Sistemas Informáticos
Curso 2004 /2005

RPG- 3D

Sergio Delgado Díez

Dirigido por:
Prof. Antonio Gavilanes Franco
Dpto. S.I.P.

Facultad de Informática
Universidad Complutense de Madrid

Agradecimientos

A Martita, por hacer el proyecto por mi.:D Bueno, y por prestarme el ordenador y la nevera!! Gracias Guapa!

A Mari, Rafa, Rubén, y en general al resto de los que me aguantáis a diario.

A Antonio, entre otras cosas, por esas indispensables revisiones ortográficas.

A Fernando,"Fer", por merecérselo, y por las "Fervecitas".

A Julia, la "*señora reprógrafa*" y a sus caramelos

A la ausencia de mis compañeros de proyecto, quienes me han permitido trabajar como si no existieran.

TABLA DE CONTENIDOS

RPG-3D

Resumen:

El proyecto consiste en el desarrollo del prototipo de un videojuego en tres dimensiones utilizando modelos y animaciones de juegos ya existentes, pero haciendo especial hincapié en la lógica del juego y la inteligencia artificial asociado al mismo. Se trata de un juego basado en casillas aunque gráficamente lo que el jugador percibe es un universo continuo en tres dimensiones, con terrenos a distintas alturas, creados mediante la técnica de HeightMap. Las búsquedas de caminos implicadas en el desarrollo del juego, el tratamiento de las colisiones, y gran parte de la lógica asociada al juego están basadas precisamente en dichas casillas. Cada ser vivo del sistema, incluyendo los avatares del jugador, basa su comportamiento en autómatas, que se han implementado mediante scripts externos a la aplicación. Los personajes del juego pueden conversar con los agentes no enemigos, siendo los diálogos dependientes también de scripts externos al código de la aplicación.

RPG -3D

Abstract:

The project involves the development of a videogame prototype. The game uses models and animations exported from commercial games, and focuses on game logic and its artificial intelligence. It is a tiled based game, even thought, graphically, the gamer sees a continued universe in three dimensions, with realistic terrains created with HeightMap techniques. Pathfinding, collision detection and most part of the game logic is based on tiles. Each living creature included in the system, even player avatars, base their behaviours on automats implemented with scripts external to the application. The game player characters will be able to speak to non-enemy agents and theirs dialogs depend on extern scripts too.

Autorización de uso del autor

Autorizo a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente al autor, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Sergio Delgado Díez

Introducción

¿Qué es un videojuego?

El termino videojuego hace referencia a una amplia gama de productos y servicios software dirigidos a públicos de muy diversa índole con grandes diferencias entre ellos, resulta por tanto un concepto difícil de definir.

Un videojuego es, ante todo, un sistema software, y por tanto su desarrollo es semejante al de un sistema software convencional. La finalidad de estos sistemas es entretener al usuario y la diversión suele ser la mejor estimación de la calidad del sistema. Tienen, además, un claro componente artístico y creativo que hace que su desarrollo sea más costoso que el de una aplicación común con aproximadamente la misma cantidad de código. La apariencia del producto puede llegar a ser más importante que la programación del mismo y puede ser la última responsable del éxito o fracaso comercial del título.

El uso de licencias comerciales de películas de éxito puede garantizar así mismo el éxito del juego, dando lugar a una compleja relación simbiótica entre el cine comercial de éxito y la industria del videojuego. Las licencias de desarrollo de videojuegos son, por norma general, extraordinariamente caras, pero producen suculentos beneficios. En algunos casos la producción de cine y del videojuego son paralelas para que las fechas de lanzamiento resulten coincidentes aumentando los beneficios de las dos producciones.

En general los videojuegos son aplicaciones de tiempo real en las que el computador administra el tiempo de que dispone para tomar las decisiones de inteligencia, para pintar la representación del mundo que verá el usuario y para procesar la entrada que éste le proporciona.

Como en todas las aplicaciones de tiempo real, el sistema deberá fraccionar el tiempo para que el usuario no perciba la carga del sistema, planificándolo según estimaciones de urgencia y dificultad de la tarea. Algunas de estas tareas pueden degradar el rendimiento, pero pueden ser necesarias a pesar de todo, resultando por tanto necesario partirlas en tareas más simples o realizarlas por pasos para disminuir su efecto y amortiguar la latencia que pudiesen producir en el resto del sistema.

Procesar la entrada del usuario es, en general, una tarea sencilla ya que basta con leer los buffers de entrada para obtener la información que el usuario proporciona al sistema. Mención especial cabe hacer a los sistemas basados en redes neuronales para procesar la entrada o los

denominados “juegos de combos” en los que la entrada puede requerir de una mayor planificación y tiempo de CPU debido a su gran complejidad o variedad.

La toma de decisiones de Inteligencia Artificial puede ser extraordinariamente fácil o difícil, dependiendo del género de juego que se esté desarrollando. En algunos géneros basta con producir comportamientos aleatorios y en otros resulta necesario recurrir a soluciones precalculadas para poder tomar las decisiones en “tiempo real”. En general, resulta difícil estimar el tiempo que será necesario utilizar para tomar determinada decisión, por tanto las tareas de Inteligencia Artificial suelen fraccionarse en el tiempo para que no sean degradantes en ocasiones y extremadamente ligeras en otros.

El muestreo gráfico es, en general, la labor que más tiempo de CPU gasta y resulta difícil de fraccionar, por tanto es común la utilización de técnicas más o menos complejas para pintar en pantalla lo menos posible y seguir manteniendo una ilusión de realidad y solidez. El sistema debe tratar de pintar la mayor cantidad de veces por segundo para dar ilusión de continuidad en el movimiento y suavidad en las animaciones. Ninguna parte de la aplicación debe depender de la tasa de refresco de pantalla, y el resto del sistema deberá funcionar como si dicha tasa no existiese ya que no es una medida fiable del paso del tiempo puesto que depende de la cantidad de polígonos que estemos renderizando, así como de su tamaño o de cálculos más complejos como es el caso de avanzadas técnicas de sombreado. Si la tasa de refresco cae por debajo de los 25 refrescos por segundo el usuario percibirá saltos y parpadeos que harán su experiencia de juego de menor calidad, lo que influirá notablemente en el grado de disfrute que la experiencia de juego le aporte.

Antecedentes históricos

La idea de producir software de entretenimiento tiene casi cincuenta años, por lo que es más antigua de lo que generalmente se cree; si bien es cierto que hasta hace poco más de treinta años ninguna compañía desarrollaba estos productos para producir dinero. Los primeros videojuegos son poco más que pruebas de la potencia de cálculo de las computadoras y no se lanzan al público, siendo los propios desarrolladores del juego y sus compañeros de trabajo los únicos en disfrutarlos.

El primer videojuego conocido es el famoso ‘Pong’, o más correctamente, ‘Tennis for two’, en el que, mediante un osciloscopio y dos potenciómetros, se simulaba una partida de tenis entre dos jugadores sin ninguna intervención de la computadora salvo la simulación física y

el renderizado. 'Pong' fue desarrollado íntegramente por William Higinbotham en el año 1958 en el Brookhaven National Laboratory con la intención de hacer mas entretenidas las visitas del público a su centro de trabajo. Pocos meses después de la muestra de 'Pong', las colas para visitar el laboratorio duraban horas.

Mucho más conocido que 'Pong' es 'Spacewar', desarrollado por Steve Russell en el Instituto Técnico de Massachussets (MIT) en su época de estudiante. En esta ocasión se enfrentan dos naves espaciales en el espacio y también es necesaria la participación de dos jugadores. 'Spacewar' será la fuente de inspiración, años después, para que Nolan Bushnel manufacturase la primera máquina recreativa de la historia en 1972.

Desde este momento el crecimiento de la recién nacida industria es rapidísimo y llega a ser una de las más lucrativas del mundo.

Y las máquinas de marcianitos invaden el mundo

En 1978 Toshihiro Nishikado diseña y programa Space Invaders. Su éxito extiende las maquinas fuera de los salones recreativos. En 1979 Atari lanza Asteroids. Por fin spacewar se convierte en un éxito. Es el primer juego donde se guardan las mejores puntuaciones. En 1980 Moru Iwatani diseña para Nazco el Pac-man y vuelve a dejar a los japoneses sin monedas. A día de hoy, siguen siendo el país en que mas horas gastan al día en videojuegos.

Nacen las consolas

Las consolas llevan los salones recreativos a los hogares, la primera consola es la Channel F que comercializa Fairchild Camera and Instrument en 1976. La consola es programable porque permite cambiar de juego con unos cartuchos, lo mismo que la Studio II de RCA que también se lanza en 1976.

La primera consola de verdad es la VCS (Video Computer System) que Atari lanza al mercado en las navidades de 1977. La VCS no termina de triunfar hasta que en 1980 sacan su versión de Space Invaders.

Los juegos por ordenador

Aparte de Spacewar, los primeros juegos por ordenador, desarrollados en los 70, son textuales. Estudiantes del MIT aficionados a Dungeons and Dragons desarrollan en 1977 Zork una aventura textual para el PDP-10. Con el advenimiento de las computadoras personales, desarrollan Z-Machine, una maquina virtual que interpreta las aventuras y en 1980 escriben un intérprete para el Apple II.

En 1976 Steve Jobs (en Atari desde 1974) y Steve Wozniak fundan Apple, que en 1977 lanza el Apple II.

En 1982 Commodore, fabricante de calculadoras, lanza el C-64, el ordenador personal más vendido de todos los tiempos.

En 1980 Ken y Roberta Williams fundan "On-Line System" que desarrolla su primer juego para Apple II. Cuando se mudan al monte se refundan como Sierra On-Line, que en 1983 produce el primer juego para IBM, el PCER.

La crisis de los 80

La mala calidad de los juegos, que se desarrollan demasiado rápido para cumplir los plazos de marketing, termina con el crecimiento de la industria dando lugar a una crisis en el sector.

En 1983 un convoy de catorce trailers entierran en Alamogordo millones de copias de las versiones de Pac-man y E.T. de Atari, hecho que paso de ser un rumor a una realidad al ser reconocido públicamente por Atari años después.

Las empresas de juegos no paran de perder dinero: Atari pierde 356 millones de dólares en 1983, Mattel 195 y Coleco 258 en 1984.

Este gran desastre hace que los inversores americanos se alejen de esta industria y dejen el camino abierto a los japoneses, quienes no dejarán pasar la oportunidad convirtiendo a Japón en el país productor y consumidor por excelencia.

Las consolas japonesas

En 1983 Nintendo comercializa su primera consola, la "Family Computer" o "Famicom" que se distribuye en USA a partir de 1985 con el nombre de "Nintendo entertainment System" (NES).

Aquí comienza la competencia con Sega y su "Sega Master System" que se prolonga hasta 1995 cuando Sony entra en el mercado con la consola Play Station.

En 1989 sale al mercado la GameBoy y se garantiza el éxito incluyendo de serie un Tetris, el juego de arcade concebido por el matemático ruso Alexey Pajitnov.

La era Nintendo.

Entre 1994 y 1996, Sega sigue una mala estrategia que le lleva a lanzar tres consolas en dos años: Sega-CD, Sega 32x y Sega Saturn. En 1994 Nintendo lanza la Super GameBoy que permite ejecutar los juegos de GameBoy en la super Nintendo. En 1995 nace la PlayStation. En 1996 se comercializa la Nintendo 64 con más de 1 millón y medio de ventas en tres meses.

Las últimas consolas

En 1998 Sega descontinua la Saturn y en 1999 lanza la DreamCast.
En 2000 Sony lanza la PlayStation 2.
En 2001 Nintendo lanza la GameCube.
En 2001 Microsoft lanza la XBox.
En 2001 Sega abandona el mercado del Hardware.

Poco a poco los videojuegos para ordenadores producen menos beneficios cada año; las consolas se han convertido en el mercado real para los videojuegos.

Géneros de videojuegos

En el transcurso de la historia de los juegos de ordenador y videojuegos, y más recientemente, también de los juegos a través de la Red, se han desarrollado diversos "géneros". Como en la literatura o en la cinematografía estos géneros presentan características específicas que permiten su clasificación. Al mismo tiempo se ha constatado la tendencia creciente a combinar elementos de diferentes géneros, para que el argumento y el "gameplay" sean más variados y presenten nuevos retos. Esta propensión ha llevado a tipificarlos como "mixed genre games" (juegos de género híbrido) y son juegos que casi no pueden clasificarse claramente como un género concreto.

Los nombres de los productos suelen tener descripciones como "arcade style shoot-em-up space combat game", "arcade combat jaunt with a smattering of detailed quest elements" y acrónimos como PBM RPG (Play-by-Mail Role Playing Game) o AD&D MPG (Advanced Dungeons and Dragons Multi-Player Game). En las revistas sobre juegos se adopta con frecuencia el término en inglés, porque se fomenta un argot atractivo para los destinatarios prototípicos. Además no es nada fácil encontrar equivalencias terminológicas adecuadas en la lengua de destino para la creación constante de neologismos en inglés. Una razón adicional que explica la frecuencia de los anglicismos es el hecho de que el inglés permite compuestos léxicos alineando simplemente palabras,

una tras otra, mientras que en otras lenguas esto sólo puede hacerse de forma limitada debido a las reglas morfosintácticas específicas de la lengua en cuestión. Una dificultad añadida para el traductor de entretenimiento interactivo, es que no existen reglas ortográficas específicas para la incorporación de anglicismos a la lengua de destino.

Para la clasificación de los juegos en categorías se han considerado las características específicas de los géneros, así como las particularidades históricas, porque éstas son de especial importancia para el trabajo de traducción. La siguiente categoría de los géneros se basa esencialmente en los juegos de ordenador y videojuegos presentados, analizados y criticados por Scholand (1996*:* 46-77*)*.

En la descripción de cada uno de los géneros se pueden constatar las diferencias fundamentales relacionadas con la temática principal, el argumento y la estructura del juego, así como también las habilidades lúdicas perseguidas. A su vez los textos contenidos en los juegos (texto en pantalla) y su documentación (manuales) varían notablemente dependiendo de cada género tratado. El traductor que desee trabajar en el ámbito de la localización de juegos, debe por tanto, estar familiarizado con los géneros más populares y sus características.

Dependiendo de la temática principal y del grupo a quien va dirigido, los textos se diferencian a nivel lingüístico, tanto estilístico como terminológico.

Juegos de aventura

1. Subgénero: Aventuras conversacionales
   Ejemplos
   - Classic
   - Adventure
   - Aventura Original, Cork

2. Subgénero: Aventuras Graficas
   Ejemplos
   - Mystery House
   - Adventure
   - KGB Indiana Jones
   - Monkey Island
   - Final Fantasy
   - Quest, Return to Zork
3. Subgénero: Juegos de rol
   Ejemplos
   - Final Fantasy
   - Baldur's Gate
   - Diablo
   - Fallout,

- Ultima,
- Might & Magic
- Wizardy

Características de los juegos de aventuras

- Presentan de forma más o menos trivial cambios, variaciones o ampliaciones de contenidos literarios.
- El contenido de las historias es el destino de un héroe o heroína (la figura con la que se identifica el jugador) en un mundo fantástico o lleno de secretos.
- Gran espectro: desde historias de aventuras y de detectives relacionadas con la actualidad y la realidad hasta mundos fantásticos e historia de un futuro lejano.
- Motivos, iconos, símbolos, imágenes, citas y esquemas argumentativos típicos provenientes de mitologías, parábolas, epopeyas, literatura fantástica y ficción científica.
- La estructura narrativa interna del juego de esta "novela interactiva" no se desarrolla de forma estrictamente lineal.
- En el marco de la matriz de información predeterminada por el programador del juego son posibles los nodos de cambio alternativos.
- La historia se interrumpe constantemente, y dependiendo de cómo el jugador se comporte, ésta se desarrolla en una u otra dirección.
- Las aventuras están sin excepción marcadas por el objetivo que se da a conocer al inicio.
- El jugador se acerca paso a paso al objetivo y a su solución mientras que va resolviendo pequeñas tareas y problemas. Éste debe encontrar, por ejemplo, obstáculos y emplearlos, sortear peligros, controlar al adversario y acertar adivinanzas. Al final de tan arduo, novelesco y aventurero viaje se llega a la liberación, la conquista o la solución.

Juegos de acción

Ejemplos

- Asteroids
- Total Carnage
- Wolfenstein
- Doom
- Mortal Kombat
- Virtua Fighter

Características de los juegos de acción

- Contenido abundante de acción y tensión.
- Requiere una rápida reacción, buena sincronización y concentración continua.

- En la interfaz icónica se reproduce siempre lo mismo: la defensa contra amenazas mediante una acción agresiva.
- Los adversarios de diferente índole (naves espaciales, robots, monstruos, etc.) se deben poner fuera de combate o aniquilarse mediante las armas disponibles.

Juegos de reacción y respuesta

Ejemplos

- Pac-Man
- Donkey Kong
- Mario Bros.
- Sonic

Características de los juegos de acción y respuesta

- La destreza sicomotriz y la velocidad de accionamiento del joystick o cualquier otro dispositivo de control es crucial para influir en el desenlace del juego y por tanto en el éxito de la partida.
- El jugador controla una o más figuras del juego que corren, saltan, sortean obstáculos, entre otras cosas.
- Los juegos están divididos en episodios o mundos que también están divididos en diferentes niveles.

Puzzles y juegos de lógica

Ejemplos

- Q*Bert
- Tetris
- Blockout
- 7th.Guest

Características de los juegos de lógica

- Además de la velocidad de reacción exigen sobre todo habilidad de combinación, una buena memoria, pensamiento lógico y sentido de la orientación.
- Dentro de los juegos de lógica también se incluyen los juegos de mesa adaptados electrónicamente (board games) como el ajedrez, Monopoly o los juegos de preguntas como Trivial Pursuit.

Simuladores

Ejemplos

- Simuladores de vuelo: MS Flight Simulator
- Simuladores de carreras
  - Nascar Racing
  - Formula I
  - GrandPrix
- Simuladores de economía y ecología

    - SimCity
    - SimLife
- Simuladores de deporte
    - Pong
    - FIFA
    - PC Football
    - NBA
    - Virtual Tennis
- Juegos de estrategia
    - Battle Isle 3
    - Civil War
    - Warhammer,
    - Age of Empires
    - Civilisation

Características de los simuladores

- En comparación con la mayoría de los otros géneros de juegos, éstos están más orientados a la realidad. Así diversos conocimientos y experiencias de diferentes ámbitos, por ejemplo, de la aeronáutica o del automovilismo, se elaboran y adaptan para el entorno lúdico.
- Los de contenidos ecológicos u económicos son modelos animados que se pueden experimentar directamente. Los jugadores de estos programas pueden acceder a las reglas del sistema.
- Si los juegos o simuladores de deporte son diferentes, así de diferentes son también lo son los desafíos a los que se somete el jugador.
- Mientras que para los juegos más sencillos se prueba ante todo la velocidad de reacción y la habilidad, en el caso de juegos de simulación más sofisticados se pone a prueba el pensamiento estratégico y la planificación a más largo plazo.

¿Qué es un RPG?

RPG es un acrónimo derivado , “Role Playing Game” juego de rol. En este tipo de juegos “de mesa”los jugadores interpretan un papel elegido y diseñado por ellos mismos o por otro jugador e interactúan con un mundo imaginado y descrito por un jugador designado a tal efecto denominado Narrador o Director de Juego. El Director de Juego prepara de antemano el esqueleto de una historia para que el resto de los jugadores la completen durante la partida, como si de una representación de teatro improvisado se tratase, en la que cada uno de ellos interpreta un papel, o rol, que en cada partida puede ser diferente.

El primer juego de rol de la historia, Dungeons & Dragons, fue producido por Gary Gigax y Dave Arneson, admiradores de la literatura

fantástica de Tolkien, quienes diseñaron un juego basado en la popular obra "El Señor de los Anillos". Gary Gigax y Dave Arneson tuvieron que financiar su producto por que ninguna compañía de juegos quiso arriesgarse con uno tan diferente de los de la época, pese al éxito comercial de la obra de Tolkien.

Hoy en día hay más de cuatrocientos títulos de juegos de rol publicados en todo el mundo, y es imposible saber cuántos juegos "caseros" existen. Las variaciones de los juegos "estándar" y los "juegos con normas de la casa" hacen que la cuenta crezca notablemente. No obstante los beneficios que este tipo de juegos producen son cada vez mayores, especialmente tras el éxito de la saga de juegos de Mundo de Tinieblas®, producido por WhiteWolf® y distribuido en España por La Factoría de Ideas® y Distrimagen®.Las producciones cinematográficas de juegos de éxito, son cada vez más comunes y beneficiosas como fue el caso de UnderWorld(2003).

Los juegos de rol de ordenador dejan a un lado la parte más "narrativa" e interpretativa de sus homólogos de papel, para centrarse más en la acción y espectacularidad gráfica. Algunos de los títulos más famosos mantienen la característica de formar un grupo de aventureros para "luchar contra el mal" como por ejemplo la saga Baldur´s Gate® o IcedWind Dale®, ambos de Bioware® y Black Isle®, mientras que en otros la acción se centra en un único protagonista, como Diablo® de Blizzard Entertainment®.

Ultimanante ha surgido una nueva concepción de RPG para ordenador, mucho más cercana a la original de los juegos de mesa, el Rol en Red. Esta modalidad de juego los jugadores se unen a partidas en línea para formar parte de un mundo más o menos consistente, dependiendo del título, y demostrando ser una de las formas de juego más beneficiosas para la productora ya que los usuarios pagan cuotas mensuales o anuales por mantener su cuenta abierta y poder jugar. Títulos como EverQuest® de Sony®, lanzado en marzo de 1999, llegó a tener mas de dos millones de cuentas anuales y en Ebay.com se subastaron personajes por miles de dólares.

Objetivos del Proyecto

Este proyecto pretende ser una muestra de las diversas tecnologías necesarias para el desarrollo de un videojuego en tres dimensiones fácilmente ampliable y modular. Para ello se ha construido un prototipo que implementa algunas de las técnicas más populares para resolver problemas comunes en los videojuegos, como son la búsqueda de caminos o la detección de superficies visibles.

El lenguaje de programación utilizado en el desarrollo del prototipo es C++ y la herramienta para el desarrollo ha sido Borland® C++Builder® 5.0 Enterprise. Las partes programadas en C++ forman el núcleo principal de la aplicación y la mayor parte del tiempo de cómputo esta ocupado por estas partes. El cálculo de colisiones, la simulación del mundo físico, el renderizado y animación de personajes, el cálculo de visibilidad, el cálculo de caminos y de las zonas potencialmente alcanzables y la interfaz con el usuario se han programado en este lenguaje.

Para dotar de versatilidad al juego y facilitar su desarrollo así como su diversidad se han implementados diversas partes mediante *scripts* programados en LUA que interactúan con el núcleo de la aplicación mediante una interfaz programada en C++. La ejecución de código LUA es considerablemente más lenta que la de C++ por tanto solo se utilizará en las partes que aportan diversidad al juego. La carga de las zonas de juego, las características y el comportamiento de los personajes, los diálogos, la selección de la música y de las texturas son algunas de las tareas que se hacen mediante estos *scripts*.

Para renderizar las escenas del juego se ha utilizado la interfaz con el hardware OpenGl integrada automáticamente en C++ mediante las librerías gl*.h. Estas librerías proporcionan, entre otras muchas cosas, las funcionalidades para renderizar y texturizar puntos, líneas y polígonos, que son en último término los elementos atómicos que forman la imagen que ve el usuario en la pantalla de su ordenador. Para el renderizado de terrenos se ha implementado una variante de la técnica Quadtree como se explicarán más a delante.

Para interactuar con el hardware de sonido se han utilizado las utilidades proporcionadas por Borland® en su producto C++Builder®, que si bien son de escasa potencia sí permiten una rápida integración con la aplicación. Para proyectos mas complejos o en los que se desee una funcionalidad sonora mayor que la tratada se recomienda el uso de la librería fmod® (www.fmod.org) u OpenAl (www.openal.org).

Alcance del Proyecto

El desarrollo de videojuegos es la actividad por la cual se diseña y produce un videojuego, desde el concepto inicial hasta el juego en su versión final, el producto terminado comercializable y plenamente jugable.

El desarrollo de un videojuego generalmente sigue el siguiente proceso:

- Concepción de la idea del juego. Esto es lo que un usuario podría decir de él, así como, en algunos casos, pequeños esbozos de la historia del juego, es decir "de lo que trata".
- Programación del prototipo para probar las diversas tecnologías. Con este prototipo resulta más fácil hacer los estudios de viabilidad del producto final, así como estimar su calidad final y posibilidades comerciales y de diversión.
- Diseño y programación de las siguientes versiones del juego hasta alcanzar la definitiva. Estas versiones podrán o no basarse en el prototipo, pero deberán ajustarse en lo posible al diseño original del juego.
- Producción de las componentes artísticas del juego. Esta parte suele realizarse en paralelo a las anteriores, y en algunas ocasiones puede comenzar antes que estas, ya que estarán más sujetas a modificaciones. En algunos casos puede llegar a ser la parte más cara del desarrollo del juego, especialmente si se utilizan técnicas de captura de movimientos para las animaciones.
- Enlazado de todas las componentes. La integración de todas las partes de la aplicación puede ser más ardua de lo que inicialmente podría imaginarse, y puede llevar un tiempo considerable.
- Pruebas del producto final y ajustes. Las pruebas de cualquier producto software son indispensables para su correcto funcionamiento, esto es especialmente necesario en el caso de los videojuegos ya que cualquier error será fácilmente percibido por el usuario porque se hará evidente en la pantalla.

El desarrollo de un videojuego es algo mucho más complicado y extenso que lo que se puede tratar en un proyecto llevado a cabo por una sola persona durante un año. Los desarrollos comerciales actuales requieren del trabajo de unos cien profesionales de la programación y el diseño artístico, y entre dos o tres años de duración, así como de un presupuesto en general superior al millón de euros.

Este proyecto no es pues tan ambicioso y se centra en las labores de diseño y programación del desarrollo de un videojuego, probando

diversas tecnologías para cada una de las áreas de desarrollo, y dejando a un lado la labor artística que queda fuera del ámbito que nos propusimos.

Posibles ampliaciones

Este proyecto se puede continuar casi indefinidamente debido a la gran cantidad de características esperables en un videojuego comercial. La arquitectura del prototipo se ha diseñado pensando en su simplicidad y modularidad, para facilitar las tareas de análisis y ampliación. Para cada sugerencia de ampliación se aportan las ideas básicas que permitan integrar con el menor trauma posible las mejoras con el prototipo.

En cada sección de este documento se introducirán propuestas de mejoras y ampliaciones para, así como sugerencias de cómo podría llevarse a cabo. No obstante, vamos a ver aquí un breve resumen de cada una de ellas.

Como se verá más adelante, el cálculo de caminos de las entidades requiere un uso de memoria elevado ya que el espacio de búsqueda es enorme, por tanto una interesante ampliación es la mejora de la definición del conjunto de estados de búsqueda para el algoritmo de búsqueda de caminos, teniendo en cuenta que está, en esta arquitectura estrechamente ligado al cálculo de colisiones y a la ilusión de la existencia de una “realidad” física en el mundo de juego.

La misma definición del mundo requiere una cantidad extraordinaria de memoria, ya que se cuadricula el suelo con baldosas lo suficientemente pequeñas para que una única entidad quepa en ellas. Una alternativa es desechar las clases usadas en la representación Tablero y Casilla y sustituirlas por menos costosa por ejemplo listas de esferas que representen posiciones “cercanas” a entidades.

La animación de los personajes está basada en Keyframe interpolation y no permite por tanto modificaciones en tiempo real. Las clases MD2 y Modelo son las responsables de la animación, por tanto deberán modificarse. El formato de los modelos deberá permitir la animación esqueletal, se recomendándose el uso de modelos Md3 o del formato de milkshape®.

Las sombras del juego se han implementado siguiendo la técnica de sombras proyectivas con el Stencil Buffer y el resultado es lento y francamente irreal.Técnicas avanzadas como shadow mapping o shadow volumen podrían dar mejor resultado, en el ultimo caso si se computa la silueta del modelo.

El la versión final de prototipo las sombras son “circulares” y a nivel del suelo, como en la saga Unreal®, ignorando la forma del objeto que la produce y la ubicación de la fuente de luz. Para cambiar las sombras será necesario modificar la clase Universo (función render()), la clase Entidad y Modelo.

El terreno se carga de un mapa de alturas en formato RAW, cuadrado y se visualiza con una única textura. La utilización de una segunda textura como “textura de detalle” proporcionará unos resultados mucho mejores. Es recomendable que se implemente mediante la extensión glMultitextureARB que permite renderizar en una sola pasada con varias texturas antes que renderizar en dos pasadas una por cada textura. El código del renderizado del terreno se encuentra en la clase Terreno y la inicialización de OpenGl en VentanaRenderizado.

No se ha dotado a la aplicación de funciones de red. En caso de que se desease hacer el prototipo multijugador deberá diseñarse e implementarse un Servidor que gestione múltiples imágenes del prototipo como Clientes. Será necesario añadir al prototipo funcionalidades de cliente para el envío y recepción de la información y de las tareas de sincronización. Se recomienda que la clase Arbitro pase a residir en el Servidor, siendo necesario así mismo refinar el control que los *scripts* hacen del Alineamiento de las entidades cercanas, ya que de otro modo dos jugadores no podrán dialogar ni luchar.

Una de las funciones más importantes que debe tener un RPG es la serialización de las partidas, ya que la vida del juego es muy elevada y el jugador espera poder guardar lo que ha hecho hasta un momento dado para poder recuperar los datos mas tarde. Debemos guardar las posiciones de las entidades relevantes así como su orientación; si se puede estar en más de una zona deberá guardarse un identificador de la zona. Deberá así mismo guardarse la hora del sistema dentro del juego ya que debe mantenerse en caso de cargado.

Una característica común de los RPG’s es la de tener “búsquedas” o “quests”, pequeñas partidas que no tienen por qué formar parte de la trama principal del juego, pero que añaden realismo y alargan la vida del juego. Para saber en cada momento del juego que cosa han pasado, que misiones se han completado con éxito y qué misiones ni siquiera se han propuesto aun, se maneja una lista de “sucesos” en la que se insertan nodos de información de forma que el motor del juego pueda “mirar” la lista para saber si el jugador terminó con éxito la búsqueda i-esima. Si queremos serializar la partida para poder recuperarla en otro momento debemos guardar los valores de dicha lista.

El formato del archivo debe ser lo suficientemente sencillo como para no hacer imposible su tratamiento y lo suficientemente complicado como

para evitar que un jugador malicioso lo edite y "engañe" al sistema o lo vuelva inestable. Para esto suelen utilizarse técnicas criptográficas que quedan fuera del alcance de este proyecto.

La serialización no se ha completado en el prototipo del proyecto, no aunque sí el desarrollo de todas las clases implicadas; en caso de que se quisiese ampliar el prototipo con esta funcionalidad habría que modificar la clase Arbitro.

Usabilidad

La usabilidad de una aplicación resulta un término difícil de definir y más difícil de proporcionar. En esencia significa que la aplicación deberá ser fácil y agradable de utilizar para el usuario y que deberá proporcionar las funcionalidades esperadas por éste de la manera más cómoda posible.

En un videojuego la usabilidad es aún más importante, si cabe, que en una aplicación convencional ya que influirá notablemente en la experiencia de juego y por tanto en el grado de diversión del usuario. Si un usuario no se siente cómodo con un juego la primera vez que juega, o le parece muy complicado de utilizar, lo más probable es que no juegue de nuevo.

La búsqueda de la usabilidad es una de las labores más desagradecidas en la creación de un videojuego ya que todos los jugadores esperan que un juego sea fácil de jugar, dando por sentado que los controles sean "amigables" y que la interacción con el entorno sea intuitiva. El jugador buscará patrones semejantes a los que utilizó en otros juegos en el pasado, por tanto el desarrollador deberá buscar interacciones "estándar" para asemejarse a títulos conocidos y exitosos a fin hacer que el jugador se sienta cómodo con el entorno.

Ningún jugador agradecerá esta labor y únicamente se fijará en ella si el desarrollador no consigue darle el efecto deseado. Por tanto lograr la usabilidad del juego es algo obligatorio para el desarrollador, quien deberá ajustar la curva de aprendizaje del juego de la manera más satisfactoria para el jugador.

En el prototipo presentado la entrada es igual a la utilizada en los juegos del tipo RPG, basada principalmente en el ratón y, en algunos casos, el teclado. Cuando el usuario pulsa con el ratón encima de un componente del juego, el personaje activo actuará con el de la forma mas intuitiva posible, ya sea atacando a un enemigo o dialogando con un personaje no jugador amistoso. Para pausar el juego el usuario deberá pulsar espacio. El jugador podrá dar órdenes a los personajes

del juego durante la ejecución del mismo, o en modo pausa, lo que facilita la gestión de un grupo de cinco personajes. El uso de objetos del inventario del personaje se hace mediante clic de ratón. Si el objeto es usable o ingerible el personaje lo usara o ingerirá, según sea conveniente.

No sólo la entrada del sistema define la usabilidad del juego, la salida del sistema deberá ser clara y fácil de comprender para el usuario, quien deberá saber en todo momento lo que está pasando en el juego, así como el estado en que se encuentra el personaje. Si el escenario de juego debe completarse en un tiempo determinado, el usuario deberá ser capaz de estimar en todo momento sus posibilidades de éxito y el tiempo de que dispone para lograr llevar a cabo su misión, si no es así seguramente considerará su fracaso injusto o trucado. El usuario deberá encontrar fácil de entender cada componente del entorno de juego, para poder interactuar con él de forma rápida y sencilla y deberá saber con facilidad el estado de salud del personaje jugador.

En entornos con entidades amigables o animosas el jugador deberá ser capaz de distinguirlas con facilidad, ya sea mediante códigos de colores o diferenciándolos por su modelo.

En el prototipo presentado en este proyecto los enemigos se pintan de color rojo al pasar el ratón sobre ellos, los jugadores se colorean de azul y los personajes neutros o amigables se colorean de verde. El color rojo se asocia con la violencia, por tanto el jugador entenderá inmediatamente que se trata de un enemigo.

La acción desencadenada por un clic de ratón sobre una entidad es acorde con la esperada: a una entidad coloreada de rojo se le atacara, se procederá a dialogar con una coloreada de verde y en el caso de las azules pasaremos a controlarla.

Ajustar el grado de dificultad del juego es también una labor importante, ya que no deberá ser demasiado sencillo ni demasiado difícil. Si el juego es muy fácil el jugador lo encontrará rápidamente aburrido ya que no supondrá un reto para él. Si es muy difícil el jugador podrá sentirse engañado o decepcionado, pudiendo llegar a pensar que el juego hace trampas para ganar.

Arquitectura

La arquitectura de un programa es el diseño de las clases y los métodos de comunicación que hay entre ellas que permite que la aplicación funcione satisfactoriamente.

Los videojuegos reúnen muy diversas tecnologías, y en general están formados por un número sorprendentemente elevado de líneas de código, por tanto es muy importante contar con un buen diseño que permita una integración fácil y una comunicación fluida de las diversas partes de la aplicación y que facilite la ampliación y reutilización sin excesivos retoques. Es deseable que las diversas funcionalidades queden "encapsuladas" en módulos lo más independientes posibles para evitar daños colaterales en caso de desechar determinadas tecnologías o utilidades.

La arquitectura de un videojuego es una de las partes mas importantes del mismo, ya que se trata de productos software de extraordinaria complejidad y de un tamaño imponente. Por poner un ejemplo Baldur´s gate® de Black Isle® tiene mas de dos millones de líneas de código, una cifra muy similar a la de algunos sistemas operativos.

A continuación se describen las diversas decisiones de diseño que se han tomado para la realización del prototipo.

Run & Render

Ocurre en muchos videojuegos, y en el caso de los RPG´s ocurre con mayor frecuencia, que es posible que la cámara siga moviéndose mientras el juego está en pausa. Esta funcionalidad puede ser necesaria para el juego que estemos desarrollando, como es el caso de algunos RPG´s en los que el jugador espera poder parar el juego mientras da órdenes a sus avatares. En otros juegos se hace por cuestiones estéticas o para facilitar la percepción del jugador, del entorno de juego.

Además es un buen test para determinar si se han separado las dos principales tareas del juego: el refresco del mundo y el de la pantalla. Si conseguimos añadir esta funcionalidad podremos estar seguros de haber logrado la independencia de las dos tareas principales del juego y podremos garantizar la máxima independencia entre ellas.

Dicha separación es relevante ya que garantiza que no estemos utilizando la tasa de refresco de pantalla como medida del paso del tiempo, algo deseable pues la tasa de refresco no es una medida fiable del paso del tiempo ya que depende de la geometría renderizada en cada momento.

De esta forma la ejecución del sistema puede verse como la de dos hilos de ejecución "independientes", uno de ellos únicamente lleva a cabo operaciones de lectura y el otro de lectura y escritura. Si la aplicación se plantea así, resulta más sencillo separar plenamente ambas funcionalidades.

Para llevar a cabo esta separación en el prototipo se han utilizado dos interfaces: Runable y Renderizable. Cada objeto del mundo que se deba pintar en pantalla implementa la interfaz Renderizable, si se trata de un objeto que "tome decisiones" a lo largo del tiempo deberá implementar la interfaz Runable. Un objeto que deba cumplir ambas funcionalidades deberá implementar ambas tareas.

La clase Universo maneja listas de objetos runables y renderizables, de forma que en cada momento utiliza con facilidad las funcionalidades de cada uno de ellos.

Runable

La interfaz *Runable* representa la funcionalidad de actualización de un objeto en el mundo del juego. Si un objeto cambia a lo largo del tiempo o tiene un comportamiento propio adicional al renderizado deberá implementar esta interfaz.

La interfaz Runable se ha incluido en el prototipo como una clase abstracta pura:

```
class Runable
{  public:
        //función que representa la actualización del objeto
        virtual void run()=0;
};
```

Para que una clase pueda ser "runable" deberá escribir el método run() obligatoriamente o no será compilable. De esta forma cada objeto runable tendrá un comportamiento distinto y característico. Además resultará sencillo para el sistema hacer polimorfismo sobre las clases que hereden Runable y de las que en algún momento solo interese su método run, como en el caso de la función run() de la clase Universo que actualiza el estado de todos los objetos del juego.

Renderizable

La interfaz *Renderizable* representa el carácter de un objeto del juego de ser visible en pantalla, todo objeto que pueda ser pintado en pantalla deberá implementar esta entidad. El objeto Universo, de la clase Universo no se pinta directamente en pantalla pero tiene componentes que sí lo hacen, por tanto también implementa esta interfaz.

Además de la funcionalidad de renderizado, esta interfaz impone otra restricción: la localidad espacial; todo objeto renderizable lo será en determinadas coordenadas. El acceso a dichas coordenadas se hace mediante la misma interfaz. Dado el carácter "especial" de la clase Universo, no implementa las funciones de acceso a las coordenadas por tanto estas funciones no son virtuales puras y no se obliga a su implementación. Cualquier clase que herede de Renderizable y no sobrecargue dichas funciones ejecutará el código de Renderizable en su lugar.

La interfaz renderizable se ha implementado mediante una clase abstracta pura:

```
class Renderizable
{  public:
        //función que las clases hijas deberán implementar forzosamente
        virtual void render()=0;

        //función que nos da la distancia del objeto renderizable al plano yz
        virtual inline float getX(){return 0;};

        //función que nos da la distancia del objeto renderizable al plano xz
        virtual inline float getY(){return 0;};

        //función que nos da la distancia del objeto renderizable al plano yx
        virtual inline float getZ(){return 0;}

      [...]
} ;
```

Esta interfaz impone la implementación de la función render() por tratarse de una funciona virtual pura, las demás funciones podrán sobrecargarse o no.

Diseño

Como ya se ha mencionado, el desarrollo de un videojuego puede necesitar la colaboración de un centenar de desarrolladores, por tanto el diseño del sistema debe ser muy robusto, claro y fácil de entender.

Es importante que el sistema sea modular para que las funcionalidades queden encapsuladas en zonas claramente delimitadas. Así mismo es importante que el diseño sea lo suficientemente sencillo y flexible como de las funcionalidades para permitir cambios en las especificaciones debidos a modificaciones que no sean drásticas, motivadas por ejemplo a estudios de mercado o a cuestiones de marketing.

Los videojuegos deben concebirse para un mercado global para maximizar las ventas, por tanto el diseño deber permitir fácilmente modificaciones para ajustarse a la legislatura de cada uno de los países en los que queramos comercializarlo. La legislación de cada país define las categorías de edad de los videojuegos así como diversas restricciones de lo que puede o no aparecer en un juego para ser legal. Al proceso de adaptación de los juegos a la legislación local se le conoce como traducción cultural.

Los cambios del juego pueden ser realmente considerables, como fue el caso de Carmageddon® que para comercializarse en Europa fue necesario que cambiase el color de la sangre a verde, o que los modelos de peatones fuesen sustituidos por robots o por muertos vivientes ya que la legislación local impedía que fuesen seres humanos.

Patrones

Los patrones de diseño, usabilidad o arquitectura son una herramienta que facilita enormente el desarrollo de cualquier producto, aportando soluciones probadas a problemas comunes que se encuentran en el desarrollo del software. El uso de patrones agiliza las labores de diseño y producción del producto y facilita las pruebas y la captura de fallos, siendo por tanto es recomendable usarlos con frecuencia en nuestros diseños.

Algunos patrones se están convirtiendo en estándar de facto para solucionar determinados problemas y un diseñador deberá tenerlos en cuenta al diseñar su producto.

Para el desarrollo del prototipo se han utilizado algunos de los patrones de diseño más comunes y algunos de usabilidad.

Singleton

Singleton es uno de los patrones más conocidos y utilizados hoy en día. Su uso es una cuestión polémica y muchos diseñadores lo consideran una solución del tipo “lo mejor de entre lo malo”.

Singleton es útil para modelar un objeto del que queremos tener acceso global y del que existirá una única instancia durante todo el ciclo de vida de la aplicación. La mayoría de las aplicaciones, y por lo tanto muchos juegos, necesitan disponer de objetos globales que deben ser visibles en distintos ámbitos y desde distintas clases. Un gestor de archivos de datos, el gestor de la entrada/salida del sistema, o incluso la clase que representa al jugador son todos ellos buenos candidatos a ser modelados mediante el patrón singleton.

Para tomar la decisión de diseño de usar singleton o no debemos buscar las siguientes características: el objeto necesita ser visible en todo momento, y solo es necesario almacenar uno en memoria.

Si no queremos utilizar el patrón singleton o lo desconocemos lo más probable es que utilicemos alguna solución claramente menos elegante e ineficaz que el patrón.

La solución inmediata implica el paso del objeto en cuestión, o de una referencia al mismo, como un parámetro extra en todas las llamadas que requieran un acceso al mismo. Esto es claramente ineficiente, debido a que el parámetro extra tendrá que entrar y salir de la pila en cada llamada aumentando la sobrecarga del sistema, además de hacer el código más incomprensible y difícil de mantener.

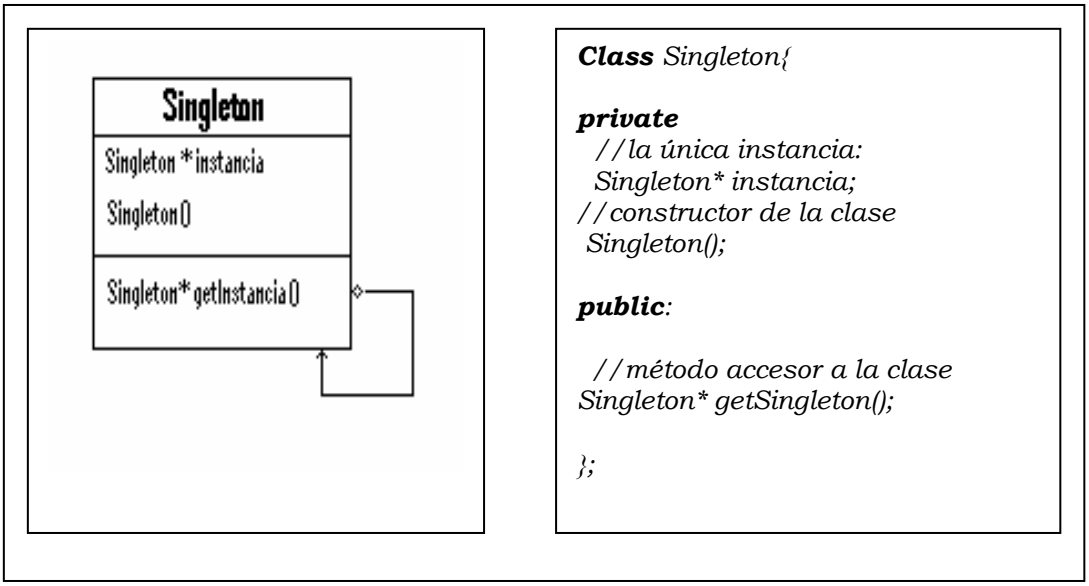
La solución al problema mediante un singleton es considerablemente más elegante que el paso de parámetros adicionales y disminuye la sobrecarga. Para que una clase pueda utilizar otra que sea singleton únicamente deberá incluir el fichero de cabecera en el propio mediante la directiva #include “Singleton.h”.

La propia clase Singleton únicamente dispone de un método público, el cual se usará para pedir una instancia al singleton, a dicho método se le suele llamar accesor y en el prototipo se denomina getNombreClase() donde NombreClase es el nombre de la clase singleton.

Todas las referencias de la clase realmente apuntan a la misma estructura, de forma que las peticiones de llamada deberán crear el singleton en su primera llamada y solo devolver punteros a las peticiones subsiguientes.

El constructor, entonces, es una función miembro protected o private, y todos los accesos externos a la clase se harán mediante una llamada al método.

La representación UML más común así como una forma simple de programarlo se muestra a continuación:



En el prototipo del proyecto el patrón singleton aparece con frecuencia en el diseño, entre otras se destacan las siguientes apariciones:

- Las Interfaces de Usuario: TSaqueo, TVentana ente otras implementan este patrón.
- La clase Universo es el mejor ejemplo de singleton del prototipo ya que es necesario acceso en casi todas las clases y únicamente hay un universo por aplicación.
- El Tablero de juego es, por comodidad, un singleton.

Factoría (Factory)

Las aplicaciones necesitan crear y destruir objetos continuamente. Como muchos programadores trabajarán sobre el mismo código, esta creación-destrucción de objetos puede propagarse hacia muchos ficheros, pudiendo causar problemas debido a inconsistencias en el protocolo, o al uso de referencias nulas a objetos que no se sabe con claridad si se han creado o no.

En otras ocasiones podremos tener una jerarquía de herencia compleja o extensa y queremos abstraerla ocultándola a otras partes de la aplicación. Desde fuera de la jerarquía podremos hacer polimorfismo sobre ella, pero si queremos hacer esto la creación de objetos se hará complicada e ilegible.

El patrón factoría centraliza la creación y destrucción de objetos, proporcionando un método sólido y universal para el manejo de los mismos.

Las factorías normalmente suelen ser de dos tipos: factorías abstractas y regulares.

- Las factorías abstractas se usan si se necesita que el producto se encuentre en una clase abstracta, y por tanto, debe derivar productos específicos mediante mecanismos de herencia. Esto es útil, debido a que permite una única llamada que devuelve un producto abstracto mediante polimorfismo. Mediante la creación de productos derivados, la clase puede acomodarse a una gran variedad de situaciones.

- Las factorías regulares, por otra parte, necesitan un método por cada tipo de productos que se quiere construir, debido a que no emplean herencia, este tipo de factorías son útiles si la cantidad de productos que crean no es demasiado elevada ya que en otro caso pueden hacer mas difícil la gestión de objetos de lo que era sin ella.

En el prototipo se ha utilizado la factoría, como factoría regular, en la clase FactoriaEntidades.

La FactoriaEntidades es, además, un Singleton. FactoriaEntidades proporciona una gran variedad de métodos para personalizar una entidad, cambiar su modelo, su guión, su estado… antes de crearla mediante la función insertarEntidad() que la inserta en el universo.

Estado (State)

El patrón estado es un patrón de usabilidad, por tanto no se tiene en cuenta para organizar la estructura de clases de la aplicación, pero sí para la secuencia de los casos de uso de ésta.

El patrón estado determina una entrada visual a la configuración actual de usuario y del entorno de la aplicación. Especifica una forma elegante de informar al jugador de sobre ciertos valores de parámetros del juego.

Un uso importante con respecto a este patrón es la prevención de la ambigüedad. El número de estados que especifica la opción debe ser pequeño y su representación simbólica clara y distinguible. Además, debe ser simple de identificar especialmente para aquellas variables que cambian constantemente de estado.

La información del estado debe ser fácilmente deducible de su representación en pantalla para que el usuario aprenda pronto a identificar los diversos estados con la interfaz de usuario.

En el prototipo se utiliza el patrón de usabilidad estado en el coloreado de las entidades cuando el usuario pasa el ratón sobre ellas y en la percepción del estado de salud de los avatares mediante barras de vida .

Progreso (Progress)

Progreso es uno de los patrones de usabilidad más extensamente usados. Su principal aplicabilidad consiste en mostrar información cuantitativa acerca de un proceso que consta de un comienzo y de un fin, de forma que el usuario conozca cuánto tiempo durará el proceso y dónde se encuentra actualmente.

El prototipo muestra el progreso mediante una barra en los tiempos de carga de la aplicación, para que el usuario tenga una estimación de lo que se ha cargado, de lo que queda por cargar y de lo que tardará dicha carga. Además, si utilizamos este patrón el usuario podrá saber con facilidad si la aplicación se ha quedado colgada o si esta haciendo algo aunque no se perciba con facilidad qué es.

Escudo (Shield)

Se trata de otro patrón de usabilidad, empleado en todo tipo de aplicaciones. Consiste en la inclusión de ventanas de información y peticiones de autorización ante cualquier actividad que pueda ser destructiva o irreversible en la aplicación. Un ejemplo son las ventanas de “salir y guardar” o “¿desea salir?” a las que estamos acostumbrados.

El patrón escudo proporciona una segunda oportunidad al usuario cuando intenta realizar una acción potencialmente destructiva o dañina, y como tal debemos implementarlo en cada una de dichas situaciones.

STL

En el mes de abril de 1995, los comités de ANSI e ISO C++ publicaron su primer documento oficial, el Committe Draft (CD), para el estudio y revisión de la comunidad de desarrollo informática internacional.

La biblioteca C++ Standard, conocida como Standard Template Library (STL), es una biblioteca que incluye clases contenedoras y algoritmos basados en plantillas.

STL es una biblioteca muy importante para el desarrollo de programas profesionales dado que facilitará considerablemente el diseño y construcción de aplicaciones basadas en estructuras de datos tales como vectores, listas, conjuntos.

Usar STL en lugar de programar las estructuras de datos a medida reducirá considerablemente el tiempo de desarrollo de las aplicaciones y los errores del programa. STL se ha probado y se revisa cada año, y es gratuito, puede recompilarse para casi cualquier plataforma por lo que su integración en cualquier sistema es extraordinariamente sencilla.

Está muy extendida la idea de que STL es muy lento e ineficaz, de hecho si se hace una comparativa de dos programas idénticos que realcen una tarea monótona, como contar la cantidad de *aes* en una lista de 10 millones de nodos en el que uno de ellos utilice STL y otro una lista hecha a medida, la versión con STL saldrá perdiendo, incluso por un orden de magnitud ya que internamente STL está hecho con herencia y las llamadas se propagan hasta el padre que las implementa y resultan mas ineficientes. Pero si los dos programas se recompilan en modo release seleccionando las mejoras de velocidad o desenrollado de inlineing y herencia la versión STL será seguramente mas rápida que la versión casera y sin duda mucho mas de lo que era antes de la optimización.

Teniendo en cuenta esto y el tiempo de desarrollo que se ahorra por utilizarlas, además de su fiabilidad y de su estandarización resulta recomendable utilizarlas en nuestras aplicaciones.

Para que una aplicación utilice correctamente los contenedores STL es necesario que la clase de la que queremos crear la estructura tenga las siguientes características:

- **Constructor por defecto**: al crear la estructura STL vacía el sistema podría considerar necesario llenarla de objetos inútiles, para ello llamará al constructor por defecto, este es el caso de la creación de un vector utilizando el constructor que determina un tamaño inicial al vector.
- **Constructor por copia**: en los redimensionados de las estructuras es posible que el gestor necesite reubicar en memoria los objetos, para ello utilizará el constructor por copia.
- **Operador de asignación**: al redimensionar la estructura será posible que la STL necesite realizar asignaciones.
- **Destructor**: al destruir la estructura se llamará automáticamente a los destructores de los objetos alojados.
- **Operador de comparación**: para realizar las operaciones de ordenación.

Dado que los objetos alojados en la STL pueden ser muy grandes, para evitar redimensionados frecuentes se recomienda el uso de STL de punteros, es decir de estructuras que almacenen punteros a objetos en lugar de los objetos en sí.

Clases contendoras STL

El componente principal de STL es la familia de clases contenedoras. Contenedores son objetos genéricos que se utilizan para almacenar otros objetos. Ejemplos de ellos son listas, vectores, conjuntos,…

La especificación STL define los contenedores de la siguiente forma:

"Contenedores son objetos que almacenan otros objetos. Controlan la asignación y liberación de estos objetos a través de constructores, destructores, operaciones de insertar y borrar"

Los contenedores soportan la implementación de la noción de agregación (u objetos empotrados) con cardinalidad de 1:N.

De entre los diferentes tipos de contenedores existentes, únicamente se han utilizado aquellos de tipo secuencial para el desarrollo del prototipo. Un contenedor secuencial es aquel que contiene datos organizados de modo lineal.

Contenedores usados en el prototipo.

Para el desarrollo del prototipo se han utilizado solo algunos de todos los contenedores que STL proporciona: vector, priority queue y deque; aunque las deque fueron posteriormente sustituidas por vectores.

Vector

Un Vector es un contenedor secuencial similar a un array inteligente que mantiene información del tamaño y que puede cambiar el tamaño de modo transparente a medida que se necesita ampliar la estructura.

La característica más atractiva de los vectores es que proporcionan acceso aleatorio a sus datos de coste constante. Cuando el vector se redimensiona es necesario reubicar todos los elementos en memoria por con el impacto que eso supone.

Los elementos de un vector se almacenan físicamente juntos, gracias a esta característica es posible calcular los incrementos necesarios para poder acceder aleatoriamente a los elementos del vector.

Las inserciones son eficientes, a menos que requieran expandir el vector ya que en ese caso sus datos se deben copiar a otro bloque de la memoria. Por consiguiente, las expansiones del vector pueden requerir

temporalmente más de dos veces la cantidad de memoria ocupada y un tiempo que puede llegar a ser visible para el usuario.

En el prototipo el universo maneja vectores de entidades declarados de la siguiente forma:

Ejemplo de declaración:

```
#include <vector.h>
[..]
//punteros para evitar problemas al redimensionar
vector<Entidad*> entidades;
[…]
```

Ejemplo de inserción:

```
entidades.push_back(entidad);
```

Ejemplo de uso:

```
for(int i=0;i<entidades.size();i++)
    entidades[i]->run();
```

Doble cola (Deque)

Es una estructura de datos de tipo cola en la que se insertan y eliminan objetos por cualquiera de los extremos. Combina las características de un vector con los de una lista. En este sentido, una deque es una lista en la cual están permitidas las operaciones de indexado. El deque estándar permite también inserciones en cualquier parte del contenedor, aunque las inserciones centrales no son tan eficientes como en los extremos.

Cola de Prioridad (Priority queue)

Una cola de prioridad, o montículo, es un adaptador que permite implementar un tipo de lista en la que un elemento inmediatamente disponible para recuperación es el mayor o el menor de los que están en la secuencia en un orden preestablecido. La obtención del valor mínimo o máximo se hace mediante la función pop.

En el prototipo la primera versión mejorada del Algoritmo A* utilizo esta estructura. Posteriormente se rechazo debido al coste de actualizar un nodo intermedio de la lista, ya que priority_queue impone la extracción y posterior inserción de todos los elementos. Para ampliar esta funcionalidad se utilizo un vector y una envoltura. En la sección de A* se profundiza mas en esta estructura.

```
#include <queue.h>

int main()
{
priority_queue<queue<char*>,greater <char*>> cola;
cola.push("cosas");
cola.push("busca");
cola.push("antes");
while(!cola.empty())
{
   cout<<cola.top();
   cola.pop();
}
return 0;
}
```

Producirá la salida "antes busca cosas"

Algoritmos STL

La biblioteca STL contiene un conjunto de algoritmos que operan sobre estructuras de datos definidas dentro del marco de trabajo STL.

Los algoritmos STL son genéricos: cada algoritmo puede operar no sólo sobre una estructura de datos sino sobre una variedad de estructura de datos. Los algoritmos genéricos usados en el prototipo son del tipo ordenación.

Además de los algoritmos del tipo ordenación, STL proporciona algoritmos basados en plantillas de tipo numérico, algoritmos de secuencia de tipo no mutable y algoritmos de secuencia de tipo mutable.

Algoritmos de ordenación STL

Los algoritmos de ordenación STL incluyen, entre otros, algoritmos de ordenación, de mezcla, de búsqueda binaria y operaciones de conjunto en secuencias ordenadas y permutaciones entre otros. Estos algoritmos tienen versiones que utilizan el objeto operador < o el objeto Compare.

Los algoritmos de ordenación ordenan una secuencia de acceso aleatorio, por tanto resultan apropiados cuando se utilizan con vector o con arrays.

Los algoritmos genéricos de ordenación ordenan una secuencia de acceso aleatorio. Se han incluido en el prototipo los siguientes:

Sort: basado en el método de ordenación Quicksort.
Merge: mezcla dos listas o rangos ordenados y sitúa el resultado en un rango o lista ordenada.

Diseño del Prototipo a grandes rasgos

Para el desarrollo del prototipo se han utilizado los patrones anteriormente mencionados así como algunas de las utilidades STL de las comentadas anteriormente.

Todas las ventanas del Prototipo, salvo TVentanaInicial se han implementado con el patrón Singleton, de esta forma es fácil para una ventana lanzar a otra o alterar sus contenidos visuales. TVentanaInicial no se ha implementado como Singleton puesto que Borland Builder necesita una Ventana que herede de TForm para inicial la aplicación, ya que la clase Aplication que gestiona los eventos de entrada necesita un formulario para su construcción.

El núcleo de la aplicación es la clase Universo, que también implementa el patrón Singleton. Mantiene una colección de los objetos que pueden formar parte del escenario, utiliza para ello vectores STL por sus facilidades de acceso aleatorio. Los renderizados se dirigen desde Universo, así como las actualizaciones del mundo de Juego.

Dentro del Universo residen las Entidades, Runables y Renderizables, representando a los personajes del juego. Todos los personajes del juego, ya sean personajes jugadores, neutrales, amigables o monstruos son entidades. Las entidades tienen asociados Modelos, que únicamente son Renderizables, que encapsulan la información para cargar, visualizar y animar modelos MD2. La carga de modelos se ha hecho usando un tutorial de *Gametutorials*, añadiéndole funcionalidades para que algunas animaciones terminasen y para que todas pudiesen pausarse.

Las entidades tienen un comportamiento basado en guiones escritos en LUA; la clase Guión encapsula la información para cargar, enlazar y ejecutar los scripts que determinan el comportamiento de las entidades.

Para el cálculo de caminos y las colisiones entre las entidades se creó la clase Tablero, formada por Casillas, que encapsula la información necesaria para realizar dichas tareas.

Para resolver el problema de la búsqueda de caminos se realizo una implementación de A* por pasos, dicha implementación esta encapsulada en la clase Navegador. Cada entidad tiene un Navegador.

Los diálogos, escritos en LUA, también se han encapsulado para facilitar su uso, en la clase Dialogo. Los ítems que los personajes pueden recoger, robar y utilizar son Ítems, que almacenan en sus Inventarios. Toda entidad tiene un Inventario, en el que caben cuatro Ítems. Esta restricción es fácilmente superable, pero no se ha hecho en el prototipo.

Las transformaciones geométricas, las proyecciones y reproyecciones se hacen en la clase Cámara, que también es un singleton. La cámara también se encarga de calcular el volumen de vista y de realizar las operaciones para saber si un objeto está dentro de dicho volumen o no.

Tecnologías

En el desarrollo de un videojuego se utilizan muchísimas tecnologías, de muy diversa índole. Para el desarrollo de este proyecto me he centrado en las labores de programación y diseño del sistema, dejando a un lado las labores y tecnologías artísticas. A pesar de centrarme exclusivamente en dichas tecnologías no ha sido posible investigar las tecnologías utilizadas en la industria profesional con la profundidad deseada debido a la amplísima gama de problemas y algoritmos que la programación de videojuegos tiene.

A pesar de todo se han utilizado muy diversas tecnologías para el desarrollo del prototipo, entre ellas cabe destacar las siguientes:

- **Gráficos:** las funcionalidades graficas se han desarrollado con OpenGL. Se ha implementado una versión de Quadtree para los terrenos y culling para terrenos y entidades. Se ha hecho la gestión de las animaciones así como la posibilidad de pausarlas en cualquier momento.
- **Inteligencia:** para la búsqueda de caminos se implementado una versión de A* que se ejecuta por pasos debido al tamaño del espacio de búsqueda. Para aumentar su efectividad se han implementado diversas mejoras.
- **Script:** para personalizar las acciones de las entidades se ha implementado un motor de scripting de LUA. Dicho motor permite la personalización de los escenarios del juego, de los objetos, el comportamiento de las entidades y los diálogos.
- **Física:** el cálculo de las colisiones se ha hecho mediante tablero y está basado en la altura de lugar en el mapa del terreno.
- **Sonido:** para las funcionalidades sonoras se han utilizado las BFC´s de Borland®.

Gráficos

En el desarrollo de un videojuego la parte más vistosa y en la que la mayoría de usuarios basarán su primera impresión del producto son los gráficos.

En el desarrollo de un videojuego comercial pocos estudios de desarrollo producen el cien por cien del código de renderizado, ya que es muy costoso y de desarrollo lento, además de ser dependiente de la plataforma y no ser en general portable a videoconsolas.

Existen herramientas comerciales fiables que proporcionan las funcionalidades para la carga y visualización de modelos, así como multitud de efectos gráficos. Estas herramientas se conocen como motores gráficos. Algunos motores gráficos incluyen además funcionalidades para incluir comportamiento físico o funciones de red.

Algunos motores gráficos, generalmente de pago, pueden utilizarse para compilar un programa para diferentes maquinas, ya sea PC o alguna videoconsola. Mediante directivas de precompilación podemos obtener código que funcione en PC o en PlayStation2® sin tener que cambiar una gran cantidad del programa. Esto facilita el desarrollo del producto para diferentes plataformas y permite que se lancen al mercado distintas versiones del producto casi simultáneamente. De esta forma, además de ampliar el mercado del producto, resulta posible que el éxito del producto en determinada plataforma incremente las ventas del mismo producto para otra plataforma. Renderware® es uno de estos motores, ha sido utilizado en Comados2® de Pyro Studios® por poner un ejemplo.

Para el desarrollo del prototipo se ha optado por no utilizar ningún motor grafico para desarrollar las funcionalidades de la aplicación directamente en OpenGL, de esta forma se han investigado diversas técnicas graficas así como la integración de dichas técnicas con las funcionalidades físicas o de inteligencia artificial.

OpenGL

OpenGL es un software que sirve de pasarela al hardware grafico. Esta interfaz está compuesta por alrededor de 250 comandos distintos (alrededor de 200 internos y 50 en las librerías de utilidades) que se usan para especificar objetos y operaciones necesarias para producir aplicaciones en tres dimensiones interactivas.

OpenGL está diseñado como una interfaz independiente del hardware, racionalizada que se puede implementar en multitud de plataformas

hardware. Para lograr estas características, no se ha incluido ningún comando para la realización de operaciones de ventanas o la obtención de entradas de usuario; en vez de eso, se debe trabajar a través de un sistema de ventanas que controle el hardware específico que se esté utilizando.

De forma similar, OpenGL no proporciona comandos de alto nivel para la descripción de modelos de objetos en tres dimensiones.
Con OpenGL de deben construir los modelos deseados a partir de un pequeño conjunto de primitivas geométricas – puntos, líneas, polígonos y cuadricas.

Una librería sofisticada que proporcione estas características podría construirse sobre OpenGL. La GLU (OpenGL Utility Library) proporciona muchas características de modelados, como pueden ser superficies cuadricas, curvas y superficies NURBS.

Texturas

De todas las funcionalidades gráficas proporcionadas por OpenGL, el texturado de los polígonos es casi con total seguridad la que mayor impacto visual produce y la que puede dar mayor ilusión de realismo por menor coste de desarrollo y de ejecución.

Mediante el uso de buenas texturas, aplicadas correctamente podemos convertir un escenario pobre en geometría en uno perfecto para dar ilusión de realidad al jugador, así como para dotar de espectacularidad gráfica a nuestra escena.

Una textura es una imagen que “pegamos” a los polígonos de nuestra escena, como si de tatuajes se trataran, de forma que al renderizar los polígonos la textura se renderizará con la textura pegada. Podemos especificar para cada polígono la textura a utilizar, la manera de utilizarla y las propiedades y coordenadas de textura a utilizar.

Las texturas pueden provenir de cualquier formato de archivo de imagen, o pueden incluso generarse proceduralmente. Los formatos más comunes en videojuegos son JPEG, TGA, aunque OpenGL no pone restricciones a la fuente de los datos que se usaran como texturas. Una vez que se ha cargado la información de textura, ésta residirá en memoria de tarjeta, no en la memoria del sistema.

Para mostrar el proceso de carga de textura de archivo se muestra este ejemplo que extrae los datos de un fichero TGA sin comprimir de tamaño 256*256 y sin transparencia. El ejemplo se ha extraído del libro de Daniel Sánchez Crespo Dalmau:

```
int cargarTextura(char* nombreFichero, int identificadorTextura)
{
  glBindTexture(GL_TEXTURE_2D,identificadorTextura);
  glPixelStore(GL_UNPACK_ALIGNMENT,1) ;

 glTextParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT) ;
 glTextParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT) ;

 gTextParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
 glTextParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);

FILE* fichero;
fichero=fopen(nombreFichero,"rb");
char* rgb= new char*[256*256*3];
fread(rgb, sizeof(unsigned char),18,fichero);//salta la cabecera
fread(rgb,256*256*3,1,fichero);

glTexImage2D(GL_TEXTURE_2D,0,3,256,256,0,GL_RGB,GL_UNSIGNED_BYTE,rgb
);
delete []rgb;
return 0;
}
```

Mediante las funciones glTexparameteri especificamos el comportamiento que esperamos tenga la textura, en este caso especificamos que la textura se repita en caso de que las coordenadas de textura que especifiquemos sean incorrectas o no estén normalizadas en el rango de cero a uno, podríamos haber especificado otro comportamiento.

Además de las opciones de repetición, en este caso estamos solicitando que la textura debe ser filtrada mediante filtros lineales en caso de que la magnifiquemos o minimicemos, ya sea por acercar el polígono que la lleva o por alejarlo, o por que el tamaño del polígono sea mucho mayor o menor que el de la textura.

Coordenadas de textura

Para poder aplicar correctamente las texturas es necesario especificar sus "coordenadas". Las coordenadas de textura, conocidas comúnmente como u y v, se entienden como las coordenadas del mapa de textura para el vértice del que las estemos especificando.

Las coordenadas de textura se normalizan entre cero y uno, por tanto independientemente del origen de los datos las coordenadas de los extremos serán las mismas.

Una vez se han especificado las coordenadas de textura, el algoritmo de de "mapeo" de texturas de OpenGL las estirará o comprimirá para amoldarlas a la forma del polígono que la utiliza, interpolando los valores intermedios a los vértices, únicos puntos de los que especificamos coordenadas de textura.

Las coordenadas de textura son demasiado complejas para especificarlas a mano para todos los vértices de la geometría de cualquier aplicación que no sea un pequeño ejemplo, por ello los programas de modelado incluyen la generación automática de dichas coordenadas, que serán incluidas en el formato del dichero del modelo, para que podamos leerlas en nuestras aplicaciones.

Las coordenadas de textura se especifican para cada vértice, antes de la llamada a la función del mismo y dentro de un bloque glBegin() glEnd(), utilizando la función glTexCoord2f(float u, float v);

El comportamiento del sistema para valores erróneos de coordenadas de textura se especifica mediante la función glTexParameteri durante su creación.

Otros Usos de las texturas

Hoy en día las texturas no se utilizan únicamente para "pegar" imágenes en los polígonos de la escena, pueden utilizarse para otras tareas, algunas incluso no son gráficas.

Las texturas secundarias pueden almacenar información de emisividad de luz, de transparencia…muchos efectos gráficos sorprendentes se basan en el uso de texturas secundarias que encapsulen información no solo de color sino de comportamiento.

Las texturas también pueden utilizarse para dar información de material del polígono, tal como dureza, temperatura, el sonido que debe sonar al pisarlas, la luz que índice sobre ellas, etc

Display Lists

Renderizar objetos utilizando únicamente primitivas básicas es fácil de codificar y de entender pero resulta poco eficiente cuando la geometría se vuelve un poco compleja o extensa. Para pintar un objeto con primitivas gráficas necesitamos realizar multitud de llamadas a funciones de OpenGL; cada una de dichas llamadas a función se

traducirá como una o mas transacciones en el bus del sistema aumentando la carga del sistema entero, no únicamente la de nuestra aplicación.

Afortunadamente OpenGL proporciona métodos alternativos para tratar con la geometría, para el desarrollo del prototipo se han utilizado las "display lists", a falta de una traducción estándar o satisfactoria.

Cada comando básico de OpenGL se traduce en una orden que el sistema dará al hardware de aceleración gráfica, situada fuera del procesador y al que únicamente podemos llegar a través del bus. Como ya se ha mencionado, la ejecución de un número elevado de dichos comandos sobrecarga notablemente el sistema, incidiendo directamente en el rendimiento de nuestras aplicaciones. Para poder renderizar un objeto que tenga 20 caras triangulares serán necesarias al menos 60 llamadas a funciones de OpenGL, imaginemos la sobrecarga que puede producirse al intentar renderizar un terreno en un escenario "out doors", de exteriores, en el que podemos necesitar mas de un millón de vértices.

Las tarjetas gráficas actuales cuentan con su propia memoria local, en la que almacenan entre otras cosas los buffers necesarios para el renderizado. Es posible utilizar dicha reserva de memoria para almacenar información relativa a la geometría y propiedades gráficas de los objetos que queramos renderizar.

Las Display lists también pueden utilizar dicha reserva de memoria para evitar que los objetos cuya geometría no cambia a lo largo del tiempo no tengan que viajar en cada frame por el bus del sistema, y puedan permanecer en la memoria de la tarjeta para poder pintarse con solo una llamada.

Una display list es una secuencia de comandos de OpenGL que se transfieren a la tarjeta para que puedan ser ejecutados localmente y a la que se puede acceder son una sola llamada. Resulta necesario precisar que no son simples bloques de información de geometría, se trata de mini programas completos que se ejecutan cada vez que queramos pintarlos.

Las display list tienen la evidente ventaja de ser tremendamente eficientes ya que no introducen ninguna sobrecarga en el bus del sistema y a que la velocidad de renderizado es la mayor posible, pero para poder utilizarlas localmente en la tarjeta grafica es necesario enviarlas primero; he ahí la gran desventaja de utilizarlas. El proceso por el cual se declara, compila y envía una display list a la tarjeta es realmente costoso e impide que podamos refrescar dinámicamente las

listas, lo que impide que sean utilizadas para objetos animados o que cambien con el tiempo.

Estas características hacen de las Display lists una excelente herramienta para el renderizado de objetos de geometría extensa o compleja pero inmóvil, como es el caso del terreno. En el desarrollo del prototipo se han utilizado exclusivamente para eso, en las primeras versiones se utilizo una única lista para todo el suelo, y en las ultimas se generó un árbol de listas para representar diferentes porciones del suelo, esto se verá con mayor detenimiento en la sección Quadtree.

Al utilizar display list debemos tener en cuenta que la mejora del rendimiento de las aplicaciones que las utilicen dependerá de que sea posible ubicar en la memoria de la tarjeta gráfica la lista de comandos. Esto no siempre será posible, ya que la memoria de la tarjeta es finita y se utiliza también para almacenar las texturas y los buffers del pipeline. Si no fuese posible almacenar las listas en memoria de tarjeta el sistema las incluirá en la memoria del sistema y perderemos todas las mejoras comentadas.

Uso de las Dysplay lists

Para poder usar las display lists en primer lugar debemos conseguir un identificador de la lista mediante el cual podamos referirnos a ella. Este nombre nos lo debe suministrar la tarjeta de vídeo, que puede denegarnos el uso de listas.

```
int identidficadorListaCuadrado=glGenList(1);
```

La función *int glGenLists(int rango)* reserva espacio para rango listas en la memoria de la tarjeta gráfica y nos da el identificador de la primera de ellas, siendo el identificador de la lista i-esima igual al de la primera mas (i-1). Si el sistema no pudo crear las listas o se produjo algún fallo nos dará un cero para señalar el error.

Una vez que tenemos el identificador de la lista debemos llenarla de datos para que sirva de algo. Esto se hace encerrando el código que queramos encapsular en la lista entre las funciones glNewList y glEndList; dichas funciones no se pueden anidar.

La función glNewList recibe dos parámetros, el primero es el identificador de la lista que se va a rellenar y el segundo puede ser:

- **GL_COMPILE**: compila la lista, reserva espacio en la memoria de la tarjeta y envía los datos compilados.

- **GL_COMPILE_AND_EXECUTE**: hace lo mismo que la versión GL_COMPILE y además ejecuta la lista.

```
glNewList(identificadorListaCuadrado, GL_COMPILE);
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f(0.0f,0.0f,0.0f);
        glVertex3f(10.0f,0.0f,0.0f);
        glVertex3f(10.0f,10.0f,0.0f);
        glVertex3f(0.0f,10.0f,0.0f);
glEndList();
```

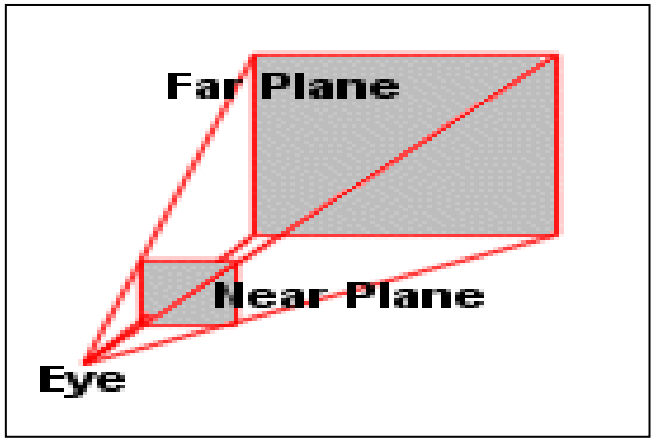
Tras estos dos pasos podremos ejecutar la lista en cualquier momento mediante la llamada a la función glCallList(int identificador).En el ejemplo, para ejecutar la lista identificadorListaCuadrado bastaría con escribir:

```
glCallList(identificadorListaCuadrado);
```

Cuando no vayamos a utilizar la lista nunca o queramos modificarla debemos liberar el espacio de memoria de tarjeta grafica, para ello utilizaremos la función *void glDeleteLists(GLuint identificador,GLsizei rango),* que liberara el espacio de tantas listas como indique rango comenzando por la indicada.

Frustum culling

La posición y orientación de la cámara, así como el FOV y la posición de los planos cercano y lejano definen un volumen, de forma piramidal truncada conocido como Frustum.



El ojo, el ángulo de la cámara, los planos cercanos y lejano definen el volumen de vista o Frustum

El Frustum define el volumen de vista, es decir el volumen del espacio de la escena que será visible en pantalla. Los objetos que no están en el Frustum no serán visibles, bien por estar detrás de la cámara, por estar muy lejos de ésta o bien por estar en una posición tal que el ángulo que forman la recta que une la posición de la cámara y la del objeto con la de dirección de la cámara sea mayor que el FOV partido por dos.

OpenGL elimina automáticamente las caras de los polígonos que no son visibles por estar "dadas la vuelta" así como los polígonos que quedan fuera del Frustum, pero lo hace en una fase avanzada del pipeline gráfico, por tanto degrada el rendimiento.

Para evitar esa pérdida de rendimiento se pueden hacer una serie de pruebas que permiten saber antes de mandar la geometría a la tarjeta grafica si dicha geometría será visible o no en pantalla. Estas técnicas están basadas en cálculos sencillos que permiten saber si un determino punto esta dentro o fuera de la porción de espacio que definen los seis planos que forman el Frustum.

Gracias a Frustum culling podemos agilizar enormemente el renderizado de nuestra aplicación, ya que podemos definir una serie de volúmenes jerárquicos tales que unos se contengan a otros, de forma que si uno no es visible no lo será ninguno los volúmenes contenidos en su interior. Este test podrá hacerse antes de enviar la geometría contenida en el volumen mediante el cálculo de las ecuaciones del Frustum y la comprobación de si están dentro o fuera del espacio delimitado por ellos.

Para poder hace el test necesitamos en primer lugar determinar el volumen de vista, es decir extraer las ecuaciones de los seis planos que lo forman. El calculo de los planos que forman el Frustum es sencillo, aunque debe hacerse en cada frame no es demasiado costoso.

Podemos obtener las ecuaciones de los planos mediante cálculos basados en la posición de la cámara, de su foco, etc… o bien podemos pedirle a OpenGL que lo haga por nosotros mediante la obtención de las matrices de proyección y modelado tras realizar las transformaciones para pintar la escena.

```
/* Obtencion de la matrizPROJECTION de OpenGL */
glGetFloatv( GL_PROJECTION_MATRIX, fPMtrx );

/* Obtencion de la matriz MODELVIEW deOpenGL */
glGetFloatv( GL_MODELVIEW_MATRIX, fMMtrx );
```

La obtención de las ecuaciones de los planos basándose en esta información es muy sencilla, pero el código es muy largo y repetitivo, por tanto no se muestra aquí. Para mas información consultar el prototipo.

Una vez hallamos calculado los planos del Frustum saber si un punto está dentro del Frustum, o si una esfera está dentro, basta comprobar que esta detrás de cada uno de los planos que lo forman:

```
bool Camara::esferaEnFrustum( float x, float y, float z, float radio )
{
 int i;
 for( i=0; i<6; i++ )
 {
 if( m_viewFrustum[i][0] * x +
          m_viewFrustum[i][1] * y +
          m_viewFrustum[i][2] * z +
          m_viewFrustum[i][3] < -radio )
          return false;
}
return true;
}
```

Mediante esta técnica se pueden testar los modelos de nuestra escena para no perder tiempo pintando geometría que no se verá en pantalla. El culling del terreno se trata en la sección Quadtree.

Para el desarrollo del prototipo se implemento otro tipo de Culling basado en el producto escalar, el producto escalar de dos vectores satisface la ecuación

A.B=|A|.|B|.cos(angulo entre A y B)

Mediante esta propiedad del producto escalar podemos calcular el ángulo que forman dos vectores normalizados y comprobar que no sea mayor que el FOV.

Esta técnica es involucra operaciones más rápidas que las necesarias para obtener las ecuaciones del Frustum a pesar de utilizar ecuaciones trigonométricas, pero el Frustum se calcula una vez por frame mientras que el producto escalar debe hacerse para cada objeto de la escena. Si la escena tiene mucha información el cálculo basado en el producto escalar será más lento y generalmente producirá peores resultados.

También se implemento en el prototipo la versión de extracción de las ecuaciones del volumen de vista y los resultados fueron mucho mejores tanto para entidades como para el terreno.

Terreno

En los RPG es frecuente que la acción se desarrolle en una escena de exteriores débilmente oculta. Los entornos de exteriores suelen ser grandes y ricos en geometría y no se adaptan bien a técnicas de oclusión populares como el uso de BSP.

Las escenas en exteriores tienen la mayor parte de la geometría en el suelo, y el suelo, en general, no está oculto por nada. Por tanto es necesario el uso de técnicas especiales para el tratamiento de esa información.

Las técnicas de renderizado de terrenos tratan de minimizar la carga que supone para el sistema la inmensa cantidad de geometría que se necesita para representar un suelo con realismo y hacer que la tasa de refresco de la aplicación siga siendo elevada.

El renderizado de terrenos no es un área exclusivamente dedicada al desarrollo de videojuegos, se han utilizado estas técnicas para realizar visitas turísticas virtuales, para entrenar pilotos militares en complejos simuladores de vuelo, o para ejecutar complejos algoritmos de predicción meteorológica.

En el campo de los videojuegos, las escenas de exteriores son algo relativamente joven, ya que hasta hace pocos años las máquinas sobre las que se ejecutaban los juegos no tenían la capacidad necesaria para poder renderizar toda la geometría en tiempo real.

El primer videojuego con un motor totalmente de exteriores que salio al mercado fue Treadmarks®, lanzado al mercado en enero de 2000.

Mapas de Altura

El terreno en un motor de exteriores esta generalmente basado en el concepto de mapa de altura, o heightmap, una imagen en escala de grises que define la altura de cada punto con un valor que va del negro al blanco. Generalmente representado con un byte, los valores pueden ser enteros del 0 al 255, dando un amplio margen para definir la altura de un punto.

El terreno del juego es una malla de puntos, generalmente cuadrada, sobre la que se hace una correspondencia con la imagen del mapa de alturas, desplazando los vértices de la malla del suelo en función del color del píxel correspondiente de la imagen.

Para el desarrollo del prototipo se han utilizado archivos en formato RAW sin cabecera cuadrados de 1024 * 1024 pixels para generar los terrenos.

Un terreno cargado de este tipo de archivos tiene 1.048.576 vértices, por tanto es necesario implementar algún tipo de algoritmo que reduzca la información que representamos en la pantalla o si no alcanzaremos una tasa de refresco elevada.

En las primeras versiones del prototipo se utilizaron mapas de 256*256 y ninguna mejora, y a pesar de todo el rendimiento no era bueno cuando se incluían muchos modelos.

Posteriormente se compiló todo el terreno en una display list, con lo que se pudieron utilizar mallas más grandes.

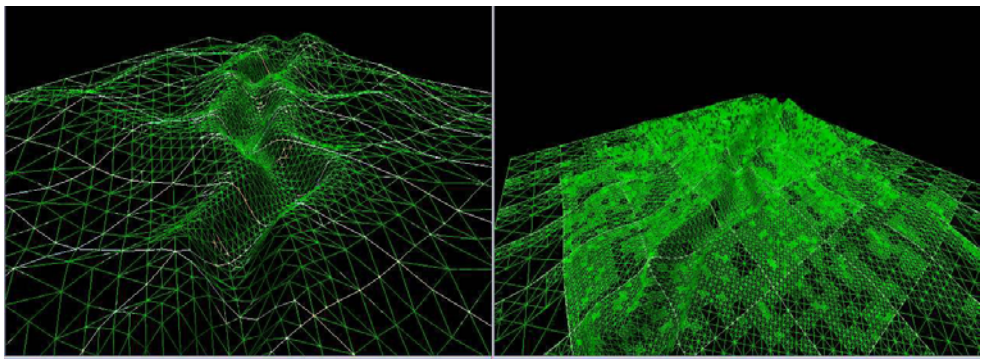
Pese al uso de display lists existe un gran parte de la geometría que no es visible y que se esta tratando como tal, y aunque no viaje por el bus por estar en memoria de la tarjeta gráfica, debe pasar por las etapas del pipeline hasta ser descartada. Por ello se implementamos una mejora del terreno, una modificación del algoritmo quadtree, descrito a continuación.

Quadtree

El algoritmo quadtree está basado en CLOD, es decir en mantener un continuo nivel de detalle en la geometría del terreno, de forma que las porciones cercanas del terreno sean las que mayor geometría contengan y que las más lejanas tengan menor cantidad de

información. Trata de dividir el terreno recursivamente formando cuatro porciones de suelo del mismo tamaño que serán testadas contra el Frustum y se seguirán dividiendo hasta alcanzar un tamaño mínimo o hasta que se determine que entran enteras en el volumen de vista y que deben renderizarse a un nivel de detalle homogéneo. En ese momento se decidirá el nivel de detalle que se le otorgará a la sección basándose en la distancia a la cámara y a la relevancia de la geometría. El árbol no se construye realmente, vive en la pila debido a las llamadas recursivas, por lo que al acabar la llamada se libera el espacio; las llamadas recursivas recorren el árbol en preorden.

Este recorrido recursivo facilita las labores de culling jerárquico de las que se hablo en secciones previas, si un volumen de terreno no es visible no lo serán tampoco ninguno de sus hijos, por tanto podremos podar el recorrido de esa rama del árbol.



En estas imágenes puede verse la ejecución de un algoritmo Quadtree, marcadas las fronteras para que pueda apreciarse la diferencia en nivlees de detalle de dos vecinos.

Los diferentes niveles de detalle causan problemas ya que generalmente una celda del terreno tendrá un nivel de detalle diferente de la de sus vecinos y aparecerán roturas en las juntas de las celdas, para evitar esto se hacen “remiendos” a la malla, o se le ponen faldones a las celdas de terreno. Ambas técnicas requieren cálculos de CPU en general costosos y pesados.

El cálculo del nivel de detalle de una sección del terreno en función de la distancia a la cámara es sencillo, pero adjudicar un nivel de detalle en función de la *forma* de la casilla es mucho mas costoso.

Si la cámara se mueve acercándose a una celda de terreno que tenía un nivel de detalle bajo puede resultar en un incremento de su nivel de detalle con la aparición de geometría que unos ciclos antes no estaba en

pantalla. Esta aparición de nueva geometría hace que el terreno parezca temblar y cambiar bruscamente de forma, para evitar que estos cambios sean muy vistosos es necesario también realizar más cálculos en la CPU.

En definitiva es una buena técnica, pero demasiado costosa en tiempo de CPU. Otras técnicas más complicadas pueden ser más efectivas, como es el caso de las dos versiones de ROAM.

Ninguna de las técnicas habituales hace uso de la capacidad de las tarjetas graficas de localizar datos en su memoria local evitando la sobrecarga del bus del sistema. A continuación se describe la técnica utilizada en el prototipo son resultados muy satisfactorios y extraordinariamente ligera para la CPU. La siguiente técnica puede depurarse aun mas, eliminando geometría del horizonte, pero esa funcionalidad no se ha implementado.

Terreno en el prototipo

La CPU esta sobrecargada durante la ejecución de un videojuego ya que esta haciendo la simulación física, los cálculos de Inteligencia Artificial y el control de animaciones, por tanto utilizar una estrategia basada en niveles de detalle puede hacer que el sistema sea demasiado lento, o gaste menos tiempo del debido en otras tareas.

Para el prototipo se planteó, diseñó e implementó una estrategia diferente, pensada para mantener a la CPU lo más lejos posible de la geometría del terreno, para disminuir lo más posible la carga que el terreno aporta al sistema.

La geometría se mantiene en la tarjeta gráfica para que no viaje por el bus y su tratamiento sea mucho más rápido. La geometría del terreno se ha dividido en casillas de 32*32 vértices para un total de 32*32 casillas para formar todo el terreno. Para cada casilla se formará una única línea de triángulos que los una a todos.

Para almacenar las casillas se utilizaron display lists. Los identificadores de las listas se almacenaron en una tabla en la clase Terreno.

Para renderizar el terreno se ha implementado una función recursiva que recibe dos puntos que forman un cuadrado de terreno, si ese cuadrado es lo suficientemente pequeño se pinta, si no es así se divide en cuatro porciones de igual tamaño que si son parcialmente visibles harán llamadas recursivas a la misma función.

El rendimiento de esta versión es el mejor de los experimentados, y deja la CPU libre en casi su totalidad

```
void Terreno::renderCuad(int x0,int y0, int x1, int y1)
{
   if( x1-x0<2 || y1-y0<2)   //CASO BASICO
      glCallList(listas[x0][y0]);
   else
   {    int xMed=(x0+x1)/2; //punto central del cuadrado
      int yMed=(y0+y1)/2;

      if(visibles(x0,y0,xMed,yMed))
         renderCuad(x0,y0,xMed,yMed);
      if(visibles(xMed,y0,x1,yMed))
         renderCuad(xMed,y0,x1,yMed);
      if(visibles(x0,yMed,xMed,y1))
         renderCuad(x0,yMed,xMed,y1);
      if(visibles(xMed,yMed,x1,y1))
         renderCuad(xMed,yMed,x1,y1);
   }
}
```

Cielo

El cielo se suele representar mediante una caja vista desde el interior que rodea a toda la escena, esta caja de denomina Skybox. Otras alternativas son el uso de semiesferas o de planos.

En el prototipo se utilizo un SkyBox de cinco caras texturizadas creado a partir del programa Terragen®.

GUI

La interfaz de usuario se ha desarrollado con Borland Builder 5.0, con el objetivo de hacer más fácil su desarrollo. Borland proporciona las BFC´s mediante las cuales resulta sencillo crear y editar las interfaces de usuario así como el manejo de eventos de entrada al sistema. Lamentablemente las BFC´s son extraordinariamente lentas e inestables y se desaconseja su uso para aplicaciones de tiempo real.

La alternativa a utilizar BFC´s u otro entorno de desarrollo de aplicaciones de usuario es utilizar el motor gráfico, o el API que estemos utilizando para el renderizado, para hacer la GUI. El manejo de los eventos es más complicado y debe llevarse a cabo por el programa pero será más rápido que el suministrado por las BFC´s.

Sombras

Para la implementación del prototipo se han utilizado dos técnicas de sombreado: sombras proyectivas y sombras planas.

Las sombras proyectivas hacen uso del Stencil Buffer. Se basan en trasformar el objeto que produce la sombra para que se "aplaste" en un plano, se renderiza posteriormente el modelo transformado y sin texturas coloreándolo de negro, mediante el buffer de plantilla el resto de la geometría no se pintará en los lugares marcados por la sombra, quedando ésta en negro.

Los resultados observados no eran satisfactorios, ya que las sombras no se adaptan a la geometría sobre la que se proyectan, además debido a la posibilidad de que el entorno esté densamente poblado y a que el stencil buffer es muy lento el algoritmo sobrecarga mucho el sistema.

La otra implementación que se utilizó es la de sombras se conoce como "planar shadows" y consiste en dibujar una sombra circular semitransparente debajo de los modelos. Estas sombras son artificiales y no tienen en cuenta la geometría que produce la sombra pero al menos no sobrecargan el sistema.

Colisiones

La gestión de las colisiones se ha hecho mediante la clase Tablero, que define unas áreas en las que únicamente puede estar una entidad a la vez.

De esta forma una entidad no podrá transitar por una casilla si está ocupada y si el ocupante de dicha casilla no es ella misma. El tratamiento de colisiones de esta forma es muy rápido, basta con leer de un array para saber si se ha colisionado con algo, y no necesita calcular distancias. Por otro lado, el uso de memoria es mayor que el que se tendría si calculásemos la distancia entre las entidades para ver si están colisionando, pero el tablero también es útil para los Triggers y para el cálculo de caminos.

Inteligencia Artificial

Para el desarrollo del prototipo ha sido necesario implementar diversas técnicas de la "inteligencia artificial tradicional" para solventar problemas comunes, el mayor de ellos es la búsqueda de caminos.

El problema de los caminos

Dados dos puntos del mapa calcular el camino mínimo, si existe, que los une no es una tarea trivial ya que existirán multitud de posibilidades en cada paso del mismo. Las estrategias algorítmicas tradicionales no funcionaran en un problema de estas características ya que es del todo imprevisible la cantidad de información que será necesario almacenar y analizar para resolverla.

Afortunadamente la búsqueda de camino es un problema resuelto hace mucho tiempo de muy diversas maneras, de entre ellas se ha implementado en el prototipo la más conocida de todas ellas A*.

A* en general

A*, leído "A estrella" es el mejor método de búsqueda en espacio de estados del tipo "primero el mejor" que sirve para resolver cualquier problema que sea resoluble mediante un conjunto finito de operaciones.

A* utiliza un estimador heurística de la distancia a la que está un estado de la solución. Combinando esa información heurística con información real del conste para llegar a ese estado es posible encontrar la secuencia de operaciones mas barata para llegar de un estado inicial a uno final.

A* maneja nodos para representar los estados posibles del problema, así como dos listas de nodos:

- Lista de nodos *Abiertos:* almacena los nodos que aún no se han explorado pero que son alcanzables
- Lista de nodos *Cerrados:* los nodos por los que ya se ha pasado pero que no son la solución, aunque podrían formar parte del camino para llegar a ella.

Siempre que exista solución al problema A* la encontrará, además encontrará la óptima y de hecho será la primera que encuentre. De entre todos los algoritmos que encuentren la solución óptima A* es el que menor numero de nodos expande, por tanto el más rápido.

El rendimiento de A* mejora notablemente si la lista de nodos Cerrados se implementa mediante una cola de prioridad en la que podamos actualizar el valor de los nodos. Para el desarrollo del prototipo se ha implementado una cola de prioridad basada en STL.

A* en la búsqueda de caminos

La búsqueda de caminos es un problema de búsqueda en un espacio de estados considerando los nodos, posiciones del espacio, por tanto podrá resolverse mediante A*, que es de hecho la solución más popular en el desarrollo de videojuegos.

Los camino producidos por A* suelen parecer artificiales y bruscos, torciendo en ángulos que no son reales, los seres humanos en raras ocasiones utilizamos el camino mínimo que une dos puntos, por tanto será necesario embellecer los caminos obtenidos mediante A*.

Splines, catmun run

Los caminos generados por el algoritmo A* resultan demasiado artificiales para resultar convincentes, por tanto es necesario tomar ciertas medidas para hacer que el resultado visual sea lo mas grato posible.

Una de las opciones que podemos considerar es la de construir un Spline tomado como puntos de control los nodos que nos da de solución al algoritmo A*.

La fórmula de Catmull-Rom es computacionalmente sencilla y construye un spline entre dos puntos considerando cuatro. De esta forma podemos generar un camino suave a partir de nuestro camino inicial repitiendo el primer y ultimo nodo del camino original.

FORMULA de Catmull-Rom

Punto de Salida = punto1*(-0.5*r*r*r + r*r - 0.5*r) +
punto2*(1.5*r*r*r - 2.5*r*r - 1) +
punto3*(-1.5*r*r*r + 2*r*r + 0.5*r) +
punto4*(0.5*r*r*r - 0.5*r*r)

R es el recorrido que queremos entre el segundo y tercer punto, desde 0 hasta 1.

Autómatas Finitos

Los autómatas finitos, máquinas de estados finitas, son una de las técnicas de inteligencia artificial más utilizadas en la programación de videojuegos. Permiten definir comportamientos complejos y convincentes de manera sencilla y eficaz.

Un autómata finito está formado por un conjunto de estados que representan los diferentes comportamientos que puede tener el

elemento en el juego, así como por las transiciones que hacen que pase de un estado a otro.

Los autómatas finitos son sencillos de diseñar utilizando notaciones gráficas, dibujándolos como grafos.

Los autómatas finitos resultan útiles para definir comportamientos en las entidades que no controla el jugador. Los diálogos, aunque no definen comportamientos se suelen implementar como autómatas finitos, siendo los estados as frases que la entidad puede decir y las transiciones las respuestas que le podemos proporcionar.

En el prototipo los diálogos y algunos comportamientos de los enemigos se han implementado mediante esta técnica.

Sistemas de reglas

Un sistema de reglas es un conjunto de sentencias de la forma "Si condición entonces Acción", y son muy útiles para el desarrollo de guiones para el comportamiento de las entidades de un videojuego ya que pueden definir multitud de comportamientos de manera elegante.

Los sistemas de reglas pueden extraerse de los autómatas finitos comentados antes.

La ejecución de sistemas de reglas es directa, se realiza un test sobre cada regla empezando por la primera, si se cumple entonces hacemos la acción que nos diga el cuerpo de la regla

Manadas

Las entidades del prototipo almacenan una referencia a otra entidad, esta entidad se conoce como "entidad objetivo". En función del carácter de la entidad objetivo se modificará el comportamiento de la entidad que la sigue. Si se trata de dos jugadores, la segunda entidad seguirá a la primera, si se trata de un personaje no jugador no hostil tratará de hablar con él, y si se trata de un enemigo intentara matarle.

Si un grupo de entidades controladas por el jugador tienen como entidad objetivo a otra también controlada por el jugador se formará una manada, un grupo de entidades que se desplazan juntas y que atacaran a cualquier enemigo que intente atacar al líder de la manada. Individualmente cada miembro del grupo se defenderá si resulta atacado, pero el resto de la manada no le defenderá a no ser que se trate del líder.

El jugador podrá formar manadas con el ratón, seleccionado a más de una entidad o pulsando sobre una entidad o su retrato mientras mantiene pulsada la tecla control.

Scripting

¿Qué es?

En el desarrollo de un RPG es indispensable implementar un motor de scripting, una herramienta que permita utilizar e integrar scripts con la aplicación desarrollada en C++ de forma que el contenido del juego esté lo más alejado de la implementación de los gráficos, el sonido o la física del juego y que adicionalmente permita la personalización de la aplicación y de los personajes.

En un RPG, y en casi cualquier género de juego, es impensable que el comportamiento de los cientos de agentes que el jugador encontrará a lo largo de su experiencia de juego estén programados en C++. Si implementamos el comportamiento, la *personalidad,* de los agentes del juego en C++ no seremos capaces de modificarla *sin recompilar* la aplicación; considerando la gran cantidad de líneas de código que puede tener un juego comercial esto supondría una carga increíble para los computadores de los desarrolladores, quienes tendrían que esperar constantemente a la recopilación para *ver* el comportamiento de los agentes y *asegurarse* de que es el correcto.

La solución a este problema es mantener esa información fuera del ejecutable del juego, ya sea en archivos de datos o en *mini programas independientes* de la aplicación que sean fácilmente integrables y manejables por ésta.

Si utilizamos archivos de datos para implementar el comportamiento de los agentes del juego será necesario implementar de todas formas la *manera* correcta de utilizar dicha información, y no se trata de una tarea sencilla.

La segunda opción es la de utilizar *programas independientes* de la aplicación y que puedan ser llamados por esta, es funcionalidad es sencilla de conseguir si utilizamos un sistemas de scripts. Un script es, en esencia, un programe en pseudo-código que adquiere su significado al ser ejecutado por una aplicación que utilice un *interprete* del lenguaje en que esta escrito. Existen multitud de lenguajes de script, o *embebidos,* con intérpretes gratuitos y fácilmente integrables.

Para el desarrollo del prototipo se ha utilizado LUA debido a su versatilidad y fácil aprendizaje.

LUA

Diseñado e implementado por el Grupo de Tecnología en Computación Gráfica - Tecgraf de la Universidad Católica de Río de Janeiro dentro del departamento de Ciencia computacional.

LUA es un lenguaje de programación potente y ligero, diseñado para ser una extensión de otras aplicaciones. LUA también es usado como lenguaje de propósito general y es un software libre.

LUA combina una sintaxis simple basada en procedimientos, con potentes descripciones de datos cuya construcción está basada en arrays asociativos y semántica extensible. LUA dispone de un tipado dinámico de datos, de una interpretación en bytecodes y de una gestión dinámica de memoria a través de un recolector de basura, lo que lo hace ideal para scripts, configuraciones y prototipos.

Un concepto fundamental en el diseño de LUA es el de proporcionar meta-mecanismos para la implementación de constructores, en lugar de proporcionar una multitud de constructores dependientes directamente de un lenguaje en particular. Por ejemplo, aunque LUA no es un lenguaje puramente orientado a objetos, proporciona ciertos mecanismos para la implementación de clases y herencia. Los meta-mecanismos de LUA generan una economía de conceptos y un lenguaje reducido, mientras que permite ampliar la semántica de multitud de maneras diferentes.

LUA es usado mayormente como lenguaje de script embebido, es posible usar enlaces a librerías SDL y OpenGL. No tiene un modelo de objetos propio, sino una estructura muy flexible denominada 'tabla', equivalente a un prototipo. Hay gran cantidad de paquetes y distribuciones modificadas.

LUA está implementada como una pequeña librería de funciones C, escritas en ANSI C y compiladas sin modificación para todas las plataformas. Su implementación tiene como objetivo la simplicidad, eficiencia y portabilidad y un bajo conste de empotramiento.

Entre sus proyectos destacan Doris, que incluye 'bindings' a OpenGL así como un visor que permite ejecutar scripts Lua con llamadas a OpenGL. Además, existe un proyecto para una distribución completa (al estilo de Python y Ruby) llamado LuaCheia, que incluye el módulo SDL y probablemente en el futuro un módulo OGL.

Lua ha anunciado la aparición de la versión 5.0 de este lenguaje de script. Entre sus principales novedades están:

- Corrutinas (multihilado colaborativo).

- Ámbitos léxicos.
- Etiquetas reemplazadas por metatablas y métodos de etiqueta reemplazados por metamétodos.
- Nuevo tipo booleano.
- Tablas débiles.
- Más rápido: nuevo algoritmo para tablas y nueva máquina virtual basada en registros.
- Nuevo protocolo de manejo de errores.
- Mejores mensajes de error.

Envoltura de objetos para LUA

Para el desarrollo de la aplicación se has debido implementar envolturas de objetos para que contengan el código de integración de C con LUA, ya que el código de integración puede ser bastante engorroso esto facilita la comprensión del código de la aplicación.

Para poder desarrollar una aplicación con LUA es necesario que incluyamos las cabeceras de LUA en nuestra aplicación, así como las librerías estáticas con las que debemos enlazar nuestro ejecutable para que nuestra aplicación pueda utilizar las funcionalidades del lenguaje.

LUA resulta extraordinariamente sencillo de utilizar como lenguaje de programación, incluso para una persona que no sepa programar en C++ su aprendizaje es suave y rápido. Su sintaxis es sencilla y parecida a la notación en pseudo código tradicional; podemos utilizar funciones, variables, tablas,...incluso podremos ejecutar código escrito en C y llamar desde C a lo que hayamos escrito en LUA.

El prototipo utilizar Scripts para definir las zonas del juego, en la clase Script, para definir el comportamiento de las Entidades en la clase Guión, para hacer los diálogos en la clase Dialogo, y para los objetos mágicos en la clase Item.

Conclusiones

El desarrollo de videojuegos es una actividad extraordinariamente compleja, que requiere la colaboración de muchos profesionales altamente especializados. Para el desarrollo de un prototipo que permita evaluar las posibilidades comerciales del producto final, es necesario invertir una gran cantidad de tiempo a la concepción y diseño del producto.

En este proyecto se ha realizado una implementación de un prototipo que podría ser utilizado para la evaluación y estimación del coste del producto final y de la cantidad de personal que podría ser necesaria.

En el desarrollo de un RPG es fundamental contar con un buen algoritmo de escenas en exteriores para poder desarrollar las características que el usuario espera en un juego. Para el desarrollo de un motor de exteriores es necesario implementar algún algoritmo que gestione el renderizado del terreno, ya que será el objeto de la escena que más información geométrica contenga. El uso de técnicas que permiten evitar las transferencias del bus en el ciclo de renderizado mejora notablemente el rendimiento.

Para dotar de un comportamiento único a las entidades que pueblan el entorno en que se desarrolla el juego, es necesario integrar nuestra aplicación con un lenguaje de scripts. Para poder usar con facilidad las utilidades que este tipo de lenguajes proporcionan, resulta conveniente dotarles de una envoltura de objetos que separe los guiones del resto de la aplicación. LUA es un lenguaje de scripts muy flexible y fácil de utilizar, altamente recomendable para implementar videojuegos.

Bibliografía

OpenGL

- "OpenGl programming Guide"-Maso Woo y otros
- "Begininig opengl game programming"-Dave Astle y kevin Hawkins
- "Real time rendering"-Tomas Akenine-Möller y Eric Haines
- "Focus on 3D terrain Programming"-Trent Polack

Game Programming

- "Game programming gems" vols 1 y 2- Varios
- "Core techniques and algorithms in game programming"-Daniel Sánchez Crespo Dumal

Scripting

- "Game scripting mastery"-Alex Varanese

Paginas web recomendadas

OpenGL

- **http://nehe.gamedev.net/**

Game Programming

- **www.gametutorials.com**
- **http://www.firestorm.go.ro**

Buscadores

- www.koders.con
- www.google.com

Inteligencia Artificial

- www.generation5.org

Glosario, Siglas y Abreviaturas

A*: “a estrella”, algoritmo genérico de búsqueda en espacios finitos, idóneo para resolver problemas de espacios de estados, utiliza información heurística para acercarse antes a la solución.

Artificial Stupidity: “estupidez artificial”, técnicas para hacer falible a la inteligencia artificial en aquellos problemas para los que es demasiado eficaz.

BSP:“binary space partition”, partición binaria del espacio frecuentemente almacenada en un árbol muy útil para el calculo de colisiones y de oclusiones en algoritmos de interiores.

BSP-tree: árbol en el que se suele almacenar la información relativa a la partición recursiva del espacio.

BFC: “Borland fundation classes”, las clases que proporciona Borland para facilitar el desarrollo de aplicaciones.

Comportamiento emergente: fenómeno por el cual un conjunto de elementos sencillos guiados por reglas sencillas produce un comportamiento de grupo complejo.

Chase: “perseguir”, algoritmos que facilitan que una entidad se acerque a otra o al lugar donde ésta se dirige para que el resultado visual sea de una persecución “inteligente”.

Culling: operación por la que se reduce geometría de una escena.

Display list: a falta de una traducción mejor “lista de dispositivo”, lista de comandos opengl que almacenamos en la memoria de la tarjeta para evitar sobrecarga en el bus del sistema.

Exteriores: serie de algoritmos y características de renderizado de entornos débilmente ocluidos.

Fmod: librería de sonido muy popular en el desarrollo de videojuegos.

FOV: ”field of view”, campo de visión, el ángulo que tiene la lente de la cámara por encima del cual no se proyectan los objetos del mundo.

Frame: un refresco de pantalla, ciclo completo de renderizado.

Frame rate: frames por segundo; medida de la velocidad del renderizado de la escena. Si el frame rate cae por debajo de 25 frames por segundo se perderá el realismo y las animaciones serán bruscas.

Frustum: volumen de vista, pirámide truncada dentro de la cual están los objetos que se proyectan en la pantalla, los objetos que están fuera de la pirámide no se proyectaran, cien por estar detrás de la cámara, bien por estar muy lejos o bien por estar fuera del FOV.

Heightmap: "mapa de altura", imagen en escala de grises utilizada para representar un terreno.

LOD: "Level of Detail", nivel de detalle, técnica para reducir la cantidad de polígonos de la imagen de pantalla basándose en cálculos de la distancia y relevancia del objeto a representar.

LUA: lenguaje de *script* muy popular debido a su fácil manejo y versatilidad.

NURBS: "non uniform B-Splines", B-splines no uniformes. Curvas paramétricas que se pueden reconstruir a partir de sus puntos de control sin información adicional de la curva.

OpenAL: librería de interfaz con le hardware de sonido, proporciona utilidades de sonido en tres dimensiones, efecto doppler y muchos efectos que añaden realismo a la experiencia de juego.

OpenGl: librería grafica de renderizado en tres dimensiones que interactúa con el hardware acelerador de video.

OutDoors: "exteriores", palabra para designar algoritmos o escenarios en exteriores, es decir débilmente ocluidos.

Pipeline grafico: "tubería", serie de etapas por las que pasa la geometría que queremos renderizar desde que la enviamos a la tarjeta hasta que se pinta en pantalla.

Patrón: solución probada a uno ovarios problemas comunes en el diseño del software que facilita la comprensión y claridad del diseño.

Quadtree: "árbol cuaternario", utilizado para el renderizado de objetos en dos dimensiones o tratables como tales, com es el caso del terreno de una escena de exteriores.

ROAM: "Real time Optimal Adaptative Mesh", algoritmo de terreno basado en niveles de detalle que produce unos resultados muy satisfactorios a costa de un elevado tiempo de CPU.

RPG: del inglés “role playing game”, juego de rol

Singleton: patrón de diseño para objetos de acceso global y frecuente.

Script: “guión”, lenguaje embebido dentro del programa que sirve para dar versatilidad a la aplicación, generalmente utilizado para dotar de un carácter único a los personajes del juego.

STL: siglas de “Standard Template Library”, del ingles la “librería estándar de plantillas” de C++ que puede utilizarse en cualquier aplicación. Incluye clases contenedoras de las clásicas estructuras de datos así como algoritmos de uso común como ordenación o búsqueda binaria.

Textura: imagen que podemos pegar a la geometría de la escena ampliando mucho el nivel de realismo y espectacularidad de nuestra escena.

Traducción Cultural: proceso por el cual se amolda un videojuego, terminado o no, a la legislación local.