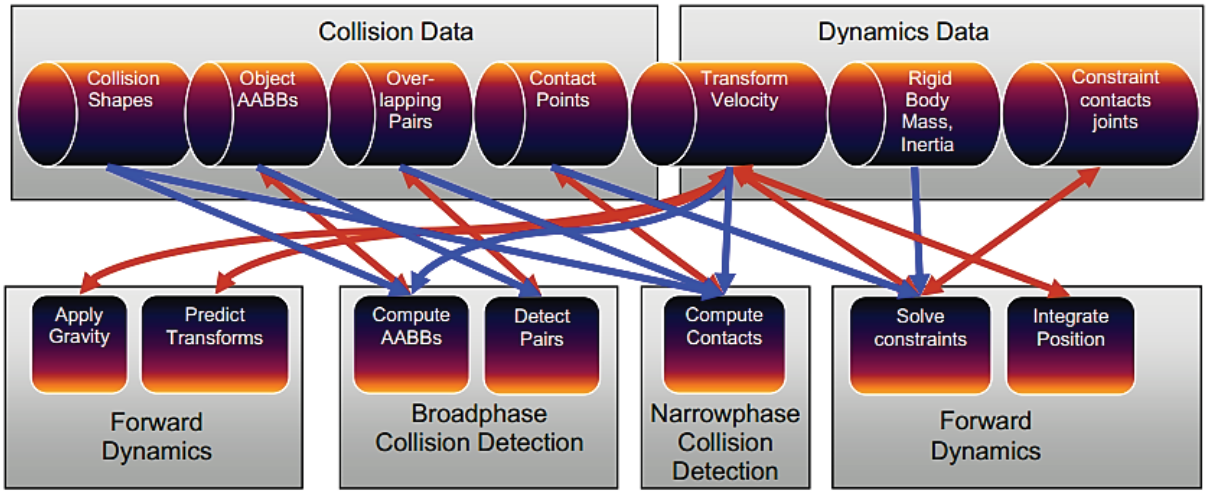


Figura 2.7 Pipeline de Bullet Physics [URL 6, 2014]

Bullet Physics es capaz de simular un mundo mediante físicas tanto para cuerpos rígidos como para cuerpos blandos, en la Figura 2.7 se presenta el pipeline de Bullet Physics de cuerpos rígidos, el cual debido a las características del proyecto será el encargado de realizar los cálculos correspondientes a las físicas involucradas en el simulador. Este pipeline se divide principalmente en seis partes:

- Datos de colisión: En esta sección se almacena el conjunto de datos de colisión utilizados para la detección de colisiones, algunos de estos elementos son las figuras de colisiones, las cuales pueden ser cajas, círculos e incluso modelos 3D; también se ubican los objetos de Axis-Aligned Bounding Box (AABB), el cual consiste en un cubo alineado al cual no se le realizan rotaciones, sirve para una detección de colisiones rápida.
- Datos dinámicos: Se define el conjunto de propiedades que determinan el dinamismo que tendrán los objetos una vez incorporados en el mundo definido por el motor físico. Se establecen propiedades como la velocidad de transformación, la inercia, la masa, entre otros.

- Dinámicas de empuje de gravedad y transformaciones: En esta fase se aplica la gravedad y se establecen las transformaciones de manera predictiva.
- Fase amplia de detección de colisiones: Se realiza una primera detección de colisiones a partir de los AABB de cada modelo, esto permite descartar el conjunto de objetos que no colisionarán, es un método bastante rápido y eficiente.
- Fase estrecha de detección de colisiones: Una vez que se ha detectado una colisión mediante AABB realizado en la fase previa, se procede a realizar una detección de colisión más compleja, en donde se calcula la rotación y traslación de los objetos correspondientes.
- Dinámicas de empuje de integración de posición y resolución de constraint: Se calculan los constraint, los cuales corresponden a las diversas restricciones de movimiento de los objetos y se integran el conjunto de posiciones de los objetos.

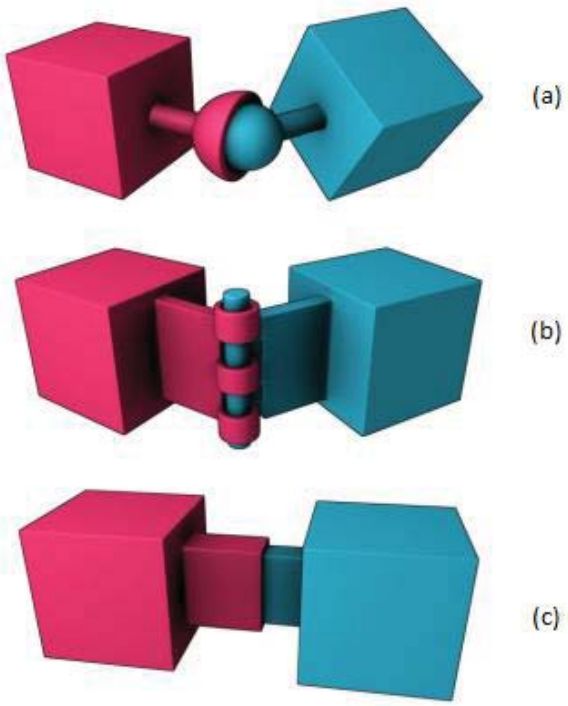


Figura 2.8 Tipos de constraint en Bullet Physics [URL 6, 2014]

Considerando el tipo de vehículo a simular, se presenta el conjunto de constraint a partir de los cuales se realizarán las diversas conexiones de las partes.

En la Figura 2.8 se presentan los constraint que implementa Bullet Physics correspondiente a las dinámicas de empuje que interactúan tanto con los datos de colisión como los datos dinámicos, los constraint que se observan en la figura corresponden a: constraint de punto a punto (a), constraint de bisagra (b) y constraint de deslice (c).

2.1.14 Animación

En la mayor parte de aplicaciones con modelos semi-orgánicos (como humanos o animales) se necesita un sistema de animación. El sistema de animación permite animar los objetos a partir de transformaciones básicas en los ejes XYZ, las cuales corresponden a la rotación, escala y traslación de dichos objetos; por otra parte se pueden animar en cuanto a la forma, en donde se aplican diversas técnicas para poder simular el esqueleto de los objetos, en donde se define una estructura central con la capacidad de afectar la forma y movimientos de los vértices que interactúan con cada hueso.

2.1.15 Dispositivos de Interfaz Humana (HID)

En esta capa se procesan las diversas entradas por parte del usuario, las cuales se obtienes a través de varias HIDs, algunas de estas entradas pueden ser de mouse y teclado, joystick, manubrios, entre otras. Por lo general el conjunto de eventos para la captura de estos interfaces son otorgados por los SDKs de terceros, en especial la API que permite la gestión de la ventana, más conocida como el *Event Handler*.

2.1.16 Audio

El audio es tan importante como las gráficas en cualquier motor gráfico. A pesar de esto, al audio por lo general no se le aplica la misma importancia que el renderizado, las físicas, la animación, la IA y el gameplay.

La aplicación de un motor de audio no es trivial en la creación de una aplicación gráfica, en especial en una aplicación gráfica 3D, esto debido principalmente a que se deben aplicar un conjunto de reglas que permitan ajustar los diversos parámetros de cada sonido en tiempo real.

2.1.17 Networking

Actualmente la red se ha convertido en una característica casi fundamental en la mayor parte de aplicaciones gráficas, en donde se ha visto un interesante incremento en las aplicaciones interactivas distribuidas (DIAs) [Smed, 2008]. Muchas aplicaciones permiten una gran cantidad de usuarios que se conectan en un único mundo virtual. Las aplicaciones con múltiples usuarios se pueden dividir en cuatro secciones:

- Multiusuarios en una única pantalla: Dos o más dispositivos de interfaz humana (teclado, joystick, manubrios, etc.) son conectados en una única pantalla.
- Multiusuario con pantalla dividida: Múltiples usuarios están conectados a un entorno único, en donde se divide la pantalla de acuerdo a la cantidad de usuarios.
- Multiusuario conectado en red: Múltiples computadores o consolas son conectadas en red simultáneamente.
- Aplicación multiusuario online masiva: Cientos de miles de usuarios se conectan a simultáneamente con un mundo virtual persistente hosteado por una gran cantidad de servidores.

2.1.18 Sistema Base de Gameplay

El término de *gameplay* hace referencia a la acción que toma lugar en la aplicación, las reglas que controlan el mundo 3D en el cual la aplicación toma lugar, las acciones que llevan a cabo las entidades, y otras acciones que lleva a cabo el usuario. Esta capa permite llenar la brecha que se encuentra entre el código de gameplay y los sistemas del motor de bajo nivel.

Cuando una aplicación empieza a crecer, esta se puede dividir en un conjunto de piezas, conocidas como *world-chunks*. En la Figura 2.9 se puede observar un conjunto de objetivos relacionados entre sí asociados con un conjunto de piezas del mundo.

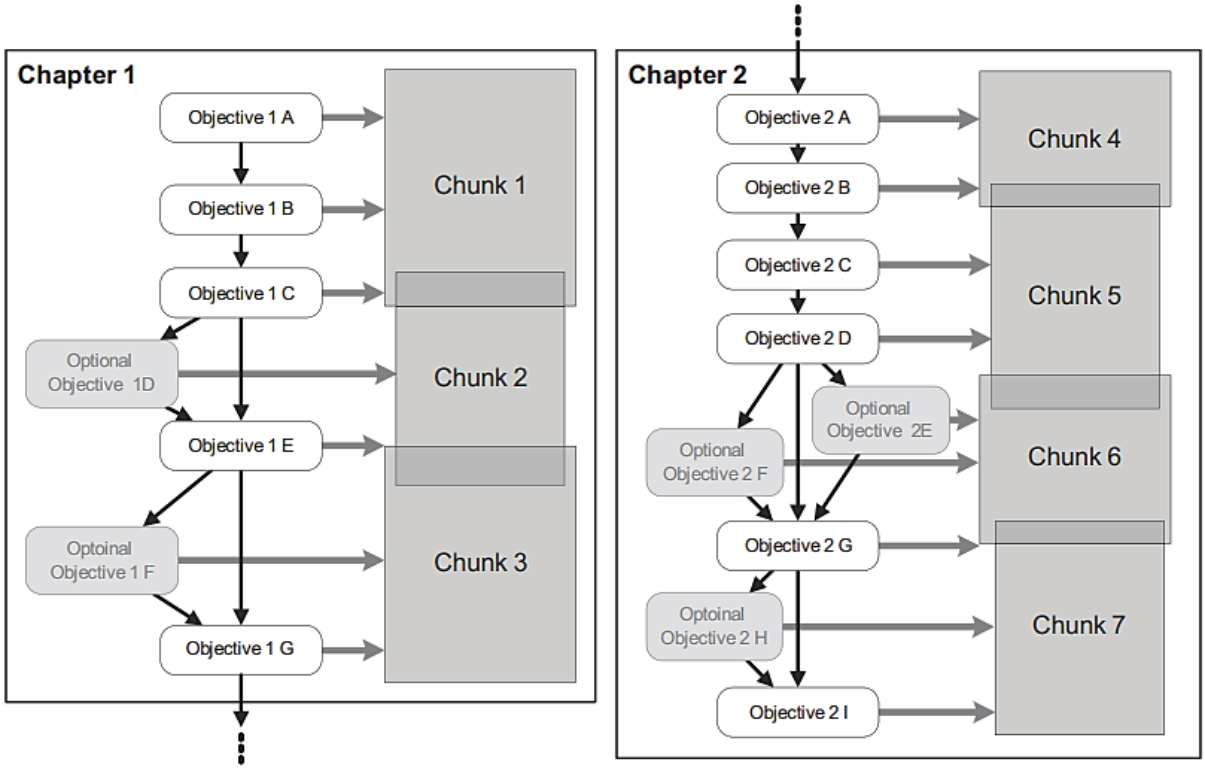


Figura 2.9 Objetivos de una aplicación por piezas [Gregory, 2009]

2.1.19 Subsistemas Específicos de la Aplicación

Esta corresponde la capa superior de la arquitectura del motor gráfico, en esta se definen varios sistemas de cámara, inteligencia artificial para el control de personajes que no son el jugador (NPCs), vehículos, etc. Si se pudiese establecer una línea clara entre el motor y la aplicación, esta se encontraría entre los subsistemas específicos y el gameplay.

2.2 Patrones de Diseño

Debido los diversos problemas que surgen cuando el código se empieza a volver cada vez más complejo, se han considerado un conjunto de patrones orientados a la programación de aplicaciones gráficas, estos patrones estructuran partes fundamentales correspondientes al diseño del motor gráfico, como también simplifican y mejoran el rendimiento de la aplicación.

Los patrones de diseño son formas bien conocidas y probadas de resolver problemas de diseño que son recurrentes en el tiempo. Los patrones de diseño son ampliamente utilizados en las disciplinas creativas y técnicas. De esta forma, la reutilización de estos componentes ayuda a mejorar el tiempo de diseño. Los patrones sintetizan la tradición y experiencia profesional de diseñadores de software experimentados que han evaluado y demostrado que la solución proporcionada es una buena solución bajo un determinado contexto [Vallejo y Martín, 2013].

A continuación se presentan algunos de estos patrones, estos se han orientado al desarrollo de aplicaciones gráficas y otros se han desarrollado principalmente con esa finalidad, cabe mencionar que se han implementado otros tipos de patrones que no se mencionan en las secciones siguientes, como son el patrón Singleton y el de Bucle Principal.

2.2.1 Prototype

Este patrón proporciona abstracción a la hora de crear diferentes tipos de objetos. Sin embargo, también ayuda a reducir el tamaño de la jerarquía de entidades que conforman un determinado modelo, permitiendo además crear nuevas entidades cargadas en tiempo de ejecución [Vallejo y Martín, 2013].

El problema radica principalmente en la sobrecarga que puede existir al momento de generar una mayor cantidad de modelos del mismo tipo, en donde incluso puede ser necesario cargarlos en tiempo real. La solución radica en atender las nuevas necesidades dinámicas en la creación de los distintos tipos de entidades, sin perder la abstracción sobre la creación de la misma.

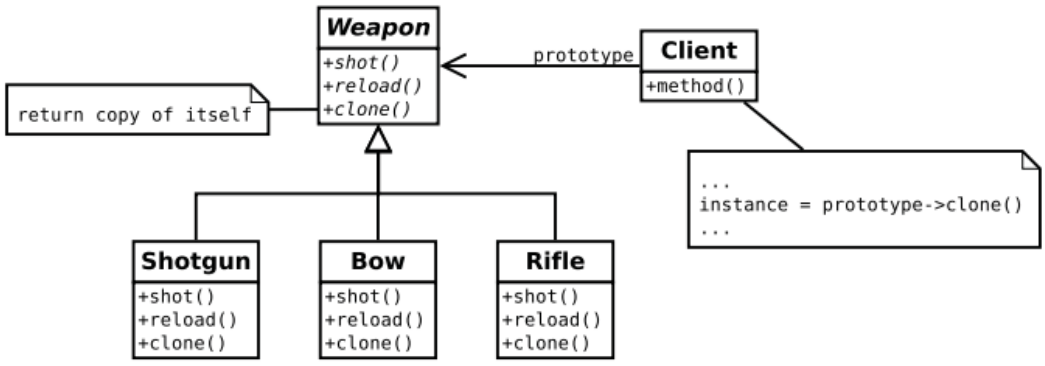


Figura 2.10 Patrón de diseño Prototype [Vallejo y Martín, 2013]

En la Figura 2.10 se puede observar la implementación del patrón de diseño Prototype, en donde se puede ver la incorporación del método *"clone()"*, el cual permite crear una instancia de la clase sin la necesidad de realizar la implementación de esta por cada elemento que se desea generar.

2.2.2 Flyweight

Este patrón de diseño consiste en la reutilización del conjunto de datos en común que podrían tener un listado de entidades. En el desarrollo del motor gráfico esto puede significar la utilización de una única textura o conjunto de vértices de una gran cantidad de modelos 3D, reduciendo considerablemente la sobrecarga inicial que significaría la lectura de dichos datos, así como también la cantidad de memoria requerida para trabajar con ellos.

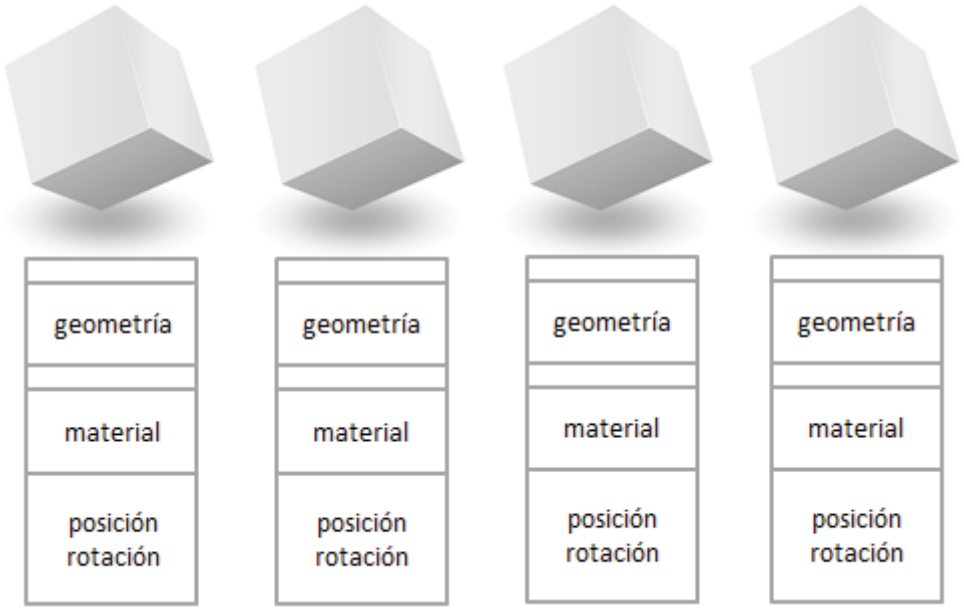


Figura 2.11 Entidades de un modelo 3D tradicional [URL 7, 2014]

En la Figura 2.11 se puede observar la típica implementación de un conjunto de modelos 3D, los cuales poseen su propia geometría, material, posición y rotación. Si se sabe de antemano que el conjunto de geometrías y materiales no cambiarán a lo largo de la aplicación, se podría considerar la reutilización de dichos elementos para facilitar la carga inicial que esto conlleva, así como también disminuir el procesamiento que realiza la GPU. Afortunadamente las APIs y tarjetas gráficas actuales permiten realizar esto mediante el denominado instanciado de geometrías, el cual puede ser realizado tanto por la API OpenGL como la API Direct3D. El instanciado de geometrías mejora el potencial de rendimiento en tiempo de ejecución de la instancia de geometría al permitir definir de manera explícita el número de copias de la malla a ser renderizada, permitiendo además especificar los parámetros diferenciadores de cada flujo de datos por separado.

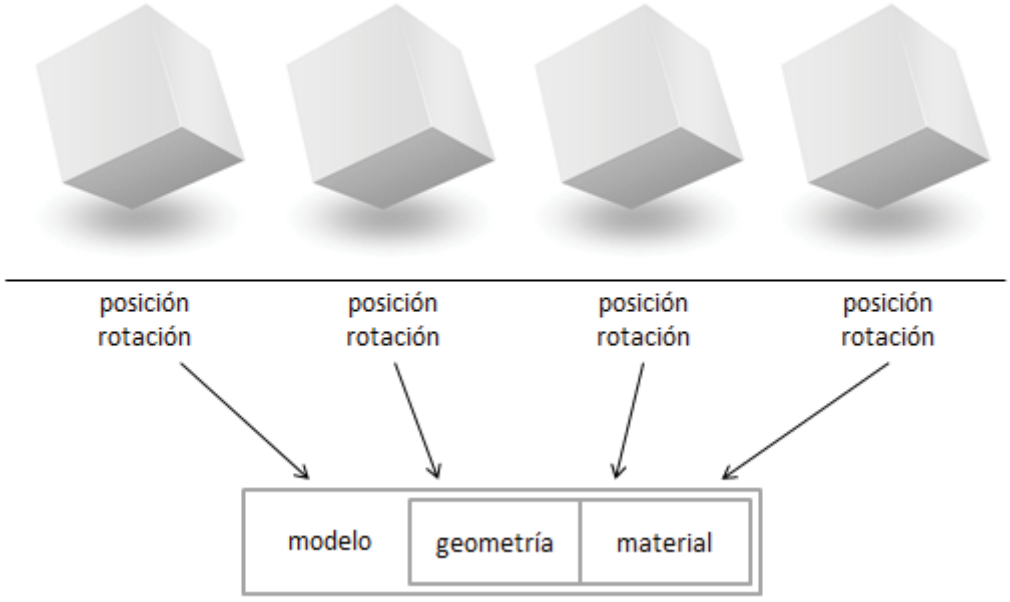


Figura 2.12 Patrón de diseño Flyweight [URL 7, 2014]

En ambas APIs gráficas se especifican dos flujos de datos. El primero es el conjunto de datos comunes (en la Figura 2.12 se pueden observar los datos de geometría y de material), y el segundo es la lista de instancias y sus parámetros a ser utilizados para variar el primer conjunto de datos que será dibujado en cada momento, el cual en el ejemplo corresponde a la información de posición y rotación del modelo. Con una simple llamada, se podrían generar una gran cantidad de entidades 3D haciendo uso del mismo set de datos genérico.

2.3 Facade

El patrón Facade eleva el nivel de abstracción de un determinado sistema para ocultar ciertos detalles de su implementación y hacer más sencillo su uso. Muchos de los sistemas que puedan ser incorporados en la aplicación con la finalidad de proporcionar un nivel de abstracción superior, basta que sean genéricos y reutilizables para que su complejidad aumente considerablemente y, por ello, su uso sea cada vez más complicado. Utilizando el patrón Facade, se proporciona un mayor nivel de abstracción al cliente de forma que se construye una clase *"fachada"* entre él y los subsistemas con un menor nivel de abstracción (ver Figura 2.13).

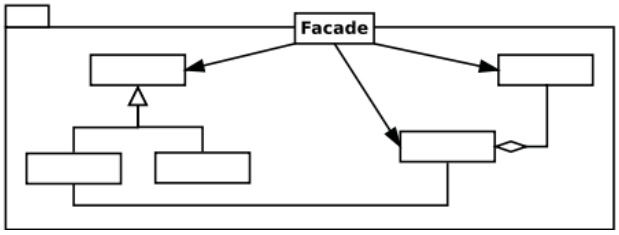


Figura 2.13 Ejemplo aplicación de patrón Facade [Vallejo y Martín, 2013]

2.4 Observer

Observer corresponde a un patrón de diseño de comportamiento, el cual se utiliza para definir relaciones 1 a *n* de forma que un objeto pueda notificar y/o actualizar el estado de otros automáticamente. Observer proporciona un diseño con poco acoplamiento entre los observadores y el objeto observado, en donde se sigue una filosofía de publicación/suscripción, lo que quiere decir que los objetos observadores se deben registrar en el objeto observado, también conocido como *subject*. Así cuando ocurra el evento oportuno, el *subject* recibirá una notificación y será el encargado de notificar a todos los elementos suscritos a él a través del método "*update()"*.

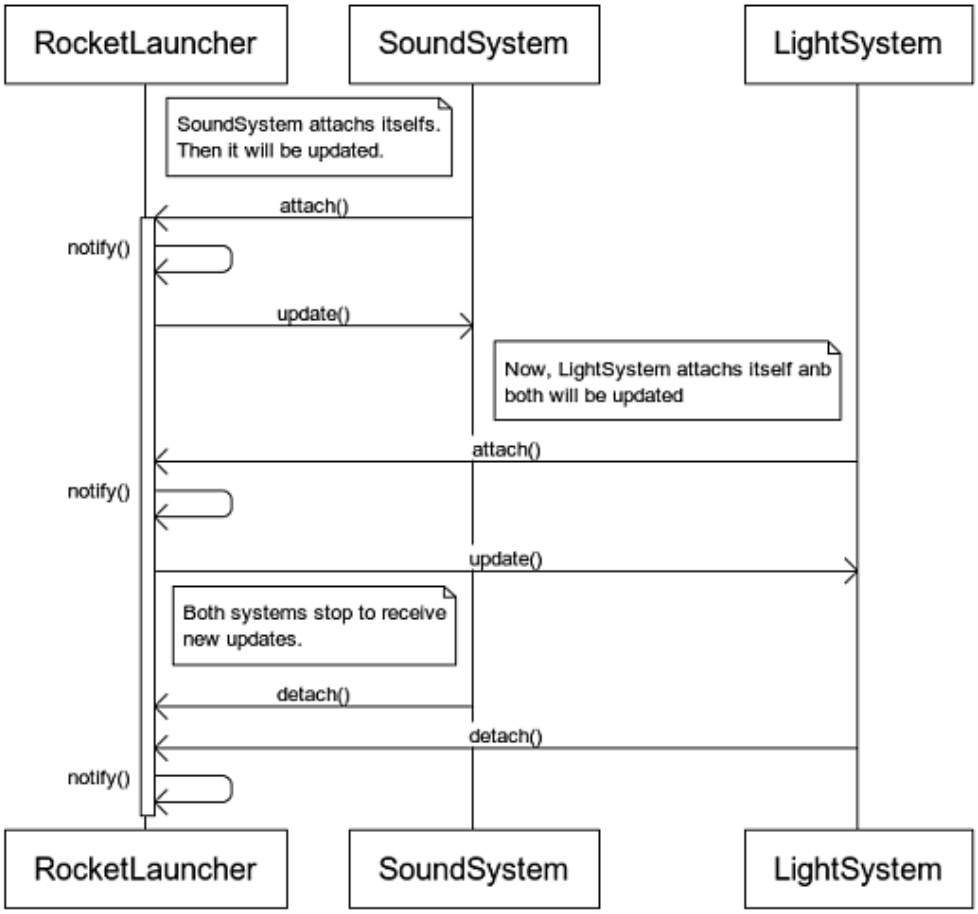


Figura 2.14 Diagrama de secuencia utilizando el patrón Observer [Vallejo y Martín, 2013]

En la Figura 2.14 se puede observar la implementación del patrón de diseño Observer mediante un diagrama de secuencia, en el cual se describe el orden de las invocaciones en el sistema. Los observadores se suscriben al subject mediante la operación "*attach()"* y a continuación reciben las actualizaciones. Para dejar de recibir dichas notificaciones hacen uso de la operación "*detach()"*.

Capítulo 3: Modelado 3D

3.1 Técnicas de Modelado

Actualmente existe una amplia gama de posibilidades a la hora de escoger una técnica de modelado, la mejor técnica depende muchas veces de lo que se desea realizar. Cabe destacar, que las diversas técnicas de modelado pueden utilizarse en conjunto para la creación de un solo modelo 3D. A continuación se mostrarán algunas técnicas para la creación de modelos 3D.

3.1.1 Modelado Poligonal

Esta es una de las técnicas más utilizada, principalmente por su rapidez y simplicidad a la hora de crear diversos tipos de superficie. El modelado poligonal consiste en la modificación directa de los vértices, aristas y polígonos de cualquier tipo de geometría. La más conocida de esta técnica es el *Box Modeling*, o modelado de caja, técnica que consiste en la creación del modelo comenzando con una caja como base. En la Figura 3.1 se puede observar cómo crear un pie a partir del desplazamiento de los vértices y la aplicación de algunas transformaciones en los polígonos.

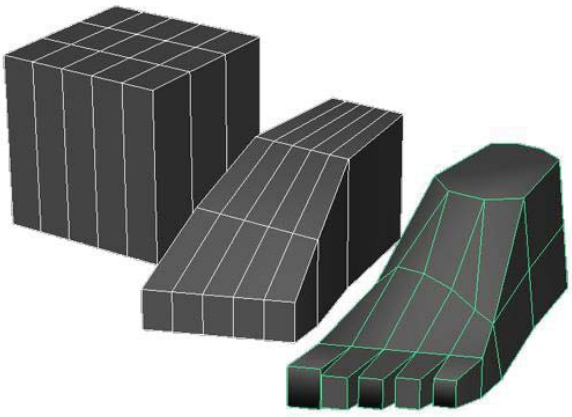


Figura 3.1 Ejemplo box modeling [URL 8, 2006]

Esta técnica resulta ideal para la creación de modelos que poseen una estructura rígida inherente, como pueden ser edificios, viviendas, muebles, o algunas superficies que no requieren de un gran detalle, entre otros.

Una de las principales dificultades de esta técnica es que se debe decidir la resolución que se le dará a la superficie con antelación, si más adelante se necesita aumentar la cantidad de polígonos será mucho más complicado.

3.1.2 Modelado mediante Splines

El modelado mediante splines o también llamado NURBS Modeling, consiste en la creación de modelos 3D a partir del uso de líneas vectoriales en 2D (splines) ubicadas en un espacio en 3D, a las cuales posteriormente se le aplican transformadores de malla para generar la superficie de la figura deseada, como puede observarse en el ejemplo de la Figura 3.2.

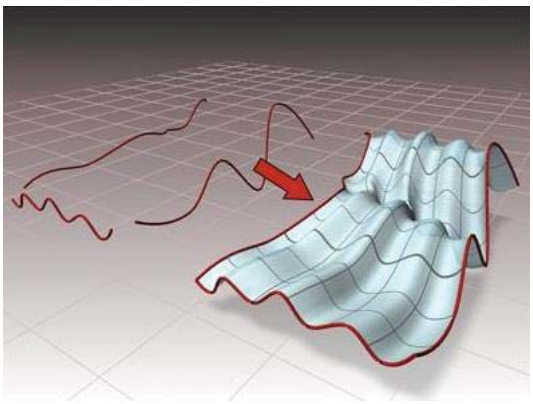


Figura 3.2 Ejemplo NURBS modeling [URL 9, 2004]

Esta técnica, a diferencia del Box Modeling, es mucho más adecuada para la creación de superficies orgánicas, es decir, modelos que posean una gran cantidad de superficies curvadas o con una complejidad media. El principal problema del modelado mediante splines son los *"perfiles"*. Los perfiles son la conexión y ubicación de las diferentes líneas vectoriales que compondrán el modelo, mientras más complejo y curvo sea el modelo, mayor será la cantidad de splines que se necesiten. En la Figura 3.3 se puede observar más claramente el perfil utilizado para la creación de una superficie.

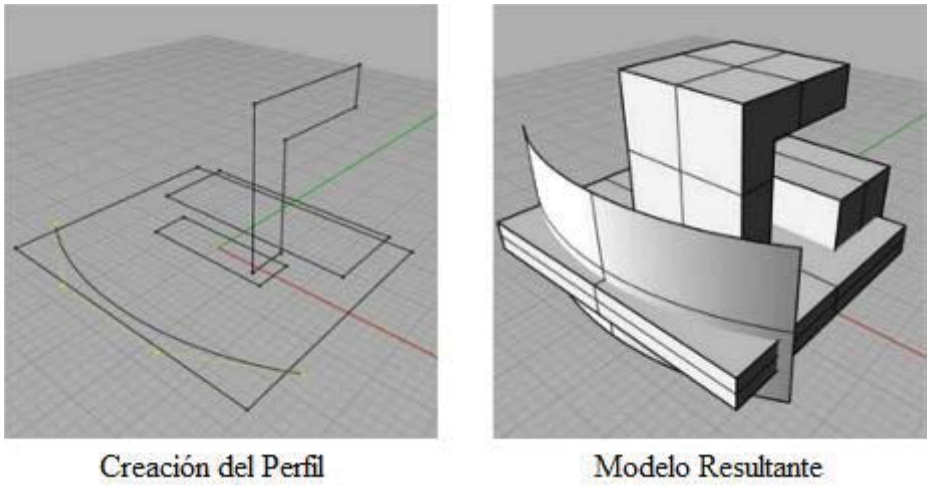


Figura 3.3 Utilización de un perfil [URL 10, 2011]

3.1.3 Digital Sculpting

Hace algunos años atrás, algunas aplicaciones para la creación de modelos 3D solían prometer la denominada *"Digital Sculpting"* o *"Escultura Digital"* para referirse a una interfaz en la cual las herramientas utilizadas para modelar eran tan intuitivas que daba la sensación de estar trabajando directamente con arcilla [Keller, 2011].

Esta técnica de modelado básicamente trabaja con mallas de muy alta resolución a la que se le aplican distintos tipos de herramientas para ir dándole forma. En algunos casos los diseñadores deben realizar modelos 3D profesionales pero manteniendo una baja cantidad de polígonos (Low Poly), una alternativa para lograr este requerimiento es crear el modelo mediante la técnica de sculpting, la cual posteriormente se convierte en un mapa de normales y se asigna al modelo en baja poligonización, aproximando de esta forma la gran cantidad de detalles que poseía la malla creada en un comienzo [Gregory, 2009].

Figura 3.4 Ejemplo de sculpting [URL 11, 2011]

En la Figura 3.4 se puede observar el proceso por el cual un modelo 3D empieza a tomar la forma deseada al ir moldeándolo y aumentando la cantidad de polígonos. A diferencia de otras técnicas de modelado, en el sculpting se puede comenzar a trabajar a partir de un modelo que ya se encuentra en alta poligonización, como puede ser por ejemplo una esfera con muchas caras.

3.2 Texturizado

El texturizado consiste en el proceso de tomar una superficie y modificar su apariencia mediante el uso de una imagen, función matemática o cualquier otro origen de datos. A este origen de datos se le denomina textura. Cada textura está compuesta por un conjunto de texeles, los cuales son representados mediante una matriz de texeles [Valient, 2001], [Glassner, 1989]. En la Figura 3.5 se puede observar claramente el cambio que se produce al aplicar una textura a una superficie en 3D, proceso denominado comúnmente como mapeado de textura.

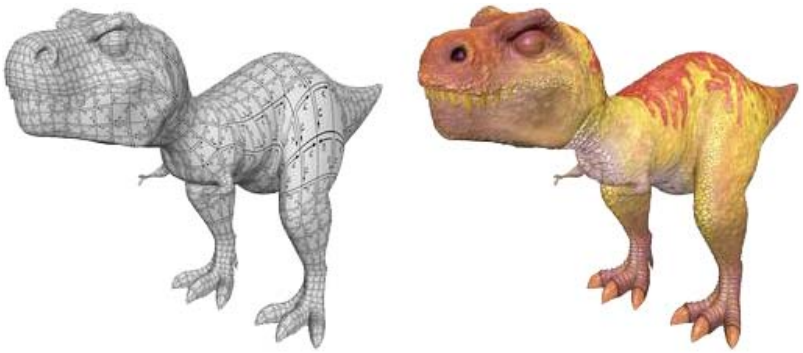


Figura 3.5 Ejemplo mapeado de textura [URL 12, 2008]

Además hay que considerar que el texturizado es una de las fases más importantes cuando se desea modelar un objeto en 3D, esto debido principalmente a que la superficie solo posee colores planos y en muchas ocasiones es necesario otorgarle un mayor grado de personalización y realismo.

3.2.1 Técnicas de Texturizado

Existen muchas formas para texturizar una superficie, a continuación se detallarán algunas de ellas.

3.2.1.1 Procedurales

Generadas mediante fórmulas matemáticas, buscan crear representaciones reales de elementos como: nubes, madera, metales, piedras, etc. Estas texturas pueden crearse a partir de dos tipos de mapas: *mapas geométricos* o *mapas aleatorios*. Los mapas geométricos poseen un patrón bien definido, por lo que no tienen aleatoriedad, mientras que los mapas aleatorios son impredecibles, pero suelen seguir un patrón regular (ver Figura 3.6).

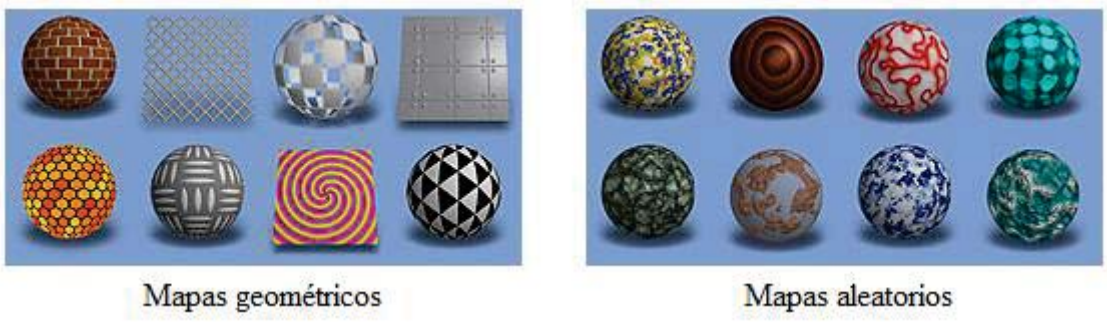


Figura 3.6 Ejemplo texturas procedurales [URL 13, 2009]

A continuación se describen algunas ventajas y desventajas de la utilización de texturas procedurales [URL 13, 2009]:

- Son densas: Esto quiere decir que no se pierde nitidez al aumentar la cercanía de esta a la cámara.
- Son infinitas: Debido a que se definen para todo el espacio, lo cual es muy útil cuando se debe texturizar modelos con un tamaño elevado.
- Ocupan poco espacio: Ya que no se almacenan en el disco como las imágenes.
- Difíciles de crear: Debido a que estas no se dibujan, sino que se crean mediante algoritmos matemáticos.
- Requieren un mayor procesamiento: A pesar de liberar el disco duro, la CPU ahora se debe encargar del procesamiento necesario para generar la textura.

3.2.1.2 Vertex Paint

El Vertex Paint, es una forma sencilla de pintar superficies de un modelo mediante la manipulación directa del color en los vértices [URL 14, 2012].

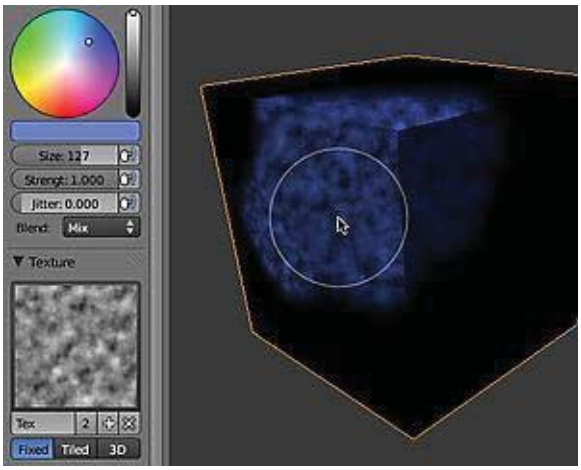


Figura 3.7 Ejemplo de vertex paint [URL 15, 2009]

Actualmente existen diversas herramientas que implementan esta técnica, esto lo realizan mediante la utilización de diversos pinceles, a los cuales se les pueden añadir diferentes propiedades, como rugosidad, opacidad, tamaño, color, intensidad, etc. En la Figura 3.7 se puede observar la utilización de esta técnica aplicada directamente a una superficie.

3.2.1.3 Texturizado por Imagen UV

Como su nombre lo indica, esta técnica se basa en la utilización de imágenes para generar la textura de un modelo 3D. Debido a que el tamaño de la textura influye directamente en el rendimiento que tendrá posteriormente

la aplicación, es necesario que esta tenga un tamaño lo más pequeño posible, considerando una proporcionalidad con el número de pixeles que se ocupará en la escena. Para el uso de memoria más eficiente, es recomendable utilizar texturas cuadradas, como 128x128, 256x256, 512x512, etc.

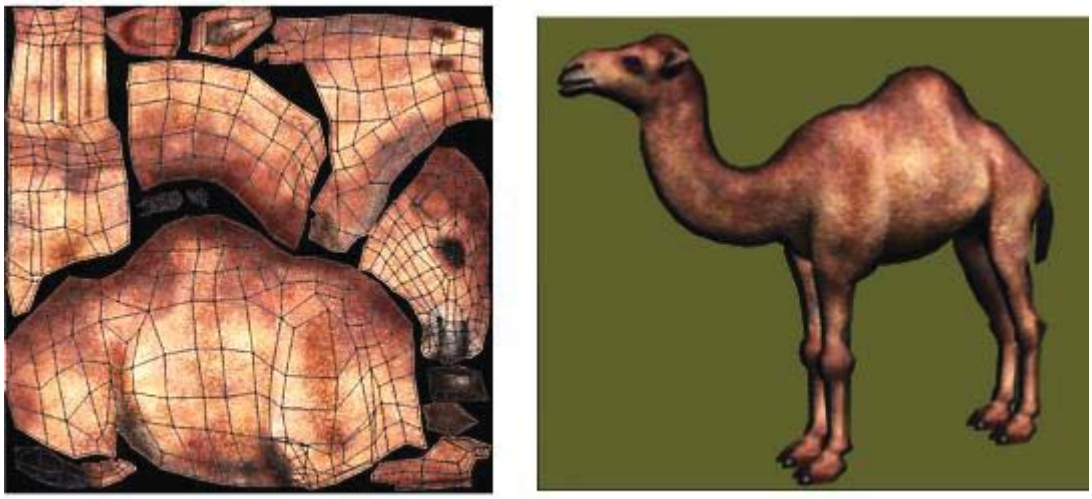


Figura 3.8 Texturizado mediante UV Map [URL 16, 2006]

La forma en que la imagen se adhiere a la malla, depende del modelo y la imagen que se utilice. Existen muchas formas de unir la imagen al modelo, una de las más utilizadas es el *UV Mapping*. Esta técnica consiste en la asignación de pixeles de la imagen 2D a la superficie del polígono 3D, mediante coordenadas UV (alternativo de XY). Cuando un modelo es creado como una malla poligonal utilizando un modelador 3D, las coordenadas UV pueden ser generadas para cada uno de los vértices de la malla, si luego el modelo quiere pasarse a un motor gráfico, es necesario que la geometría posea sus respectivas coordenadas UV. En la Figura 3.8 se puede ver como una imagen con coordenadas UV se aplica a la malla.

3.3 Listado Softwares de Modelado 3D

A continuación se describirán algunos de los softwares más utilizados para la creación de modelos 3D:

- **3D Studio Max**:

3D Studio Max es un programa de modelado, animación y renderización en 3D desarrollado por Autodesk [URL 17, 2014]. La utilización de 3D Studio Max permite al usuario la fácil visualización y representación de los modelos en una intuitiva interfaz, así como su exportación y salvado en otros formatos distintos a los utilizados por el propio programa. Este software otorga una serie de herramientas orientadas a diferentes perfiles, como desarrolladores de videojuegos, creadores de efectos visuales, diseñadores de gráficos de movimiento, arquitectos,

ingenieros, etc. 3D Studio Max es una de las herramientas más utilizadas profesionalmente, debido a su larga trayectoria y potentes capacidades, además de ser compatible con una gran cantidad de softwares externos.

- **Maya**:

Maya es un software de modelado 3D destinado principalmente a la animación cinematográfica [URL 18, 2014]. Al igual que 3D Studio Max, su actual desarrollador es Autodesk. Maya se caracteriza por su potencia, posibilidades de expansión, personalización y herramientas, además este software posee el MEL (Maya Embedded Language), código núcleo de Maya, con el cual se pueden crear diversos scripts y personalizaciones del software. Este software otorga herramientas para animación, modelado, simulación, renderizado, técnicas de movimiento, y composición.

- **Google SketchUp**:

Este software permite la creación de modelos 3D mediante herramientas intuitivas y fáciles de comprender. Está destinado a cualquier tipo de usuario, gracias a su gran cantidad de tutoriales, un completo centro de asistencia y a una comunidad global de usuarios. Gracias a SketchUp, se pueden construir modelos desde cero, o descargar los que se necesiten mediante la galería disponible en Google [URL 19, 2014].

A diferencia de otros programas de modelado 3D, en SketchUp, todos los modelos 3D están formados únicamente de dos elementos: aristas y polígonos. Además existen herramientas como el *"empujar/tirar"*, el cual permite extruir superficies planas a formas tridimensionales. Algunas características de este software son: coloreado directo en la superficie, dividir el modelo en secciones (para verlo por dentro), utilización de capas para ordenar las figuras, intercambio de información con Google Earth (debido a que pertenecen a la misma familia), entre otros.

- **Cinema4D**:

Desarrollado por Maxon, Cinema4D es un software de creación de gráficos y animación 3D de nivel profesional [URL 20, 2014]. Algunas características de este software son: implementación de materiales y texturas clásicas y avanzadas, herramientas de animación, permite importar y exportar a una gran variedad de formatos de archivo para integrar con muchos otros softwares, gran cantidad de posibilidades de iluminación y sombreado, un poderoso motor de renderizado, entre otros.

Algunas de las ventajas de Cinema4D son su interfaz altamente personalizable y flexible, lo cual permite adaptarse a la forma de trabajo del software más fácilmente; y una curva de aprendizaje bastante alta (en comparación con otros tipos de software para la creación de modelos 3D).

- **Zbrush**:

Software de Digital Sculpting y pintado de modelos 3D desarrollado por la empresa Pixologic [URL 21, 2014]. Zbrush se introdujo en el mundo como una aplicación de arte experimental con una tecnología única, la cual

era capaz de crear modelos con una gran cantidad de detalles y efectos. Los modelos con los cuales se trabaja en este software, son muy difíciles de realizar en los programas convencionales de modelado, debido a su arquitectura orientada a la implementación de técnicas de modelado poligonal.

Un flujo de trabajo típico, es el crear la base de la geometría del modelo en un programa de modelado convencional y luego exportarlo a software Zbrush para incrementar la cantidad de detalles.

- **Blender**:

Blender es un software libre, multiplataforma, y de código abierto. Actualmente es desarrollado por la Fundación Blender bajo licencia GPL [URL 22, 2014]. Actualmente, Blender es compatible con las plataformas Windows, Mac OS X, Linux, Solaris, FreeBSD e IRIX.

Este software ofrece un amplio espectro de modelado, animación, texturizado y video post-procesamiento, todo esto en un solo paquete. Algunas características claves de este software son [Hess, 2010]:

- Una suite de creación totalmente integrada, la cual otorga una amplia gama de herramientas esenciales para la creación de modelos 3D, entre las que destacan son: la creación de modelos mediante técnicas de modelado convencionales, soporte para Digital Sculpting, texturizado mediante Vertex Paint, entre otros.
- Peculiar interfaz gráfica, la cual es completamente flexible y adaptable a los diferentes requerimientos del usuario.
- Multiplataforma, debido a su interfaz gráfica basada en OpenGL.
- Gran calidad, rápida creación y eficiente flujo de trabajo.
- Gran cantidad de usuarios que utilizan este software.
- Tamaño ejecutable bastante reducido y de fácil distribución.
- Soporte para la creación de videojuegos mediante el lenguaje de programación Python.

Capítulo 4: Desarrollo del Proyecto

4.1 Caso de Estudio

A continuación se explican los diferentes puntos que componen el caso de estudio del presente proyecto, enfocándose en un comienzo en el estado del arte del proyecto, luego el conjunto de tecnologías a utilizar para el desarrollo del motor gráfico, posterior a esto el desarrollo del caso de estudio y finalmente las especificaciones del vehículo escogido para la simulación. Teniendo en mente en todo momento el cómo simular los diferentes eventos de acuerdo a los objetivos del proyecto.

4.1.1 Estado del Arte

Una gran parte de los simuladores actuales son desarrollos por empresas dedicadas únicamente a la simulación 3D, en donde ofrecen además de la aplicación misma, un entorno casi real que simula las condiciones con la que uno operaría realmente, este entorno puede considerar un conjunto de monitores, manubrios idénticos a los reales, sistemas de suspensión, aceleración por inclinación, entre muchos otros.

En [Johnson, 2009] se hizo uso del motor gráfico Torque Engine para simular en 3D el puerto de Seattle, el objetivo inicial era realizar un estudio de factibilidad y comprensión del proceso de producción de la simulación a través de un motor de juego de disparos en primera persona. El análisis se enfocó en un lugar del mundo real para representar más fielmente los problemas prácticos que esto involucra. Se evaluaron tanto el resultado gráfico de la simulación del puerto como también el rendimiento en base a la cantidad de modelos cargados en tiempo real. En la Figura 4.1 se puede observar la carga del modelo 3D al motor Torque Engine, en donde se produjo una pérdida de fidelidad del modelo respecto al entorno de modelado que se utilizó.

Figura 4.1 Carga de fidelidad del modelo 3D a la aplicación [Johnson, 2009]

La simulación 3D no solo permite establecer reglas para el aprendizaje de personal, sino que también puede jugar un rol importante en otras áreas, como es la detección de elementos nucleares a partir de sensores especializados. En [Christiansen *et al.*, 2008] se evaluó la detección de dichas amenazas en un puerto a partir de la simulación basada en un motor de videojuegos. Este considera sensores capaces de detectar un conjunto de ondas omnidireccionales emitidas por las amenazas, como puede observarse en la Figura 4.2, (a) corresponde a la amenaza, (b) las ondas que esta emite y (c) el sensor capaz de detectar dichas ondas. Este proyecto se realizó en base al motor OpenArena, el cual es un derivado del motor de Quake III Arena.

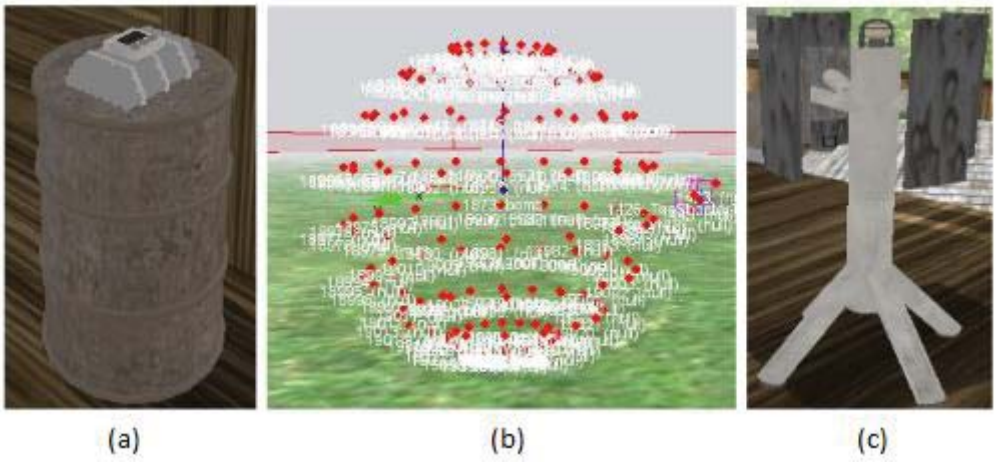


Figura 4.2 Detección de amenazas mediante sensores [Christiansen *et al.*, 2008]

Las simulaciones 3D permiten además presentar datos generados a partir de múltiples fuentes en el mismo entorno de ejecución, esto ayuda al proceso de aprendizaje y comprensión del entorno.

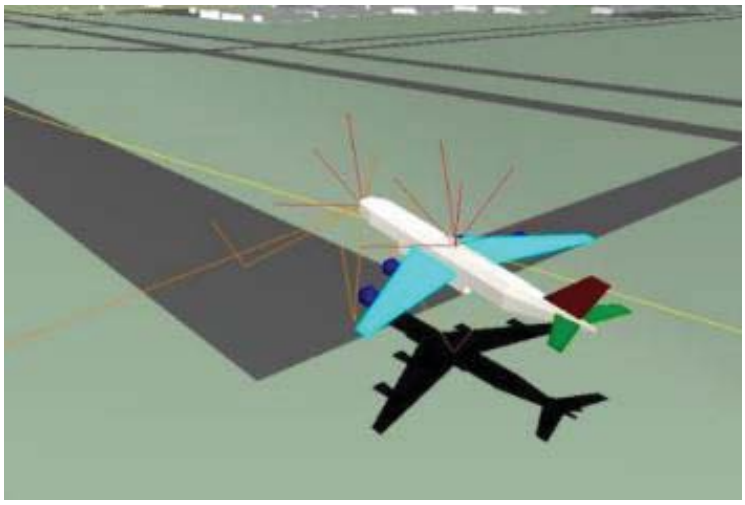


Figura 4.3 Visualización de datos de múltiples fuentes [URL 28, 2014]

En la simulación realizada por el Institute of Geodesy and Navigation [URL 28, 2014] se generó el conjunto de objetos del escenario en tiempo real, basándose en los diversos sistemas satelitales de navegación global (GNSS).

4.1.2 Tecnologías a Implementar

En este caso, se aplicarán diversas tecnologías y conceptos expuestos en capítulos anteriores con el fin de crear diversos prototipos incrementales, las tecnologías se han escogido a partir de un análisis de compatibilidad entre estas y las necesidades del proyecto. La mayor parte de estas se han escogido en base a las funcionalidades que otorgan y la cantidad de documentación disponible en base a su integración con el plataforma de desarrollo, como también la integración con las otras tecnologías.

Como se ha mencionado anteriormente, la API gráfica a utilizar corresponde a OpenGL, principalmente por las grandes capacidades gráficas que esta ofrece y por la posibilidad de portar el proyecto a diferentes plataformas, incluso a una plataforma Web. Además se ha considerado hacer uso de las nuevas características de OpenGL, por lo que se implementarán las funciones del nuevo pipeline, de este modo se podrá hacer uso de los diferentes shader mediante el lenguaje de programación de shader GLSL (OpenGL Shader Language).

El lenguaje de programación escogido es C++, esto por la compatibilidad que existe con una gran cantidad de librerías relacionadas con el desarrollo propio que conlleva un motor gráfico (librerías físicas, librerías de manejo de eventos, librerías para la creación de ventanas, librerías de sonidos, entre otras), y además por su extenso uso y documentación existente con la implementación de OpenGL en todas sus versiones (tanto las versiones que hacen uso del pipeline antiguo de OpenGL como el nuevo).

Considerando los elementos anteriores, para la creación y manejo de las ventanas se ha escogido SDL 2.0 (Simple DirectMedia Layer), el cual es una librería de desarrollo multiplataforma que provee un acceso a bajo nivel de audio, teclado, mouse, joystick y hardware gráfico mediante OpenGL y Direct3D [URL 23, 2014]. Esta librería está escrita en C y funciona de manera nativa con C++. En su versión 2.0 permite además la creación de diversas ventanas, lo que en un futuro podría permitir extender la visibilidad del simulador a un entorno multipantalla con pocas modificaciones al código fuente del simulador.

Para la carga de las imágenes que serán aplicadas mediante el texturizado por coordenadas UV a los diferentes modelos, así como también para cualquier otro tipo de imagen que se desee representar en el simulador (como elementos de GUI), se hará uso de la librería FreeImage [URL 24, 2014], la cual de manera simple permite cargar una gran cantidad de formatos de imagen (PNG, JPEG, BMP, GIF, entre otras), obteniendo de estas el conjunto de bits necesarios para los cuatro canales (RGBA) que luego serán leídos por OpenGL.

Para la lectura de las fuentes, se hará uso de la librería SDL_ttf 2.0 [URL 25, 2014], la cual permite generar una imagen a partir de un archivo de fuente TTF (TrueType) y un texto predefinido, generando finalmente una

imagen de manera similar a las imágenes generadas por SDL_image, permitiendo posteriormente cargar el conjunto de bits a OpenGL.

Para la realización de los diversos sonidos se ha optado por la librería SDL_mixer [URL 26, 2014], el cual otorga un conjunto de funcionalidades para la reproducción de contenido de audio, aceptando una gran cantidad de formatos. Además de esto, SDL_mixer permite implementar fácilmente sonidos y música al desarrollo de una aplicación en C/C++. Se ha optado por esta librería y no por OpenAL (Open Audio Library), principalmente por el bajo nivel de abstracción con el cual se trabaja en OpenAL.

Debido a que las funciones de OpenGL que permitían almacenar una pila de matrices tanto para la proyección como para la vista y los modelos se han establecido como obsoletas en la versión simplificada del pipeline, se hace necesario hacer uso de una librería matemática que facilite los diversos cálculos algebraicos intrínsecos en el desarrollo del motor gráfico. Es por esto que se ha escogido la librería GLM (OpenGL Mathematics) [URL 27, 2014], la cual posee una sintaxis bastante similar al lenguaje de programación de shader GLSL. Esta librería permite además generar de manera simple matrices de proyección tanto ortogonales como en perspectiva para la escena final del entorno 3D, además permite aplicar traslaciones, rotaciones y escalado a los modelos a través de matrices de transformación predefinidas.

Finalmente como motor físico se hará uso de Bullet Physics [URL 6, 2014], el cual permite generar el entorno físico que simulará los diversos movimientos y rotaciones de cada uno de los elementos que componen la escena 3D, así como también el cálculo de las diversas colisiones que se podrían efectuar en cada momento. Un elemento importante a considerar es la implementación de constraint, los cuales permitirán simular de manera mucho más realista los movimientos de las diferentes partes que componen la grúa Reach-Stacker, permitiendo aplicarle complejas conexiones a estas, intentando representar los movimientos que tendrían en un escenario real de la manera más cercana posible.

Cabe mencionar que el desarrollo de los diferentes modelos 3D que conforman el entorno del simulador se realizarán mediante el software de modelado Blender, ya que como se mencionó anteriormente, es gratuito y otorga una gran cantidad de herramientas que facilitan considerablemente todo el proceso de desarrollo que conlleva la creación de modelos 3D.

4.1.3 Desarrollo del Caso de Estudio

El proyecto estará compuesto por un escenario tridimensional en el cual se ubicarán los diferentes modelos 3D, estos representarán todas las entidades con las cuales podrá interactuar el usuario. Dicho entorno estará simulado a partir de los diversos fenómenos físicos que podrían ocurrir, por ejemplo, la caída de un contenedor por una mala ubicación o la caída de un conjunto de contenedores apilados por un posible choque con el vehículo.

Los objetos 3D que interactuarán de manera principal en el entorno son los contenedores y el vehículo Reach-Stacker. El vehículo estará separado en un conjunto de piezas las cuales serán conectadas mediante animaciones procedurales con la ayuda de la librería física, esto permitirá además aplicarle diferentes tipos de rotaciones a las partes de acuerdo a sus características físicas y limitaciones, todo mediante cálculos en tiempo real de los diferentes constraint con los cuales se unirán dichas partes, el movimiento de las ruedas también será calculado con la librería física, lo que permitirá desplazarse por superficies tanto planas como irregulares. Respecto a los contenedores, estos tendrán el peso correspondiente al tamaño de este (20ft o 40ft), y podrán ser apilados entre varios. Las cuatro esquinas de la superficie del contenedor tendrán puntos de conexión a partir de los cuales el Reach-Stacker podrá fijar la grúa, de este modo se podrá levantar el contenedor y desplazar a otro lugar, esto se piensa realizar incorporando cuatro figuras de colisión con forma de cubo en las esquinas del contenedor, las cuales deberán unirse con otras cuatro figuras de colisión correspondientes a los puntos de conexión de la grúa del vehículo, el cálculo de colisión entre estas cajas se realizaría mediante AABB, debido principalmente a su sencillo y rápido método de implementación; posterior a dicho cálculo si la conexión se encuentra establecida, se le aplicará el constraint correspondiente a cada uno de los puntos, conectándolo y aplicándole las físicas correspondientes a dicha unión.

Para mantener un mayor control del entorno visible del simulador se ha considerado la incorporación de un conjunto de cámaras de diferentes características, estas cámaras son:

- Una cámara en primera persona, la cual corresponde a la cámara principal del simulador, en donde se debería desenvolver principalmente el usuario que controle el vehículo. Esta cámara se encuentra fijada a los movimientos del vehículo.
- Una cámara libre, la cual se puede desplazar libremente por el entorno, recorriendo este sin depender del movimiento del vehículo. Podría utilizarse para fijar un punto de recolección de contenedores e indicar al usuario donde debe dejar dicho contenedor.

4.2 Diseño del Sistema

En este apartado se realizará la fase de diseño correspondiente al funcionamiento del motor gráfico y también del simulador, en donde se identificarán por una parte las características otorgadas al usuario del simulador, y por otra las funcionalidades realizadas de forma interna por el motor.

4.2.1 Diagrama Casos de Uso

A continuación se presenta el diagrama de casos de uso correspondiente a las diferentes acciones que puede realizar el usuario en el simulador (ver Figura 4.4), estas funcionalidades se detallan a continuación:

- Modificar Cámara:
    - Cambiar Cámara 1era Persona: Modifica la posición de la cámara y la ubica dentro de la cabina del vehículo, dando una perspectiva mucho más real de cómo se visualizan los diversos objetos en el entorno.
    - Cambiar Cámara Espectador: Modifica la posición de la cámara y la ubica afuera de la cabina del vehículo, en modo libre, permitiendo un mayor campo de visión del entorno.
- Desplazar Vehículo:
    - Avanzar: Desplaza el vehículo hacia adelante, en la dirección en que este se encuentre.
    - Retroceder: Da marcha atrás al vehículo según el sentido en que se encuentre.
    - Girar: Permite girar las ruedas traseras del vehículo.
- Manipular Spreader:
    - Rotar Spreader: Permite girar el spreader para alcanzar contenedores en ubicaciones difíciles de alcanzar cuando el vehículo no se encuentra totalmente alineado con dicho contenedor.
    - Extender Spreader: Permite extender el spreader para poder enganchar este con contenedores más grandes (específicamente contenedores de 40ft).
    - Encoger Spreader: Permite encoger el spreader para poder enganchar este con contenedores más pequeños (específicamente contenedores de20ft).
    - Enganchar Contenedor: Una vez se hayan acoplado las piezas correspondientes del contenedor con las piezas del spreader, se procede a enganchar dicho contendor.
    - Soltar Contenedor: Libera la carga que posea el spreader.
- Manipular Boom:
    - Rotar Boom: Permite aumentar o disminuir el ángulo del boom del vehículo, logrando de este modo alcanzar objeto a una altura superior.
    - Extender Boom: Permite extender el boom del vehículo, permitiendo alcanzar contenedores a una distancia más lejana.

- Encoger Boom: Permite encoger el boom, reduciendo la distancia necesaria para alcanzar un determinado objeto.

- Desplazar Cabina: Permite desplazar la cabina del vehículo hacia adelante o hacia atrás, esto permite tener una mayor visibilidad de las piezas del vehículo cuando la cámara del simulador se encuentra en 1era persona.

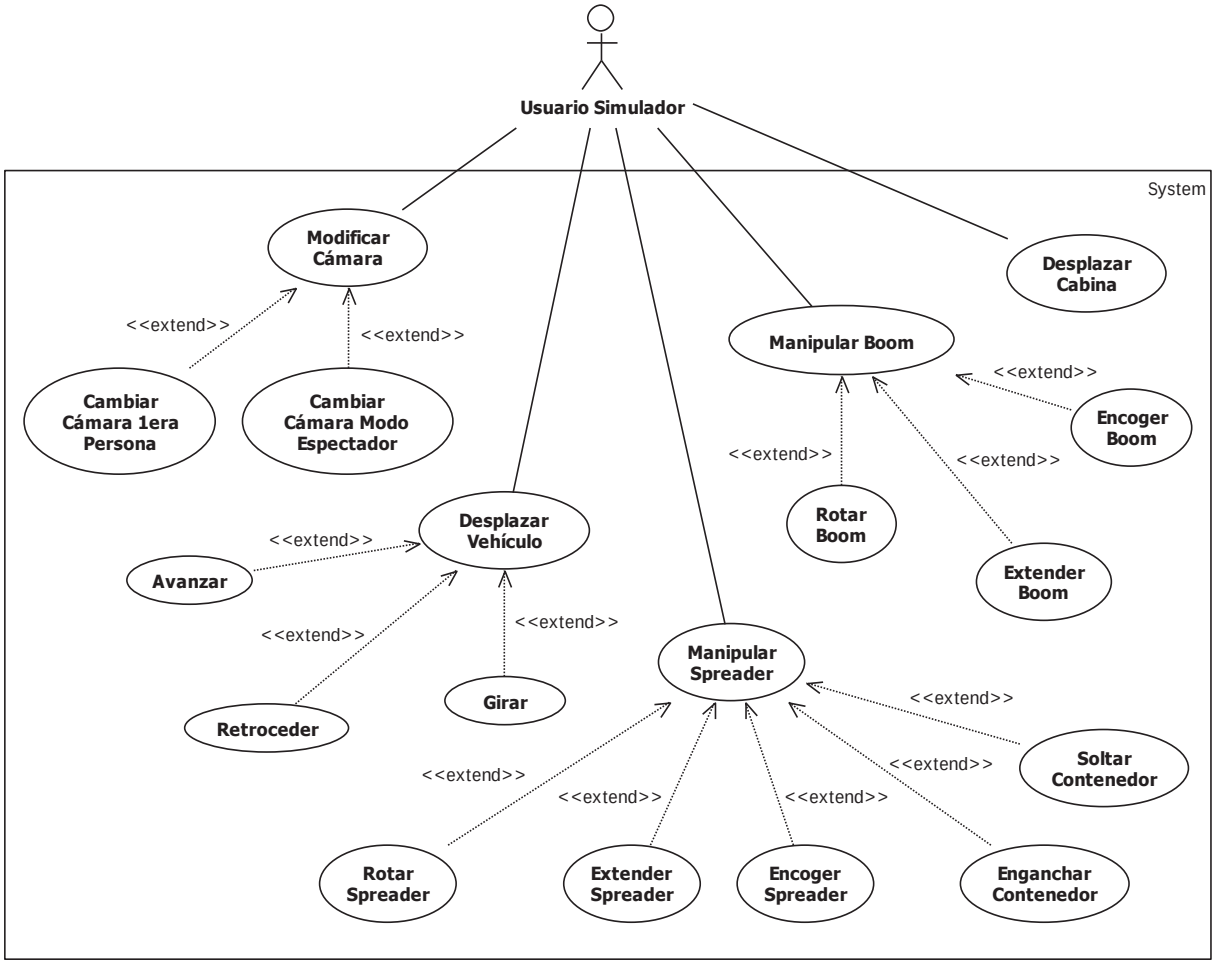


Figura 4.4 Diagrama casos de uso funcionalidades usuario [Elab. Propia]

4.2.2 Diagrama de Clases

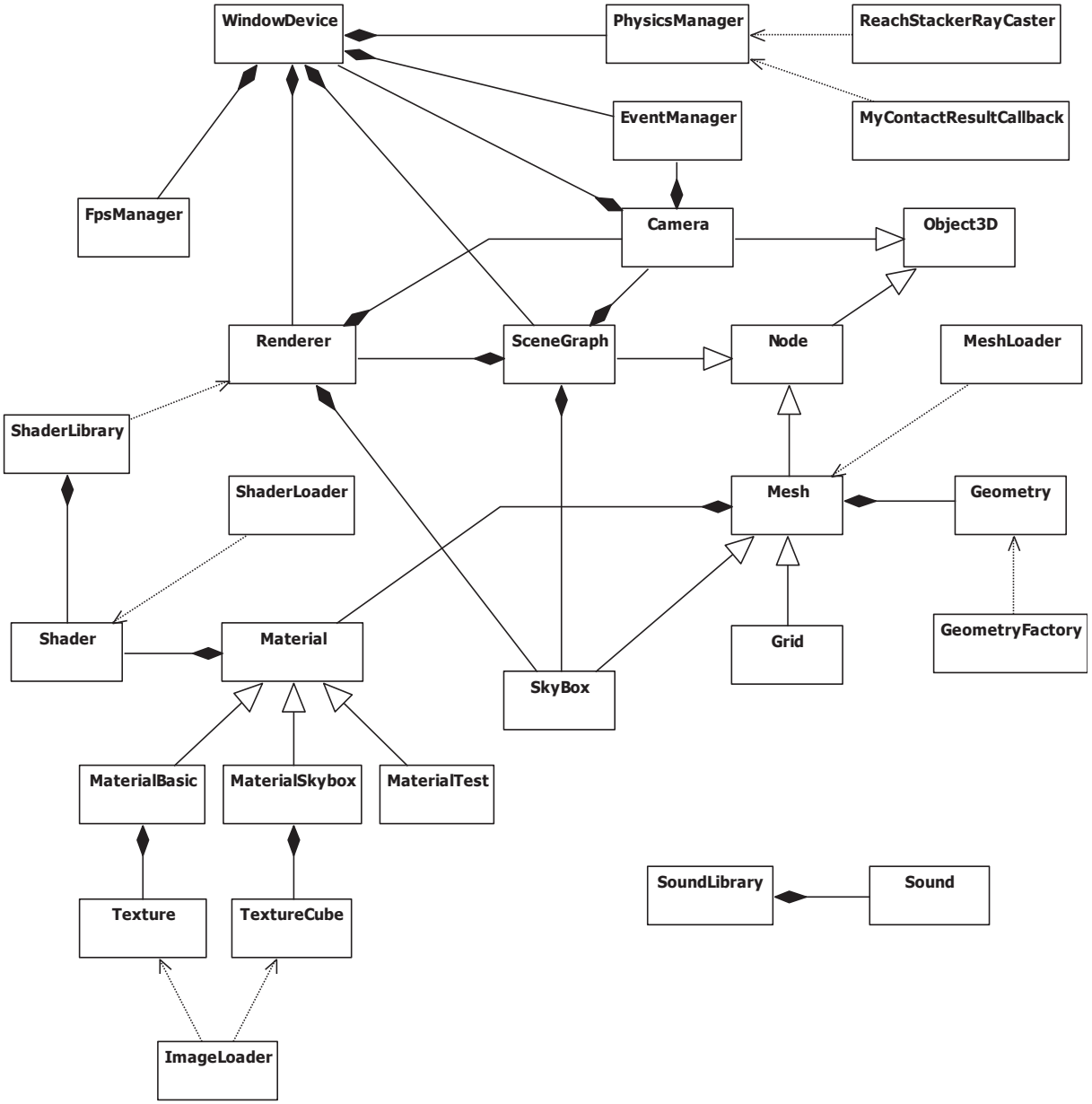


Figura 4.5 Diagrama de clases motor gráfico [Elab. Propia]

En la Figura 4.5 se pueden observar las diferentes clases involucradas en el funcionamiento del motor gráfico. A continuación se describen cada una de estas según su nivel de abstracción, en donde se presentan primero las que posean un nivel de abstracción superior:

- WindowDevice: Esta clase permite la creación y manipulación de la ventana, a partir de la cual se gestionan los diferentes administradores del motor gráfico, tales como el manipulador de eventos y el de las imágenes

por segundo. Esta clase corresponde al principal punto de comienzo del simulador, en donde se realiza la instancia de la ventana y se envían los diversos modelos 3D para ser posteriormente renderizados.

- Event Manager: Permite administrar de mejor forma los diversos eventos que son capturados por el administrador de la ventana, definiendo métodos que permiten saber si se han presionado alguna tecla del computador, si se ha desplazado el mouse o presionado un botón de algún dispositivo externo.
- FpsManager: Regula la cantidad de imágenes que son renderizadas en la pantalla por segundo.
- Renderer: Encargado de realizar el proceso de renderizado de los diversos objetos 3D del entorno, estableciendo los algoritmos necesarios para determinar cuáles figuras deben ser ocultadas, buscando mejorar el rendimiento final de la aplicación. En esta clase se calculan las coordenadas del mundo y la cámara, las cuales son enviadas a los diversos shaders para realizar el cálculo final de cada uno de los vectores de los objetos.
- Object3D: Administra la matriz de transformación de los objetos que serán representados en el escenario, esta matriz es multiplicada por cada uno de los vértices de cada objeto, determinando finalmente su posición y rotación. Del mismo modo, se almacena la matriz de transformación generada por el motor físico una vez realizado los cálculos de las diferentes físicas involucradas.
- Geometry: Almacena la información básica de posicionamiento, escalado y rotación de cada uno de los objetos del entorno. La clase *Renderer* hace uso de esta clase para poder generar la posición final del objeto.
- Camera: Establece el punto de vista que tiene el usuario en el simulador. Esta cámara puede establecerse en 1era persona o 3era persona. Gestiona principalmente las matrices de vista y perspectiva del motor gráfico.
- Node: Considerando la cantidad de elementos que se han incorporado al motor gráfico, se ha establecido como unidad de renderizado el nodo. A partir de este se pueden crear estructuras enlazadas con variada información, administradas principalmente por el SceneGraph.
- SceneGraph: Corresponde a una estructura de datos de árbol, en donde se define la raíz de este, y los métodos necesarios para renderizar recursivamente todos los nodos.
- Material: Clase base para la creación de los diferentes materiales que pueden ser aplicados a los objetos del escenario. Este almacena la información correspondiente al shader con el cual se renderizará finalmente el modelo.
- MaterialBasic: Material aplicado a los diferentes modelos 3D, este almacena la información de la imagen que será aplicada mediante coordenadas UV al modelo.
- MaterialSkybox: Material creado únicamente con la finalidad de ser aplicado al SkyBox, este almacena la información de las imágenes correspondientes a cada una de las caras del cubo a partir del cual se genera el cielo.
- MaterialTest: Material de prueba para la depuración del código.
- Texture: Almacena información de una imagen que es cargada posteriormente en memoria. Esta imagen es enviada a un shader, permitiendo aplicarla a los modelos según las coordenadas UV que este tenga.

- TextureCube: Almacena información de una imagen compuesta por 6 partes, las cuales son aplicadas finalmente al *SkyBox*.
- Shader: Almacena la información correspondiente a un programa shader, en donde se definen el vertex shader y fragment shader, a partir de los cuales se establece la ubicación final de cada vértice del objeto 3D, así como también la información relacionada con los diferentes pixeles que componen el objeto.
- ShaderLibray: Clase estática que permite acceder al conjunto de shaders cargados en el simulador. Utilizado principalmente por el renderizador del motor gráfico.
- Mesh: Clase principal para la creación de objetos 3D en el entorno, esta clase almacena la información correspondiente a los diferentes buffer del modelo, las referencias a su geometría y su material. Las diferentes componentes o piezas del Reach Stacker se modelarán a partir de la creación de varios objetos 3D, los cuales tendrán características físicas propias, y serán acoplados a partir de diferentes constraint otorgados por el motor de físicas Bullet.
- SkyBox: Permite la creación del cielo a partir de la textura de cubo.
- Grid: Define una grilla dividida por líneas, sirve para diferenciar el punto base del motor físico cuando no se ha establecido ningún escenario.
- ImageLoader: Utilitario que permite la carga de imágenes de distintos formatos, entre los que se encuentran GIF, JPG, PNG, BMP, entre otros.
- GeometryFactory: Utilitario para la creación de diferentes geometrías, entre las que se encuentran las geometrías de la grilla y cubos.
- MeshLoader: Utilitario para la carga de modelos 3D a partir de archivos, se ha realizado la implementación de los archivos OBJ.
- ShaderLoader: Utilitario para la carga de archivos de vertex shader y fragment shader.
- PhysicsManager: Permite añadir los diferentes objetos físicos al mundo (lo que es diferente a los objetos tridimensionales), realizando el cálculo de las colisiones en el mundo, creación de constraint o uniones entre los diferentes objetos, generar una aproximación de cuerpo de colisión a partir de los modelos tridimensionales, entre otros.
- ReachStackerRayCaster: Debido a la especificación de colisiones que existen entre los diferentes tipos de cuerpos en el mundo, se ha implementado una modificación en el algoritmo de *Ray Casting* realizado en las ruedas del vehículo.
- MyContactResultCallback: Permite detectar si dos cuerpos del motor físico se encuentran en colisión. Esta clase es utilizada principalmente para la detección de la colisión entre el spreader y los contenedores.
- Sound: Almacena una pista de sonido en un determinado canal de audio. Permitiendo reproducir y detener dicha pista.
- SoundLibrary: Gestor de los diferentes sonidos que van a ser reproducidos en la aplicación, simplificando la carga y asignación de los canales de audio a cada sonido.

A continuación se describirá de manera más detallada las clases principales del motor gráfico.

4.2.2.1 WindowDevice

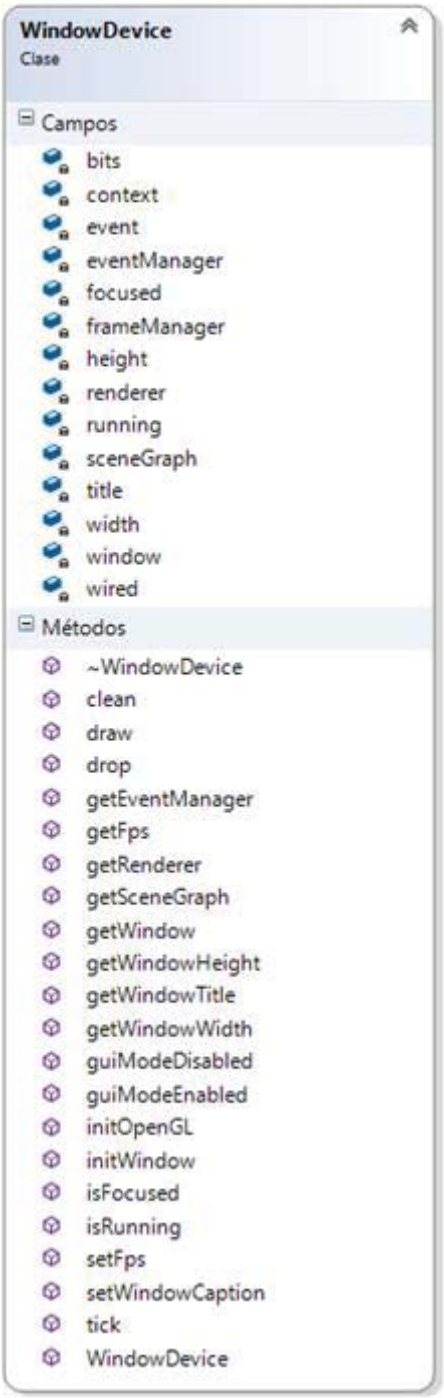


Figura 4.6 Clase WindowDevice [Elab. Propia]

Como se puede observar en la Figura 4.6, la clase *WindowDevice* gestiona los diversos administradores del motor gráfico, entre los que se encuentran: físicos, de eventos, de renderizado y de manejo de imágenes por segundo. Esta clase permite generar la ventana en donde funciona el canvas de OpenGL. Al momento de generar la ventana principal, se debe especificar el conjunto de configuraciones tanto de la ventana como de la API gráfica. Debido a que se está haciendo uso de la última versión del pipeline de OpenGL, el contexto de este debe adecuarse a dicho pipeline, permitiendo hacer uso de shaders. Una vez inicializada la clase, ésta genera automáticamente los administradores del motor gráfico, permitiendo hacer uso de estos en cualquier parte de la aplicación.

Otro punto importante de esta clase es la gestión del bucle principal de la aplicación, la cual puede ser llamada de manera externa, invocando los métodos correspondientes para la limpieza de buffers, captura de eventos, y el procesamiento y generación de la imagen final. Los métodos mencionados anteriormente corresponden son: *clean*, *tick* y *draw*. El método *clean* se encarga de limpiar los buffer de color y profundidad, posterior a esto invoca automáticamente la captura de eventos mediante la llamada del método *tick*, esta captura es enviada a la clase *EventManager* con la cual se puede trabajar posteriormente. Finalmente el método *draw* es el encargado de realizar el volcado de los buffer a la pantalla y la actualización del administrador de imágenes por segundo.

4.2.2.2 FpsManager

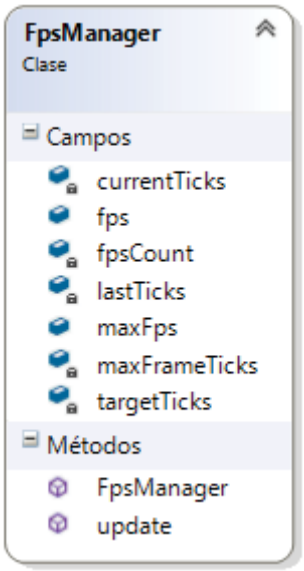


Figura 4.7 Clase FpsManager [Elab. Propia]

Debido a la gran velocidad de procesamiento que pueden realizar los procesadores, tanto del sistema como gráficos, la cantidad de imágenes generadas por segundo (fps) excede las necesarias, realizando reprocesamientos que podrían finalmente influir en el funcionamiento del motor gráfico. Considerando además que se está implementando un motor físico, el cual debe realizar cálculos que estén acorde a las imágenes presentadas por segundo, es necesario establecer un valor determinado de fps.

La clase *FpsManager* se encarga de regular el número de imágenes generadas por segundo, estableciendo un valor máximo para esta, de este modo las llamadas a los métodos del bucle de actualización son bastante inferiores, liberando una gran cantidad de carga a los procesadores tanto del sistema como gráficos.

4.2.2.3 EventManager

A partir de la actualización de los eventos realizados en la clase *WindowDevice*, se actualiza el conjunto de botones correspondientes a los códigos del teclado. Cada vez que un botón es presionado, este actualiza el conjunto de teclas almacenado en el vector *keyIsDown*, permitiendo consultar posteriormente por dicha tecla en cualquier parte de la aplicación.

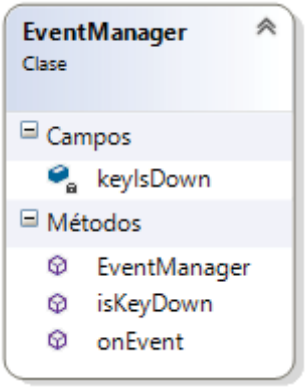


Figura 4.8 Clase EventManager [Elab. Propia]

4.2.2.4 Renderer

El *Renderer* es la clase encargada de renderizar las imágenes en el buffer que posteriormente será volcado en la pantalla. Inicialmente realiza la carga de los diferentes shaders a utilizar por la aplicación, ya sean vertex shader o fragment shaders. Al poseer un puntero a la clase *Camera* de la aplicación, se puede realizar el envío de las matrices de vista, modelo y proyección (MVP) a cada uno de los shaders utilizados por los modelos tridimensionales.

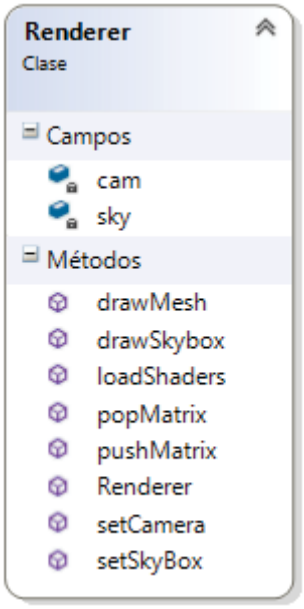


Figura 4.9 Clase Renderer [Elab. Propia]

4.2.2.5 PhysicsManager

La clase *PhysicsManager* almacena los atributos utilizados por el motor físico para la inicialización del entorno a simular.

A continuación se detallarán algunos de los atributos que deben ser previamente inicializados antes de hacer uso del motor físico:

- El *broadphase* es una etapa en el motor a partir de la cual se eliminan los pares de objetos que no colisionan entre sí utilizando un algoritmo sencillo (como el AABB), todo esto previo a realizar el cálculo denominado "narrow collide", el cual es mucho más lento debido a que considera el cuerpo de colisión exacto de la figura.
- El *collisionConfiguration* hace uso de los diferentes filtros de colisiones que son utilizados por la *broadphase*, permitiendo descartar colisiones que no serán realizadas por el resto del sistema.
- El *debugDrawer* permite generar modelos tridimensionales a partir de un depurador integrado que viene con el motor físico Bullet, enfocado principalmente al programador, debido a que son renderizados los cuerpos de colisiones de los objetos, formas y ángulos de los constraint, entre otros aspectos propios de cada entidad física.

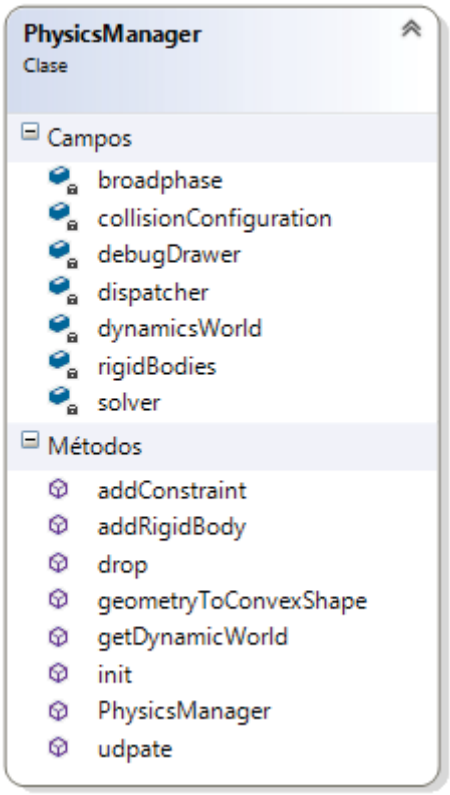


Figura 4.10 Clase PhysicsManager [Elab. Propia]

- El *solver* permite que los objetos interactúen entre sí correctamente, tomando en cuenta la gravedad, las fuerzas involucradas en los movimientos, colisiones, y constraints.
- El *dynamicsWorld* corresponde al mundo físico en donde se lleva a cabo la interacción entre todos los objetos entregados al motor físico.

Se ha implementado un método que simplifica en gran medida la generación de los cuerpos de colisiones para los diferentes modelos 3D que se vayan cargando en la aplicación. El método corresponde a *geometryToConvexShape*, al cual se le envía una geometría y devuelve como retorno una aproximación de cuerpo de colisión, esto debido a la gran cantidad de procesamiento que se requiere en la etapa de narrow collide para determinar si esta colisionando con otro objeto. Una vez generado este cuerpo de colisión, se puede agregar la figura 3D al mundo físico mediante el método *addRigidBody*.

4.2.2.6 Object3D

Debido a la naturaleza de los diferentes objetos que interactúan con el entorno tridimensional, se ha creado una clase que permite representar las transformaciones que estos objetos pueden sufrir en el tiempo. Considerando además que se ha implementado un motor físico, es necesario permitir hacer uso de las matrices de transformación

pre-procesadas por dicho motor, esto quiere decir, que una vez que los objetos han sido transformados físicamente, se puede obtener la matriz de transformación final de dicho objeto (posición, escalado y rotación) directamente desde el motor físico y aplicarlo al objeto correspondiente.

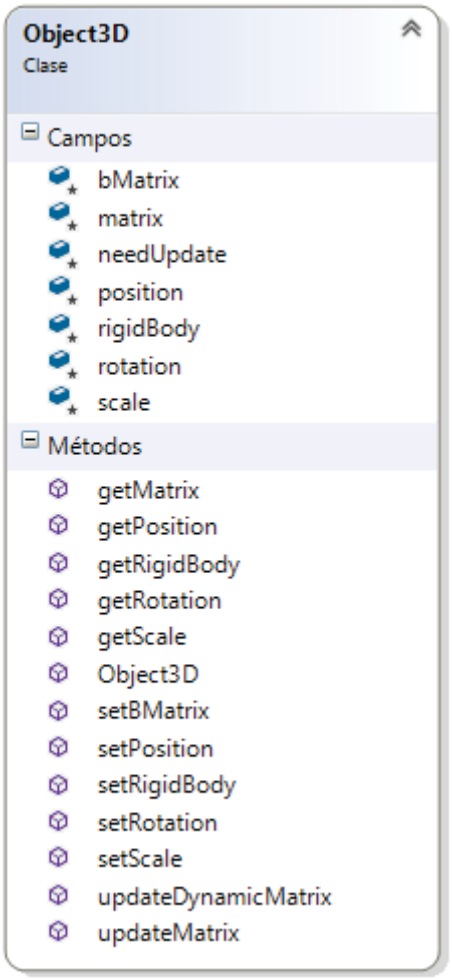


Figura 4.11 Clase Object3D [Elab. Propia]

Si el objeto no posee un rigid body (como la cámara), se le pueden asignar directamente sus transformaciones (*setPosition*, *setRotation*, *setScale*), o enviarle una matriz la cual puede tener estas transformaciones ya multiplicadas entre sí. En el caso que si posea un rigid body (como la mayoría de objetos 3D), simplemente se le asigna la matriz obtenida a partir del motor físico, el cual actualiza dicha información en el atributo *btRigidBody* correspondiente de dicho modelo.

4.2.2.7 Camera

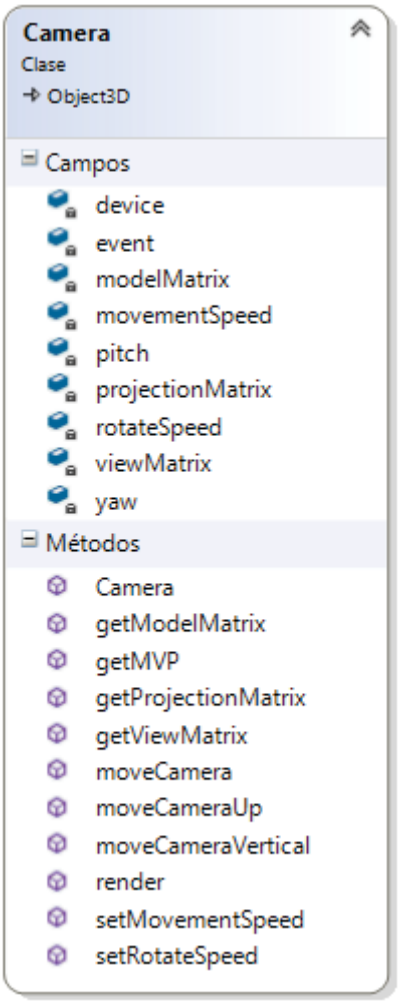


Figura 4.12 Clase Camera [Elab. Propia]

La cámara constituye uno de los objetos más importantes del motor gráfico, en esta se realiza inicialmente la creación de la matriz de proyección, la cual se establece como perspectiva, y las matrices de vista y modelo, las que conforman finalmente la matriz MVP, que corresponde a una matriz que almacena el resultado de la multiplicación entre las matrices mencionadas anteriormente. Esta matriz es esencial para que los objetos 3D sean posicionados gráficamente en el mundo 3D correctamente, una vez que al modelo 3D se le ha establecido su matriz de transformación, esta se multiplica por la matriz MVP, posicionando, rotando y escalando cada uno de los vértices del modelo en el vertex shader que le corresponde. Si se desea implementar un posicionamiento de esta en tercera persona, se puede hacer uso de los métodos que esta posee, en el caso que se desee un posicionamiento en primera persona, la posición de esta se encuentra determinada por el posicionamiento de la cabina del vehículo, por lo que se asigna directamente la matriz de transformación correspondiente al modelo de la cámara.

4.2.2.8 Mesh

Cada uno de los modelos tridimensionales cargados en la aplicación son finalmente representados por un *Mesh*, esta clase hereda de un *Node*, el cual es necesario para poder añadir los modelos al *SceneGraph*, clase encargada finalmente de enviar a renderizar uno por uno los modelos 3D.

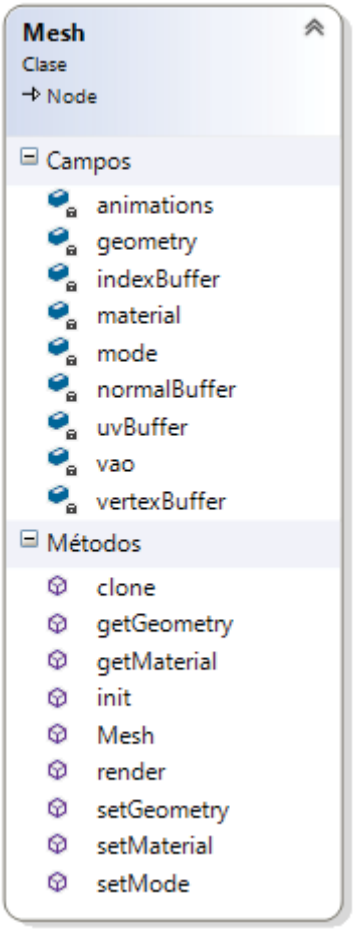


Figura 4.13 Clase Mesh [Elab. Propia]

Una vez cargada la geometría del modelo a partir del archivo Wavefront OBJ, se debe llamar al método *init*, encargado de asignar los diferentes atributos de los shader al modelo y generar los buffer de vértices, normales, coordenadas UV e índices. Estos buffer se almacenan en la GPU, por lo que su renderización es mucho más rápida.

Previo a la invocación del método *render*, la clase *Renderer* se encarga de actualizar los atributos de modelo, vista o proyección a enviar al shader correspondiente de dicho objeto, posterior a esto se invoca el método *render* del modelo. En este método se realiza un bind del shader a utilizar, lo que significa que las acciones de renderizado posteriores a dicho bindeo se realizarán haciendo uso de dicho shader.

4.2.2.9 Skybox

Esta clase permite la creación de un cubo que representa el cielo del mundo 3D. Básicamente es una clase *Mesh*, pero a la cual se le ha incorporado la posibilidad de especificar las dimensiones del cubo a generar. La aplicación de la textura en el modelo se obtiene a partir de la clase *TextureCube*, en donde se puede procesar el conjunto de imágenes almacenadas en memoria correspondientes a cada una de las caras del cubo.

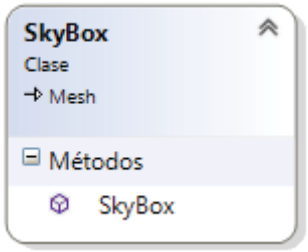


Figura 4.14 Clase SkyBox [Elab. Propia]

El skybox posee su propio shader, el cual permite recibir como parámetro una textura del tipo "samplerCube", la cual posee las seis caras del cubo. Debido a que este *Mesh* debe encontrarse posicionado siempre al centro de la aplicación, es decir, no ser afectado por alguna matriz de transformación que modifique su traslación, solo se le envían la matriz de proyección y de vista, obviando la multiplicación con la matriz de modelo.

4.2.2.10 Shader

Cada modelo 3D que va a ser finalmente renderizado debe hacer uso de un shader, cada shader posee sus propios parámetros, los cuales deben ser enviados previo a la llamada de renderización. Para la carga de un shader, se hace uso de la clase *ShaderLibrary*, la cual crea una instancia de shader, generando un programa que se compila y ejecuta en la GPU. En este proyecto se hacen uso tanto de vertex shaders como de fragment shaders, descartando los geometry shaders o tessellation shaders, ya que no son fundamentales para la renderización final de la imagen.

El envío de los argumentos de tipo *uniform* de cada shader se hace a través del método *transmitUniform*, el cual acepta diversos tipos de parámetros, entre los que se encuentran texturas, texturas de cubo, enteros, punto flotante, vectores, matrices, entre otros.

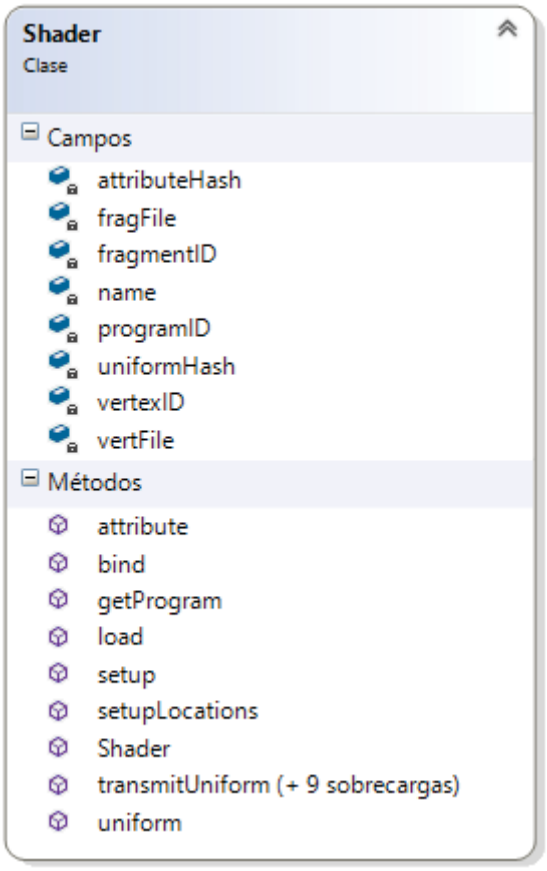


Figura 4.15 Clase Shader [Elab. Propia]

4.2.2.11 Material

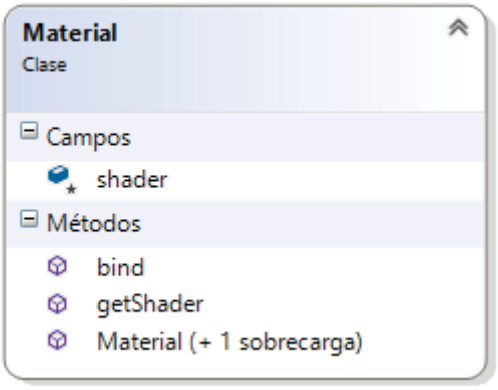


Figura 4.16 Clase Material [Elab. Propia]

Como última clase se presenta el *Material*, encargado principalmente de generalizar los diversos tipos de materiales que pueden ser aplicados a los modelos 3D, así como también de realizar el bind del shader

correspondiente. Existen diferentes tipos de material, entre los cuales se encuentra el *MaterialBasic*, *MaterialSkybox* y *MaterialTest*. El *MaterialBasic* corresponde a un material genérico para la mayor cantidad de modelos 3D que poseen una textura UV con su correspondiente mapa de bits, el *MaterialSkybox* es muy similar al anterior, pero es capaz de almacenar texturas del tipo *TextureCube*, la cual difiere de la clase genérica *Texture* en el método de almacenamiento de las imágenes. Finalmente el *MaterialTest* es un material que puede ser aplicado para realizar pruebas en los modelos debido a que hace uso de un shader que puede ser modificado sin afectar los otros shaders.

4.2.3 Especificaciones del Vehículo

En este capítulo se describirá el conjunto de especificaciones del vehículo Reach-Stacker seleccionado para la simulación 3D.

El Reach-Stacker seleccionado corresponde a un TCM MR450. Este vehículo es capaz de levantar contenedores con un peso máximo de 45 toneladas, con tamaños variables entre 20ft y 40ft. Este vehículo permite rotar los contenedores hasta un ángulo de 185° (ver Figura 4.17).

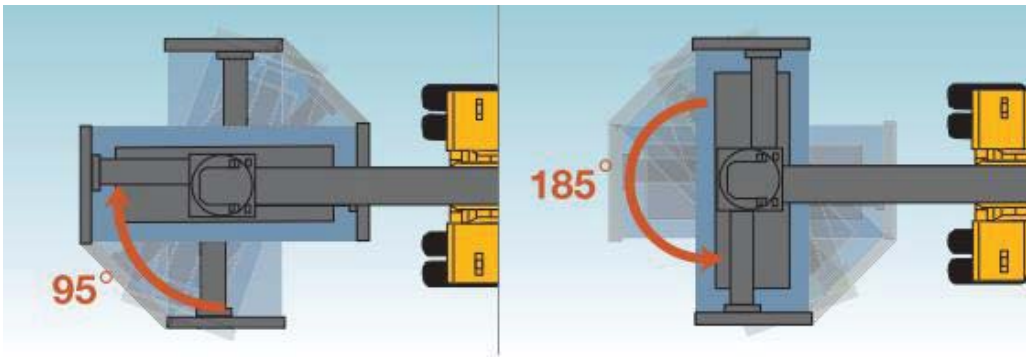


Figura 4.17 Ángulo de rotación del spreader [URL 2, 2014]

Para mejorar la visibilidad del conductor, este vehículo permite desplazar la cabina de mando alrededor de 2150mm a varias posiciones de adelante y atrás (ver Figura 4.18), esto permite aumentar la seguridad y eficiencia del trabajo realizado por el chofer.

Figura 4.18 Cabina deslizable [URL 2, 2014]

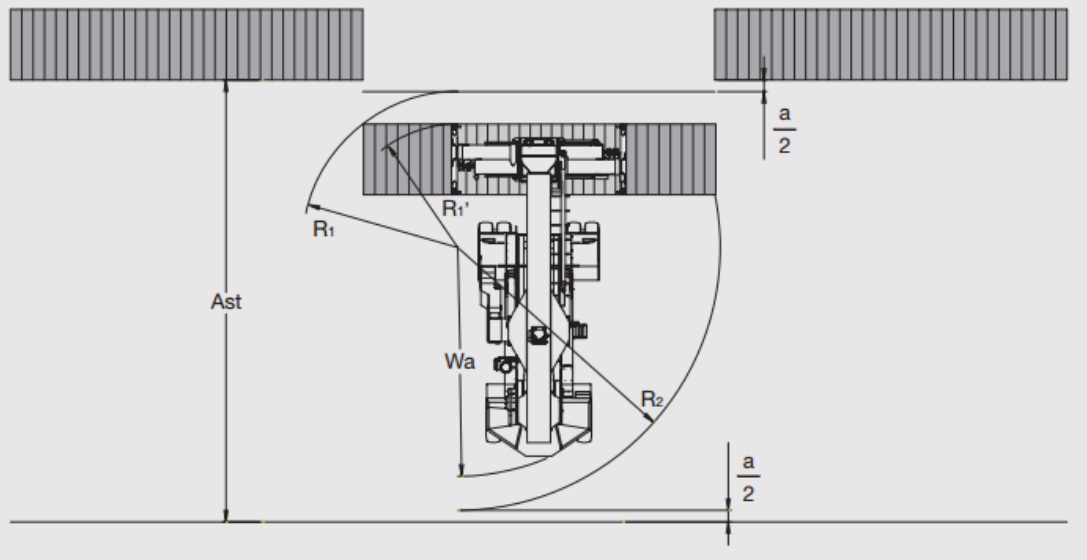


Figura 4.19 Esquema distancias respecto al ángulo de rotación [URL 2, 2014]

Las distancias correspondientes en la Figura 4.19 se pueden observar en la Tabla 4.1.

Tabla 4.1 Distancias respecto al ángulo de rotación [URL 2, 2014]

Modelo		**Tamaño (mm)**
Contenedor 20ft	$R_1{}'$	4300
	$\mathrm{Wa}(\mathrm{Wa} > R_2)$	8300
	Ast	13000
Contenedor 40ft	R_1	5400
	$R_2(\mathrm{Wa} < R_2)$	9100
	Ast	14900

Capítulo 5: Implementación

En este capítulo se realizará un análisis en base al cumplimiento de los diferentes objetivos del proyecto, evaluando en una primera instancia la creación e integración de las diferentes componentes que conforman el motor gráfico, luego se continuará con el desarrollo e integración de los diferentes modelos 3D que componen tanto el vehículo como el mundo, posterior a esto se realiza un análisis sobre la implicancia física relacionada con el mundo a simular, el sistema de colisión, los constraint, el sistema de conducción, y finalmente modificaciones anexas que se han realizado con la finalidad de mejorar la aplicación.

5.1 Motor gráfico

El desarrollo del motor gráfico se ha realizado basándose en otros tipos de motores gráficos, tales como Irrlicht [URL 29, 2014], CrystalSpace [URL 30, 2014] y Ogre3D [URL 31, 2014]. Cada uno de estos motores gráficos implementa diversas técnicas de programación gráfica que han sido utilizadas en el desarrollo de este simulador. A continuación se explicarán algunas de las técnicas que se han implementado en el motor gráfico.

5.1.1 Sistema de Coordenadas y MVP

El sistema de coordenadas se ha implementado considerando un área de clipping centrado en el punto de origen, es decir en el punto (0,0,0), esto simplifica las diversas transformaciones que se pueden realizar en los modelos tridimensionales. En la Figura 5.1 se pueden observar dos regiones de clipping, de las cuales se ha escogido la segunda de izquierda a derecha según el sistema de coordenadas cartesianas 3D.

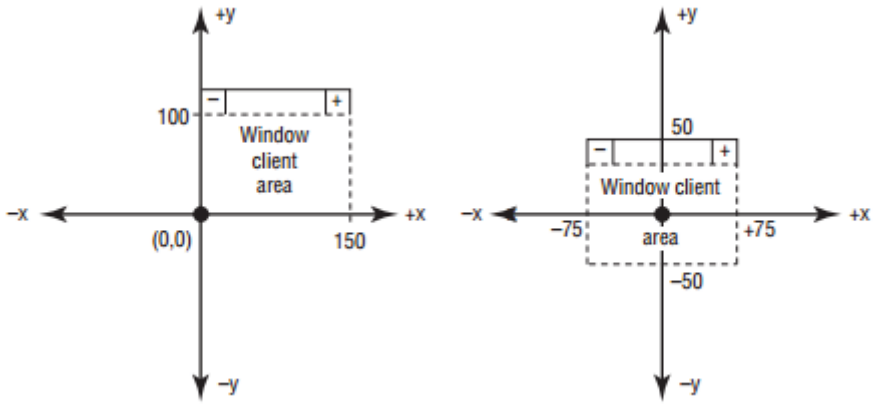
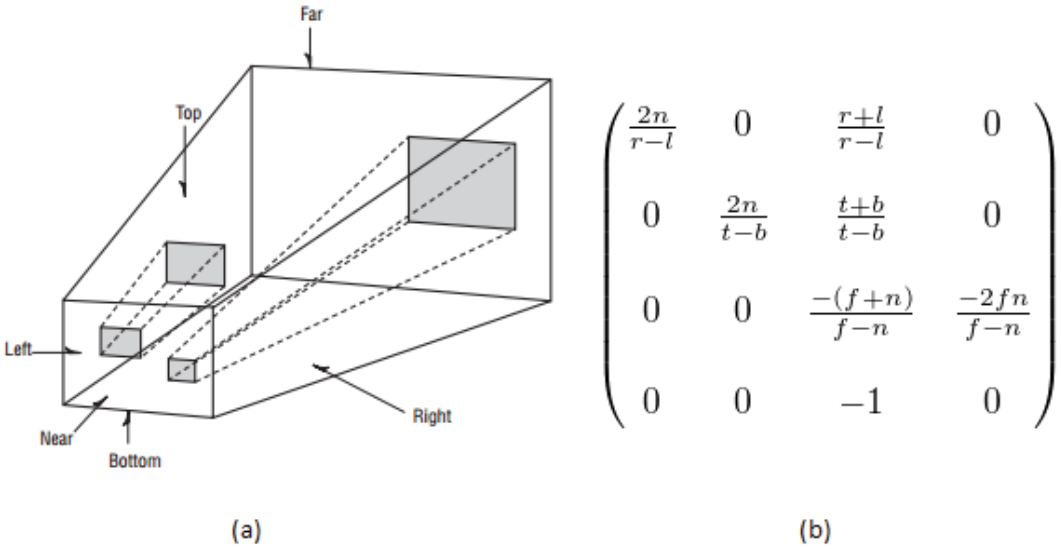


Figura 5.1 Regiones de clipping [Wright *et al.*, 2007]

Una vez establecido el sistema de coordenadas y clipping, se debe establecer el tipo de proyección que modificará los vértices de cada modelo que desee ser representado en la pantalla, así como también la matriz de vista y modelo, conformando finalmente la matriz MVP. En el antiguo pipeline de OpenGL, existen matrices predefinidas

y sistemas de pilas de matrices a partir de las cuales uno puede ir añadiendo o quitando matrices según sea conveniente. A pesar de la simplicidad que otorga este método, es poco extensible y difícil de modificar en un ambiente de desarrollo dinámico, es por esto que en el nuevo pipeline de OpenGL se han establecido como funciones obsoletas, permitiendo de este modo al usuario definir sus propias matrices y pilas, las cuales pueden ser enviadas a los shaders de manera independiente.



$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Figura 5.2 Matriz de proyección [Wright et al., 2007]

A partir de lo anterior, mediante la utilización de la librería matemática GLM, se han definido las nuevas estructuras de datos que permiten trabajar con vectores y matrices, además que esta librería posee matrices predefinidas, como es la matriz de proyección en perspectiva u ortogonal, la cual se puede observar en la Figura 5.2, en donde se presenta la vista que se obtiene a partir de la cámara (a), y su matriz de proyección correspondiente (b). Esta matriz es multiplicada por todos los vértices que posteriormente van a ser representados en la pantalla. Las matrices de vista y modelo inicialmente son la matriz identidad, lo que quiere decir que las multiplicaciones con estas matrices no afectarán el resultado. A continuación se presenta la inicialización de las tres matrices, realizada a partir de los métodos que otorga GLM:

```
// set projection matrix to perspective view
projectionMatrix = glm::perspective(
       glm::radians(45.0f),
       (float)device->getWindowWidth()/(float)device->getWindowHeight(),
       0.01f,
       1024.0f
);

// set model*view matrix for transformations
modelMatrix = glm::mat4(1.0f);
viewMatrix = glm::mat4(1.0f);
```

5.1.2 Texturas

Cada una de las texturas que son aplicadas a los materiales de cada modelo, son cargadas en memoria mediante la librería FreeImage, esta librería permite cargar una amplia variedad de formatos de imagen en diferentes profundidades, almacenando cada valor RGBA por separado y permitiendo obtener el alto y ancho de la imagen. En el caso de una textura plana con coordenadas UV convencionales, se realiza la carga de una sola imagen, se obtiene el conjunto de bits de esta, se carga en memoria y finalmente se genera la textura en OpenGL. A continuación se presenta el código que envía un set de bits a una textura en OpenGL:

```
glActiveTexture(texID);
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_2D, texID);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, bits);
```

Para establecer el tipo de textura, se debe definir el *target* de esta, la cual para una textura de dos coordenadas se establece como GL_TEXTURE_2D. El conjunto de parámetros definidos al momento de generar la textura permiten establecer el algoritmo de ampliación y reducción del pixel, así como también el algoritmo de envoltura que se aplica en las coordenadas UV. Finalmente la llamada a la función *glTexImage2D* permite generar la textura, la cual puede ser utilizada posteriormente realizando el bind de esta según su *texID*.

En el caso de la textura cúbica utilizada para la creación del SkyBox, es necesario cambiar el *target* a GL_TEXTURE_CUBE_MAP, en donde se envían parámetros similares a una textura convencional, con la diferencia que deben enviarse los mapas de bits de cada una de las caras. Debido a que el *target* enviado al método glTexImage2D es un conjunto de valores consecutivos, no es necesario declarar cada uno de estos por separado, permitiendo hacer uso de una sentencia *for* para su carga. A continuación la carga de un mapa de textura cúbica:

```
glActiveTexture(texID);
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_CUBE_MAP, texID);

glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

for(int i = 0; i < 6; i++)
       glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGBA, width, height, 0, GL_BGRA,
GL_UNSIGNED_BYTE, bits[i]);
```

5.2 Desarrollo e Integración de Modelos 3D

La mayor parte de los modelos que conforman el mundo tridimensional se han desarrollado mediante la aplicación Blender, el resto de los objetos se han obtenido a partir de bases de datos de modelos gratuitos. Cada uno de estos modelos se exporta finalmente al formato OBJ, debido a que la carga de modelos en la aplicación se hace mediante un importador propio que acepta únicamente este formato.

El modelo del Reach-Stacker se compone de varias partes, las cuales son unidas posteriormente en la aplicación. En la Figura 5.3 se puede observar el modelo del vehículo completo, en donde se encuentran de manera separada las partes que lo componen. Cada una de estas partes se ha realizado únicamente haciendo uso de la técnica de modelado poligonal, sin la necesidad de implementar NURBS o sculpting (herramientas que se encuentran disponibles en la aplicación de modelado).

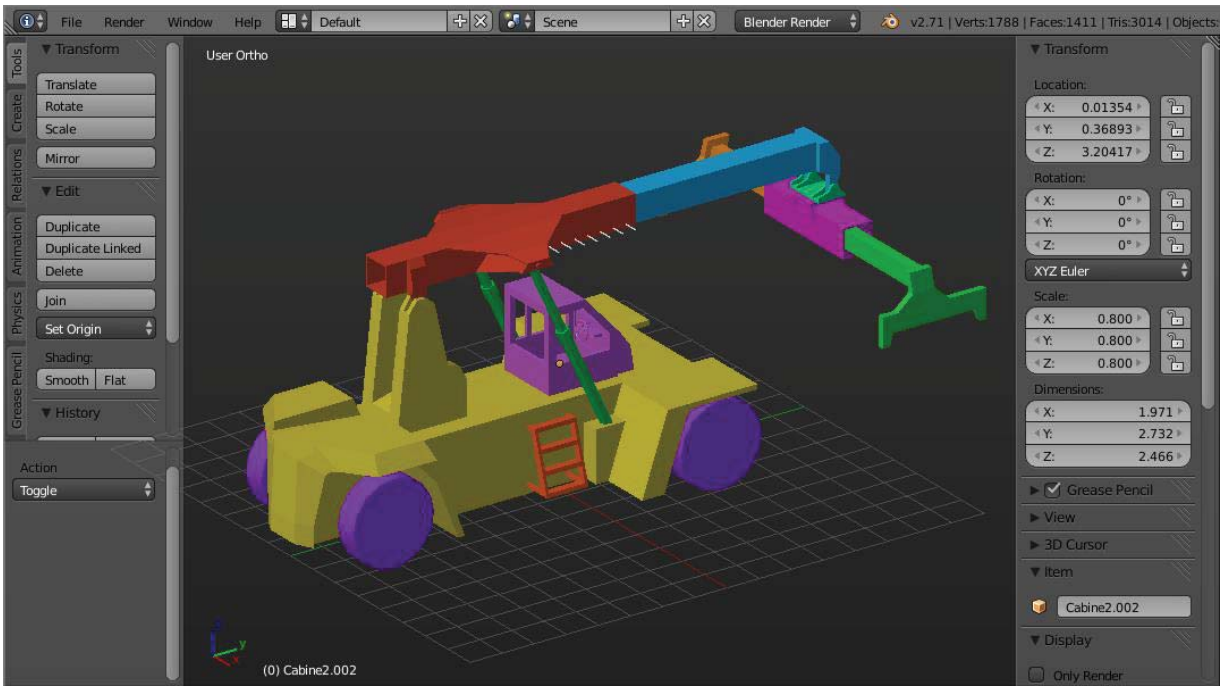


Figura 5.3 Modelo Reach-Stacker Blender [Elab. Propia]

Como se mencionó anteriormente, el formato de exportación utilizado para generar los diversos modelos del simulador es el Wavefront OBJ, principalmente por su simplicidad y el hecho de que no es necesaria una animación en los modelos. Este formato almacena la información de cada uno de los vértices del modelo, las normales de dichos vértices, las coordenadas UV de la textura y los triángulos que componen cada cara visible del modelo.

El motor gráfico permite la carga de este formato almacenando en vectores la información de cada elemento mencionado anteriormente, y un buffer que almacena el conjunto de índices de los vértices que componen cada triángulo.

A continuación se presenta un ejemplo de un modelo 3D almacenado en formato OBJ, correspondiente a una rueda del vehículo Reach-Stacker. El set de datos que se debe almacenar en un conjunto de vectores se identifica por la primera palabra de cada línea, en el caso de los vértices se denota por una "v", las coordenadas UV de cada vértice por un "vt", las normales por un "vn" y finalmente las caras de cada triángulo por una "f". De este modo, a medida que se va avanzando en el archivo se van guardando los diferentes vectores de datos hasta que se llega a las caras de los triángulos, en donde se deben ir formando a partir de los vectores de datos anteriormente guardados según el índice que la cara indique.

```
# Blender v2.71 (sub 0) OBJ File: 'reachstacker.blend'
# www.blender.org
o Cylinder_Cylinder.004
v -0.621132 0.000000 1.558658
v -0.621132 -0.596473 1.440012
v -0.621132 -1.102138 1.102138
...
vt 0.518199 0.681926
vt 0.217293 0.982106
vt 0.479564 0.873766
...
vn 0.000000 -0.831500 -0.555600
vn 0.000000 -0.555600 -0.831500
vn 0.000000 -0.195100 -0.980800
...
f 58/86/50 74/103/50 75/104/50
f 57/85/50 56/88/50 72/105/50
f 54/98/50 70/106/50 71/107/50
f 52/92/50 68/108/50 69/109/50
f 64/90/50 63/94/50 79/110/50
...
```

En la primera cara se puede observar el set de datos "58/86/50 74/103/50 75/104/50", lo que quiere decir que el vértice número uno corresponde al 58, tiene la textura 86 y la normal 50, luego el vértice número dos corresponde al 74, tiene la textura 103 y la normal 50; y finalmente el vértice tercero corresponde al 75, tiene la textura 104 y la normal 50. En la mayor parte de los casos las normales de los tres vértices que componen una cara es la misma, pero esto podría cambiar según como se trabaje el modelo 3D en la herramienta de modelado que se utilice.

5.3 Integración Motor Físico

Una vez se ha establecido un sistema de posicionamiento, rotación y escalado para los diversos modelos 3D, la incorporación de un motor físico reutiliza dichos valores y los modifica según los diversos eventos o acciones que se lleven a cabo en el mundo físico. El mundo físico corresponde a un sistema independiente al entorno gráfico el cual va realizando constantemente cálculos en base a los cuerpos físicos que se encuentran en este, y del cual una vez realizados los cálculos correspondientes se puede obtener información de las matrices de transformación de dichos cuerpos. Cada uno de estos cuerpos corresponde a aproximaciones geométricas de los diversos modelos 3D que se han sido cargados en el motor gráfico.

Bullet Physics otorga un conjunto de cuerpos de colisión que se podrían adaptar a ciertos modelos 3D como aproximaciones geométricas de estos. Algunos de estos cuerpos de colisión corresponden a cubos, esferas, tubos, entre otros. A pesar de lo anterior, se ha optado por implementar un método que permita realizar una aproximación más realista del cuerpo de colisión de la mayor parte de los modelos del simulador, basándose principalmente de la geometría original del modelo.

En el siguiente algoritmo se puede observar la generación de un *btConvexShape*, el cual corresponde a un cuerpo de colisión formado por a partir de la geometría del modelo 3D. El resultado de la generación de dicho cuerpo de colisión es procesado nuevamente para simplificarlo (lo que permite realizar cálculos de manera mucho más rápida), retornando finalmente el cuerpo de colisión aproximado para dicho modelo.

```
btConvexShape* PhysicsManager::geometryToConvexShape(Geometry *geometry){
	// generate triangle mesh
	btTriangleMesh *triMesh = new btTriangleMesh();

	for(int i = 0; i < (int)geometry->indices.size(); i += 3){
		glm::vec3 vertexA = geometry->vertices[geometry->indices[i]];
		glm::vec3 vertexB = geometry->vertices[geometry->indices[i + 1]];
		glm::vec3 vertexC = geometry->vertices[geometry->indices[i + 2]];

		btVector3 btVertexA(vertexA.x, vertexA.y, vertexA.z);
		btVector3 btVertexB(vertexB.x, vertexB.y, vertexB.z);
		btVector3 btVertexC(vertexC.x, vertexC.y, vertexC.z);

		triMesh->addTriangle(btVertexA, btVertexB, btVertexC);
	}

	// convert triangle mesh into convex shape
	btConvexShape *tmpShape = new btConvexTriangleMeshShape(triMesh);

	//create a hull approximation
	btShapeHull *hull = new btShapeHull(tmpShape);
	btScalar margin = tmpShape->getMargin();
	hull->buildHull(margin);
	tmpShape->setUserPointer(hull);

	return tmpShape;
}
```

Una vez se posee el cuerpo de colisión para el modelo 3D, este debe ser incorporado al mundo físico, para esto se debe generar un *btRigidBody*, definir la masa del cuerpo, su inercia, rotación, posicionamiento y fricción. El centro de masa del cuerpo es preferible definirlo en la herramienta de modelado, el cual corresponde al punto de origen del sistema de coordenadas cartesianas, es decir, el punto (0, 0, 0).

Si el modelo 3D ha sido inicializado mediante la creación de *btRigidBody* y luego se le ha añadido al mundo físico, la matriz de transformación que finalmente es enviada al shader que le corresponde se puede obtener a partir del método *getOpenGLMatrix* otorgado por Bullet Physics. Este método retorna una matriz de transformación ya procesada, es decir, una matriz que posee la rotación, traslación y escalado final que debe aplicarse al modelo.

A continuación se presenta la forma de obtener dicha matriz:

```
btTransform transform;
rigidBody->getMotionState()->getWorldTransform(transform);
transform.getOpenGLMatrix(glm::value_ptr(matrix));
```

5.3.1 Máscaras de Colisión

Debido a que ciertos cuerpos es preferible que no colisionen entre sí, en especial las partes que componen el Reach-Stacker, es necesario definir ciertas máscaras de colisiones que descartan validaciones de colisión entre cuerpos en tempranas etapas del pipeline del motor físico. Las máscaras de colisión se declaran mediante una estructura de datos sencilla, en el algoritmo presentado a continuación se puede observar la declaración de un conjunto de máscaras que permiten agrupar los diferentes tipos de cuerpos mediante un nombre, en donde posteriormente son utilizados para declarar el comportamiento que lleva a cabo de un cuerpo de colisión con los demás.

```
#define BIT(x) (1<<(x))

enum collisiontypes {
	COL_NOTHING = 0, // collide with nothing
	COL_GROUND = BIT(0), // collide with ground
	COL_REACHSTACKER = BIT(1), // collide with reachstacker
	COL_CONTAINER = BIT(2), // collide with container
	COL_BOX = BIT(3), // collide with box
};

// define collision types
int groundCollidesWith = COL_REACHSTACKER | COL_CONTAINER;
int reachStackerCollidesWith = COL_GROUND | COL_CONTAINER;
int containerCollidesWith = COL_GROUND | COL_REACHSTACKER | COL_CONTAINER;
int collisionBoxCollidesWith = COL_BOX;
```

Una vez definidos los tipos de máscaras, al momento de incorporar los modelos al mundo físico, se debe especificar su cuerpo rígido, su tipo de máscara y las máscaras con los que este colisiona, es decir, enviar su *btRigidBody*, su tipo a partir del enumerable *collisiontypes* (o también conocido como *filter group*) y las máscaras con los que este colisiona respectivamente.

5.3.2 Implementación de Constraint

Debido a que las diferentes partes que componen el vehículo requieren de una conexión física que permita aplicar diversas fuerzas para modificar el ángulo de estas, se hacen uso de constraint, los cuales básicamente permiten establecer una conexión entre dos cuerpos físicos manteniendo intactas sus otras propiedades, así como también su figura de colisión.

Para el vehículo Reach-Stacker se implementan constraint del tipo *slider* y del tipo *hinge*. Los constraint *slider* se caracterizan principalmente por permitir realizar una especie de deslice entre dos cuerpos, uno de los cuales se mantiene rígido y el otro se mueve a través de fuerzas. Además de la aplicación de fuerzas para su desplazamiento, se pueden definir límites tanto de ángulo como de traslación lineal.

Como se puede observar en la Figura 5.4, la sección del boom del vehículo está conectada mediante este tipo de constraint, en donde la sección azul del boom es la que se mantiene rígida, mientras que la sección en rojo se mueve a través de la aplicación de fuerzas.

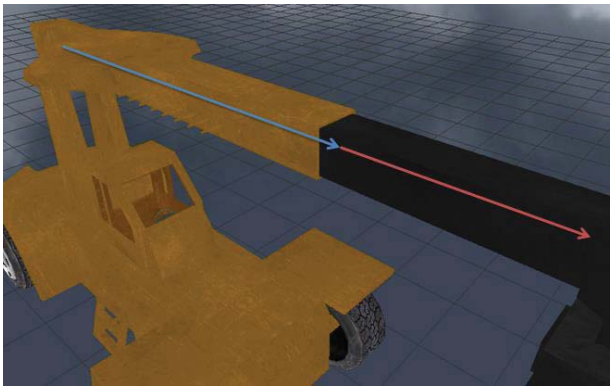


Figura 5.4 Constraint tipo slider en boom Reach-Stacker [Elab. Propia]

A continuación se presenta el algoritmo mediante el cual se genera el constraint de tipo *slider*:

```
// connect boom_1 with boom_2
localA.setIdentity();
localB.setIdentity();
localA.setRotation(btQuaternion((float)M_PI_2, 0.0f, 0.0f));
localA.setOrigin(btVector3(0.0f, 0.0f, 0.0f));
localB.setRotation(btQuaternion((float)M_PI_2, 0.0f, 0.0f));
localB.setOrigin(btVector3(0.0f, 0.0f, 0.0f));
btSliderConstraint *sliderBoom2Ct = new btSliderConstraint(*rigidBoom_1, *rigidBoom_2, localA,
localB, true);
```

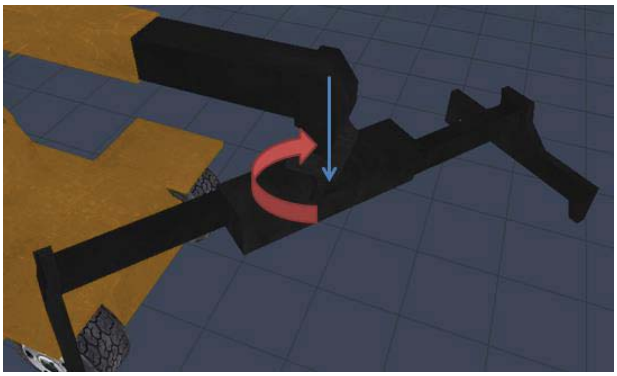


Figura 5.5 Constraint tipo hinge en spreader Reach-Stacker [Elab. Propia]

El constraint de tipo *hinge* se caracteriza por permitir realizar rotaciones a través de un eje entre dos cuerpos, en donde al igual que el constraint anterior, uno de estos se mantiene rígido. En este constraint también se pueden definir límites de rotación. En la Figura 5.5 se puede observar la conexión realizada entre dos partes del spreader del Reach-Stacker, en donde la sección en azul se mantiene rígida y la sección en rojo realiza una rotación en base a un determinado eje.

A continuación se presenta el algoritmo a partir del cual se genera el constraint *hinge*:

```
// connect spreader_1 with spreader_2
localA.setIdentity();
localB.setIdentity();
localA.setRotation(btQuaternion((float)M_PI_2, (float)M_PI_2, 0.0f));
localA.setOrigin(btVector3(0.0f, -0.5f, 0.0f));
localB.setRotation(btQuaternion((float)M_PI_2, (float)M_PI_2, 0.0f));
localB.setOrigin(btVector3(0.0f, 0.5f, 0.0f));
btHingeConstraint *hingeSpreader2Ct = new btHingeConstraint(*rigidSpreader_1, *rigidSpreader_2,
localA, localB, true);
```

El vehículo está compuesto finalmente por siete constraint, los cuales son una combinación entre constraint del tipo *slider* y constraint del tipo *hinge*.

5.3.3 Sistema de Conducción

El sistema de conducción podría pensarse inicialmente mediante la utilización de constraint del tipo *hinge* que permitan girar las ruedas delanteras del vehículo, el problema de este enfoque son las limitadas capacidades que otorga una vez que se quiera ir mejorando el realismo de este, ya sea mediante la aplicación de amortiguación en las ruedas, sistema de frenos, maniobras del vehículo, entre otras. Bullet Physics otorga un sistema de conducción mediante el algoritmo de *Ray Casting*, utilizado principalmente para la detección de superficies visibles. En este sistema de conducción otorgado por Bullet es posible establecer la cantidad de ruedas que se estime conveniente, la compresión de la suspensión, fricción de las ruedas, influencia de rotación, relajación de las ruedas, entre otros. Considerando además el tipo de conducción particular que poseen los vehículos Reach-Stacker, en donde las ruedas delanteras ejercen la fuerza de movimiento y las traseras permiten modificar la dirección del vehículo, el sistema de conducción del motor físico permite definir cuáles de las ruedas son las encargadas de cumplir cada función, estableciendo límites de velocidad, fuerzas de frenos, ángulos máximos de rotación, entre otros.

Debido a que el sistema de máscaras de colisión no aplica a las ruedas a partir del cuerpo rígido en donde estas se encuentran unidas, es necesario redefinir el método de raycasting utilizado, de lo contrario, el vehículo no colisionará con ningún cuerpo, descendiendo por el escenario infinitamente. Para que las máscaras de colisión funcionen adecuadamente en las ruedas al momento de realizar el raycasting, es necesario redefinir una nueva clase que herede de las clases predefinidas de Bullet Physics encargadas de realizar el raycasting.

Esta implementación se lleva a cabo en la clase *ReachStackerRayCaster*, la cual hereda de la clase *btVehicleRaycaster*.

5.4 Sistema de Enganche

Debido a los tipos de contenedores con los cuales puede trabajar el Reach-Stacker, se realizó la implementación de un sistema de cadenas unido a cada una de las puntas del spreader, así como también a los puntos de enganche de cada uno de los contenedores. Esto permite ajustar el spreader al tamaño correspondiente a cada contenedor y detectar cuando estos sistemas de cadenas están colisionando entre sí. En la Figura 5.6 se puede observar el sistema de cadenas asociado tanto al Reach-Stacker como a los contenedores.

Figura 5.6 Sistemas de cadenas [Elab. Propia]

Considerando que se debe realizar una detección de colisiones para cada uno de los puntos de colisión, se ha incorporado la clase *MyContactResultCallback*, la cual redefine el método *addSingleResult* de una clase de detección de colisiones base del motor físico. Esta redefinición permite especificar mediante un valor booleano si dos cuerpos se encuentran en estado de colisión. Una vez realizado esto, se puede comprobar de manera independiente la colisión entre cada una de las cadenas, ya sean del vehículo Reach-Stacker como de los contenedores. En el caso que exista una colisión de cada uno de estos puntos, el usuario es capaz de realizar el enganche del contenedor al spreader, lo que internamente realiza una asociación del contenedor al vehículo mediante constraints. La liberación del contenedor se realiza mediante la eliminación del constraint asociado a dicha conexión.

A continuación se presenta el algoritmo para la creación de las cadenas de colisión para cuatro contenedores:

```
Container *newContainer = new Container(cargo);

for(int j = 0; j < 4; j++){
       Mesh *collisionBoxCargo = temp_collisionBox->clone();
       collisionBoxCargo->init();
       scene->addChild(collisionBoxCargo);
       physicsManager->addRigidBody(collisionBoxCargo, 0.001f, glm::vec3(0), 1.0f,
                                    glm::vec3(0), glm::vec3(0.0f), shapeCollisionBox,
                                    COL_BOX, collisionBoxCollidesWith);

       newContainer->addCollisionBox(true, collisionBoxCargo->getRigidBody(), j,
                                     physicsManager);
}

containers.push_back(newContainer);
```

Debido a que los contenedores son variables tanto en cantidad como en tipo, se almacena un arreglo de contenedores en donde cada uno de estos guarda la información de los cuerpos rígidos del contenedor y de sus cuatro cadenas. Esto permite por una parte realizar la detección de la colisión entre las cadenas del Reach-Stacker con cada una de las cadenas de los contenedores, y en el caso que se encuentren colisionando los cuatro puntos, construir el constraint entre el cuerpo rígido del spreader y el cuerpo rígido del contenedor.

5.5 Integración de Sonidos

La integración de sonidos a la aplicación brinda una mayor experiencia de usuario al momento de hacer uso de esta, permitiendo además comprender con mayor facilidad acciones que se están llevando a cabo de manera simultánea. La librería SDL_mixer permite cargar en memoria diversos sonidos, los cuales pueden ser reproducidos en diversos canales configurables en tiempo de ejecución. Además de la carga y reproducción de estos sonidos, es necesario ajustar el periodo de reproducción de estos según las condiciones que se estén llevando a cabo en cada momento, es decir, poder especificar si un determinado sonido se debe reproducir infinitamente hasta un determinado momento, o si este se reproducirá una sola vez.

Alguna de las condiciones que permiten determinar el tiempo de duración de un sonido son los límites de rotación o traslación de los constraint, si el vehículo se encuentra en movimiento, si se encuentra en reversa, si acaba de enganchar un contenedor, entre otros.

Considerando lo anterior, se ha implementado la clase *Sound*, la cual almacena la información de una pista de audio y el canal en cual va a ser reproducida, permitiendo cargar el sonido, reproducirlo, detenerlo y eliminarlo de memoria una vez finalizada la aplicación. Considerando que cada sonido debe ser almacenado en un canal diferente, se ha implementado la clase *SoundLibrary*, la cual gestiona el conjunto de sonidos totales cargados en la aplicación, ajustando dinámicamente la cantidad de canales necesarios para la reproducción de cada sonido de manera

independiente. Una vez finalizada la aplicación, esta clase también permite realizar la liberación de memoria de todos los sonidos cargados.

A continuación se presenta el método estático de la clase *SoundLibrary* para la carga de un sonido en memoria y posteriormente un ejemplo de su uso:

```
Sound* SoundLibrary::addSound(const char *name, const char *soundFile){
       Mix_AllocateChannels(lastChannel + 1);

       Sound *newSound = new Sound(name);
       newSound->load(soundFile, lastChannel++);
       soundHash[name] = newSound;

       return newSound;
}

// the sound effects that will be used
Sound *snd_truck_idle = SoundLibrary::addSound("truck_idle", "media/truck_idle.wav");
```

5.6 Modificaciones Anexas

Para mejorar la visualización del entorno tridimensional, se ha incorporado la posibilidad de ver el mundo en modo *wireframe*, es decir, ver únicamente las líneas que componen a cada uno de los modelos que se encuentran en el escenario tridimensional. En la Figura 5.7 se puede observar el modo wireframe activado, permitiendo ver a través de los objetos fácilmente.

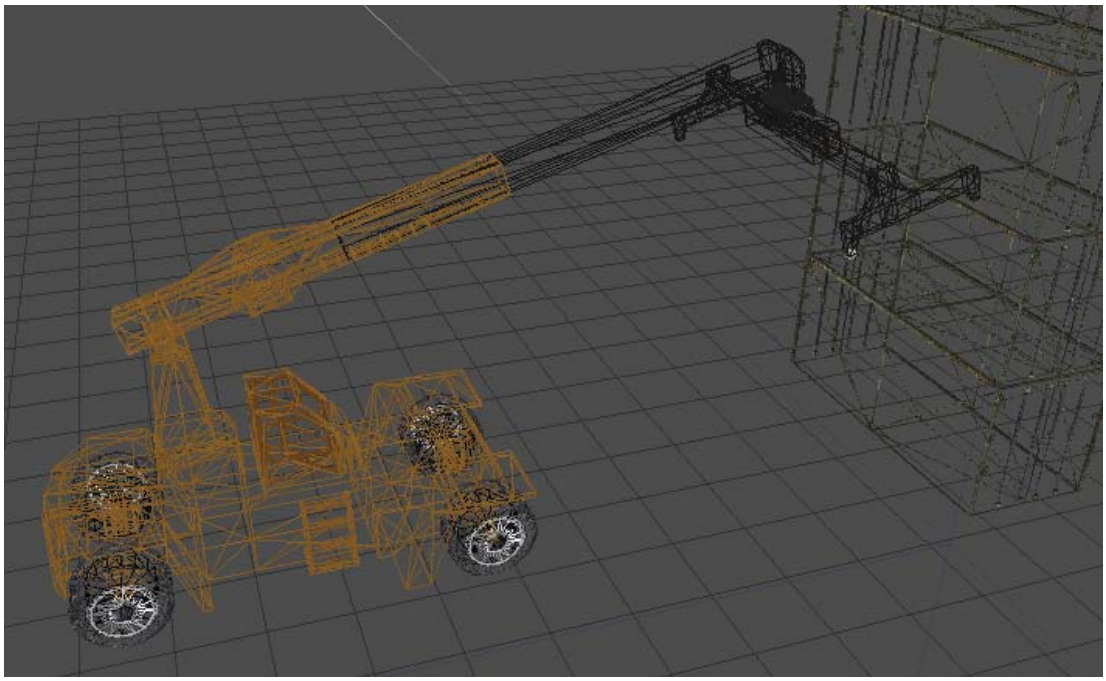


Figura 5.7 Simulador modo wireframe [Elab. Propia]

Capítulo 6: Conclusiones

A partir del desarrollo del presente proyecto se han comprendido las diferentes variables que están involucradas en el desarrollo de una aplicación gráfica, además de la importancia del uso de una metodología de trabajo que permita ir concretando hitos de manera rápida, considerando a su vez un diseño simple y eficaz. Debido a la compleja curva de aprendizaje, es necesario además ir adaptándose a los diversos cambios que se presentan a lo largo del proyecto, los cuales se deben principalmente a la incorporación de nuevas librerías al desarrollo del proyecto y la aplicación de algoritmos que vayan mejorando el funcionamiento base del motor.

El conjunto de librerías utilizadas se ha seleccionado en base a un análisis de compatibilidad entre estas, basándose en la simplicidad y potencialidad que ofrecen, considerando además que estas sean respaldadas por una amplia documentación y una comunidad activa, permitiendo de este modo agilizar el proceso de aprendizaje e ir adaptándose a los diversos cambios que el motor va sufriendo a lo largo del tiempo. Debido a la naturaleza del proyecto, se han escogido además librerías multiplataforma, permitiendo de este modo portar el proyecto fácilmente a diversas plataformas. A pesar que muchas de los métodos y funciones otorgados por el conjunto de APIs simplifican enormemente el trabajo a desarrollar, la integración de las diferentes partes que componen la arquitectura del motor gráfico no es un proceso trivial, debido a la amplia variedad de formas en que estas pueden ser programadas, dependiendo principalmente del nivel de abstracción que se otorgará finalmente al desarrollador del simulador.

Si bien existe una amplia variedad de motores gráficos, actualmente es muy difícil encontrar información relacionada con la creación de uno, a pesar que muchas de las técnicas utilizadas son ampliamente reconocidas, su integración muchas veces varía dependiendo de las necesidades específicas del desarrollador, teniendo que implementar en muchas ocasiones algoritmos que requieren de amplios conocimientos matemáticos orientados a la computación gráfica. En el caso de los motores gráficos utilizados como puntos de referencia, muchas de las técnicas que estos presentan se encuentran implementadas en diversas APIs, ya sean manipuladores de ventanas, APIs gráficas como OpenGL o Direct3D, diversas librerías físicas, librerías para la gestión de imágenes, entre otras; dificultando el proceso de análisis y abstracción de su implementación.

La implementación de un motor físico ha permitido abstraerse de complejos cálculos matemáticos involucrados en la mayor parte de las acciones que se llevan a cabo en la simulación, permitiendo generar una aproximación bastante cercana a la realidad. El uso de constraint del mismo modo ha permitido definir las conexiones fundamentales que poseen las partes que componen el vehículo Reach-Stacker, sin dejar de lado sus propiedades físicas, tales como la fricción, inercia, masa, entre otros. La generación de cuerpos de colisiones aproximados a partir de las geometrías originales de los modelos 3D, también ha significado una mejora sustancial en la simulación, estableciendo colisiones mucho más realistas y que finalmente mejoran la experiencia del usuario al momento de hacer uso del simulador.

La idea principal del presente proyecto es otorgar un sistema de renderización sencillo, que permita incorporar diversos modelos tridimensionales a la escena sin mayor dificultad, permitiendo a su vez otorgarle a cada uno de estos modelos características físicas que simulen de manera más realista su interacción entre sí. Debido a la estructuración del motor gráfico, el desarrollo de aplicaciones que pueden crearse a partir de este no está limitado únicamente al ámbito portuario, sino que a cualquier tipo de simulación que se tenga pensado.

Respecto al trabajo futuro, podrían realizarse mejoras en el apartado gráfico del motor, tales como la incorporación de un sistema de sombras dinámicas mediante mapas de sombras, sistema de luces, incorporación de un sistema de GUI compatible con shaders haciendo uso de la matriz de proyección ortogonal, integración de reflejos, nuevos mapas de texturas, etc. Así como también mejoras en el sistema de renderizado, tales como el sistema de frustrum culling utilizado para la ocultación de los modelos que no se encuentran en el espectro de visión de la cámara, el almacenamiento de los modelos mediante árboles de partición binaria del espacio, sistema de depuración gráfica mediante la utilización de geometry shaders, entre otros.

Considerando que el mundo tridimensional podría ser dinámico, la ubicación y tipo de los diferentes contenedores podría estar dado a partir de un editor gráfico de escenarios compatible con el simulador, simplificando considerablemente la generación de entornos dinámicos que se adapten a las necesidades de cada usuario. Debido a que los métodos para la carga de modelos tridimensionales y generación de los cuerpos dinámicos de colisión correspondientes a estos son genéricos, podrían integrarse modelos tridimensionales que se ajusten al tipo de entorno que se desea simular, ya sea un patio de contenedores, barcos, trenes, grúas de carga, entre otros.

Referencias

[Shalini, 2004] Shalini Govil-Pai, *"Principles of Computers Graphics"*, ISBN: 0-387-95504-6, 2004, pp 110.

[Prabhudas, 2008] Siddharth Prabhudas, *"Agile Unified Process"*, UnitedHealth Group Information Services, Haryana, Pure Conference Solution Pvt Ltd., 2008, pp 2-3.

[Canós *et al*., 2003] José H. Canós, Patricio Letelier, Mª Carmen, *"Metodologías Ágiles en el Desarrollo de Software"*, Universidad Politécnica de Valencia, Taller VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Grupo ISSI, Alicante, 2003, pp 59.

[Gregory, 2009] Jason Gregory, *"Game Engine Architecture"*, ISBN: 978-1-4398-6526-2, 2009, pp 28-49.

[Reddy, 2011] Martin Reddy, *"API design for C++"*, Morgan Kaufmann Publishers, ISBN 978-0123850034, 2011, pp 10-50.

[Hearn y Baker, 2005] Donald Hearn, M. Pauline Baker, "*Gráficos por computadora con OpenGL"*, Pearson Prentice Hall, ISBN 978-84-205-3980-5, 2005, pp 81-82.

[Shreiner, 2010] Dave Shreiner, "*OpenGL Programming Guide Seventh Edition"*, Addison-Wesley Professional, ISBN 978-0-321-55262-4, 2010, pp 10-14.

[Smed, 2008] Jouni Smed, *"Networking for Computer Games"*, International Journal of Computer Games Technology, Volume 2008, Article ID 928712, 1 page, 2008.

[Maroto, 2005] Joaquín Maroto Ibáñez, *"Metodología para la generación de entornos virtuales distribuidos y su aplicación a simuladores de conducción"*, Universidad Politécnica de Madrid, Tesis Doctoral en Ingeniería Mecánica, 2005.

[Wright y Lipchak, 2004] Richard S. Wright, Jr., Benjamin Lipchak, *"OpenGL SuperBible: Comprehensive Tutorial and Reference"*, Addison-Wesley Educational Publishers Inc., ISBN 978-0321712615, 2004, pp 5-30.

[Wright *et al*., 2007] Richard S. Wright, Jr., Benjamin Lipchak, Nicholas Haemel, "*OpenGL SuperBible 4th Edition"*, Computer Graphics, ISBN 0-321-49882-8, 2007, pp 1-23.

[Wright *et al.*, 2014] Richard S. Wright, Jr., Nicholas Haemel, *"OpenGL SuperBible Comprehensive Tutorial and Reference Sixth Edition"*, Graham Sellers Pearson Education, ISBN: 978-0-321-90294-7, pp 1-150, 2014.

[Thomson, 2006] Richard Thomson, *"The Direct3D Graphics Pipeline"*, Guide to Learn Direct3D, 2006, pp 43-58.

[Engel y Geva, 2000] Wolfgang Engel, Amir Geva, *"Beginning Direct3D Game Programming"*, Argosy, ISBN 0-7615-3191-2, 2000, pp 3-6.

[Lengyel, 2012] Eric Lengyel, *"Mathematics for 3D Game Programming and Computer Graphics"*, 2012, pp 1-30.

[Gómez *et al*., 1999] Pedro Gómez, Marco Gómez, Enrique Alonzo, *"Funcionamiento de los Árboles BSP"*, Eresmas, 1999, pp 5.

[Vallejo y Martín, 2013] David Vallejo Fernández, Cleto Martín Angelina, *"Desarrollo de Videojuegos, Arquitectura del Motor de Videojuegos 2°da Edición"*, ISBN: 978-84-686-4028-0, 2013, pp.117-159.

[Keller, 2011] Erick Keller, "*Introducing Zbrush 4"*, SYBEX, ISBN 04705276412011, pp 15-30.

[Valient, 2001] Michal Valient, *"3D Engines in games - Introduction"*, HTML Article and Guide, 2001, pp 1-6.

[Glassner, 1989] Andrew Glassner, *"An Introduction to Ray Tracing"*, San Francisco, Morgan Kaufmann Publishers, ISBN 978-0122861604, 1989, pp 4-10.

[Christiansen *et al.*, 2008] Allen Christiansen, Damian Johnson, and Lawrence Holder, *"Game-Based Simulation for the Evaluation of Threat Detection in a Seaport Environment"*, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, ICEC 2008, LNCS 5309, pp. 221–224, 2008.

[Hess, 2010] Roland Hess, "*Blender Foundations: The Essential Guide to Learning Blender 2.6"*, Focal Press, ISBN 9780240814308, 2010, pp 1-15.

[Johnson, 2009] Damian Johnson, *"Port Locale Modeling and Scenario Evaluation in 3D Virtual Environments"*, Master of Science in Computer Science, Washington State University, 2009.

[URL 1, 1969] ACM SIGSIM, Special Interest Group (SIG) on Simulation and Modeling (SIM), *http://www.acm-sigsim-mskr.org/*, última vez visitada: 09/06/2016, 1969.

[URL 2, 2014] TCM Corporation, *http://tcmforktrucks.net/index.php*, última vez visitada: 09/06/2016, 2014.

[URL 3, 2005] Agile Unified Process, *http://www.cc.una.ac.cr/AUP/index.html*, última vez visitada: 09/06/2016, 2005.

[URL 4, 2011] Árboles BSP, *http://blogs.aerys.in/jeanmarc-leroux/tag/quake-3/*, última vez visitada: 09/06/2016, 2011.

[URL 5, 2009] Classical LOD Algorithms, *http://habib.wikidot.com/techniques*, última vez visitada: 09/06/2016, 2009.

[URL 6, 2014] Bullet Physics, *http://bulletphysics.org/*, última vez visitada: 09/06/2016, 2014.

[URL 7, 2014] Game Programming Patters, *http://gameprogrammingpatterns.com/flyweight.html*, última vez visitada: 09/06/2016, 2014.

[URL 8, 2006] Box Modeling, *http://wiki.cgsociety.org/index.php/Modeling_a_foot*, última vez visitada: 09/06/2016, 2006.

[URL 9, 2004] NURB Modeling, *http://www.3dmax-tutorials.com/_2_Rail_Sweep_Surface.html*, última vez visitada: 09/06/2016, 2004.

[URL 10, 2011] Perfiles de Splines, *http://www.etereaestudios.com/training_img/subd_tips/introduccion.htm*, última vez visitada: 09/06/2016, 2011.

[URL 11, 2011] Sculpting, *http://estab1986.com/blog/tag/hard-surface-sculpting*, última vez visitada: 09/06/2016, 2011.

[URL 12, 2008] Texturizado, *http://ptex.us/ptexpaper.html*, última vez visitada: 09/06/2016, 2008.

[URL 13, 2009] Texturas Procedurales, *http://www.oocities.org/valcoey/texturas_procedurales.html*, última vez visitada: 09/06/2016, 2009.

[URL 14, 2012] Vertex Paint, *http://wiki.blender.org/index.php/Doc:2.4/Manual/Materials/Vertex_Paint*, última vez visitada: 09/06/2016, 2012.

[URL 15, 2009] Vertex Paint, *http://wiki.blender.org/index.php/Doc:2.6/Manual/Materials/Special_Effects/Vertex_Paint*, última vez visitada: 09/06/2016, 2009.

[URL 16, 2006] UV Map, *http://www.members.shaw.ca/nickyart/Comp/Camel.htm*, última vez visitada: 09/06/2016, 2006.

[URL 17, 2014] 3D Studio Max, *http://usa.autodesk.com/3ds-max/*, última vez visitada: 09/06/2016, 2014.

[URL 18, 2014] Maya, *http://usa.autodesk.com/maya/*, última vez visitada: 09/06/2016, 2014.

[URL 19, 2014] Google SketchUp, *http://sketchup.google.com/intl/es/*, última vez visitada: 09/06/2016, 2014.

[URL 20, 2014] Cinema 4D, *http://www.maxon.net/es/products/cinema-4d-studio.html*, última vez visitada: 09/06/2016, 2014.

[URL 21, 2014] Zbrush, *http://www.pixologic.com/home.php*, última vez visitada: 09/06/2016, 2014.

[URL 22, 2014] Blender, *http://www.blender.org/*, última vez visitada: 09/06/2016.

[URL 23, 2014] Simple DirectMedia Library, *http://www.libsdl.org/*, última vez visitada: 09/06/2016, 2014.

[URL 24, 2014] FreeImage, *http://freeimage.sourceforge.net/*, última vez visitada: 09/06/2016, 2014.

[URL 25, 2014] SDL_ttf, *https://www.libsdl.org/projects/SDL_ttf/*, última vez visitada: 09/06/2016, 2014.

[URL 26, 2014] SDL_mixer, *http://www.libsdl.org/projects/SDL_mixer/*, última vez visitada: 09/06/2016, 2014.

[URL 27, 2014] OpenGL Mathematics Library, *http://glm.g-truc.net/0.9.5/index.html*, última vez visitada: 09/06/2016, 2014.

[URL 28, 2014] GNSS Software Simulation, *http://www.ifen.unibw-muenchen.de/research/gnss_simulator.htm*, última vez visitada: 09/06/2016, 2014.

[URL 29, 2014] Irrlicht Engine, *http://irrlicht.sourceforge.net/*, última vez visitada: 09/06/2016, 2014.

[URL 30, 2014] Crystal Space 3D, *http://www.crystalspace3d.org/main/Main_Page*, última vez visitada: 09/06/2016, 2014.

[URL 31, 2014] Ogre3D, *http://www.ogre3d.org/*, última vez visitada: 09/06/2016, 2014.