Universidad de las Ciencias Informáticas

Facultad 6

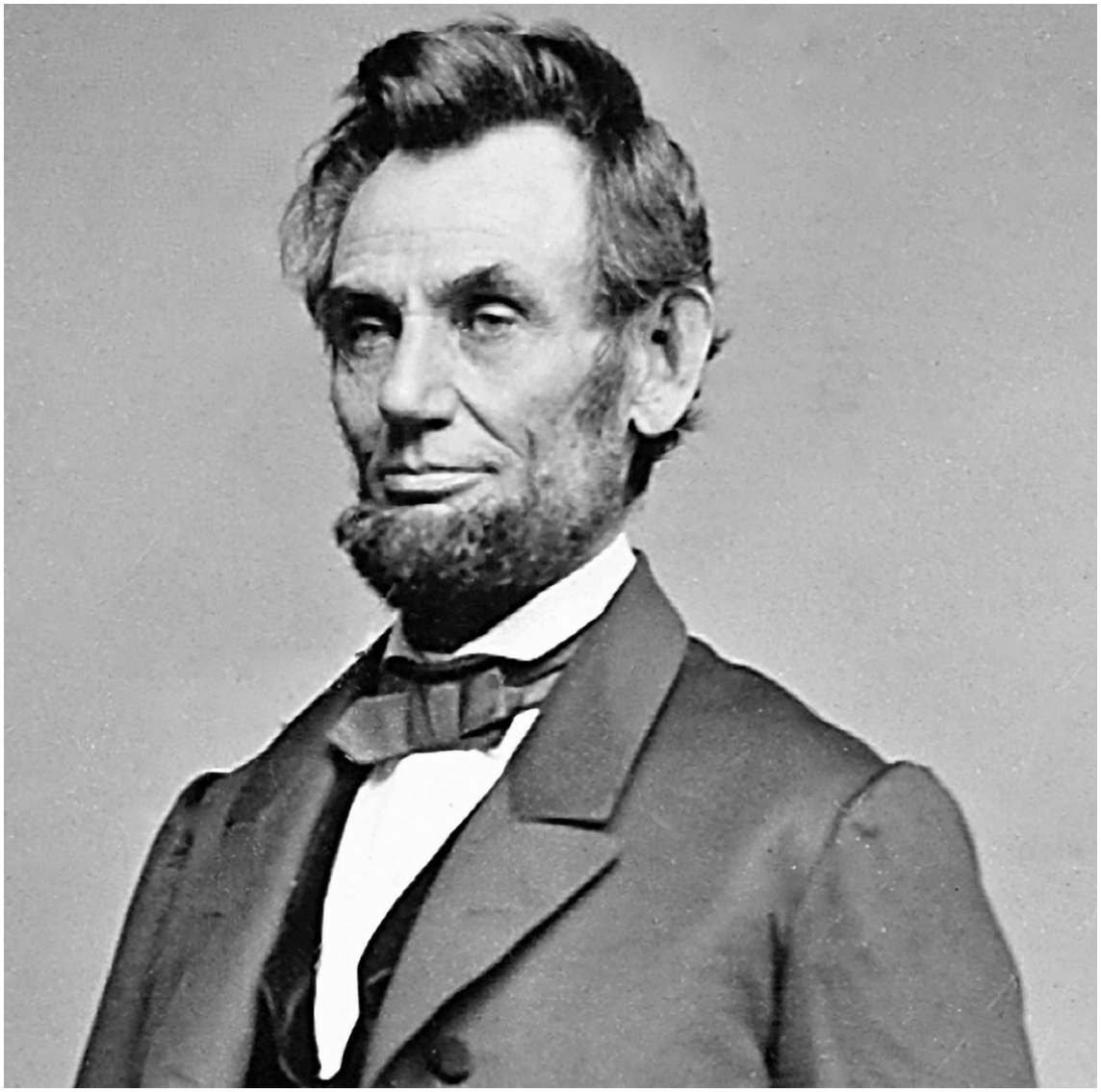
Título: Entorno virtual 3D para la visualización de simulaciones de eventos discretos.

Autor: Yosbel Lázaro Guirola Manresa

Tutor: Ing. Yadian Fernández Pérez

MSc. Gilberto Arias Naranjo

La Habana, 05/07/2015



El adquirir conocimientos es la mejor inversión que se puede hacer.

Abraham Lincoln.

Declaración de Autoría

Declaro ser autor del presente trabajo de diploma y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste, firmo la presente a los ___ días del mes de _____ del año ____.

Yosbel Lázaro Guirola Manresa

_____

Firma del autor

Ing. Yadian Fernández Pérez

_____

Firma del tutor

MSc. Gilberto Arias Naranjo

_____

Firma del tutor

Datos de Contacto

Autor:

Yosbel Lázaro Guirola Manresa

Universidad de las Ciencias Informáticas, La Habana, Cuba.

Correo electrónico: ylguirola@estudiantes.uci.cu

Tutores:

Ing. Yadian Fernández Pérez

Universidad de las Ciencias Informáticas, La Habana, Cuba.

Correo electrónico: yfernandezp@uci.cu

MSc. Gilberto Arias Naranjo

Universidad de las Ciencias Informáticas, La Habana, Cuba.

Correo electrónico: gilbertoa@uci.cu

Dedicatoria:

Dedico esta tesis a mis padres Julia y Felicio, a mi hermana Lidice, a mi novia Diana y al resto de mi familia.

Agradecimientos:

Quisiera agradecer a mi madre, por su apoyo, sacrificio incondicional, consejos y motivarme en el momento necesario a emprender los estudios en la Universidad.

A mi padre por su apoyo, sacrificio y consejos.

A mi hermana Lidice.

A mi novia Diana por su compañía, amor y revisiones al documento.

A mis tutores y a Castrillón por dedicarme parte de su tiempo y sus provechosas revisiones.

A los compañeros de apartamento durante la carrera entre los que se encuentran Yamil, Victor, Cuso, Blanco, Norge, Ginori y Asiel.

A todas mis amistades.

A mis profesores de la carrera.

A todos los profesores que me apoyaron durante el desarrollo de la tesis.

Al movimiento de programación competitiva “Tomás López Jiménez” de la UCI. En especial a las siguientes personas Rubén, Antonio, Roniel, Dovier, Rancel, Guillermo, Oreste y Reinier.

A los compañeros del Centro de Estudios de Matemática Computacional.

Resumen

La visualización científica en tres dimensiones ha sido de gran utilidad en el campo de la simulación, permitiendo alcanzar un mejor entendimiento de sistemas modelados de alta complejidad. En el proyecto Sistema de Gestión de Contenedores Robotizado se crean simulaciones de eventos discretos que pueden contener millones de objetos con posiciones y formas cambiantes en el tiempo, lo que dificulta la visualización fluida de las mismas. En el presente trabajo se desarrolló una aplicación informática con el objetivo de visualizar de manera eficiente dichas simulaciones. Se diseñó un proceso para visualizar las simulaciones creadas en el proyecto, el cual fue implementado utilizando el motor de juego Unity3D en conjunto con el lenguaje de programación C#. Se utilizaron además técnicas de optimización para aumentar la velocidad de visualización tales como el *frustum culling*, los niveles de detalle, el *batching* y la niebla. También se utilizaron técnicas para mejorar la calidad visual de la representación de las escenas como es el caso de la iluminación, el texturizado y el sombreado. Los resultados obtenidos en los experimentos permitieron validar la eficiencia de la herramienta implementada. La misma es extensible a simulaciones de eventos discretos llevadas a cabo en otros contextos como es el caso de fábricas, control de tráficos y en proyectos de construcción.

Palabras Claves: motor de juegos, motor gráfico, simulación, Unity3D, visualización 3D.

Abstract

The 3D scientific visualization has been useful in the simulation field, making possible a better understanding of high complexity modelled systems. In the project Robotic Container Management System discrete event simulations are created which might contain millions of objects with changing positions and shapes through time, what makes it difficult to obtain a fluid visualization of them. In this work a software application was developed with the goal of visualizing efficiently these simulations. A process for the visualization of the created simulations was designed and implemented using the game engine Unity3D in addition to the C# programming language. Optimization techniques like frustum culling, level of detail, batching and fog were also employed to increase the visualization speed. Techniques to improve the quality of the representation of the scenes like lighting, texturing and shading were also used. The results obtained in the experiments allowed the validation of the efficiency of the implemented tool. This tool is extensible to discrete event simulations carried out in other contexts like factories, traffic control and construction operations.

Keywords: *game engine, graphic engine, simulation, Unity3D, visualization 3D.*

Índice

Introducción

El campo de la visualización científica ha experimentado crecimiento sostenido en las últimas décadas debido a que cada día más científicos necesitan una mejor comprensión de sus investigaciones a partir de la representación de los fenómenos y procesos estudiados. La visualización en tres dimensiones (3D) permite que una mayor cantidad de información sea interpretada en un período de tiempo más corto. Volúmenes de datos que habrían tardado años para analizarse ahora se pueden interpretar de forma gráfica en cuestión de segundos (Rohrer, 2000).

La simulación es la imitación de las operaciones de los procesos del mundo real o de un sistema en el tiempo. Esta puede implicar la generación de un historial artificial del sistema, y la observación de dicho historial para sacar conclusiones relativas a las características de funcionamiento del sistema real que se representa (Wenzel, Bernhard y Jessen 2003). Aunque los experimentos de simulación pueden producir una gran cantidad de datos, la visualización en tres dimensiones de un sistema simulado proporciona una comprensión más completa de su comportamiento (Bijl and Boer, 2012).

En mayo de 2015 un conjunto de instituciones de la Unión Europea (UE) creó el proyecto Sistema de Gestión de Contenedores Robotizado (RCMS que refiere a sus siglas en inglés). Entre las líneas de trabajo llevadas en RCMS se encuentra crear modelos de simulación de eventos discretos de las Terminales de Contenedores de Gdansk y Koper. La ejecución de los modelos de dichas terminales genera ficheros de historial de gran cantidad de información, que son difíciles de entender e interpretar por los usuarios de estos sistemas.

Los ficheros de historial generados pueden llegar a contener millones de objetos, cada uno con millones de descripciones de movimiento en el tiempo. Esto último se debe a que una simulación de RCMS puede durar varios días. Cada descripción de movimiento representa el estado de los objetos solamente en dos instantes discretos de tiempo (instante inicial e instante final), el estado de un objeto en un instante intermedio de tiempo se aproxima mediante una función a partir del estado inicial y el estado final. La elevada cantidad de objetos en conjunto con los cálculos de los estados puede comprometer la visualización fluida de las simulaciones. Actualmente no se cuenta con un mecanismo que permita visualizar en tres dimensiones de forma eficiente las simulaciones de eventos discretos generadas en RCMS. Dígase eficiente que la

cantidad de fotogramas por segundo visualizados sea mayor que 30, mínimo requerido según (Microsoft, 2003) para lograr una visualización fluida.

A partir de lo descrito anteriormente se plantea como **problema de investigación**:

¿Cómo visualizar eficientemente en tres dimensiones simulaciones de eventos discretos?

Dado el problema anterior se define como **objeto de estudio:**

Aplicaciones informáticas para la visualización de simulaciones.

Estableciendo como **campo de acción**:

Aplicaciones informáticas para la visualización en tres dimensiones de simulaciones de eventos discretos.

Objetivo General: Desarrollar una aplicación informática que permita la visualización en tres dimensiones de las simulaciones de eventos discretos creadas para RCMS.

A partir del marco teórico desarrollado se formula la siguiente **hipótesis de investigación**:

Aplicando las técnicas y algoritmos de optimización se logra una visualización eficiente de simulaciones de eventos discretos.

Se plantean un grupo de **tareas de investigación** para satisfacer el objetivo general.

- Análisis de la literatura acerca de la visualización en 3D de simulaciones de eventos discretos.
- Selección de los principales algoritmos y técnicas de optimización para la visualización eficiente.
- Selección de las herramientas y tecnologías a utilizar para el desarrollo de la aplicación.
- Definición de los estándares de codificación para implementar una aplicación más fácil de mantener.
- Implementación de los algoritmos seleccionados para visualizar la información.
- Diseño de los casos de prueba para validar el correcto funcionamiento de la aplicación.
- Validación de la solución.

Métodos de investigación:

- Método analítico-sintético: Para estudiar por separado las diferentes técnicas y procesos utilizados en la visualización en 3D de simulaciones de eventos discretos, y luego sintetizarlos en la elaboración de las funcionalidades.

- Análisis documental: Para la revisión de la literatura, que permitió consultar la información necesaria en el proceso de investigación sobre la visualización en 3D de simulaciones de eventos discretos.

- Método de modelado: Con vista a representar los diagramas correspondientes a las etapas de análisis, diseño e implementación de la solución.

- Método hipotético-deductivo: Para la formulación de la hipótesis y arribar a conclusiones.

Posibles resultados: Obtención de una aplicación informática que permita visualizar en 3D de manera eficiente las simulaciones de eventos discretos creadas por el proyecto RCMS.

Capítulo 1 Fundamentación Teórica

En el presente capítulo se presentan los conceptos asociados a las simulaciones y a los motores de videojuego. Se hace una revisión de las soluciones existentes para la visualización de simulaciones. Se realiza un análisis para la selección de las técnicas de optimización, herramientas y tecnologías a utilizar para el desarrollo del trabajo.

1.1 Conceptos asociados al dominio del problema

1.1.1 Simulación

La simulación es el proceso de diseñar un modelo de un sistema concreto y la realización de experimentos con él, con la finalidad de entender el comportamiento de un sistema concreto y/o evaluar varias estrategias para la explotación del sistema (Shannon, 1975).

Un **modelo de simulación de eventos discretos** se define como uno en el que las variables de estado cambian solamente en aquellos puntos discretos en el tiempo en que ocurren los eventos. Los eventos ocurren como consecuencia de los tiempos de actividad y retrasos. Las entidades pueden competir por los recursos del sistema, eventualmente pueden unirse a las colas mientras esperan por la disponibilidad de un recurso. Las actividades y tiempos de retardo pueden sostener entidades por periodos de tiempo. Un modelo de simulación de eventos discretos se lleva a cabo a través del tiempo (corrida) por un mecanismo que mueve el tiempo simulado hacia adelante. El estado del sistema se actualiza en cada evento junto con la captura y la liberación de los recursos que puede producirse en ese momento (Banks, 1999).

1.1.2 Formatos de salida de la simulación

Es el equipo de desarrollo de software de simulación quien decide cómo escribir los ficheros de historial. Estos formatos dependen del tipo de simulación que se realice y de la información que se requiera para describir los estados del sistema simulado. Ejemplos de formatos utilizados en la salida de simulaciones son los formatos VRML[1], XYZ (Open Babel, 2007). La mayoría de las aplicaciones de simulación definen su propio formato, por lo que es difícil compartir estos ficheros entre aplicaciones. El proyecto RCMS usa un formato basado en XML[2] para almacenar estos ficheros, dicho formato permite

[1] https://www.w3.org/MarkUp/VRML/
[2] https://www.w3.org/XML/

describir escenarios complejos con objetos que cambian de posición y forma en el tiempo.

El fichero de historial tiene los datos asociados a los objetos presentes en la simulación tales como nombre, tipo y descripción de sus movimientos. Estos últimos describen los cambios de posición y forma de los objetos durante el tiempo simulado y tienen datos como tipo, estado inicial, estado final, tiempo inicial, velocidad y duración.

1.1.3 Rendering 3D

El término *rendering* normalmente se utiliza en la literatura para designar el proceso de creación de gráficos en entornos tridimensionales en computadoras (Valle, 2009). Este es un proceso creativo similar a la fotografía o cinematografía debido a que se crea una imagen a partir de escenas. La principal diferencia del *rendering* con respecto a la fotografía regular es que se representan escenas imaginarias y no reales, esto se debe a que todos los objetos representados mediante esta técnica tienen que ser creados previamente en el ordenador para su posterior visualización (Birn, 2002).

1.1.4 Motor de videojuegos o motor de juego

Desde la década de los 90, se comenzaron a utilizar bibliotecas gráficas que simplificaban la tarea de trabajar con entidades tridimensionales, ofreciendo al usuario de una interfaz estable con la que implementar funciones gráficas de manera más sencilla. Dichas bibliotecas convergen básicamente en dos tipos de referencia OpenGL (código abierto) y DirectX (propietaria de Microsoft) (Alonso Rodríguez, 2012).

En conjunto a la aparición de dichas bibliotecas, surgen los motores de juego. Formalmente, un motor de juegos podría definirse como una solución de software compuesta al menos de un motor gráfico y una serie de bibliotecas, ideado para la realización de aplicaciones 3D en las que la interacción con el usuario sea parte central de las mismas. Las bibliotecas incluidas en un motor de juegos pueden ser un motor de física, que se encargue de otorgar comportamiento realista a los objetos de la escena, biblioteca de audio, de inteligencia artificial, entre otros (Alonso Rodríguez, 2012).

Conviene hacer una distinción entre motor de gráficos y motor de juegos. Un motor de gráficos es parte de un motor de juegos y únicamente se encarga de otorgar al usuario los medios necesarios para el manejo de gráficos tanto bidimensionales como 3D, mientras que un motor de juegos, como se ha dicho anteriormente, integra una serie de bibliotecas que extienden su funcionalidad hasta abarcar el desarrollo completo de un

videojuego. Dentro de un motor de juegos, lo más visible es el motor gráfico, ya que es el encargado de la parte visual, y por ello se tiende a asociar ambos conceptos en uno solo (Alonso Rodríguez, 2012).

1.2 Soluciones informáticas existentes

Existen sistemas que permiten la visualización de simulaciones en 3D con la finalidad de validar, de realizar análisis y marketing. Entre estos se encuentra AnyLogic, Arena, Flexsim y otros con similares características.

AnyLogic

AnyLogic es una herramienta de simulación basada en Java que soporta varios formalismos de simulación. Las simulaciones se crean utilizando máquinas de estado, diagramas de acción y eventos. Para cada componente de la simulación se puede seleccionar un objeto 3D para representarlo. Se proporciona una biblioteca con objetos 3D estándares, pero también es posible importarlos de otras fuentes. También es posible añadir otros caminos y formas primitivas en la animación 2D y 3D. Con AnyLogic, los gráficos que definen la lógica de la simulación se separan de la especificación del esquema de la visualización. De esta manera los gráficos para la lógica se pueden estructurar de una manera clara y ordenada, y al mismo tiempo de la animación se pueden colocar desordenados u organizados como es en la realidad. Aparte de la animación 2D o 3D, el usuario también puede diseñar el esquema de cómo la simulación debería ser mostrada. También puede agregar gráficos estadísticos y ajustes personalizables a este diseño. De esta forma el usuario puede asegurarse de que, durante la ejecución de la simulación, tiene una buena visión general de los avances y resultados de la simulación. AnyLogic es privativo y cuenta con tres licencias Profesional, Investigaciones Universitarias y Edición Libre para el Aprendizaje Personal (AnyLogic Company, 2015).

Arena

Arena es un paquete en el que las animaciones 3D se usan principalmente para fines de presentación. Cuando se construye o corre la simulación no hay animación 3D, el usuario tiene que hacerlo con una simple animación 2D. Para la animación 3D hay un programa separado de reproducción en 3D, que viene con la edición empresarial de Arena. Para producir la animación 3D con Arena, hay varios pasos a seguir. En primer lugar, un modelo de simulación se debe crear y ejecutar. Luego, el usuario puede configurar un entorno 3D en el reproductor 3D, y configurar cómo los diferentes aspectos

de la simulación deberían ser mostrados en 3D. Por último, el usuario puede ver la animación 3D, y guardarlo como una película. El software es privativo y cuenta con dos ediciones una para la educación y una comercial. La edición para la educación tiene tres licencias Versión para Estudiantes (libre), Laboratorios Académicos e Investigaciones Académicas. Por su parte la edición comercial tiene dos versiones la Estándar y la Profesional (Rockwell Automation, 2015).

Flexsim

Flexsim ha integrado la animación 3D en el núcleo de su sistema. Tiene un entorno 3D, que se utiliza para construir la simulación, diseñar la animación y visualizar la corrida de la simulación. Los nodos de la simulación se representan por modelos 3D, que pueden ser colocados en el entorno 3D. Entonces pueden ser conectados a los otros nodos, y su apariencia se puede modificar si es necesario. El modelo 3D que se utiliza para una entidad o un nodo se puede seleccionar de la biblioteca “Modelo 3D” que se suministra por Flexsim, pero modelos 3D personalizados también se pueden importar. Cuando se inicia la simulación, este entorno 3D viene a la vida y muestra el progreso de la simulación. Durante la ejecución de la simulación todavía es posible seleccionar objetos y editar sus propiedades. Flexsim es un software privativo que tiene varias licencias, existe una para estudiantes que es gratuita pero con pocas funcionalidades (FlexSim Software Products, Inc, 2015).

Los sistemas analizados no son capaces de interpretar y visualizar los ficheros de historial generados por los modelos de simulación usados en el proyecto RCMS. Además, estas aplicaciones son de carácter privativo, imposibilitando el uso o reutilización de sus códigos fuentes para enfrentar el problema planteado. Por los motivos anteriores es necesario desarrollar una nueva solución que resuelva las necesidades de RCMS.

1.3 Técnicas de optimización

Al interactuar con objetos en una escena virtual es necesario un proceso de optimización para poder realizar diversas funciones gráficas. La optimización de la visualización trata de lograr a través de técnicas y algoritmos que se muestre la mayor cantidad de información en el menor tiempo posible. Además, posibilita que los entornos virtuales sean lo más fieles posibles a la realidad, de manera que la interacción con estos les resulte fluida a los usuarios. Lo anterior se logra independientemente de que la carga

de objetos en la escena sea muy grande, o que la cantidad de procesos que se necesiten visualizar sea muy elevada.

1.3.1 Técnicas de recorte o culling

Las técnicas de recorte son aquellas que se utilizan para eliminar de la escena la geometría que no se necesita dibujar en pantalla porque no aparece en la imagen final o no aporta casi nada a la misma. Con esto se logra un incremento considerable de velocidad, sobre todo en escenas donde sólo se visualiza una mínima parte de la misma. Son usadas ampliamente en el mundo de los videojuegos, así como en la realidad virtual y en herramientas para producción de películas (Vega, 2013).

Las técnicas de recorte más utilizadas según (Vega, 2013) son :

- *Backface culling* o recorte de cara de atrás: Elimina los polígonos que se encuentran en la parte oculta detrás de los objetos.
- *Oclussion culling*: Elimina los objetos que no serán visualizados por encontrarse detrás de otro(s) objeto(s).
- *Portal culling*: Deforma el *frustum* identificando los portales o grietas en la visualización y calcula el *frustum* que pasa a través de estos.
- *Frustum culling*: Selecciona los objetos que se encuentran dentro del *frustum* para su representación.

Para satisfacer las necesidades de eficiencia planteadas en este trabajo se consideró conveniente utilizar las siguientes:

Para satisfacer las necesidades para satisfacer las necesidades planteadas en este trabajo se consideró conveniente utilizar *backface culling* pues reduce la cantidad de polígonos a representar y *frustum culling* para evitar la carga innecesaria de objetos que no se encuentran presentes en el campo de visualización.

Para aplicar estas técnicas es necesario realizar procesos de búsquedas sobre los objetos presentes en las simulaciones de RCMS. Estos procesos resultan ser altamente costosos debido a que pueden ser millones de objetos y estos pueden estar ubicados espacialmente en diferentes lugares para instantes de tiempos distintos.

Lo recomendable es hacer uso de estructuras de datos que permiten organizar los objetos en forma jerárquica. Al organizar los objetos en un árbol, la complejidad en

tiempo de búsqueda puede ser reducida a un orden logarítmico a diferencia de una lista o un arreglo que tomaría un tiempo de orden lineal (Vega, 2013).

Estructuras de datos espaciales

El termino estructura de dato espacial es usado para describir una clase de estructuras de datos jerárquicas cuya propiedad común es la descomposición recursiva del espacio que organizan la geometría en 2D, 3D o más dimensiones y son necesarias en la mayoría de las técnicas de aceleración (Akenine-Möller, Haines y Hoffman, 2008). Entre las más usadas se encuentra el Quadtree, el Octree (Samet, 1984) y el R-tree (Guttman, 1984).

Estas estructuras permiten ordenar de forma jerárquica los elementos de una escena, con esto se logra que se pueda acceder a ellos y realizar cualquier acción sobre los mismos de una manera más rápida. El problema de estas radica cuando los elementos no tienen una ubicación constante en el espacio, o sea cambian en el tiempo.

Estructuras de datos espacio temporales

Para atacar este problema han surgido estructuras de datos espacio temporales (EDET). Las EDET permiten almacenar puntos y objetos móviles, u objetos con geometrías cambiantes en el tiempo (Mehta y Sahni, 2004). En la bibliografía revisada se encuentran las siguientes DR-Tree (Gilberto, 2003), D*R-Tree(Gagliardi, Dorzán y Gutiérrez Retamal, 2005), PMR-quadtree for moving object (Tayeb, Ulusoy y Wolfson, 1998), I+3 R-Tree (Gagliardi et al., 2005) y otras.

En el presente documento no se aborda sobre estas estructuras debido a que se hace uso de una biblioteca implementada en el Centro de Estudios de Matemática Computacional que utiliza la estructura D*R-Tree para indexar y recuperar la información espacio temporal.

1.3.2 Niveles de Detalles

Los niveles de detalle (LOD por sus siglas en inglés) son simplificaciones que se le realizan a un modelo complejo, buscando una versión más simple para representarlo, cuando este se encuentre a una larga distancia del observador (Vega, 2013).

Tienen como objetivo la simplificación de la imagen sin crear una degradación visual, que permita una mayor optimización del escenario virtual, pues no es necesario visualizar un objeto de múltiples polígonos si este se encuentra lejos del observador.

Esto permite ahorrar capacidad para cargar otros elementos e incrementar la velocidad en la escena (Vega Infante y Fernández Balbuena, 2012).

Los niveles de detalle almacenan varias versiones de un mismo objeto en dependencia del modelo que vayan a utilizar. Esto permite que según el área que ocupe, se pueda elegir un modelo u otro para ser representado en pantalla. A esto se le conoce como usar diferentes niveles de detalle. Existen algoritmos que se clasifican en niveles de detalles discretos o estáticos, continuos o dinámicos, o dependientes del punto de vista (Vega, 2013).

El **LOD discreto** crea múltiples versiones de cada objeto en tiempo de preprocesamiento, cada una con diferentes niveles de detalle. En tiempo de ejecución el LOD apropiado es seleccionado para representar el objeto. Debido a que los objetos distantes utilizan un LOD rústico, el número total de polígonos se reduce y aumenta la velocidad de visualización (Luebke, 2003).

En la solución presentada se hace uso de los LODs discretos, los mismos tienen varias ventajas. La desvinculación entre simplificación y *redering* convierte a este modelo en el más sencillo de programar: el algoritmo de simplificación puede llegar a tardar todo el tiempo que sea necesario para generar los LODs y en tiempo de ejecución el algoritmo de *rendering* simplemente elige qué nivel de detalle asignarle a cada objeto (Luebke, 2003).

1.3.3 Batching

El *Batching* es una técnica para optimizar la representación de grandes cantidades de objetos sin movimiento. Para una tarjeta gráfica es más rápido dibujar un objeto complejo que miles de objetos simples, porque el cambio entre objetos y materiales consume tiempo. En esta técnica se agrupan los objetos pequeños en uno más grande, para realizar menos cambios (Unity Technologies, 2015b).

1.3.4 Niebla

Al añadir niebla a la escena se permite que los objetos que están más allá de cierta distancia no sean visualizados. Esto puede ser usado para acercar más el plano lejano de la cámara y de esta forma se pueden optimizar las técnicas de recorte de *frustum*. También puede ser utilizada para enmascarar los cambios bruscos en los niveles de detalle (Goldstone, 2011).

1.4 Técnicas para aumentar el realismo

Para que la visualización de simulaciones cumpla las expectativas es necesario dar realismo a las escenas mejorando la calidad visual. Dicho realismo eleva el consumo de recursos computacionales lo que va en contra de la optimización. Los modelos con gráficos pocos reales son criticados por su carencia de credibilidad en la representación de los objetos (Rohrer, 2000). Hay varias formas para hacer que las superficies de los objetos queden más reales con un equilibrio entre optimización y realismo. A continuación, se detallan algunas de las formas para lograr realismo.

Iluminación: La iluminación es una de las técnicas usadas para obtener realismo en las escenas. Esta se refiere a colocar luces en escena para obtener efectos con gran aproximación a la realidad. Según (Gregory, 2009) la clave para representar imágenes foto-realistas es tener bien en cuenta el comportamiento de la luz a medida que interactúa con los objetos de la escena. La iluminación es el corazón de toda la representación de gráficos por ordenador. Sin una buena iluminación, una escena muy bien modelada se verá plana y artificial. De igual forma, incluso la más simple de las escenas puede parecer muy realista cuando se ilumina con precisión.

Para calcular el sombreado de un objeto 3D, es necesario saber la intensidad, la dirección y el color de la luz que incide sobre él (Unity Technologies, 2015). Una de las cuestiones relacionadas con la iluminación, que es menos notable, pero que todavía tiene una importante contribución al realismo, es la oclusión ambiental. En el mundo real, los espacios pequeños y esquinas cóncavas reciben menos luz, por lo tanto, aparecen más oscuros. La oclusión ambiental utiliza algunos algoritmos rápidos para detectar la mayoría de estas áreas, y las oscurece en consecuencia (Bijl y Boer, 2011).

Texturizado: Las texturas son imágenes o fotos que se aplican a una forma 3D. En motores de juegos modernos, cada forma utiliza varias texturas, especificando las diferentes propiedades de la superficie (Bijl y Boer, 2011).

Sombreado: Las Sombras juegan un papel importante en el realismo de los gráficos. No sólo porque hacen la iluminación de la escena más creíble, sino también porque aumentan la percepción de la profundidad (Bijl y Boer, 2011).

1.5 Tecnologías, Herramientas y Metodología a utilizar

A continuación, se definirán una serie de herramientas que son utilizadas para desarrollar el tipo de solución que se presenta en este trabajo.

1.5.1 Tecnologías para la Representación 2D/3D

Actualmente existe una extensa cantidad de motores de video juegos que pueden facilitar el desarrollo del visualizador. En particular existen dos herramientas de software que se destacan: Unity5 (Unity) y Unreal Engine4 (Unreal Engine) debido mayormente a su popularidad, funcionalidad y versatilidad. Tomando en cuenta las características de la aplicación que se quiere construir fueron analizados varios factores para la selección del motor de video juego (ver Tabla 1), como: licencia, gráficos, iluminación y sombras, lenguajes de programación, herramientas para analizar rendimiento, requisitos mínimos para el desarrollo, documentación, comunidad. Los datos de Unity y Unreal Engine que se presentan Tabla 1 fueron extraídos de (Unity Technologies, 2015a) y (Epic Games, 2015) respectivamente.

Tabla 1 Comparación entre Unity y Unreal Engine

	Unity	Unreal Engine
Licencia	Tiene dos licencias una Personal y otra Profesional, la Personal permite usar todas las características del motor, generalmente es usada para fines educativos, tiene ciertas restricciones a la hora de comercializar los productos. Por su parte la Profesional tiene un precio de $1500 y otros $1500 por plataforma a exportar.	Una libre de uso con regalías. Para juegos o aplicaciones comerciales es necesario pagar el 5% de todos los ingresos brutos después de los primeros $3000. Puede ser usado para proyectos de cine, la educación, la contratación y la consulta de proyectos como la arquitectura, la simulación y visualización sin pagar regalías.
Gráficos	Integra la tecnología de iluminación en tiempo real Enlighten. Incluye Sondas de reflexión HDR para mejorar la fidelidad visual. El Standard Shader basado en la física para hacer que sus materiales se vean consistentes en	Unreal Engine 4 tiene soportes avanzados para DirectX 11 y 12 ofreciendo características de representación como Reflexión HDR, miles de luces dinámicas por escenas, teselación y desplazamiento programable por el artista, sombras y materiales

	cualquier entorno de iluminación y plataforma.	basado en la física y perfil de iluminación IES.
Iluminación y Sombras	Unity ofrece Iluminación global en tiempo real Enlighten, efectos de posprocesamiento de pantalla completa, sombras suaves para luz direccional y Deferred Shading.	Tiene funcionalidades avanzadas de iluminación dinámica y un sistema de partículas que puede manejar hasta un millón de partículas en una escena a la vez.
Soporta Texturas	Si.	Si.
Incluye Terrenos	Si.	Si.
Lenguajes de programación	C#, JavaScript, Boo.	C++, Blueprints para scripting gráfico, una versión avanzada de Kismet con la que se puede llegar a realizar por completo un videojuego sin necesidad de programar en C++.
Herramientas para analizar rendimiento	Si.	Si.
Requisitos mínimos para el desarrollo	Sistema Operativo: Windows 7 SP1+, 8, 10; Mac OS X 10.8+. Tarjeta gráfica: Capacidades de tarjeta de vídeo con DX9 (modelo de shader 2.0). Todo lo que se haya lanzado desde 2004 debería funcionar. El resto depende principalmente	PC de escritorio con Windows 7 64-bit o Mac con Mac OS X 10.9.2 o superior. Memoria RAM de 8GB+, procesador quad-core Intel o AMD. Tarjeta de video con soporte para DX11. Puede correr en PC con menos prestaciones, pero el rendimiento puede ser limitado.

	de la complejidad de sus proyectos.	
Documentación	Documentación en el sitio oficial la cual incluye tutoriales, en el manual, en el fórum de la comunidad.	Documentación en sitio oficial y tutoriales disponibles en Unreal Developer Network (UDN refiere a sus siglas en inglés).
Comunidad	Amplia comunidad que aporta conocimiento.	Tiene una comunidad con muy pocos usuarios.

El análisis realizado muestra que ambos motores de juego propuestos pueden ser utilizados con el propósito de visualizar simulaciones. En el presente trabajo se seleccionó Unity5 teniendo en cuenta que el número de usuarios, la comunidad y soporte con que este cuenta supera considerablemente la de Unreal Engine. En la Figura 1 se muestra un análisis de popularidad según Google Trends donde puede observase como Unity supera a Unreal Engine. Unity consume menos recursos computacionales que Unreal Engine. Además, los lenguajes de programación que utiliza son de fácil implementación como es el caso de C#, el cual permite una mejor estructura de clases en el proceso de implementación.

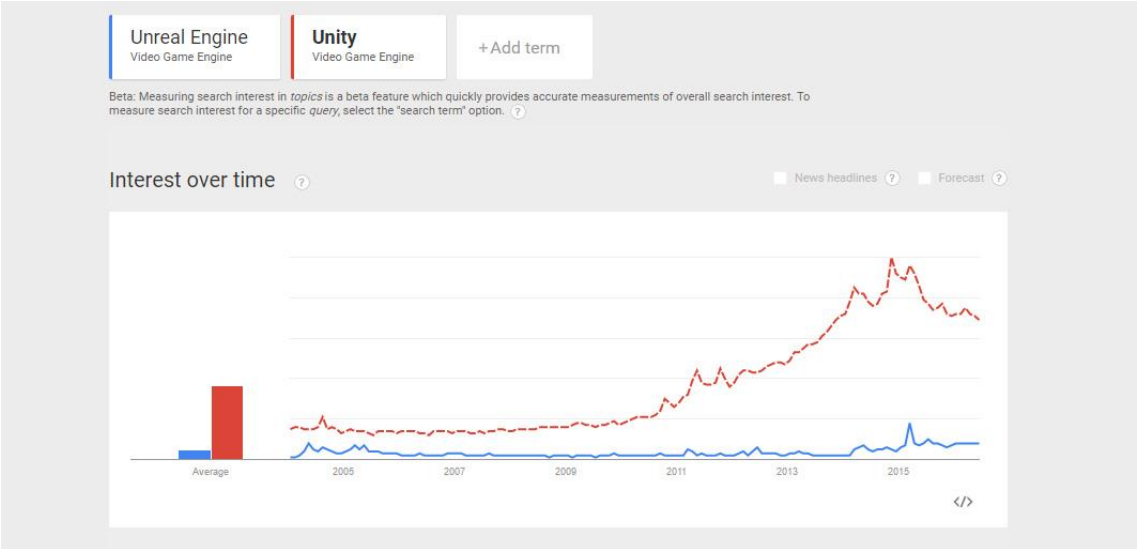


Figura 1 Análisis de popularidad de Unity y Unreal Engine (tomado de: https://www.google.com/trends)

1.5.2 Metodología de desarrollo

“Una metodología es una colección de procedimientos, técnicas, herramientas y documentos auxiliares que ayudan a los desarrolladores de software en sus esfuerzos por implementar nuevos sistemas de información.” (Avison y Fitzgerald, 1995). En los últimos años el uso de las metodologías ha incidido positivamente en la calidad del software.

Las metodologías de desarrollo de software se dividen en dos vertientes:

- Metodologías Tradicionales o Pesadas.
- Metodologías Ágiles.

Las tradicionales llevan una documentación exhaustiva de todo el proyecto. Se utilizan en grandes equipos de desarrollo y necesitan generar una gran cantidad de documentación para mantener controlado el proceso de desarrollo y gestionar la comunicación entre los departamentos en los que puede dividirse el equipo. Las metodologías ágiles son flexibles ante requisitos cambiantes y son utilizadas en equipos de desarrollo pequeños.

Con las características mencionadas relacionadas a ambas vertientes, las metodologías tradicionales son descartadas, por tanto, para la solución a desarrollar se decide el uso de una metodología ágil. Entre las metodologías ágiles fueron analizas EXtreme Programing (XP) y OpenUp. La elección de las metodologías a analizar se determinó tratando de elegir las más utilizadas y cubrir diferentes aspectos y enfoques entre las metodologías.

XP

La metodología XP consiste en una programación rápida o extrema centrada en la simplicidad de las soluciones implementadas y coraje para enfrentar los cambios en los requisitos y las tecnologías. Incorpora al cliente como un miembro más del equipo. Se enfoca en potenciar las relaciones interpersonales como clave para el éxito en el desarrollo de software, promueve el trabajo en equipo, se preocupa por el aprendizaje de los desarrolladores, y propicia un buen clima de trabajo. Es adecuada para proyectos pequeños con requisitos imprecisos y muy cambiantes. Elimina las actividades improductivas para reducir los costos y la frustración de todos los involucrados (Wells, 2013).

A continuación, se detallan las principales ventajas de la metodología XP: (Figueroa, Solís y Cabrera, 2008)

- Apropiada para entornos volátiles.
- Preparada para el cambio, lo que significa reducir su coste.
- Planificación más transparente para los clientes, conocen las fechas de entrega de las funcionalidades.
- Permite definir en cada iteración cuáles son los objetivos de la siguiente.
- La presión está a lo largo de todo el proyecto y no en una entrega final.

OpenUp

OpenUP es un proceso unificado ligero que aplica enfoques iterativos e incrementales y define las fases, actividades y artefactos que se generan durante el ciclo de desarrollo del software. Tiene una filosofía ágil que se centra en la naturaleza colaborativa de desarrollo de software. Ofrece los artefactos esenciales necesarios para capturar y comunicar decisiones. La finalidad de esta metodología de desarrollo es garantizar la eficacia mediante el cumplimiento de los requisitos iniciales y minimizar las pérdidas de tiempo en el proceso de generación del software (Eclipse Foundation, 2013).

Entre las ventajas de OpenUP se encuentran:

- Es extensible pues los procesos se pueden agregar o adaptar según lo vayan requiriendo los sistemas.
- Es ligero y proporciona una comprensión detallada del proyecto, beneficiando a clientes y desarrolladores sobre productos a entregar y su formalidad.
- Se centra en una arquitectura temprana para reducir al mínimo los riesgos y organizar el desarrollo.
- Maneja el ciclo de vida del desarrollo de software de manera apropiada al ofrecer una buena administración de las áreas del proyecto.

Se selecciona como metodología de desarrollo OpenUP por ser una metodología ágil que aplica un enfoque iterativo e incremental dentro de su estructura del ciclo de vida y que puede adaptarse para desarrollar diversos tipos de proyectos. OpenUP es apropiado para proyectos pequeños y de bajos recursos, permite disminuir las probabilidades de fracaso en los proyectos pequeños e incrementar las probabilidades de éxito. Permite detectar errores tempranos a través de un ciclo iterativo, evita el uso de una documentación exhaustiva ya que solo incluye el contenido fundamental. Tiene un enfoque centrado al cliente y con iteraciones cortas.

1.5.3 Lenguajes de programación

C #

C# es un lenguaje de programación simple, moderno, orientado a objetos y de tipo seguro. Forma parte de la plataforma .NET de Microsoft, aunque es un lenguaje de programación independiente (Microsoft, 2016c). Las muchas innovaciones en C# permiten el desarrollo rápido de aplicaciones al tiempo que conserva la expresividad y la elegancia de los lenguajes del estilo C (Microsoft, 2016b).

Se selecciona C# por ser de los lenguajes usados para programar los scripts en Unity, además el desarrollador del proyecto tiene experiencia en su uso.

1.5.4 Entorno de Desarrollo Integrado

Un Entorno Integrado de Desarrollo (IDE por sus siglas en inglés) es un software que provee facilidades comprensivas a los programadores de computadora para el desarrollo de software. Para el desarrollo de la solución se seleccionó el IDE Visual Studio 14.0.23107.0 D14REL del 2015. Visual Studio y Unity se integran por medio de la extensión Visual Studio Tools. Con dicha extensión el desarrollador dispone de todas las funciones del IDE, el depurador de Visual Studio y las funciones de productividad diseñadas para desarrolladores de Unity. Visual Studio Tools para Unity 2.0 Preview 2 agrega compatibilidad con Visual Studio 2015, además de una serie de características nuevas, como, por ejemplo, una mejor visualización de los objetos en las ventanas Inspección y Variables locales (Microsoft, 2016a). Visual Studio fue seleccionado por las características que se mencionan a continuación:

- Visual Studio permite a los desarrolladores de Unity depurar el proyecto.
- Muestra el explorador de proyectos que permite visualizar todos los ficheros y archivos en la misma jerarquía que lo hace Unity Editor.
- Incrementa la productividad aprovechando todas las funciones que Visual Studio ofrece como *IntelliSense*[3], refactorización, y las capacidades de búsqueda en el código.
- Incorpora una lista de errores que incluye los mostrados en la consola de Unity.

[3] https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx

1.5.5 Lenguaje de modelado

Para la comunicación de ideas entre desarrolladores y para el análisis de algunos procesos es necesaria una forma estandarizada de representar un modelo o diseño. Para esto se necesita un lenguaje en el que se pueda modelar lo antes descrito.

Durante el desarrollo de la solución se utilizará como lenguaje de modelado el Lenguaje Unificado de Modelado (UML por sus siglas en inglés) en su versión 2.0, el cual permite especificar, visualizar y construir los artefactos que exigen la ingeniería del software (Object Management Group, 2005).

1.5.6 Herramienta CASE

Las herramientas para la Ingeniería de Software Asistida por Computadoras (CASE por sus siglas en inglés) son aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software, ayudan a economizar el consumo de recursos como puede ser el tiempo. Estas herramientas pueden ayudar en todos los aspectos del ciclo de vida de desarrollo del software, ya sea en el diseño del proyecto, cálculo de costos, implementación de parte del código automáticamente a partir del diseño, documentación, entre otras. La herramienta CASE seleccionada para realizar los artefactos pertinentes a las etapas de análisis y diseño de la solución es Visual Paradigm para UML en su versión 8.0 pues se caracteriza por:

- Uso de un lenguaje estándar común a todo el equipo de desarrollo que facilita la comunicación.
- Diseño centrado en casos de uso y enfocado al negocio.
- Capacidades de ingeniería directa e inversa.
- Modelo y código que permanece sincronizado en todo el ciclo de desarrollo.
- Diagramas de flujo de datos.
- Disponibilidad en múltiples plataformas (Microsoft Windows y GNU\Linux).

1.6 Conclusiones

En la elaboración de este capítulo se abordaron conceptos generales que sirvieron de guía para lograr un mejor entendimiento de esta investigación. El estudio de algunas herramientas con funcionalidades similares a las deseadas para dar solución al problema a resolver, estas permitieron constatar la necesidad de implementar un nuevo sistema.

Para optimizar el proceso de visualización fueron estudiadas y analizadas las técnicas de optimización. Como resultado del estudio y análisis se decidió utilizar el *Frustum Culling*, el *Backface Culling*, los Niveles de Detalle, el *Batching* y la Niebla. Además, se analizaron las técnicas más utilizadas para aumentar el realismo mejorando la calidad, lo que permitió decidir que se utilizaría la iluminación, el texturizado y el sombreado.

Además, fueron analizadas las tecnologías y herramientas que se utilizan en la representación de contenido en tres dimensiones. Se selecciona el motor gráfico Unity y el lenguaje de programación C#. Como entorno de desarrollo va a ser utilizado Visual Studio y OpenUp como metodología de desarrollo. Para realizar los diagramas de análisis, diseño e implementación de la aplicación se hará uso de Visual Paradigm 8.0 como herramienta CASE y UML como lenguaje de modelado.

Capítulo 2 Análisis y Diseño

En este capítulo se muestran los artefactos que más se ajustan a las necesidades de la solución, generados a partir de la metodología OpenUp. Se muestra la arquitectura del sistema, así como la descripción de los elementos que la componen. Se presentan los requisitos funcionales y no funcionales detectados durante el proceso de captura de requisitos, así como los casos de uso que los agrupan.

2.1 Proceso de visualización

En el proceso de representación de simulaciones de eventos discretos se utilizarán cuatro fases que se relacionan entre sí, de tal forma que los datos de salida de una fase van a ser los datos de entrada de la siguiente. Las fases en que se desglosa este proceso son:

1. Preprocesamiento del historial.
2. Selección y actualización de los objetos a visualizar.
3. Modelación 3D.
4. Visualización.

En la primera fase es seleccionado el fichero de historial que se desea representar. Si ya se encuentra creada la EDET correspondiente se pasa a la segunda fase, en caso contrario se procede a extraer los datos de dicho fichero y a almacenarlos mediante una EDET.

La segunda fase tiene como dato de entrada la referencia a la EDET, el campo de visualización de la cámara o frustum y el tiempo actual de la simulación. Los objetos que se encuentran dentro del campo de visión se obtienen a partir de consultas sobre la EDET. Posterior a la consulta se procede a actualizar la escena con los nuevos objetos y a eliminar los que ya no son necesarios para la visualización.

En la tercera fase a cada uno de los objetos nuevos a representar se le asigna el modelo 3D correspondiente en conjunto con las descripciones de sus movimientos.

La cuarta fase recibe como entrada los objetos modelados. A estos se les aplican técnicas de optimización y técnicas para obtener realismo. Como resultado se obtiene una representación en 3D de los mismos.

2.2 Especificación de los requisitos del sistema.

La ingeniería de requisitos es el amplio espectro de tareas y técnicas que conducen a la comprensión de los requisitos (Pressman, 2014). Con esta se facilita el entendimiento de lo que desea el cliente, arribando a una solución razonable y sin ambigüedades. Estos requerimientos permiten confirmar la viabilidad de la solución, gestionándolos de forma correcta se pueden transformar en un software funcional ya que determinan qué hará el mismo y definen las restricciones de su operación e implementación.

2.2.1 Requisitos funcionales

Los requisitos funcionales de un sistema describen el comportamiento del sistema. Estos requisitos dependen del tipo de software que se comience a desarrollar, lo que esperan los usuarios del software y el enfoque general tomado por la organización al escribir los requisitos (Sommerville, 2011).

RF1 Visualizar simulación.

Representa en pantalla la simulación.

RF2 Manipular reproducción de la simulación.

RF2.1 Iniciar visualización.

Inicia la visualización de la simulación.

RF2.2 Pausar visualización.

Detiene la visualización de la simulación.

RF2.3 Aumentar velocidad de visualización.

Aumenta la velocidad de la simulación.

RF2.4 Disminuir velocidad de visualización.

Disminuye la velocidad de la simulación.

RF3 Manipular escena.

RF3.1 Acercar escena.

Permite visualizar de un punto más cercano el escenario de la simulación.

RF3.2 Alejar escena.

Permite visualizar de un punto más lejano el escenario de la simulación.

RF3.3 Rotar escena.

Permite al usuario cambiar el punto que la cámara tiene como objetivo moviendo el puntero del ratón.

RF3.4 Desplazar escena.

Permite al usuario que mueva el escenario, variando con el puntero del ratón la posición de la vista que se presenta.

2.2.2 Requisitos no funcionales.

Pueden estar relacionados con las propiedades del sistema como la fiabilidad, tiempo de respuesta y la capacidad de almacenamiento. Alternativamente, pueden definir restricciones en la implementación del sistema, como las capacidades de los dispositivos de Entrada/Salida (E/S) o de las representaciones de datos utilizadas en las interfaces con otros sistemas. Los requisitos no funcionales, como el rendimiento, la seguridad y la disponibilidad, suelen especificar o restringir características del sistema en su conjunto (Sommerville, 2011).

Requerimientos de Hardware

Los requerimientos mínimos de hardware que deben existir en las computadoras para poder asegurar un correcto funcionamiento del sistema son:

RNF1.

- Procesador: Dual Core 2.60 GHZ
- Memoria RAM: 2 GB
- Tarjeta Gráfica: NVIDIA GeForce GTX 260. La tarjeta utilizada debe tener soporte para DX9 (shader modelo 2.0).

Requerimientos de Eficiencia

RNF2. Rendimiento: La velocidad de visualización debe ser de 30 o más tramas por segundo.

Requerimientos de Software

RNF3. La versión del sistema operativo para ejecutar la aplicación debe ser alguna de las siguientes: Windows XP SP2 o superior, Mac OS X 10.8 o superior, Ubuntu 12.04 o superior.

2.3 Actores del sistema

Un actor es una persona o sistema que se comunica con el sistema o producto y que es externo al mismo. Son quienes usan el producto dentro del contexto de la función y comportamiento que debe ser descrito (Pressman, 2014). En la Tabla 2 se describe el actor del sistema.

Tabla 2 Actor del Sistema

Actor	Descripción
Sistema	Sistema que ejecuta la aplicación.
Usuario	Persona que interactúa con el Visor de Simulaciones.

2.4 Casos de Uso del Sistema

Un caso de uso es lo que describe una función o característica del sistema desde el punto de vista del usuario. Sirve como una base para la creación de un modelo de requisitos más comprensivo (Pressman, 2014).

Diagrama de Casos de Uso del Sistema

El Diagrama de Casos de Uso refleja cómo los actores interactúan con los casos de uso (Figura 2).

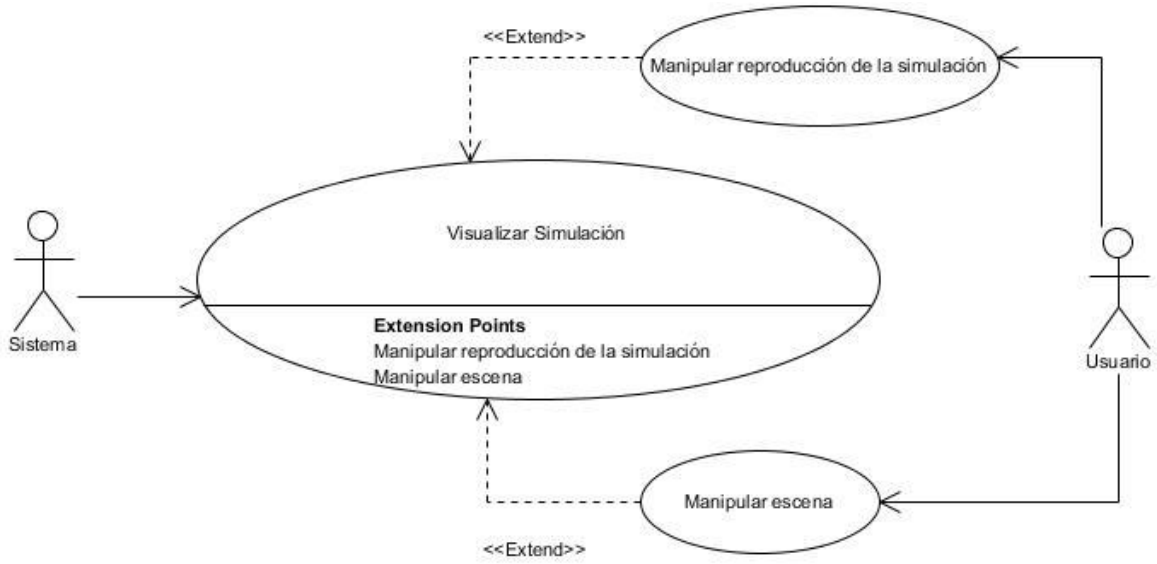


Figura 2 Diagrama de Casos de Uso del Sistema

Descripción de los casos de uso del sistema

Los casos de uso se elaboran adicionalmente para proporcionar considerablemente más detalles sobre la interacción. Los casos de uso se escriben a menudo informalmente (Pressman, 2014). Sin embargo, la descripción formal que se muestra en las tablas 3, 4, 5, es para asegurar que se aborden todas las cuestiones claves.

Tabla 3 Descripción del caso de uso Visualizar simulación

Objetivo	Visualizar el historial de la simulación seleccionada.
Actores	Usuario (inicia)
Resumen	El caso de uso se inicia cuando el usuario selecciona la opción visualizar simulación. Finaliza cuando se visualiza en pantalla la simulación.
Complejidad	Alta.
Prioridad	Crítica.
Referencias	RF1.
Precondiciones	Debe existir un fichero de historial de una simulación.
Poscondiciones	Se visualiza la simulación.
Flujo de eventos	

Flujo básico <Visualizar simulación>		
	Actor	**Sistema**
1. Proporciona el fichero de historial de simulación.		2. Comprueba si existe el fichero proporcionado. 3. Comprueba si existe un fichero con el mismo nombre y extensión .det. 4. Extrae los datos del fichero de historial. 5. Inserta los datos en una EDET. 6. Guarda la EDET en un fichero binario con extensión *.det*. 7. Carga los datos necesarios de la EDET. 8. Representa la simulación en la escena.
9. Presiona el botón derecho del ratón en el botón cerrar.		10. Termina ejecución de la aplicación y finaliza.
Flujos alternos		
No existe el fichero de historial pasado por parámetro		
	Actor	Sistema
		2.1. Notifica al usuario y finaliza.
Existe el fichero .det		
	Actor	Sistema
		3.1. Carga una referencia de la EDET y continua con el flujo normal en 7.
Relaciones	**CU Incluidos**	No incluye otros casos de uso.
	CU Extendidos	Manipular la reproducción de la simulación, Manipular escena.
Requisitos no funcionales	RNF1, RNF2, RNF3.	

Asuntos pendientes	No se considera la realización de ningún asunto de este tipo.

Tabla 4 Descripción del caso de uso Manipular reproducción de la Simulación

Objetivo	Manipular la reproducción de la simulación.
Actores	Usuario (inicia)
Resumen	El caso de uso se inicia cuando el usuario realiza una acción para cambiar el estado de la reproducción. Finaliza cuando el estado de la reproducción se cambia.
Complejidad	Media.
Prioridad	Normal.
Referencias	RF2.
Precondiciones	Se está visualizado una simulación.
Poscondiciones	Se modifica la forma de reproducir la simulación.
Flujo de eventos	
Flujo básico <Manipular la reproducción de la simulación>	

	Actor	**Sistema**
1.	El caso de uso se inicia cuando el usuario decide realizar una de las siguientes opciones: • Si desea iniciar o pausar la visualización ir a sección "Iniciar o pausar visualización". • Si desea aumentar la velocidad de reproducción ir a sección “Aumentar velocidad de reproducción”.	

• Si desea disminuir la velocidad de reproducción ir a sección “Disminuir velocidad de reproducción”.	
Sección 1 "Iniciar o pausar visualización"	
Actor	**Sistema**
1.1. Presiona el botón izquierdo del ratón sobre el icono de play-pause.	1.2. Inicia a reproducirse la simulación a una velocidad igual a la que estaba antes de entrar en pause.
Flujo alterno de la sección 1 "Iniciar o pausar visualización"	
Se está reproduciendo la simulación	
Actor	**Sistema**
	1.2.1. Se detiene la reproducción y finaliza.
Sección 2 “Aumentar velocidad de reproducción”	
Actor	**Sistema**
2.1. Presiona el botón izquierdo del ratón sobre el icono de aumentar velocidad.	2.2. Aumenta la velocidad de reproducción y finaliza.
Flujo alterno de la sección 2 “Aumentar velocidad de reproducción”	
La simulación estaba en pause	
Actor	**Sistema**
	2.2.1. No se aumenta la velocidad y finaliza.
Sección 3 “Disminuir velocidad de reproducción”	
Actor	**Sistema**

3.1. Presiona el botón izquierdo del ratón sobre el icono de disminuir velocidad.	3.2. Disminuye la velocidad de reproducción y finaliza.
Flujos alterno de la sección 3 “Disminuir velocidad de reproducción”	
Actor	**Sistema**
	3.2.1. No se disminuye la velocidad y finaliza.

Relaciones	**CU Incluidos**	No incluye otros casos de uso.
	CU Extendidos	No extiende otros casos de uso.
Requisitos no funcionales	RNF1, RNF2.	
Asuntos pendientes	No se considera la realización de ningún asunto de este tipo.	

Tabla 5 Descripción del caso de uso Manipular escena

Objetivo	Cambiar la forma en que el usuario ve escenario.
Actores	Usuario (inicia)
Resumen	El caso de uso se inicia cuando el usuario utiliza el ratón para controlar la cámara. Finaliza cuando el sistema responde acorde a la acción realizada por el usuario.
Complejidad	Media.
Prioridad	Normal.
Referencias	RF3.
Precondiciones	Se está visualizado una simulación.
Poscondiciones	Se modifica la forma de visualizar el escenario.
Flujo de eventos	
Flujo básico <Manipular escena>	

	Actor	Sistema
1. El caso de uso se inicia cuando el usuario decide realizar una de las siguientes opciones: • Si desea rotar la cámara ir a sección "Rotar cámara". • Si desea alejar la escena ir a sección "Alejar escena". • Si desea acercar escena ir a sección "Acercar escena". • Si desea mover escena ir a sección "Mover escena".		
Sección 1 “Rotar cámara”		
	Actor	**Sistema**
	1.1. Presiona el botón izquierdo del ratón y lo mueve sobre el área visualizada.	1.2. Gira la cámara en el mismo sentido y dirección que el movimiento del ratón.
Sección 2 “Alejar escena”		
	Actor	**Sistema**
	2.1. Gira la rueda del ratón hacia atrás.	2.2. Aleja la escena y finaliza.
Sección 3 “Acercar escena”		
	Actor	**Sistema**
	3.1. Gira la rueda del ratón hacia delante.	3.2. Acerca la escena y finaliza.
Sección 4 “Mover escena”		
	Actor	**Sistema**
	4.1. Presiona el botón derecho del ratón y lo mueve sobre el área visualizada.	4.2. La escena se mueve en el mismo sentido y dirección que el movimiento del ratón.
Relaciones	**CU Incluidos**	No incluye otros casos de uso.

	CU Extendidos	No extiende otros casos de uso.
Requisitos no funcionales	RNF1, RNF2.	
Asuntos pendientes	No se considera la realización de ningún asunto de este tipo.	

2.5 Arquitectura del sistema

La arquitectura de software debe modelar la estructura de un sistema y la manera en que los datos y los componentes colaboran unos con otros (Pressman, 2014). La arquitectura en capa organiza el sistema en capas con funcionalidad relacionada asociada a cada capa. Las capas proporcionan servicios a la capa por encima de ella por lo que las capas de nivel más bajo representan los servicios centrales que son susceptibles de ser utilizados en todo el sistema. Entre sus ventajas de encuentra que logra separación e independencia, lo cual es fundamental en un diseño arquitectónico ya que permite que los cambios sean localizados. El enfoque en capas ayuda el desarrollo incremental de los sistemas. Mientras se desarrolla una capa, alguno de los servicios prestados por esa capa puede ponerse a disposición de los usuarios. Esta arquitectura es cambiable y portable. En tanto la interfaz no tenga cambios, la capa puede ser sustituida por otra equivalente. Además, cuando las interfaces de una capa cambian o se añaden nuevas facilidades a una capa, sólo la capa adyacente se ve afectada (Somerville, 2011). La arquitectura del Visor de Simulaciones fue definida en capas, definiendo para el software tres capas:

- Presentación: Es la que se encarga de la visualización de los datos y de que el usuario interactúe con el sistema. Está formada por los elementos que posibilitan la visualización y la interacción del usuario con el sistema como botones, textos que brindan información y los objetos 3D generados para la visualización.
- Lógica de Negocio: Se encarga de construir y actualizar la escena. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados tras la consulta a la capa de Acceso a Datos.
- Acceso a Datos: Construye la consulta basada en los paramentos recibidos desde la capa de lógica de negocio. Realiza la consulta a la estructura de datos almacenada en disco y devuelve la respuesta a la capa lógica.

Diagrama de clases

El diagrama de clases describe gráficamente las especificaciones de las clases de software (Larman y Valle, 2003). En otras palabras, se evidencian las clases con sus relaciones. Para un mejor entendimiento de la solución se plantea un diagrama de clases donde los *gameObject* se muestran siguiendo la analogía de los diagramas UML en forma de clases, conteniendo los scripts que heredan de *MonoBehaviour*.

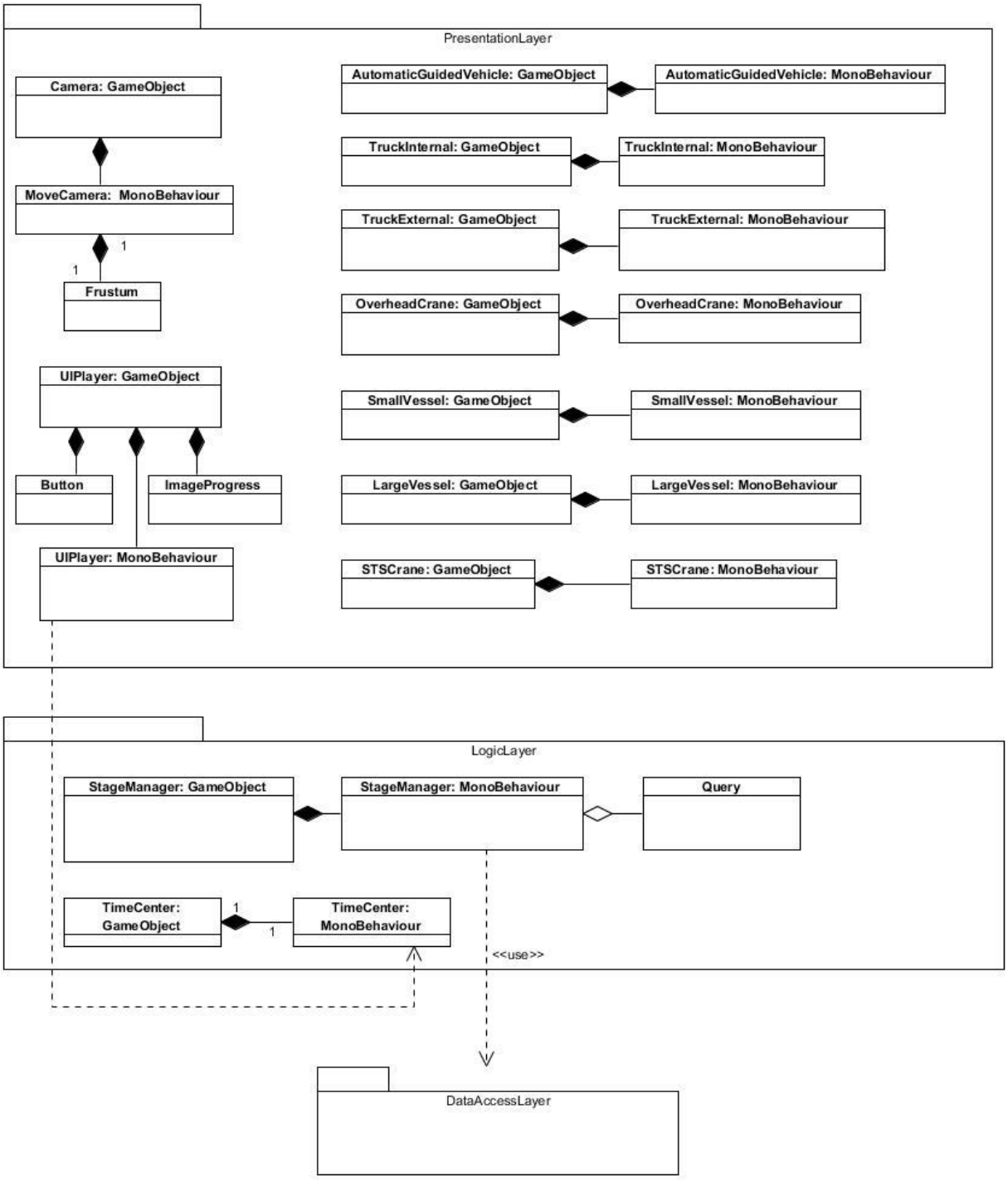


Figura 3 Diagrama de clases

MoveCamera: Es un script que permite al usuario desplazar la cámara por el escenario.

Frustum: Describe el campo de visión de la cámara en forma de pirámide.

UIPlayer: Es un script que actualiza los controles de la reproducción y captura los eventos producidos en estos.

AutomaticGuidedVehicle, TruckInternal, TruckExternal, OverheadCrane, SmallVessel, LargeVessel, STSCrane: Son los scripts encargados de realizar las transformaciones en los objetos visualizados.

StageManager: Se encarga de crear la escena. Así como de cargar y descargar los objetos innecesarios.

TimeCenter: Representa el tiempo en el que transcurre la simulación, este puede avanzar a menor, igual o menor velocidad que un reloj común.

2.6 Patrones de diseño

Los patrones de diseño de software son los que permiten describir fragmentos de diseño y reutilizar ideas de diseño, ayudando a beneficiarse de la experiencia de otros. "… Los patrones de diseño comunican los estilos y soluciones consideradas como buenas prácticas, que los expertos en el diseño orientado a objetos utilizan para la creación de sistemas." (Larman y Valle, 2003).

Entre los patrones de diseño más conocidos están los Patrones Generales de Asignación de Responsabilidades de Software, más conocidos por sus siglas en inglés como GRASP (Larman y Valle, 2003). Para lograr un diseño eficaz se utilizaron los que se describen a continuación.

Patrón Experto: Es asignar una responsabilidad al experto en información. El experto en información es la clase que posee la información necesaria para cumplir con dicha responsabilidad (Larman y Valle 2003). Este patrón es usado en todas las clases ya que cada una posee la información para realizar la tarea que le corresponde. Un ejemplo se evidencia en la clase GameManager, que es responsable de actualizar la escena. Para actualizar la escena es necesario conocer los objetos que se están visualizado actualmente y los que serán visualizados en la escena, como esta clase es la que maneja esa información entonces es la responsable de actualizar la escena.

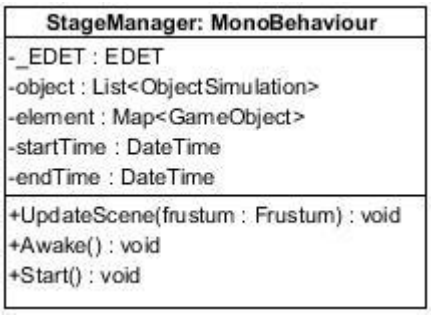


Figura 4 Diagrama de la clase StageManager

Patrón Creador: El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos. El propósito fundamental de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento (Larman y Valle, 2003). En la solución se evidencia la utilización de este patrón en la clase MoveCamera que es responsable de crear las instancias de la clase Frustum que describen el campo de visión de la cámara.

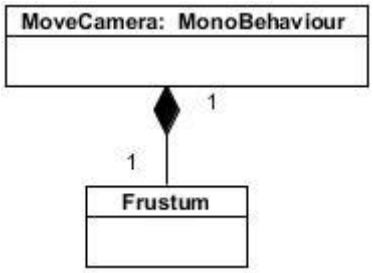


Figura 5 Relación entre la clase MoveCamera y la clase Frustum

Patrón Bajo Acoplamiento: El acoplamiento es "… una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos." (Larman y Valle, 2003).Un elemento con bajo acoplamiento sufre de pocos cambios o ninguno al realizarse modificaciones en las clases con las que se relaciona por lo que facilita su reutilización ya que no tiene dependencias fuertes con otros elementos. Con el objetivo de lograr un diseño flexible, con facilidad de soporte y que posea elementos reutilizables se utiliza este patrón.

Otros patrones muy usados en el desarrollo de aplicaciones son los conocidos como **GoF** en la solución fueron usados los siguientes:

Singleton: El patrón singleton tiene como objetivo asegurar que una clase solo posee una instancia y proporcionar un método de la clase único que devuelva esta instancia. Este se usa en la aplicación con el objetivo de que exista una única instancia de la clase TimeCenter.

Adapter: Este patrón tiene como objetivo convertir la interfaz de una clase existente en la interfaz esperada por los clientes también existentes para que puedan trabajar de forma conjunta. Es utilizado en la clase EDETAdapter que implementa los métodos para el Acceso a Datos.

2.7 Conclusiones del capítulo

Durante el análisis y diseño de la solución se definieron los requisitos funcionales y no funcionales que apoyarán al desarrollo de un software útil y funcional. La utilización de los patrones GRASP y GoF permitió dar respuestas eficientes a algunos problemas de diseño. Se propuso un proceso para la visualización de simulaciones creadas por RCMS, detallando los pasos para poder realizar su implementación.

Con una implementación que se ajuste al diseño y modelado de la solución se logrará una aplicación sencilla, robusta, de fácil comprensión, mantenimiento y reutilización, satisfaciendo los requisitos funcionales y no funcionales.

Capítulo 3

En este capítulo se describe cómo se desarrolló la solución, en el mismo se muestra el diagrama de componentes correspondiente a las funcionalidades desarrolladas y las descripciones para su entendimiento. También se presentarán el estándar de codificación usado, la implementación de los algoritmos principales, las pruebas de software realizadas y por último la validación de la solución.

3.1 Estándares de Codificación

Los estándares de codificación definen directrices para aplicar un estilo y formato coherentes (Hunt, 2007). Permite que el código sea más fácil de entender y ayuda a los desarrolladores a evitar errores comunes de codificación. Los estándares de codificación que se siguieron en la presente solución son los estándares de C# para .NET detallados en (Hunt, 2007). A continuación, se describirán los aspectos más relevantes de dichos estándares.

3.1.1 Convenciones de nombre

En esta sección se describen algunas convenciones usadas para los nombres de variables, métodos, clases, atributos, entre otros (Tabla 3). De esta forma se logra que el código tenga una mayor consistencia, siendo la clave para obtener un código sostenible (Hunt, 2007).

Tabla 6 Uso y sintaxis de nombre

Identificador	Convención
Archivo fuente	Se escribe en *Pascal Case*[4]. El nombre de la clase y el fichero son el mismo.
Clase o estructura	Se escribe en *Pascal Case*.
Método	Se escribe en *Pascal Case*. Se utiliza un verbo o un par verbo-sustantivo.

[4] *Pascal Case*: Una palabra con la primera letra capitalizada y la primera letra de cada palabra o parte de palabra subsecuente capitalizada.

Propiedad	Se escribe en *Pascal Case*.
	
	El nombre siempre representa la entidad a la que pertenece la propiedad, usando la misma palabra.
	
	Ejemplo:
	
	public string Name
	
	{
	
	get { return _name; }
	
	set { _name = value; }
	
	}

Atributo de clase	Si es *public*, *protected* o *internal* se escriben en *Pascal Case*. Si es *private* se escribe en *Camel Case*[5] y con un prefijo "_". Ejemplo: protected string Name; private DateTime _time;
Variable	Se escribe en *Camel Case*. No se enumeran variables como text1, text2, etc;
Parámetro	Se escribe en *Camel Case*.

3.1.2 Estilo de Código

Es la manera de dar formato al código con el objetivo de crear una implementación más legible, clara, consistente y de fácil mantenimiento. Un mal estilo de código puede causar polémica entre desarrolladores. A continuación, se mencionan algunas de las convenciones de estilo de código más relevantes en la solución.

[5] Camel Case: Una palabra con la primera letra en minúscula y la primera letra de cada palabra o parte de palabra subsecuente capitalizada.

- Un archivo fuente contiene una única clase.
- Las llaves de comienzo y fin ({ }) siempre se colocan en una nueva línea.
- Siempre se usan las llaves de comienzo y fin en sentencias condicionales.
- Se usa el margen adicional de 4 espacios.
- Las variables se declaran independientes en líneas nuevas, no en las mismas sentencias.
- Las declaraciones de uso (*using*) de espacios de nombre se colocan en la cabeza del archivo.
- Las implementaciones de las clases tendrán el siguiente orden:
    1 Atributos de la clase.
    2 Constructores y Finalizadores.
    3 Propiedades.
    4 Métodos.
- Las declaraciones de atributos tendrán el siguiente orden según sus modificadores de acceso:
    1 Públicos.
    2 Protegidos.
    3 Internos.
    4 Privados.
- Los atributos relacionados se declaran en una misma línea. El resto se declara en líneas separadas.
- Los comentarios están declarados en el mismo idioma, gramaticalmente correctos y contienen los signos de puntuación apropiados.

Para los comentarios se usa // o /// pero no /*…*/.

3.2 Implementación del proceso de visualización de simulaciones

En esta sección se abordarán los detalles de implementación del proceso de visualización propuesto en el Capítulo 2.

3.2.1 Preprocesamiento del historial

La biblioteca EDET.DLL se encarga de extraer los datos almacenados en el fichero de historial en formato XML y almacenarlos en una estructura de datos D*R-Tree para poder realizar las consultas de forma eficiente. Cuando se crea la estructura esta es almacenada en disco en el mismo directorio donde se encontraba el fichero XML.

Almacenar la estructura en disco permite no tener que repetir el preprocesamiento cuando se vaya a visualizar nuevamente el mismo fichero de historial. Cuando se cuenta con una referencia a la estructura de datos espacio temporal se procede a visualizar la simulación.

3.2.2 Selección y actualización de los objetos a visualizar

La simulación está compuesta por millones de objetos de los cuales solo una minoría será visualizada, en correspondencia al campo de visión de la cámara y el instante de tiempo que se esté visualizando. En consecuencia, no es necesario reproducir el comportamiento de todos los objetos con el fin de visualizar solo unos pocos. Para solucionar la situación anterior se utiliza la técnica *frustum culling* la cual es explicada a continuación.

Frustum culling

El *frustum* es una figura geométrica en forma de pirámide, como se muestra en la Figura 6. El *frustum culling* o recorte de *frustum* pretende solo cargar los objetos necesarios para la visualización. Unity implementa esta técnica automáticamente, que se encarga de dibujar solo los objetos visibles. Esta tiene como desventaja que los objetos que no se encuentran en el *frustum* continúan cargados en la memoria del sistema y ejecutándose los comportamientos asociados a los objetos. Para enfrentar las desventajas anteriores se decidió implementar otra variante de la técnica, la cual cargaría solo los objetos necesarios con el fin de visualizar un pequeño intervalo de tiempo. Para cargar los objetos a representar en la escena es necesario saber cuáles son los que se encuentran dentro del *frustum* en un intervalo de tiempo, para lo cual se utiliza la biblioteca EDET.DLL que implementa la estructura de datos D*R-Tree. Esta última permite recuperar la información necesaria de forma eficiente. A dicha estructura se le solicita la información pasándole por parámetro el *frustum* y el intervalo de tiempo a visualizar, este último comienza en el instante en que se realiza la consulta y tiene una duración de 30 segundos.

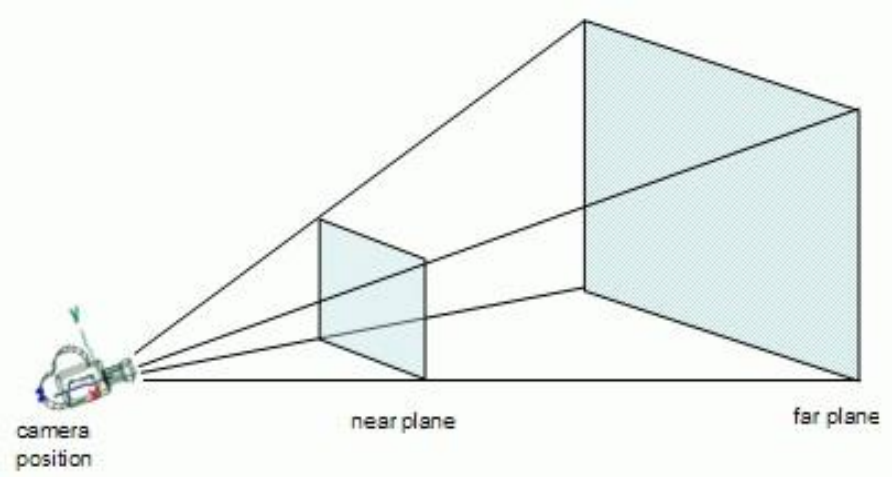


Figura 6 Prisma que representa el *frustum* de una cámara

Cálculo del *Frustum*

En la implementación de la técnica de *frustum culling* es necesario calcular el mismo. Para explicar el cálculo del *frustum* se representó con triángulos en dos dimensiones, a diferencia del utilizado en la aplicación (Figura 6). El *frustum* de la cámara utilizada, es representado por el triángulo ABC, este fue reducido al introducir la niebla, dando como resultado que el espacio que contiene los objetos visibles es el área que visualiza la cámara. Para aumentar la eficiencia en los movimientos de la cámara se cargan objetos que potencialmente formarán parte de la escena. Esto evita que al mover la cámara ocurran pausas o apariciones de objetos de manera no deseada.

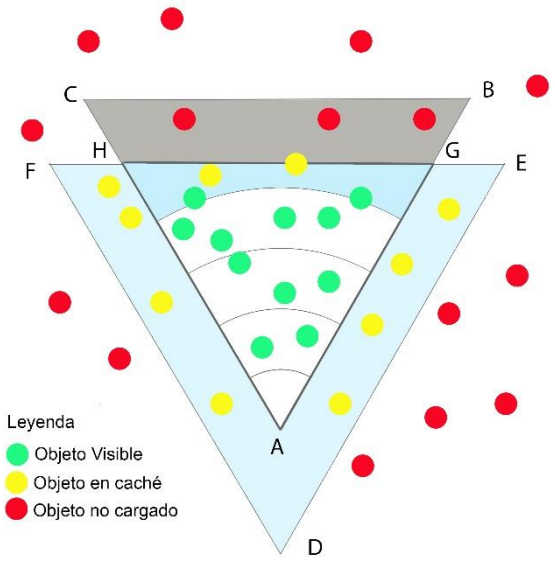


Figura 7 Estado de los objetos

Cálculo del tiempo

El tiempo utilizado en la consulta a la estructura de datos espacio temporal es manejado por el script *TimeCenter* del *GameObject* del mismo nombre, de este existe una sola instancia por el uso del patrón *Singleton*. La clase Time de Unity proporciona la variable *deltaTime* que indica el tiempo que tardo el ultimo fotograma en ser representado. La variable *deltaTime* es usada para incrementar el tiempo controlado por el script *TimeCenter*.

Actualización de la escena

Para la selección y actualización de los objetos a visualizar se utiliza el algoritmo que se describe en el siguiente pseudocódigo. En este algoritmo es aplicada la técnica *frustum culling*.

```
1   UpdateScene(frustum, startTime, endTime)
2       simulationObjects = EDET.ObjectInRange(startTime, endTime, frustum)
3       keysToRemove = elements.Keys.Except(simulationObjects)
4       for each (key in keysToRemove)
5                   Destroy(elements[key])
6                   elements.Remove(key)
7       for each (object in simulationObjects)
8           if elements.ContainsKey(object.Name)
9               UpdateEvents(object.Name, object.Events)
10          else
11              sceneObject=CreateGameObjectForType(object)
12              elements.Add(object.Name, sceneObject)
```

Figura 8 Pseudocódigo del método UpdateScene

En la línea 2 se obtienen los objetos contenidos dentro de *frustum* en el intervalo de tiempo solicitado. En la línea 3 se obtienen las llaves de los objetos que se están visualizado y no es necesario representar, seguidamente en las líneas 4, 5 y 6 son eliminados. De la línea 7 a la 12 se actualizan los objetos que ya se encontraban con nuevos eventos y se incorporan con sus respectivos eventos los que no se encontraban en la escena. En la línea 11 se hace uso del método CreateGameObjectForType, el cual corresponde a la fase Modelación 3D.

3.2.3 Modelación 3D

En esta fase se especifican los modelos 3D que van a representar a cada uno de los objetos mediante distintos niveles de detalle. La entidad que Unity provee con este fin es el *GameObject*. Este cuenta con varios componentes y propiedades que permiten la visualización de los objetos tal es el caso del mallado, texturas, materiales y *scripts*

orientados a comportamientos (Unity Technologies, 2015). Se configuraron los *GameObject* para los objetos presentes en las simulaciones de RCMS y posteriormente fueron almacenados en *Prefab* o Prefabricados.

3.2.4 Visualización

En esta fase se realiza la visualización de los objetos modelados anteriormente. Unity es capaz de representar los GameObject que cuenten con los compones y propiedades adecuadas. En la fase de modelación se instanciaron los GameObject correctamente configurados para la visualización a partir de los *Prefab*. Los mismos tienen una lista con los movimientos que realizan. Para lograr una visualización fluida se realiza una reproducción de los movimientos de los objetos.

Reproducción de los movimientos de los objetos

Para mostrar cómo se realiza la reproducción de los movimientos fue seleccionada la traslación. Este es uno de los movimientos más utilizados, el mismo fue implementado de manera que actualizara la posición del objeto antes de mostrarlo en cada fotograma. El movimiento de traslación cuenta con varios datos entre los que se encuentra la posición inicial, la posición final, el tiempo inicial (ts), el tiempo final (te), longitud y la velocidad. El cálculo de la posición en un instante de tiempo entre ts y tf se realiza mediante el método de interpolación linear entre dos vectores propuesto en el manual de Unity (Unity Technologies, 2015).

La visualización de los objetos seleccionados, modelados y ubicados anteriormente es costosa computacionalmente. La fluidez de la visualización puede verse comprometida por el alto número de polígonos contenidos en los modelos. Para solucionar este problema son utilizadas técnicas de optimización como los niveles de detalles y el *backface culling*.

Selección del nivel de detalle para la representación.

El nivel de detalle de los objetos en escena es seleccionado de acuerdo a la distancia entre este y el observador. Los objetos cercanos a la cámara tendrán un mayor nivel de detalle con respecto a aquellos que se encuentran más alejados. Se decidió utilizar tres niveles de detalle como se muestra en la Figura 9. Los cambios de los niveles de detalle fueron ocultados con el uso de la niebla. En la figura también se representa la niebla, es representada con el color gris. La unidad de medida u en las escenas de Unity es equivalente a un metro en la realidad.

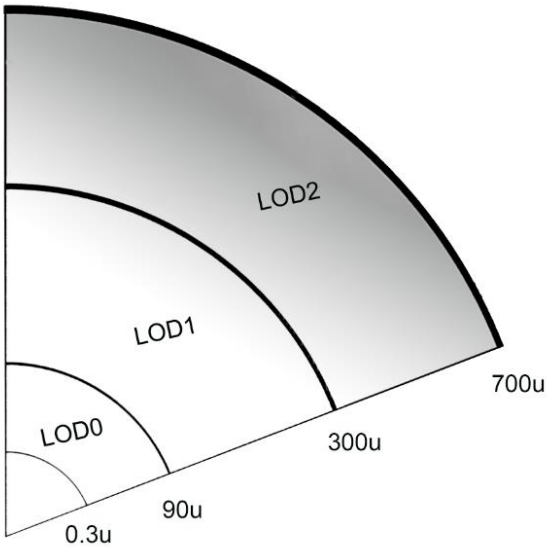


Figura 9 Niveles de detalle según la distancia

Para la selección del nivel de detalle fue añadido un *script* en forma componente a cada *GameObject*. Dicho componente puede ser configurado desde el editor de Unity, entre las opciones de configuración se encuentran adición de los niveles de detalle y especificar la distancia a la que se observara cada uno. El siguiente pseudocódigo describe el algoritmo para la selección del nivel de detalle.

```
1   CalculateLOD()
2       distance = Distance(camera.position, object.position)
3       level = cantLOD - 1
4       selected = false
5       for i=0 to cantLOD - 1
6           if distance < distances[i]
7               level = i
8               selected = true
9           if selected
10              break
11      if level != activeLevel
12          ChangeActiveLevel(level)
```

Figura 10 Selección del nivel de detalle

El método *Distance* (línea 2) calcula la distancia cuadrada entre la cámara y el objeto tal y como se evidencia en la siguiente fórmula.

$$d = (x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2$$

Se utiliza la distancia cuadrada en vez de la distancia convencional para evitar el costo computacional que implica el cálculo de la raíz cuadrada. Seguidamente desde la línea 3 hasta la línea 10 se obtiene el nivel de detalle del objeto. En las líneas 11 y 12 se

cambia el LOD del objeto solo cuando este es distinto del LOD que se encuentra actualmente activo.

Backface culling

Otras técnicas de optimización que pueden ser utilizadas para reducir la cantidad de polígonos a representar es la técnica *backface culling*. La misma pretende no representar los polígonos que se encuentran en la cara de atrás del objeto. Esta técnica es implementada y activada en Unity por defecto.

Realismo en la visualización

Para lograr los objetivos de la visualización fueron aplicadas técnicas para lograr un grado de realismo aceptable en la visualización de simulaciones. A continuación, se explica cómo fueron incorporadas estas técnicas a la escena.

La **iluminación** fue añadida a la escena adicionando el componente *Light* a un *GameObject* vacío. La iluminación seleccionada fue la direccional porque aporta realidad a la escena con un bajo consumo de recursos computacionales. En la Figura 11 se muestra la configuración utilizada para la iluminación en la aplicación.

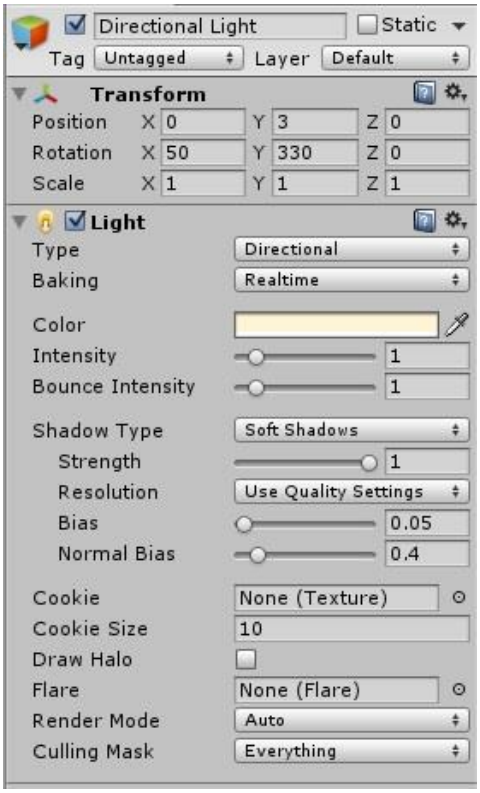


Figura 11 Configuración de la iluminación en la aplicación

Unity permite seleccionar el tipo **sombreado** con la propiedad *Shadow Type* en cada una de las luces en la escena. Una vez seleccionado el tipo de sombreado es posible configurarlo. Por otra parte cada *Mesh Render* debe tener habilitada las propiedades *Cast Shadows* y *Receive Shadows* de manera adecuada (Unity Technologies, 2015). En la aplicación se configuraron estos parámetros como se puede ver en la Figura 12 y en la Figura 13.

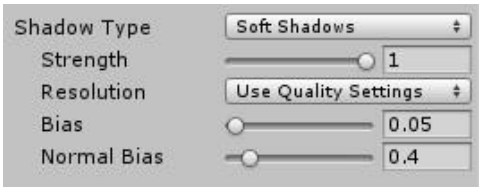


Figura 12 Configuración del sombreado en el componente Light

Figura 13 Configuración del componente Mesh Render de un tren

A cada objeto se le añadieron sus **texturas** correspondientes por medio del componente material, como puede verse en la Figura 14.

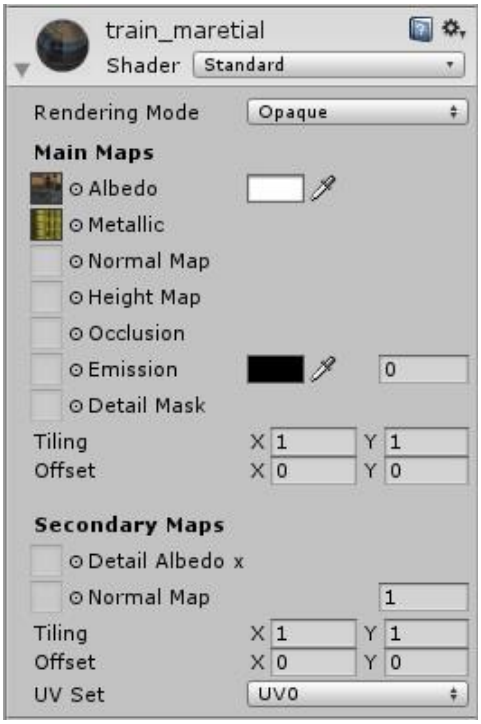


Figura 14 Configuración del componente material de un tren

3.3 Diagrama de Componentes

Según (Pressman, 2014) un componente es una parte modular, desplegable y reemplazable de un sistema que encapsula implementación y expone un conjunto de interfaces. Un diagrama de componentes modela cómo un sistema de software se divide en componentes y representa las dependencias entre ellos en la Figura 15.

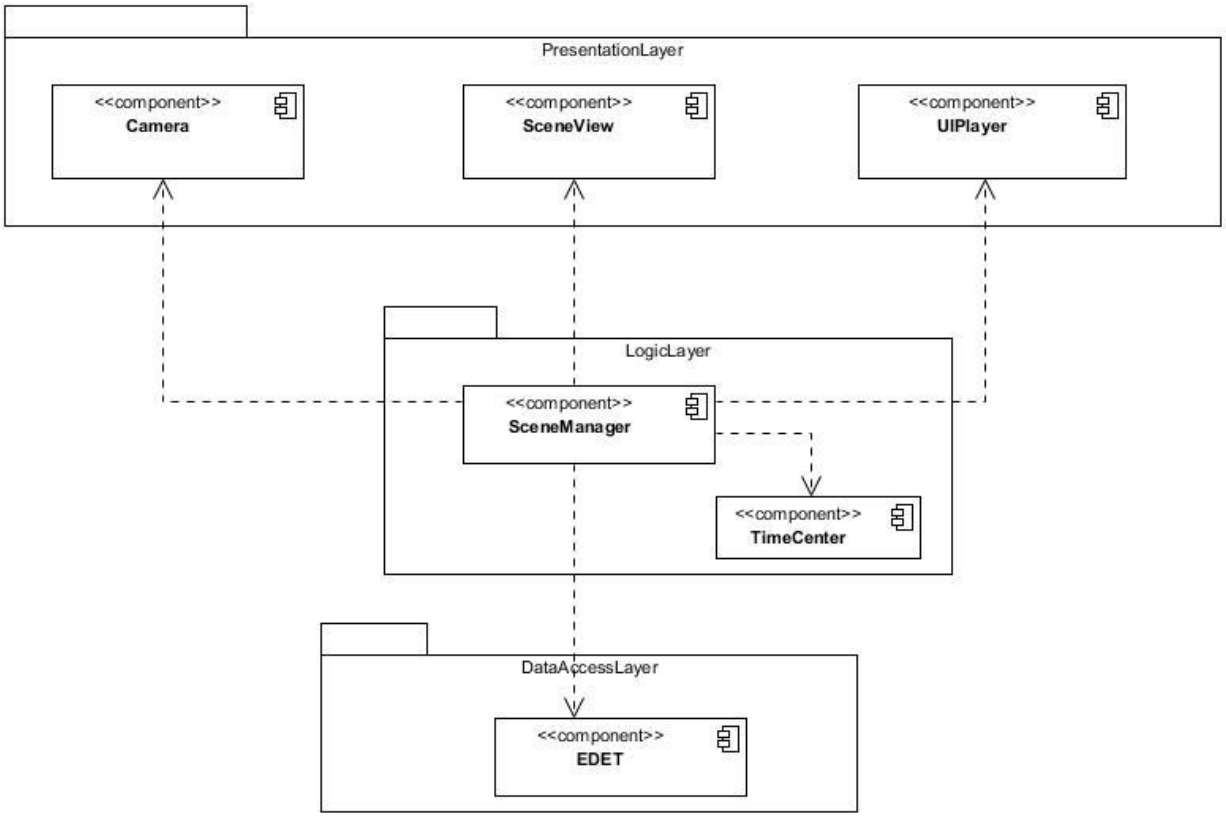


Figura 15 Diagrama de Componentes

UIPlayer: Contiene las interfaces de usuario que permiten la interacción del usuario con el sistema.

SceneView: Este es el componente principal para la representación gráfica. Contiene toda la información y algoritmos necesarios para la visualización.

SceneManager: Es el componente que se encarga de controlar los objetos presentes en el escenario. Así como solicitar los datos necesarios para la visualización al componente BDET.

TimeCenter: Es quien se encarga de controlar la variable tiempo de la simulación.

El componente *Camera* es quien maneja todo lo referente a la visualización, punto de visión, ángulo, perspectiva, entre otros.

EDET: Contiene toda la información de una simulación y algoritmos para realizar consulta a esta. Este componente es externo a la solución pues no fue implementado durante la misma.

3.4 Pruebas de software

Las pruebas del software son un elemento crítico para la garantía de la calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación. El objetivo de las mismas es encontrar el máximo número posible de errores con una cantidad manejable de esfuerzo aplicado en un período realista de tiempo (Pressman, 2014). Con las mismas se garantiza que el producto final funcione como fue diseñado e implemente de manera correcta los requerimientos identificados.

Las pruebas son aplicadas para diferentes tipos de objetivos, en diferentes escenarios o niveles de trabajo. Los niveles de pruebas más comunes son:

Nivel de Unidad

- Enfocada al código fuente de los componentes.
- Se utiliza para verificar todos los flujos de control.
- Primero pasa por la revisión del programador.

Nivel de Integración

- Prueba los componentes combinados para ejecutar un CU.
- Se utiliza para verificar, descubrir errores o que estén incompletas las especificaciones de las interfaces de las clases.

Nivel de Sistema

- Prueba el software funcionando como un todo.

Nivel de Aceptación

- Prueba final antes del despliegue del sistema.
- Generalmente lo realizan los usuarios finales.

Para validar la solución, de acuerdo a las características de los niveles de pruebas presentados anteriormente y a las características del sistema desarrollado se seleccionó el Nivel de Código y el Nivel de Sistema.

Pruebas a Nivel de Unidad

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software. En el caso de un contexto orientado a objetos el concepto de unidad cambia.

La menor unidad que se prueba es la clase (Pressman, 2014). La prueba de las clases es dirigida por las operaciones que esta encapsula y el estado de comportamiento de las mismas.

Para estas pruebas se utilizó el método de Caja Blanca, la técnica de Camino Básico y la herramienta *UnityTestTools*. El método de Caja Blanca es una filosofía de diseño de casos de prueba que utiliza la estructura de control descrita para derivar casos de prueba que:

- Garanticen que todos los caminos independientes dentro de un método han sido ejercitados al menos una vez.
- Ejerciten todas las decisiones lógicas en sus lados verdaderos y falsos.
- Ejecuten todos los ciclos en sus límites y dentro de sus límites operacionales.
- Ejerciten las estructuras de datos internas para verificar su validez.

La técnica de Camino Básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y utilizar esta medida como una guía para la definición de un conjunto base de rutas de ejecución. De esta forma se garantiza que los casos de pruebas derivados ejecuten cada sentencia al menos una vez durante la prueba.

Unity Test Tools es un paquete que proporciona a los desarrolladores los componentes necesarios para la creación y ejecución de pruebas automatizadas. Unity Test Tools incluye el *framework NUnit* en el editor lo que permite ejecutar pruebas unitarias desde el interior de Unity. Esto significa que pueden crear instancias de *GameObjects* y operar con ellos lo que no sería posible fuera de Unity.

La aplicación fue dividida en unidades para realizar las pruebas unitarias. El método MoreSpeed de la clase Controllers fue escogido para mostrar el uso de la técnica del camino básico durante la etapa de pruebas unitarias. Este método aumenta la escala en la que avanza el tiempo, a mayor escala aumenta la velocidad de reproducción. En la Figura 16 se muestran la asignación de los nodos del grafo de flujo a las sentencias.

```
1   public void MoreSpeed()
2   {
3       if (!pause) ----------------------------> 1
4       {
5           if (FactorSpeed + 2 > 24) ------------> 2
6           {
7               FactorSpeed = 24; ----------------> 3
8           }
9           else
10          {
11              FactorSpeed += 2; ----------------> 4
12          }
13      }
14  }
```

Figura 16 Código del método MoreSpeed

La Figura 17 muestra el grafo de flujo correspondiente al método MoreSpeed mencionado anteriormente. La cantidad de juegos de datos necesarios para ejecutar todas las sentencias es al menos la complejidad ciclomática del grafo del flujo. El valor de la complejidad ciclomática $V(G)$ del grafo de flujo G se obtiene mediante la ecuación:

$$V(G) = E - N + 2$$

donde E y N definen la cantidad de aristas y la cantidad de nodos respectivamente de G.

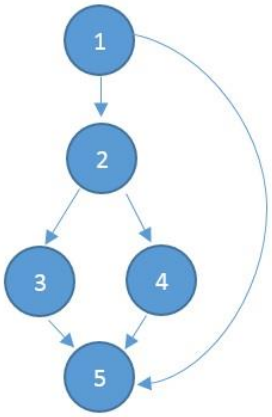


Figura 17 Grafo de flujo del método MoreSpeed (Elaboración propia)

Sustituyendo los valores de E y N se obtiene:

$$V(G) = 6 - 5 + 2 = 3$$

En la Tabla 7 se observan los caminos del grafo de flujo obtenidos.

Tabla 7 Caminos del grafo de flujo

No	Camino
1 2 3	1-5 1-2-3-5 1-2-4-5

Ejecución de las pruebas unitarias

Para cada una de las unidades de la aplicación se realizó este proceso. Luego para cada uno de los caminos obtenidos fue derivado un caso de prueba. Luego de derivar los casos de prueba fueron preparados y aplicados en la herramienta Unity Test Tools. En la Figura 18 son mostrados los resultados de las pruebas unitarias aplicadas a las unidades de la aplicación.

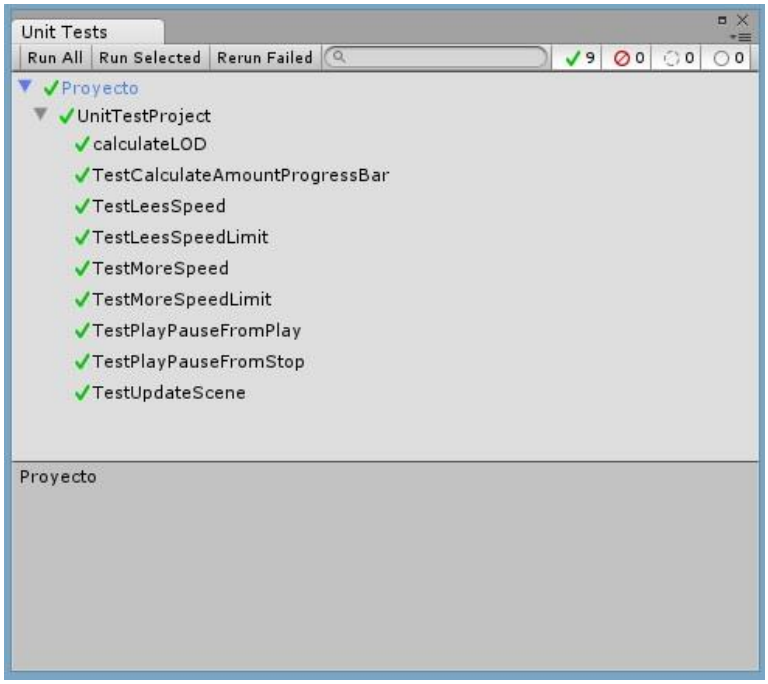


Figura 18 Resultados de las pruebas unitarias

Durante la realización de la primera iteración se detectaron tres no conformidades. Al corregirlas se realizó la segunda iteración sin detección de errores.

Pruebas a Nivel de Sistema

Las Pruebas a Nivel de Sistema están constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Cada prueba tiene un propósito diferente trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.

Los métodos de prueba del software tienen el objetivo de diseñar pruebas que descubran diferentes tipos de errores con menor tiempo y esfuerzo. Para la realización de las pruebas en la aplicación se utilizará el método de caja negra.

Pruebas de caja negra

Las pruebas de caja negra, también denominadas pruebas funcionales, se concentran en los requisitos funcionales del software. Permiten derivar conjuntos de condiciones de entrada que ejercitaran por completo todos los requisitos funcionales de un programa. Las pruebas de caja negra tratan de encontrar errores en las siguientes categorías: funciones incorrectas o faltantes, errores en estructuras de datos externas, errores de interfaz, errores de comportamiento o desempeño, errores de inicialización y termino (Pressman, 2014).

La técnica aplicada para la realización de las pruebas al Visualizador es partición de equivalencia, la cual, divide el dominio de entrada en clases de datos a partir de los cuales se pueden derivar casos de prueba. La partición de equivalencia se esfuerza por definir un caso de prueba que descubra ciertas clases de errores, reduciendo así el número de casos de prueba que deben desarrollarse (Pressman, 2014).

Diseño de las pruebas de caja negra

Caso de Prueba del Caso de Uso "Manipular escena"

Descripción: El caso de uso se inicia cuando el usuario utiliza el ratón para controlar la cámara. Finaliza cuando el sistema responde acorde a la acción realizada por el usuario.

Condiciones de ejecución: Se está visualizado una simulación.

Tabla 8 Escenarios del Caso de Prueba "Manipular escena"

Escenario	Descripción	Respuesta del Sistema	Flujo Central
EC 1.1	El caso de uso comienza cuando el usuario realiza alguna de las siguientes acciones: • Presiona el botón izquierdo del ratón y lo	El sistema realiza una operación según la opción seleccionada por el usuario: • Si se presiona el botón izquierdo del ratón y lo mueve	Se realiza la manipulación de la escena visualizada, en dependencia de las diferentes acciones

	mueve sobre el área visualizada. • Gira la rueda del ratón hacia atrás. • Gira la rueda del ratón hacia delante. • Presiona el botón derecho del ratón y lo mueve sobre el área visualizada.	sobre el área visualizada entonces la cámara gira en el mismo sentido y dirección que el movimiento del ratón. • Si gira la rueda del ratón hacia atrás entonces la escena se aleja. • Si gira la rueda del ratón hacia delante entonces la escena se acerca. • Si presiona el botón derecho del ratón entonces la escena se mueve en el mismo sentido y dirección que el movimiento del ratón.	llevadas a cabo por el usuario. El sistema realiza una operación según la opción seleccionada por el usuario.

Caso de Prueba del Caso de Uso "Manipular la reproducción de la simulación"

Descripción: El caso de uso se inicia cuando el usuario selecciona cambiar el estado de la reproducción. Finaliza cuando el estado de la reproducción se cambia.

Condiciones de ejecución: Se está visualizado una simulación.

Escenario	Descripción	Respuesta del Sistema	Flujo Central
EC 1.1	El caso de uso comienza cuando el usuario realiza alguna de las siguientes acciones:	El sistema realiza una operación según la opción seleccionada por el usuario:	Se realiza la manipulación de la escena visualizada, en dependencia de las diferentes

	• Presiona el botón izquierdo del ratón sobre el icono de *play*. • Presiona el botón izquierdo del ratón sobre el icono de pause. • Presiona el botón izquierdo del ratón sobre el icono de aumentar velocidad. • Presiona el botón izquierdo del ratón sobre el icono de disminuir velocidad.	• Si presiona el botón izquierdo del ratón sobre el icono de *play-pause* cuando la reproducción estaba en pause la reproducción continua a una velocidad igual a la que tenía antes de pasar al estado de pause. • Si presiona el botón izquierdo del ratón sobre el icono de *play-pause* cuando la reproducción estaba en *play* se detiene la reproducción de la simulación. • Si presiona el botón izquierdo del ratón sobre el icono de aumentar velocidad cuando la reproducción se encuentra en *play* aumenta la velocidad de reproducción. • Si presiona el botón izquierdo del ratón sobre el icono de disminuir cuando la reproducción se encuentra en *play*	acciones llevadas a cabo por el usuario. El sistema realiza una operación según la opción seleccionada por el usuario.

		velocidad disminuye la velocidad de reproducción.	

Ejecución de las pruebas.

Se realizaron dos iteraciones de pruebas funcionales aplicándose los dos casos de prueba diseñados y se detectaron tres no conformidades en la primera iteración, todas fueron de código. En la revisión realizada en la segunda iteración se verificó que las no conformidades de la etapa anterior fueron resueltas y no se detectaron nuevas.

3.4.1 Pruebas de rendimiento

A la aplicación se le realizaron pruebas de rendimiento, debido a que para sistemas en tiempo real es inaceptable el software que proporciona las funciones requeridas, pero no se ajusta a los requisitos de rendimiento (Pressman, 2014). Esta prueba está diseñada para probar el cumplimiento de dichos requisitos en tiempo de ejecución dentro del contexto de un sistema integrado.

Para llevar a cabo las pruebas de rendimiento al sistema implementado se le realizó un recorrido por el entorno descrito en el fichero de historial de simulación CCXmove, el cual tiene un total de 1200 objetos móviles. El recorrido se llevó a cabo en varios intervalos de tiempo de 2 minutos cada uno. Se definieron las variables “Promedio de Objetos en la Escena Visualizada”, “Cantidad Promedio de Polígonos Representados”, “Cantidad Promedio de Vértices Representados” y “Promedio de Fotogramas por Segundo”. Las pruebas fueron realizadas en un ordenador con las siguientes prestaciones:

- Procesador: DualCore Inter Core i3, 3.30 GHz.
- Memoria RAM: 8GB DDR3.
- Tarjeta Gráfica: NVIDIA GeForce GTX 750 Ti

Se ejecutó el mismo recorrido seis veces en intervalos de dos minutos no solapados, los resultados de los mismos se muestran en la Tabla 9.

Tabla 9 Datos y resultados de las pruebas de rendimiento aplicadas al software.

Intervalo de tiempo visualizado	Objetos cargados por la aplicación	Cantidad de polígonos representados	Cantidad de vértices	Fotogramas por segundo
1-3	49	189.9k	256.6k	68
6-8	100	344.6k	456.1k	63
10-12	150	750.8k	901.2k	59
13-15	126	530.5k	676.5k	60
20-22	160	731.9k	920.1k	57
29-31	192	910.4k	1.2M	55

Los resultados obtenidos muestran un desempeño promedio de 60.33 fotogramas por segundo. Teniendo en cuenta que 30 fotogramas por segundo es el mínimo necesario (Microsoft, 2003) para una visualización en tiempo real, se puede afirmar que aplicando las técnicas y algoritmos de optimización se logra una visualización eficiente de simulaciones de eventos discretos.

3.4.2 Conclusiones del Capitulo

Con la aplicación de los estándares de C# para .NET se logró una implementación legible y sostenible. El diagrama de componentes sirvió de apoyo para representar cómo la solución está dividida en componentes y cómo se relacionan entre ellos.

Finalmente se realizó una validación del componente basándose en los resultados de las pruebas de software, demostrando la validez de la hipótesis de investigación planteada en la introducción de este trabajo.

Conclusiones

Después de haber realizado la investigación y llevar a cabo el desarrollo del software, se ha obtenido un visor de simulaciones de eventos discretos. Las tecnologías y herramientas seleccionadas para el desarrollo del software ofrecieron el soporte necesario para dar cumplimiento a los requisitos funcionales y no funcionales del mismo. La aplicación de patrones en el diseño de la solución propició resolver problemas de forma eficiente. El estudio y uso de las técnicas de optimización permitió lograr una visualización fluida para el usuario con un desempeño por encima de 30 fotogramas por segundo. Se mejoró la calidad gráfica de la visualización aplicando técnicas para aumentar el realismo.

Se obtuvo la documentación técnica del proceso de desarrollo de software, la cual permite un mejor entendimiento del visualizador desarrollado, y puede ser utilizada como guía para futuras actualizaciones o para continuar el desarrollo del sistema, llevando a cabo la implementación de nuevas funcionalidades.

Recomendaciones

Permitir al usuario seleccionar el modelo 3D que será visualizado para un tipo de objeto debido a que esto se predefine durante el desarrollo de la aplicación.

Configurar automáticamente la calidad visual teniendo en cuenta las prestaciones con que cuenta el ordenador donde se va a ejecutar la aplicación.

Utilizar estructuras de datos espaciales para la gestión de los objetos estáticos.

Referencias Bibliográficas.

AKENINE-MÖLLER, T., HAINES, E. y HOFFMAN, N., 2008. *Real-time rendering*. S.l.: CRC Press.

ALONSO RODRÍGUEZ, B., 2012. Integración del Catastro 3D en una plataforma de simulación 3D.

ANYLOGIC COMPANY, 2015. Why AnyLogic? — AnyLogic Simulation Software. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: http://www.anylogic.com/features.

AVISON, D.. y FITZGERALD, G., 1995. *Information Systems Development: Methodologies, Techniques, and Tools. New York*. New York: McGraw-Hill.

BANKS, J., 1999. Introduction to simulation. *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*. S.l.: ACM, pp. 7–13.

BIJL, J.L. y BOER, C.A., 2011. Advanced 3D visualization for simulation using game technology. *Proceedings of the Winter Simulation Conference*. S.l.: Winter Simulation Conference, pp. 2815–2826.

BIRN, J., 2002. 3dRender.com Glossary. [en línea]. [Consulta: 21 junio 2016]. Disponible en: http://www.3drender.com/glossary/index.htm.

CRAIN, R.C., 1997. Simulation using GPSS/H. *Proceedings of the 29th conference on Winter simulation*. S.l.: IEEE Computer Society, pp. 567-573. ISBN 0-7803-4278-X.

DEBRAUWER, L., 2012. *Patrones de diseño para C#: Los 23 modelos de diseño: descripción y soluciones ilustradas en UML 2 y C#* [en línea]. S.l.: Ed. ENI. Expert IT. ISBN 978-2-7460-7260-2. Disponible en: https://books.google.com.co/books?id=B82QLbKSsHYC.

ECLIPSE FOUNDATION, 2013. OpenUP. [en línea]. [Consulta: 3 abril 2016]. Disponible en: http://epf.eclipse.org/wikis/openup/.

EPIC GAMES, 2015. Unreal Engine 3 Features. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: https://www.unrealengine.com/previous-versions.

FIGUEROA, R.G., SOLÍS, C.J. y CABRERA, A.A., 2008. Metodologías tradicionales vs. Metodologías ágiles. *Universidad Técnica Particular de Loja, Escuela de Ciencias en Computación. (En línea), Disponible en: http://adonisnet. files. wordpress. com/2008/06/articulo-metodologia-de-sw-formato. Doc*.

FLEXSIM SOFTWARE PRODUCTS, INC, 2015. Simulation software for manufacturing, material handling, healthcare, etc. - FlexSim Simulation Software. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: https://www.flexsim.com/.

GAGLIARDI, E.O., DORZÁN, M.G., CARRASCO, F.D., GARCÍA SOSA, J.C. y GUTIÉRREZ RETAMAL, G.A., 2005. Métodos de Acceso Espacio-Temporal: nuevas propuestas basadas en métodos existentes. *VII Workshop de Investigadores en Ciencias de la Computación*. S.l.: s.n.

GAGLIARDI, E.O., DORZÁN, M.G. y GUTIÉRREZ RETAMAL, G.A., 2005. D* R-Tree: un método de acceso espacio-temporal. *XI Congreso Argentino de Ciencias de la Computación*. S.l.: s.n..

GILBERTO, G., 2003. Propuesta de un método de acceso espacio-temporal. *II WorkShop de Bases de datos*. S.l.: s.n.

GOLDSTONE, W., 2011. *Unity 3.x Game Development Essentials* [en línea]. S.l.: Packt Publishing. Community experience distilled. ISBN 978-1-84969-145-1. Disponible en: https://books.google.com.cu/books?id=RJ5fsGXbqXwC.

GREGORY, J., 2009. *Game engine architecture*. S.l.: CRC Press.

GUTTMAN, A., 1984. *R-trees: a dynamic index structure for spatial searching*. S.l.: ACM. 2.

HAN, S.H., AL-HUSSEIN, M., AL-JIBOURI, S. y YU, H., 2012. Automated post-simulation visualization of modular building production assembly line. *Automation in construction*, vol. 21, pp. 229–236.

HENRIKSEN, J.O., 1998. Stretching the boundaries of simulation software. *Simulation Conference Proceedings, 1998. Winter*. S.l.: IEEE, pp. 227-234. ISBN 0-7803-5133-9.

HENRIKSEN, J.O., 1999. General-purpose Concurrent and Post-processed Animation with PROOF TM. *Simulation Conference Proceedings, 1999 Winter*. S.l.: IEEE, pp. 176-181. ISBN 0-7803-5780-9.

HUNT, L., 2007. Coding Standards for .NET.

INCONTROL SIMULATION SOLUTIONS, 2015. Enterprise Dynamics Features. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: http://www.incontrolsim.com/en/enterprise-dynamics/enterprise-dynamics-features.html.

LARMAN, C. y VALLE, B.M., 2003. *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. S.l.: Pearson Educación. ISBN 978-84-205-3438-1.

LUEBKE, D.P., 2003. *Level of Detail for 3D Graphics* [en línea]. S.l.: Morgan Kaufmann Publishers. The Morgan Kaufmann series in computer graphics and geometric modeling. ISBN 978-1-55860-838-2. Disponible en: https://books.google.com.cu/books?id=M-gl4aoxQfIC.

MEHTA, D.P. y SAHNI, S., 2004. *Handbook of Data Structures and Applications* [en línea]. S.l.: CRC Press. Chapman & Hall/CRC Computer and Information Science

Series. ISBN 978-1-4200-3517-9. Disponible en: https://books.google.com.cu/books?id=fQVZy1zcpJkC.

MICROSOFT, 2003. Understanding Frames Per Second (FPS). [en línea]. [Consulta: 24 marzo 2016]. Disponible en: https://support.microsoft.com/en-us/kb/269068.

MICROSOFT, 2015. Novedades de Visual Studio 2015. [en línea]. [Consulta: 2 abril 2016]. Disponible en: https://msdn.microsoft.com/es-es/library/bb386063.aspx.

MICROSOFT, 2016a. Build Unity Games with Visual Studio. [en línea]. [Consulta: 2 abril 2016]. Disponible en: https://www.visualstudio.com/features/unitytools-vs.

MICROSOFT, 2016b. C#. [en línea]. [Consulta: 3 abril 2016]. Disponible en: https://msdn.microsoft.com/en-us/library/kx37x362.aspx.

MICROSOFT, 2016c. C# Language Specification. [en línea]. [Consulta: 3 abril 2016]. Disponible en: https://msdn.microsoft.com/en-us/library/ms228593.aspx.

NASCIMENTO, M.A., SILVA, J.R., THEODORIDIS, Y. y OTHERS, 1998. *Access structures for moving points*. S.l.: Citeseer.

OBJECT MANAGEMENT GROUP, 2005. What is UML | Unified Modeling Language. [en línea]. [Consulta: 3 abril 2016]. Disponible en: http://www.uml.org/what-is-uml.htm.

OGRE3D TEAM, 2015. OGRE – Open Source 3D Graphics Engine. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: http://www.ogre3d.org/.

OPEN BABEL, 2007. XYZ (format) - Open Babel. [en línea]. [Consulta: 22 mayo 2016]. Disponible en: http://openbabel.org/wiki/XYZ_(format).

PRESSMAN, R.S., 2014. *Software engineering: a practitioner's approach*. S.l.: Palgrave Macmillan.

ROCKWELL AUTOMATION, 2015a. Arena Simulation. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: https://www.arenasimulation.com/what-is-simulation.

ROCKWELL AUTOMATION, 2015b. Discrete Event Simulation Software | Manufacturing, Supply Chain & Healthcare Simulation Software | Arena Simulation. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: https://www.arenasimulation.com/.

ROHRER, M.W., 2000. Seeing is believing: the importance of visualization in manufacturing simulation. *Proceedings of the 32nd conference on Winter simulation*. S.l.: Society for Computer Simulation International, pp. 1211–1216.

SAMET, H., 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260. ISSN 0360-0300. DOI 10.1145/356924.356930.

SHANNON, R.E., 1975. *Systems simulation: the art and science*. S.l.: Prentice-Hall Englewood Cliffs, NJ.

SHIVA TECHNOLOGIES SAS, 2015. ShiVa 1.9.2 - Cross Platform Game Development. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: http://www.shivaengine.com/.

SIMIO LLC, 2015. Simio Simulation Software | Simio. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: http://www.simio.com/products/simulation.php.

TAYEB, J., ULUSOY, Ö. y WOLFSON, O., 1998. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, vol. 41, no. 3, pp. 185–200.

TSAI, Y., AI, C., WU, Y. y SIPLON, P., 2010. Simulating port logistics operations using 3D visualization technology. *Proceeding of the International Conference on Computing in Civil and Building Engineering*. S.l.: s.n.

UNITY TECHNOLOGIES, 2015a. Unity - Game Engine. [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: https://unity3d.com/.

UNITY TECHNOLOGIES, 2015b. Unity - Manual: Procesamiento de Lotes de Draw Calls(Draw Call Batching). [en línea]. [Consulta: 10 diciembre 2015]. Disponible en: http://docs.unity3d.com/es/current/Manual/DrawCallBatching.html.

UNITY TECHNOLOGIES, 2015c. Unity - Manual: Unity Manual. [en línea]. [Consulta: 31 mayo 2016]. Disponible en: http://docs.unity3d.com/Manual/index.html.

VALLE MARTÍNEZ, Y., 2009. *Modelación y visualización de superficies de terrenos en tres dimensiones*. S.l.: UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS.

VEGA, G.E., 2013. Algoritmo de recortes y de niveles de detalles para el incremento de la velocidad de visualización de modelos 3D en dispositivos de bajo coste. *3 c TIC: cuadernos de desarrollo aplicados a las TIC*, vol. 2, no. 4, pp. 2.

VEGA INFANTE, D. y FERNÁNDEZ BALBUENA, C., 2012. Algoritmo de Niveles de Detalles para la Visualización de Modelos 3D.

WELLS, D., 2013. Extreme Programming: A Gentle Introduction. [en línea]. [Consulta: 24 junio 2016]. Disponible en: http://www.extremeprogramming.org/.

WENZEL, S., BERNHARD, J. y JESSEN, U., 2003. Visualization for Modeling and Simulation: A Taxonomy of Visualization Techniques for Simulation in Production and Logistics. *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation* [en línea]. New Orleans, Louisiana: Winter Simulation Conference, pp. 729–736. ISBN 0-7803-8132-7. Disponible en: http://dl.acm.org/citation.cfm?id=1030818.1030915.

Glosario de términos

Assets: (Bloques de construcción), son utilizados por el editor de entornos de Unity 3D para estructurar y organizar los elementos que se utilizan en los videojuegos.

Fotograma: cuadro o imagen particular dentro de una sucesión de imágenes que componen una animación.

***Prefab*:** Un *Prefab* es un tipo de Asset que le permite almacenar un objeto GameObject completamente con *components* y propiedades. El prefab actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Cualquier edición hecha a un *prefab asset* será inmediatamente reflejado en todas las instancias producidas por él, pero, también se puede anular componentes y ajustes para cada instancia individualmente.

***Components*:** Los Components son las tuercas y tornillos de los objetos. Son las piezas funcionales de cada GameObject.

Transform: El *Transform* es usado para almacenar la posición, rotación, escala y el estado.

GameObject: cada objeto en el juego o aplicación es un GameObject, sin embargo, estos no desarrollan ninguna función por sí solos. Los *GameObjects* necesitan propiedades especiales que definan su comportamiento, de esta manera se pueden convertir en un personaje, un entorno o un efecto especial.

Un GameObject es un contenedor para muchos components distintos. Por defecto, todos los GameObjects automáticamente tienen el componente Transform. El componente Transform indica dónde está ubicado, cuál es su rotación y cuál es su escalado.

***MonoBehaviour*:** es la clase base de todos los *scripts* derivados de Unity3D. Usando C# cada clase se deriva automáticamente de *MonoBehaviour*. Aunque no es absoluto puede eliminar esta dependencia para hacer clases aisladas.

Object Management Group (OMG): es un grupo destinado originalmente a establecer normas para distribuir sistema orientado a objetos y ahora se centra en el modelado (programas, sistemas y procesos de negocio).

Mesh Filter: El *Mesh Filter* toma una malla desde sus *assets* y lo pasa al *Mesh Renderer* para la representación en la pantalla.

Mesh Render: El *Mesh Render* toma la geometría del *Mesh Filter* y lo representa en la posición del objeto definida en el componente *Trasform*.