

DESARROLLO DE UN PROTOTIPO DEMOSTRATIVO DE UN VIDEO JUEGO PARA
PLATAFORMA PC UTILIZANDO TECNOLOGÍAS OPEN SOURCE Y GRATUITAS DE CALIDAD
COMERCIAL

Development of a Demonstrative Prototype of a Videogame for the PC Platform Using
Open Source and Free Technologies of Commercial Quality



UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
PEREIRA - COLOMBIA
2007

DESARROLLO DE UN PROTOTIPO DEMOSTRATIVO DE UN VIDEO JUEGO PARA
PLATAFORMA PC UTILIZANDO TECNOLOGÍAS OPEN SOURCE Y GRATUITAS DE CALIDAD
COMERCIAL

Development of a Demonstrative Prototype of a Videogame for the PC Platform Using
Open Source and Free Technologies of Commercial Quality

RICARDO ANDRÉS ARANGO SLINGSBY
ALEJANDRO LÓPEZ PATIÑO

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

PEREIRA - COLOMBIA

2007

DESARROLLO DE UN PROTOTIPO DEMOSTRATIVO DE UN VIDEO JUEGO PARA
PLATAFORMA PC UTILIZANDO TECNOLOGÍAS OPEN SOURCE Y GRATUITAS DE CALIDAD
COMERCIAL

RICARDO ANDRÉS ARANGO SLINGSBY

ALEJANDRO LÓPEZ PATIÑO

Proyecto de grado para optar el título de Ingeniero de Sistemas y Computación

Asesor:

John Alexis Guerra

Ingeniero de Sistemas

UNIVERSIDAD TECNOLÓGICA DE PEREIRA

FACULTAD DE INGENIERÍAS

PROGRAMA INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

PEREIRA - COLOMBIA

2007

NOTA DE ACEPTACION

Presidente del Jurado

Jurado

Jurado

Ciudad y fecha (día mes año)

AGRADECIMIENTOS

Queremos agradecer conjuntamente a todas las personas que nos ayudaron en el desarrollo de este proyecto, particularmente a Alex, estudiante de la Universidad de Caldas por su colaboración con el modelo 3D, al estudiante Luis Miguel Echeverri de la Universidad Tecnológica de Pereira por su apoyo en la elaboración de la interfaz, a Daniel Herrera, diseñador industrial, por su colaboración con los diseños de la interfaz, a la organización Parquesoft por brindarnos el lugar para desarrollar el desarrollo del proyecto. Finalmente queremos agradecer a nuestro tutor John Alexis Guerra por las orientaciones y buenos consejos que nos dio durante el desarrollo del proyecto.

Ricardo:

Quiero agradecer a mi familia, en especial a mis padres, por copar con mi horario de trabajo nocturno y apoyarme en la prolongación de mis estudios de manera incondicional. A Natalia por estar conmigo en las buenas y en las malas, dándome sus mejores consejos y a los amigos que me ayudaron y acompañaron durante el desarrollo del proyecto.

Alejandro:

Quiero agradecer a mi familia por el gran esfuerzo que han hecho al apoyarme durante mis años de estudios y por siempre intentar darme la mejor educación, por creer en mí como persona y por apoyarme en todas las cosas que estoy.

A mi novia por entenderme en esta etapa de mi vida, a mis amigos que siempre han confiado en mis proyectos, a mis tíos y tíos que creen en mi y me dan todo su apoyo, y a todas las personas que están detrás de mi mandándome buenas energías siempre, enseñándome algo nuevo, y sobre todo creyendo en mi.

CONTENIDO

Pág.

INTRODUCCIÓN	20
1. JUSTIFICACION.....	22
2. OBJETIVOS	23
2.1. OBJETIVO GENERAL	23
2.2. OBJETIVOS ESPECÍFICOS	23
3. MARCO TEORICO.....	24
3.1. INTRODUCCIÓN	24
3.2. LOS VIDEOJUEGOS	24
3.3. DESARROLLO DE UN VIDEOJUEGO	28
3.4. DISEÑO DE UN VIDEOJUEGO	30
3.4.1. Contenido de un videojuego	30
3.4.2. Dimensiones de los retos	32
3.4.3. Aprendizaje.....	33
3.5. ESTRUCTURA DE UNA APLICACIÓN TIPO VIDEOJUEGO.....	37
3.5.1. Sistemas de Tiempo Real.....	37
3.5.2. Lenguajes de Programación	38
3.5.3. El Motor de Juego.....	38
3.6. CONCEPTOS DE GRÁFICOS 3D	41
3.6.1. Programación de Gráficos 3D.....	41
3.6.2. Fundamentos de Programación 3D	42
3.6.3. Las tarjeta de Aceleración 3D	42
3.6.4. Modos de Dibujado	43
3.6.5. Sistema de Coordenadas.....	46
3.6.6. Primitivas	48
3.6.7. Transformaciones geométricas	50
3.6.8. El paradigma de renderizado.....	52
3.6.9. El canal de dibujado.....	53
3.6.10. Texturizado	57
3.6.11. Mapas	58
3.6.12. Filtrado.....	58
3.6.13. Iluminación	59
3.6.14. Modelos de Iluminación	61

3.6.15.	Shading o Coloreado.....	66
3.6.16.	Otros Efectos	69
3.7.	MOTOR GRÁFICO.....	69
3.7.1.	Algunos objetos característicos de los motores gráficos	71
3.7.2.	Animación.....	72
3.7.3.	Librerías	74
3.7.4.	Comparación de los motores Gráficos	76
3.8.	MOTOR FÍSICO	77
3.8.1.	La Física en los videojuegos.....	77
3.8.2.	Propiedades del motor.....	78
3.8.3.	Aspectos importantes de un Motor Físico	85
3.8.4.	Librerías Físicas.....	86
3.8.5.	Chip de procesamiento físico	87
3.8.6.	Comparación de los motores Físicos	88
3.9.	SCRIPTING	89
3.9.1.	Lenguajes de Scripting.....	89
3.9.2.	Librerías	90
3.10.	MOTOR DE ENTRADA	92
3.10.1.	Programación con y sin buffer	92
3.10.2.	Sistema configurable	93
3.10.3.	Mouse y teclado vs Joystick	93
3.10.4.	Librerías	93
3.10.5.	Comparación de las Librerías de Entrada	95
3.11.	MOTOR DE SONIDO	96
3.11.1.	Tipos de Sonidos	96
3.11.2.	Audio 3D	97
3.11.3.	El Motor de Sonido	98
3.11.4.	Librerías	99
3.12.	INTERFAZ GRÁFICA DE USUARIO.....	100
3.12.1.	HUD.....	101
3.12.2.	Menús	102
3.12.3.	Librerías	103
3.13.	EDICION DE NIVELES	104
3.13.1.	El Editor de Nivel.....	105
3.13.2.	Visualización del nivel.....	107
3.13.3.	Características de los editores	108
3.13.4.	Creación del Editor	109
3.13.5.	Software gratuito o de bajo costo para edición de niveles.....	110
4.	DESCRIPCIÓN DEL PROBLEMA.....	111

4.1. Concepto Del Juego.....	111
4.2. Características	111
4.3. Motivación del Jugador.....	112
5. SOLUCIÓN PROPUESTA	113
5.1. ANÁLISIS.....	116
5.1.1. Características del Juego	116
5.1.2. Personajes	117
5.1.3. Ítems del Juego.....	118
5.1.4. Escenario	119
5.1.5. Reglas del Juego	121
5.1.6. Movimientos del Personaje principal	121
5.1.7. Menús del Juego.....	123
5.1.8. Requerimientos de Software.....	124
5.2. DISEÑO.....	126
5.2.1. Diagramas de Clase	126
5.2.2. Diseño Arquitectónico	139
5.2.3. Implementación	146
6. CONCLUSIONES	162
7. RECOMENDACIONES	164
8. APORTES	165
Bibliografía	166
Anexos.....	168
Anexo 1. Diagramas de Clase	168

LISTA DE FIGURAS

Figura 1. Diagrama de módulos de software que puede contener un videojuego.....	40
Figura 2. Diagrama de comunicación entre la CPU y la GPU. Fuente. LENGYEL, Eric. Mathematics for 3D game programming and Computer Graphics. Pág. 3.....	43
Figura 3. Captura del juego 2D Pac-Man. Fuente: Periodismo escolar en internet http://www.newsmatic.e-pol.com.ar/ . 2007	44
Figura 4. Captura de pantalla del juego Battlezone.....	45
Figura 5. Sistema de Coordenadas cartesiano con la regla de la mano izquierda en la imagen izquierda, y la regla de la mano derecha en la imagen derecha.....	47
Figura 6. Vector A definido como (x_a, y_a, z_a)	47
Figura 7. Lista de Puntos	48
Figura 8. Lista de Lineas.....	49
Figura 9. Tira de Lineas.....	49
Figura 10. Lista de Triángulos.....	49
Figura 11. Tira de Triángulos	50
Figura 12. Abanico de Triángulos	50
Figura 13. Modelo 3D con mapas de textura.....	53
Figura 14. Proceso de transformación de los vértices.....	56
Figura 15. La imagen (a) corresponde a una esfera sin textura, (b) la textura a aplicar, (c) es la esfera con la textura aplicada.....	57
Figura 16. (a) Textura de madera. (b) Plano formado por 2 triángulos. (c) Plano formado por dos triángulos con la textura repetida 25 veces.....	58
Figura 17. Dos esferas mapeadas con texturas. Fuente: Siggraph.....	58

Figura 18. La imagen del centro es la original, las de la izquierda y derecha son las usadas por el filtro isotrópico para diferentes ángulos desde donde se observa. Fuente: Advanced Graphics Programming Techniques Using OpenGL.....	59
Figura 19. En el gráfico se puede observar como una textura de una pared es modulada por una textura de luz, lo que produce un efecto de luz y sombra aplicada sobre la pared.	59
Figura 20. Tipos comunes de luces.....	61
Figura 21. Vectores que del modelo usados Phong de iluminacion.	62
Figura 22. Imagen donde se muestra cada uno de los colores que se calculan en la ecuación de Phong y el resultado final. Fuente: http://en.wikipedia.org/wiki/Blinn%20Phong_shading_model	64
Figura 23. Vectores del modelo Phong.....	64
Figura 24. El vector H de Blinn se aproxima al vector R de Phong.	65
Figura 25. Comparación resultados obtenidos con los modelos Blinn-Phong, Phong y Blinn-Phong con el valor α ajustado. Fuente: http://en.wikipedia.org/wiki/Blinn-Phong_shading_model	66
Figura 26. Esfera renderizada con coloreado plano.....	67
Figura 27. En la figura se puede ver la mejora que produce renderizar usando Gouraud (b) y coloreado plano (a).	67
Figura 28. Esfera renderizada con coloreado Phong	68
Figura 29. Estanta renderizada con normal mapping. La estanta de la izquierda tiene 4 millones de triángulos. La estanta de la derecha utiliza normal mapping, es casi idéntica a la de la izquierda y solo tiene 500 triángulos. Fuente: http://en.wikipedia.org/wiki/Normal_mapping	69
Figura 30. Dibujo de la cámara y los planos límites cercano y lejana que limitan el dibujado.....	71
Figura 31. Modelo 3D con esqueleto	73
Figura 32. Cara animada animación por vértices. Fuente: http://www.bigworldtech.cn/technology/client_animation_en.php . 2007.....	74
Figura 33. Ejemplos de aplicaciones que usan simulación física.	79
Figura 34. Formas usadas en las colisiones	80
Figura 35. Colisión entre objetos	80

Figura 36. La caja roja es un disparador que es atravesado por un objeto de colisión de color azul en la imagen.....	82
Figura 37. Simulación de una vehículo de dos ejes.....	83
Figura 38. Dos cuerpos unidos, el cuerpo de arriba es estático, el de abajo cuelga del él y se mueve rotando en el eje definido en la union.....	84
Figura 39. Dos figuras de ragdolls.....	84
Figura 40. Ejemplo de un cuerpo suave siendo deformado al aplicar fuerzas sobre su superficie... ..	85
Figura 41. Script para escribir Hola Mundo en la consola del Motor de Juego Unreal.....	90
Figura 42. Joystick tipo gamepad.	93
Figura 43. Filtrado haciendo uso de HRTF. Fuente: http://www.digit-life.com/articles2/sound-technology/index.html	98
Figura 44. En el dibujo se muestra las fuentes sonoras	99
Figura 45. Ejemplo de un HUD de videojuego.....	101
Figura 46. HUD simulando un casco en el juego Metroid Prime para la consola GameCube.....	102
Figura 47. Ejemplo de un Menú de Juego	103
Figura 48. Editor de niveles Unreal Ed 2.0.....	106
Figura 49. Flujo de datos en un editor de nivel.....	107
Figura 50. Editor de nivel con varias ventanas de edición.	109
Figura 55. Diagrama de Estados	127
Figura 56. Diagrama de Estados del Jugador.....	129
Figura 57. Diagrama de Actividades 1.	133
Figura 58. Diagrama de Actividades 2	135
Figura 59. Diagrama de Actividades 3	137
Figura 56. Diagrama de Componentes.....	138
Figura 57. Secuencia de Ventanas	139
Figura 58. Logo de la Empresa	140

Figura 59. Menú de Perfiles.....	141
Figura 60. Menú Principal	141
Figura 61. Menú de Configuración.....	142
Figura 62. Menú de Configuración de Sonido.....	142
Figura 63. Menú de Configuración de Controles.....	143
Figura 64. Menú de Configuración de Gráficos	143
Figura 65. Captura de pantalla del juego en ejecución.	145
Figura 66. Menú en Juego.....	145
Figura 67. Parámetros del objeto Nivel en gmax.....	Error! Bookmark not defined.
Figura 68. Captura de la edición del nivel con gmax	149
Figura 69. Objetos físicos de colisión del personaje.....	152
Figura 70. Flujo del Motor de Juego.....	Error! Bookmark not defined.

GLOSARIO

API (APPLICATION PROGRAMMING INTERFACE): Se le da este nombre a las funciones y métodos que provee un sistema operativo para que el programador desarrolle aplicaciones y use funciones definidas de él. En el caso de APIs 3D, son librerías de software que realizan operaciones para dibujar en 3D directamente conectadas al sistema operativo.

BUMP MAPPING: Es una técnica usada en el cálculo de la iluminación de polígonos texturizados, en la cual el valor de la normal se obtiene de una imagen en escala de grises. El color blanco significa máxima altitud, y negro mínima altitud, la cual puede ser definida por el programador. La dirección de la normal se obtiene de la normal del polígono que se está dibujando. El resultado obtenido tiene más detalles en comparación con el que se obtiene renderizando usando sólo la normal del polígono. Esta técnica ha sido mejorada por otras más recientes, entre ellas Normal Mapping.

BYTCODE: Es una representación binaria de código ejecutable diseñada para ser ejecutada por una máquina virtual (software) en vez de ser ejecutada por hardware dedicado. Dos lenguajes que compilan bytecode son Java de Sun Microsystems y C# de Microsoft.

DIRECT3D: Hace parte del API DirectX de Microsoft. Es el API gráfico usado en las consolas XBOX y XBOX360 y también puede ser utilizado para programar aplicaciones 3D en las plataformas Windows 95, 98, Me, XP, y Vista.

DIRECTX: Es una colección de APIs para manejar tareas relacionadas con la programación multimedia en las plataformas de Microsoft. El API y las librerías de ejecución están disponibles de forma gratuita, pero el código es cerrado. La versión 10 de DirectX solo está disponible para Windows Vista.

DISPARADOR (TRIGGER): Los disparadores o triggers son funciones usadas en los programas en que el sistema no sigue un flujo de control fijo, sino que su flujo es definido por eventos enviados por otros agentes, que pueden ser externos a la aplicación. Estos programas se ejecutan en un ciclo indefinido en el cual se chequea por eventos (peticiones) enviados por las diferentes partes del sistema. El disparador es la función llamada cuando se detecta un evento para resolverlo.

FACTORIES: Los factories es un patrón de diseño de software en el cual la creación y destrucción de objetos se hace de una forma simple para el programador. El programador no necesita saber el nombre de las clases involucradas, sino que a través de un método puede construir y destruir los objetos que necesita. Además, los factories permiten que se puedan definir subclases que se pueden construir usando siempre el mismo método.

FFP (FIXED FUNCTION PIPELINE): Es el camino estándar de dibujado usado en las tarjetas aceleradoras 3d, donde se realizan las operaciones sobre la información 3d de la escena para dibujarla. A los procesos realizados en el no se les pueden hacer muchas modificaciones, por esto se le dice fixed (fijo), pero es también por esto más sencillo de usar. Ha sido reemplazado en las tarjetas de aceleración 3d modernas por el camino programable, a través de shaders.

FILTRADO ANISOTROPICO: Es un método para mejorar la calidad de renderizado de objetos que están lejos y en ángulos agudos con respecto al observador.

FILTRADO BILINEAR: Es un método para suavizar las texturas cuando se ven a una escala diferente a 1:1, calculando el color de un pixel a través de una interpolación usando los 4 texeles mas cercanos de los que se obtiene el color final.

FILTRADO TRILINEAR: El filtrado trilinear es una mejora al filtrado bilinear añadiendo interpolación entre mipmaps. A grandes distancia el filtrado bilinear falla por que obtiene muestras incorrectas del texel más cercano al pixel que se quiere dibujar. El filtrado trilinear utiliza muestras de los mipmaps, versiones escaladas de la imagen y así calcular de forma mas acertada el color final. Sin embargo, presenta errores cuando el polígono al cual se le calcula el color se encuentre a un ángulo agudo con respecto al observador y que se solucionan con el filtrado anisotrópico.

FPS (First Person Shooter): Es un tipo de videojuego en el que el jugador controla a un personaje desde el punto de vista del personaje que controla, y usualmente se puede ver el arma o artefactos que esta utilizando en la parte inferior de la pantalla, como si los estuviera llevando.

FRAME (FOTOGRAMA): Es una imagen bidimensional estática. En gráficos en tiempo real por computador, una frame es una imagen renderizada de la escena. En un segundo se pueden hacer muchos frames, que se conoce con el término FPS o frames per second.

GPU (GRAPHICS PROCESSING UNIT): Unidad de procesamiento de gráficos. Es el procesador dedicado al procesamiento de las instrucciones de la tarjeta de gráficos. Se ubica usualmente en la en una tarjeta que se conecta a la tarjeta madre o directamente conectado a la tarjeta madre. Es el encargado de renderizar los gráficos y en chips avanzados de procesar los shaders. Los principales fabricantes de los GPUs son NVidia y ATI. Funcionan de forma asíncrona con el CPU, asi este le puede delegar la carga del renderizado y procesamiento de los gráficos mientras realiza otras operaciones.

JUGABILIDAD: Es un termino utilizado para describir la experiencia del jugador en un juego. Una buena jugabilidad indica que el juego brinda una experiencia agradable. Una mala jugabilidad, se refiere a un juego que tiene deficiencias que puede ser en el diseño de la idea o en la forma en que se desempeña la aplicación de juego.

LIGHTMAPS: Es una estructura de datos que contiene la información de luz de una superficie usado en los motores gráficos. Usualmente los lightmaps se almacenan como imágenes en escala de grises, que definen que tan brillante u opaco es la superficie a la que se le aplica.

LIGHTMAPPING: Así se llama en general a la técnica de aplicar iluminación a las superficies a partir de una imagen en escala de grises. Esta técnica es muy eficiente porque solo se debe hacer una multiplicación de color para obtener el resultado final de un pixel, comparado con usar una técnica de iluminación dinámica como Gouraud o Phong. Además se puede utilizar cualquier método de obtener el lightmap. Su mayor desventaja es que se calculan durante la edición y esta tarea puede tomar mucho tiempo. Además, no refleja cambios que se realice en la geometría o el movimiento de las luces.

MIDDLEWARE: Su nombre significa software intermedio. Es software que se utiliza de forma modular para realizar tareas complejas, facilitando el trabajo del programador. En el desarrollo de los videojuegos el middleware puede variar desde módulos específicos para realizar una tarea, como renderizar árboles eficientemente, hasta el sistema completo del juego.

MIPMAPPING: Es la técnica en la cual se utilizan versiones pre-calculadas más pequeñas (Mipmaps) de una imagen, para hacer más eficiente el dibujado de objetos 3d a grandes distancias y eliminar artefactos en la obtención del color final del pixel.

MOTOR (SOFTWARE): Es un software que realiza un conjunto de operaciones específicas y el cual se alimenta con información para obtener resultados. En los juegos los motores se usan para encapsular diferentes tipos de tareas, para que trabajen de una forma modular y provean al desarrollador la funcionalidad que necesita fácilmente. Un motor se diferencia de una librería en que la librería puede realizar una tarea muy específica, mientras que el motor se espera realice toda una operación completa. Por ejemplo, un motor gráfico está encargado de todo lo relacionado con el dibujado de la escena, mientras que una librería puede hacer parte del motor gráfico y estar encargada de hacer solamente la animación de los personajes.

MOD: En los videojuegos se les llama mods a las modificaciones realizadas a un videojuego. Un Mod puede incluir nuevas armas, nuevos personajes, enemigos, historias, niveles, etc. Un Mod puede ser tan solo la modificación de la forma en que se ve el juego, hasta una conversión total en la que se crea una nueva versión del juego. Algunas empresas desarrolladoras fomentan la creación de modificación de su juego brindando a los usuarios editores y librerías.

MODDING: Se refiere a la acción de modificar un videojuego.

NORMAL MAPPING: Es una modificación de la técnica conocida como bump mapping. En esta se almacena en una imagen el valor de la normal en un punto de un triángulo de un modelo 3d. Esto permite que modelos de muchos polígonos puedan ser dibujados usando modelos de pocos polígonos, utilizando los valores de las normales. Los valores de las normales se almacenan en una estructura de color RGB, donde cada color es un valor de una coordenada del vector unitario de la normal, que varía de 0 a 255. Se considera que cuando el valor del color es 127, el valor del eje de

la coordenada es cero, entonces, valores menores a 127 son negativos y mayores son positivos. El valor final se obtiene dividiendo el valor del color sobre 127. Este vector de la normal se utiliza cuando se calcula la iluminación del modelo.

PIXEL: Un pixel es un punto de color en la pantalla. El número de pixeles que muestre una pantalla determina que tan fina es la imagen que se dibuja en la pantalla. Entre mas resolución tenga, mas información de color se puede ver. Un pixel también puede ser un punto de color en una imagen.

RASTERIZACIÓN: Es el proceso de dibujar una imagen en la pantalla.

RENDERIZAR: Es el proceso de obtener una imagen bidimensional a partir de la información de una escena tridimensional, como la geometría, las luces, los materiales, los efectos de la escena, etc. La renderización en aplicaciones de tiempo real como los videojuegos se realiza constantemente varias veces por segundo. En los paquetes de software es la tarea que se realiza cuando se quiere obtener el resultado final de un trabajo de edición.

RGB y RGBA: Son dos formas de representar la información de color aditivo usado las pantallas de computador. Las siglas significan rojo (Red), verde (Green) y Azul (Blue), y corresponde a la cantidad que contiene de cada color. El valor alfa de RGBA indica la opacidad del color. En su máximo valor el color es completamente opaco. Cuando vale cero, el color es completamente transparente.

SHADER: Un shader es un programa que se ejecuta en la GPU, para calcular el resultado final de un proceso de renderización. Los shaders pueden determinar aspectos como la iluminación y el color, pero también la posición de los vértices y mas cosas para cada frame que se dibuja.

SINGLETONS: El singleton es un patrón de diseño de software que se usa cuando se tienen objetos de los cuales solo existirá una instancia a la vez en todo el ciclo de ejecución de una aplicación. Se puede utilizar para limitar la existencia de más instancias del objeto, pero también para obtener el apuntador al objeto desde cualquier parte de la aplicación sin tener que estar pasando parámetros entre funciones, o guardando apuntadores en cada objeto que lo necesite.

SDK (SOFTWARE DEVELOPMENT KIT): En español kit de desarrollo de software, es un conjunto de herramientas de desarrollo que le permiten a un programador crear aplicaciones para una tarea específica.

TEXEL: Un texel es una unidad de color de una textura aplicada en un objeto 3d. Un texel es usado por la tarjeta gráfica para obtener el valor final de un pixel, haciendo filtrados y operaciones de color. Cuando el objeto texturizado esta muy cerca del observador, un texel puede cubrir varios pixeles. En estos casos es cuando se utiliza filtrado para suavizar el resultado. De la misma forma, cuando el objeto esta lejos del observador, un pixel puede representar a varios texels, en estos casos se utiliza Mipmapping para obtener el color del texel que representa a los demás.

TEXTURA: Una textura es una imagen aplicada sobre un objeto 3d para darle detalle a la superficie del objeto.

RESUMEN

El propósito de este proyecto es especificar librerías y herramientas de código libre y gratuitas que puede ser usadas para el desarrollo de videojuegos 3D en plataforma PC. Aunque hay muchas alternativas que pueden ser tenidas en cuenta, se limitó el alcance a las principales partes de un software de videojuego: sonido, física, gráficos 3D, controles, interfaz de usuario y edición de niveles. El trabajo está dividido, en forma general, en una introducción a los conceptos de programación de videojuegos en que se definen las partes principales y algunas librerías y herramientas de código libre y/o gratuitas, y en el diseño e implementación de un motor de juego 3D, aplicando un diseño modular para obtener una librería de componentes reusables. Se concluye analizando los resultados y se hacen unas recomendaciones.

Palabras clave: motor de juego, desarrollo de videojuegos, gráficos 3d.

ABSTRACT

The purpose of this project is to specify free and open source libraries which can be used for the development of 3D video games for the PC platform. Although there are many alternatives that can be considered, the scope was limited to the main modules of videogame software: sound, physics, 3D graphics, controls, user interface and level editing. The work is mainly divided in two parts, an introduction to the concepts of video game programming in which the main elements are defined along with some applicable free and/or open source tools and libraries, and the design and implementation of a 3D game engine, applying a modular design to obtain a library of reusable components. It concludes analyzing the results and making some recommendations.

Key words: game engine, videogame development, 3D graphics.

INTRODUCCIÓN

El mundo actual se ha caracterizado por ser objeto de grandes transformaciones culturales y sociales. En poco más de 50 años se ha vivido las mejoras en muchos países de los derechos civiles, la protección al medio ambiente, la exploración espacial, y el desarrollo de el área tecnológica que mas cambios ha producido en nuestro estilo de vida: la electrónica y la computación. Hoy en día existen infinidad de dispositivos y aparatos electrónicos de alto, medio y bajo rango para todas las clases económicas. La electrónica y los computadores se han inmiscuido en todas las aéreas que toca el hombre: la ciencia, los deportes, la educación, la música, la vida en el hogar, entre otros.

Una de las aéreas que ha revolucionado el mundo por su rápido ascenso en términos de ganancias y cobertura que ha sido producto de la electrónica y la computación es la de los videojuegos. Hoy en día los videojuegos generan más dinero que la industria del cine, US\$18 Billones estimados para el 2007¹ solo en EEUU. A la par con el crecimiento de las ventas, hubo un crecimiento del numero de empresas, numero de empleados y países involucrados en la industria, y fueron surgiendo nuevas profesiones y tecnologías para suplir la demanda del mercado por cada vez mas videojuegos y de mejor calidad. Los institutos educativos, que en sus inicios estuvieron rezagados, comenzaron a ofrecer programas de estudio enfocados a la industria de los videojuegos. Hoy en día se tiene registradas 486 ofertas educativas² con programas orientados al desarrollo de videojuegos, la mayoría de ellas en Estados Unidos.

A pesar de este crecimiento y la oportunidad de negocio, en Colombia la industria de los videojuegos es incipiente. Aunque en el país se estimen unos 53.000 jugadores registrados³ (se supone hay muchos mas), la industria propia del país se podría decir esta rezagada 40 años o más, en la época en que los desarrollos eran mínimos y solo existían unas pocas empresas aventurándose en el negocio. Las razones fundamentales para esto son tres: la piratería, la falta de conocimientos y la falta de recursos. A nuestro criterio, creemos que teniendo profesionales con conocimientos sobre el proceso de producción y comercialización de videojuegos, se pueden solucionar o buscar alternativas a la piratería y se pueden encontrar recursos.

¹ Revista GameSpot UK, En una entrevista a una analista del grupo NPD, líderes en información de ventas de videojuegos. 14 de Septiembre de 2007,

http://uk.gamespot.com/news/6178789.html?action=convert&om_clk=latestnews&tag=latestnews;title;4.

Última visita el 16 de Septiembre de 2007

² Game Career Guide, sitio web especializado en la orientación profesional en el área de videojuegos.
<http://www.gamecareerguide.com/schools/>. Última visita el 16 de Septiembre de 2007

³ Revista Enter 2.0, Junio 15 2007, http://enter.com.co/enter2/ente2_actu/ente2_actu/ARTICULO-WEB-NOTA_INTERIOR_2-3543560.html. Última visita el 16 de Septiembre de 2007

El objeto de este proyecto no es elaborar un curso de creación de videojuegos, seria insensato pensar en hacerlo en un solo documento, ni analizar las causas de la piratería. Se busca proveer suficiente información acerca del proceso de desarrollo de los videojuegos, desde el punto de vista del programador y así dar inicio a una búsqueda de conocimientos mas avanzados en el tema de cómo diseñar y programar videojuegos. En el desarrollo del proyecto se mencionan varias herramientas que se pueden usar para crear videojuegos de calidad comercial y se elabora una aplicación demostrativa de los temas tratados en el documento.

Estructura del Proyecto

Documento:

- Capítulo 1.** Justificación, se presenta la problemática que da origen a la tesis.
- Capítulo 2.** Objetivos, se define los resultados que esperan alcanzar en el desarrollo del proyecto.
- Capítulo 3.** Marco Teórico, se dará a conocer una compilación de conocimientos relacionados con las tecnologías aplicadas en el desarrollo de aplicaciones de videojuegos.
- Capítulo 4.** Descripción del Problema, se propone crear un prototipo demostrativo de un videojuego en el cual se aplicaran las tecnologías descritas en el Capítulo 3.
- Capítulo 5.** Solución propuesta, se hace un análisis del problema propuesto y haciendo uso de las técnicas de ingeniería de software y se diseña una aplicación que muestra el uso de las tecnologías y desarrolla parte del videojuego descrito en el Capítulo 4.
- Capítulo 6.** Conclusiones, se presentan los resultados obtenidos a largo del proyecto.
- Capítulo 7.** Recomendaciones, se comparten algunas experiencias obtenidas durante el desarrollo de la tesis, y se proponen algunas ideas de como complementar lo expuesto en el trabajo.
- Capítulo 8.** Aportes, se enumeran las contribuciones que hace el proyecto a la comunidad universitaria.

- DVD:** Archivos de ejecución del Videojuego.
Tutoriales de programación usando la librería Ogre3d.
Copia digital del documento.
Librerías utilizadas en el Videojuego.

1. JUSTIFICACION

La industria de videojuegos genero ingresos por el valor de US\$13.5 billones en los Estado Unidos en el año 2006 solamente⁴, generados por la comercialización videojuegos y artículos relacionados. Esta incluye una gran cantidad de profesiones, y emplea a miles de personas en el mundo.

En Colombia actualmente es escaso el desarrollo de video juegos, ya que existen pocas empresas desarrolladores de este tipo de software. Esto se debe en gran parte a que el tiempo de recuperación de la inversión suele ser mayor a un año, el número de personas que se necesitan para desarrollar un juego de alta calidad es muy grande, las habilidades requeridas son muy especializadas, los costos de hardware, software y los salarios son altos para una empresa nueva o pequeña, el mercado interno esta invadido de piratería y no existe una cultura de compra de videojuegos originales en Colombia.

El avance de las tecnologías utilizadas en los videojuegos ha dado como resultado que los videojuegos actuales sean aplicaciones complejas en contenidos y desarrollo, tanto en calidad como en cantidad. Como consecuencia las herramientas y actividades que hacen parte del desarrollo son cada vez más sofisticadas y especializadas. Como consecuencia los costos de desarrollo han aumentado, creando una barrera cada vez más alta para quienes quieren incursionar en este medio. Además, en Colombia no existen ofertas educativas especializadas que enseñen lo necesario para la creación de un videojuego, lo que hace más lejana la posibilidad de que la industria se desarrolle localmente.

Esto hace valioso encontrar una solución a estas barreras, buscando herramientas de bajo costo que permitan desarrollar videojuegos con la mayor calidad posible, integrándolas para crear una respuesta completa y hacer uso de ellas para desarrollar un prototipo en el cual se pueda ver su alcance. Este conocimiento podrá ser usado por otros como cimiento para el desarrollo de sus propios juegos, beneficiéndolos en costos y tiempo.

⁴ NPD Group, líderes en información de ventas de videojuegos. Sitio web

http://www.npd.com/press/releases/press_070119.html Ultima visita realizada el 14 de Septiembre de 2007

2. OBJETIVOS

2.1. OBJETIVO GENERAL

Desarrollar un demo de un videojuego para Windows XP utilizando tecnologías de software libre o gratuito de gráficos 3D, simulación física, sonido e interacción usando joystick.

2.2. OBJETIVOS ESPECÍFICOS

- Investigar los productos y procesos realizados en el desarrollo de un videojuego.
- Definir de forma generales tecnologías y áreas de sistemas computacionales involucradas en el desarrollo de videojuegos.
- Examinar las tecnologías libres o gratuitas existentes para gráficos 3D, sonido, simulación física, inteligencia artificial y dispositivos de entrada para el desarrollo de videojuegos.
- Crear un diseño conceptual del juego, con énfasis en la cultura autóctona de la región, que defina los objetivos, personajes, y jugabilidad.
- Desarrollar una versión demo del juego utilizando las metodologías de programación orientada a objetos que integre las diferentes tecnologías.
- Identificar áreas de aplicación del trabajo desarrollado.

3. MARCO TEORICO

3.1. INTRODUCCIÓN

En este capítulo se presenta una recopilación de conocimientos relacionados con las tecnologías aplicadas en el desarrollo de videojuegos. Se hace una breve introducción a los conceptos básicos sobre los videojuegos y su producción. En cada uno de los subcapítulos siguientes se enfoca en un área específica utilizada en la creación de videojuegos: gráficos 3d, simulación física, inteligencia artificial, uso de controles, entre otros. Al final de cada subcapítulo se recomiendan algunas de las librerías de código libre y gratuitas disponibles actualmente para la programación de videojuegos en computador, con el fin de que sean investigadas por el lector.

Hay que resaltar que los videojuegos actualmente utilizan tecnologías complejas de software que no podrían ser expuestas en un solo documento y a las cuales se dedican varios autores por tecnología. Aun así, se intenta explicar brevemente las tecnologías más importantes.

Requerimientos

Para entender estos capítulos se requieren conocimientos de programación, preferiblemente orientada objetos. En la teoría de gráficos 3d se mencionan matrices y vectores, para esto es necesario tener conocimientos del álgebra vectorial.

Lo que se va a aprender

Al leer este capítulo el lector podrá tener una noción clara de las tecnologías de software que se utilizan en el desarrollo de los videojuegos. Conocerá acerca de gráficos 3d, simulación física, uso de controles, inteligencia artificial, scripting, interfaz de usuario, edición de contenido del juego y como esto se enlaza en el motor de juego.

Estará en capacidad de enumerar algunas de las librerías disponibles para crear juegos en computador para cada una de las áreas tecnológicas.

Entenderá que es un videojuego, como se desarrolla y conocerá algunas de las profesiones que aplican a su desarrollo.

3.2. LOS VIDEOJUEGOS

Marcela Castro, magíster en Literatura de la Universidad Javeriana, en su estudio del juego (CASTRO, 2002) enumera seis características propias de la actividad del juego, y de tal forma de los

juegos en si, obtenidas de su análisis de la obra del escritor y sociólogo Roger Callois, en su libro "Los Juegos y Los Hombres" (CAILLOIS, 1986):

- Libre, de otra forma no podría disfrutarse.
- Separada, con un espacio y tiempo determinados.
- Incierta, su desenlace no se puede predecir.
- Improductiva, porque no produce bienes ni riqueza.
- Reglamentada, con su propias leyes.
- Ficticia, con su propia realidad

El videojuego o videogame es un juego que se basa en un soporte electrónico como un ordenador, consola, máquina recreativa, etc. No existe diferencia más que en forma entre los juegos y los videojuegos, aun más, en la clasificación que se presenta de géneros de videojuegos, se puede ver que los videojuegos incluyen los cuatro tipos de juegos de la clasificación de Callois (CAILLOIS, 1986), según el elemento predominante:

- Competencia
- Azar
- Simulacro
- Vértigo

Es decir, podemos definir a los videojuegos como un subconjunto de los juegos.

Comúnmente se conoce a los videojuegos que se ejecutan en un computador como "Juegos para PC", para consolas como Playstation⁵ o XBox⁶ se les llama "Juegos para consola". La palabra videojuego entonces reúne a todos los videojuegos sin diferenciar la plataforma en la que se utilicen.

Lo videojuegos se pueden catalogar en diferentes géneros dependiendo de la forma en que se juegan o su jugabilidad. Los géneros según Diego Levis (LEVIS, 1997) más conocidos y en uso hoy en día son:

- **Juegos de lucha**

⁵ Videoconsola desarrollada por Sony lanzada en 1994. Se destacó por utilizar el CD y tener gráficos en 3D. Sitio Web. <http://www.playstation.com>

⁶ Videoconsola desarrollada por Microsoft lanzada en el 2001. Se basó en el sistema x86 y ejecuta una versión modificada de Windows 2000. Se destacó por su fácil conversión de juegos de PC a la consola, tener un disco duro y poder jugar en línea. <http://www.xbox.com>

Uno, dos o más jugadores se enfrentan en una pelea. El ganador es quien consigue más puntos o quien derriba a su oponente. Las peleas son cuerpo a cuerpo o con armas de cortas. Es uno de los géneros donde se expresa mayor violencia. Algunos títulos representativos de este género son: Street Fighter II, Soul Calibur, Killer Instinct, Super Smash Bros Melee, Tekken, Dead or Alive.

- **Juegos de combate**

Muy similares a los juegos de lucha, pero en este caso uno o varios jugadores deben vencer a muchos oponentes luchando contra ellos en un recorrido por varios escenarios. Los escenarios se muestran hostiles y la violencia es la única forma de vencer. Algunos juegos más relevantes de este género: Golden Axe, Teenage Mutant Ninja Turtles, Battletoads, Final Fight, Double Dragon.

- **Juegos de tiro**

Uno o más jugadores se enfrentan a varios oponentes, utilizando armas de fuego o adaptaciones de ficción (por ejemplo armas láser). Los juegos giran alrededor de hacer buen uso de las municiones, las armas y tener la mejor puntería. Hoy en día se han actualizado al uso de internet y se puede jugar con varias personas a la vez que estén en cualquier lugar del mundo, lo que también ha añadido complejidad reflejada en la elaboración de tácticas de grupos usada para ganar. En este género podemos encontrar a: Doom, Halo, Counter-Strike, Battlefield, Medal of Honor, James bond, Unreal.

- **Juegos de plataforma**

Son los juegos en el que el jugador debe atravesar el juego andando, saltando sobre plataformas, con enemigos, mientras se recogen ítems para poder completar el juego. Es un género que fue muy importante en los 90's cuando los gráficos de los videojuegos se limitaban a 2D. Con el desarrollo de los gráficos 3D perdieron vigencia, aunque la han recuperado gracias a los dispositivos móviles como agendas, celulares, y las consolas portátiles. Algunos de los juegos importantes son: Prince of Persia, Super Mario Bros, Castlevania, Metroid, Super Monkey Ball, Super Paper Mario.

- **Simuladores**

Son los juegos en que se quiere hacer una simulación de la realidad. Se busca que el jugador pueda cumplir las reglas de la simulación de la mejor forma. Este tipo de juegos se extiende en muchos temas, así que podemos encontrar simuladores de vehículos (Need for Speed, Project Gotham Racing, Test Drive), simuladores de sistemas económicos

(Railroad Tycoon, Lemonade Tycoon), construcción de ciudades (Simcity, Caesar, City Life), entre otros.

- **Deportes**

Los juegos de deportes se caracterizan por imitar un deporte. La idea es que el jugador pueda realizar las mismas actividades que se realizan en un deporte y ganarle al juego o a otros jugadores, en el mismo sitio o en línea. También se suelen agregar actividades como administración de los equipos, elección de estrategias, compra-venta de jugadores, entre otras. En este género podemos encontrar a: FIFA, NBA, NFL, Winning Eleven, Pro Evolution, Punch Out, Wii Sports, Nascar, BMX, Tony Hawk's Pro Skater

- **Juegos de estrategia**

En estos juegos lo más importante son las decisiones que el jugador toma en el juego, debe pensar estratégicamente. La rapidez de los movimientos y la puntería pasan a segundo plano, y gana quien piense más y mejor. El jugador generalmente les da órdenes a varios personajes para que realicen diferentes tareas que en conjunto sirven para conseguir un objetivo. Los juegos más conocidos de este género son los juegos de estrategia en tiempo real, o RTS por sus siglas en inglés. Algunos juegos importantes de estrategia son: Civilización, Final Fantasy, Age of Empires, Warhammer, Warcraft.

- **Juegos de mesa**

Son adaptaciones de juegos de mesa clásicos, como por ejemplo: Ajedrez, Crucigrama, Tres en Raya, Monopolio.

- **Ludo-educativos**

Son juegos con fines educativos mezclados con actividades lúdicas. Se busca que el jugador adquiera conocimientos específicos.

- **Porno-eróticos**

Son los juegos basados en el erotismo y la pornografía, orientado a los adultos. Algunos se presentan como simuladores de relaciones sexuales entre personas. También los hay de aventura donde se busca que el jugador conquiste a un personaje del juego. Una de las sagas más exitosas de juegos en este género es Leisure Suit Larry.

Muchos de los juegos que se crean actualmente son una mezcla de uno o más géneros. También existen los videojuegos educativos formados por un conjunto de actividades que los usuarios deben realizar, con unos objetivos claros que representan el conocimiento que los usuarios deben adquirir durante la realización de la misma. Estos juegos se diferencian de los demás videojuegos

en que la actividad si suele ser productiva. Algunos de estos videojuegos han atraído la atención de la industria, y se catalogan hoy en día como juegos “serios”⁷.

Otras aplicaciones más recientes de videojuegos ha sido en el entrenamiento de personas, rehabilitación de enfermos, divulgación de problemas sociales y ecológicos, entre otros.

3.3. DESARROLLO DE UN VIDEOJUEGO

La industria de los videojuegos es una industria muy joven, con menos de 40 años de historia comercial, desde la primera consola que llegó al mercado, la Magnavox Odyssey⁸ en 1972. Las técnicas y procesos de desarrollo de videojuegos se han transformado a la par con los avances tecnológicos traídos por las mejoras en la electrónica, lo cual ha hecho que el desarrollo de videojuegos se considere todavía un área sin madurar. Es más, la oferta académica especializada en el desarrollo de videojuegos solo comenzó desde 1996 cuando el DigiPen Institute of Technology⁹ ofreció el primer curso en desarrollo de videojuegos. Sin embargo, ya existen publicaciones y estudios fundamentados en el desarrollo de software y las experiencias de la industria,

La creación de un videojuego está a cargo del desarrollador, el cual puede ser una empresa o una sola persona. Los juegos a gran escala comerciales en la actualidad se desarrollan típicamente con equipos de 20 o más personas y con costos promedio de US\$10 millones¹⁰. Usualmente Los fondos para el desarrollo son aportados por un distribuidor, y el tiempo para completar el juego varía entre uno y tres años, aunque hay excepciones.

Un equipo actual para el desarrollo de un videojuego requiere de personal con diversas habilidades, entre ellos encontramos:

- Productores
- Diseñadores
- Artistas
- Programadores
- Ingenieros de Sonido
- Ingenieros de Software

⁷ Serious Games es una organización dedicada al desarrollo de la industria de videojuegos serios.

<http://www.seriousgames.org>

⁸ The Online Odyssey Museum. <http://www.magnavox-odyssey.com>

⁹ DigiPen Institute of Technology. <http://www.digipen.edu>

¹⁰ Cost of making games set to soar. <http://news.bbc.co.uk/1/hi/technology/4442346.stm>

- Relacionistas Pùblicos
- Publicistas
- Probadores

Es usual en equipos pequeños que uno de los desarrolladores ocupe varios de estos cargos. Por ejemplo el dueño de una empresa, puede ser el productor y diseñador o programador líder.

El primer paso en el desarrollo de la idea un videojuego, es el de elaborar un documento de diseño, creado por el diseñador del juego, quien se encarga de darle forma a la idea, y concebir junto con el resto del equipo, el concepto global del proyecto. El diseñador también se encarga de eliminar aquellas ideas que no sean realizables o que no sirvan en concreto para el proyecto, y de refinar aquellas que si sean realizables. Cuando éste acaba, los desarrolladores formulan un documento técnico con las primeras especificaciones del proyecto, trabajando a la par con el diseñador para refinar equilibrar los diseños con el análisis técnico y los artistas comienzan a trabajar en el concepto gráfico que tendrá el proyecto, en esta etapa, se deciden cosas como el nombre del proyecto, la plataforma operativa (PC, consola, sistema operativo, etc.), lenguaje de programación y herramientas de diseño, entre otras cosas. Y de tratarse de un proyecto serio, aquí es cuando el Productor, decide entre otras cosas, el presupuesto del proyecto.

La parte más intensa de todo el proceso es la producción, aquí, los programadores trabajan en el motor del juego (el corazón de la aplicación), la inteligencia artificial, crean herramientas de desarrollo, interfaces de usuario, etc. Los artistas y animadores, crean el arte del proyecto, imágenes, dibujos, animaciones, los personajes, los objetos, etc. todo debe quedar bien definido, los músicos, crean la banda sonora del juego y los efectos de sonido necesarios durante todo el proceso, en ésta etapa, estos equipos no trabajarán aislados, sino que en la mayor parte del tiempo, retroalimentarán a los desarrolladores con el material producido. Esta es la fase más larga, ya que depende del tipo de proyecto que se haya elegido y del tiempo de desarrollo que se le dedique, además existen ocasiones en que debido a diferentes motivos, el proceso se retrasa y entonces hay que mover fechas.

En la etapa de post-producción se realizan pruebas al proyecto que ya está en su fase final. Estas pruebas consisten en emplear personas para que prueben el juego de forma meticulosa en busca de errores y fallas de diseño, los cuales son documentados y analizados para su corrección. Las pruebas pueden durar mucho tiempo, ya que una la resolución de un error puede llevar a mas errores, además, los errores en los productos finales son mal recibidos por los jugadores. A estas personas que realizan las pruebas, se les conoce como probadores, en inglés testers.

Paralelamente durante ese tiempo el resto del equipo (diseñadores, músicos, etc.) trabaja para finalizar su parte del proyecto. Cuando las pruebas acaban, el proyecto ya es un producto terminado y está listo para su producción masiva y distribución.

3.4. DISEÑO DE UN VIDEOJUEGO

Factor de diversión

Los avances tecnológicos se ven reflejados en avances para los videojuegos, día a día las tarjetas aceleradoras de gráficos tiene nuevos componentes que permiten tener visualizaciones de mayor realismo, aunque esto no hace que los videojuegos sean más entretenidos, hay algo más que hace que los juegos sean divertidos, y frecuentemente es llamado el factor diversión (en inglés Fun Factor). Como dice uno de los grandes diseñadores de videojuegos, Chris Crawford¹¹, en su libro "Chris Crawford on Game Design" (CRAWFORD, 2003):

"Si un videojuego no es entretenido, es un videojuego malo".

3.4.1. Contenido de un videojuego

Retos

Objetivos o esfuerzos difíciles de llevar a cabo, y que constituyen por ello estímulos y desafíos para quien lo afronta.

Reglas

Las condiciones en las cuales el reto es presentado. Definen qué se puede y qué no se puede hacer, cuáles son las condiciones de victoria y derrota.

Precauciones

No importa cómo sea creado el reto, alguien pensará la forma para burlar al sistema. Se debe prevenir que el jugador pueda evadir los retos.

Actividades que se realizan en los juegos

Las siguientes son algunas de las actividades que se pueden incluir en un juego. Generalmente se utiliza una combinación de ellas en un juego.

Razonamiento espacial

El jugador prueba sus habilidades para orientarse y moverse correctamente y/o fácilmente en el espacio.

Reconocimiento de patrones

Consiste en analizar lo que hay en el juego para ordenar y seleccionar objetos con características similares o diferentes, y realizar acciones a partir de este análisis. Por ejemplo, en un juego de

¹¹ Chris Crawford es un respetado diseñador y escritor de videojuegos. Ha publicado seis libros relacionados con la industria y el desarrollo de videojuegos.

sigilo, se puede crear una actividad en la cual el jugador debe aprender el patrón de movimiento de los enemigos para no ser descubierto.

Razonamiento secuencial

El jugador debe analizar todas las posibles secuencias que existan y tomar la decisión que mejor lo favorezca para lograr un objetivo.

Razonamiento numérico

Juegos generalmente de problemas matemáticos. Sudoku es un juego basado completamente en una actividad de razonamiento numérico.

Manejo de recursos

El jugador debe cuidadosamente administrar una cantidad limitada de recursos para resolver los problemas que encuentra.

Conflictos

El conflicto hace que el reto se vuelva personal. El conflicto es el factor de tensión que revela las habilidades. Un reto sin conflicto es predecible, cuando se juega en un razonamiento secuencial se sabe exactamente que va a suceder, pero cuando existe un agente activo, el jugador ya no es quien toma la iniciativa. Se crean retos en los cuales el jugador no puede anticiparse.

Niveles de dificultad

Uno de los desafíos al diseñar un videojuego es encontrar el nivel de dificultad correcto, si es demasiado difícil el jugador puede retirarse rápidamente y si es muy fácil el jugador perderá interés. Para esto se pueden crear diferentes niveles de dificultad.

Recompensas

Una recompensa es algo que hace que una persona se sienta bien. Las recompensas son esenciales en un videojuego, si no se tiene suficientes el juego puede volverse rutinario y frustrante. Usando recompensas el videojuego creará una experiencia placentera. Existen muchas clases de recompensas que se pueden usar para cada tipo de juego.

Tipos de Recompensas:

- Recursos**

En videojuegos de rol generalmente recibir recursos es una recompensa ya que servirán para completar los retos del juego. Ejemplos de estos son: comida, dinero, soldados, armas.

- Habilidades**

Algunos videojuegos tienen sistemas explícitos para que el jugador mejore sus habilidades.

- **Extensiones**
Algunas recompensas pueden darle al jugador tiempo extra, fuerza extra.
- **Extras**
Gráficos, música o videos puede ser recompensas para los jugadores. No ofrece al jugador nada en términos de jugabilidad, pero la experiencia de juego se incrementa.
- **Recompensas por realización**
Cuando el jugador concluye una actividad en el juego puede ser recompensado.
- **Motivacionales**
Puntajes, tiempo.

3.4.2. Dimensiones de los retos

Crawford (CRAWFORD, 2003) define dos tipos de retos que a los que responde el cerebro:

Retos al Cerebelo

Son retos que no requieren demasiada entrada sensorial como por ejemplo el lanzamiento de disco o Javalina.

Retos Senso-Motrices

La mayoría de los retos del cerebro incluyen un elemento sensorial, los músculos no se mueven en una determinada secuencia. Se deben usar los sentidos (generalmente la visión) para dirigir y controlar la actividad muscular.

El sentido mas usado es el de la visión. El usuario reacciona a partir de eventos que suceden en la escena de juego, pero hay que destacar que el oído también es un elemento que sirve como medio de interacción. El procesamiento de estas señales es toma tiempo, y para un jugador principiante puede ser lento y no muy divertido. La diferencia entre un jugador principiante y un avanzado es que el jugador avanzado ha construido caminos neurales cortos y rápidos desde la corteza visual hasta el cerebelo.

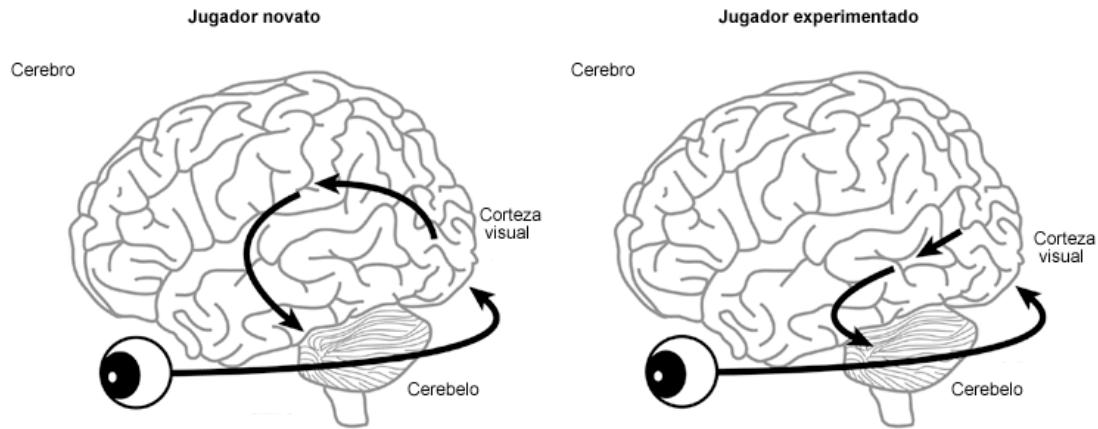


Figura 1. El cerebro del jugador novato (izq.) toma mas tiempo en responder al juego en comparación con el del jugador experimentado (der.), pues esta apenas reconociendo patrones para crear caminos neuronales mas cortos.

A medida que se generan estos caminos en el cerebro, la velocidad de reacción del jugador se incrementa y la toma de decisiones no es consciente o deliberada, es frecuentemente descrita como instintiva. El jugador ve y actúa sin un pensamiento consciente.

3.4.3. Aprendizaje

Los humanos estamos programados para aprender, y si lo que se aprende es útil y se completa general placer. Teniendo esto en cuenta los videojuegos deben estar cuidadosamente diseñados para proveer al jugador con un flujo regular de aprendizaje, que frecuentemente es llamada la curva de aprendizaje.

Aprendizaje

Se debe diseñar como el usuario va a aprender a usar el videojuego, como usar botones, formas de movimiento, y como avanzar por los niveles. La curva de aprendizaje del videojuego tiene un gran impacto en como los jugadores perciben lo que es el videojuego.

Curvas de aprendizaje:

- Curva pendiente o empinada

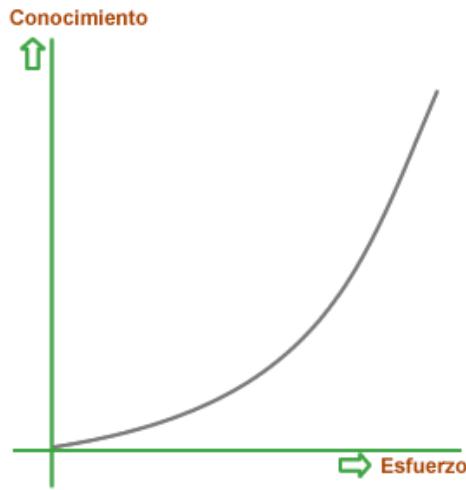


Figura 2. Curva de aprendizaje en pendiente (lenta)

A mayor esfuerzo, mayor conocimiento del juego, lo que se busca es que el jugador comprenda todas las reglas del juego.

- **Curva superficial o poco profunda**

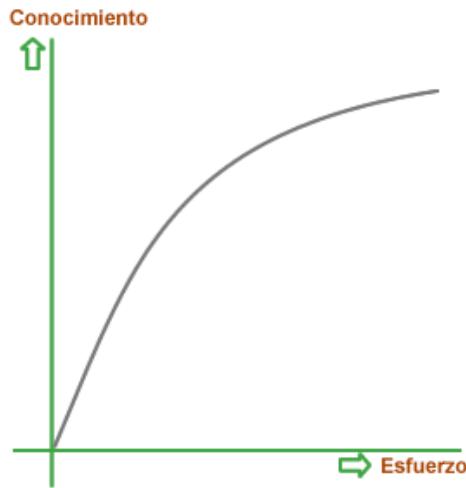


Figura 3. Curva superficial (rápida) de aprendizaje.

Usada generalmente para juegos casuales, el usuario busca jugar rápidamente sin tener que aprender todas las reglas del juego.

- **Tutoriales**

No puede existir un juego sin reglas, las cuales deben ser enseñadas de alguna forma al usuario. Una forma es a través de un manual de usuario, en el cual se describe el juego al estilo de los juegos de mesa, en los cuales se lee el manual, se memoriza y se juega. Pero a la mayoría de los usuarios no les gusta leer los documentos de ayuda, y para esto se inventaron los tutoriales, o niveles introductorios, dentro del videojuego donde se enseñen las reglas.

Con un tutorial la curva de aprendizaje puede ser la siguiente.

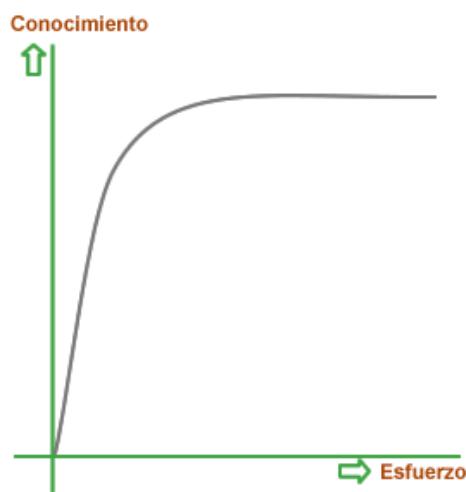


Figura 4. Curva de aprendizaje de un tutorial.

Existen también videojuegos donde las reglas son obvias, debido a que son juegos conocidos por todos, así que el esfuerzo para aprender será mucho menor, ejemplo de esto son versiones de juegos populares, como póker, solitario, o elementos del juego que sean obvios y/o comunes en otros juegos.

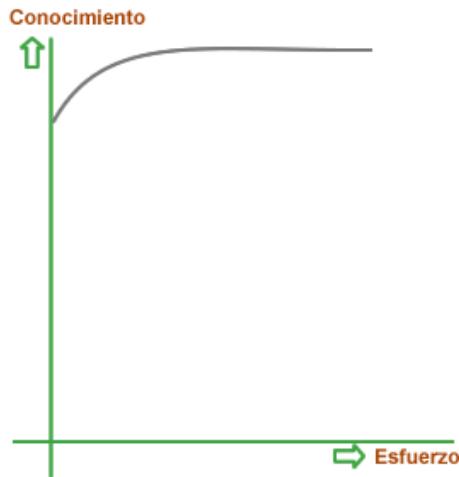


Figura 5. Curva de aprendizaje de un juego de cultura popular

En los juegos complejos con muchas reglas, se suele ir iterando paso a paso por el aprendizaje de ellas. De esta forma se pueden aprender ciertas reglas básicas, se juegan un par de niveles, se presentan nuevas reglas, se juegan otros niveles y así sucesivamente. De esta forma es posible disfrutar el juego sin tener que conocer todas las reglas, y además el jugador no se siente saturado por una gran cantidad de reglas que aprender.

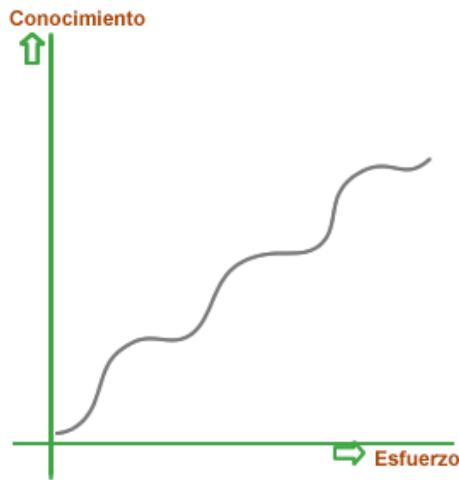


Figura 6. Curva de aprendizaje de un juego con reglas complejas.

3.5. ESTRUCTURA DE UNA APLICACIÓN TIPO VIDEOJUEGO

Los videojuegos son aplicaciones de software que comparten características con las aplicaciones de tiempo real (SANCHEZ-CRESPO, 2003). Aunque existen algunos juegos en los cuales no importa la velocidad de respuesta del sistema, por ejemplo aquellos en que el juego ocurre por turnos, la norma es que los videojuegos requieren de una respuesta rápida a las acciones del jugador.

3.5.1. Sistemas de Tiempo Real.

Los sistemas de tiempo real son aquellos donde el funcionamiento correcto del sistema depende de obtener resultados correctos dentro de un tiempo finito y generalmente corto. Los sistemas de tiempo real se clasifican en dos categorías, tiempo real ‘duro’ y tiempo real ‘suave’.

Los sistemas de tiempo real ‘duros’ son aquellos en donde no debe existir ningún fallo ni demora en la obtención de los resultados del sistema. Un cálculo mal hecho o tardío implica un fallo del sistema. Generalmente se encuentran en sistemas empotrados, como por ejemplo el sistema que controla el electrocardiógrafo en un hospital, y que brinda la información con los signos vitales y el estado del paciente.

Los sistemas de tiempo real ‘suaves’ son los que permiten alguna demora o algún error en la información. Esta información tardía es válida si se mantiene dentro de unos rangos de error. Un ejemplo típico de sistema de tiempo real suave son los videos sobre demanda por Internet. El usuario necesita que el video fluya de manera constante y casi-instantánea, pero una perdida de algunos cuadros del video o una pequeña reducción de velocidad de no impide que la información sea transmitida y la comunicación se logre.

La forma en que funcionan los videojuegos hace que se clasifiquen como sistemas de tiempo real interactivos. Se pueden además clasificar como software de tiempo real suaves, que de forma similar a un video sobre demanda desde Internet, pueden perder calidad y algunos cuadros por segundo, y hasta ‘pegarse’ (que ocurre al cargar datos en medio de un nivel, o cuando hay escenas con mucha información por momentos, mientras se conserve la interacción y no se vea perjudicada la experiencia de juego).

En un videojuego se pueden encontrar 3 módulos principales para el funcionamiento:

- Módulo de simulación del mundo virtual que alimenta el sistema con información.

- Módulo de presentación que le muestra el mundo al jugador.
- Mecanismo de Control para interacción con el jugador, generalmente a través del teclado y el ratón para juegos en computador, y controles para juegos de consola.

Sin embargo esta estructura de un videojuego es de muy alto nivel y omite muchas áreas que están involucradas en un videojuego actual. Es necesario hacer una división más orientada a las funciones que realizan, divididas en módulos más pequeños y más manejables en términos de programación y tareas que realizan.

3.5.2. Lenguajes de Programación

En términos generales un videojuego puede ser programado usando cualquier lenguaje de programación. Sin embargo, los juegos de calidad comercial deben mostrar cualidades técnicas superiores, que solo se consiguen sacando el máximo provecho de la maquina. Por esto los lenguajes mas utilizados son ensamblador, C y C++, que tienen el mejor rendimiento y el programador mayor control sobre el manejo de los recursos. Existen excepciones como en el caso de los celulares, donde se usa el lenguaje que requiera el celular, por ejemplo Java, juegos que se ejecutan en el navegador web, que utiliza generalmente Flash¹² o Javascript, y algunas librerías que son específicas de algún lenguaje, como la librería XNA¹³ de Microsoft que utiliza C# para programar juegos de XBOX 360.

3.5.3. El Motor de Juego

Un concepto generalizado en el desarrollo de videojuegos es el del motor de juego, en inglés “Game Engine”. Un motor de juego es un componente de software que permite darle toda la funcionalidad al juego. Reúne todas las diferentes tecnologías que requiere el juego para su funcionamiento, típicamente gráficos, sonido, scripting, física, colisiones, animación, sistema de menús, inteligencia artificial, redes o multi-jugador, entre otras y esta encargado de la lógica del juego en sí. Cabe anotar que no todos los juegos tienen los mismos requerimientos, algunos no necesitan de todos estos módulos para su funcionamiento.

¹² Flash es un conjunto de aplicaciones desarrollada por Adobe Systems Incorporated para la producción y ejecución de contenido multimedia en navegadores web y dispositivos móviles.

<http://www.adobe.com/products/flash/>

¹³ Microsoft XNA es un SDK para el desarrollo rápido de videojuegos para las plataformas Windows XP y Vista, y Xbox 360, <http://msdn.microsoft.com/xna>

La idea detrás del término motor, es que similar a un vehículo, el motor está escondido a la vista pero es quien hace posible que funcione el automóvil. Todos los recursos del juego como los modelos geométricos, las texturas (imágenes), sonidos y más harían parte de la carrocería, lo que se ve. Esta analogía además tiene un valor agregado, con el mismo motor podemos hacer un carro rojo, azul, verde, o un sedan, un descapotado, o un montero. Es decir idealmente con un mismo motor podríamos hacer varios juegos, de tantas formas como la flexibilidad del software lo permita.

Algunas compañías desarrollan sus propios motores que utilizan para hacer sus juegos. A principios de los ochenta cuando hubo un boom de juegos de aventura para PC, la compañía LucasArts¹⁴ creó un motor llamado SCUMM, con el cual desarrolló 12 juegos en un período de más o menos 10 años. Este motor separaba el código del diseño del juego, lo que les permitió ahorrar dinero ya que para hacer un juego nuevo no necesitaban software nuevo. El término sin embargo se consolidó en los noventa gracias a motores creados por compañías para ser vendidos a otras compañías para hacer sus juegos. El ejemplo más conocido es Id Software¹⁵ con los diferentes motores que ha producido, el más notorio fue Quake III Arena¹⁶ el cual fue utilizado en decenas de juegos diferentes.

En el desarrollo de un juego es común entonces hablar del motor de juego y de las tecnologías mencionadas y otras más que va a incluir, y como se van a trabajar. Algunas veces se utiliza middleware para módulos de software. Es así que el motor de Unreal 2.0, desarrollado por la empresa Epic Games¹⁷ y utilizado en los juegos Unreal Tournament 2004 y Unreal 2 utilizó la librería comercial Havok¹⁸ para realizar la interacción física de los objetos y personajes con el mundo. En el desarrollo del motor se debe tener planeado el uso de middleware desde el comienzo para crear las interfaces adecuadas. Aun si todo el software va a ser desarrollado por la empresa, sigue siendo importante que las diferentes funciones del motor se lleven a cabo por librerías modulares, que permitan correcciones, mejoras e independencia en su manejo buscando alta cohesión y bajo acoplamiento.

¹⁴ LucasArts. <http://www.lucasarts.com>

¹⁵ Id Software <http://www.idsoftware.com>

¹⁶ Quake II Arena <http://www.idsoftware.com/games/quake/quake3-arena/>

¹⁷ Epic Games <http://www.epicgames.com>

¹⁸ Havok. <http://www.havok.com/>

Se pueden hacer sub-divisiones del motor de juego, dependiendo de la función que cumpla un área específica, en algunos casos combinando varias tecnologías relacionadas en un solo módulo, como por ejemplo el Motor Gráfico encargado de crear y configurar el sistema gráfico, mostrar los modelos, hacer animaciones, etc. A continuación está una subdivisión del Motor de Juego, basada en los módulos más comunes que se conectan con middleware a un motor de juego y su función general:

- Gráficos: Mostrar el juego en pantalla
- Entrada de Usuario: Controles, Teclado y/o Mouse
- Física: Colisiones, Simulación Física
- Scripting: Scripts (programas interpretados) para interacciones
- Red: Comunicación por red. Multijugador.
- Sonido: Efectos de Sonido. Música de ambiente.
- Lógica del Juego: Reacciones y adaptaciones del juego a los cambios. Puede incluir un módulo adicional de Inteligencia Artificial si se requiere.



Figura 7. Diagrama de módulos de software que puede contener un videojuego.

Cada uno de estos módulos puede ser creado de forma separada si el proyecto es grande, o si el juego es pequeño o sencillo puede resultar más eficiente el integrar varios o todos en un solo núcleo, disminuyendo el tiempo de desarrollo y/o complejidad.

En el capítulo siguiente se profundiza acerca de cada uno de estos módulos que pueden hacer parte de un motor de juego.

3.6. CONCEPTOS DE GRÁFICOS 3D

3.6.1. Programación de Gráficos 3D

Por gráficos 3D de computador se entiende a los trabajos de arte, diseño, técnicos o de simulación gráficos generados con la ayuda de un computador y software, y que están definidos en el espacio tridimensional, es decir tienen alto, ancho y profundidad. La programación 3D o de gráficos 3D, se realiza usualmente¹⁹ a través de Interfaces de Programación de Aplicaciones (APIs), que proveen un capa intermedia entre los dispositivos gráficos o tarjetas de video y las aplicaciones que se desarrollen, y permite crear, manipular y mostrar gráficos en 3D en el computador. Estos APIs definen unas interfaces para las cuales los fabricantes de tarjetas de video crean implementaciones en hardware y software mediante los controladores.

Hoy en día existen dos APIs principales utilizados para la programación de gráficos 3D: OpenGL²⁰ creado por Silicon Graphics Inc.²¹ ahora manejado por el grupo Khronos²² y Direct3D²³ creado y mantenido por Microsoft. Son los más importantes por su calidad y nivel de utilización, y son usados en la elaboración de todo tipo de aplicaciones con gráficos 3D, como juegos, software educativo, software de simulación, realidad virtual y arquitectura. Existen otros APIs además como Mesa3D y Java3D con características similares, pero Direct3D y OpenGL ofrecen el mejor desempeño. También existen kits de desarrollo gráficos que encapsulan a Direct3D o a OpenGL para simplificar su uso, como Ogre3D, SDL, GLUT, entre otros. Direct3D además hace parte de una

¹⁹ La programación 3d también se puede hacer complemento en software desde cero, pero no es lo normal.

²⁰ OpenGL es la especificación más utilizada en las aplicaciones de software de tiempo real ya que cualquiera puede crea su propia implementación. Existen adaptaciones para casi todos los sistemas operativos de escritorio y el PlayStation 3. <http://www.opengl.org>

²¹ Silicon Graphics Inc. Es una empresa que se especializa en soluciones computacionales de alto rendimiento de software y hardware. <http://www.sgi.com>

²² The Khronos Group es un grupo formado por varias empresas para desarrollar estándares abiertos para aplicaciones multimedia. <http://www.khronos.org>

²³ Direct3D es la parte de gráficos del SDK de multimedia de Microsoft llamado DirectX para las plataformas Windows. <http://www.microsoft.com/directx>

librería completa de programación (Software Development Kit - SDK) multimedia para plataforma Windows llamada DirectX.

3.6.2. Fundamentos de Programación 3D

La programación 3D tiene fundamentos teóricos que aplican de forma similar a todos los APIs 3D, y los que definen algunas reglas básicas para trabajar con ellos. La teoría de gráficos 3D en tiempo real abunda en algoritmos y técnicas para obtener la mejor calidad y el mejor desempeño en sistemas restringidos por tiempo y hardware. Es un campo de la computación que está en constante evolución, y cuyos desarrollos son sucedidos por otros mejores muy rápidamente, en parte por la alta competitividad del mercado de los videojuegos, y el desarrollo de computadores con mayores y mejores capacidades.

Pasar de un sistema 2D a un sistema 3D no es simplemente agregar una tercera coordenada. Por ejemplo en el espacio 2D solo se hacen rotaciones en el eje *xy*, en el espacio 3D se hacen rotaciones en cualquier eje tridimensional. Además de esto, la diferencia entre una escena en 3D y la pantalla en 2D, hace que se le añada complejidad al proceso de dibujado de la escena en la pantalla, ya que se deben especificar muchos más parámetros y procesos.

Sin embargo, aunque sea más complejo, existen unas bases que se utilizan en la mayoría de los casos como fundamento para la programación de gráficos 3D, como la geometría euclíadiana y el espacio euclíadiano tridimensional, sus conceptos básicos como línea, punto, plano, y además los diferentes teoremas de relaciones de ángulos. También se utilizan los vectores, las matrices y las operaciones entre estos para realizar transformaciones a los modelos geométricos.

3.6.3. Las tarjeta de Aceleración 3D

Las tarjetas gráficas con aceleración 3D actuales tienen un procesador dedicado, llamado Unidad de Procesamiento Gráfico (GPU) la cual ejecuta instrucciones de forma independiente a la Unidad Central de Proceso (CPU). La CPU le envía comandos de dibujado al GPU, el cual realiza las operaciones necesarias de dibujado mientras que el CPU sigue realizando otras tareas. Es una operación asíncrona, lo que permite que en los juegos se puedan realizar operaciones intensivas en CPU y además dibujar gráficos de alto requerimiento computacional a la vez.

Una aplicación 3D se comunica con la GPU enviándole comandos a la librería de dibujado, como OpenGL o Direct3D, la cual a su vez le envía comandos a el controlador de la tarjeta. El controlador de la tarjeta le envía los comandos de bajo nivel y la información a la tarjeta de video.

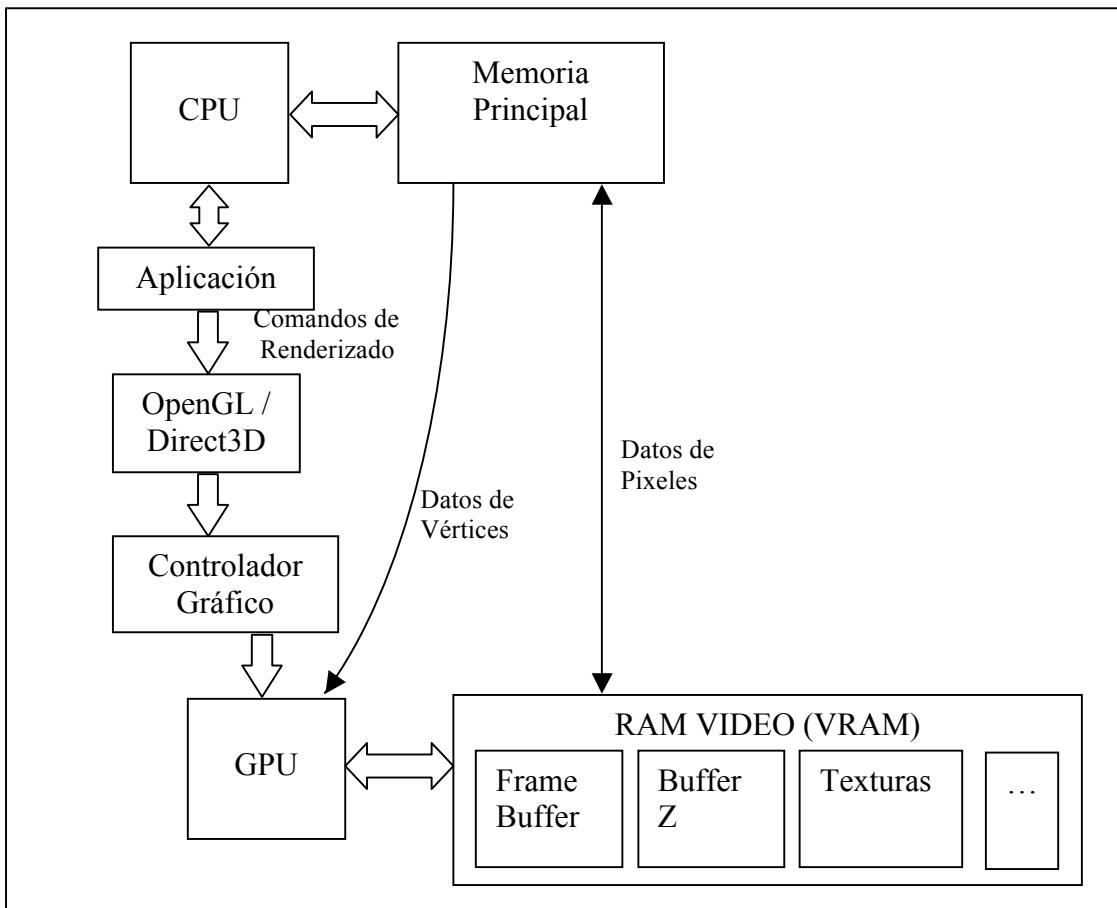


Figura 8. Diagrama de comunicación entre la CPU y la GPU. Fuente. LENGYEL, Eric. Mathematics for 3D game programming and Computer Graphics. Pág. 3.

3.6.4. Modos de Dibujado

Cuando se habla de dibujar en pantalla, se refiere al proceso de poner pixeles en la superficie de la pantalla. El término mas preciso de este concepto es rasterización.

Hay tres modos en que se dibuja en pantalla. Dibujando Puntos, dibujando líneas, y dibujando triángulos. Estos modos corresponden a diferentes arquitecturas de hardware para el dibujado de gráficos en computador.

El modo mas simple es el de dibujar puntos, equivalente a dibujar una imagen en la pantalla, punto por punto. Generalmente se debe dar el color y la posición donde se quiere dibujar en la pantalla. Las imágenes que se dibujan en pantalla como JPG o GIF, son un ejemplo de gráficos que se generan pixel por pixel. Son una matriz de datos de colores rojo, verde y azul en algunos casos (RGB) las cuales definen los colores de un área rectangular de la pantalla, punto por punto.

Aunque se puedan representar como una matriz, la información es en verdad guardada de forma lineal cuando se va a dibujar, es decir, una imagen representada por una matriz NxM de colores con 24bits por color - **RGB** (8bits, 8bits, 8bits) - es en verdad un vector de NxMx3 bytes de longitud en memoria. Este tipo de dibujado da el termino a los gráficos en dos dimensiones o 2D. La mayoría de los juegos de los años ochenta y principios de los noventa todos funcionaban usando este modo de dibujado. Para sacar el máximo de provecho de estos gráficos se desarrollaron muchas técnicas, como definir un color que no será dibujado para mostrar áreas transparentes, transformaciones de tamaño y rotación, y movimiento por capas para simular perspectiva, este ultimo muy utilizado en los side-scrollers (juegos de vista lateral donde el jugador debe ir de izquierda a derecha para completar los objetivos).



Figura 9. Captura del juego 2D Pac-Man.

Fuente: Periodismo escolar en internet <http://www.newsmatic.e-pol.com.ar/>. 2007

Dibujar líneas es una técnica de dispositivos especiales que pueden dibujar vectores. En las pantallas CRT (Tubos de Rayos Catódicos) la pistola de rayos se mueve de izquierda a derecha y de arriba hacia abajo, dibujando un píxel a la vez. Esto requiere que se tenga información sobre todos los puntos a dibujar, es decir, tener la matriz con la información de color de cada uno de los píxeles de la pantalla. Para poder tener escenas en 3D, hay que tomar la información 3D de la escena y convertirla en una imagen 2D, es decir, en la matriz de colores para cada píxel de la pantalla. Este proceso es costoso computacionalmente.

A finales de los años 80s se encontró una solución a este problema, utilizando una pistola para la pantalla que no se movía igual que la pistola de las pantallas CRT, sino que podía dibujar líneas en cualquier orientación en la pantalla. Es decir, podía dibujar a partir de listas de puntos, toda una escena compuesta por objetos tridimensionales. Un ejemplo de esto es el juego Battlezone²⁴ creado por Atari²⁵ en 1980. Su gran ventaja era también su gran limitación, pues no podía 'rellenar' los objetos de color.

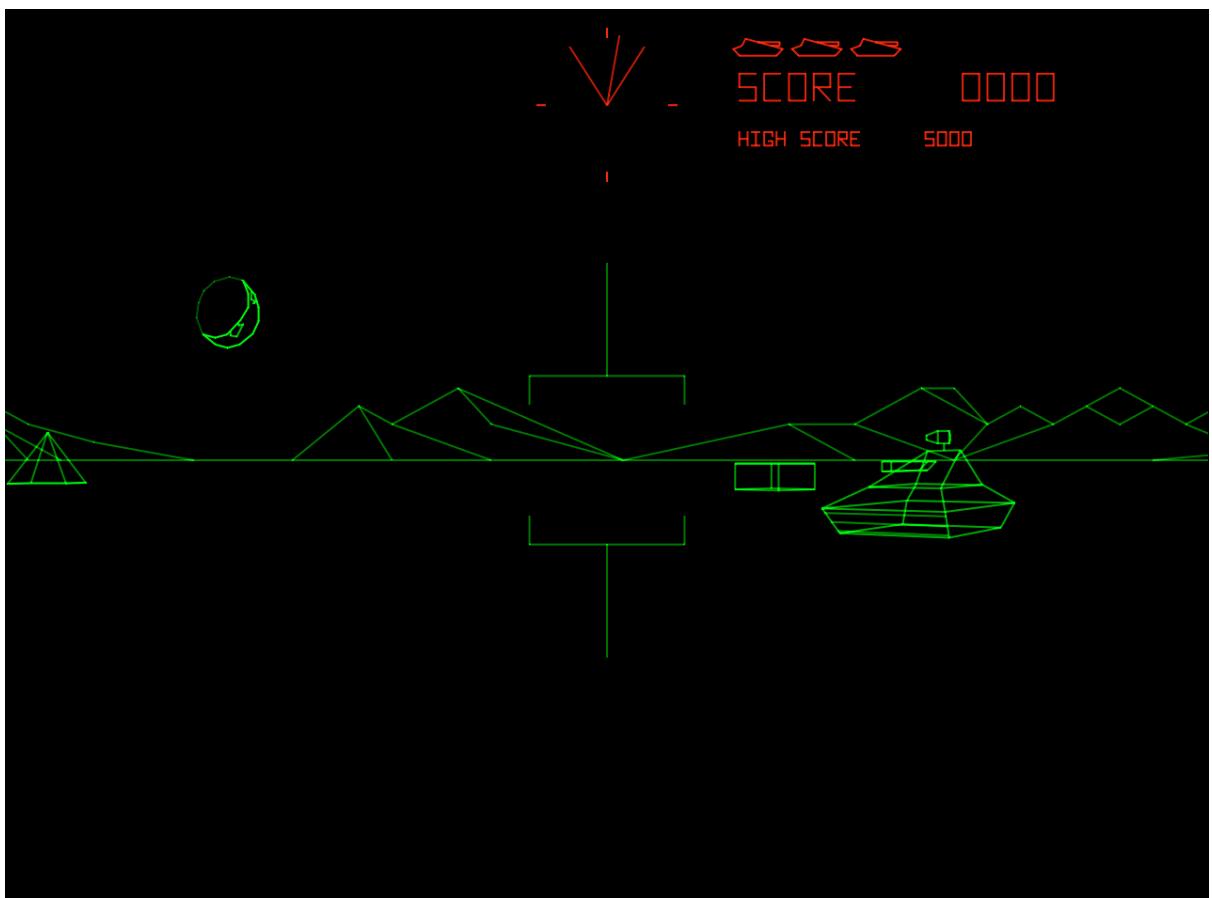


Figura 10. Captura de pantalla del juego Battlezone.

Dibujar triángulos y polígonos corresponde a los métodos usados por las tarjetas gráficas hoy en día para dibujar escenas 3D en la pantalla. Hay que hacer una diferencia entre los métodos usados por el programador y la forma que se dibuja la imagen en la pantalla, la cual sigue siendo

²⁴ Battlezone fue un juego para la consola Atari. Una versión en línea se puede jugar en http://games.atari.com/playgames/arcade/battlezone/battlezone_400.jsp

rasterización. Las pantallas LCD (Liquid Crystal Display) que utilizan una malla de pixeles alimentada por voltaje también dibujan imágenes planas pixel por pixel en la pantalla, así que también se considera como rasterización. Renderizar es generar la imagen 2D a partir de información 3D y rasterizar es dibujar la información 2D (ya sea obtenida o no de una escena tridimensional) en la pantalla.

En este método de dibujado las imágenes se generan a partir de modelos tridimensionales, creados a partir de polígonos, y luego mapeados, o transformados al espacio 2D para ser representados en una pantalla.

Los dispositivos gráficos simples son basados en píxeles, es decir, solo saben dibujar píxeles, pero los dispositivos con capacidades 3D saben recibir modelos 3D y dibujarlos en pantalla. Aunque las salidas del dispositivo son igualmente píxeles, el dispositivo gráfico 3D procesa la información tridimensional para generar esos píxeles, cada vez haciendo uso de mas y mas técnicas. Un dispositivo que no tenga buenas capacidades 3D, no es suficiente para las demandas de capacidades de técnicas de dibujado que utilizan las aplicaciones y juegos modernos.

3.6.5. Sistema de Coordenadas

Existen diferentes sistemas de coordenadas que pueden ser usados para representar objetos en el espacio tridimensional, pero la mayoría de los APIs utilizan el sistema de coordenadas cartesiano, ya sea utilizando la regla de la mano izquierda o derecha. Existen otros sistemas de coordenadas como el esférico, cilíndrico y polar, pero no se usan por que son mas difíciles de programar y para los artistas mas difíciles de entender.

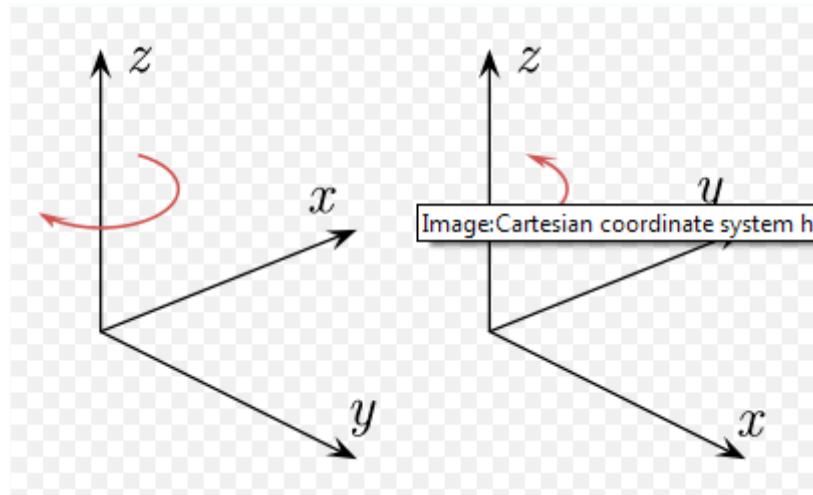


Figura 11. Sistema de Coordenadas cartesiano con la regla de la mano izquierda en la imagen izquierda, y la regla de la mano derecha en la imagen derecha

El sistema de coordenadas cartesiano utiliza tres ejes para definir un punto en el espacio, como una coordenada (x, y, z) . Los tres ejes se intersecan en el punto $(0, 0, 0)$. Cualquier punto antes del cero tiene un valor negativo y después del cero tiene un valor positivo. Se suelen utilizar los vectores \hat{i} , \hat{j} y \hat{k} acompañados de una magnitud para representar un punto en un eje. Es decir, $3\hat{i} + 5\hat{j} - 5\hat{k} = \vec{v}$, es un vector desde el origen $(0, 0, 0)$ hasta $(3, 5, -5)$ y representa un punto en el espacio cartesiano.

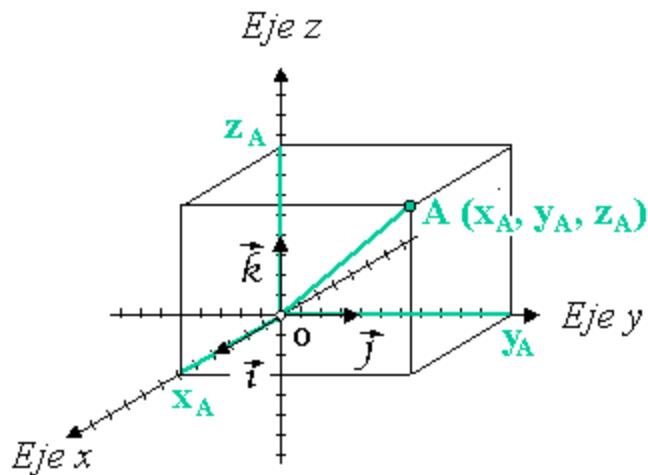


Figura 12. Vector A definido como (x_a, y_a, z_a)

3.6.6. Primitivas

Las primitivas definen los objetos de bajo nivel o entidades atómicas que pueden ser representadas en el espacio. Ejemplos de primitivas son los triángulos, las líneas, las esferas y los puntos. Debido a las limitaciones algunos objetos no pueden ser representados, especialmente aquellos que son infinitesimalmente variables y no están en línea recta, como los círculos o las esferas. En realidad, el hardware gráfico de hoy en día puede mostrar objetos con las siguientes características:

- Un objeto debe ser estar descrito por su superficie, se puede pensar que está vacío.
- Un objeto se define por un grupo de vértices en tres dimensiones.
- Un objeto esta compuesto de polígonos convexos.

De esta forma, una primitiva de dos dimensiones como un círculo debe ser enviada al hardware 3D como una primitiva en tres dimensiones. Además, las superficies curvas como las esferas deben ser aproximadas. Esta condición se debe a la arquitectura del hardware y las transformaciones que deben realizarse para poder dibujar.

Los APIs 3D debido a estas restricciones han definido sus primitivas basados en las capacidades del hardware actual. Las primitivas tanto de Direct3D como de OpenGL son definidas por colecciones de vértices que forman una entidad en 3D. Algunas de las primitivas de Direct3D (sacadas de gda.utp.edu.co) son:

- **Lista de Puntos**

Es una lista que describe vértices en el espacio, uno por uno.

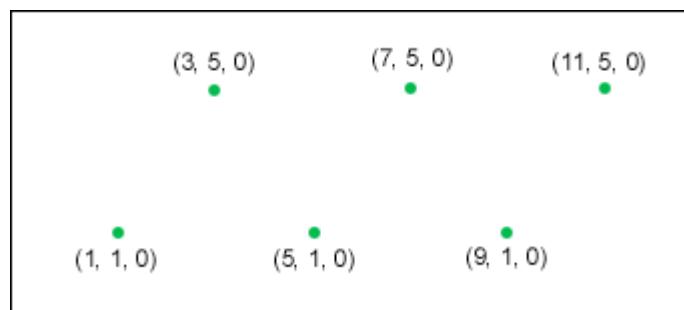


Figura 13. Lista de Puntos

- **Lista de Líneas**

Es una lista de líneas que se definen por parejas de vértices.

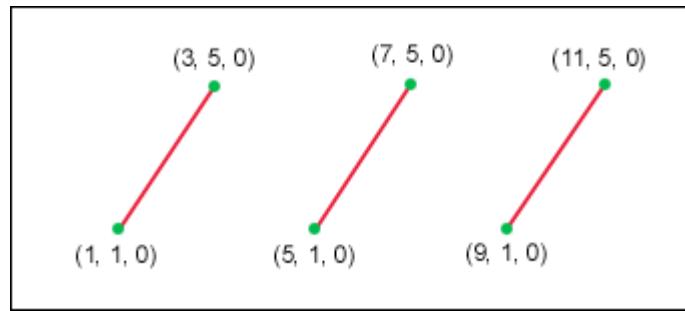


Figura 14. Lista de Lineas

- **Tira de Líneas**

Es una lista de líneas unidas definidas por una lista de vértices.

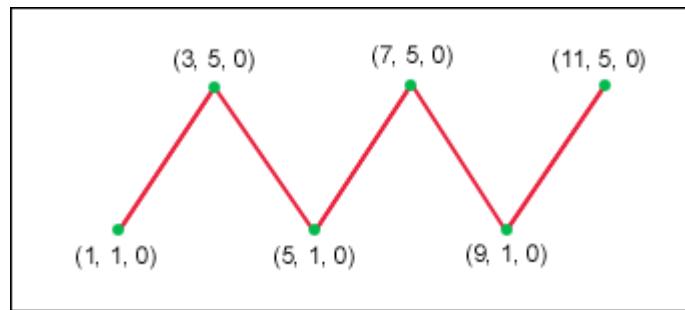


Figura 15. Tira de Lineas

- **Lista de Triángulos**

Es una lista de triplets de vértices. Cada tres puntos definen un triángulo. Cada triángulo es independiente de los demás.

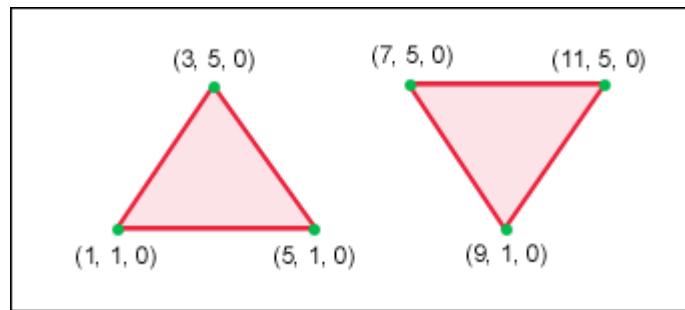


Figura 16. Lista de Triángulos

- **Tira de Triángulos**

Es una lista de triángulos continuos, que comparten lados.

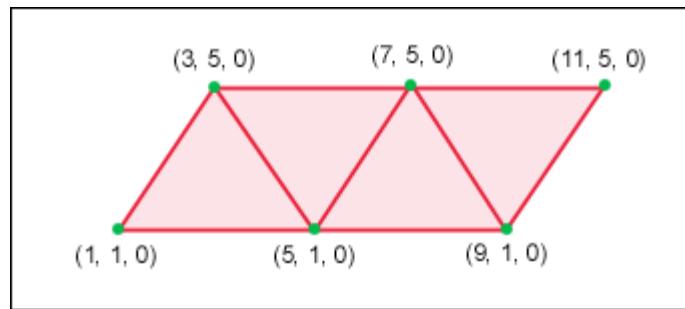


Figura 17. Tira de Triángulos

- **Abanico de Triángulos**

Es una lista de triángulos, donde todos tienen un vértice en común.

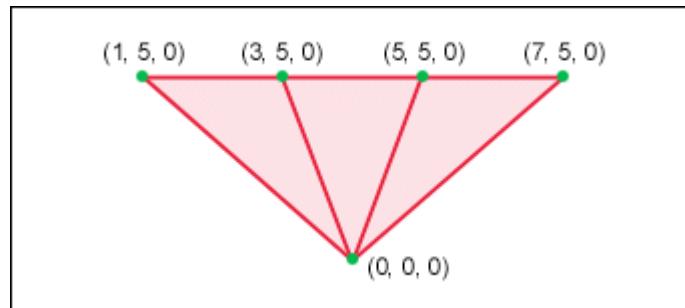


Figura 18. Abanico de Triángulos

3.6.7. Transformaciones geométricas

Las transformaciones geométricas son usadas para mover un punto de una posición a otra posición. Si se aplica a un grupo de puntos que denotan un objeto, entonces se trata de cambiar el objeto no solo de posición, sino también de tamaño y orientación o rotación sobre los ejes cartesianos. Estas transformaciones se pueden simplificar con el uso de matrices. Se puede usar una matriz para escalar, otra para rotar y otra para mover. Si estas matrices se concatenan se puede obtener una sola matriz que realiza las tres operaciones en una sola operación.

Transformación de Traslación

Para mover un vector se debe sumar otro vector con el desplazamiento que se quiere realizar así:

$$\vec{C} = \vec{A} + \vec{B}$$

Por componentes:

$$\vec{C} = (C_x, C_y, C_z) = (A_x, A_y, A_z) + (B_x, B_y, B_z)$$

$$\vec{C} = (C_x, C_y, C_z) = (A_x + B_x, A_y + B_y, A_z + B_z)$$

Usando una matriz, se le añade un cuarto componente A_w al vector:

$$\vec{C} = (A_x \ A_y \ A_z \ A_w = 1) * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ B_x & B_y & B_z & 1 \end{bmatrix}$$

$$\vec{C} = (A_x + 0 + 0 + B_x, A_y + 0 + 0 + B_y, A_z + 0 + 0 + B_z, 1)$$

Eliminando la última coordenada se obtiene el vector desplazado.

Transformación de Rotación

La rotación en el espacio 3D se puede representar de varias formas, una de las mas comunes es a través de las rotaciones en los ejes X, Y y Z. Cada una de estas rotaciones se puede definir por una matriz que rota el vector:

Rotación en el eje X:

$$\vec{C}' = \vec{C} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotación en el eje Y:

$$\vec{C}' = \vec{C} * \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotación en el eje Z:

$$\vec{C}' = \vec{C} * \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Concatenando las matrices se obtiene una sola transformación que se aplica al vector para efectuar una rotación en los ejes X, Y y Z.

$$\vec{C}' = \vec{C} * \begin{bmatrix} \cos\beta\cos\theta & -\cos\beta\sin\theta & \sin\beta & 0 \\ \cos\theta\sin\beta\sin\alpha + \sin\theta\cos\alpha & \cos\theta\cos\alpha + \sin\theta\sin\beta\sin\alpha & -\cos\beta\sin\theta & 0 \\ \sin\theta\sin\alpha - \cos\theta\sin\beta\cos\alpha & \sin\theta\sin\beta\cos\alpha + \sin\alpha\cos\theta & \cos\beta\cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformación de Escala

Para escalar un vector se utiliza la siguiente matriz:

$$\vec{C}' = \vec{C} * \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde S_x, S_y, S_z indican cuanto se va a escalar el vector en cada eje.

3.6.8. El paradigma de renderizado

El renderizado por hardware es basado en polígonos. Es decir, esta sustentado en dibujar primitivas (unidades básicas geométricas) como el triángulo. Hay operaciones para darle color al triángulo y la forma en que varia el color en el triángulo dependiendo del ángulo de incidencia de la luz sobre el. Otra operación importante es el uso del mapeado de texturas, que puede ser usado para colocar texturas (imágenes) en un triángulo para representar mejor la superficie de un objeto.



Figura 19. Modelo 3D con mapas de textura.

Para pasar la información de los vértices de la escena a la tarjeta de video se utilizan diferentes estructuras de datos. OpenGL tiene mas estructuras definidas que Direct3D, pero generalmente no son necesarias para la mayoría de aplicaciones.

3.6.9. El canal de dibujado

En el proceso de dibujar una escena tridimensional en la pantalla del computador, se utilizan diferentes sistemas de coordenadas, diferentes transformaciones entre estos sistemas de coordenadas y proyecciones. Los sistemas de coordenadas mas usados son:

- Coordenadas 3D de Modelado (de Objeto).
- Coordenadas 3D de Mundo (Coordenadas Globales)
- Coordenadas 3D desde el punto de Vista (Cámara)
- Coordenadas Normalizadas de Proyección
- Coordenadas 2D del dispositivo

Entre estos sistemas de coordenadas se usan las siguientes transformaciones:

- Transformación de Modelado
- Transformación de Mundo
- Transformación de Vista
- Transformación de Proyección

El canal de dibujado comienza con los datos de los vértices enviados por la aplicación. Generalmente la información de los vértices es enviada usando arreglos de vértices ordenados

como algún tipo de primitiva y que además tienen otros datos, como la posición del vértice, el color, el vector normal a la superficie donde está el vértice usado para la iluminación, la coordenada de textura o cualquier otra información que se requiera para la operación de dibujado.

Tres vértices componen un triángulo. Antes de que un triángulo pueda ser dibujado en la pantalla, cada vértice del triángulo debe ser procesado. Esto se debe a que la información geométrica enviada al hardware gráfico está en el contexto de un espacio tridimensional. Uno de los trabajos de la tarjeta de video es transformar la información 3D en información 2D que puede ser dibujada en la pantalla. Tradicionalmente el término con el que se conoce este proceso es transformación, realizada por el Fixed Function Pipeline (FFP) o camino de dibujado fijo, los algoritmos predeterminados del dispositivo gráfico, y al cual uno le puede definir algunos parámetros para que realice su labor con diferentes resultados. Una de sus deficiencias es que una vez enviados los datos de los vértices al dispositivo no se podían hacer otras transformaciones en la mitad del proceso. Por esto se desarrolló la Programmable Pipeline o camino de dibujado programable, el cual permite definir cualquier algoritmo para transformar los vértices utilizando shaders, y el nombre del proceso de transformación se conoce como vertex shading.

Los shaders son programas que se ejecutan en la GPU. Un shader de vértices procesa la información de los vértices, es decir realiza la transformación de los vértices. El nombre transformación viene de transformación de la posición de los vértices usando una matriz. Esta tarea todavía se realiza en la mayoría de los shaders de vértices, pero también pueden realizar muchas otras tareas. Esta fue la razón primaria de la creación de los shaders, la infinidad de posibilidades de modificación de la información de la escena. Gracias a esto y al poder de procesamiento de las GPUs, se realizan otras tareas usando shaders como simulaciones y cálculos científicos. Esta aplicación de los GPU se llama programación de propósito general de las GPU (GPGPU).

Como se enunció previamente, hay diferentes sistemas de coordenadas que se usan para dar flexibilidad tanto al diseñador como al programador, y que además son necesarios para la transformación de un objeto que está en una escena en una imagen 2D. Cuando se envía la información geométrica también se envían las matrices que convierten una escena 3D en una imagen plana posible de ser presentada en la pantalla. Una vez se aplica cada una de estas matrices se habla de un distinto espacio en el cual se encuentra la información. Es decir, al aplicar una matriz de transformación, la información del modelo cambia de sistema de coordenadas. Se utilizan matrices porque permiten de forma rápida mover, rotar y cambiar el tamaño de un vértice, y además se pueden combinar entre ellas lo que permite en una sola matriz realizar varios cálculos, lo que a su vez permite tener sistemas jerárquicos de transformaciones donde hay hijos

dependiendo de la posición del parente. Esto se ve por ejemplo en sistemas de esqueletos, donde la mano esta pegada (es hija) al antebrazo, el antebrazo al brazo, y el brazo al cuerpo. Si se mueve el cuerpo se debe mover todos los miembros pegados a él. Si se hiciera paso a paso, la mano debería ser rotada y movida un poco por cada matriz de sus antecesores, pero si en cambio se multiplican todas las matrices se obtiene una sola matriz que hace el mismo trabajo. Esto permite por ejemplo animar un personaje compuesto por decenas de huesos, y hacer que lleve en su mano por ejemplo una espada.

En una escena 3D, los objetos generalmente se crean uno por uno de forma separada. Por ejemplo, en un juego de fútbol no hay forma de que el diseñador pueda tener un modelo para cada posible estado del juego. La solución es crear los modelos de cada uno de los jugadores de forma separada, cada uno en su propio sistema de coordenadas. Este sistema de coordenadas de cada modelo se llama espacio de modelo o espacio de objeto. Cuando se colocan los jugadores en la cancha de fútbol, el escenario del partido de fútbol tendrá un sistema de coordenadas global para todos los objetos llamado espacio de mundo, y cada jugador será ubicado respecto a este sistema de coordenadas en una posición determinada. De esta forma en la aplicación se mantiene información por objeto que lo posiciona, escala, y rota en el mundo.

Antes de que un objeto sea dibujado, debe posicionarse también respecto al ojo del observador, que en Direct3D y OpenGL se representa con una cámara. Así los objetos se ubican dependiendo del lugar hacia donde el jugador esté mirando en el espacio 3D del juego o aplicación. Este espacio se llama espacio de cámara o espacio de vista.

Una vez se han transformados los vértices de los modelos desde el punto de vista del jugador, se les añade perspectiva y proyectan a una plano 2D. La perspectiva es la reducción del tamaño de los objetos con la distancia y se realiza a través de la matriz de proyección. El espacio luego de aplicar esta operación se llama espacio de proyección normalizado. Después de esta transformación, la posición del vértice tiene valores x , y , y z , donde x y y indican la posición en espacio homogéneo $[-1, 1]$ y z indica la profundidad.

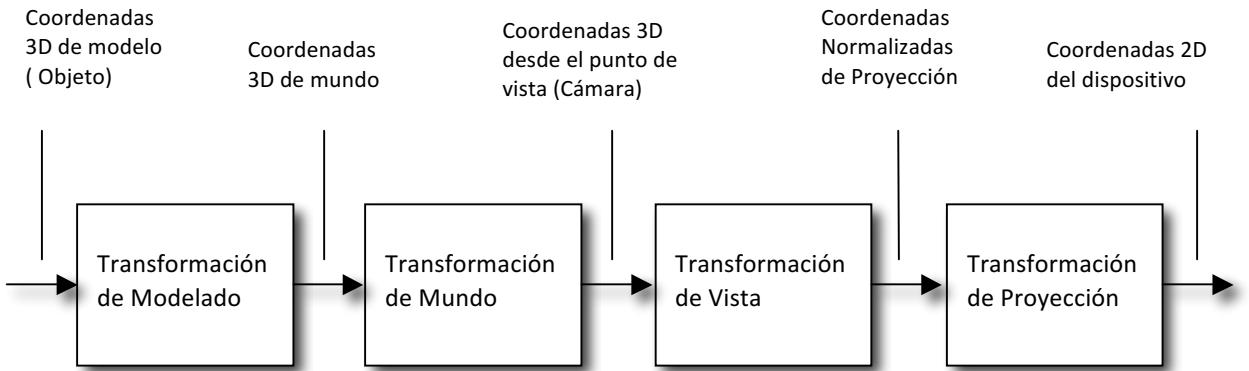


Figura 20. Proceso de transformación de los vértices.

Cuando los tres vértices han sido procesados por el shader de vértices o por la tarjeta si se usa el Fixed Function Pipeline el triángulo esta listo para ser dibujado en pantalla. En el proceso de dibujado del triángulo se debe cubrir con el color del triángulo los pixeles de la pantalla que cubre. Para cada pixel que este dentro del triángulo se procesa una unidad llamada fragmento. Un fragmento es básicamente un pixel que esta siendo procesado. Cuando se procesa pasa al framebuffer, el espacio en memoria donde se guardan los pixeles de una pantalla.

Un fragmento es procesado por un shader de fragmento si se usa el canal de dibujado programable (Programable Pipeline). Sino por los algoritmos del dispositivo de hardware. En esta etapa se calcula el color del fragmento, el cual puede ser un color proveniente de una coordenada de una textura, modulado con un cálculo de iluminación o cualquier cálculo que se quiera realizar en el shader sobre el color, y puede también opcionalmente cambiarse su profundidad. Normalmente cuando la profundidad se deja sin cambiar, este valor se deriva del valor z proveniente del shader o etapa de vértices. Esta profundidad es luego comparada con el valor para ese pixel en el buffer de profundidad o z-buffer. El buffer de profundidad es un buffer del tamaño de la pantalla que contiene los valores de profundidad de los vértices escritos previamente en el mismo lugar de la pantalla. Se usa para determinar si un fragmento esta detrás o delante de un pixel escrito previamente. Así se puede determinar la visibilidad del fragmento, y si resulta que es visible (esta por delante del pixel escrito anteriormente, o no había ningún pixel en esa posición), entonces es escrito en el framebuffer, y el buffer de profundidad se actualiza con el valor de profundidad del pixel recién escrito. Si esta detrás del anterior, el fragmento es descartado. Opcionalmente, el pixel puede ser dibujado sin tomar en cuenta el valor del buffer de profundidad, o en vez de reemplazar el que ya esta en esa posición, se pueden mezclar los colores. Esto es lo que se utiliza por ejemplo para la transparencia.

Estos pasos de dibujado se aplican para todas las primitivas de las que esta compuesta la escena, lo cual produce como resultado la imagen en el framebuffer. Este buffer sera puesto luego en la pantalla, y el ciclo se puede repetir continuamente para producir mas cuadros o pantallazos. Si se realiza muchas veces por segundo se puede entonces hablar de gráficos 3D en tiempo real.

3.6.10. Texturizado

El texturizado o mapeado de texturas es el proceso en el que se le añaden detalles a las superficies de los objetos 3D utilizando imágenes. En este proceso se realiza una acción similar a envolver un objeto con papel de regalo. En la imagen se puede entender mejor el efecto del mapeado de texturas:

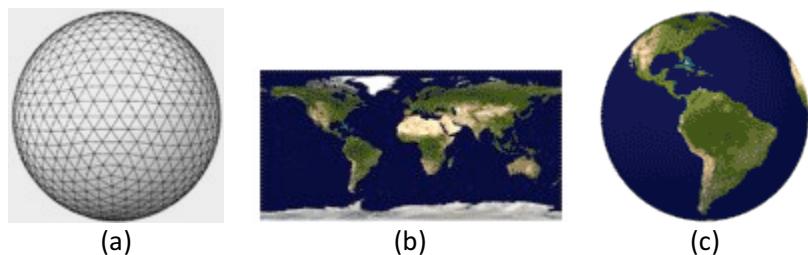
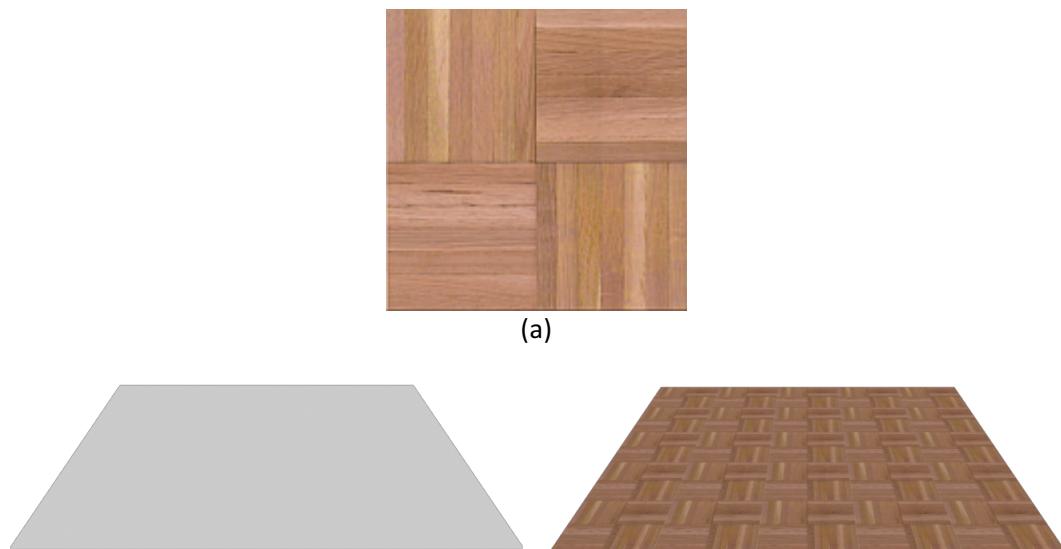


Figura 21. La imagen (a) corresponde a una esfera sin textura, (b) la textura a aplicar, (c) es la esfera con la textura aplicada.

Para cada triángulo en la esfera, corresponde una sección de la imagen. Hay casos en que la misma imagen se puede repetir varias veces en un triángulo, por ejemplo, un piso de madera:



(b)

(c)

Figura 22. (a) Textura de madera. (b) Plano formado por 2 triángulos. (c) Plano formado por dos triángulos con la textura repetida 25 veces.

Una textura no tiene que ser necesariamente un archivo de imagen, también puede ser una imagen creada en la marcha usando un algoritmo (ej. ruido aleatorio), almacenada en la estructura para texturas específica del API gráfico que se este usando.

3.6.11. Mapas

El mapeado requiere que se especifique para cada vértice que parte de la imagen se va a utilizar. Se utilizan dos coordenadas, u y v , cada una indica en qué posición de la textura esta localizado el vértice. Para cada punto en el espacio tridimensional se debe encontrar el punto correspondiente en dos dimensiones en la textura.

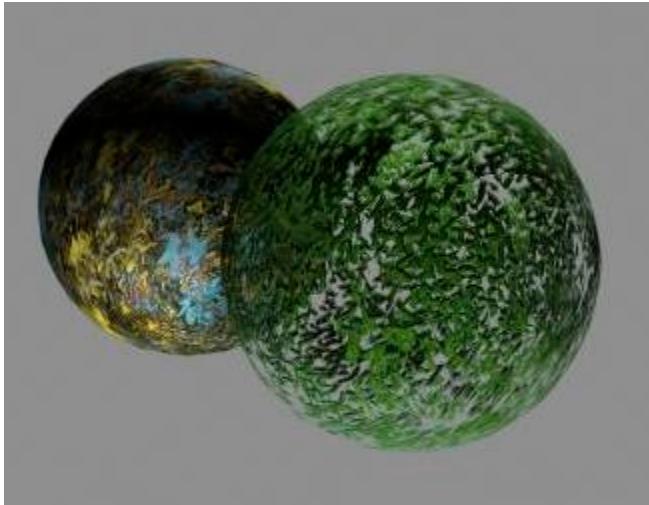


Figura 23. Dos esferas mapeadas con texturas. Fuente: Siggraph

3.6.12. Filtrado

Si el triángulo es más grande que la textura, se debe realizar una aproximación o filtrado para obtener los pixeles que faltan, las usuales son: más cercano (**nearest**), **bilineal**, **trilineal** y **anisotrópico**. Si al contrario el triángulo es mas pequeño se debe buscar el pixel mas representativo.



Figura 24. La imagen del centro es la original, las de la izquierda y derecha son las usadas por el filtro isotrópico para diferentes ángulos desde donde se observa.

Fuente: Advanced Graphics Programming Techniques Using OpenGL

3.6.13. Iluminación

En los juegos existe la posibilidad de tener dos tipos de iluminación: estática o dinámica. Ambas pueden coexistir sin interferirse con adecuada programación.

La iluminación estática es la iluminación pre-calculada y es idéntica en cada ejecución del juego. Este tipo de iluminación es muy económica en términos de procesamiento en tiempo real, porque no hay que calcularla para usarla. La iluminación estática se crea a partir de la información de las luces en un nivel y la geometría del nivel. A partir de esta se calcula la cantidad de luz que cada entidad de luz le está agregando a cada triángulo.

Una técnica de luz estática muy utilizada son los mapas de luz (**lightmaps**). Los mapas de luz son imágenes en escalas de grises que al ser moduladas o multiplicado su color con el color del triángulo se obtiene una imagen con sombras. Este método de iluminación estática se llama **lightmapping** y permitió crear escenas con más realismo cuando las tarjetas de video no tenían el poder de iluminar dinámicamente y de forma completa una escena.



Figura 25. En el gráfico se puede observar como una textura de una pared es modulada por una textura de luz, lo que produce un efecto de luz y sombra aplicada sobre la pared.

Este tipo de iluminación fue muy común en juegos como Quake y Unreal, y crearon un gran interés por los juegos tipo FPS o First Person Shooter (Juegos de Disparar en Primera Persona). Uno de los problemas de esta técnica es que hay que calcular la iluminación antes de poder ver el resultado, lo cual puede llevar varias horas o días para completar, y que limita en tiempo a los artistas durante la creación de contenido para los juegos. Por otro lado una de las grandes ventajas es que se puede utilizar cualquier tipo de luz (no solo las fijadas por la iluminación en tiempo real) y cualquier método para calcular la luz, porque no importa el tiempo en que se calcule.

Iluminación Dinámica

La iluminación dinámica se calcula para cada cuadro del juego. Esta iluminación utiliza la geometría de las escenas y la información de las luces para dar el efecto de iluminación. La iluminación dinámica es más costosa de calcular, porque debe hacerse en el método más simple para cada triángulo de la escena, y para cada luz por triángulo. Es decir, una escena con 100.000 triángulos y 3 luces, tendría que hacer 300.000 cálculos de luz. Este costo de GPU tan alto hace que las luces dinámicas se utilicen moderadamente en los juegos. Sin embargo, los artistas han podido crear utilizando una combinación de pocas luces efectos casi-reales. Hoy en día, gracias al desarrollo de los shaders (programas que se ejecutan en la GPU), también los programadores tienen mas libertad de crear su propias técnicas de iluminación y variar el nivel derealismo con que se calcula la iluminación, permitiendo flexibilidad en los requerimientos de hardware para la ejecución.

Tipos de Luz

Las fuentes de luz estándares para OpenGL y Direct3D son las luces direccionales, luces focales y las luces puntuales.

Luces Direccionales.

Esta fuente envía rayos paralelos de luz hacia todos los vértices de la escena. Se supone que la fuente de luz esta tan lejos que los rayos llegan con el mismo ángulo de incidencia. El ejemplo que siempre se usa de este tipo de fuente de luz es el Sol, esta tan lejos de la Tierra que se puede decir que se puede ver en un lugar que todos los objetos reciben la misma cantidad de luz y en el mismo ángulo.

Luces Puntuales

Las luces puntuales son el equivalente a un bombillo. Son puntos en el espacio donde se emite luz en todas las direcciones.

Luces Focales

Las luces focales simulan el efecto de un foco de teatro. Emite luz dentro de un espacio en forma de cono.

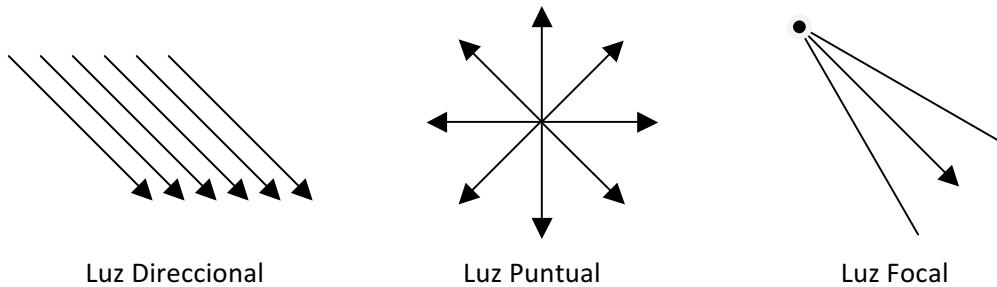


Figura 26. Tipos comunes de luces

Estos tres tipos de luces son aproximaciones simples de las luces en el mundo, y su combinación puede servir para obtener diversos efectos.

3.6.14. Modelos de Iluminación

Se han creado varios modelos para simular la luz en el mundo. Estos modelos varían su costo de aplicación, y en el efecto que producen. Algunos de ellos son, clasificados por costo computacional:

- Iluminación Blinn-Phong²⁶
- Iluminación Phong²⁷
- Iluminación Cook-Torrence²⁸
- Ray tracing.
- Photon Mapping

Los modelos que brindan la mejor calidad son Photon Mapping y Raytracing. Sin embargo son también los más costosos de calcular. El estándar usado en los juegos siempre había sido Blinn-Phong hasta el desarrollo del pipeline programable. En este documento se especifican en detalle los modelos Phong y Blinn-Phong de iluminación los cuales son los más usados en los videojuegos por su rendimiento.

²⁶ Blinn, James F. Models of light reflection for computer synthesized pictures. ACM Press. 1977

²⁷ Phong, Bui Tuong. Illumination for computer-generated images. ACM Press. 1973.

²⁸ Torrance, K. E. y Sparrow, E. M. Theory for off-specular reflection from roughened surfaces. 1967

Modelo Phong

El modelo Phong es un modelo de iluminación creado por Bui Tuong Phong en 1973. Es una simplificación de la ecuación de renderizado la cual describe el flujo de la energía lumínica por una escena. Este modelo es una simplificación porque es un modelo de iluminación local, es decir no toma en cuenta los reflejos de otros objetos, y además, porque divide la iluminación en 3 partes muy fáciles de manejar, ambiente, difuso y especular (brillos).

El modelo funciona definiendo (usualmente como valores RGB) el valor ambiente, difuso y especular para cada fuente de luz, y de igual forma se crean materiales para los objetos en la escena, los cuales definen cuanta luz ambiente, difusa y especular refleja cada material. Para este modelo se tienen en cuenta la posición de la luz, la posición del observador, la normal a la superficie siendo iluminada, y los atributos de la luz y el material de la superficie.

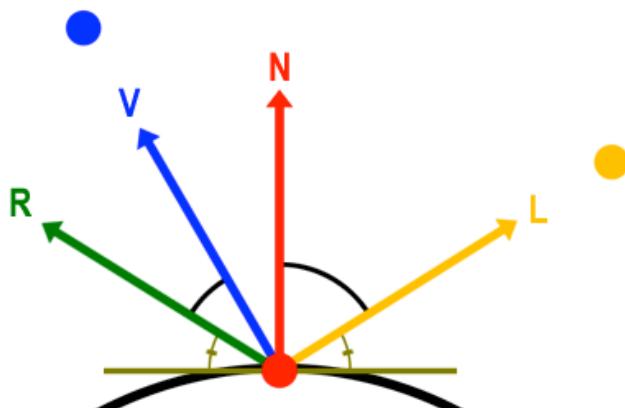


Figura 27. Vectores que del modelo usados Phong de iluminacion.

El valor ambiente representa la luz que es añadida a la escena sin importar la posición ni la orientación de los objetos o el observador. Su objetivo es hacer una contribución falsa de los reflejos indirectos, valores que no se pueden calcular al ser un modelo local. Este valor se utiliza usualmente para evitar que las sombras sean absolutamente negras, lo cual se ve irreal.

El valor difuso representa la luz que golpea una superficie y se reparte de igual forma en todas las direcciones. Su intensidad depende del ángulo al cual la luz llega a la superficie, es decir, el ángulo que se forma entre el vector de luz y la normal a la superficie. Esto significa decir que no importa donde se pare el observador el punto siempre tendrá el mismo color. Entre mas grande sea el ángulo, menos brillante aparecerá la luz. Si el ángulo es mayor a 90 grados esto significa que la luz esta detrás de la superficie y no debe dar ninguna luz a ese punto.

El valor espectral es la luz que se refleja en una dirección particular. Es el efecto que se obtiene cuando se mira un bombillo usando un espejo. Por esto, los valores especulares dependen de hacia donde este mirando el observador. Su intensidad depende del ángulo entre el vector de luz y el vector del observador. El valor espectral utiliza un exponente α que indica como el brillo se reparte por la superficie. Entre mas grande sea el valor de α , mas brillante se considera la superficie.

La ecuación que define el modelo Phong de iluminación para un punto en la superficie es:

$$I_p = k_a i_a + \sum_{luzes} (k_d (L \cdot N) i_d + k_s (R \cdot V)^\alpha i_s)$$

Donde:

i_a , i_d e i_s son los componentes ambiente, difuso y especulares de la luz

k_a , k_d y k_s son los valores ambiente, difuso y espectral.

α es el exponente, el cual define como se extiende el brillo sobre la superficie. Valores pequeños crea brillos repartidos por toda la superficie, valores grandes crea brillos pequeños.

N es la normal normalizada a la superficie del vértice.

L es el vector normalizado desde el punto hasta la fuente de luz.

R es el vector normalizado hacia donde se reflejaría la luz.

V es el vector normalizado desde el punto hasta el observador.

La luz ambiente es un valor constante que solo se suma una vez a la iluminación de la superficie, pero los valores difusos y especulares se calculan y se suman para cada luz en la escena. Es decir, la luz es aditiva.

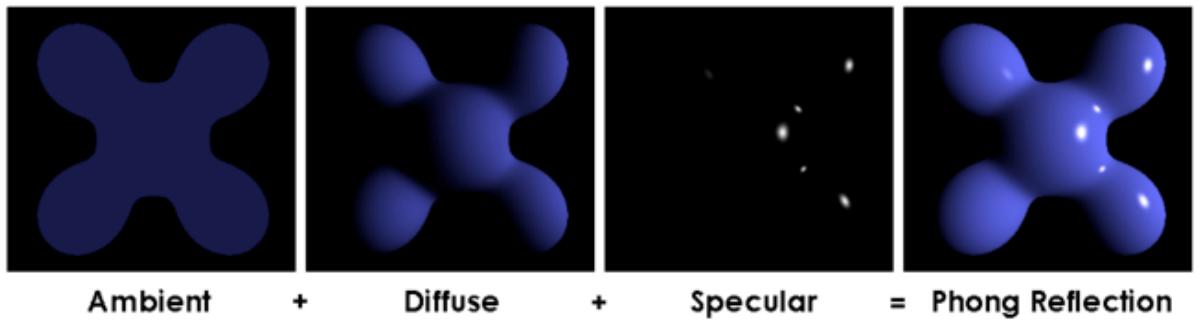


Figura 28. Imagen donde se muestra cada uno de los colores que se calculan en la ecuación de Phong y el resultado final. Fuente: http://en.wikipedia.org/wiki/Blinn%20Phong_shading_model

El resultado que se obtiene indica el color que tiene un punto en la superficie del modelo.

Este modelo es dinámico porque para cada cuadro se deben calcular repetidamente los mismos valores. De esta forma también se pueden cambiar las propiedades de las luces y conseguir muchos efectos, como luces intermitentes, luces que se mueven, luces que cambian de color, y de igual forma efectos para los objetos.

Modelo Blinn-Phong

El modelo Blinn-Phong es una modificación del modelo Phong original, que lo hace más eficiente. El problema del método de Phong es que el vector R es muy costoso de calcular:

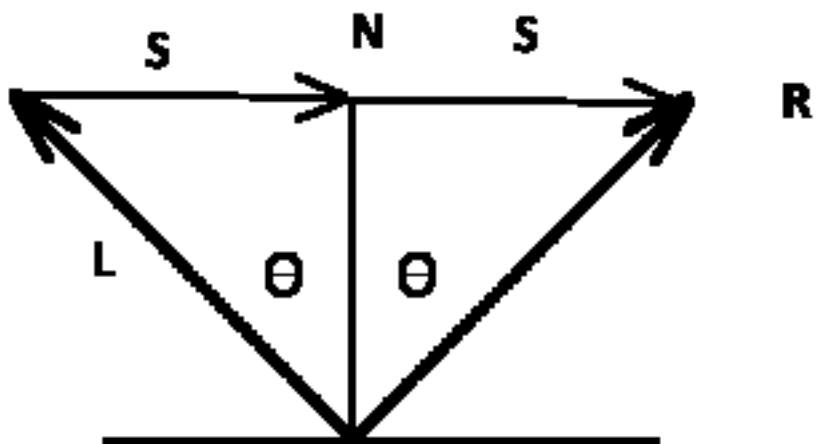


Figura 29. Vectores del modelo Phong.

$$R = N \cos(\theta) + S$$

$$L + S = N \cos(\theta)$$

$$S = N \cos(\theta) - L$$

$$R = 2N \cos(\theta) - L = 2N(N \cdot L) - L$$

Jim Blinn hizo un cambio pensando en que los brillos del modelo Phong dependen del ángulo que se forma entre la luz de reflejo y el observador. El introdujo un nuevo vector llamado el vector intermedio o halfway vector, el cual es la suma de los vectores V y L dividida por 2.

$$H = (L + V)/2$$

Reemplazando $R \cdot V = \cos(\theta)$ por $N \cdot H = \cos(\theta/2)$ se obtiene una relación similar aunque no idéntica. Sin embargo se puede modificar el valor α para que el resultado se acerque al del modelo Phong. De esta forma se pueden simplificar los cálculos, reduciendo el número de instrucciones requeridas:

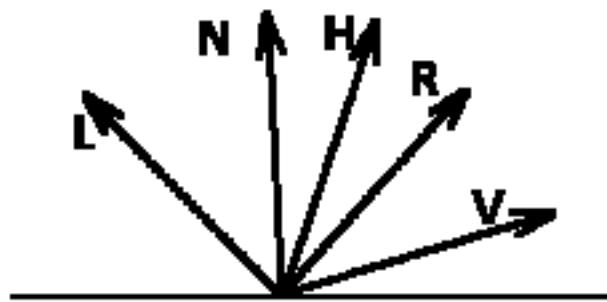


Figura 30. El vector H de Blinn se aproxima al vector R de Phong.

La ecuación del modelo Blinn-Phong seria:

$$I_p = k_a i_a + \sum_{luces} (k_d (L \cdot N) i_d + k_s (N \cdot L)^{\alpha} i_s)$$

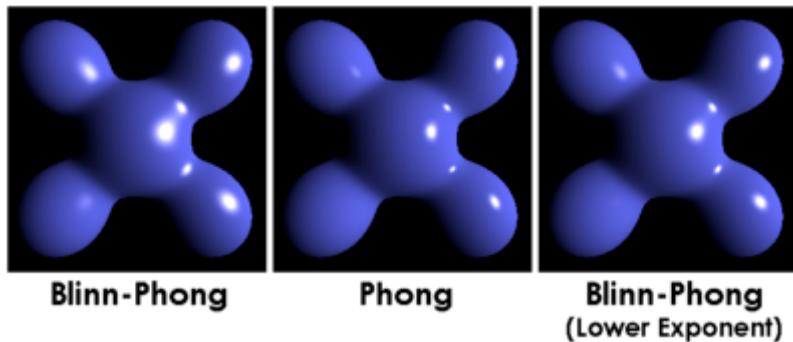


Figura 31. Comparación resultados obtenidos con los modelos Blinn-Phong, Phong y Blinn-Phong con el valor α ajustado. Fuente: http://en.wikipedia.org/wiki/Blinn-Phong_shading_model

3.6.15. Shading o Coloreado

Shading es el proceso de llenar de color un triángulo definido por tres vértices o puntos en el espacio. Utilizando el modelo Phong se puede calcular la iluminación en un punto cualquiera en el espacio definido por un vector de posición y un vector con la normal a la superficie, además de la orientación del observador y la posición y parámetros de la luz.

Sin embargo para un triángulo solo se tienen tres vértices con información que representa a una superficie que puede ocupar cientos de píxeles en la pantalla. Es decir, para darle color a toda la superficie dentro del triángulo solo se tienen tres posiciones y tres normales (o puede ser solo una normal para el triángulo entero) y cientos o miles de puntos. Para poder hacer una iluminación obteniendo normales para cualquier punto en la superficie se han desarrollado varias técnicas que permiten sombrear cualquier punto en un triángulo. Estas son:

Coloreado Plano (Flat)

En este sombreado se calcula la iluminación Phong de igual forma para toda la cara. Es decir se utiliza una sola normal y un solo vértice del triángulo para calcular la intensidad de luz y se obtiene un resultado como este:

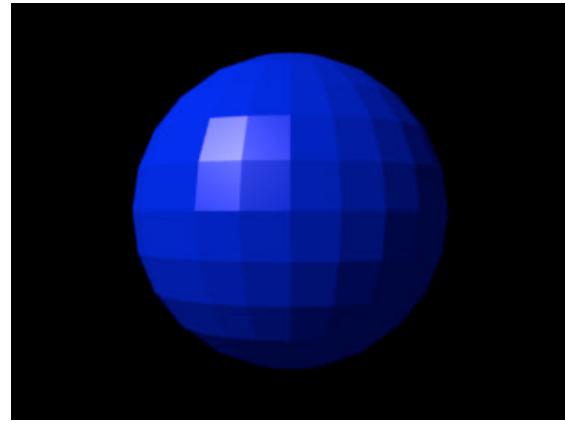


Figura 32. Esfera renderizada con coloreado plano

Coloreado Gouraud

El sombreado Gouraud es una interpolación publicada por Henry Gouraud en 1971. Con este método se puede calcular la iluminación Phong en los tres vértices del triángulo y se interpola linealmente con el método Gouraud el color reflejado para obtener la iluminación de un punto cualquiera en el triángulo. Ofrece mejores resultados que el sombreado plano. Sin embargo tiende a producir errores cuando los reflejos o brillos se producen en la mitad de los triángulos y no cerca de los vértices pues no son calculados. Los sombreados Flat y Gouraud son los que soportan los dispositivos gráficos 3D por defecto en el Fixed Function Pipeline.

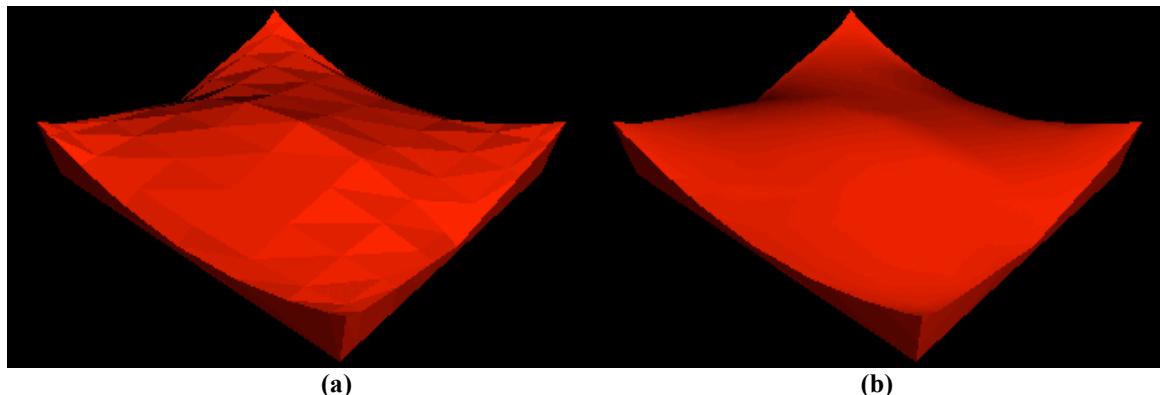


Figura 33. En la figura se puede ver la mejora que produce renderizar usando Gouraud (b) y coloreado plano (a).

Coloreado Phong

En el sombreado Phong se interpolan los parámetros del modelo para cada pixel: vector a la luz, vector hacia al observador y la normal, y se calcula la intensidad de luz reflejada. Es el que brinda los mejores resultados, sin embargo también es el más costoso computacionalmente porque generalmente hay más pixeles que vértices. El sombrado Phong solo se puede realizar con shaders.

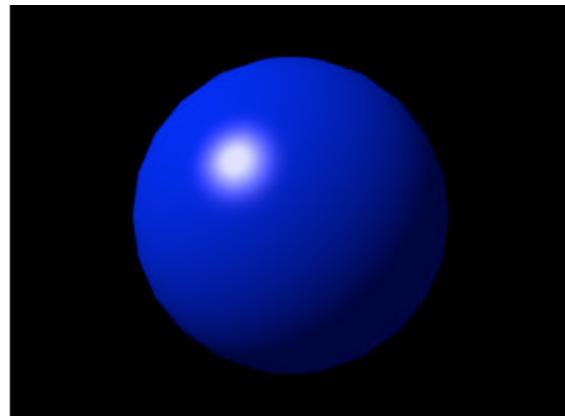


Figura 34. Esfera renderizada con coloreado Phong

Coloreado usando Normal Mapping

El Normal Mapping o mapeado de normales es una técnica relativamente nueva en su aplicación en la cual la información de las normales en cada punto esta definida por una textura. De esta forma cada punto del triángulo tiene un archivo donde se especifica la normal en ese punto. Es una técnica que da resultados muy exactos y de forma muy eficiente pues reduce el número de polígonos necesarios para modelos resultados de alta calidad, sin embargo genera más trabajo para los artistas, ya que la textura de normales debe crearse durante el modelado. El Normal Mapping solo se puede usar con shaders.

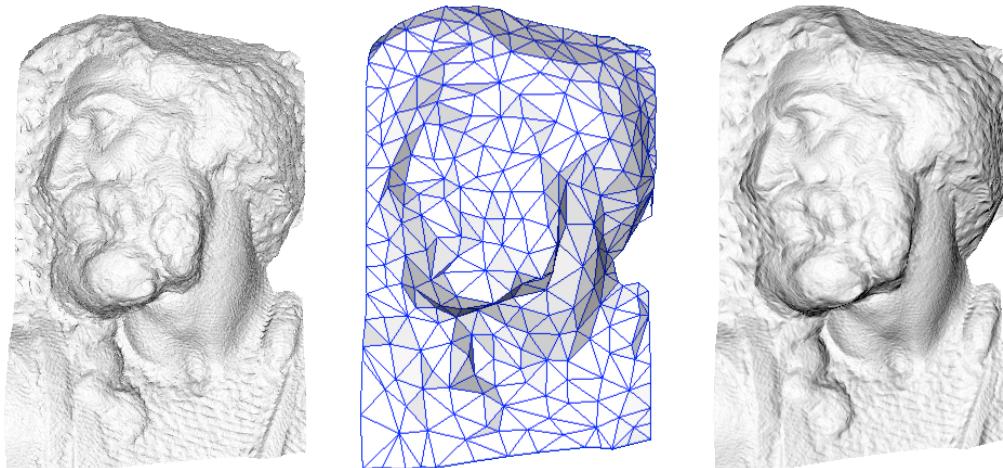


Figura 35. Estatua renderizada con normal mapping. La estatua de la izquierda tiene 4 millones de triángulos. La estatua de la derecha utiliza normal mapping, es casi idéntica a la de la izquierda y solo tiene 500 triángulos. Fuente: http://en.wikipedia.org/wiki/Normal_mapping

3.6.16. Otros Efectos

Además de la geometría, las texturas aplicadas sobre la geometría y la iluminación, existen además otros efectos que suelen ser utilizados para añadir realismo y gracia a la escena. Estos efectos se obtienen con diferentes técnicas que están fuera del alcance de este documento sin embargo se pueden enumerar algunos de ellos:

- Sombras dinámicas.
- Sistemas de Partículas como Polvo y chispas.
- Cielo dinámico.
- Niebla.
- Blur.
- Visión de Calor, Visión Nocturna.

3.7. MOTOR GRÁFICO

El módulo de gráficos (o motor gráfico) es la parte del software encargada de mostrar el juego en la pantalla y poder así proveer al jugador con la información para que interactúe con el sistema de juego. En juegos de mesa, el motor gráfico haría el papel de mostrarnos todo el material que viene con el juego, como el tablero, las fichas, las tarjetas de premios y castigos, los dados, etc. En deportes, el motor gráfico estaría mostrando los jugadores, las canchas, los balones e

instrumentos deportivos. Pero existe un valor agregado que lo diferencia de los juegos del mundo real, y es su posibilidad de representar mundos que no existen, de cambiar o quitar y agregar otras reglas diferentes a las del mundo real, permitiendo que por ejemplo los jugadores de un partido de fútbol sean animales, o que se realice un viaje por el espacio a planetas distantes para rescatar a una princesa alienígena. Muchos juegos encuentran su nicho de venta en proveer los mejores gráficos, utilizando motores de gráficos de última generación.

Los motores gráficos hoy en día funcionan en su mayoría en 3D. Todavía existen en algunas plataformas como los celulares, las PDAs, y las consolas portátiles juegos que solo se muestran en 2 dimensiones. También existen juegos, cada vez mas pocos, que utilizan una visión isométrica del juego para dar una apariencia 3D, utilizando imágenes planas. Un ejemplo de este tipo de motor gráfico es el juego The Sims²⁹ en su versión original.

En su forma más simple un motor gráfico es un conjunto de tipos y funciones o métodos (librería) de alto nivel destinados al manejo eficiente y en tiempo real de gráficos 3D. El objetivo de un motor gráfico es permitir al programador hacer operaciones muy potentes y eficientes de la forma más fácil posible.

Características básicas que se esperan de un motor de gráficos 3d actual:

- Cargar y dibujar objetos 3D con materiales y efectos, en pantalla.
- Permitir la fácil integración con otros subsistemas del videojuego como sonido, física, etc.
- Ser multiplataforma.
- Implementar un sistema de materiales que simplifique el manejo de texturas y programas de la tarjeta de video en métodos y estructuras más simples de usar.
- Ser autosuficiente en la recolección de basura y re-uso de objetos.
- Hacer un manejo de las escenas para optimizar el renderizado.
- Utilizar técnicas de animación variadas, encapsuladas con métodos/funciones para uso simplificado.
- Calcular y dibujar sombras dinámicamente.
- Hacer uso de luces.
- Proveer métodos para usar archivos de imagen en diferentes formatos.
- Permitir configurar la calidad de los gráficos dinámicamente.

²⁹ The Sims. <http://thesims.ea.com/>

3.7.1. Algunos objetos característicos de los motores gráficos

Entidad. Es un objeto móvil el cual contiene un modelo 3d, y su información geométrica como los vértices y triángulos. Una entidad es usualmente un objeto cargado de un archivo.

Nodo. Es una representación abstracta de un punto en la escena. Se usan nodos para facilitar el manejo de objetos 3d en la escena, cada entidad esta asociada a un nodo generalmente, el cual define la posición, orientación y escala de este en la escena o respecto a su padre. Se suele utilizar jerarquía de nodos, donde un nodo puede tener varios hijos, pero un solo padre.

Escena. Una escena se considera un conjunto de entidades posicionadas en el espacio, que comparten unos atributos comunes. Generalmente se usan escenas para definir contenedores de entidades y nodos, y atributos que se aplican a todos los elementos por igual, como la luz de ambiente.

Cámara. Es un objeto abstracto que define un punto de vista de un observador, y el cual se usa para renderizar una escena. Algunas de las propiedades que suele definir son la posición, el área donde se va a renderizar y desde donde y hasta donde queremos ver la escena.

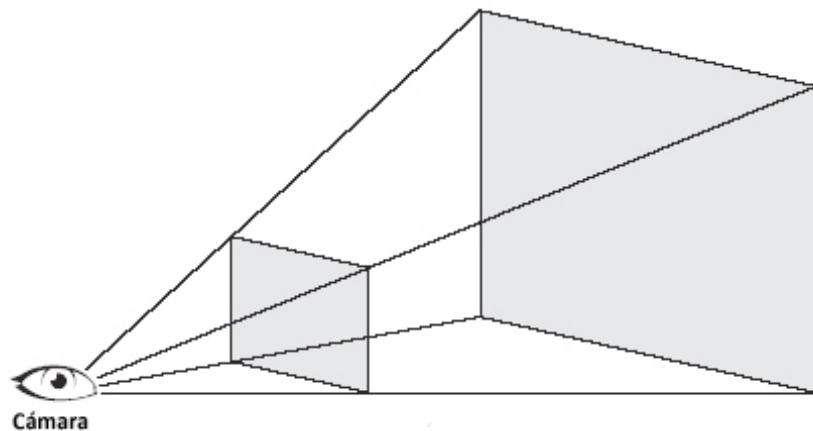


Figura 36. Dibujo de la cámara y los planos límite cercano y lejana que limitan el dibujado.

Vectores. La forma más fácil de representar en un punto en el espacio es por medio de vectores, para esto el motor gráfico provee clases para el manejo de vectores 2d o 3d.

Quaterniones. Los quaterniones son un elemento común de algunos motores gráficos, debido a que son una forma más efectiva de representar las rotaciones. Un quaternion esta compuesto de cuatro componentes: un vector con coordenadas **x**, **y**, **z**, y un ángulo de rotación **w**. El Motor gráfico Ogre3d utiliza quaterniones en vez de vectores. Los quaterniones son muy útiles pues guardan la misma información de orientación, posición y escala, que una matriz 4x4 con tan solo cuatro valores, a diferencia de los 16 de la matriz.

3.7.2. Animación

El motor Gráfico debe permitir la importación de modelos 3d, para su renderización, estos modelos pueden venir acompañados de animaciones y el motor debe poder incorporar estas animaciones y permitir su fácil uso. Hay dos técnicas comunes de animación que generalmente se implementan en los motores gráficos: una es animación usando esqueletos y la otra es animación por vértices.

Animación por esqueleto

La animación por esqueleto es una técnica que funciona de forma similar a la forma en que se mueven los seres vertebrados. En esta se definen una serie de “huesos” que tiene asociados unos vértices que hacen parte del modelo del objeto, y que son movidos conforme se mueve el hueso.

Cada hueso tiene una posición, tamaño y orientación en el espacio local. Opcionalmente puede tener hijos o un padre. El padre de un hueso puede tener a su vez un parente y así sucesivamente, creando una jerarquía. La posición y orientación de un vértice están dadas por la conjugación de la posición y orientación suya con la de su parente. Ya que el esqueleto utiliza esta jerarquía, se suele tener un solo hueso “raíz”, desde el cual se empiezan a calcular todas las posiciones y orientaciones de los huesos “hijos”, para finalmente actualizar los vértices. Este hueso es también el que permite mover y rotar todo el objeto desde un punto común.

Dependiendo de las capacidades del sistema de animación de esqueletos del motor gráfico, un vértice puede estar relacionado con varios huesos a la vez usando “pesos”, es decir un número que indica la incidencia de un hueso en la posición y orientación final del vértice. La ventana principal de las animaciones con huesos es que las animaciones se calculan para los huesos, y luego se actualizan las posiciones de los vértices respecto a los huesos, lo que reduce el cálculo de las animaciones enormemente.

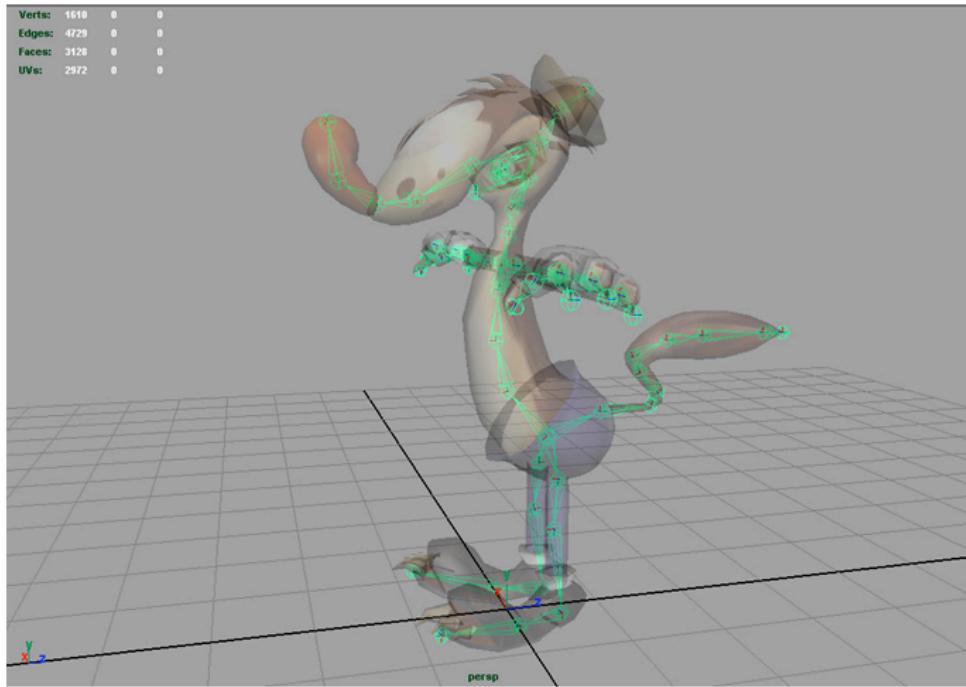


Figura 37. Modelo 3D con esqueleto

Animación por vértice

Existen ocasiones en que la animación usando esqueletos, aunque muy eficiente, se hace muy difícil de usar porque los objetos no tienen una forma en la que encajen los huesos o sus movimientos no se pueden realizar satisfactoriamente. Un ejemplo de estos es la animación facial, en que los músculos de la cara no se pueden cambiar por huesos fácilmente.

Para esto existe entonces la animación por vértices, en inglés **morph animation**, en la cual la posición de cada vértice se guarda para cada posición el objeto a animar. Cada posición del objeto guardada se conoce como un keyframe o cuadro clave, y define una pose del objeto. La tarea del sistema de animación, es calcular la posición de los vértices entre pose y pose a medida que transcurre el tiempo. Se puede pensar que es similar a la forma como se crean los dibujos animados por televisión, cuadro a cuadro, aunque en este caso no se necesitan dibujar todos los cuadros intermedios entre posición y posición, sino solo los que el artista considere necesario, dependiendo del método que se utilice para calcular la posición de los vértices entre keyframes. En algunos casos no se calculan posiciones intermedias, sino que solo se tienen unas posiciones fijas y se itera por ellas.



Figura 38. Cara animada animación por vértices. Fuente:
http://www.bigworldtech.cn/technology/client_animation_en.php. 2007

Este método le permite al artista más control sobre la animación, ya que puede mover cada vértice individualmente a la posición que desee. La desventaja de este método, es que en un modelo muy complejo la animación de la totalidad de los vértices es una tarea computacionalmente costosa. En los juegos se suelen utilizar este tipo de animaciones en animaciones de la cara, tejidos, y objetos que cambian de un estado de animación a otro sin posiciones intermedias.

3.7.3. Librerías

El área de los motores gráficos es una de las más populares, así que podemos encontrar cientos de librerías disponibles. Las siguientes son algunas de las librerías Open Source o gratuitas para crear aplicaciones con gráficos 3D más utilizadas y más robustas, y con comunidades activas que soportan su desarrollo.

OpenSceneGraph³⁰

Open Scene Graph es una librería Open Source de alto rendimiento multiplataforma, que permite crear aplicaciones 3D. Su propósito específico es la facilidad de uso, eficiencias, extensibilidad y estabilidad.

Irrlicht³¹

Como especifica en su sitio Web, Irrlicht es un motor de gráfico 3d en tiempo real, multiplataforma escrita en C++. Es una librería poderoso para crear aplicaciones 2D y 3D como juegos y visualizaciones científicas. Viene con una documentación excelente e integra todas las

³⁰ OSG Open Scene Graph, <http://www.openscenegraph.org>

³¹ Irrlicht Engine, <http://irrlicht.sourceforge.net>

características de última tecnología para la representación visual como sombras, sistemas de partículas, animación de personajes, tecnologías interiores-exterioras y detección de colisiones. Todo esto es accesible a través de una interfaz C++.

Ogre3D³²

OGRE (Object-Oriented Graphics Rendering Engine) es un motor 3D, orientado a escenas escrito en C++ diseñado para hacer más fácil e intuitivo para desarrolladores la producción de aplicaciones que utilicen gráficos 3D con aceleración por hardware. La librería de clases abstrae todos los detalles de utilización de las librerías del sistema como Direct3D y OpenGL y provee una interfaz basada en objetos del mundo y otras clases intuitivas.

***Crystal Space*³³**

Crystal Space es un SDK gratuito multiplataforma para el desarrollo de aplicaciones 3D escrito en C++, en particular juegos. El proyecto está formado por varios sub-proyectos, Crystal Space el cual es el motor de renderizado, CEL (Cristal Entinty Engine) que añade manejo de entidades para crear juegos, CELStart el cual es un motor de juego completo programable usando scripts y Apricot como un proyecto para Blender.

³² OGRE Object Oriented Graphics Engine <http://www.ogre3d.org>

³³ Crystal Space <http://www.crytalspace3d.org>

3.7.4. Comparación de las librerías Gráficas

CARACTERÍSTICAS	OGRE	Open Scene Graph	Irrlicht	Crystal Space
Arquitectura de Plug-in	Sí	Sí	Sí	
Implementación para Directx	Sí	No	Sí	No
Implementación para OpenGL	Sí	Sí	Sí	Sí
Iluminación dinámica	Sí	Sí	Sí	Sí
Sombras	Sí	Sí	Sí	Sí
Vertex Shaders	Sí	Sí	Sí	Sí
Pixel Shader	Sí	Sí	Sí	Sí
Manejador de escenas	Sí	Sí	Sí	Sí
Animaciones con esqueletos	Sí	Sí	Sí	Sí
Animaciones con vértices	Sí	Sí	Sí	Sí
Sistema de partículas	Sí	Sí	Sí	Sí
Sistema de GUI	Sí	No	Sí	No
Sistema de materiales	Sí	Sí	Sí	Sí
Manejador de recursos	Sí	Sí	Sí	Sí
Editor	No	No	Sí	No
Foro activo / Wiki	Sí	Sí	Sí	Sí
Usado comercialmente	Sí	Sí	Sí	Sí
Soporte actual de los creadores	Sí	Sí	Sí	Sí
En continuo desarrollo	Sí	Sí	Sí	Sí
Tutoriales de los desarrolladores	Sí	Sí	Sí	Sí
Plataformas	Sí	Sí	Sí	Sí
Código abierto	Sí	Sí	Sí	Sí
Libros	Sí	No	No	No
Plataformas	- Windows - Linux - Mac OSX	- Windows - Linux - Mac OSX	- Windows - Linux - Mac OSX	- Windows - Linux - Mac OSX
Compiladores	- Visual C++ .NET - Visual C++ .NET 2003 and Visual C++ 2005	- Microsoft Visual Studio - Mingw - Cygwin	- GCC 3+ - Code::Blocks - Microsoft Visual C++ 6	- GCC 3+ - Code::Blocks - Microsoft Visual C++ 6

	-  GCC 3+		- Microsoft Visual Studio Express 2005	- Microsoft Visual Studio Express 2005
Fecha de Inicio	1999	1999	2002	
Desarrolladores	8	1 + 264 Contribuyentes	6	63 (algunos contribuyentes)
Licencia	LGPL	LGPL	Zlib/libpng	LPGL
Ultima Version	1.4.4	2.0	1.3.1	10

3.8. MOTOR FÍSICO

El Motor Físico es un elemento del Motor de Juego que ha ido adquiriendo más importancia y notoriedad en los últimos años. Esto se debe a la evolución de los procesadores que han permitido realizar operaciones más complejas en el corto periodo de un pantallazo del juego. El Motor Físico promedio de un videojuego realiza la simulación de los modelos físicos newtonianos como la velocidad, masa, fricción, resistencia del viento, inercia, etc. Puede simular y predecir, usando el tiempo como medida de cambio, los efectos sobre los objetos bajo diferentes condiciones para generar un comportamiento similar al del mundo real, o variando parámetros el de un mundo de fantasía.

3.8.1. La Física en los videojuegos

La mayoría de los videojuegos dan una representación de la realidad, y por esto día a día se busca que el videojuego tenga elementos gráficos, sonoros y físicos, que permitan crear un mundo virtual. Aun en muchos de los juegos que no imitan el mundo real, se manejan las leyes físicas del mundo real para crear comportamientos que puedan ser mas fácilmente entendidos por el jugador.

La imitación de los fenómenos físicos es uno de los aspectos más importantes en un videojuego, debido a que hacen parte de nuestro vivir diario. Cosas como la gravedad, la inercia, el movimiento de fluidos, la velocidad, la aceleración etc., son fenómenos que aprendemos a reconocer y usar de forma innata, y al verlos reflejados en un videojuego generan en nuestro cerebro una sensación de realidad haciéndolo mucho más inmersivo.

3.8.2. Propiedades del motor

Un motor físico es un software el cual usa variables como masa, velocidad, fricción, resistencia, para simular y predecir fenómenos físicos en diferentes condiciones que pueden ser configuradas para imitar el comportamiento en la tierra o adaptarlo para crear efectos distintos, en un mundo virtual.

“La ilusión y la experiencia inmersiva de un mundo virtual, cuidadosamente construido con modelos de muchos polígonos, texturas detalladas, e iluminación avanzada, es frecuentemente rota tan pronto el objeto comienza a moverse e interactuar”³⁴, esto es debido a que son tantos los elementos y propiedades físicas que existen en un escena y deben ser configurados de forma correcta para tener una simulación consistente con la realidad, de caso contrario la física puede ocasionar que el videojuego pierda su ilusión de realismo.

“Todo es interactividad e inmersión”³⁵

Algunos ejemplos de elementos físicos aportan realismo a juegos:

- Colisiones entre objetos.
- Trayectorias de cohetes.
- Efectos de gravedad para objetos como planetas.
- Estabilidad en autos de carreras.
- Dinámica de vehículos de agua.
- Fluidos.
- Efectos de magnetismo.

Estos son algunos ejemplos, pero cualquier cosa en un videojuego que se mueve, vuele, ruede, o que no sea un objeto estático puede ser modelado realísticamente para crear un comportamiento real en el videojuego.

³⁴ Gary Powel, creador del motor MathEngine Plc.

³⁵ Steven Collins, CEO de Havok.com

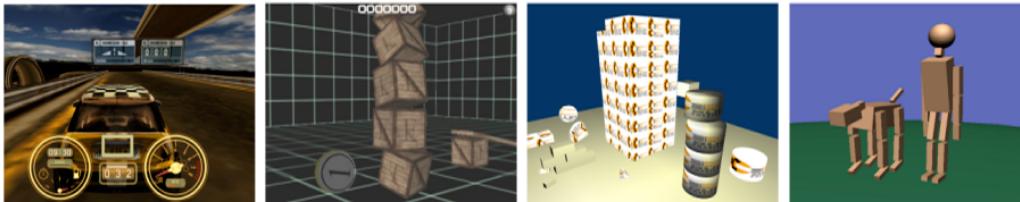


Figura 39. Ejemplos de aplicaciones que usan simulación física.

Dependiendo de las necesidades de un videojuego la implementación de la física será diferente. Algunos juegos como juegos de carreras, se enfocan en la estabilidad del auto, mientras que otros juegos buscan la interacción del personaje con el mundo.

Conceptos Importantes

Los siguientes son algunos conceptos que manejan la mayoría de los motores físicos, pueden variar sus nombres y algunos estar mezclados en la variedad librerías físicas existentes, pero en general representan eventos y objetos físicos que se utilizan en los motores que aplican la física newtoniana.

Mundo. El mundo es donde se lleva a cabo la simulación física del videojuego, solo los elementos que se encuentran en el mundo físico serán afectados por la física.

Escenas. Dentro del mundo se tienen las escenas, pueden ser consideradas como dimensiones dentro del mundo. Cada escena puede tener configuraciones diferentes, los objetos de una escena no interactúan con los de otras escenas.

Cuerpos. Dentro de las escenas, se encuentran objetos que representan un cuerpo físico completo, los cuales se ven afectados por la simulación física, y que reaccionan ante elementos como fuerza, fricción, velocidad, inercia etc. Estos objetos son generalmente unidos a entidades del motor gráfico, de tal forma que al actualizar su posición y rotación debido a las interacciones físicas, se actualizan la posición y rotación de las entidades gráficas asociadas con ellos.

Cada cuerpo tiene una o varias formas usadas para detectar las colisiones. Existen diferentes tipos de formas. Entre ellas se encuentran: Esfera, cápsula, caja y cono. Los motores físicos definen métodos o funciones que evalúan las colisiones entre cada uno de estos tipos, aunque estos se hacen de forma interna.

En algunos casos se puede utilizar la misma malla del objeto gráfico, pero hay que ser cuidadosos con el número de triángulos de la malla. Además es posible que la forma del objeto gráfico sea demasiado irregular para que se pueda calcular bien una colisión con otro objeto.

Cada vez que se realice una colisión entre estos objetos, el motor físico informará que ha ocurrido una colisión entre los dos cuerpos dueños de estos objetos de colisión, para así poder crear una acción.

Algunas formas típicas de los motores físicos:

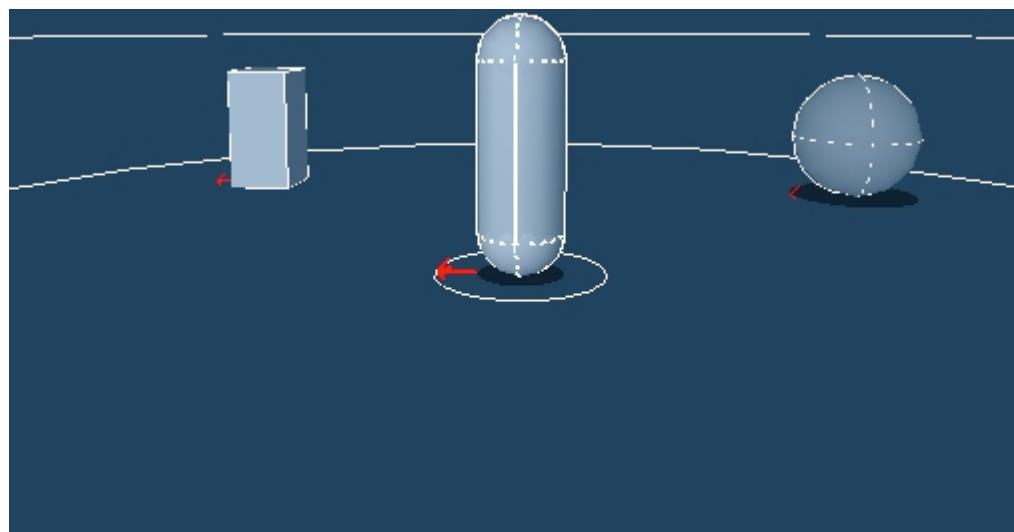


Figura 40. Formas usadas en las colisiones. Fuente: Tutoriales AGEIA PhysX

Colisiones. Si los objetos en el juego tienen formas complicadas, crear colisiones reales puede ser complicado. No en todos los juegos se necesita precisión física, para esto entonces se puede reducir la complejidad usando las formas básicas. Se debe reducir el cálculo para el motor físico, para esto debemos deshabilitar las colisiones entre objetos que nunca colisionar o que no se encuentren presentes.

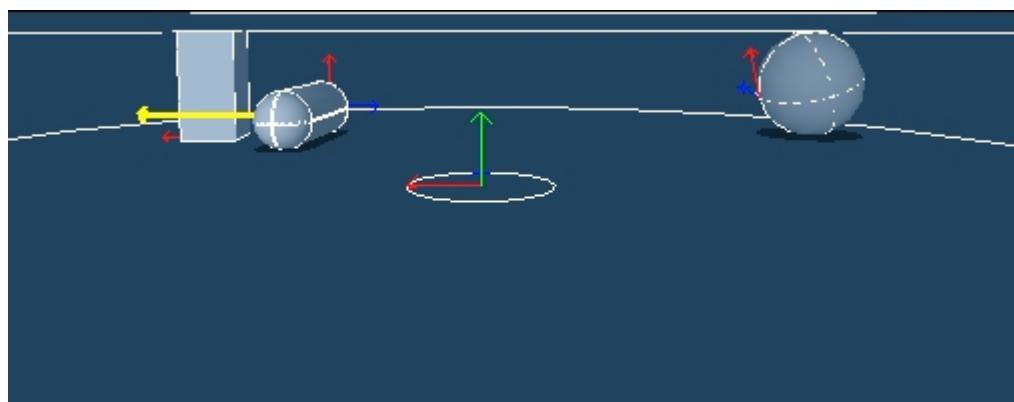


Figura 41. Colisión entre objetos. Fuente: Tutoriales AGEIA PhysX

La figura muestra la colisión entre el cilindro y la caja. La flecha amarilla indica que al cilindro le están aplicando una fuerza hacia la izquierda y por esto se ha movido en ese sentido.

Para reducir el cálculo el motor físico puede hacer distinción de objetos, por ejemplo:

- Cuerpos dinámicos
- Cuerpos estáticos

Esto se hace debido a que para los cuerpos estáticos no es necesario calcular todas las propiedades físicas, y así se optimiza el funcionamiento del motor.

Grupos de objetos de colisión

Además de poderse definir si un objeto es estático o no, también se pueden agrupan los objetos en conjuntos o grupos de colisión. Estos grupos permiten establecer filtros de colisión, los cuales pueden ser configurados para eliminar colisiones entre grupos de objetos, o por el contrario habilitarlas.

Propiedades físicas

A continuación se enuncian algunas propiedades de un motor físico

- Densidad
- Gravedad
- Fricción
- Sistemas de materiales
- Suavidad
- Elasticidad
- Velocidad
- Aceleración
- Fuerza
- Torque
- Inercia
- Centro de masa

Sistemas de respuesta

Una de las partes más importantes que un motor físico brinda, es poder obtener información de los objetos que están colisionados, para esto los motores proveen métodos/funciones que nos dan información sobre la colisión como, punto de contacto, velocidad, normal del contacto, posición y

así determinar una acción. El motor reportan las colisiones entre los objetos y los parámetros de colisión: Algunos parámetros de la colisión que se reportan usualmente son:

- Normal al punto de colisión.
- Vector que define la longitud y dirección de la penetración de un cuerpo en el otro.
- Parámetros físicos de los dos objetos en el momento de colisión, como la fuerza, velocidad, etc.
- Si es una colisión con una malla de triángulos, puede devolver información relacionada con ese triángulo, como su material físico.

Disparadores

Un disparador o trigger es un elemento en la escena representado por una forma física como caja o capsula, que se encuentra en la escena pero no se muestra, permitiendo que los objetos físicos pasen a través de disparador sin crear colisión, pero generando un evento cuando el objeto entra en el, se sale, o se encuentra dentro de el. Es muy útil para determinar cuando el personaje esta en un lugar determinado, y así generar eventos del juego, como sonidos, explosiones, recompensas, etc.

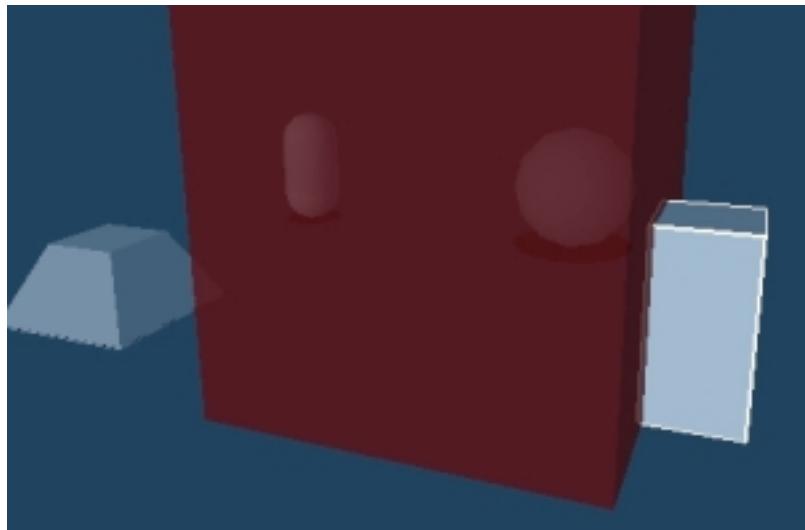


Figura 42. La caja roja es un disparador que es atravesado por un objeto de colisión de color azul en la imagen. Fuente: Tutoriales AGEIA PhysX

Control de personajes

En muchos motores físicos para mover un objeto se debe hacer a través de fuerzas o impulsos, aplicando una fuerza en al objeto, esto puede generar inconvenientes para el programador, ya que es difícil predecir con exactitud el efecto de una fuerza sobre un objeto en la simulación, cuando

existen otros parámetros usados, como por ejemplo, la fricción, la velocidad angular y el momento angular. Por esto los motores físicos suelen implementar métodos o funciones específicas para controlar personajes, que permiten manipular un objeto en una escena simulando el comportamiento de un personaje con mayor control sobre la simulación.

Algunas de las características, que tiene el control de personaje de Ageia PhysX para el control de personajes son.

1. Control estándar FPS, el personaje se mueve con un vector de desplazamiento y no a través de fuerzas o impulsos.
2. Estabilidad perfecta
3. Controlado a través de cajas o capsulas.
4. Se puede configurar para que suba escaleras, o escale.
5. Se puede limitar la pendiente máxima por la cual puede subir un personaje.
6. Interacciones con objetos, el personaje puede empujar objetos.
7. Diferentes sistemas de respuesta: se inicializa un evento diferente cuando toca un objeto u otro personaje.
8. Se puede cambiar la forma del objeto de colisión en tiempo de ejecución.

Permite un gran nivel de control sobre los personajes en comparación con objetos dinámicos, y manteniendo la influencia de la física de la escena.

Control de vehículos

Algunos motores físicos incorporan librerías para crear vehículos, las cuales proporcionan elementos que permiten simular comportamientos de movimiento de vehículos, por ejemplo llantas, suspensión y frenos.

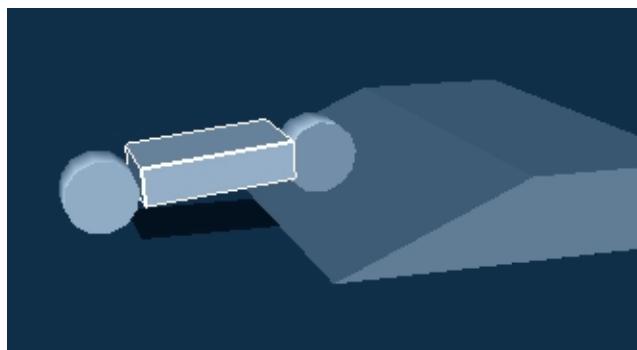


Figura 43. Simulación de un vehículo de dos ejes Fuente: Tutoriales SDK de AGEIA PhysX

Uniones

Son objetos que conectan dos cuerpos creando así un movimiento de uno a través del otro. Por ejemplo una bisagra o articulación.

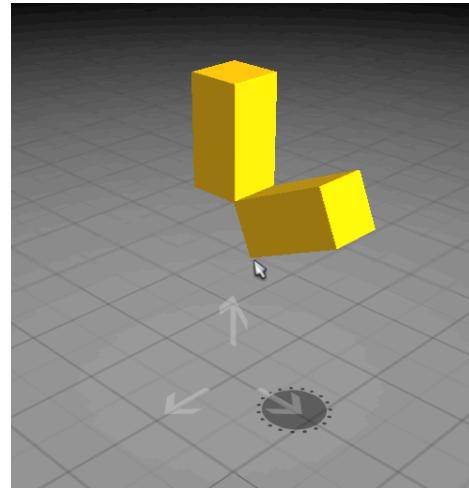


Figura 44. Dos cuerpos unidos, el cuerpo de arriba es estático, el de abajo cuelga del él y se mueve rotando en el eje definido en la unión. Fuente: Tutoriales NxOgre www.nxogre.org

Ragdolls

Esta es una técnica de animación y simulación, en la que se muestra el movimiento de un personaje cuando no tiene fuerzas que lo sostengan, por ejemplo un cuerpo sin vida). Esto trata al cuerpo del personaje como una serie de huesos rígidos conectados con bisagras en las uniones. La simulación muestra que sucede cuando el personaje, cae al suelo. Los ragdolls son una aplicación de objetos físicos de colisión y uniones

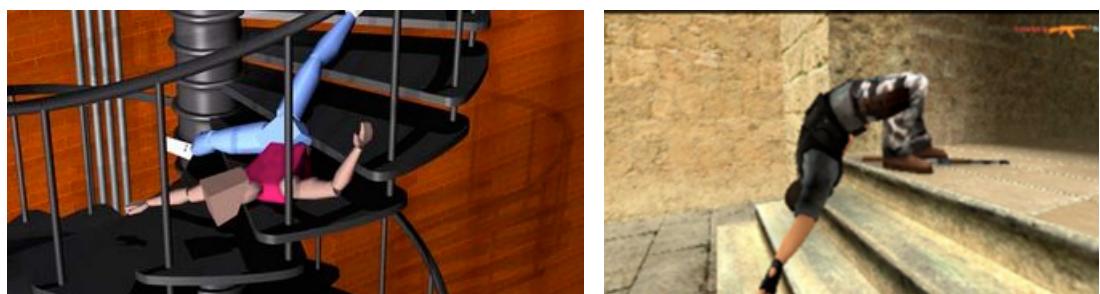


Figura 45. Dos figuras de ragdolls.

Cuerpos suaves

Cuerpos suaves son objetos que se a diferencia de los objetos de colisión mencionados anteriormente (cajas, esferas, capsulas), se pueden deformar. Esto provee gran nivel de realismo debido a que elementos como ropa, cabello, arena, telas pueden ser simulados en la escena.

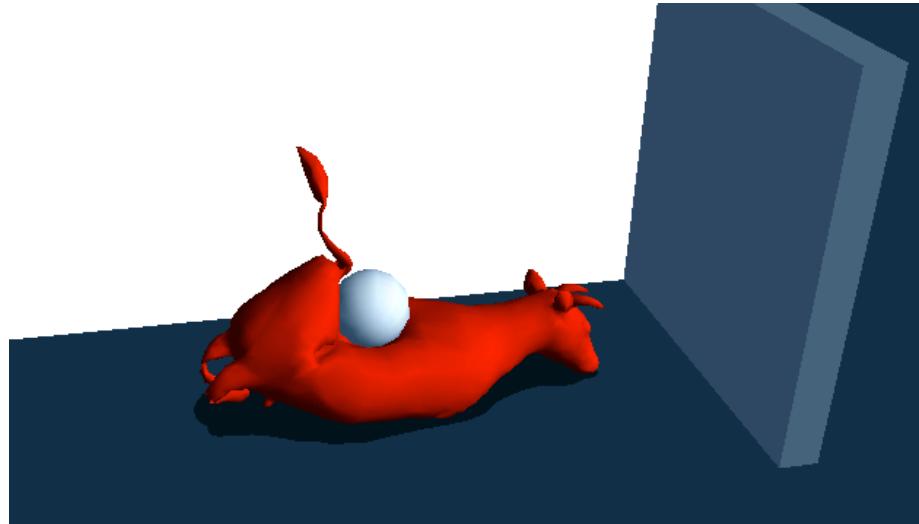


Figura 46. Ejemplo de un cuerpo suave siendo deformado al aplicar fuerzas sobre su superficie. Fuente: Tutoriales AGEIA PhysX

3.8.3. Aspectos importantes de un Motor Físico

Ventajas de los motores físicos

- Realismo
- Simplificación de algunos elementos como:
 - Gravedad
 - Colisiones
 - Control de vehículos
 - Sistemas de partículas
 - Detección de zonas
- Brinda nuevas herramientas de interacción.

Desventajas de los motores físicos

- Demasiados cálculos y puede llegar ocupar muchos ciclos de la CPU.
- Funciones complejas de implementar.
- Posible inestabilidad del sistema por aproximación (falta de precisión) en los cálculos.

Dependiendo de las características del videojuego, se debe escoger el motor de física que mejor cumpla con los requerimientos del proyecto de videojuego, para lo cual se recomienda hacer una evaluación detallada de los motores existentes en el mercado.

Las siguientes son algunas de las variables que influyen en la elección del Motor Físico:

- Necesidad de utilizar simulación precisa de la física en el juego.
- Existirán colisiones.
- Cuales y cuantos objetos se estarán simulando.
- Costo y licenciamiento del motor físico vs. presupuesto de la empresa.
- Plataforma objetivo del videojuego.
- Documentación del motor físico y soporte.
- Integración con herramientas de diseño del juego.
- Modularidad del código.

3.8.4. Librerías Físicas

Las siguientes son algunas de las librerías de código libre o gratuitas disponibles actualmente:

Bullet³⁶

Librería para detección de colisiones y cuerpos rígidos en tres dimensiones para videojuegos y animaciones, gratuita para uso comercial, incluyendo para PlayStation 3, código abierto, multiplataforma.

Newton Game Dynamics³⁷

Solución integrada para simulaciones en ambientes de tiempo real. La librería provee manejo de escenas, detección de colisiones, comportamientos dinámicos siendo pequeña, rápida, estable y fácil de usar.

ODE³⁸

³⁶ Bullet Physics Library. <http://www.continuousphysics.com>

³⁷ Newton Game Dynamics. <http://www.newtondynamics.com>

³⁸ ODE Open Dynamics Engine. <http://www.ode.org/>

Librería de alto rendimiento para simular dinámicas de cuerpos rígidos, multiplataforma, de fácil uso con C/C++. ODE es muy útil para simular vehículos, objetos en ambientes de realidad virtual y criaturas virtuales. Usada actualmente en muchos videojuegos y herramientas de simulación. Código abierto.

Tokamak³⁹

Librería de alto rendimiento para tiempo real, especialmente diseñada para videojuegos. interfaz simple de programar y de alto nivel.

PhysX⁴⁰

El kit de programación SDK de AGEIA PhysX es la única solución para física que se ha construido desde la base para ser compatible con la arquitectura de las consolas más modernas, así como con el hardware de juegos para PC impulsado por el procesador AGEIA PhysX de arquitectura masivamente paralela.

Licencia gratuita para desarrollar videojuegos para plataforma PC, que no usen el chip de acelerador de física.

3.8.5. Chip de procesamiento físico

En los últimos años se ha venido hablando de incorporar una tarjeta especializada para la simulación de la física en las consolas y computadores, llamada PPU (**Physics Processing Unit**). Esta tarjeta es un microprocesador que está diseñado para manejar los cálculos de la física, especialmente para motores de física de videojuegos. Una desventaja es que los costos para usar un videojuego en un PC se incrementarían, debido que además de la tarjeta aceleradora de gráficos 3d, habría que comprar la tarjeta aceleradora de física.

Algunos ejemplos donde podría ser útil un microprocesador dedicado a la física son: cuerpos rígidos dinámicos, cuerpos suaves dinámicos, detección de colisiones masivas, fluidos dinámicos, simulación de cabello y ropa, objetos compuestos.

³⁹ Tokamak <http://www.tokamakphysics.com>

⁴⁰ Ageia PhysX. <http://www.ageia.com>

3.8.6. Comparación de algunas librerías Físicas

CARACTERÍSTICAS	PhysX	ODE	Newton	Bullet
Separación de Escenas Físicas	Sí	No	No	No
Objetos de colisión	Sí	Sí	Sí	Sí
Grupos de colisiones	Sí	No	Sí	Sí
Sistemas de materiales	Sí	No	Sí	No
Sistemas de respuesta	Sí	Sí	Sí	Sí
Disparadores	Sí	No	Sí	No
Control de personajes	Sí	No	No	No
Control de vehículos	Sí	Sí	Sí	Sí
Mapa de alturas	Sí	No	Sí	Sí
Uniones	Sí	Sí	Sí	Sí
Fluidos	Sí	No	No	No
Cuerpos suaves	Sí	No	No	No
Partículas	Sí	No	No	No
Aceleración por hardware	Sí	No	No	No
Foro activo	Sí	Sí	Sí	Sí
Cantidad de usuarios	Grande	Poca	Mediana	En crecimiento
Cantidad de videojuegos o demos	Gran cantidad	Gran cantidad	Pocos	Pocos
Tiene grupos activos de desarrollo actualmente	Sí	Sí	Sí	Sí
Soporte actual de los creadores	Sí	No	Sí	Sí
En continuo desarrollo	Sí	Sí	Sin actualizar desde el 2006	Sí
Documentación	Sí	Sí	Sí	Sí
Multiplataforma	Sí	Sí	Sí	Sí
Código abierto	Sí	Sí	No	No
Licencia	Gratis para juegos comerciales para PC sin usar el chip de aceleración física	GNU	Gratis para juegos comerciales	Zlib/libpng
Última Versión	2.7.2	0.8.1	1.53	2.61
Fecha de inicio	2002	2001	2003	

3.9. SCRIPTING

Los scripts son programas que son interpretados en el momento de ejecución, es decir, no son compilados, sino que un Motor de Scripts ejecuta las acciones descritas en el script durante la ejecución de la aplicación. Los scripts son útiles en los videojuegos porque permiten que los diseñados añadan interacciones y lógica de juego desde un editor de juego o editor de texto, eliminando la necesidad de recompilar para realizar algún cambio en la lógica del juego.

3.9.1. Lenguajes de Scripting

Los lenguajes de scripting se crean pensando en la simplicidad, es decir, eliminan los elementos complejos de los lenguajes de programación tradicionales. En (GUTSCHMIDT, 2003) se definen algunas de las cualidades de los lenguajes de scripting:

- Son lenguajes interpretados.
- Poseen una sintaxis simple.
- Las variables son dinámicas, tal que pueden actuar como números o cadenas dependiendo en la operación que se realice con ellas.
- Las variables son creadas cuando son referenciadas.
- Poseen funciones de alto nivel para manejo de cadenas, como concatenación y búsquedas.
- No poseen apuntadores.
- No hay que administrar la memoria.
- El manejo de basura es automático.
- El lenguaje es interactivo y puede dar información durante la ejecución, como errores y problemas.
- Hay independencia de procesadores y sistemas operativos.
- Son optimizados para la eficiencia del programador, no del programa.

Una empresa puede crear su propio sistema de scripting, o utilizar un lenguaje de scripting existente adaptándolo a su Motor de Juego. Epic Games creo para su serie de Motores de Juego Unreal, el lenguaje UnrealScript⁴¹, el cual es muy similar a Java y permite modificar ampliamente el juego en ejecución.

```
class HolaMundo extends Mutator;  
  
function PostBeginPlay()
```

⁴¹ UnrealScript unreal.epicgames.com/UnrealScript.htm

```
{  
    Super.PostBeginPlay();  
    (Mutator.PostBeginPlay).Log("Hola Mundo");  
}
```

Figura 47. Script para escribir Hola Mundo en la consola del Motor de Juego Unreal.

Los scripts en los juegos pueden realizar tareas simples, como crear menús, movimientos de personajes y cámaras, hasta tareas más complejas como la lógica de todo el juego completo, definición de cada uno de los personajes, sus características, eventos de juego y metas que se deben cumplir.

La idea de los lenguajes de scripting es que puedan ser usados por personas sin conocimientos de programación, lo cual ha permitido que muchos juegos sean modificados por los usuarios, actividad que se conoce como Modding, y la modificación del juego se llama un Mod. Sin embargo algunas tareas pueden ser complejas y necesitar de un programador para su implementación, ejemplo de esto son los scripts para inteligencia artificial. A la hora de diseñar un lenguaje de scripting se debe pensar en que tanto poder se le quiere dar al usuario sobre el Motor de Juego.

3.9.2. Librerías

Algunas librerías (lenguajes) de scripting de tipo software libre actuales son:

Lua⁴²

Multiplataforma, software libre bajo la licencia MIT⁴³

Tomado del sitio web de Lua: "Lua combina sintaxis procedural con una poderosa descripción de datos basados en arreglos asociativos, y semánticas extensibles. Lua usa tipos de variables dinámicas, es interpretado de bytecodes, y tiene manejo automático de recolección de basura, siendo ideal para configuraciones, scripting y desarrollo rápido de prototipos".

Lua se puede ser enlazado con C++, a través de la librería LuaBind también bajo la licencia MIT.

Python⁴⁴

Multiplataforma, software libre bajo la licencia GPL⁴⁵.

⁴² Lenguaje Lua <http://www.lua.org>

⁴³ Licencia de Software MIT. <http://www.opensource.org/licenses/mit-license.php>

⁴⁴ Lenguaje Python. <http://www-python.org>

⁴⁵ Licencia de Software GPL. <http://www.gnu.org/copyleft/gpl.html>

Tomado del sitio web de Python: “*Python es un lenguaje de programación dinámico orientado a objetos que puede ser usado para muchos tipos de desarrollos de software. Ofrece un soporte robusto para su integración con otros lenguajes y herramientas, viene con una extensa librería estándar, y puede ser aprendido en pocos días.*”

Python puede también ser enlazado con otros lenguajes de programación, específicamente existe la librería Swig⁴⁶ para hacer el enlace con C++.

Ruby⁴⁷

Multiplataforma, software libre bajo la licencia GPL.

Tomado del sitio web de Ruby: “*Ruby es un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad. Su elegante sintaxis se siente natural al leerla y fácil al escribirla.*”

Ruby se destaca por ser completamente orientado a objetos. Todo en Ruby es un objeto, hasta los tipos de datos más simples como los enteros o booleanos.

Squirrel⁴⁸

Multiplataforma, software libre bajo la licencia zlib/libpng.⁴⁹

Tomado del sitio web de squirrel: “*Squirrel es un lenguaje de alto nivel imperativo orientado a objetos, diseñado para ser una herramienta robusta que satisface en tamaño, uso de memoria y requerimientos de tiempo de real de aplicaciones como juegos.*”

GameMonkey⁵⁰

Multiplataforma, libre bajo la licencia MIT.

⁴⁶ Swig. <http://www.swig.org>

⁴⁷ Lenguaje Ruby. <http://www.ruby-lang.org/>

⁴⁸ Lenguaje Squirrel. <http://squirrel-lang.org/>

⁴⁹ Licencia zlib/libpng. <http://www.opensource.org/licenses/zlib-license.php>

⁵⁰ Lenguaje GameMonkey. <http://www.somedude.net/gamemonkey/>

Es un lenguaje de scripting empotrado que esta hecho para juegos y aplicaciones de herramientas. Sin embargo es aplicable a cualquier proyecto que requiera soporte simple de scripting.

3.10. MOTOR DE ENTRADA

El módulo de entrada del usuario es el encargado de hacer el manejo de las interfaces hombre maquina disponibles al usuario. En un computador estos dispositivos son usualmente el ratón y el teclado, aunque también se pueden utilizar una variedad de dispositivos orientados a los videojuegos como los gamepads, joysticks, volantes entre otros.

Algunas de las responsabilidades del módulo de entrada son:

- Enumerar los dispositivos disponibles y sus características.
- Proveer interfaces para hacer uso de los dispositivos.
- Mantener el estado del dispositivo disponible para la aplicación.
- Puede tener un sistema de eventos que son enviados al sistema cuando el dispositivo es usado.
- Liberar el uso del recurso una vez termine la sesión de juego.

El manejo de las entradas del usuario suele ser específico para cada sistema operativo donde se ejecute la aplicación de juego. Por ejemplo, en Windows XP, se puede utilizar la librería "windows.h" y a través de los eventos que Windows envía a la ventana saber que cambio hubo en los dispositivos de entrada. Sin embargo también existen soluciones middleware o más completas para manejar las entradas, que permiten programar una sola vez y correr en múltiples plataformas, o proveen interfaces más sencillas para obtener información de los dispositivos.

3.10.1. Programación con y sin buffer

Generalmente las librerías proveen la opción de implementar la captura de entradas usando un buffer o sin usarlo. Cuando se usa un buffer el motor de entrada esta constantemente chequeando o esperando eventos del sistema de los dispositivos que se estén usando. Cada vez que se reciba un mensaje del sistema o se detecte un cambio, se guarda en un buffer el mensaje o cambio. Es usual que se implemente un sistema de eventos, donde cada mensaje o cambio dispara un evento, el cual el motor de juego debe procesar. Los buffers se utilizan para prevenir perdida de datos, ya sea porque la aplicación no puede chequear el estado del dispositivo cuando el jugador lo esta usando, o porque se envían muchos mensajes muy seguidos.

El otro método para usar dispositivos es sin buffer, donde es responsabilidad de la aplicación obtener el estado de los cada uno de los que este usando cuando lo necesite. Este método tiene

la ventaja de que es más organizado y se puede limitar la búsqueda de cambios a solo ciertos botones o dispositivos.

3.10.2. Sistema configurable

Un videojuego puede ser usado por muchos usuarios y cada usuario puede querer configurar los dispositivos de entrada de forma diferente, como más cómodo le parezca. Por esto es recomendable que las acciones que se realicen en el videojuego puedan ser realizadas por dispositivos asignados por el usuario y que su configuración este presente cuando el usuario juegue de nuevo.

Esto para el programador puede ser un poco tedioso de implementar, además se pueden presentar problemas con diferentes joysticks, debido a que no todos manejan los mismos estándares. Para esto es recomendable hacer las pruebas con la mayor cantidad de joystick posibles. Este es un problema que se evitan los juegos para consola, debido que generalmente son los dispositivos de entrada son los mismos.



Figura 48. Joystick tipo gamepad.

3.10.3. Mouse y teclado vs Joystick

Dependiendo del tipo de juego, puede que un dispositivo sea mejor que otro para ser usado en el videojuego. Hay movimientos en los juegos que no se pueden realizar bien con el teclado, como lo hace un joystick, y en juegos de puntería hay jugadores que prefieren el mouse porque otorga más precisión. Esto es decisión del usuario en sí, pero el sistema debe estar listo para reconocer y configurar el dispositivo de entrada que se escoja.

3.10.4. Librerías

OIS⁵¹

⁵¹ OIS Open Input System. <http://www.wreckedgames.com>

Librería para el manejo de dispositivos de entrada (joystick, ratón, teclado) de fácil uso y multiplataforma. Puede usar buffers, individuales para cada dispositivo si se necesita, los cuales envían eventos a la aplicación. Esta programado en C++ como software libre, y mantiene una comunidad de desarrollo activa. Permite manejar teclados de diferentes países. Es la librería de dispositivos de entrada por defecto del Motor Gráfico Ogre3D.

SDL⁵²

Simple DirectMedia Layer es una librería multiplataforma diseñada para proveer acceso de bajo nivel para audio, teclado, mouse, joystick con aceleración por hardware a través de OpenGL. Es usada por reproductores de video, emuladores, y videojuegos de calidad comercial.

SDL soporta Linux, Windows, Windows CE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, QNX, AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS

SDL es distribuida bajo la licencia GNU GPL versión 2.

OpenInput⁵³

Gratuita, de fácil uso y multiplataforma, el objetivo de la librería es proveer una interface al estilo SDL para dispositivos de entrada. Todas las funciones y estructuras de datos son intuitivas y bien documentadas, la librería tiene soporte para entradas basadas en eventos (con buffer) y para eventos basados en estados (entrada sin buffer). Se pueden programar fácilmente controladores adicionales si se requiere. Soporte para Linux, Windows y Mac OS.

⁵² SDL Simple DirectMedia Layer. <http://libsdl.org>

⁵³ OpenInput. <http://home.gna.org/openinput>

3.10.5. Comparación de las Librerías de Entrada

CARACTERÍSTICAS	OIS	SDL	OPENINPUT
Arquitectura de Plug-in	Sí	Sí	Sí
Programación con buffer	Sí	Sí	Sí
Dispositivos que reconoce	Teclado, Ratón y Joysticks	Teclado, Ratón y Joysticks en todas las plataformas soportadas	<ul style="list-style-type: none"> - X11 (Sistema X Window): Ratón y teclado. - Windows: ratón y teclado usando el API Win32 - Linux: Joysticks, ratón y teclado (/dev/js) - Señales ANSI
Configuración varios idiomas teclado	Sí	Sí	Sí
Foro activo / Wiki	Sí	Sí	Sí
Soporte actual de los creadores	Sí	Sí	Sí
En continuo desarrollo	Sí	Sí	Sí
Buena Documentación y ejemplos	Sí	Sí	Sí
Plataformas	<ul style="list-style-type: none"> - Windows - Linux - MacOSX 	<ul style="list-style-type: none"> - Linux - Windows - BeOS - MacOS - Mac OS X - FreeBSD - OpenBSD - BSD/OS - Solaris - IRIX - QNX 	<ul style="list-style-type: none"> - Linux: Joysticks, ratón y teclado (/dev/js) - GNU/Kernels de Linux 2.4 y 2.6 - Microsoft Windows 95 y superiores - X Window System (X11)
Compiladores	<ul style="list-style-type: none"> - GCC 3+ - Code::Blocks - Microsoft Visual 	<ul style="list-style-type: none"> - GCC 3+ - Code::Blocks - Microsoft Visual C++ 6 	<ul style="list-style-type: none"> - GCC 3+ - Code::Blocks - Microsoft Visual C++ 6

	C++ 6 - Microsoft Visual Studio	- Microsoft Visual Studio - Windows CE - CodeWarrior - Dec-C++	- Microsoft Visual Studio
Código abierto	Sí	Sí	Sí
Fecha de Inicio	2005-10-03	1998	2005-07-08
Desarrolladores	9	1 + decenas de contribuyentes	2
Licencia	Zlib/libpng	GNU LGPL	LGPL
Versión para 16 de septiembre de 2007	1.0	1.2	0.2.4
Sitio Web	http://sourceforge.net/projects/wgois	http://www.libsdl.org	http://home.gna.org/openinput

3.11. MOTOR DE SONIDO

Un aspecto importante para crear juegos inmersivos y realísticos es el sonido, encargado de mejorar la experiencia que brinda un juego a través del oído. La implementación del sonido en los videojuegos es un área en constante desarrollo, ya que se busca constantemente alcanzar el mayor nivel de inmersión posible. El sonido aporta al juego atmósfera, ambientación y acompañan en sincronía a los gráficos y los eventos que suceden en el juego.

3.11.1. Tipos de Sonidos

En un juego se tiene normalmente tres diferentes tipos de sonidos:

- Música de Fondo.
- Sonidos de Ambiente.
- Efectos de Sonido.

Música de Fondo. La música de fondo son piezas de música que le dan acompañamiento musical al juego, pero que no tienen una relación directa con lo que este sucediendo en el juego. Un músico se limita a crear una pieza de un tono acorde a la situación general del juego, por ejemplo, acción o suspense. La música de fondo en los juegos es usual que se repita constantemente, por lo

cual se trata de que tenga una duración suficiente para que la repetición pase desapercibida. La música se reproduce generalmente en estéreo.

Sonidos de Ambiente. Los sonidos de ambiente son aquellos que por lo general acompañan un suceso especial en el juego. Su tarea es enfatizar en el jugador una emoción producida por el juego. Así por ejemplo, cuando el jugador se enfrenta a una situación de peligro, se reproduce un sonido que pone al jugador en estado alerta. Estos sonidos también se reproducen en estéreo.

Efectos de Sonido. Los efectos de sonido son los sonidos con los que el jugador tiene un contacto más cercano. Son aquellos que se producen por acciones del jugador, los objetos o personajes que están en el juego. En un juego de acción, algunos efectos que podríamos encontrar son las pisadas del jugador, los disparos de armas, explosiones, voces humanas, maquinas en funcionamiento, etc. Estos sonidos son los que tienen el papel mas importante en la inmersión del jugador, pues están directamente asociados a lo que sucede y se muestra en el juego. Son sonidos monofónicos y se posicionan dependiendo de la ubicación relativa de sus respectivas fuentes en el espacio en tres dimensiones que rodea al oyente.

3.11.2. Audio 3D

El audio en 3D o sonido posicional, es una forma de reproducir los sonidos de la forma que parezca que ocurran en un lugar cualquiera alrededor de quien escucha, ya sea al frente, a los lado, encima, debajo o detrás.

El método mas sencillo para conseguir este efecto es haciendo uso de los parlantes estéreos, un sonido con mas volumen en un parlante que en el otro parece que viene del lado donde esta situado el parlante. Este sonido sin embargo se considera es solo en una dimensión.

El verdadero sonido en 3D se puede hacer de dos formas hoy en día. La primera es usando varios parlantes colocados alrededor del oyente, y reproduciendo los sonidos desde uno u otro parlante, o una combinación de ellos según la posición del sonido respecto al oyente en la escena en que se encuentra. En este método se oye hablar de sistemas 5.1 y 7.1, una combinación de 5 parlantes y un Sub-Woofer y 7 parlantes y un Sub-Woofer respectivamente. El Sub-Woofer es un parlante que reproduce sonidos graves entre los 20 y 80 Hertz. La segunda forma es a través de algoritmos de posicionamiento 3D que funcionan imitando la forma en que percibimos los sonidos dependiendo de la posición, a través de una función. Una función muy común es la HRTF (Head Related Transfer Function), la cual filtra el sonido recibiendo un ángulo y una elevación.

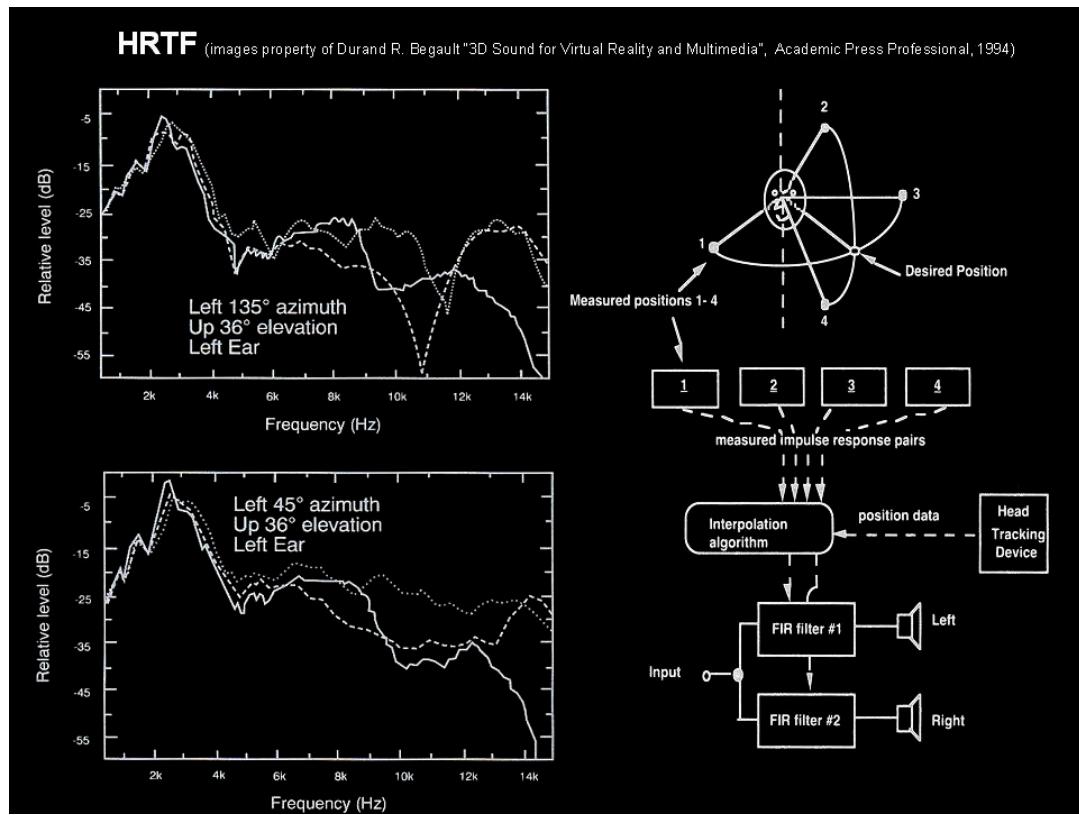


Figura 49. Filtrado haciendo uso de HRTF. Fuente: <http://www.digit-life.com/articles2/sound-technology/index.html>

3.11.3. El Motor de Sonido

Hoy en día no es muy común hablar de motor de sonido, sino de librería de sonido. Se emplea este término aquí para asignarle la misma importancia que a los demás módulos que hacen parte de un motor de juego. Una librería de sonido hoy en día está encargada de proveer al programador con métodos o funciones que le permitan hacer uso completo del hardware de sonido de la máquina, para cargar, reproducir y usar efectos dinámicos sobre los sonidos.

Algunas de las cualidades que se esperan en un Motor de Sonido actual son:

- Manejo de múltiples formatos de audio.
- Uso de streaming.
- Asignación automática de canales de reproducción.
- Canales virtuales de reproducción.

- Uso de audio posicional y con movimiento, actualizando volumen y frecuencia según la posición y velocidad del jugador y de las fuentes de sonido.
- Transiciones entre sonidos y música.
- Efectos de sonido como eco, reverberación, ruido, efecto Doppler, entre otros.
- Adaptabilidad a diferentes configuraciones de parlantes.



Figura 50. En el dibujo se muestra las fuentes sonoras

3.11.4. Librerías

Las librerías disponibles para reproducir sonidos de tipo comercial son generalmente las más usadas por los que crean videojuegos, ya que vienen con más herramientas. Otro aspecto importante es que los formatos para guardar sonidos están en su mayoría patentados, lo que limita su uso. Sin embargo, en los últimos años han aparecido tecnologías y formatos libres que han permitido la reproducción de sonidos sin tener que pagar derechos, como es el caso de la librería Ogg Vorbis⁵⁴.

FMOD⁵⁵

Multiplataforma, uso libre para juegos no comerciales, soporta sonido 3d, MIDI, MP3, OGG VORBIS, WMA, AIFF, obstrucción/occlusión, reproducción de CD, streaming de internet y muchas más opciones que pueden ser encontradas en la página oficial. Ha sido usado para muchos

⁵⁴ Ogg es una estructura de datos orientada a multimedia. Vorbis es el códec de compresión.

<http://www.vorbis.com>

⁵⁵ FMOD. <http://www.fmod.org>

videojuegos comerciales debido a su alcance, y facilidad de implementación, pero su licencia es costosa para una empresa pequeña.

OpenAL⁵⁶

Multiplataforma, licencia LGPL, “es una librería de audio diseñada para usar con videojuegos y muchas otras aplicaciones de audio” existe gran cantidad de usuarios trabajando en él y creando librerías intermedias para usar mas fácilmente la librería con diferentes aplicaciones.

3.12. INTERFAZ GRÁFICA DE USUARIO

La interfaz gráfica de usuario o GUI en inglés (Graphical User Interface) es el módulo del Motor de Juego que permite mostrarle al jugador un sistema de menús e información sobre el juego, y además interactuar con el sistema. Existen en los juegos generalmente dos tipos de interfaces, una es el menú de juego y la otra los elementos que se muestran en pantalla mientras se está jugando.

Las GUIs generalmente tienen unos tipos de objetos comunes los cuales facilitan la creación dinámica de la interfaz, que sumado a una definición de estructuras de archivos de interfaz, pueden separar la tarea del diseño de la interfaz del programador. A estos objetos se les suele llamar Widgets. Algunos widgets comunes son:

- Botones
- Imágenes
- Ventanas
- Deslizadores
- Cuadros de selección
- Lista de opciones
- Spinners

Un sistema de interfaces se espera defina para cada uno de estos objetos funciones que permitan obtener su estado y generar eventos cuando el jugador los utilice.

⁵⁶ OpenAL. <http://www.openal.org>

3.12.1. HUD

El Heads Up Display (HUD) es el conjunto de elementos de interfaz gráfica que permiten mostrarle información al jugador mientras juega, específicamente aquella que no puede conocer fácilmente a través de lo que se ve en la escena del juego. Esta información que se muestra depende del tipo de videojuego, por ejemplo, en un videojuego de carreras de automóviles el HUD suele mostrar el mapa completo de carretera, el tacómetro, las vueltas se han completado, las vueltas que faltan, la posición respecto a los otros corredores, etc.

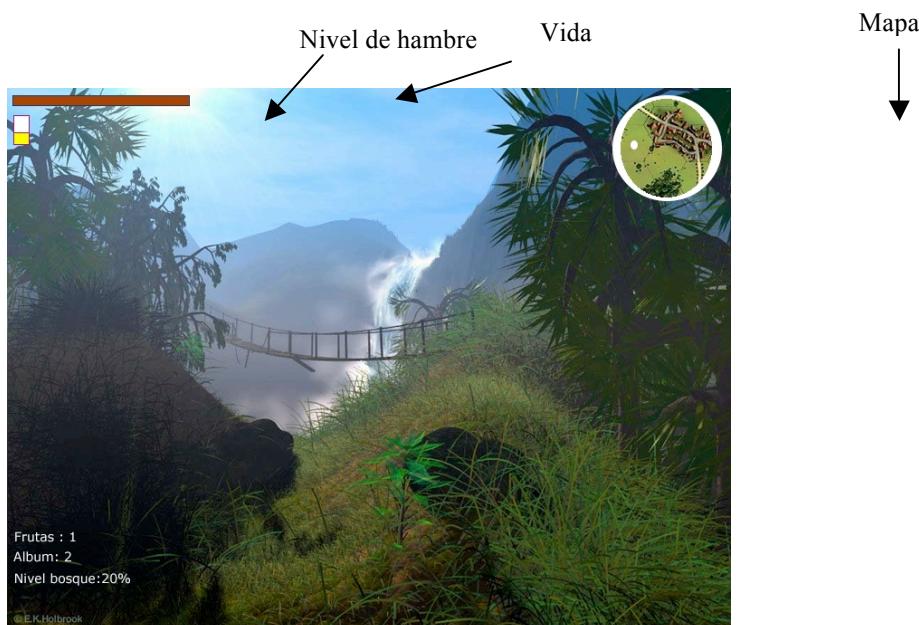


Figura 51. Ejemplo de un HUD de videojuego

El HUD es normalmente dibujado sobre el resto de la escena, es decir, se dibuja después de haber dibujado la escena del juego. Algunos juegos utilizan HUDs con elementos 3D, y también se puede diseñar de tal forma que el HUD parezca ser parte de la escena. Por ejemplo, en el videojuego Metroid Prime, el HUD se muestra como el casco que lleva puesto el personaje principal.



Figura 52. HUD simulando un casco en el juego Metroid Prime para la consola GameCube.

La forma final en que se muestra el HUD es generalmente decisión del diseñador, por esto un sistema de interfaces de juego debe ser lo suficientemente flexible para aplicar cualquier idea.

El HUD suele ser una aplicación del sistema de interfaz del juego, y no se suele separar uno del otro pues generalmente utilizan los mismos widgets.

3.12.2. Menús

Los Menús del videojuego son una herramienta que permiten al jugador configurar y realizar acciones sobre la aplicación del juego. Los menús de juego son un elemento que se busca minimizar al máximo, ya que cada vez que accedemos a ellos se pierde la inmersión. Algunas características importantes que se esperan de un menú de juego:

- Sencillo.
- Atractivo.
- Intuitivo.
- Rápido.



Figura 53. Ejemplo de un Menú de Juego

Un menú típico de juego tiene opciones como:

- Empezar el juego
- Cargar partida de juego.
- Configurar el juego
- Ver los créditos
- Salir

3.12.3. Librerías

Un sistema de menús viene generalmente ligado al motor gráfico propio del juego, es decir un motor gráfico debe proveer un sistema de interfaces. Sin embargo existen algunas librerías que pueden ser adaptadas a un motor gráfico cualquiera que no lo tenga, según los requerimientos.

CEGUI (Crazy Eddie's GUI System)⁵⁷

El propósito principal de esta librería es crear interfaces de videojuegos. CEGUI es muy flexible debido a que usa archivos XML para describir el aspecto de las interfaces. Tiene un editor de interfaces llamado CeguiLayoutEditor, el cual permite crear las ventanas y agregarles y posicionar los widgets en ellas. Esta licenciada bajo LGPL / MIT.

OpenGL GUI⁵⁸

Es una librería GUI para crear interfaces de juegos. Esta basada en OpenGL y escrita en C++. Completamente gratuita y de código libre.

AntTweakBar⁵⁹

Es una librería pequeña programada en C/C++ que permite añadir una GUI a aplicaciones que utilizan DirectX o OpenGL, licenciada bajo zlib/libpng.

GLUI⁶⁰

Librería para programar GUIs usando OpenGL. Es multiplataforma pues utiliza la librería GLUT. La librería está licenciada bajo LGPL.

3.13. EDICION DE NIVELES

La tarea de desarrollar un videojuego requiere de la elaboración de grandes cantidades de material visual, además de sonidos, música, y los eventos que suceden en el juego. Todo este material suele producirse en paquetes especializados de software de forma separada, para luego ser unidos en lo que se suele llamar un nivel de juego, es decir, un área determinada del juego con características propias. Por ejemplo, en un juego de carreras de Formula 1, un nivel podría ser una pista de una ciudad del mundo. En un juego de aventura, un nivel puede ser todo el escenario que se recorre hasta conseguir un objetivo específico.

⁵⁷ CEGUI <http://www.cegui.org.uk>

⁵⁸ OpenGL GUI <http://www.bramstein.nl/gui>

⁵⁹ AntTweakBar <http://www.antisphere.com/Wiki/tools:anttweakbar>

⁶⁰ GLUI <http://glui.sourceforge.net>

La tarea de reunir todo este material, no puede depender del programador, es decir, el proceso de diseño del los niveles, no debe estar escrito en código, sino que debe brindársele a los diseñadores la flexibilidad suficiente para crear y probar su material de juego sin tener que depender de un proceso de compilación. Para solucionar este problema se crearon los editores de nivel.

3.13.1. El Editor de Nivel

Un editor de nivel (también conocido como editor de mapas o de escenarios) es la aplicación de software utilizada por los diseñadores de niveles para la creación de los escenarios del juego, o mapas del juego, y que permiten la visualización y edición del contenido del juego para luego ser exportado a un formato específico usado por el motor de juego. Un editor de nivel puede tener opciones avanzadas de modelado y edición de geometría 3d, o simplemente permitir cargar archivos de los objetos del juego para ser situados en el escenario y configurar parámetros específicos del juego. El editor debe permitir agregar toda la información necesaria para que el resultado final pueda ser usado sin ningún tipo de intervención del programador, así que puede incluir aspectos como las características físicas, sonidos, interacciones, y eventos que ocurren en el nivel. Algunos editores pueden hacer parte del juego, es decir, se puede hacer edición del nivel desde el juego en ejecución, otros, la mayoría, funcionan de forma separada.

Los editores de nivel muchas veces se ponen a disposición del público para que los jugadores puedan crear sus propios niveles usando las características propias del juego. Muchos juegos comerciales hoy en día realizan esta tarea, ya que es una forma de mantener el interés en el juego y además alargar el ciclo de vida del producto. Algunos editores de nivel de juegos importantes disponibles al público (algunos solamente cuando se compra el videojuego) son:

- Age of Empires Scenario Editor – Age of Empires
- Valve Hammer Editor - Half-Life
- UnrealEd - Unreal
- GtkRadiant – Quake I, II, III, 4, entre otros
- StarEdit – StarCraft
- Tomb Raider Level Editor – Tomb Raider Chronicles
- Warcraft III World Editor – Warcraft III

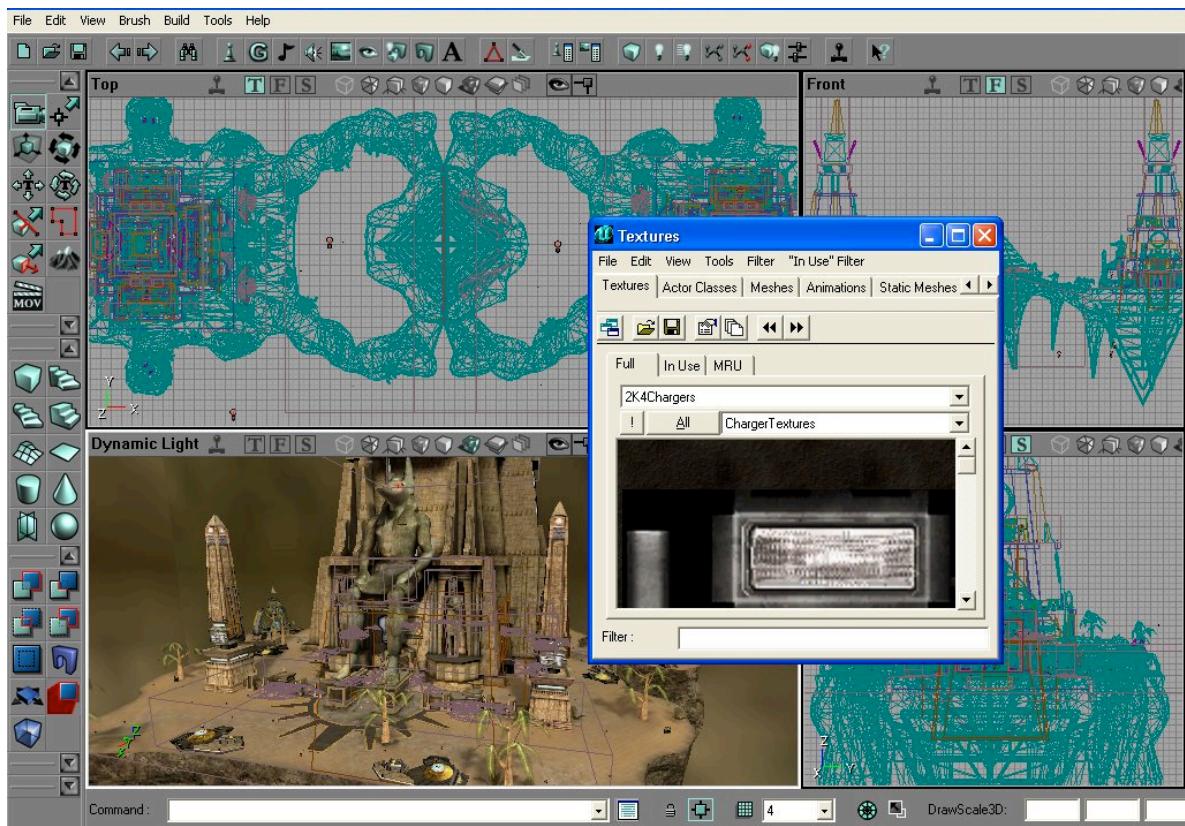


Figura 54. Editor de niveles Unreal Ed 2.0.

Alrededor de algunos de estos editores de nivel se crean sitios web donde los usuarios pueden mostrar su trabajo y compartirlo para que otros lo jueguen. Estos sitios crean comunidades que giran alrededor de los juegos que ellos mismos editan, y son útiles a los desarrolladores como una forma de evaluar el juego, el editor, y encontrar nuevas oportunidades de desarrollo. Sin embargo no todos los juegos liberan sus editores al público, ya que no se tiene el tiempo, las herramientas, el juego no es del tipo de los que se pueden editar o porque se quieren proteger tecnologías específicas del software.

Existen casos en que no solo se crean editores para reunir todo el material creado. También se pueden crear editores auxiliares para otro tipo de contenido del juego. Por ejemplo, puede haber un editor de interfaces, un editor de terrenos, un editor de modelos, y un editor de nivel donde todos los objetos se unen.



Figura 55. Flujo de datos en un editor de nivel.

Algunos editores de nivel de juegos comerciales se incluyen en el paquete del videojuego para permitir a los usuarios crear sus propios niveles directamente. En algunos casos el editor de niveles esta integrado al videojuego y se puede hacer edición directa del contenido con el videojuego en ejecución. En otros casos y más frecuentemente, se encuentra separado del videojuego y genera archivos que son cargados por la aplicación.

A medida que el videojuego aumenta su contenido, la creación de un editor de nivel se hace muchas más necesaria. Además se requiere que la facilidad de editar cualquiera de los niveles del videojuego y realizar cambios sin tener que ejecutar una nueva compilación del código. Entre más información del juego sea editable dentro del editor de nivel, mas se podrá separar el código detrás del videojuego de su contenido.

3.13.2. Visualización del nivel

La finalidad más importante de las herramientas de creación de niveles es permitir al diseñador ver el resultado final de los niveles que están siendo editados. Esto es frecuentemente llamado “lo que usted ve es lo que obtiene” (WYSIWYG), también llamada la “vista del jugador” ya que esta muestra lo que el jugador vera en el juego.

En el editor de nivel, el diseñador debe crear lo que el jugador deberá ver, por esto, la mejor opción es usar en el editor, si es posible, el mismo motor gráfico del videojuego. Así el trabajo del diseñador será más eficiente, si no tendrá que crear el nivel en una aplicación que puede verse muy distinta al resultado final del juego, y visualizar su efecto de forma separada en el videojuego, lo que puede retrasar el proceso de diseño.

3.13.3. Características de los editores

Las propiedades de un editor dependen en gran medida del videojuego en si, es decir, un editor es generalmente un software que se crea a la medida para el proyecto. Sin embargo, además de las propiedades típicas que se esperan de cualquier software como estabilidad, eficiencia, eficacia, etc., también se espera tenga estas características:

- Movimiento libre de las cámaras en las ventanas de edición, para poder observar la escena desde cualquier ángulo.
- Vista del resultado final o cercano al resultado final.
- Administrador de los recursos utilizados en el nivel.
- Herramientas para tareas frecuentes de edición del nivel.
- Posibilidad de cargar, y guardar archivos de nivel.
- Manejo de varias ventanas de edición, con la opción de determinar vistas de proyección estándares como superior, lateral, frontal, etc.

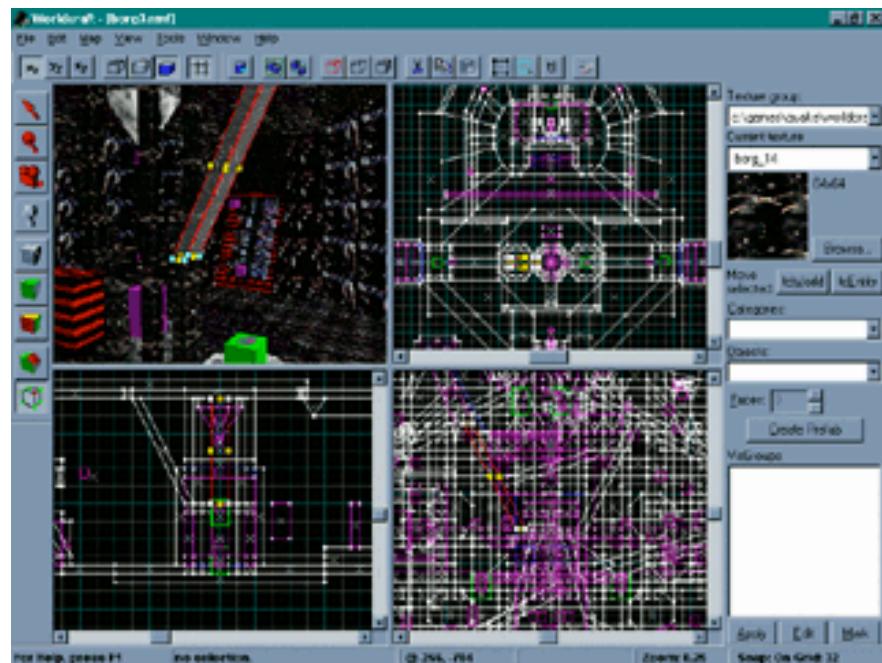


Figura 56. Editor de nivel con varias ventanas de edición.

3.13.4. Creación del Editor

El editor de niveles puede ser creado de varias formas:

1. Puede ser un software que se realice con las mismas librerías gráficas con que se creo el motor gráfico dándole a esto la posibilidad de que los objetos se vean como estarán exactamente en el juego y además no tengan inconsistencias en el proceso de exportado.
2. Utilizando programas de modelado existentes, como Blender, 3ds Max, XSI Softimage, etc. y exportando los objetos a un formato entendible por el Motor de Juego, ya sea a través de plugins o scripts. Por ejemplo con Blender se puede hacer uso de scripts escritos en Python para exportar la información en un formato propio, y además añadir nuevos objetos de interfaz para trabajar en la edición.
3. Creando un editor de niveles con librerías diferentes a las del Motor Gráfico. Esto puede ser porque muchas veces el Motor Gráfico no esta listo para cuando se están construyendo las herramientas, o no es lo suficientemente modular como para usarse en otra aplicación diferente a un juego.

3.13.5. Software gratuito o de bajo costo para edición de niveles.

Existen paquetes de software edición de código abierto, gratuito o de muy bajo costo disponibles para la edición 3D, que pueden ser modificados fácilmente para ser usados como editores de juego.

Blender⁶¹

Paquete de modelado 3d y animación, multiplataforma, gratuito. Esta en constante desarrollo y evaluación por su comunidad de usuarios. Posibilidad de incluir Scripts de Python.

gmax⁶²

Paquete de modelado 3d, es la versión gratuita del software de edición y renderizado 3ds Max 5 (con limitaciones). No tiene soporte actual de los creadores. Posibilidad de incluir scripts con el lenguaje MaxScripts.

Freeworld3D⁶³

Editor de terreno, permite la edición de mapas de alturas, creación de sombras, posicionamiento de mallas y entidades, permite exportar niveles en formato para Ogre3d. Licencia comercial.

GtkRadiant⁶⁴

Software diseñador de niveles desarrollado por id Software y Loki Software, utilizado para crear mapas de una cierta variedad de videojuegos.

Licencia: GPL

⁶¹ Blender <http://www.blender.org/>

⁶² gmax www.autodesk.com/gmax

⁶³ FreeWorld3D www.freeworld3d.org

⁶⁴ GtkRadiant <http://www.qeradiant.com>

4. DESCRIPCIÓN DEL PROBLEMA

En este proyecto se busca desarrollar una aplicación demostrativa de algunas de las tecnologías usadas en los videojuegos utilizando software libre, gratuito, o de bajo costo, para reducir las barreras de entrada al mercado a aquellas empresas con pocos recursos económicos.

Como la variedad y profundidad de tecnologías es tan extensa, se decidió hacer énfasis en los gráficos, la física, el manejo de entradas, la inteligencia artificial y el sonido. Para esto se fue necesario diseñar un videojuego que hiciera uso de estas tecnologías. El videojuego planeado hace parte de un proyecto de creación de empresa desarrolladora de videojuegos y aplicaciones 3d. Por esto el diseño se extiende a un juego mas extenso de lo que requiere la demostración, pero se incluye para dar a conocer la idea completa de diseño.

4.1. Concepto Del Juego

El juego busca enseñar acerca de las habilidades y características del animal, y como las usa en su sobrevivencia y relaciones con otros seres vivos. El jugador controlará a un mono aullador (*Alouatta Seniculus*), quien debe regresar a su hábitat natural en una travesía en la cual recorre varios ecosistemas, conociendo sus animales y plantas, y superando pruebas y obstáculos en el camino.

4.2. Características

- Vista en 3ra persona, permite ver al jugador, y el ambiente que lo rodea.
- El juego ocurrirá en 3 ambientes. La zona Rural con fincas, animales de granja y caminos de piedra y quebradas, donde se puede notar la presencia del hombre. Los Llanos con clima y decoración tipo Sabana, con largas planicies, humedales y días lluviosos y calurosos y por último el Bosque Húmedo Tropical, en donde el mono encontrará su manada.
- El mono podrá correr, saltar, balancearse en árboles e interactuar con otros animales, ya sea cazando, evadiéndolos o peleando. La sobrevivencia del mono dependerá de cómo el jugador maneje sus habilidades.

- En cada área habitarán diferentes animales e insectos. Algunos peligrosos, los cuales habrá que evitar a toda costa. Otros podrán ayudar al mono en su camino de retorno.
- De cada animal que se vea se podrá conocer su información, y se le darán pistas al jugador de dónde podrá encontrar algunos de los animales en el juego.
- También se conocerán las características de cada hábitat, como su temperatura, humedad, altura, biodiversidad y datos curiosos.

4.3. Motivación del Jugador

El jugador avanzará a través de diferentes escenarios tratando de superar los obstáculos que encuentra. Tendrá que vencer animales que lo ataquen, o le impidan avanzar, peleando o usando ítems que encuentre en el camino que le puedan ayudar. A su vez deberá encontrar la manera de cruzar obstáculos geográficos como ríos, paredes de tierra, y depresiones. También podrá columpiarse de árbol en árbol para moverse dentro del juego.

5. SOLUCIÓN PROPUESTA

Uno de los aspectos más importantes en el desarrollo de software es la reusabilidad del código. En este proyecto se trabajó desde el comienzo para que todas las partes del software tuvieran alta cohesión, bajo acoplamiento y pudieran ser usadas fácilmente en otro proyecto. Esta práctica se llevo a cabo también en los scripts desarrollados para el editor del juego programado en gmax. Se busco mantener una distancia entre los objetos reusables y los propios del juego. Sin embargo no se pudo eliminar completamente la información específica del videojuego Alouatta en si, y por esto el editor, si fuese a ser usado en otro proyecto deberá tener algunas, aunque muy pocas, modificaciones.

Se escogió trabajar cada una de las secciones importantes como módulos separados. La idea es que aunque se esta resolviendo un problema específico, el videojuego Alouatta, se busca desarrollar una solución que pueda ser reutilizada en otros juegos o aplicaciones multimedia. De esta forma se definieron los siguientes módulos:

- Motor de Juego
- Motor Gráfico
- Motor Físico
- Motor de Entrada
- Motor de IA
- Motor de Sonido
- Módulo para Video
- Módulo para Menús

Cada uno de estos módulos es independiente del otro. Sin embargo algunos objetos del software, como por ejemplo la clase Personaje, tienen varias dependencias y no pueden ser usados sin los módulos que utiliza, en el caso de la clase Personaje, utiliza el Motor de Entrada y el Motor Físico.

El Motor Gráfico hace uso de la librería gráfica OGRE. Aunque OGRE es en si una librería completa y autónoma, fue necesario encapsularla en un módulo propio donde se simplificaron tareas y añadieron funcionalidades a través de métodos.

El motor físico utiliza la librería PhysX de Ageia, conectada con la librería de gráficos OGRE a través de la librería NxOgre⁶⁵.

El Motor de Sonido utiliza la librería de sonido FMOD, y permite reproducir tres tipos de sonidos: efectos, sonidos de ambiente y música.

El Motor de Entrada esta encargado de manejar el dispositivo de entrada usado en el juego. El Motor de Entrada fue diseñado usando la librería OIS de tal forma que todas las acciones del juego pueden ser asignadas a cualquier dispositivo o interfaz compatible. Estas acciones están definidas en un archivo de cabecera que puede ser editado añadiendo o quitando acciones según se necesite sin afectar la funcionalidad del Motor de Entrada, es decir, pueden cambiarse las acciones fácilmente para adaptarse a cualquier otro proyecto.

El Motor de IA hace uso de la librería OpenSteer para generar movimientos naturales en los pájaros que sobrevuelan el escenario. Esta librería a su vez hace uso de módulos de IA, así que puede ser extendida para otros comportamientos.

El Módulo para Video utiliza la librería DirectShow para mostrar video en la pantalla. Puede mostrar cualquier tipo de video con cualquier codificación, mientras este registrado el decodificador que se vaya a usar en Windows.

El módulo de Menús y el motor gráfico utiliza la librería CEGUI para el manejo de las interfaces. Es el encargado de cargar todos los menús y recibir las actualizaciones de ellos.

El Motor de Juego es el eje que mueve a todos los módulos en sincronía. El Motor tiene un método que es llamado antes de dibujar cada cuadro en pantalla, el cual se usa para actualizar los módulos que se deban actualizar dependiendo del estado del juego. También esta encargado de la creación y destrucción de los módulos. Cada módulo se encarga de crear y destruir los objetos que le pertenecen, por ejemplo, el Motor de Sonido debe encargarse de liberar los recursos de sonido una vez se dejan de utilizar.

En la programación de la aplicación se hizo uso de varios paradigmas de diseño de software. Uno fue el uso de **Singletons**, los cuales permiten acceder a la instancia única de una clase desde cualquier lugar en el software. Otro fue el uso de **Factories** o “fabricas” de objetos, es decir, algunos objetos se crean y destruyen usando métodos ayudantes, simplificando el proceso ya que

⁶⁵ NxOgre es una librería auxiliar a OGRE, que permite una rápida integración con PhysX.
<http://www.nxogre.org>

estos objetos crean otros, y en el momento en que deben destruirse, se deben realizar varias tareas también.

5.1. ANÁLISIS

5.1.1. Características del Juego

Género

Juego de Plataforma Educativo.

Audiencia (Clientes Potenciales)

Hombres y Mujeres entre los 7 y 16.

Puntos Únicos (diferenciación)

- Mostrará datos reales de los ambientes y animales en donde se desarrolle el juego.
- Enseñará de forma divertida, no será el típico juego educativo de tablero.
- En la mayor parte los animales reaccionarán como lo hacen comúnmente.
- El jugador podrá ser un animal y comportarse como el animal.

Plataforma

PC con las siguientes características:

- Sistema Operativo Windows XP SP2 o Windows Vista.
- 512 RAM.
- 50 MB libres de disco duro.
- Aceleradora 3D con soporte de Shaders v2.0 o superior y 64 MB de memoria.
- Tarjeta de Sonido.
- Ratón y Teclado.

Objetivos de Diseño

Real: Lograr que el ambiente y los animales se comporten los más cercano posible a la realidad.

Contenido: Lograr que a través del juego se muestre mucha información.

Inmersión: Incentivar al jugador a que se motive para resolver los obstáculos y retos del juego.

Aprendizaje: Brindarle información al jugador acerca de la flora y fauna colombiana a través del juego.

5.1.2. Personajes

Nombre:	Mono Aullador
Características:	El mono es el protagonista del juego. Por esto es quien mas interactúa con el mundo, y los demás personajes.
Acciones:	Moverse. Saltar. Agacharse. Rodar. Tirar. Recoger. Atacar. Usar. Trepar. Colgarse. Columpiarse. Protegerse. Aullar. (Nadar)

Nombre:	Animales Activos
Características:	Son los animales que están en el universo del juego y que participan en el desarrollo de la historia. Interactúan con el jugador. Tienen unas acciones definidas y ayudan o amenazan al mono.
Acciones:	Moverse. (Saltar) (Agacharse)

	<p>(Rodar) (Tirar) (Recoger) (Atacar) (Trepas) (Colgarse) (Columpiarse) (Protegerse) (Aullar) (Volar) (Nadar)</p> <p><i>() Campos Opcionales, dependen del animal.</i></p>
--	--

5.1.3. Ítems del Juego

- **Semillas y Nueces**

En cada nivel el jugador debe recoger mínimo el 80% de las semillas para terminar el nivel. Si recoge el 100% obtiene un bono.

- **Hojas de Árboles**

Las hojas de los árboles proporcionan la energía que el jugador necesita. Las hojas no se acaban, pero no son fáciles de conseguir pues están en las copas de los árboles.

- **Piedras**

El jugador puede coger una piedra a la vez y tirarla para defenderse, atacar, o realizar otras actividades en el juego.

- **Palos**

El mono puede recoger palos que utiliza como arma para defenderse.

- **Frutos**

Los frutos también le dan más energía al mono que las hojas, pero son más escasos.

5.1.4. Escenario

El escenario estará formado un espacio 3D, que representará un ambiente natural. En él se podrán encontrar los siguientes elementos:

Terreno

El terreno será el espacio en el cual el jugador podrá moverse. Esto incluirá, las montañas, el cielo, los ríos o quebradas, lagos, pantanos, caminos. Todos los demás objetos estarán sobre el terreno. El terreno incluirá objetos de flora que no tienen interacción con el jugador. Es decir, el terreno puede tener un árbol del cual no se tiene información.

Flora

La flora serán todos los elementos de vegetación, con algunos se podrá tener algún tipo de interacción:

- El jugador podrá trepar algunos arboles.
- El jugador pasará por arbustos o malezas y estos se moverán.
- El jugador podrá obtener información de algunos objetos de flora.
- El jugador podrá columpiarse en las ramas de algunos árboles.

Fauna

Son todos los animales del juego diferentes al protagonista. Algunos interactuarán con él y otros no. Los animales se podrán moverse en el terreno dentro de unas áreas específicas determinadas por el papel que desempeñen dentro del juego. También realizaran diferentes movimientos particulares a su especie, y las interacciones con el jugador serán variadas. Los animales tendrán salud y acciones particulares a su función en el juego.

Objetos Inanimados

Son todos los objetos del juego con los cuales se interactúa, pero no tienen vida propia, entonces no se mueven por sí mismos. Las interacciones provienen todas del jugador. Por ejemplo, una piedra que puede recoger el jugador, un coco que se rompe cuando el jugador lo golpea, etc. Algunos son estáticos y otros pueden ser movidos.

Objetos de Información para el jugador

Son objetos que cumplen una función en el desarrollo del juego. Pueden ser avisos para informar al jugador de eventos del juego, o pueden ser disparadores que activen un evento en el juego. Algunos serán visibles en el juego y otros no. Incluyen:

- Símbolos que indiquen que acción se puede realizar.
- Señales que muestren información del área, como el mapa, las características del escenario natural y otros datos importantes. Cuando el jugador esté cerca de una de estas señales, tendrá la opción de hacer un acercamiento usando el dispositivo de entrada.
- Disparadores que activen los símbolos de información.

Ambientación

Sonidos

- Todas las acciones del jugador tendrán un sonido asociado, como caminar, saltar, correr, trepar, columpiarse, golpear, etc.
- Habrá sonidos que darán ambientación a eventos del juego, como ganar una pelea, o terminar un reto.
- Estarán las voces de los animales, el sonido del ambiente como los pájaros, ríos, el viento, lluvia, las hojas de los árboles, etc.
- La interfaz de usuario tendrá un sonido para cuando se cambia de selección, otro para cuando se cancela la acción o se devuelve a la pantalla anterior, y otro sonido para cuando se acepta una opción en el menú.

Música

- El juego tendrá música de fondo.

Efectos Gráficos

- Huellas de pisadas.
- Polvo cuando corre, da vueltas, etc.
- Corona del Sol.
- Lluvia.
- Insectos.
- Tonos de verdes de las hojas.
- Formas y colores de flores y frutos.
- Movimientos de las hojas, plantas y árboles con el viento.
- Sombras.

- Rocío.
- Niebla.
- Ríos y cascadas.

5.1.5. Reglas del Juego

Reglas Generales

- El jugador debe completar una serie de niveles para terminar el juego.
- Cada nivel tiene subniveles o actividades que debe completar para terminar el nivel.
- El jugador tiene que recoger mínimo el 80% de las semillas y nueces que están por todo el nivel para que pueda terminarse.
- La energía del jugador disminuye con el tiempo, si llega a cero el jugador se muere. Las hojas que se pueden comer son infinitas, pero están en lugares de difícil acceso.
- Los frutos son finitos, y otorgan más energía que las hojas.
- El jugador puede hacer que el mono haga deposiciones en lugares específicos, para esparcir semillas y así sembrar árboles. Esto al final del nivel se informa con el bosque recuperado.
- Si el mono se cae de gran altura, o es atacado pierde energía.
- Para recuperar energía el jugador debe comer hojas o frutos.
- El juego tiene un álbum que se puede ir llenando con la información de ciertos animales y plantas. Para colocar un animal o planta en el álbum hay que obtener la lámina en el nivel. Las láminas se consiguen mirando la información del animal o planta. Al inicio del juego, el único animal en el álbum es el mono.

5.1.6. Movimientos del Personaje principal

Árboles

- Si tiene hundido el botón de saltar, y está cerca de una rama en la que se puede colgar, el mono se cuelga con los brazos. Cuando se cuelgue el jugador puede soltar el botón de saltar.
- Si está colgado de una rama, puede hundir el botón de acción para subirse sobre la rama.

- Cuando está sobre una rama de la cual se pueda colgar, se puede hundir el botón de acción para que se enrolle con la cola a la rama.
- Si está colgado puede soltarse saltando.
- Si está colgado con los brazos o la cola, se puede columpiar hundiendo hacia adelante y hacia atrás.
- Cuando esté sobre una rama, el jugador puede caminar sobre ella con el control de dirección (joystick).
- Para trepar un árbol, el jugador se debe acercar al tronco y hundir el botón de acción.
- Para caminar sobre una rama, debe saltar (o dejarse caer) sobre ella.
- El mono no se cae de las ramas, tiene que saltar para bajarse de ellas.
- Cuando está trepando, tiene que saltar para soltarse, o llegar al fin del área donde puede trepar.

Terreno

- Para moverse el jugador utiliza el control de dirección:
 - Arriba es adelante.
 - Abajo es atrás.
 - Izquierda y derecha para dar vueltas un círculo. El radio del círculo lo da la cámara.
 - La cámara es similar a la de Súper Mario 64.

Otros

- EL jugador puede saltar cuando esta parado o corriendo, con el botón de saltar.
- El jugador puede dar una vuelta si presiona el botón de enrollarse (rodar), pero no en el aire.
- Si tiene una piedra la puede tirar con el botón de golpear (tirar).
- Cuando camina sobre frutas las recoge.
- Cuando hay un objeto del cual puede obtener información, se ve en la pantalla un ícono que indica esto.
- Cuando el jugador se acerca a un objeto que tiene información, el botón de acción sirve para ver esta información.
- El jugador puede apuntarles a los enemigos cercanos con el botón de apuntar (mira).
- Si está cerca de un objeto que puede recoger o usar, lo recoge o usa con el botón de acción.

Física

El mundo y sus objetos deben tener un comportamiento parecido al mundo real, pero limitándose a lo necesario para el juego:

- Debe haber gravedad.
- Algunos objetos podrán rebotar.
- El terreno y los objetos estáticos no son afectados por los golpes o el movimiento.

5.1.7. Menús del Juego

• Menú Principal

Es el menú que le aparece al jugador cuando inicia o termina el juego. En este menú tendrá estas selecciones:

- Juego nuevo.
- Álbum de animales y plantas.
- Créditos.
- Salir.

• Menú en Juego

Cuando el jugador hunde el botón de menú se abre el menú en el juego y el juego se detiene. Éste menú contiene:

- Álbum de animales y plantas.
- Mapa.
- Salir del Juego.

Tipos de arboles

Solo algunos arboles especiales funcionan de forma diferente, le permite al jugador realizar una acción especial con el personaje.

Columpio

Le permite al personaje saltar y columpiarse en un árbol.

Liana

Lleva al personaje de un lugar a otro, sin interacción del jugador.

Trepador

Tipo de árbol el cual puede escalar el jugador.

Pasamanos

Permite al personaje avanzar mientras se esta colgado de un árbol.

5.1.8. Requerimientos de Software

Los siguientes son los requerimientos generales de cada uno de los módulos del juego.

Motor Gráfico

- Cargar y dibujar modelos 3d con animaciones y materiales.
- Sistema de GUI.
- Hacer uso de luces dinámicas.
- Mostrar sombras dinámicas.
- Sistema para dibujar arboles y pasto.

Motor Juego

- Cargar y guardar archivos de preferencias de configuración del usuario.
- Actualizar y conectar los otros motores.
- Cargar de niveles.
- Proveer un sistema para manejar varios usuarios (perfiles).

Motor de Física

- Detectar y resolver colisiones entre objetos físicos.
- Reportar información de colisiones.
- Sistema de grupos para filtrar colisiones.
- Tener elementos disparadores para manejar interacciones.
- Elementos de interacción con el personaje: columpios, arboles.
- Simulación de los movimientos del personaje.

Motor de Entrada

- Configuración y detección de dispositivos entrada.
- Proveer la información sobre el estado del dispositivo de entrada.

- Configurar el dispositivo de entrada a través de un archivo de configuración.

Motor de Sonido

- Reproducir música y sonidos.
- Utilizar sonidos 3d.
- Permitir definidos sonidos cíclicos, o de una sola reproducción.

Motor de IA

- Mover naturalmente los animales del juego.
- Crear interacciones entre los personajes.

Editor de Niveles

- Importar objetos 3d.
- Crear, animar y aplicar los materiales a los objetos 3d.
- Permitir diseñar el escenario.
- Definir información específica de un nivel, como los objetos físicos.
- Exportar a un archivo el diseño de un nivel.

5.2. DISEÑO

5.2.1. Diagramas de Clase

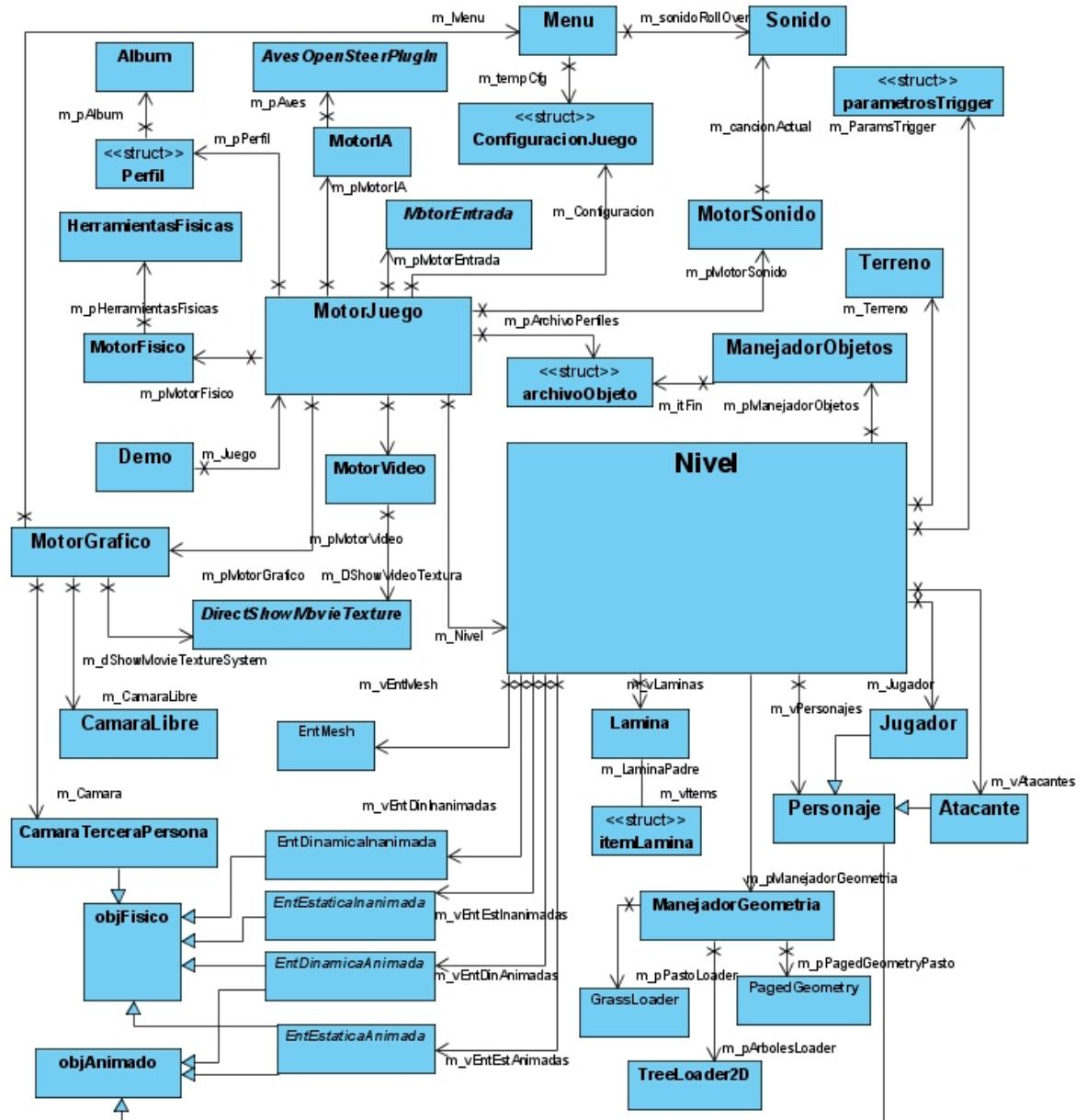


Figura 57. Diagrama de Clases más importantes

Diagrama de estados del juego

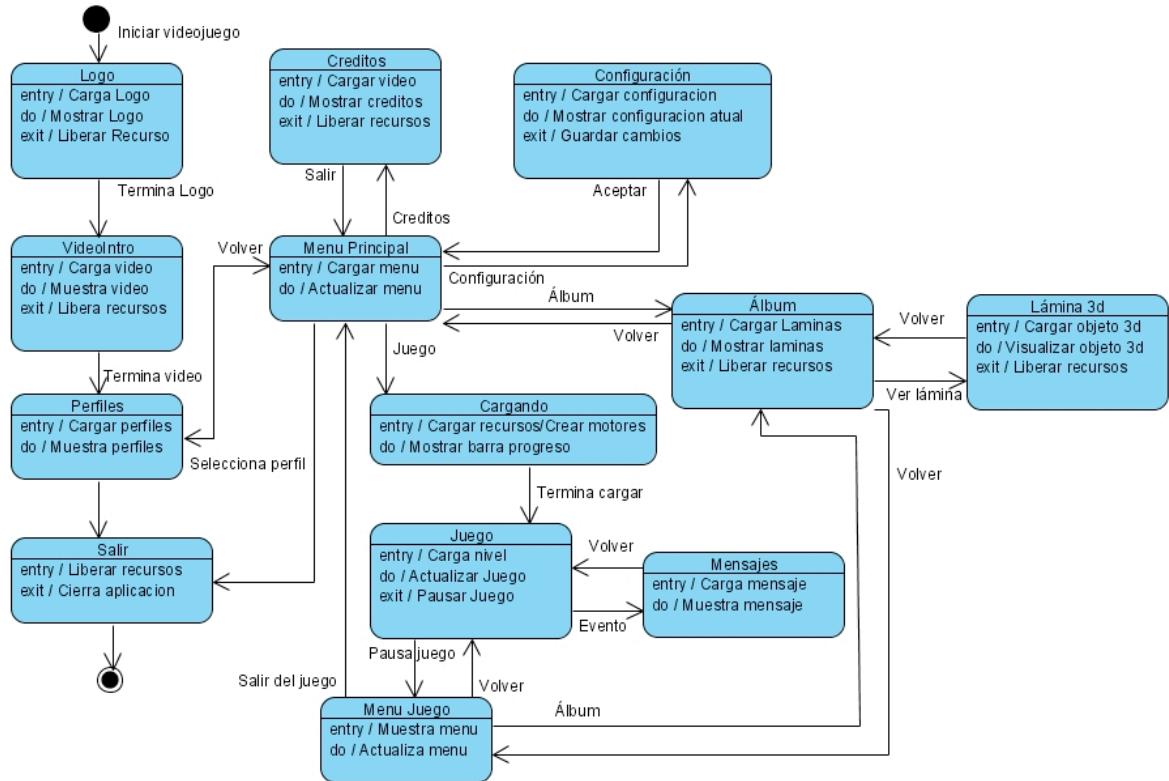


Figura 58. Diagrama de Estados

Logo. Se muestra el logo de la empresa

Video Introductorio. Se carga el video que introduce al jugador en la historia del videojuego.

Perfiles. Se espera a que el jugador seleccione su perfil, para ser cargado.

Menú principal. Se muestra el menú del videojuego. Se carga la música que acompaña el menú.

Configuración. El videojuego muestra los tipos de configuración disponibles:

- Configuración sonido
 - Configuración de juego

- Configuración de los gráficos
- Configuración del control

Cargando. Carga todos los motores del juego e inicializa los recursos.

Jugando. Se inicializa el juego y se muestra todo el nivel, el cual el jugador es libre de recorrer.

Menú del juego. El jugador pausa el videojuego y se muestra el menú de juego.

Álbum. Se carga el álbum con las láminas que el jugador haya recolectado.

Lamina 3D. Se carga la visualización de la lámina en 3d.

Mensajes. Se muestra un mensaje al jugador dependiendo del evento que haya ocurrido

Salir. Se destruyen todas las instancias de objetos, y se liberan los recursos.

Diagrama de estados del jugador

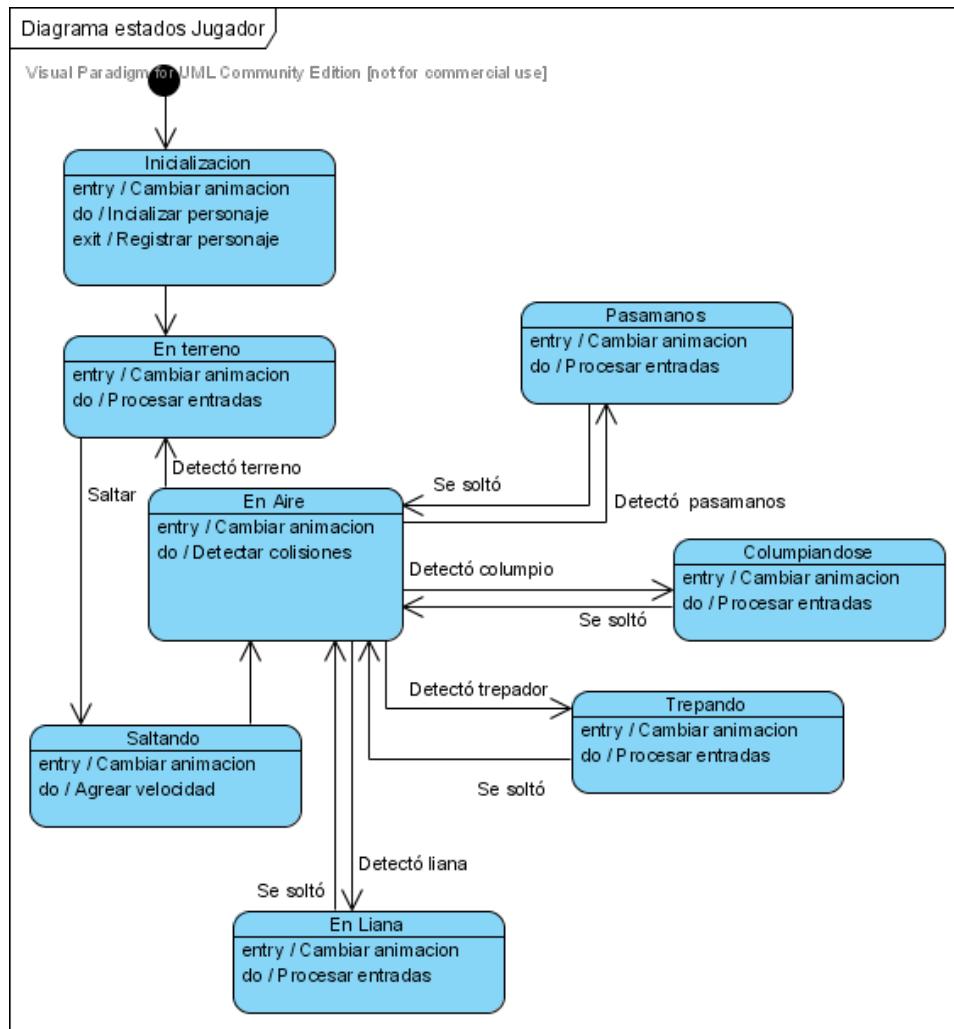


Figura 59. Diagrama de Estados del Jugador

Inicialización. Se inicializa el personaje, se carga el modelo3d, las animaciones y se crea el modelo físico.

En terreno. El jugador se encuentra tocando el terreno.

Saltando. El jugador presiona la tecla de salto, el jugador se encuentra ascendiendo.

En aire. El jugador esta en el aire descendiendo.

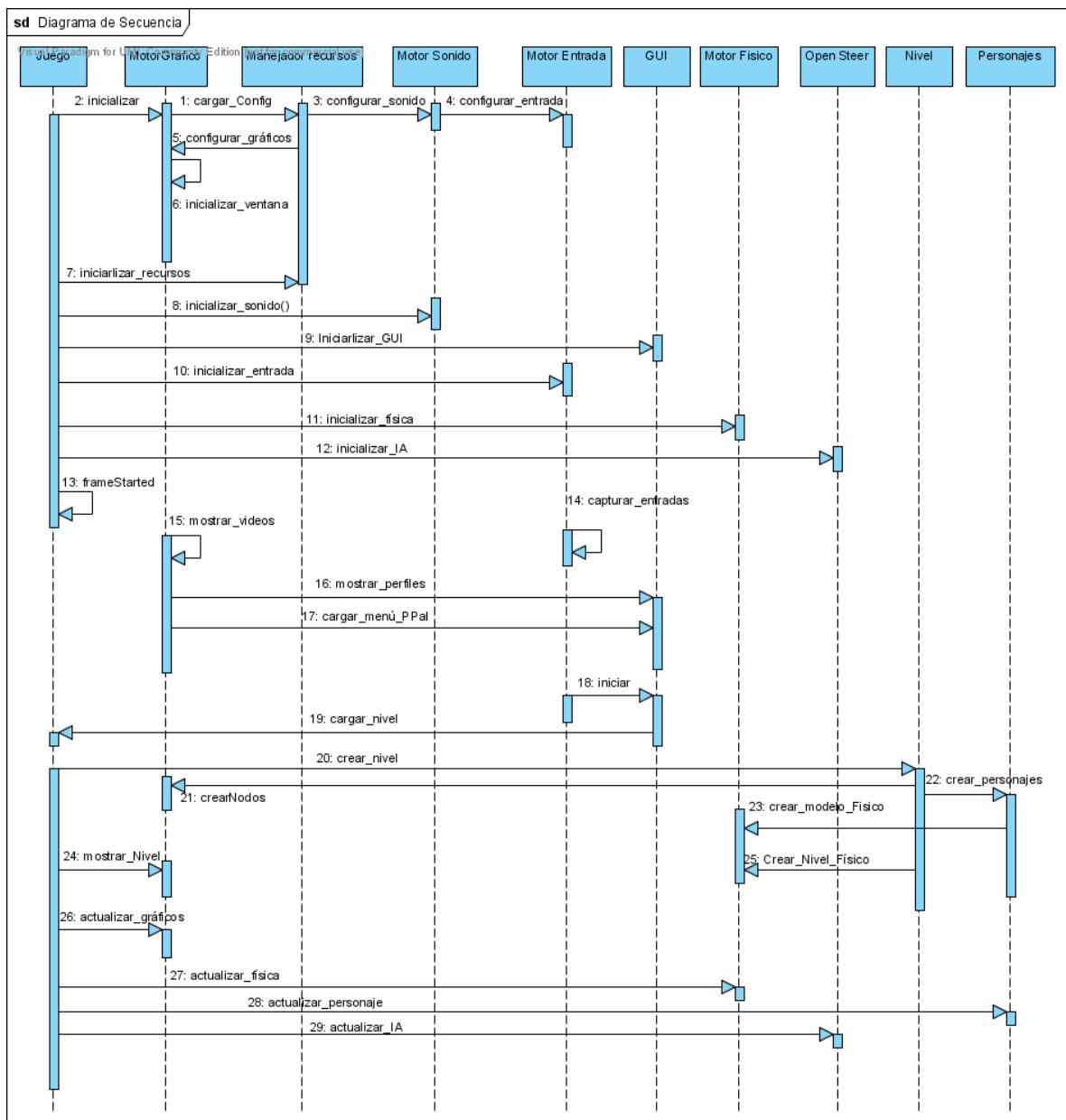
Pasamanos. El jugador esta colgado de un árbol de tipo pasamanos, la animación del personaje cambia.

Columpiándose. Se crea el columpio físico, cambia animación, al presionar el botón de adelante el jugador se columpia en el, al presionar saltar el personaje se suelta del columpio.

Trepando. El jugador esta unido a un árbol de tipo trepador, al presionar adelante el personaje asciende, al presionar atrás, el personaje desciende, al presionar saltar el personaje se suelta del árbol.

En liana. El jugador esta unido a la liana, y moverá con ésta.

Diagrama de secuencia



Diagramas de actividades del juego

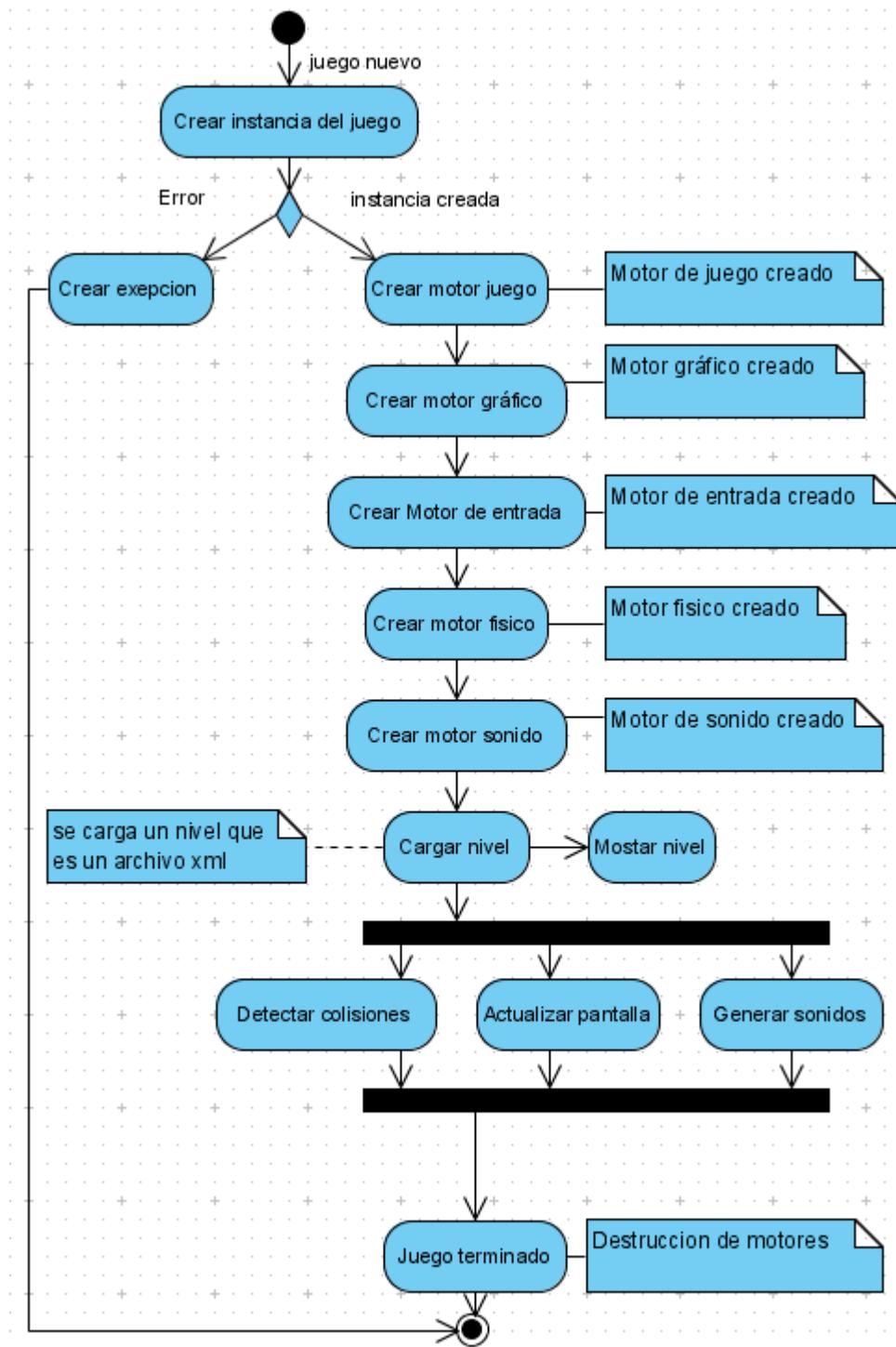


Figura 60. Diagrama de Actividades 1.

Crear Instancia. Se crea la instancia del juego

Crear excepción. Se muestra una ventana con el error.

Crear motor de juego. Iniciar los motores del juego.

Crear motor gráfico.

- Inicializar la ventana de la aplicación usando la configuración gráfica.
- Crear e inicializar el sistema de luz y sombras.
- Cargar los menús del juego.
- Cargar los recursos del nivel.

Crear motor de entrada. Inicializa los dispositivos de entrada.

Crear motor físico. Se crean los materiales y se asignan a los objetos. Se crean los sistemas de respuesta para los objetos.

Crear motor de sonido. Se inicializan los sonidos

Cargar nivel. Se lee el archivo de especificación del nivel y se carga

Mostrar nivel. Se da inicio a la renderización de imágenes

Detectar colisiones. Se detectan las colisiones entre los objetos y se llaman a las funciones que están declaradas para que ejecuten una acción predeterminada.

Actualizar pantalla. Se actualiza los visores y menús del juego

Generar sonidos. Se generan sonidos dependiendo de los eventos.

Juego terminado. Se liberan los recursos.

Diagramas de actividades Interacción GUI

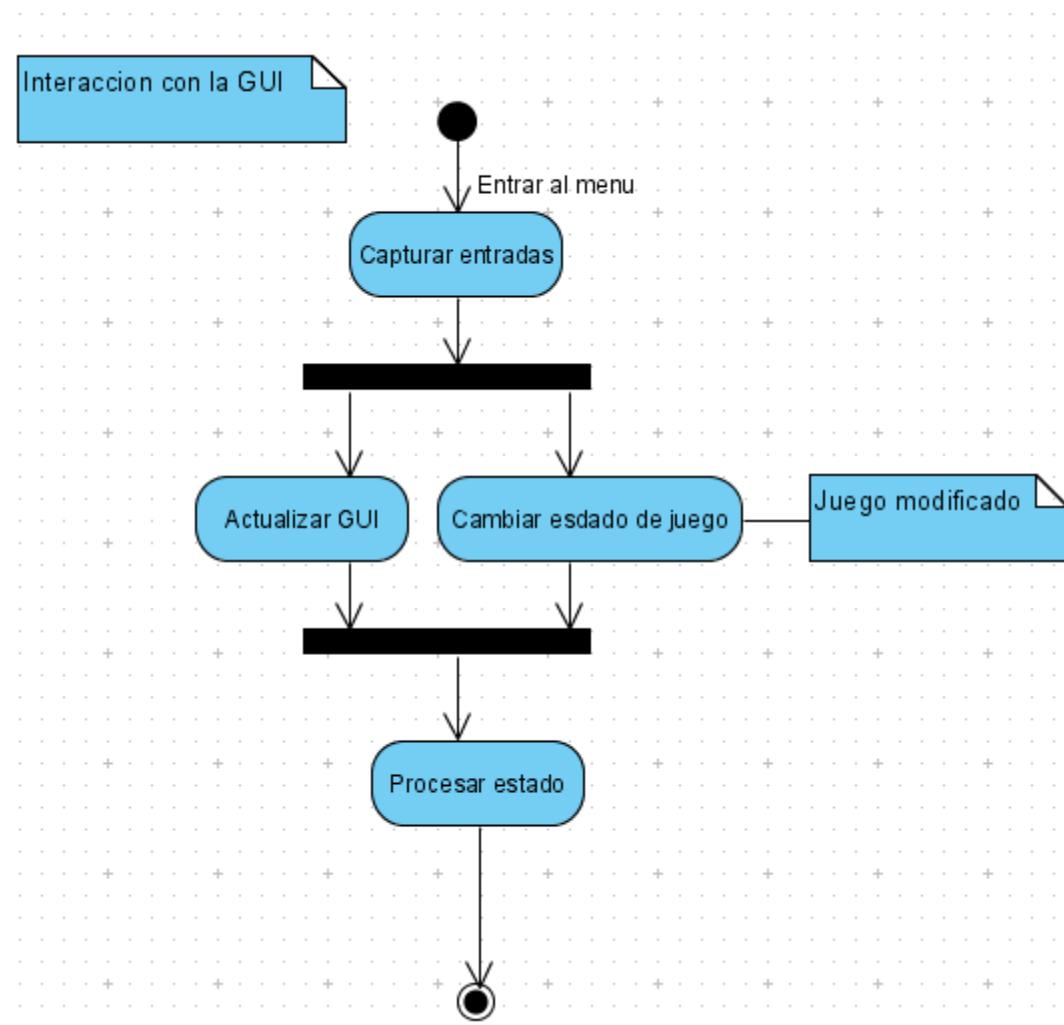


Figura 61. Diagrama de Actividades 2

Capturar entradas. Se actualizan los estados de los dispositivos.

Cambiar estado de juego. Se cambian el estado del juego.

Actualizar GUI. Se actualizan las imágenes y sonidos de la interfaz de usuario dependido del estado del sistema.

Procesar estados. Se hace un llamado a las funciones dependiendo del evento

Diagramas de actividades del jugador

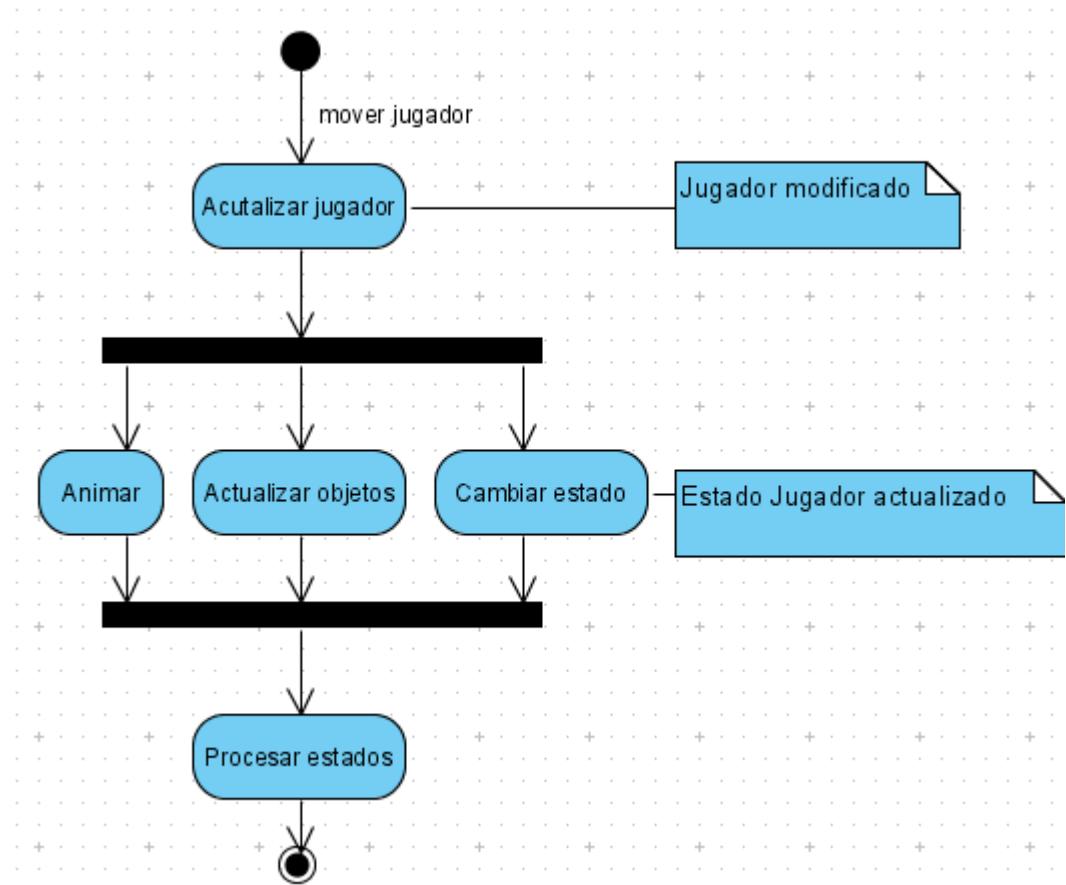


Figura 62. Diagrama de Actividades 3

Actualizar jugador. Se actualiza la posición del jugador.

Animar. Se llama a la función de actualizar las animaciones de cada objeto.

Actualizar objetos. Actualizar las posiciones y propiedades de los objetos.

Cambiar estado. Actualizar los estados de los objetos

Procesar estado. Se procesan los eventos que tiene relación con el jugador.

Diagrama de Componentes

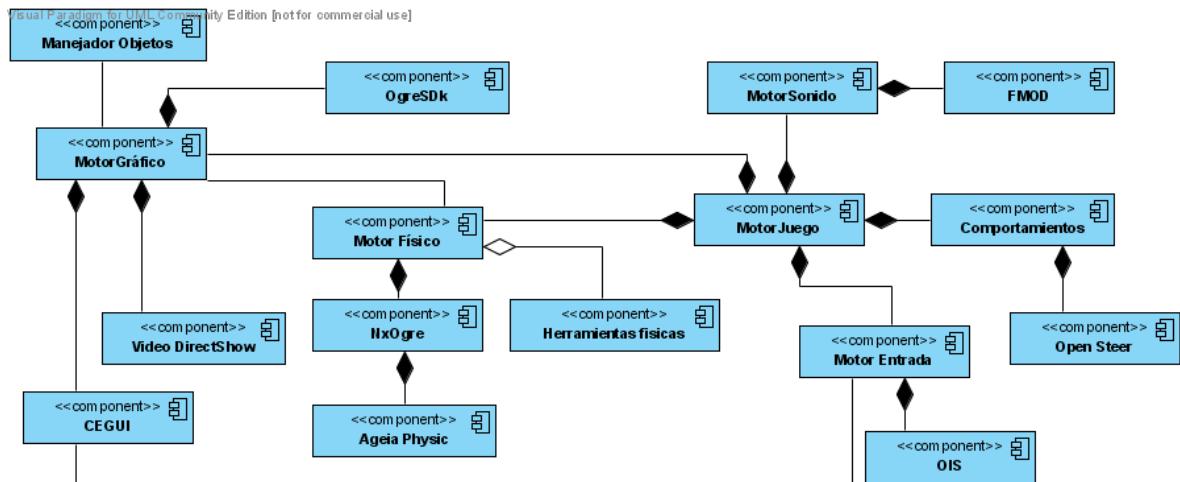


Figura 63. Diagrama de Componentes

MotorJuego. El corazón del sistema, el cual configura los demás componentes y carga el nivel del juego.

Ogre. Encargado de realizar todas las acciones que tengan que ver con mostrar y crear imágenes 3d para el juego.

Ageia PhysX. Encargado de realizar la simulación física

OIS. Encargado de capturar las entradas de los dispositivos de entrada.

FMOD. Encargado de proveer los métodos para reproducir sonidos

CEGUI. Encargado de crear la interfaz de usuario.

OpenSteer. Permite crear comportamientos para personajes autómata en videojuegos.

DirectShow. Framework multimedia creada por Microsoft para desarrolladores de software que necesiten utilizar archivos de media como video, sonidos.

Herramientas físicas. Conjunto de clases que permiten crear objetos de interacción del juego, como lianas, árboles para trepar, columpios, y árboles para trepar.

Manejador de objetos. Permite crear los objetos del juego, si un elemento esta repetido no se carga de nuevo si no que se duplica.

5.2.2. Diseño Arquitectónico

Secuencia de ventanas

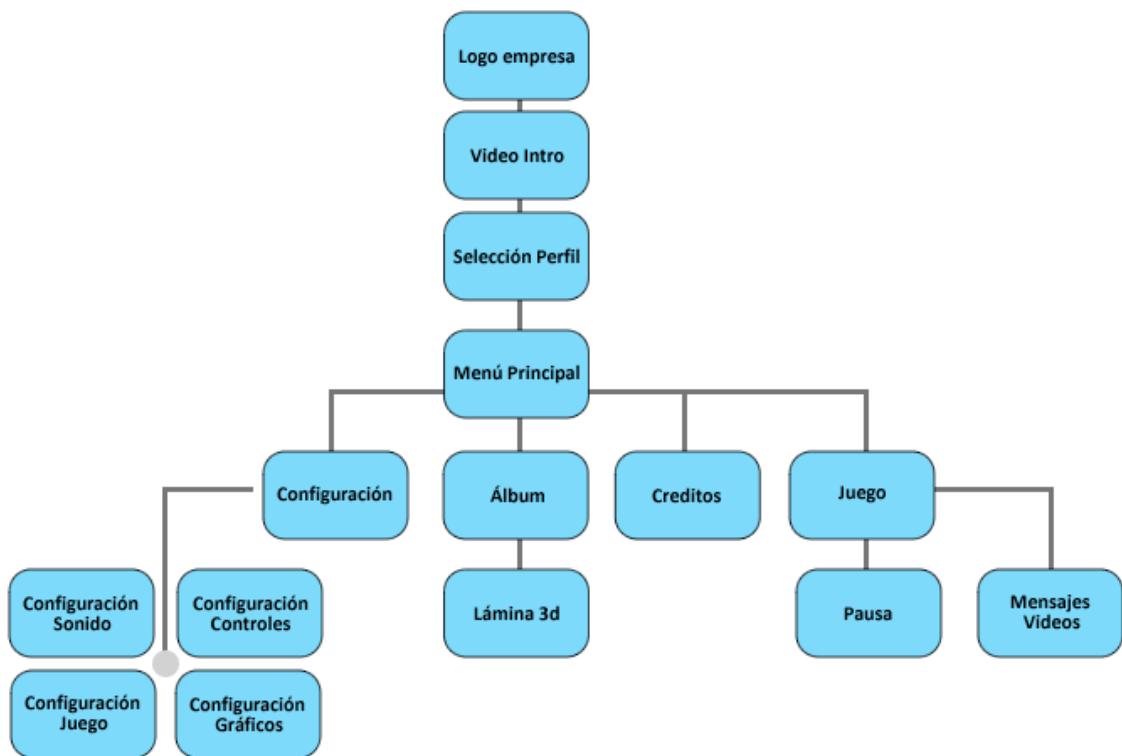


Figura 64. Secuencia de Ventanas

Logo Empresa

Al iniciar el juego se mostrará un video 3d de presentación de la empresa.



Figura 65. Logo de la Empresa

Video Introductorio. Se reproduce un video animada del juego. Después de cargar la introducción de la empresa, se iniciará el video introductorio del juego, donde se muestra de que se trata el juego en una combinación de animación 3d y secuencia de imágenes.

Selección Perfil. El jugador selecciona el perfil que haya sido previamente guardado

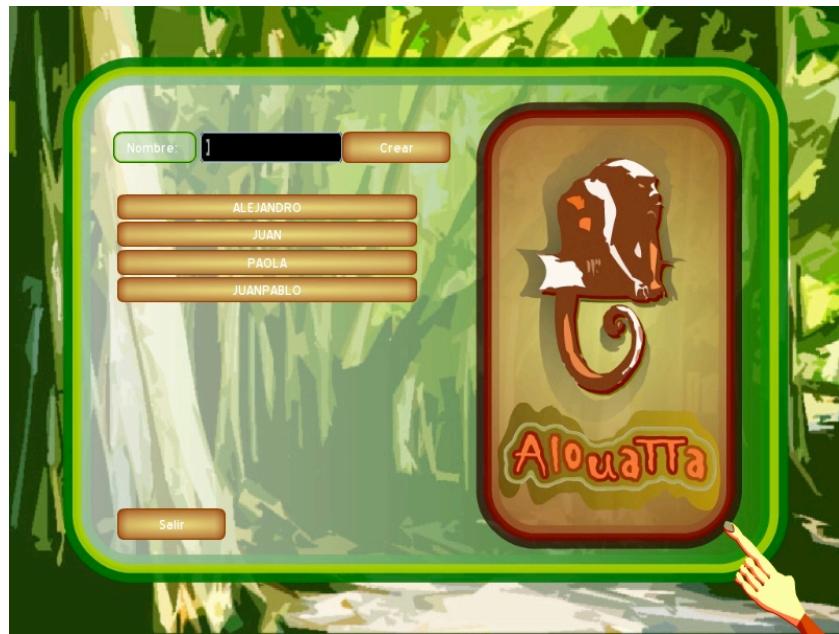


Figura 66. Menú de Perfiles

Menú Principal. El jugador selecciona si quiere iniciar el juego, cambiar las opciones del juego, ver el álbum, ver los créditos o salir.



Figura 67. Menú Principal

Configuración de Juego. El jugador selecciona el nivel de dificultad del juego



Figura 68. Menú de Configuración

Configuración de Sonido. Menú en el que se pueden ajustar los niveles de volumen.

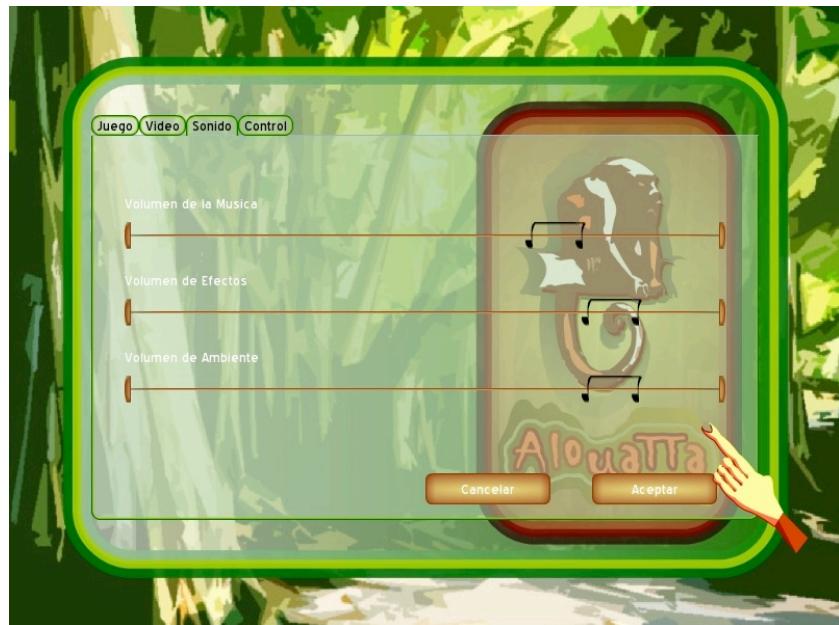


Figura 69. Menú de Configuración de Sonido

Configuración de Controles. El jugador configura los controles del juego usando el mouse. El sistema detecta automáticamente los joysticks.



Figura 70. Menú de Configuración de Controles

Configuración de Gráficos. El jugador puede escoger la resolución y escoger ejecutar el videojuego entre pantalla completa o ventana.

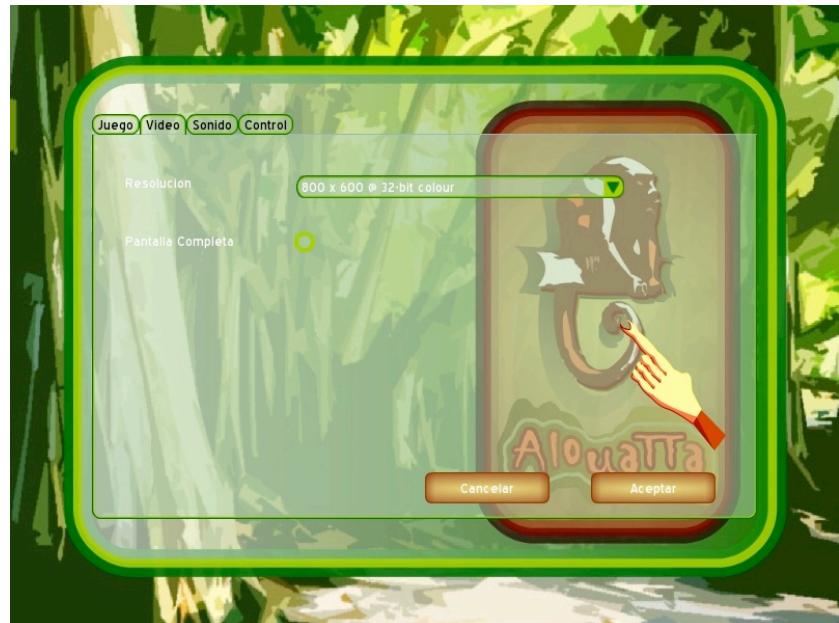


Figura 71. Menú de Configuración de Gráficos

Álbum

Al entrar en el álbum el jugador puede ver que animales ha colecciónado, y leer sus características.

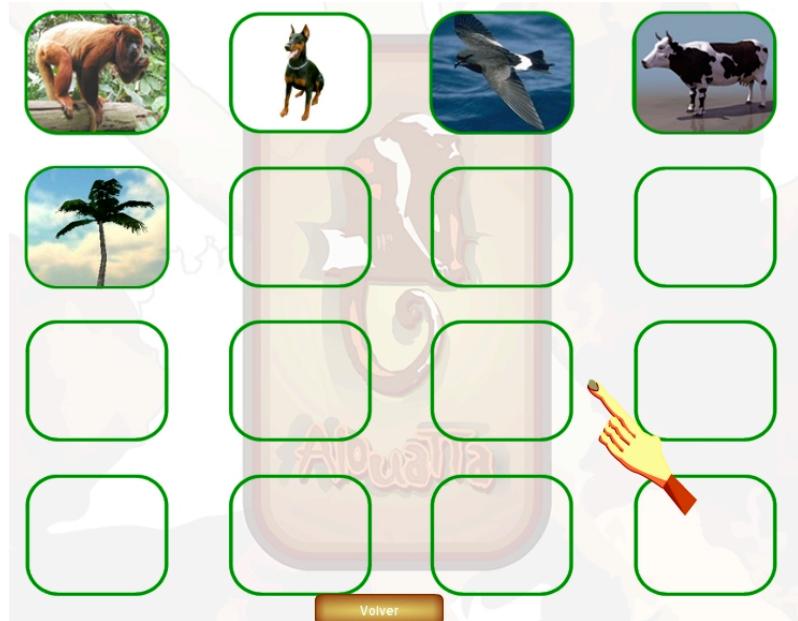


Figura 72. Menú de Álbum

Lamina 3D. El jugador puede observar al objeto en 3D, puede acercarlo y rotarlo. Además se incluye una descripción del objeto.



Figura 73. Lamina siendo examinada

Juego. Acá se inicia el juego, el usuario es libre de pausar o salir del juego en cualquier momento.



Figura 74. Captura de pantalla del juego en ejecución.

Menú al pausar. Al presionar el botón de pausa, el juego se detiene y se muestra el siguiente menú.



Figura 75. Menú en Juego

5.2.3. Implementación

La implementación del videojuego se realizó utilizando el compilador gratuito de Microsoft Visual C++ Express 2005⁶⁶, y la librería también gratuita para programación en plataformas Windows, Microsoft PlatformSDK⁶⁷.

Librerías intermedias

Se utilizaron también librerías intermedias específicas de las librerías usadas para acelerar el desarrollo:

NxOgre

Permite la fácil integración de la librería física PhysX con Ogre3D. Es una librería en continuo desarrollo a la par con los avances de Ageia.

Editable Terrain Manager⁶⁸

Es una librería para Ogre3D la cual incorpora un nuevo manejador de escenas, y permite la edición de terrenos: levantamiento y texturizado.

Diseño de modelos 3D

La creación de los modelos 3d se realizó en el software de modelado gmax debido a que los diseñadores tenían experiencia usando 3ds Max, del cual se produjo gmax, y además porque permite crear scripts para exportar información propia del juego. Cabe anotar que el gmax es gratuito.

Los modelos que usa el motor gráfico son exportados usando un exportador gratuito de la librería gráfico OGRE programado en MaxScript.

Diseño de Nivel

⁶⁶ <http://www.microsoft.com/spanish/msdn/vstudio/express/default.mspx>

⁶⁷ <http://www.microsoft.com/downloads/details.aspx?familyid=0BAF2B35-C656-4969-ACE8-E4C0C0716ADB&displaylang=en>

⁶⁸ <http://www.ogre3d.org/phpBB2/viewtopic.php?t=30955>

Para el diseño de los niveles se crearon objetos propios del juego usando MaxScript, los cuales se pueden agregar usando la interfaz de gmax. Los objetos creados permitieron añadirles propiedades físicas a los objetos y definir las propiedades específicas de los objetos del juego, tales como el tipo de objeto de juego, que en el momento de ser cargados a través de un archivo de nivel se relacion con una clase y/o unos parámetros de inicialización.

Los objetos usados para crear un nivel en gmax son:

- Nivel
- NxActor, NxBox, NxCylinder, NxSphere
- Pasto



Para crear un nivel se debe crear un objeto de tipo Nivel, el cual define las propiedades específicas del nivel, como la dirección del viento para la animación del pasto, los niveles de detalle de la vegetación para la configuración alta, media y baja de gráficos, el archivo y propiedades del terreno y las propiedades de la escena física. A través de este objeto se puede exportar el archivo con la información de nivel.

Los objetos que se vayan a usar en el nivel se deben cargar en el juego como objetos *Xref*, es decir, se crean de forma separada y se guardan en un archivo de gmax. Luego son instanciados en el Nivel usando *XRef*. Para que sean incluidos en el nivel se les debe aplicar un modificador de tipo *Objeto Alouatta*.

Este modificador es específico del juego y define los objetos que hacen parte de él. El archivo que define estos objetos se puede editar fácilmente para que se puedan cambiar por otros. A través de este modificador se puede seleccionar el tipo de objeto de juego que es. Si no es un objeto específico se puede escoger que sea de tipo *General*. Todos los objetos del juego deben tener un archivo separado con extensión *.alo* donde se definen sus propiedades físicas. Si solo se quiere colocar un objeto geométrico sin propiedades físicas se puede seleccionar el objeto *Mesh* del modificador *Objeto Alouatta*.

Los archivos *.alo* sirven para definir las propiedades de un objeto del juego. El desarrollo se hizo solo hasta el nivel de propiedades físicas, pero puede ser extendido a sonidos, animaciones, etc. ya que el archivo tiene formato flexible XML. Para crear el objeto de juego se le aplica el

Figura 76.
Parámetros del objeto
Nivel en gmax.

modificador *Entidad Alouatta* al objeto. A esta entidad se le asocia un objeto NxActor, el cual a su vez tiene asociados todos los objetos de tipo NxShape que definen las propiedades físicas del objeto. En el modificador *Entidad Alouatta* se selecciona el tipo de entidad que es, de cuatro opciones posibles:

- Entidad Dinámica
- Entidad Estática
- Entidad Dinámica Animada
- Entidad Estática Animada

En el producto final solo se implementaron entidades estáticas y dinámicas sin animaciones, así que si se selecciona una entidad tipo animada no será usada por el juego.

En el nivel se puede agregar también objetos físicos, a través de objetos NxActor. Un solo objeto NxActor es suficiente para incluir todos los objetos físicos del nivel, sin embargo se suele separar en varios actores mantener la edición mejor organizada. Los objetos tipo NxShape, es decir, los objetos NxBox, NxCylinder y NxSphere puede también actuar como disparadores y generar eventos en el juego. Los disparadores definidos y en uso en el juego son:

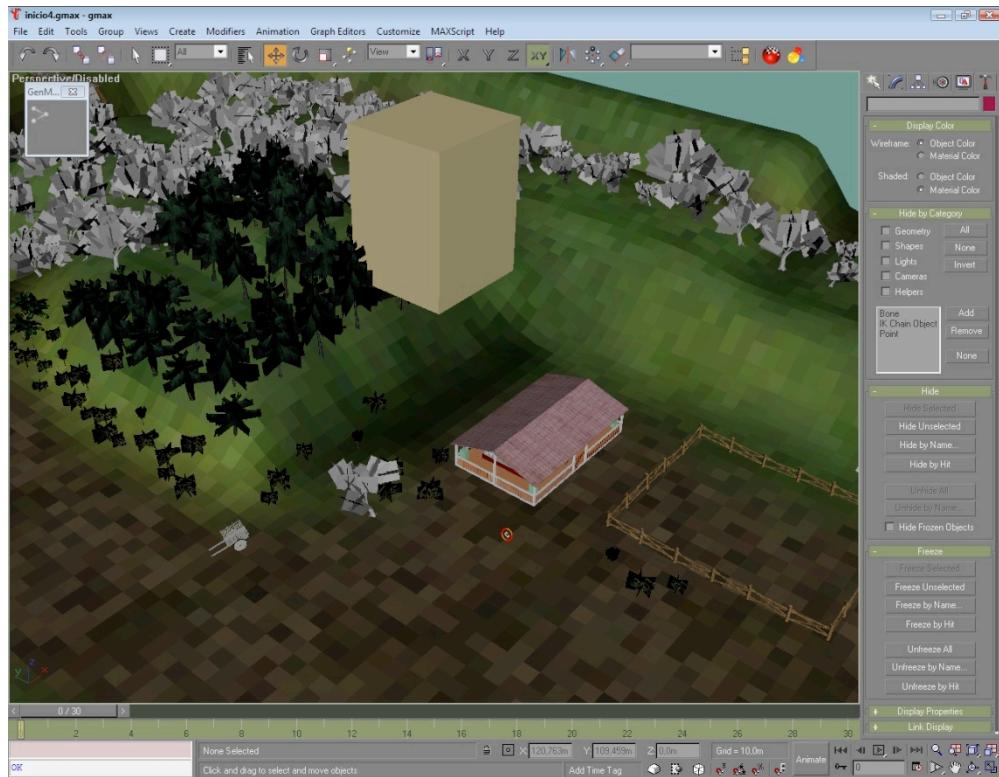


Figura 77. Captura de la edición del nivel con gmax

Terreno

Se utilizó una técnica de renderizado de terrenos, llamada Heightmap (POLACK, 2002), en la cual el terreno es creado dinámicamente a partir de un archivo de imagen. Esta técnica permite tener terrenos de gran extensión, livianos en consumo de memoria, de alto rendimiento y fácilmente reeditables.

Pasos para la creación del terreno.

1. Se crea el modelo 3D del terreno.
2. Se exporta como mapa de alturas.
3. Se carga en el ETM (Editable Terrain Manager).
4. Se pinta el terreno.
5. Se exporta como mapa de alturas.
6. El motor gráfico carga el mapa de alturas y genera la visualización tridimensional.
7. El motor físico carga el mapa de alturas y genera el modelo físico del terreno optimizado.

Física

Se decidió implementar la física debido a las siguientes características del juego

- Búsqueda de realismo.
- Interacción del personaje con el mundo.
- Habilidades del personaje como saltar, columpiarse, usar una liana, son implementadas con mayor sencillez a través de las herramientas del motor físico.

Para añadir objetos físicos en el nivel se utilizaron scripts en gmax. Se crearon dos formas para añadirlos. La primera insertando actores, que debían tener objetos de colisión añadidos a ellos. Estos actores se crea en el archivo de gmax de la escena y se exportan al archivo de nivel, en una etiqueta de xml en la que se definen todos los objetos físicos.

```
<Nivel version="0.1" >
.
.
.

<NxOgre>
  <NxActor Nombre="NxActor01" Tipo="Estatico" nShapes="10" Grupo="Defecto" >
    <Pos x="119.83" y="22.8425" z="-130.095" />
    <Rot x="0.0" y="0.0" z="0.0" w="1.0" />
    <NxShape Nombre="NxBox04" Tipo="NxBox" Trigger="true" >
      <Pos x="-6.73759" y="-13.3148" z="23.3044" />
      <Rot x="0.0" y="0.0" z="0.0" w="1.0" />
      <Trigger Flags="3" Grupo="Sensores" Texto="Perro_Finca" />
      <Box h="0.744664" w="18.7785" l="11.5909" />
    </NxShape>
  </NxActor>
</NxOgre>
</Nivel >
```

Figura 78. Ejemplo de una etiqueta física, con un actor el cual tiene un objeto de colisión de tipo NxBox (caja) que es a su vez un disparador que le avisa al motor de juego cuando el jugador esta cerca al personaje identificado con la cadena “Perro_Finca”.

Elementos físicos usando en el videojuego

Escenas

Se crea una sola escena, debido a que no necesitamos separar las escenas físicas, se configura la gravedad de la escena y como deben responder los objetos a las colisiones y demás elementos de la escena

Depurador

Visualizar los elementos físicos que se añaden a la escena para probar que estén funcionando correctamente puede ser una tarea compleja ya que, aunque son volúmenes que ocupan espacio, no son renderizados por el motor gráfico, a menos de que se implemente esta funcionalidad. Sin embargo esto no fue necesario para este proyecto puesto que PhysX provee una aplicación de visualización 3d que utiliza el modelo cliente-servidor, y permite que la aplicación pueda conectarse y enviarle la información física de escena, para que esta los dibuje en 3d. Es muy útil para corregir posiciones erróneas y ver las propiedades del objeto físico en tiempo real.

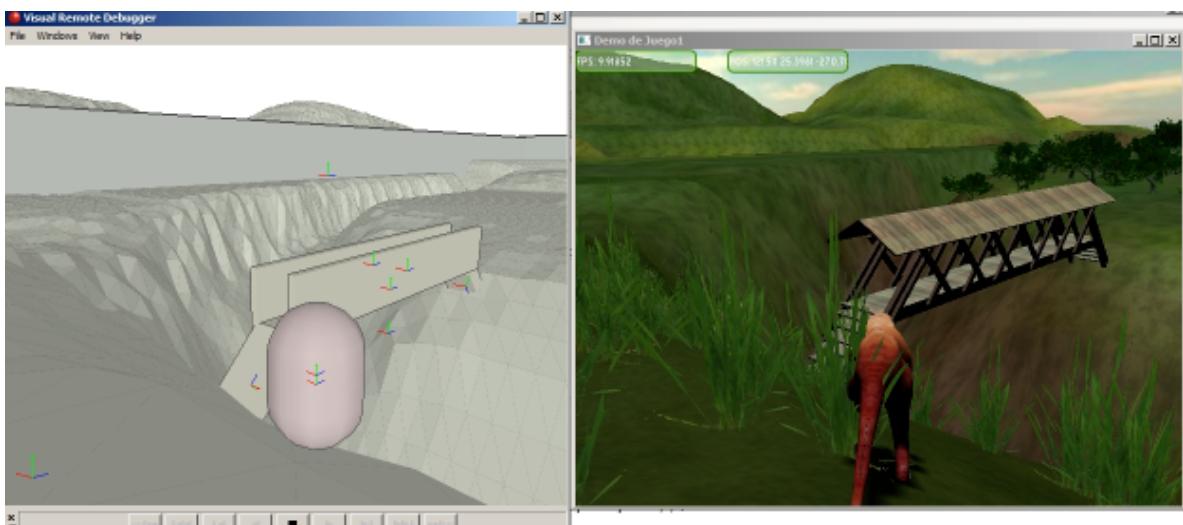


Figura 79. Imagen del juego y el depurador en ejecución

Control de personaje

Una característica muy importante para los juegos que usan personajes que se controlan directamente por el jugador, es que su movimiento sea natural. Para esto se debe configurar cuidadosamente el motor físico para que la simulación de movimiento se calcule adecuadamente. PhysX provee una clase específica para el control de caracteres. El personaje del juego utiliza una capsula para detectar colisiones y se mueve mediante la clase de caracteres. También se decidió usar otro objeto físico como ayudante para detectar colisiones con disparadores. Este objeto físico puede cambiar de forma dependiendo del lugar donde se encuentre el personaje.

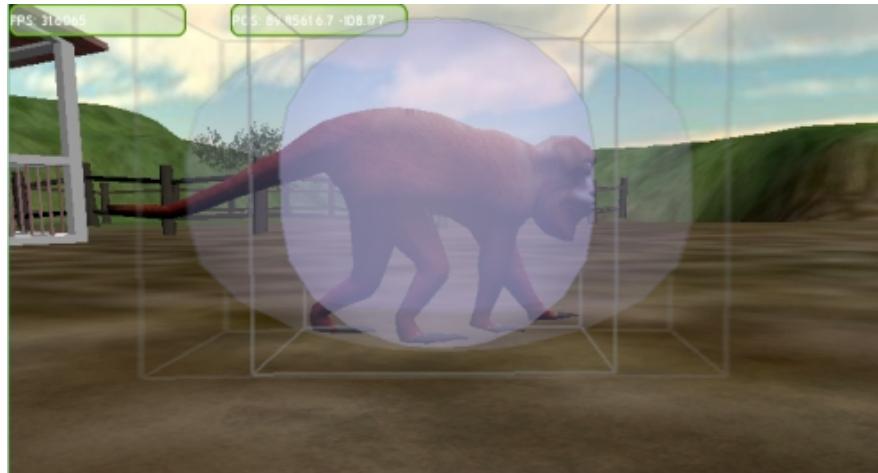


Figura 80. Objetos físicos de colisión del personaje.

Elementos de interacción

El motor físico se bastante flexible permitiendo la creación de elementos complejos donde el personaje principal puede interactuar, de forma similar a la realidad.

Se diseñaron cuatro elementos:

1. **Sistema de Liana.** El mono puede coger el extremo de un objeto cloth, balancearse y luego soltarse.
2. **Sistema de columpio.** Se creo utilizando las uniones del motor físico para simular una bisagra. Así el mono se puede colgar de ramas y columpiarse.
3. **Sistema de para trepar un árbol.** El mono puede acercarse a un tronco y subir por el eliminando temporalmente la gravedad.
4. **Sistema pasamanos.** El mono se puede colgar de una rama y desplazarse por la rama mientras esta colgado.

Fotos

Triggers

La mayoría de los eventos del juego se detectan utilizando triggers o disparadores. Estos triggers son objetos de colision, cajas, capsulas y esferas, los cuales se configuran para que cuando algún personaje se ponga en contacto con ellos genere una serie de eventos al motor fisico y así ejecutar una determinada acción. Los eventos que se pueden generar son:

- Algún objeto entro
- Algún objeto esta adentro
- Algún objeto salió

Gráficos

Paged Geometry⁶⁹

Es una extensión para Ogre la cual provee métodos optimizados para visualizar grandes cantidades de modelos estáticos que cubren un área posiblemente infinita, como arboles, rocas, arbusto, pasto, etc. La forma en que funciona es dividiendo la escena en paginas, o aéreas donde se almacenan todos los objetos que están dentro de ellas. El sistema actualiza dinámicamente cuales se muestran usando la posición de la cámara del la ventana. El solo muestra las páginas que estén a cierta distancia, que se puede configurar al gusto del diseñador. Además de esto permite dibujar diferentes versiones de los objetos dependiendo de la distancia. A distancias cercanas se puede utilizar la versión completa, y a distancias lejanas se puede mostrar solo una imagen del objeto, optimizando el uso de la tarjeta grafica. Otra característica es que puede animar el pasto definiéndole la intensidad y dirección del viento, que se puede cambiar en cualquier momento.

Debido a las características del videojuego Alouatta esta librería permite mostrar en los escenarios de exteriores, los arboles y hierba que se necesitan. Para la edición se le añadió al editor una funcionalidad extra para configurar en la escena la cantidad de objetos de vegetación, la dirección e intensidad del viento, y su posición y cobertura.

Sistema de cámaras

⁶⁹ Sitio Web http://www.ogre3d.org/wiki/index.php/PagedGeometry_Engine

La cámara debe permitir ver al personaje y a su alrededor de la mejor forma posible, es decir sin realizar movimientos bruscos, y manteniendo el enfoque siempre en el jugador. En la aplicación se creo una cámara que se siempre esta enfocada un poco mas arriba del jugador, permitiendo ver mejor la escena sin perderlo de vista. La cámara cumple con las siguientes características:

- La cámara sigue siempre al jugador.
- La cámara no cambia su rotación cuando el jugador lo hace para evitar movimientos bruscos.
- Si el jugador se queda quieto por varios segundos la cámara actualiza su posición.
- El jugador puede acercar y alejar la cámara para ver mejor el escenario.
- A través el control se puede rotar la cámara alrededor del personaje y aumentar y disminuir el ángulo vertical desde el cual se observa.

Como se construyó el código

Escenario de exterior (plantas y arboles)

TODO

Entrada

El Motor de Entrada es el encargado de suministrar a la aplicación la información sobre los dispositivos conectados, y proveer métodos para conocer cuales acciones se realizaron a través de ellos. Esta programado para recibir entradas del teclado, ratón y un joystick.

Para implementarlo se utilizo la librería OIS, la cual hace todo el reconocimiento y adquisición del estado de los dispositivos. OIS permite utilizar eventos que hacen llamados a funciones cada vez que se reciben o simplemente brindar información sobre el estado de los dispositivos. En la aplicación se decidió usar el segundo método porque brinda más control sobre cuando obtener el estado de los dispositivos, ya que los eventos se pueden recibir en cualquier momento, y dependiendo del estado de la aplicación se deben manejar o descartar. El método usado permite que se pregunte acerca del estado del dispositivo asociado a una acción del juego solo cuando se requiere. La desventaja de este método es que cuando la aplicación se pega por usar muchos recursos, no se alcanza a actualizar el estado de los dispositivos y se pierden entradas del jugador.

Al principio de cada frame se guardar el estado de las partes del dispositivo que se estén usando (teclas, botones, rueda del mouse, ejes del joystick) en un arreglo de enteros, y al final del frame

se guardan en otro arreglo de enteros, que guardan el valor dado al usar el dispositivo. Cuando se quiere saber si se uso alguna de estas partes se comparan los dos estados. Por ejemplo para el teclado, cuando se hunde una tecla, el estado anterior es cero, y el nuevo estado es uno. Cuando se suelta es el caso contrario. En el caso del movimiento del mouse, se indica en enteros cuanto se movió. Para los ejes del joystick se indica la posición respecto a un eje bidimensional que envía valores en los rangos de [-32768,32768] para los ejes x y y.

Álbum

El álbum se creó para recompensar al jugador por vencer los obstáculos del juego. Además de esta forma puede conocer información extra acerca de los elementos que conoce en el videojuego, como animales, árboles y lugares. Cada lámina del álbum es única y una vez recogida puede ser consultada desde el juego o desde el menú principal.

Características

- Muestra los modelos 3d del objeto al que pertenezca la lámina.
- Reproduce todas las animaciones del objeto si las tiene.
- Permite al usuario visualizar leer información extra acerca del objeto.
- Permite rotar el modelo para verlo desde diferentes ángulos
- Tiene controles para acercar y alejar la cámara del objeto.

Tiempo requerido para la implementación

6 meses

Herramientas usadas

Editor de código: Visual C++ Express Edition.

Editores de Gráficos

GimpShop, Irfanview

Otras herramientas

Editable Terrain Manager: Creación de textura para el terreno

Características de los equipos utilizados en el desarrollo

Computador 1

- Computador Dell 4700
- 1 Giga de memoria RAM
- Procesador Pentium 2.8 GHZ

- Tarjeta de video Ati Radeon X300
- Disco Duro 40 GB.

Computador 2

- Computador Dell Dimension 8200
- 512 Megas de memoria RAM
- Procesador Pentium 2.2 GHZ
- Tarjeta de video NVidia GeForce Ti 4600
- Disco duro 160 GB.

Creación de arte

Modelos 3d

Los modelos se crearon en gmax. Una vez terminados se exportaron utilizando el plugin de OGRE de maxscript para exportar objetos al formato de texto XML, que puede luego ser convertido el formato binario *.mesh*, nativo para los modelos 3d en la librería de gráficos OGRE. Los materiales se aplicaron utilizando el material Standard de gmax, usando una imagen para el color difuso. A través del manejador de materiales también se pudieron configurar propiedades de la iluminación como los valores para la luz ambiente, difusa y especular.

Utilizando el mismo plugin de OGRE se exportaron a archivos de texto que se guardaron con la extensión *.material*. En algunos casos se editaron estos archivos manualmente dependiendo del resultado esperado.

Si el modelo necesitaba definir objetos físicos, se podía hacer a través del modificador entidad. Al ser aplicado se puede escoger el tipo en entidad (dinámica o estática) y relacionarlo con un actor con las propiedades físicas, el grupo de actores al que pertenece y los objetos de colisión, los cuales pueden ser triggers. Estos objetos del juego con propiedades físicas se exportan en un archivo de texto, con estructura XML y extensión *.alo*.

Animación de personajes

Gmax permite utilizar esqueletos para animar objetos, y el exportador de OGRE de igual forma permite exportar las animaciones creadas en gmax. Una vez creadas y probadas las animaciones, se utiliza el exportador de maxscript de Ogre, en el cual se definen las animaciones que se definen por el rango de frames en que ocurren y por un nombre que las identifica. Este exportador crea dos archivos de texto en formato XML, uno con la información del objeto y otro con la información del esqueleto y las animaciones de ese objeto. Usando el conversor de Ogre se convierten en

archivos *.mesh* y *.skeleton* en formato binario nativos de OGRRE. Ambos usan el mismo nombre para asociarse.

Cuando se carga el objeto en la aplicación se pueden utilizar los métodos de Ogre para obtener la información del esqueleto y las animaciones, y con ella, animar los personajes a la velocidad que se desee.

Arte 2d

Sonidos

GUI

El siguiente gráfico muestra algunos de los elementos del videojuego y como funcionan de forma general en el juego.

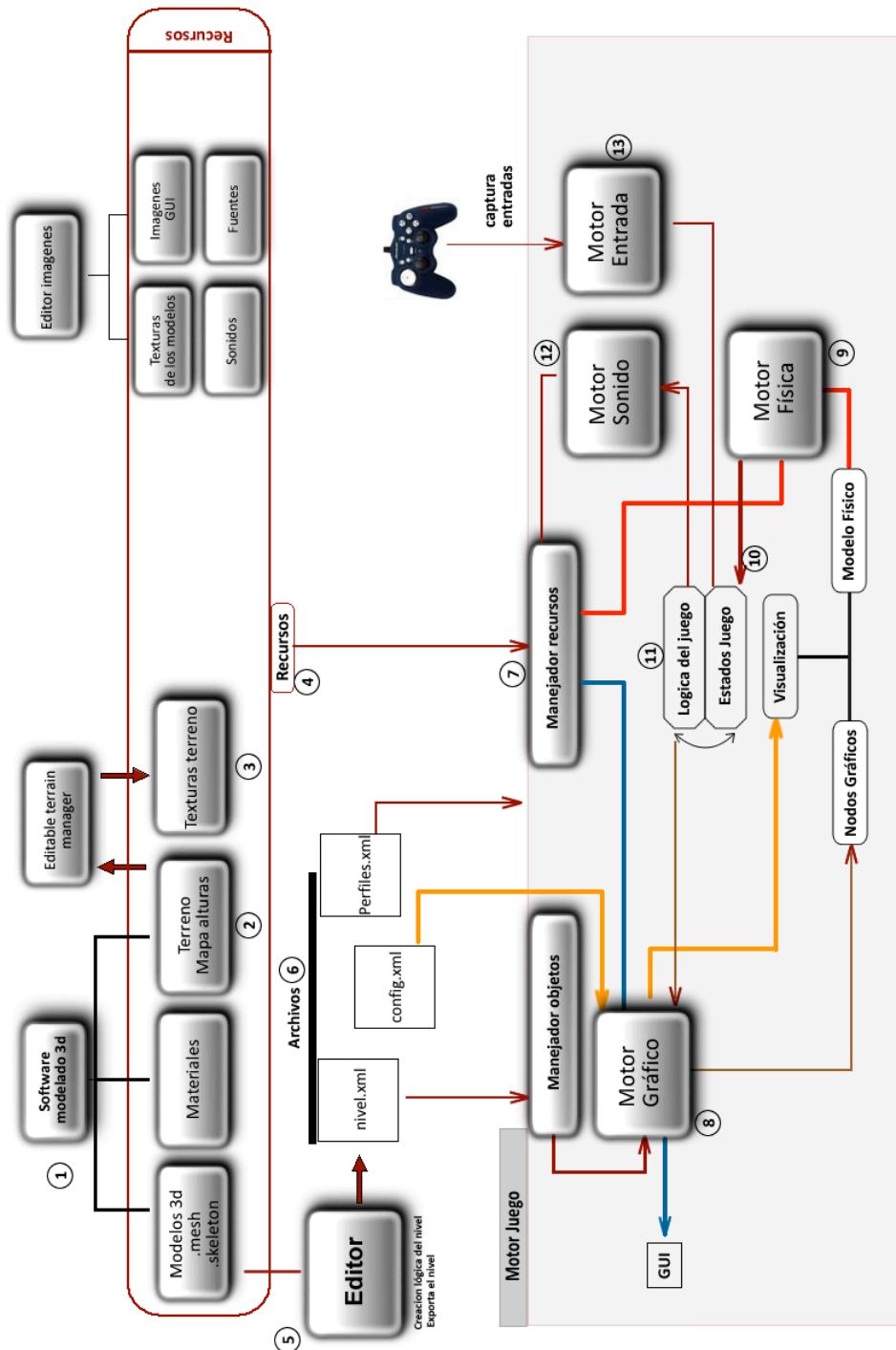


Figura 81. Flujo del Motor de Juego

1. Modelado

Inicialmente en un software de modelado 3d se crean los objetos 3d que van a estar en el videojuego tales como personajes, terrenos, y objetos de ambientación, estos son exportados en un formato entendible para el motor gráfico, en el caso del motor Ogre son **.mesh** para modelos y **.sketelon** para las animaciones de los modelos, y se exporta también un archivo que define la textura que tendrá el modelo conjunto con la iluminación propia del modelo y características avanzadas como shaders o rutinas avanzadas. Anterior a esto el diseñador debe crear imágenes que servirán para texturizar para cada modelo 3d e imágenes para las interfaces de usuario. Para el videojuego Alouatta se necesitó crear un modelo de terreno bastante grande, como se mencionó anteriormente esto se implementó con un mapa de alturas para mejorar el rendimiento de los motores gráficos y físicos.

2. Mapa de alturas

El terreno que se ha modelado se exporta como imagen de mapa de alturas con el software de modelado 3D.

3. Textura del terreno

Esta imagen es luego cargada en un software utilitario desarrollado por los usuarios de Ogre, ETM(Editable Terrain Manager) debido a que permite la fácil edición de terrenos y creación de texturas para este, al dibujar sobre el terreno los colores que se requieren y visualizando el resultado final directamente. Se exporta una imagen la cual sirve de textura del terreno.

Otra forma para generar la textura del terreno es crearla en cualquier programa de edición de imágenes como comúnmente se hace para los modelos 3D.

4. Recursos

Los sonidos idealmente deben ser creados por un ingeniero de sonido en un estudio de grabación, si se cuenta con el presupuesto, si no deberán ser adquiridos de otras formas por ejemplo buscando librerías de sonidos gratuitos en Internet. Las fuentes, imágenes, modelos y sonidos conforman los recursos del videojuego.

5. Edición del nivel

Como se explicó anteriormente en el editor se crea lógicamente el nivel, se posicionan los objetos como deben estar en el juego y se agrega información pertinente al nivel.

Luego un archivo es exportado el cual contiene la información de como debe configurarse el nivel del videojuego e información para eventos y mensajes. Esta labor es realizada por personas especializadas en diseño de videojuegos.

6. Configuración del juego

El archivo de nivel permite configurar el nivel, pero existen otros archivos que configuran las demás opciones de un juego. El archivo de perfiles carga los valores guardados por los jugadores que llevan en el transcurso del videojuego, y el archivo de configuración carga las preferencias que se hallan guardado en cuanto a gráficos, sonidos, jugabilidad y dispositivos de entrada, si estos archivos no existen al inicio de un sesión el motor deberá crearlos.

El archivo de perfiles permite configurar parámetros del motor de juego, el archivo de nivel deberá ser cargado por un manejador de objetos que permite cargar los objetos y hacer un uso eficiente de la memoria. El archivo de configuración permite configurar el motor gráfico.

7. Manejador de recursos

El manejador de recursos deberá cargar todos los recursos del videojuego, como lo son texturas, modelos, sonidos, scripts, fuentes, materiales y permitir el fácil acceso a estos para los demás motores.

8. Motor Gráfico

El motor gráfico por medio de las librerías para interfaz de usuario dibuja el menú del usuario. El motor gráfico crea y configura los nodos gráficos que se van a renderizar en la escena, y se definen los modelos que están unidos a estos nodos y demás propiedades visuales. El motor grafico renderiza todos los modelos 3d y 2d.

9. Física

El motor de física crea modelos físicos de colisión y los asocia a nodos gráficos, de modo que si un modelo físico es cambiado de posición el nodo visual como se explicó anteriormente.

10. Reporte de colisiones

El motor físico detecta colisiones, y los objetos de tipo disparadores envían eventos al motor físico los cuales pueden cambiar los estados del juego.

11. Lógica del juego

12. Motor de Sonido

Dependiendo del estado del juego y de los eventos generados por el videojuego el motor de sonido reproducirá un sonido definido.

13. Motor de Entrada

El motor de entrada captura las entradas para que el motor de juego pueda conocer el estado de los dispositivos en cualquier momento.

6. CONCLUSIONES

Luego de haber elaborado la recopilación de información acerca de las librerías y herramientas de software libre y gratuito para la elaboración de videojuegos, y desarrollado una propuesta de videojuego para demostrar algunas de las características de las librerías de software, se pudieron obtener resultados que permiten presentar las siguientes conclusiones:

La creación de un videojuego es un proceso complejo que incorpora tanto elementos de software como elementos de contenido. Este contenido puede incluir modelos de geometría 3d, sonidos, imágenes, scripts, archivos de nivel, entre otros. El proceso de elaboración de este contenido para un producto comercial generalmente requiere de personal profesional capacitado en la elaboración de cada uno de estos. De forma general este proceso se puede resumir en los siguientes pasos:

1. Diseño de la idea de juego.
2. Elaboración de prototipo.
3. Análisis del Sistema.
4. Diseño del Software.
5. Implementación y elaboración del contenido.
6. Pruebas.
7. Correcciones.
8. Publicación.

El software del videojuego utiliza distintas tecnologías para brindar la experiencia interactiva multimedia. Las tecnologías más comunes que se implementan son las de gráficos 3d, reproducción de sonidos monofónicos, estereofónicos, y 3d, simulación física, inteligencia artificial, comunicación en redes, sistemas de interfaz y dispositivos de entrada. Cada una de estas áreas se expande en sub-áreas especializadas y que pueden ser muy complejas. Por esto se hace necesario utilizar librerías existentes para ahorrar tiempo y recursos en el desarrollo del videojuego.

Actualmente existen librerías de código libre, disponibles para todos los que quieran producir sus propios videojuegos. Estas librerías cuentan con comunidades activas que sirven de soporte al desarrollo de ellas mismas y a quienes las utilizan. La mayoría de las librerías tienen licencias que permiten usarlas para crear software comercial. La única limitación generalmente es la licencia GPL, la cual exige liberar los cambios que se le hagan al código de la librería. También existen

herramientas para la creación de contenidos, de código libre y gratuitas, tales como modelos 3d, edición de imágenes, sonidos, compiladores, entre otros.

Se pudo hacer el diseño de un videojuego utilizando algunas técnicas de diseño de videojuegos existentes. Es muy importante tener en cuenta que la jugabilidad es más importante que el realismo en un videojuego, existen elementos que ayudan a recrear nuestra realidad pero el objetivo de un juego es entretenir, en ocasiones no hace falta una simulación totalmente realística para que un videojuego sea bueno y se venda.

Con este desarrollo se demuestra que es posible crear videojuegos con poco presupuesto basándose en herramientas que se encuentran en el mercado de uso gratuito o de bajo costo y de calidad comercial.

Las características de los videojuegos son útiles para crear otro tipo de aplicaciones que se pueden ver beneficiadas por su posibilidad de mostrar aplicaciones ricas en contenido multimedia de alta calidad interactivo. Es así como se distingue una oportunidad de negocio en la economía regional, nacional y mundial, como también líneas de investigación no abordadas por la Universidad Tecnológica de Pereira.

7. RECOMENDACIONES

El desarrollo de un videojuego es una labor que debe ser realizada por profesionales. El modelo del programador de garaje ya no sirve. Por eso una empresa que planea desarrollar un videojuego debe equiparse de profesionales de todas las áreas que necesite, como programadores, artistas gráficos, ingenieros de sonido, productor, entre otros.

Aunque este trabajo demuestra algunas de las tecnologías actualmente en uso, apenas hace un sobrevuelo por cada una de ellas. Existen temas que no se pudieron profundizar y además el constante cambio de la industria hace que los que si se vieron puedan hacerse obsoletos rápidamente. Se recomienda consultar permanentemente diferentes fuentes de información acerca del desarrollo de juegos y evaluar los juegos que están siendo lanzados al mercado para permanecer actualizado.

El diseño de software del videojuego Alouatta sirve como base para una implementación completa. El videojuego muestra de forma didáctica aspectos importantes de la flora y fauna colombiana, creando así una forma más interactiva de conocer los elementos de la biodiversidad del país. Se puede reutilizar el diseño en otros proyectos similares.

Este trabajo sirve como punto de partida para posteriores trabajos que quieran seguirse desarrollando en el tema de los videojuegos en la Universidad Tecnológica de Pereira. Sería beneficioso para la academia profundizar cualquiera de los temas tratados en el documento como gráficos, física, sonido, entradas, diseño de videojuegos, inteligencia artificial, scripting etc.

8. APORTES

Se entrega un documento que sirve como guía en idioma español para conocer las principales características y conceptos teóricos prácticos de los componentes de un videojuego o aplicaciones para visualizaciones tridimensionales en tiempo real. En el documento se tocan temas como gráficos 3d, simulación física, manejo de controles, reproducción de video, reproducción de sonido y elaboración de contenido 3d.

Se entregan tutoriales en idioma español creados en el transcurso del proyecto los cuales permitirán a cualquier persona con conocimientos de programación en C++ que quiera aprender a construir aplicaciones de tiempo real interactivas, introducirse en la programación con la librería Ogre3d.

Se creó la empresa GenMedia dedicada al desarrollo de videojuegos y aplicaciones multimedia que hasta la fecha se encuentra vigente y vinculada a Parquesoft encontrando un nuevo nicho de mercado poco explorado en Colombia. La creación de este tipo de empresas incentiva a otras a trabajar en el tema.

Se presenta unas tablas comparativas de motores de gráficos y físicos actuales donde se “ponen x en” las características de cada uno y sirve como guía previa para la elección de alguno de ellos.

Se dio una conferencia en la Universidad de Caldas en Octubre del 2006 sobre construcción de videojuegos. Se dictó un curso de programación de videojuegos para 20 estudiantes de un total de 40 horas en esta universidad. Se está planeando replicar la conferencia en la UTP.

Bibliografía

Alouatta Seniculus. (s.f.). Recuperado el 14 de Septiembre de 2007, de Convenio Andres Bello:

http://www.convenioandresbello.org/cab3/sibd4/index.php?option=com_content&task=vie w&id=60&Itemid=58

BARTLE, R. A. (2003). *Designing Virtual Worlds*. New Riders Games.

BLINN, J. F. (1977). *Models of light reflection for computer synthesized pictures*. New York: ACM Press.

BOURG David M., S. G. (2004). *AI for Game Developers*. O'Reilly.

CAILLOIS, R. (1986). *Los Juegos y los hombres: la máscara y el vértigo*. México: F.C.E.

CASTRO, M. (2002). *Los libros de Arena*. Bogota: Pontificia Universidad Javeriana.

CRAWFORD, C. (2003). *Chris Crawford on Game Design*. New Riders Games.

David M. Bourg, G. S. (2004). *AI for Game Developers*. O'Reilly.

FIEDLER, G. (2 de Septiembre de 2006). *Physics in 3D*. Recuperado el Septiembre de 2007, de Gaffer on Games: <http://www.gaffer.org/game-physics/physics-in-3d>

FLYNT, J. P., & SALEM, O. (2004). *Software Engineering for Game Developers*. Course Technology PTR.

GALLARDO, A. (2000). *3D Lighting: History, Concepts, and Techniques*. Charles River Media.

Gutschmidt, T. (2003). *Game Programming with Python, Lua and Ruby*. Premier Press.

GUTSCHMIDT, T. (2003). *Game Programming with Python, Lua and Ruby*. Premier Press.

- HELGELAND, A., & ANDREASSEN, Ø. (s.f.). *Introduction to Computer Graphics and Visualization*. Obtenido de <http://prosjekt.ffi.no/unik-4660/lectures04/chapters/Introduction.html>
- JUNKER, G. (2006). *Pro OGRE 3D Programming*. Apress.
- LERMA, H. (2005). *Presentacion de Informes*. Bogotá: Ecoediciones.
- LEVIS, D. (1997). *Los videojuegos, un fenómeno de masas*. Barcelona: Paidos.
- MARCUS, F. (21 de Enero de 2003). *What Designers Need to Know About Physics*. Recuperado el Septiembre de 2007, de Gamasutra: http://www.gamasutra.com/resource_guide/20030121/marcus_01.shtml
- MCLAURIN, M. (21 de Enero de 2003). *Engine, Outsourcing Reality: Integrating a Commercial Physics*. Recuperado el Septiembre de 2007, de Gamasutra: http://www.gamasutra.com/resource_guide/20030121/maclaurin_02.shtml
- MCREYNOLDS, T. (2005). *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann.
- PHONG, B. T. (1973). *Illumination for computer-generated images*. ACM Press.
- POLACK, T. (2002). *Focus On 3D Terrain Programming*. Course Technology PTR.
- ROUSEIII, R. (23 de Marzo de 2000). *Designing Design Tools*. Recuperado el Septiembre de 2007, de Gamasutra: http://www.gamasutra.com/features/20000323/rouse_01.htm
- ROUSEIII, R. (2001). *Game Design: Theory and Practice*. Wordware Publishing, Inc.
- RUCKER, R. (2002). *Software Engineering and Computer Games*. Addison Wesley.
- RYAN, T. (17 de Diciembre de 1999). *The Anatomy of a Design Document*. Recuperado el Septiembre de 2007, de Gamasutra: <http://www.gamasutra.com>
- SANCHEZ-CRESPO, D. (2003). *Core Techniques and Algorithms in Game Programming*. New Riders Games.
- TORRANCE, K. E., & SPARROW, E. M. (1967). *Theory for off-specular reflection from roughened surfaces*.

Anexos

Anexo 1. Diagramas de Clase

Álbum

Album
#m_pEntidadAlbum : Entity*** #m_pNodoObjeto : SceneNode*** #m_vAlbum : StringVector #it : iterator #itFin : iterator #m_vAnimaciones : StringVector #m_pAnimationSet : AnimationStateSet*** #animacionActual : String #animacionAnterior : String #m_bAnimandoAlbum3d : bool #m_bMovObjetoArriba : bool #m_bMovObjetoAbajo : bool #m_bMovObjetolzq : bool #m_bMovObjetoDer : bool =>+Album() =>+~Album() +cargarObjeto(mesh : String) : void +liberarObjeto() : void +girarObjetoX() : void +girarObjetoXn() : void +girarObjetoY() : void +girarObjetoYn() : void +pararAnimarObjeto() : void +play() : void +siguienteAnimacion() : void +agregarLamina(nombre : String) : void +tieneLamina(lamina : String) : bool +cargarAlbum(pArchivoXML : void***, nombreJugador : String) : bool +mostrarAlbum() : void +guardarAlbum() : void =>+getEntidadAlbum() : Entity*** =>+getAlbumXML(: void) : String =>+setAnimandoAlbum3D(anim : bool) : void =>+setMovObjetoArriba(mov : bool) : void =>+setMovObjetoAbajo(mov : bool) : void =>+setMovObjetolzq(mov : bool) : void =>+setMovObjetoDer(mov : bool) : void

Figura 82. Diagrama de la clase Álbum

ArchivoObjeto (estructura)



Figura 83. Diagrama de la estructura ArchivoObjeto

Atacantes



Figura 84. Diagrama de clase Atacante

AvesOpenSteerPlugIn

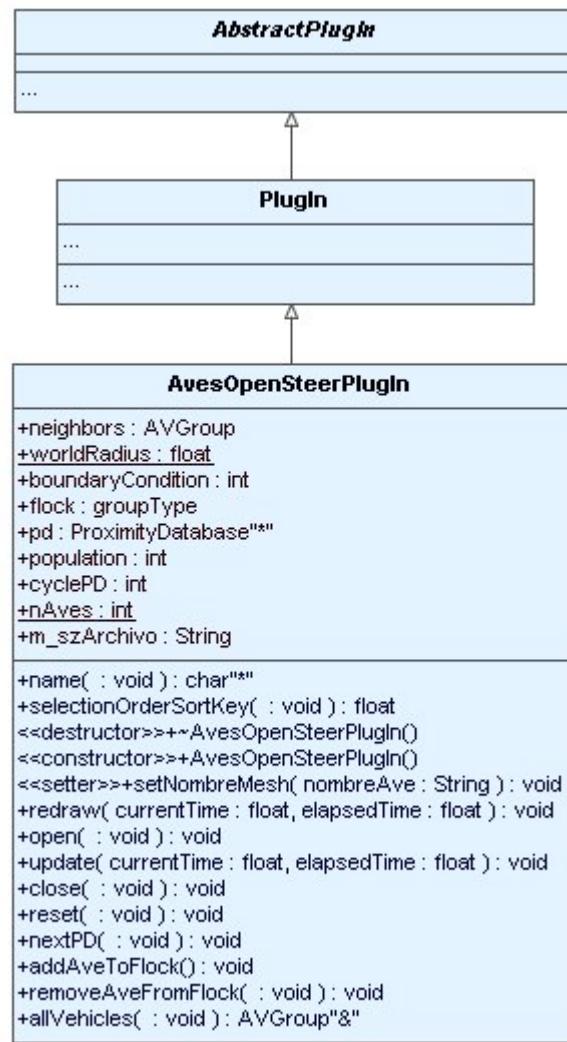


Figura 85. Diagrama de clase AvesOpenSteerPlugIn

CamaraLibre

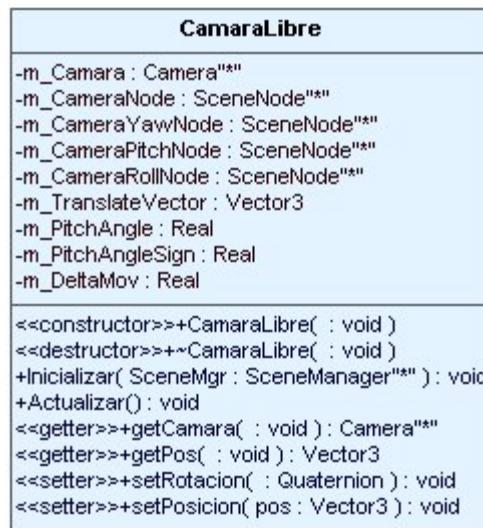
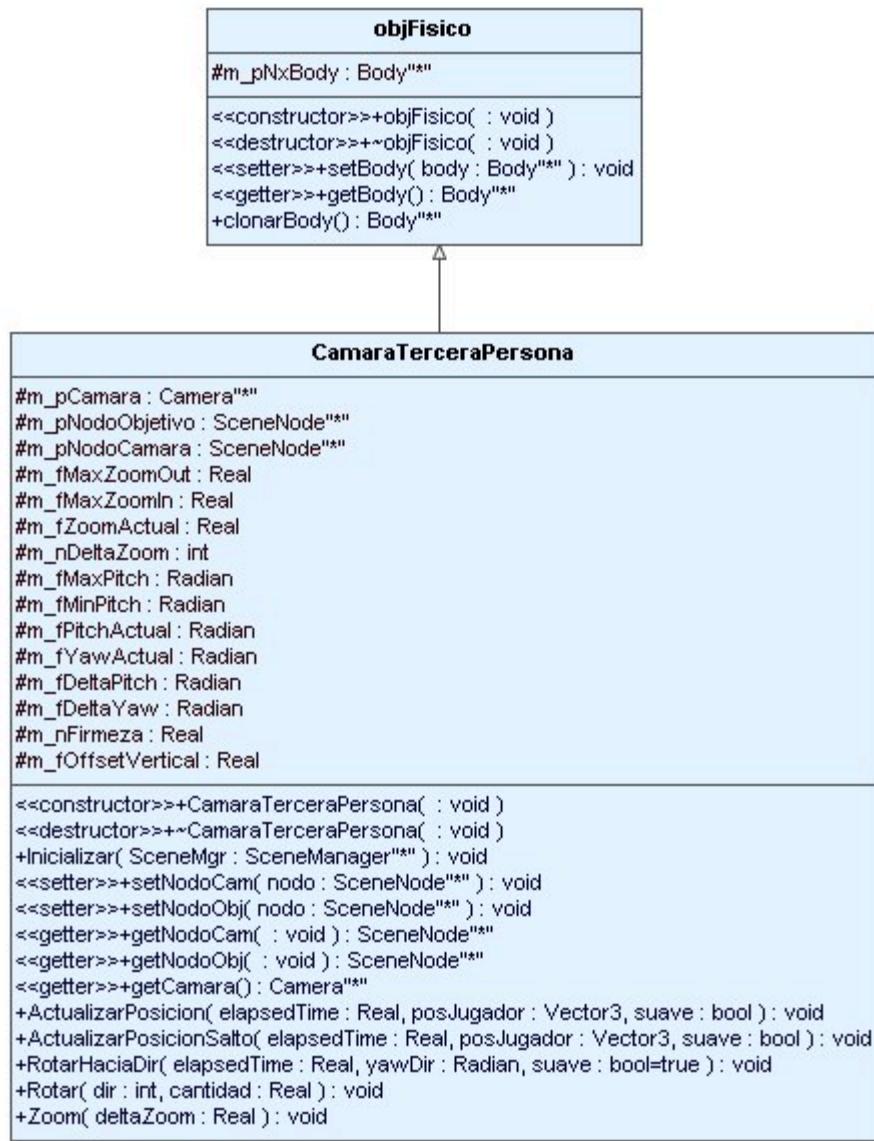


Figura 86. Diagrama de clase CamaraLibre

CamaraTerceraPersona



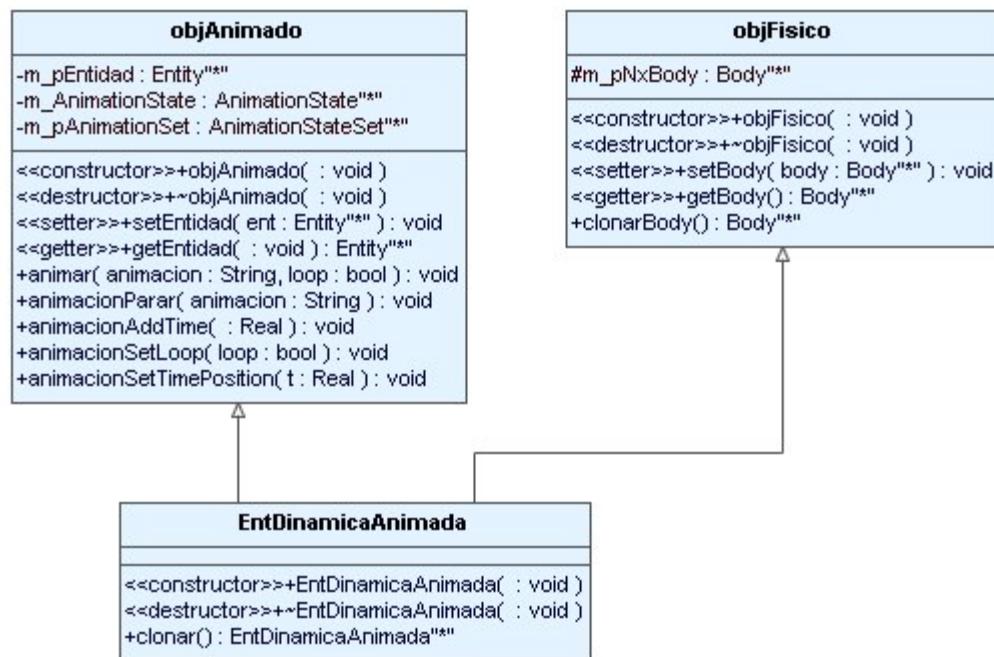
ConfiguracionJuego

ConfiguracionJuego
#m_szResolucion : String #m_szRenderer : String #m_bFullscreen : bool #m_nCalidadGraficos : int #m_fVisibilidadArboles : float"[3]" [0..*] #m_fVisibilidadPastos : float"[3]" [0..*] #m_fVisibilidadImpostors : float"[3]" [0..*] #m_fVolMusica : Real #m_fVolEfectos : Real #m_fVolAmbiente : Real +szResolucion : String +szRenderer : String +bFullscreen : bool +fVolMusica : Real +fVolEfectos : Real +fVolAmbiente : Real +m_Map : int"[TOTAL_ACCIONES][3]" [0..*] <<constructor>>+ConfiguracionJuego(: void) <<destructor>>+~ConfiguracionJuego(: void) <<getter>>+getResolucion(: void) : String <<getter>>+getResolucionEnVector(: void) : Vector2 <<getter>>+getRenderer(: void) : String <<getter>>+getFullscreen(: void) : bool <<getter>>+getVolMusica(: void) : Real <<getter>>+getVolEfectos(: void) : Real <<getter>>+getVolAmbiente(: void) : Real <<getter>>+getMapaAccion(accion : int) : int <<getter>>+getDispositivoAccion(accion : int) : int <<getter>>+getSubdispositivoAccion(accion : int) : int <<getter>>+getAccion(accion : int) : int*** <<getter>>+getCalidadGraficos(: void) : int <<getter>>+getVisibilidadPasto(calidad : int) : float <<getter>>+getVisibilidadArboles(calidad : int) : float <<getter>>+getVisibilidadImpostors(calidad : int) : float <<setter>>+setResolucion(ancho : int, alto : int, bpp : int) : void <<setter>>+setResolucion(resolucion : String) : void <<setter>>+setRenderer(renderer : String) : void <<setter>>+setFullscreen(fullscreen : bool) : void <<setter>>+setVolMusica(vol : Real) : void <<setter>>+setVolEfectos(vol : Real) : void <<setter>>+setVolAmbiente(vol : Real) : void <<setter>>+setMapaAccion(accion : int, mapa : int) : void <<setter>>+setDispositivoAccion(accion : int, dispositivo : int) : void <<setter>>+setSubdispositivoAccion(accion : int, subdispositivo : int) : void <<setter>>+setCalidadGraficos(calidad : int) : void <<setter>>+setVisibilidadPasto(v : float, calidad : int) : void <<setter>>+setVisibilidadArboles(v : float, calidad : int) : void <<setter>>+setVisibilidadImpostors(v : float, calidad : int) : void +copiarConfiguracion(cfg : ConfiguracionJuego***) : void

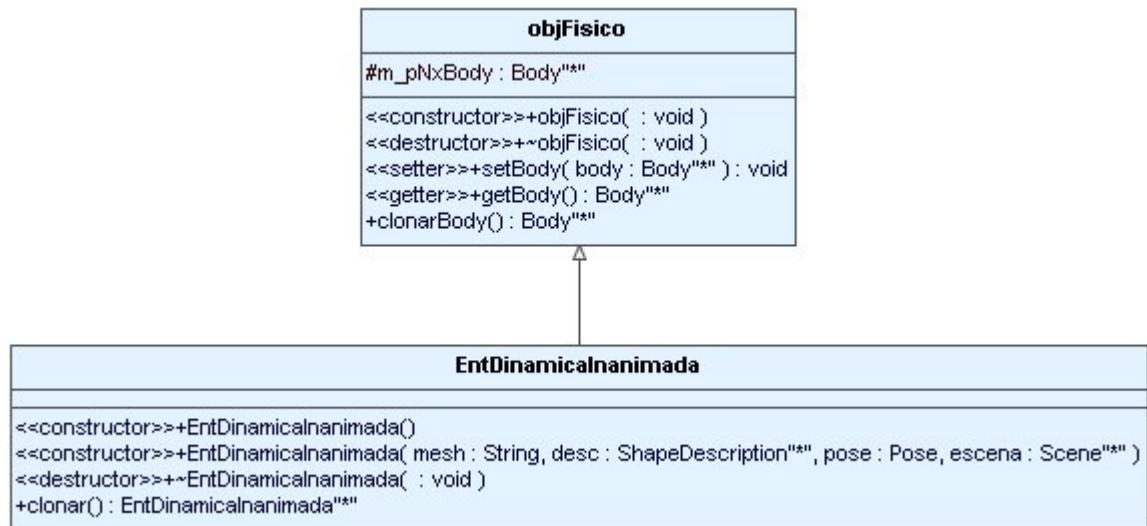
Demo

Demo
#m_Juego
<<constructor>>+Demo(: void)
<<destructor>>+~Demo(: void)
<<getter>>+getSingleton(: void) : Demo&"
<<getter>>+()
+actualizar(evt : FrameEvent&") : void
+inicializarAves() : void
+getMotorJuego() : MotorJuego**"

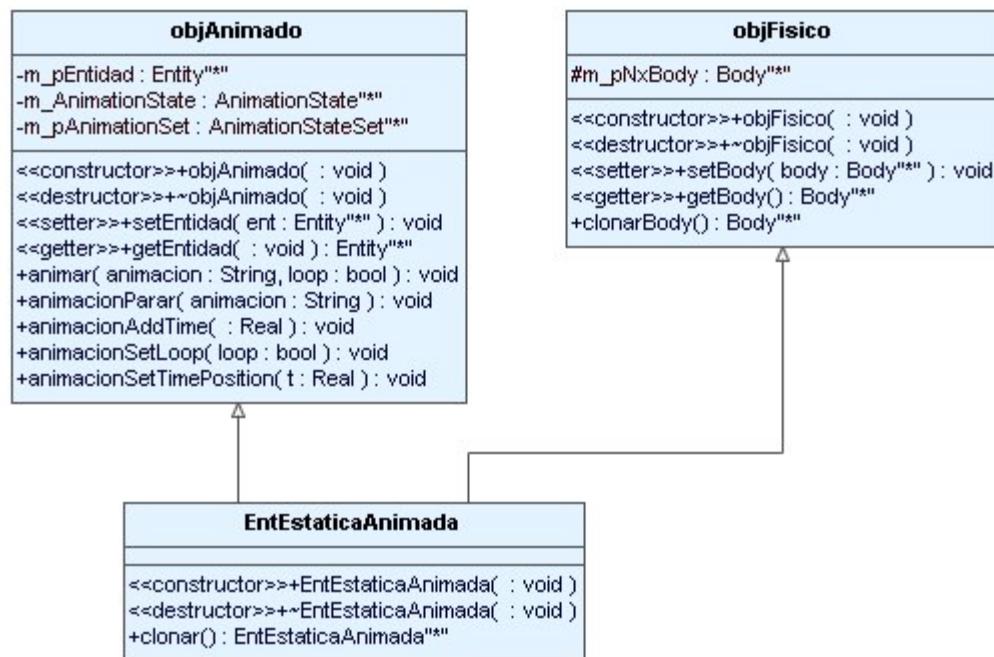
EntDinamicaAnimada



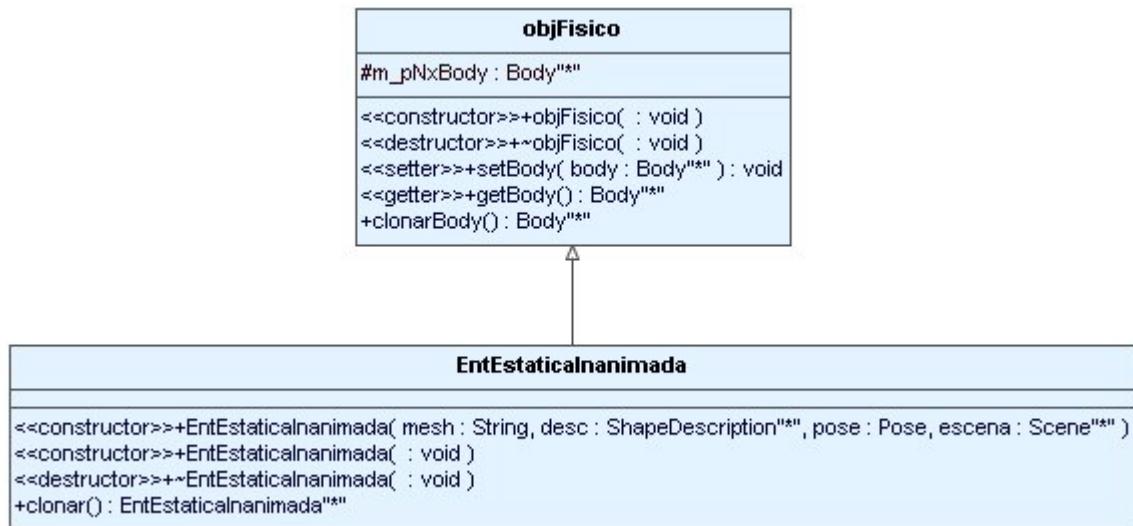
EntDinamicalnanimada



EntEstaticaAnimada



EntEstaticaInanimada



EntMesh

EntMesh
#m_pEntity : Entity** #m_pSceneNode : SceneNode**
<<constructor>>+EntMesh(: void) <<destructor>>+~EntMesh(: void) +cargarMesh(mesh : String, pos : Vector3&, orientacion : Quaternion&, scnMgr : SceneManager**) : bool <<setter>>+setEntidad(ent : Entity**) : void <<setter>>+setSceneNodo(nodo : SceneNode**) : void <<getter>>+getEntidad(: void) : Entity** <<getter>>+getSceneNode(: void) : SceneNode**

HerramientasFisicas

HerramientasFisicas	
+columpioAbajo : Body***	
+estadoJoint : int	
+revJoint : NxRevoluteJoint***	
+descripcionJoint : NxRevoluteJointDesc	
+EjeJoint : NxVec3	
+PosicionJoint : NxVec3	
+HabilitarLimites : bool	
+HabilitarColision : bool	
+AnguloAdelante : NxReal	
+AnguloAtras : NxReal	
+SuavidadGolpeAdelante : float	
+SuavidadGolpeAtras : float	
+DampingFuerte : Real	
+DampingDebil : Real	
+Fuerza_aplicada : int	
+m_Scene : Scene***	
+m_liana : Cloth***	
+m_tronco : Body***	
+bodyAbajo_liana : Body***	
+OrientacionLiana : Quaternion	
+parametros : ClothParams***	
+nLianas : int	
<<constructor>>+HerramientasFisicas(: Scene***)	
<<destructor>>+~HerramientasFisicas()	
+CrearColumpio(: Actor***, posicion_choque : Vector3, ori : Quaternion) : void	
+destruirColumpio() : void	
+ControlarColumpio() : void	
+PosicionarJoint(: NxVec3) : void	
+DefinirEjeJoint(: NxVec3) : void	
+DefinirAmortiguacion(: void) : void	
+CrearLiana(malla_liana : String, posicion_liana : Vector3, posicion_body : Vector3, Orientacion : Quaternion) : void	
+ConfigurarLiana(: void) : void	
+DestruirLiana() : void	
+ControlarLiana() : void	

itemLamina

itemLamina
+m_pEntidad : void** +m_nTipoEntidad : int +m_LaminaPadre : Lamina**

Jugador



```

<<setter>>+setAgua( sed : Real ) : void
<<setter>>+setComida( hambre : Real ) : void
<<setter>>+setSalud( salud : Real ) : void
<<getter>>+getAgua( : void ) : Real
<<getter>>+getComida( : void ) : Real
<<getter>>+getSalud( : void ) : Real
+ComerFruta() : void
+ComerSemilla() : void
+ComerHojas() : void
+TomarAgua() : void
+ActualizarHUD() : void
<<getter>>+getYawMovimiento( : void ) : Radian
#InicializarFisical escena : Scene"", pos : Vector3, orientacion : Quaternion ) : bool
#actualizarEstadoEnAire( : FrameEvent&"") : bool
#actualizarEstadoEnSuelo( : FrameEvent&"") : bool
#actualizarEstadoRascandose( : FrameEvent&"") : bool
#actualizarEstadoNadando( : FrameEvent&"") : bool
#actualizarEstadoCorrer( : FrameEvent&"") : bool
#actualizarEstadoSaltando( : FrameEvent&"") : bool
#actualizarEstadoCayendo( : FrameEvent&"") : bool
#actualizarEstadoTrepando( : FrameEvent&"") : bool
#actualizarEstadoPasamanos( : FrameEvent&"") : bool
#actualizarEstadoLiana( : FrameEvent&"") : bool
#actualizarEstadoColumpiandose( : FrameEvent&"") : bool
#actualizarEstadoAgarrado( : FrameEvent&"") : bool
#actualizarEstadoSubiendose( : FrameEvent&"") : bool
#actualizarEstadoBajandose( : FrameEvent&"") : bool
#actualizarEstadoNinguno( : FrameEvent&"") : bool

```

Lamina

Lamina
#m_pLamina : void*** #m_nTipoEntidad : int #m_szNombreLamina : String #m_vItems : vector<itemLamina*> #m_nLaminas : int <<constructor>>+Lamina(: void) <<destructor>>+-Lamina(: void) +mostrarLamina(: bool) : void <<setter>>+setLamina(entidad : void***, tipo : int) : void +agregarItem(: void***, : int) : itemLamina*** +borrarItem(: itemLamina***) : void <<getter>>+getEntidad(: void) : void*** <<setter>>+setTipoEntidad(t : int) : void <<getter>>+getTipoEntidad(: void) : int <<setter>>+setNombreLamina(n : String) : void <<getter>>+getNombreLamina(: void) : String

ManejadorGeometria

ManejadorGeometria
+ raySceneQuery + updateRay # m_pArbolesLoader # m_pPagedGeometryArboles # m_pPastoLoader # m_pPagedGeometryPasto # m_vMeshes # m_vEntities # m_itMeshes # m_itMeshesFin # m_itEntities # m_itEntitiesFin + ManejadorGeometria() + ~ManejadorGeometria() + actualizar() + crearManejadorDeArboles() + crearManejadorDePasto() + agregarCapaDePasto() + agregarArbolEntity() + buscarMesh() + agregarEntity() + agregarMesh() + getPagedGeometryArboles() + getPagedGeometryPasto() + getArbolesLoader() + getPastoLoader()

ManejadorObjetos

ManejadorObjetos
#m_vObjetosCargados : vector<archivoObjeto*> #m_it : iterator #m_itFin : iterator <<constructor>>+ManejadorObjetos(: void) <<destructor>>+-ManejadorObjetos(: void) +cargarObjeto(archivo : String, pObjeto : void***, tipo : int***, escena : Scene***) : bool +cargarArchivoXML(archivo : String, pArchivoXML : void***) : bool #crearObjetoXML(pArchivoObjeto : archivoObjeto***, ObjetoNuevo : void***, TipoNuevoObjeto : int***, escena : Scene***) : bool #buscarObjetoXML(archivo : String, pArchivoObjeto : archivoObjeto***) : bool #agregarObjetoXML(archivo : String, : void***) : archivoObjeto*** #destruirTodo() : void +unnamed1()

Menu

Menu
<pre> #m_GUIRenderer : OgreCEGUIRenderer""" #m_GUISystem : System""" #m_windowManager : WindowManager""" #m_imagesetManager : ImagesetManager""" #m_sheetMenuPpal : Window""" #m_sheetMenuEnJuego : Window""" #m_sheetHUD : Window""" #m_sheetMenuPerfil : Window""" #m_sheetMenuCargando : Window""" #m_fadeWnd : Window""" #m_pMensajeWnd : Window""" #m_configWnd : ScrollablePane""" #m_perfilWnd : ScrollablePane""" #m_szNombrePerfilesWnd : String #m_nPerfilSeleccionado : int #m_szNombreConfigWnd : String #m_nAccionEntrada : int #m_tempCfg : ConfiguracionJuego #m_sheetAlbum : Window""" #m_sheetAlbumLamina3d : Window""" #m_pVentanaObjeto3d : Window""" #m_nLaminas : int #m_pRenderTexture : RenderTexture""" #m_pCamaraTextura : Camera""" #m_pNodoCamaraTextura : SceneNode""" #m_pViewPortCamara : Viewport""" #m_pCEGUIRenderTexture : Texture""" #rttImageSet : Imageset""" #m_bFade : int #m_nSonidos : int = 6{readOnly} #m_sonidoRollOver : Sonido*[m_nSonidos]" [0..*] #m_szNombreMensajeWnd : String #m_pFncMenu : bool = *(const CEGUI::EventArgs&) <<constructor>>+Menu(: RenderWindow""" , : SceneManager""") <<destructor>>+~Menu(: void) <<getter>>+getCEGUISystem(: void) : System""" <<getter>>+getWindowManager(: void) : WindowManager""" +Actualizar(evt : FrameEvent&") : bool +SetHandlersDeEventos(: int) : void +CargarMenu(: int) : void +MostrarMouse(: bool) : void +MostrarMenu(: int , : bool) : void <<getter>>+getCEGUIRenderer(: void) : OgreCEGUIRenderer""" <<getter>>+getCEGUIM WindowManager(: void) : WindowManager""" +crearFade() : void <<getter>>+getFade(: void) : int +iniciarFadeIn(: void) : bool +iniciarFadeOut(: void) : bool +finFade(: void) : void +hacerFadeIn(: Real) : bool +hacerFadeOut(: Real) : bool +hacerFade(evt : FrameEvent&") : void +crearVentanaMensajes() : bool +esconderVentanaMensajes() : void +mostrarVentanaMensajes(: String) : void +MenuPpalConfigControlActualizar(: void) : bool </pre>

```

<<setter>>+setAccionEntrada( a : int ) : void
<<getter>>+getAccionEntrada( : void ) : int
<<getter>>+getMenuPpMWhd() : Window"""
<<getter>>+getMenuEnJuegoWhd() : Window"""
<<getter>>+getMenuHUDWhd() : Window"""
+crearVentanaPerfiles( szPerfiles : String""", nPerfiles : int ) : void
+destruirMenuJuego( : void ) : void
+clickEnBoton( boton : PushButton""") : void
#OISaCEGUI( buttonID : int ) : MouseButton
#destruirMenuPerfil( : void ) : void
#MenuPerfilCrear( : EventArgs&" ) : bool
#MenuPerfilSalir( : EventArgs&" ) : bool
#MenuPerfilAceptar( : EventArgs&" ) : bool
#MenuPerfilEliminar( : EventArgs&" ) : bool
#MenuPpalClickPerfil( : EventArgs&" ) : bool
#destruirVentanaPerfiles( : void ) : void
#destruirMenuPpal( : void ) : void
#MenuPpalSalir( : EventArgs&" ) : bool
#MenuPpalJugar( : EventArgs&" ) : bool
#MenuPpalAlbum( : EventArgs&" ) : bool
#MenuPpalVolverAPerfiles( : EventArgs&" ) : bool
#MenuPpalConfig( : EventArgs&" ) : bool
#MenuPpalConfigControl( evt : EventArgs&" ) : bool
#MenuPpalConfigCancelar( : EventArgs&" ) : bool
#MenuPpalConfigGuardarCambios( : EventArgs&" ) : bool
#MenuPpalConfigSelectCalidadGraficos( : EventArgs&" ) : bool
#crearVentanaConfigControl( : void ) : void
#destruirVentanaConfigControl( : void ) : void
#guardarConfiguracion( : void ) : bool
#copiarConfiguracion( : void ) : bool
#restaurarConfiguracion( : void ) : bool
#cambiarResolucion( : void ) : bool
#cambiarCalidadGraficos() : bool
#destruirMenuCargando( : void ) : void
#MenuPpalConfigScrollVolMusica( : EventArgs&" ) : bool
#MenuPpalConfigScrollVolEfectos( : EventArgs&" ) : bool
#MenuPpalConfigScrollVolAmbiente( : EventArgs&" ) : bool
#MenuPpalConfigVideoResolucion( : EventArgs&" ) : bool
#MenuPpalConfigVideoFullscreen( : EventArgs&" ) : bool
#MenuPpalAlbumLamina( : EventArgs&" ) : bool
#MenuPpalAlbumVolver( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaVolver( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaPlay( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaStop( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaSiguiente( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaMover_arriba( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaMover_abajo( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaMover_izq( evt : EventArgs&" ) : bool
#MenuPpalAlbumLaminaMover_der( evt : EventArgs&" ) : bool
#MenuPpalAlbumSoltolaminaMover_arriba( evt : EventArgs&" ) : bool
#MenuPpalAlbumSoltolaminaMover_abajo( evt : EventArgs&" ) : bool
#MenuPpalAlbumSoltolaminaMover_izq( evt : EventArgs&" ) : bool
#MenuPpalAlbumSoltolaminaMover_der( evt : EventArgs&" ) : bool
#MenuPpalCreditos( : EventArgs&" ) : bool
#destruirHUD( : void ) : void
#cargarMenuJuego() : void
#MenuJuegoVolver( : EventArgs&" ) : bool
#MenuJuegoSalir( : EventArgs&" ) : bool
#MenuJuegoConfirmarGuardar( : EventArgs&" ) : bool
#MenuJuegoGuardar( : EventArgs&" ) : bool
#MenuEnJuegoAlbum( e : EventArgs&" ) : bool

```

```
#MenuEnJuegoAlbumVolver( e : EventArgs&" ) : bool  
#MenuRollOver0( : EventArgs&" ) : bool  
#MenuRollOver1( : EventArgs&" ) : bool  
#MenuRollOver2( : EventArgs&" ) : bool  
#MenuRollOver3( : EventArgs&" ) : bool  
#MenuRollOver4( : EventArgs&" ) : bool  
#MenuRollOver5( : EventArgs&" ) : bool  
#MessageBoxClickAceptar( evt : EventArgs&" ) : bool  
#MessageBoxClickCancelar( : EventArgs&" ) : bool  
#VentanaMensajesClickAceptar( : EventArgs&" ) : bool
```

MotorEntrada

MotorEntrada
<pre>+SIN_ASIGNAR : int = -1{readOnly} +JOYSTICK_MAX_EJES : int = 4{readOnly} +SHIFT_JOYSTICK : int = 12{readOnly} +JOYSTICK_MIN_VALOR_EJE : int = (int)(1<<SHIFT_JOYSTICK){readOnly} -m_InputSystem : InputManager""" -m_Mouse : Mouse""" -m_Keyboard : Keyboard""" -m_Joysticks : vector<OIS::JoyStick*> -itJoystick : iterator -itJoystickEnd : iterator -m_fTiempoSinAccion : Real <<constructor>>+MotorEntrada(: void) <<destructor>>+-MotorEntrada(: void) <<getter>>+getSingleton(: void) : MotorEntrada&"" <<getter>>+getSingletonPtr(: void) : MotorEntrada""" <<setter>>+setWindowExtents(width : int, height : int) : void <<getter>>+getMouse(: void) : Mouse""" <<getter>>+getKeyboard(: void) : Keyboard""" <<getter>>+getJoystick(index : unsigned int) : JoyStick""" <<getter>>+getNumDeJoysticks(: void) : int +Inicializar(renderWindow : RenderWindow""", bufferedKeys : bool=false, bufferedMouse : bool=false, bufferedJoy : bool=false) : void +Capturar(: void) : void +InicioFrame(: void) : void +FinFrame(: void) : void +HundidoBotonAccion(e : int) : bool +HundidoBotonAccion(e : int) : bool +SoltóBotonAccion(e : int) : bool <<getter>>+getEstadoAccion(e : int) : int <<getter>>+getEstadoAnteriorAccion(e : int) : int <<getter>>+getMapa(e : int) : int <<getter>>+getDispositivo(e : int) : int <<getter>>+getSubDispositivo(e : int) : int <<getter>>+getMapaString(e : int) : String +MapearPorDefecto(mapa : int"Juego: TOTAL_ACCIONES"31, Joystick : bool=false) : void +mapaEntradasPorDefectoXML() : String +mapaEntradasXML() : String +mapearDeConfiguracion() : bool +ActualizarEstadoAcciones(: void) : bool +ActualizarEstadoAccionesMenu(: void) : bool +ObtenerEntrada(accion : int) : bool +reiniciarTiempoSinAccion(: void) : void <<getter>>+getTiempoSinAccion(: void) : Real +agregarTiempoSinAccion(t : Real) : void -keyPressed(e : KeyEvent&"") : bool -keyReleased(e : KeyEvent&"") : bool -mouseMoved(e : MouseEvent&"") : bool -mousePressed(e : MouseEvent&"", id : MouseButtonID) : bool -mouseReleased(e : MouseEvent&"", id : MouseButtonID) : bool -povMoved(e : JoyStickEvent&"", pov : int) : bool -axisMoved(e : JoyStickEvent&"", axis : int) : bool -sliderMoved(e : JoyStickEvent&"", sliderID : int) : bool -buttonPressed(e : JoyStickEvent&"", button : int) : bool -buttonReleased(e : JoyStickEvent&"", button : int) : bool <<getter>>-getMapaEjeJoystick(nEje : int, Dir : int) : int <<getter>>-getMapaDigitalJoystick(X : int, Y : int) : int <<getter>>-getMapaBotonJoystick(nEje : int) : int</pre>

MotorFisico

MotorFisico
<pre>+m_pColumpioEnUso : Actor*** +m_pPasamanosEnUso : Actor*** +m_pLianaEnUso : Actor*** +m_pTrepadorEnUso : Actor*** +m_pJointColumpio : NxRevoluteJoint*** +m_vPosicionContacto : Vector3 +m_pHerramientasFisicas : HerramientasFisicas*** #m_Mundo : World*** #m_terreno : Body*** <<constructor>>+MotorFisico(: void) <<destructor>>+~MotorFisico(: void) <<getter>>+getSingleton(: void) : MotorFisico&*** <<getter>>+getSingletonPtr(: void) : MotorFisico*** +Iniciar(root : Root***, sceneManager : SceneManager***, FPS : int) : bool +crearEscena(nombreScene : String, sceneManager : SceneManager***, parametros : String) : Scene*** +PosicionarAyudante(: void) : void +LanzarFruta(pos : Vector3, orientacion : Quaternion) : void +crearMateriales(: void) : void + inicializarMateriales(: void) : void +removerMateriales(: void) : void <<getter>>+getWorld() : World*** <<setter>>+setPausa(p : bool) : void +crearTriggers(pos : Vector3, dim : Vector3, ori : Quaternion, forma : String, escena : Scene***, tipoTrigger : int=NX_TRIGGER_ENABLE, grupottrigger : int=0) : void +crearParejaContacto(a : NxActor***, b : NxActor***, _start : bool, _touch : bool, _end : bool, : Scene***) : void +crearGrupoContacto(grupo1 : int, grupo2 : int, _start : bool, _touch : bool, _end : bool, : Scene***) : void +crearm_pTrepadorEnUso(pos : Vector3, altura : Real, radio : Real, : Scene***) : void</pre>

MotorGrafico

MotorGrafico
<pre>#m_Root : Root*** #m_SceneMgr : SceneManager*** #m_ViewPort : Viewport*** #m_Window : RenderWindow*** #m_Camara : CamaraTerceraPersona*** #m_CamaraLibre : CamaraLibre*** #m_Menu : Menu*** #m_dShowMovieTextureSystem : DirectShowMovieTexture*** #m_fTiempoFrame : float #m_bUsandoCaelum : bool #m_CaelumSystem : CaelumSystem*** #m_CaelumModel : SkyColourModel*** #m_nVelDiaCielo : int <<constructor>>+MotorGrafico(: void) <<destructor>>+~MotorGrafico(: void) <<getter>>+getSingleton(: void) : MotorGrafico& <<getter>>+getSingletonPtr(: void) : MotorGrafico*** +Inicializar(: char***, : ConfiguracionJuego***) : bool +SetSceneManager(: SceneType) : void +SetColorFondo(: ColourValue) : void +IniciarMenu() : bool <<getter>>+getRenderWindow(: void) : RenderWindow*** <<getter>>+getRoot(: void) : Root*** <<getter>>+getSceneManager(: void) : SceneManager*** <<getter>>+getCamara(: void) : CamaraTerceraPersona*** <<getter>>+getCamaraLibre(: void) : CamaraLibre*** <<getter>>+getMenu() : Menu*** <<getter>>+getTiempoFrame(: void) : float <<setter>>+setTiempoFrame(t : float) : void +crearRoot() : bool +inicializarRecursos() : bool +inicializarCieloYSol(: void) : bool +destruirCieloYSol(: void) : bool +inicializarCaelum(: void) : bool +pausarActualizacionCaelum(: bool) : void +actualizarCaelum() : void +usandoCaelum(: void) : bool <<setter>>+setVelDiaCielo(v : int) : void <<getter>>+getVelDiaCielo() : int <<setter>>+setCalidadGraficos(: int) : bool #InicializarVentana(: char***, : ConfiguracionJuego***) : bool #configurarSombras(calidad : int) : bool</pre>

MotorIA

MotorIA
-m_pAves : AvesOpenSteerPlugin**
<<constructor>>+MotorIA(: void)
<<destructor>>+~MotorIA(: void)
<<getter>>+getSingleton(: void) : MotorIA&"
<<getter>>+getSingletonPtr(: void) : MotorIA**"
+actualizar(evt : FrameEvent&") : void
+initializarAves() : void
+destruirAves() : void

MotorSonido

MotorSonido
<pre>+NUM_CANALES : int +result : FMOD_RESULT -m_FMODSystem : System*** -version : unsigned int -speakermode : FMOD_SPEAKERMODE -m_Sonidos : SoundMap -m_pGruposSonidos : ChannelGroup**[TOTAL_GS] [0..*] -m_cancionActual : Sonido*** <<constructor>>+MotorSonido() <<destructor>>-+MotorSonido() <<getter>>+getSingleton(: void) : MotorSonido& <<getter>>+getSingletonPtr(: void) : MotorSonido*** +Inicializar(rate : MixRate, numeroCanales : int) : bool <<getter>>+getFMODSystem(: void) : System*** +addSonido3D(nombre : string, archivo : string, min : Real, max : Real, loop : int, pos : Vector3) : Sonido*** +addCancion(nombre : string, archivo : string, loop : int) : Sonido*** +addSonidoAmbiente(nombre : string, archivo : string, loop : int) : Sonido*** +removerSonido(nombre : string) : void +playSonido(nombre : string) : void <<getter>>+getSonido(nombre : string) : Sonido*** +encodeSonido(infile : string, outfile : string) : void <<setter>>+setPosicionListener(pos : Vector3&, vel : Vector3&, fdir : Vector3&, updir : Vector3&) : void <<setter>>+setPosicionListener(cam : Camera***, vel : Vector3&) : void +actualizarSistema(pos : Vector3=Ogre::Vector3::ZERO, vel : Vector3=Ogre::Vector3::ZERO, up : Vector3=Ogre::Vector3::ZERO, front : Vector3=Ogre::Vector3::ZERO) : void <<getter>>+getCancionActual(: void) : Sonido*** <<setter>>+setCancionActual(s : Sonido***) : void <<setter>>+setVolumenMusica(v : Real) : void <<setter>>+setVolumenEfectos(v : Real) : void <<setter>>+setVolumenAmbiente(v : Real) : void <<setter>>+setVolumenGrupo(: Real, : int) : void <<getter>>+getVolumenGrupo(g : int) : Real <<getter>>+getGrupoSonidos(i : int) : ChannelGroup***</pre>

MotorVideo

MotorVideo
-m_DShowVideoTextura : DirectShowMovieTexture*** -m_szArchivoVideo : String -sheetVideo : Window***
<<constructor>>+MotorVideo() +inicializar(ancho : int, alto : int, noModificarDimesiones : bool, loop : bool, ceguiRenderer : OgreCEGUIRenderer***) : bool +cargarVideo(archivo : String, flipH : bool=false) : bool +DShowVideoTextura(: void) : DirectShowMovieTexture*** <<destructor>>+~MotorVideo(: void)

ContactReporter

ContactReporter
<<constructor>>+ContactReporter(: void) <<destructor>>+~ContactReporter(: void) +onStartTouch(a : Actor***, b : Actor***, : Vector3) : void +onTouch(a : Actor***, b : Actor***, : Vector3) : void +onEndTouch(a : Actor***, b : Actor***, : Vector3) : void +onContactNotify(pair : NxContactPair&, events : NxU32) : void

TriggerCallback

TriggerCallback
<<constructor>>+TriggerCallback(: void) <<destructor>>+~TriggerCallback(: void) +onTrigger(triggerShape : NxShape"&", otherShape : NxShape"&", status : NxTriggerFlag) : void

Nivel

Nivel
<pre># m_Archivo # m_Jugador # m_Terreno # m_vPosInicialJugador # m_vOrientacionInicialJugador # m_Escena # m_pManejadorObjetos # m_pManejadorGeometria # m_vEntDinInanimadas # m_vEntDinAnimadas # m_vEntEstInanimadas # m_vEntEstAnimadas # m_vEntMesh # m_vPersonajes # m_vAtacantes # m_vNxActores # m_vObjetosTexto # m_vLaminas # m_ParamsTrigger # m_nPorcentajeCarga</pre> <pre>+ Nivel() + ~Nivel() + InicializarJugador() + actualizar() + getJugador() + setJugador() + getEscena() + setPosicionInicialJugador() + setOrientacionInicialJugador() + getPositionInicialJugador() + getOrientationInicialJugador() + cargarNivel() + agregarObjeto() + crearEscena() + crearActor() + getManejadorGeometria() + InicializarManejadorGeometria() + cerrarManejadorGeometria() + setParametrosShape() + agregarObjetoTexto() + eliminarObjetoTexto() + destruirObjetosTexto() + cargarLamina() + destruirLamina() + cargarPersonaje()</pre>

```
+ getTerreno()
+ crearTerreno()
+ getPorcentajeCarga()
+ setPorcentajeCarga()
+ actualizarLaminas()
+ actualizarAtacantes()
+ getSingleton()
+ getSingletonPtr()
```

objAnimado

objAnimado
<pre>-m_pEntidad : Entity***\n-m_AnimationState : AnimationState***\n-m_pAnimationSet : AnimationStateSet***\n\n<<constructor>>+objAnimado(: void)\n<<destructor>>+~objAnimado(: void)\n<<setter>>+setEntidad(ent : Entity***) : void\n<<getter>>+getEntidad(: void) : Entity***\n+animar(animacion : String, loop : bool) : void\n+animacionParar(animacion : String) : void\n+animacionAddTime(: Real) : void\n+animacionSetLoop(loop : bool) : void\n+animacionSetTimePosition(t : Real) : void</pre>

objFisico

objFisico
#m_pNxBody : Body**
<<constructor>>+objFisico(: void)
<<destructor>>+-objFisico(: void)
<<setter>>+setBody(body : Body***) : void
<<getter>>+getBody() : Body***
+clonarBody() : Body***

Personaje

Personaje
#m_pCaracter : Character*** <<constructor>>+Personaje() <<constructor>>+Personaje(escena : Scene***, nombre : String, mesh : String, posicion : Vector3, radio : Real, alto : Real, skin : Real) <<destructor>>+~Personaje(: void) <<getter>>+getCharacter() : Character*** <<setter>>+setCharacter(ch : Character***) : void <<getter>>+getNodoFisico() : SceneNode***

PersonajeHitReport

PersonajeHitReport
<<constructor>>+PersonajeHitReport() <<destructor>>+-PersonajeHitReport() +onActor(: Character "", : Actor "", : Shape "", : ActorGroup "", : NxControllerShapeHit "&"): Response +onCharacterHit(: Character "", : Actor "", : Shape "", : ActorGroup "", : NxControllersHit "&"): Response

Sonido

Sonido
<pre>-m_nGrupo : int -m_Canal : Channel** -m_nEstado : EstadoSonido -m_szNombre : string -m_Ptr : Sound** -m_Timer : Timer <<constructor>>+Sonido() <<destructor>>+~Sonido() +CargarSonido(nombre : string, archivo : string, grupo : int) : bool +play() : void +play(vol : int) : void +play(vol : int, pos : Vector3"8", vel : Vector3"8"=Ogre::Vector3::ZERO) : void +parar() : void +pausar() : void <<getter>>+getFDMSound(: void) : Sound** +estaSonando() : bool <<setter>>+setPosicionVelocidad(pos : Vector3"8", vel : Vector3"8"=Ogre::Vector3::ZERO) : void <<getter>>+getPosicionVelocidad(pos : Vector3"8", vel : Vector3"8") : void <<setter>>+setVolumen(vol : float) : void <<getter>>+getVolumen() : float <<setter>>+setFrecuencia(freq : float) : void <<getter>>+getFrecuencia() : float <<setter>>+setLoop(onoff : bool) : void <<getter>>+getEstado() : EstadoSonido <<getter>>+getNombre() : string"8" <<getter>>+getGrupo(: void) : int <<setter>>+setGrupo(g : int) : void <<getter>>+getCanal(: void) : Channel** <<getter>>+getTiempoUltimoPlay() : Real</pre>

Terreno

Terreno
<pre>+m_fAlturaTerreno : Real #m_vDatos : vector<float> #m_vTamTerreno : Vector3 #m_fProfundidad : Real #m_nAncholImagen : unsigned int #m_nAltolImagen : unsigned int #m_szArchivolImagen : String #m_pActorTerreno : NxActor***" #m_pHeightField : NxHeightField***" #m_TerrainMgr : TerrainManager***" #m_TerrainInfo : TerrainInfo</pre> <pre><<constructor>>+Terreno(: void) <<destructor>>+-Terreno(: void) +inicializarTerreno(root : Root***, scnMgr : SceneManager***, cam : Camera***) : bool <<getter>>+getAltura(x : Real, y : Real) : Real <<getter>>+getAlturaDeHeightmap(x : unsigned int, y : unsigned int) : Real +cargarTerreno(escena : NxScene***, sceneMgr : SceneManager***, imagen : String, cfgTerreno : String, tamTerreno : Vector3, profundidad : Real) : bool #crearHeightField(escena : NxScene***) : NxHeightField***"</pre>

triggerCallback

TriggerCallback
<<constructor>>+TriggerCallback(: void) <<destructor>>+~TriggerCallback(: void) +onTrigger(triggerShape : NxShape"&", otherShape : NxShape"&", status : NxTriggerFlag) : void