MEMORIA DEL PROYECTO

ESTUDIO DE APLICACIÓN:

UNREAL DEVELOPMENT KIT

Autor: Fernando González Ramos
Director: Manuel Agustí Melchor
Titulación: Ingeniería Técnica en Informática de Sistemas

26 de septiembre de 2011
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

RESUMEN:

Esta memoria recorre todo el proceso de investigación y aprendizaje llevado a cabo sobre el motor de desarrollo de videojuegos Unreal Development Kit, y su aplicación práctica para la creación del edificio de la Escuela de Informática de la UPV, el cual se podrá visitar virtualmente.

PALABRAS CLAVE:

UDK, Unreal Development Kit, Edificio, Visita, Virtual, 3D, Arquitectura, Videojuegos, Interactivo, Motor Unreal

TABLA DE CONTENIDOS

- Sobre la licencia de uso

- Estructura de carpetas de la instalación

- Comenzando con UnrealEditor – Fundamentos Básicos

- ¿Qué es un mapa?

- Los modos de Vista

- Modelando con *brushes*

- *Content Browser* (Navegador de contenidos)

- CREANDO UN MAPA

Aquí se describe el procedimiento y elementos básicos para crear un mapa sencillo en UDK.

- Construir *brushes*

- Poniendo Materiales

- Añadiendo modelos estáticos

- Iluminación

- Volúmenes especiales: *LightMassImportanceVolume* y *PostProcessVolume*

- IMPORTAR MODELOS AL EDITOR UDK

Como unir aquellos modelos o arte creada por herramientas 3D con el ámbito de UDK.

- Importar modelados 3D a Unreal

- Importar modelos esqueléticos

- EDITOR DE TERRENO

Introducción breve a la herramienta para crear paisajes.

- UNREAL KISMET

Introducción a la herramienta de creación de scripts de interfaz gráfica.

- UNREAL MATINÉE

Introducción a la herramienta de Animación integrada en UDK.

- MENÚ Y HUD

Como se crean las interfaces de usuario para nuestro juego.

- Scaleform GFX

- Ejemplo de menú sencillo

- Poniendo nuestro menú en nuestro mapa

- UNREAL SCRIPT

Introducción en todo lo referente a utilizar un entorno de desarrollo de scripts para nuestros proyectos UDK, mediante el uso de esta herramienta.

- Consideraciones básicas – Juego y *Mod*

- La estructura de clases

- Flujo del juego en *UnrealScript*

- Configurar el entorno de desarrollo

- Algunos ejemplos

- Modificar archivos de configuración

- Unreal Frontend

- ALGUNOS EJEMPLOS DE MAPAS

Este apartado exhibe varios trabajos realizados por mí con el kit UDK, con el fin de mostrar de forma práctica las posibilidades que ofrecen algunas de las funcionalidades de tal kit.

- Ejemplo de tutorial: *Simple level*

- Ciclo día-noche

- *Mouse interface*

- Edificio

- UN EDIFICIO REAL: ESCUELA DE INFORMÁTICA DE LA UPV

Como objetivo del proyecto, se muestra un edificio real realizado íntegramente en UDK y su proceso de desarrollo.

-Desarrollo

-Trabajos futuros

- CONCLUSIONES

Una reflexión sobre lo descubierto durante el estudio de UDK.

- BIBLIOGRAFÍA Y REFERENCIAS

INTRODUCCIÓN

A mi entender, la evolución de los videojuegos, rápida como cualquier otra tecnología, se ha intensificado en la última década. No tanto en lo que se refiere a la evolución de la tecnología en sí, sino como negocio y referencia del ocio electrónico, hasta el punto de haberse convertido en un pilar del ocio digital, equiparable, y en muchos casos superior a la industria cinematográfica y discográfica.

Y el motivo de esto, en mi opinión, es que los videojuegos se han convertido en la experiencia multimedia definitiva. Actualmente un videojuego puede abarcar todos los ámbitos imaginables. Desde escenas cinemáticas codificadas como vídeo, compatibilidad total con toda tecnología de vídeo y audio, existencia en todo tipo de plataformas y entornos, posibilidad de integrar al jugador en comunidades masivas virtuales a través de Internet, hasta la opción de usar cualquier dispositivo de entrada, ya sea por pantalla táctil, el clásico mando, o incluso mediante la captura del movimiento por medio de la visión artificial (por no mencionar el avance en el renderizado 3D). Así pues, apoyada en la enorme evolución tecnológica multimedia, el videojuego ha desplegado un potencial nunca visto hasta hace pocos años.

Como es de esperar, toda industria que soporte un negocio masivo, tiene que estar en continua evolución, investigación, y tiene que invertir cantidades enormes de recursos con el fin de mejorar lo más rápido posible, para satisfacer así las exigencias del público. Esto ha propiciado que en la actualidad, la creación de un videojuego está casi totalmente modulada; se han creado herramientas para desarrollar cada uno de los aspectos del videojuego, facilitando enormemente el trabajo, ahorrando tiempo y permitiendo crear videojuegos de calidad sin necesidad de partir de cero, sino basándonos en un *engine* versátil y bien optimizado, con años de depuración y actualizado.

Por otra parte, el acercamiento de estas tecnologías a los desarrolladores no profesionales es fundamental, para poder promover la ampliación de las franquicias de videojuegos y la creación de nuevas compañías, que estimularán el mercado. Es por esto que algunos estudios se han preocupado en los últimos años de que sus motores gráficos y sus herramientas de desarrollo no estén estrictamente disponibles sólo para aquellas

grandes compañías que puedan permitirse la compra de la licencia de uso y derechos de distribución. Y ciertamente, está dando resultado: existe actualmente un auge de desarrolladores independientes que se están sirviendo de estas herramientas para hacer sus productos. Pero no sólo eso, gracias a estas facilidades tecnológicas, **la industria del videojuego no es la única que se está beneficiando** de estas facilidades. Quiero resaltar que la principal seña de identidad de los videojuegos como disciplina multimedia es la **interactividad y la ejecución en tiempo real**, (y ésta es la idea en la que se basa este proyecto). Por eso, es comprensible que muchas compañías que desarrollan software para aplicaciones interactivas, pero no videojuegos, se hayan aventurado a hacer uso de *engines*, ideados en un principio para hacer juegos. Por ejemplo, existen empresas que crean **simuladores** de conducción o aeronáutica, visitas arquitectónicas en tiempo real, o representaciones del cuerpo humano y sus órganos, para realizar estudios médicos (más adelante se detallarán algunos ejemplos concretos).

Todo esto es gracias a la ejecución en tiempo real, que debido a la actual potencia de procesado, permite fácilmente hacer escenarios de simulación virtuales, facilitando el trabajo de los profesionales y obteniendo resultados fiables. Hay que tener en mente el gran realismo que los videojuegos han alcanzado en la actualidad. Y lo bueno de usar motores de videojuegos, es que la empresa no necesita hacer una gran inversión para la creación de un software especializado.

El motor Unreal Engine 3 (creado por la compañía **Epic)**, actualmente, y desde finales del 2009, está a completa disposición de cualquiera que quiera hacerse con él de forma gratuita. Para ello se ha desarrollado el Unreal Development Kit, un Kit de desarrollo que compila todo tipo de aplicaciones para crear un producto basado en motor Unreal Engine 3. Este motor es uno de los más populares y prestigiosos en la actualidad, y no son pocos los títulos que se deben su existencia.

Es un hecho que toda empresa arquitectónica o de diseño urbanístico o doméstico debe contar con algún programa que permita a sus clientes visualizar sus proyectos, previamente a que se hayan construido en la realidad. Por otro lado, existen programas para simular visitas a edificios emblemáticos, como museos, con el fin de aproximar la experiencia de la visita a aquellos que no pueden ir en persona. Así pues, el motor Unreal 3 parece ser la herramienta idónea para este tipo de proyectos.

Pero centrémonos en lo que va a ser esta memoria; un estudio sobre un motor concreto: el **Unreal Engine 3**, y un objeto de investigación: la realización de una visita simulada a un edificio virtual (La escuela de Informática de la UPV). En él se van evaluar las posibilidades que el motor Unreal tiene y las herramientas que UDK nos ofrece para explotarlas. No va a ser un manual en profundidad ni una guía, más bien va a ser una referencia para estar al día en las posibilidades que UDK nos aporta, y un empujoncito para que el usuario nuevo pueda ahorrarse algo de tiempo a la hora de familiarizarse con el vasto entorno de UDK. Esta memoria contará además con algunos ejemplos ilustrados de trabajos que se han llevado a cabo y que serán de relevancia para el propósito de nuestro proyecto.

Escena de la demo técnica *Samaritan* de EPIC [1]

MOTIVACIONES DEL PROYECTO

La ambición de este proyecto surge de la idea de contar con un **programa interactivo de visita a un edificio de la UPV**. El motor Unreal es capaz no sólo de implementar dicha tarea a nivel gráfico, si no de crear un software que cuente con una interfaz propia, en función de los requisitos del usuario, y que sea de fácil instalación. Y para esto nos vamos a valer del relativamente nuevo Kit de desarrollo UDK, del cual se realizará una profunda investigación.

Está claro que Unreal se utiliza para crear juegos. Pero seamos prácticos. Tenemos en nuestra mano un sofisticado motor 3D que ejecuta escenarios en tiempo real. Y no sólo eso, puede recrear simulaciones, situaciones virtuales. Todo eso sin la necesidad de contar con un equipo de programadores que cree un software dedicado y costoso.

A continuación ilustraré algunos ejemplos de empresas ajenas al mercado de los videojuegos, que han decidido invertir en el Motor Unreal para llevar a cabo proyectos propios.

Visualización arquitectónica en tiempo real.

Una forma de visualizar un proyecto arquitectónico, algo que entra directamente a los ojos de aquel al que queremos presentar nuestro proyecto, ya sea una urbanización, un centro comercial, un palacio de congresos, o una facultad universitaria.

La empresa Tripod3D ha usado Unreal para estos fines. Esta empresa ilustra un buen ejemplo de lo que representa este proyecto. Fijémonos en las figuras 1 y 2. Obviamente, tales proyectos arquitectónicos son de alto nivel, desarrollados por un lucrado equipo de profesionales.

Figura 1

Los resultados saltan a la vista. El producto consiste en un software que se instala fácilmente, y ejecuta el Motor Unreal bajo un juego que el usuario puede manejar a su antojo. El usuario es un visitante dentro de ese entorno virtual, camina por las habitaciones, las calles, puede incluso modificar el color de las paredes, todo en tiempo real.

Figura 2

Al final, el usuario tiene una experiencia que le aproxima a la idea del proyecto lo mas fielmente posible, o por lo menos, bastante mejor que el observar unos simples planos esquemáticos sobre una pantalla.

Con el fin de complementar con más información acerca de las bondades del motor Unreal 3, se puede añadir que otras compañías no lo usan solamente con fines arquitectónicos o lúdicos. Un simulador de conducción en un entorno de condiciones realistas. Ése es el proyecto llevado a cabo por el Departamento de Transporte de Michigan, y que tiene por nombre *IntelliDrive*. *IntelliDrive* tiene como objetivo la simulación de las condiciones de seguridad sobre el transporte por carretera con vehículos. Podemos ver la simulación *IntelliDrive* en funcionamiento en la figura 3.
El equipo de desarrolladores ha soportado el aspecto gráfico del proyecto íntegramente en el motor de Unreal.

Figura 3

El proyecto objetivo de esta memoria, consiste en plantearse un modelo de **visita virtual**, que se podría aplicar a cualquier edificio de la UPV, y que tendría como objeto la posible ayuda que puede ofrecer al alumno o usuario para poder familiarizarse con dicho edificio, sin la necesidad de estar presente físicamente en él.

CONSIDERACIONES SOBRE EL DESARROLLO DE VIDEOJUEGOS

Extraído del libro *Mastering Unreal Technology*, Volumen 1, Capítulo 2

El proceso de creación de un videojuego es un proceso enorme, largo, muchas veces no se puede decir dónde empieza y tampoco es fácil decidir cuando acaba.
Hace falta un gran equipo, todos con distintas habilidades. Y sobre todo estar dispuesto a los cambios; la mayoría de las veces el rumbo del desarrollo va cambiando conforme se van viendo los resultados.

La Iteración

La iteración define el proceso de realimentación en el que se repiten las distintas etapas por las que pasa el producto, de forma cíclica. Así, el producto inicialmente no empieza siendo lo mejor posible, más bien es una versión llana, simple, un prototipo. Mediante la repetición del ciclo este producto solo puede ir haciéndose mejor, se va puliendo, y en ello juega un papel fundamental cada parte del equipo por el que pasa el juego; es analizado, criticado, y por tanto susceptible de ser mejorado.

Lo primero que hay que hacer es comenzar con una idea. Hay que responder a: ¿En qué va a consistir la jugabilidad? ¿Cuál es el concepto del juego? ¿Va a ser un genero en concreto? ¿Cuál es la historia o trasfondo del juego? ¿Cuál es la estética, el estilo y diseño de entornos y personajes?

El calendario de producción

Todo videojuego necesita un calendario de producción. Ayuda a crear un desarrollo disciplinado, y a motivar para llevar adelante el proyecto. Por supuesto, si una productora o distribuidora nos está pagando por el juego, es vital ajustarnos a su plazo de lanzamiento.
El tiempo no siempre juega a nuestro favor, y muchas veces simplemente nos quedaremos sin tiempo suficiente para llevar a cabo una tarea. La clave es hacer todo lo que se pueda hacer, sin prestar excesiva atención en los detalles que puedan acaparar un tiempo valioso para aspectos más generales.
Trabajar con el *Unreal Engine* simplifica enormemente el proceso, en el sentido de que el trabajo que hay que hacer consiste sobre todo en la creación de los objetos, tanto artísticos como de *scripting,* ya que el motor ya "está hecho" (es decir, no hay que

preocuparse por la tecnología de los gráficos, renderizado, iluminación, animación, etc. Simplemente esa tecnología ya está ahí para ser usada).
Por supuesto, es una opción el modificar aspectos del motor Unreal de forma crítica, mediante *UnrealScripts,* de hecho, es lo que suelen hacer las compañías que desarrollan sus propios videojuegos basados en este motor.

Testeo y *Feedback*

Testear es la parte del ciclo de desarrollo más crítica. El equipo de desarrollo debe jugar el juego, reuniéndose después y comentando las impresiones que cada uno ha tenido y lo que ha encontrado. Es una parte del proceso iterativo, por tanto, en sus estados iniciales, los jugadores encontrarán un juego en bruto, sin pulir, que cuenta solo con algunos aspectos jugables. A este tipo de testeo se le llama *alpha testing.* En este estado, el público nunca debería ver el juego, ya que es aun susceptible de profundos cambios y mejoras, e interpretar una falsa idea del juego.
En algún punto el juego está próximo a su finalización. En este punto hay que considerar el *beta testing*. En este testeo la idea es utilizar a gente ajena al proyecto para que lo pruebe. Puede ser gente contratada para tal labor, o bien elegir un grupo cerrado de jugadores del público, para que lo prueben. Esto es el *closed beta testing*, y se hace así cuando se quiere limitar el rango de usuarios, en función de cuanta gente queremos que lo pruebe, o qué de especificaciones mínimas de equipo queremos que dispongan.
Por otro lado, el *open beta testing* se publica una versión del juego al público general, permitiendo a cualquiera descargarlo y probarlo. Esto permite al público dar sus impresiones sobre el producto casi acabado.

Finalizando la producción

Una vez el ciclo de producción acaba, y el juego es publicado, si no se ha llegado a un acuerdo de lanzamiento con una distribuidora, aún estamos a tiempo de crear extras y añadir aspectos del juego que deseábamos ver pero que no se pudieron incluir por incompatibilidad del calendario.
En todo caso, aún después de la publicación del juego, hay una labor fundamental que todo desarrollador debe atender con dedicación: el soporte y mantenimiento, mediante la adición de *fixes*, con el fin de solucionar *bugs*. Por muy optimizado y testeado que pueda estar un juego, rara vez o nunca está exento de necesitar un parche para solucionar *bugs* aparecidos tras la publicación. Además es interesante, durante los años

posteriores de la vida del juego, si es que este cuenta con una importante comunidad de jugadores, que sean añadidos parches de optimización gráfica, adecuándose a los avances en tecnología de videojuegos.

MOTORES GRÁFICOS

Los programas que durante su ejecución muestran gráficos en tiempo real, requieren de una serie de rutinas de programación que establecen cómo va a ser su diseño, su representación. Al conjunto de estas rutinas de programación se les llama motor gráfico. En videojuegos, el motor del videojuego no sólo se centra en los aspectos gráficos, si no en otros muchos aspectos que definirán la jugabilidad, como la IA, la interfaz de entrada o las propias reglas del juego.
El motor Unreal 3, que es el motivo de este estudio, es uno de los más populares y con más trayectoria en el mundo de los videojuegos. Es un motor muy versátil y recomendado para juegos con perspectiva en tercera persona, aunque puede ser modificado para otros propósitos. Otros motores gráficos de renombre son el IDTech 5, heredero del Quake engine, que creó clásicos como la saga Quake; el RAGE, potente motor con el que se implementó el GTA IV, o el Euphoria, usado para dotar a los juegos de físicas hiperealistas.

El motor gráfico puede estar, además, soportado por las librerías gráficas propias del sistema operativo para la ejecución de gráficos en tiempo real. Estas librerías proveen un lenguaje de programación con el que el motor puede comunicarse con el hardware acelerador de gráficos, describiéndole el uso de primitivas, como vértices, polígonos, todo lo relativo a la renderización…
Existen 2 librerías gráficas fundamentales: Direct3D, que se usa en sistemas como Windows y en la consola Xbox 360, y OpenGL, que es de ámbito de código abierto y funciona en gran variedad de plataformas. La tabla 1 resume el uso de estas dos librerias:

Principales librerías gráficas: [2]

	Direct3D	**OpenGL**
Plataforma	Microsoft Windows	Multiplataforma
Plataforma Móvil	Direct3D Mobile	OpenGL para sistemas embebidos
Licencia	Propietario	Open source

Tabla 1

El desarrollo de un motor gráfico es además muy susceptible de otros elementos como el del hardware acelerador 3D, lo que vienen a ser las tarjetas gráficas. Concebir un motor que abusa de efectos como sombreados extremadamente detallados, texturas con demasiada resolución o muchos reflejos, repercute en que la tarjeta gráfica no tiene suficiente potencia para representar tales gráficos, o directamente no funciona. También puede pasar que las librerias gráficas usadas en el desarrollo del motor sean exclusivas para una versión determinada de tarjeta gráfica, por ejemplo, existen algunos videojuegos que sólo funcionan en equipos con gráficas DirectX 10, ya que se han implementado exclusivamente con este tipo de librerías.
Es por estos motivos que conviene siempre ajustarse a la configuración del equipo del usuario objetivo. Si nuestro producto va a ser para un público general, lo ideal es que adaptemos las configuraciones según el promedio, y que nos basemos en la potencia estándar de las gráficas de gama media – alta de ese momento. Muy importante tener en cuenta que la evolución tecnológica resulta en gráficas que mejoran mucho año tras año, por tanto, en una etapa de desarrollo larga, de varios años, va a ser preciso probablemente actualizar el motor gráfico para que no acabemos con un producto gráficamente desfasado.

Aparte del hardware gráfico, hay otros elementos que influyen en la ejecución del videojuego, fundamentalmente son el procesador y la memoria principal. La llegada de los procesadores de doble núcleo revolucionó en parte el mundo del videojuego ya que esto permitió evolucionar y hacer realistas muchos aspectos, gracias al *multithreading*, como la IA, o el cálculo de físicas en tiempo real; éstos son factores importantes en un videojuego y son trabajo del procesador, no de la gráfica. Más adelante se detallarán los requerimientos mínimos de hardware y software para ejecutar aplicaciones con motor Unreal 3.

EL MOTOR UNREAL 3

En esencia, el Motor Unreal organiza todos los "bienes" (modelados, personajes, trabajo artístico, sonidos, música) dentro de un entorno interactivo.

Componentes del motor Unreal

Extraído del libro *Mastering Unreal Technology*, Volumen 1, Capítulo 1

Todos funcionando de manera independiente, pero coordinado por un núcleo central. El motor Unreal está altamente modularizado y cada componente individual puede ser reemplazado sin realizar cambios drásticos en el núcleo. Esta sección resume los componentes del Motor Unreal:

Motor gráfico:

Controla todo aquello que se ve en el juego. Procesa enormes listas de parámetros en continuo cambio, y las traduce a todo lo relativo al aspecto visual. De la misma manera gestiona en qué forma se puede optimizar el uso de los recursos, y cuándo algunos objetos deben aparecer en pantalla o no según estén ocultos. Normalmente los desarrolladores tienen que programar la optimización de forma manual con el fin de aprovechar mejor los ciclos de procesado. El motor Unreal lo maneja por su cuenta.
Concretamente, en el motor Unreal se utiliza la característica llamada *level streaming* que permite que ciertas partes del nivel se carguen o descarguen en función de si es necesario que sean renderizadas o no, lo cual alivia una importante carga de uso de recursos.Por supuesto, también se ocupa de los *shaders* y de la iluminación.

Motor de sonido:

El motor Unreal maneja de forma muy sencilla todos los efectos de sonido, diálogos y música que vayan a formar parte del proyecto. Se crean mediante *sound cues* (señales de sonido), con los que configuramos y modificamos sonidos en crudo para dotarlos de una profundidad necesaria para que parezcan integrados dentro del juego (el entorno, la posición, etc., influyen en cómo se ha de reproducir el sonido)

Motor de físicas:

Implementa cómo se deben comportar los objetos en el mundo del juego y cómo interaccionan entre ellos. Es fundamental para dotar de realismo el juego. En Unreal 3, ésta labor es controlada por los controladores NVIDIA Physx, que es un motor capaz de simular cantidad de interacciones físicas, desde objetos rígidos, pasando por fluidos, *ragdolls* (muñecos de trapo) e incluso tejidos.

Controlador de entrada:

Todo lo relativo a la interacción del usuario con el juego. La respuesta, tanto si es desde mando como desde teclado ha de ser lo mas inmediata posible.

Infraestructura de Red:

Computa toda la comunicación necesaria para cada jugador con el servidor que da soporte al juego (a la partida, más bien). De modo que los ordenadores de los jugadores actúan como clientes, conectados a un servidor que ejecuta el juego en modo servidor. También se puede jugar desde el ordenador servidor. Si el ordenador se utiliza exclusivamente para servidor, se le denomina servidor dedicado. En el Motor Unreal, que principalmente se basa en juegos de acción en tiempo real, la baja latencia entre cliente y servidor es fundamental.

El Intérprete UnrealScript:

Es uno de los aspectos más revolucionarios del Motor Unreal. Es un lenguaje de scripting que permite a programadores ajustar virtualmente cualquier cosa que esté haciendo el motor, sin tocar el motor real. Es la causa de que se creen numerosos *mods* para los juegos basados en motor Unreal. Básicamente transforma *scripts* creados por alguien en código que el motor puede procesar. Se detallarán aspectos de UnrealScript más adelante.

Se puede resumir la interacción de los componentes en el Motor Unreal con la gráfica de la figura 4.

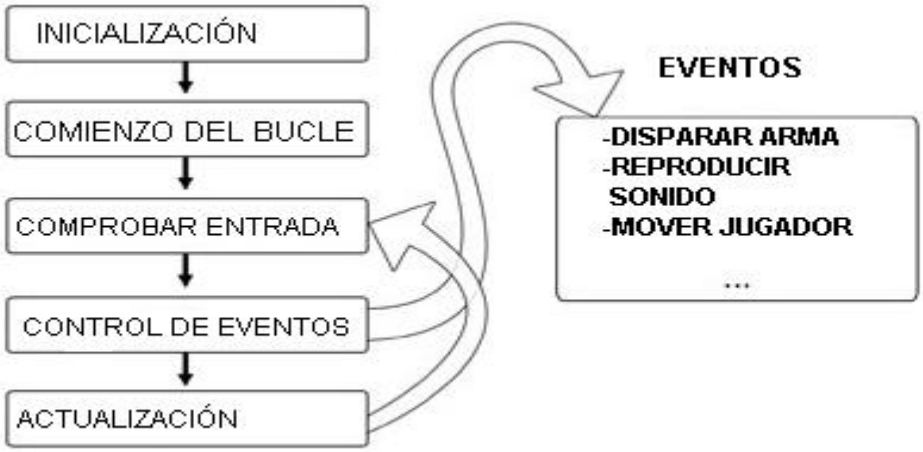


Figura 4

Los eventos se manejan en función de si se ha llegado a un momento en el que un componente requiere de su uso. En tal caso, se cargan los recursos necesarios. Todo esto sucede dentro de una ejecución en bucle que conforma la ejecución del juego.

UNREAL DEVELOPMENT KIT (UKD)

En octubre de 2009, Epic Games, famoso estudio de desarrollo de videojuegos, decide hacer públicas sus herramientas de desarrollo así como su motor de juegos. El renombre de dicho motor, logrado por numerosos éxitos en el mercado de videojuegos, hace que este Kit de desarrollo tenga una enorme aceptación y uso, por parte no sólo de desarrolladores independientes y profesionales, si no de un público general.

Así, el UDK se distingue por ser el primer motor de videojuegos de vanguardia y tecnología puntera que se pone a disposición del público.

Y un aspecto más importante, existe un gran soporte por parte de Epic, con investigaciones y actualizaciones continuas, así como publicaciones de guías para la iniciación en el uso del UDK, y foros en los que se tratan aspectos y se ofrece ayuda a los usuarios. Esto es gracias a la masiva comunidad de seguidores del Motor Unreal.

Ficha técnica

Las especificaciones técnicas se resumen en la tabla 2, no obstante hay que resaltar que las características del Unreal son muy numerosas, y acapararlas escapa a la idea de este proyecto. Si se quieren ver con más detalle, se puede consultar en el enlace. **[3]**

*Nota: El Motor Unreal está un continuo desarrollo y puede cambiar cada poco tiempo. Estas son las características a Marzo de 2011:

Autor	Epic Games Inc.
Sistemas operativos soportados	Windows, iOS
API Gráficos	OpenGL, DirectX
Lenguaje de programación del Motor	C/C++
Estado de la versión	Beta
Existe Documentación	Si
Algunas Características	Diseño orientado a objetos Interfaz sencilla Lenguaje de scripting de alto nivel (parecido a C++) Script visual (Kismet) Editor 3D Editor de terreno Editor de materiales Animaciones (Matinee) Navegador de contenidos (content browser) Editor de plantas (SpeedTree) Editor de Interfaces (Basado en ScaleForm) Editor de Sonidos Editor de post-procesado Físicas, detector de colisiones, físicas de vehículos Objetos fracturables Simulador de multitudes Iluminación: por vértice, por píxel, volumétrica, mapa de luz, radiosidad, mapa de brillo, etc Mapas de sombras, sombras volumétricas Multi-textura, *bumpmapping, mipmapping,* texturas volumétricas, proyectadas, procedurales... *Vertex Shader, píxel shader.* Teselación (materiales DX11) Multiples efectos especiales

	Red: Cliente-Servidor, P2P, Servidor Maestro Sonido 2D, 3D, multicanal *Miles Sound System* (5.1 y 7.1) Inteligencia Artificial Trazado de rayos, proyección de rayos
Uso de licencia:	Gratis si es para propósito no comercial, si no: 99$ + 25% de todas las ganancias por encima de 50.000$
Código fuente disponible	No

Tabla 2

Requerimientos:

Mínimos:

- Windows XP SP2, Windows Vista, or Windows 7
- Procesador de 2.0+ GHz
- 2 GB RAM
- Tarjeta gráfica compatible con **Shader Model 3.0**
- 3 GB de espacio en el disco duro

Recomendados para desarrollo de contenido (desarrolladores profesionales pueden necesitar muchos recursos si crean un producto complejo y bien acabado):

- Windows 7 64-bit
- Procesador de 2.0+ GHz multi-núcleo
- 8 GB RAM
- Tarjeta gráfica NVIDIA series 8000 o superior
- Puede necesitar muchísimos objetos para contenidos, por lo que se aconseja mucho espacio en disco duro

Para desarrollo en DX11 (En la actualización de marzo de 2011, Epic introduce compatibilidad con DirectX 11, lo cual conlleva las posibilidades que ofrece el nuevo hardware):

- Windows Vista, Windows 7
- Procesador de 2.0+ GHz
- 2 GB RAM
- Tarjeta gráfica que soporte DirectX 11:
    - Nvidia: series 400 o superior
    - ATI: series 5000 o superior
- 3 GB de espacio en el disco duro.

Como se puede observar, un ordenador de gama media cumple con los requerimientos mínimos para poder usar UDK. El requerimiento más importante es tal vez disponer de una tarjeta gráfica que no sea demasiado antigua. Es fácil reconocer la versión de Shader Model que usa la tarjeta gráfica mirando sus especificaciones. En cuanto a conocer la versión de DirecX que estamos usando, podemos hacerlo mediante el fichero "dxdiag.exe", que podemos ejecutar desde la línea de comandos. No obstante, para

llevar a cabo un proyecto profesional, se aconseja mucho una gráfica de gama alta y lo más actual posible.

Nota: Pese a que El motor Unreal puede ejecutar en plataformas basadas en OpenGL, como MacOs o Linux, el kit **UDK no crea proyectos para estas plataformas, ni tampoco se puede adquirir una versión que funcione en dichas plataformas,** sólo iOS soporta proyectos UDK como alternativa a Windows. Desconozco el motivo de esto, no he encontrado información en páginas oficiales donde se haga mención de este asunto.

Software compatible con UDK y formatos que utiliza

Existe cantidad de material que puede, y debe ser importado al proyecto UDK desde software externo. Todo tipo de programas de creación digital pueden crear materiales, entre los más importantes están: **3ds Max**, **Maya, Blender** (poco aconsejable) para modelados 3D; **Zbrush, Photoshop**, **GIMP**, para creación de texturas y texturas UV (texturas aplicadas a modelos geométricos), **Bink** para los codec de vídeo, **Flash** para menús e interfaces, y todos los programas similares que puedan crear este tipo de datos.

Figura 5

UDK importa numerosos formatos de varios tipos de objetos, y es importante que los enumere con sus **extensiones**:

- Vídeos: bik (Bink Video)
- Modelos estáticos: ase, t3d, fbx (FBX)
- Modelos esqueléticos: psk, dae (Collada), fbx (FBX)
- Material: t3d (se crean en el propio editor de UDK)
- Textura: bmp, float, pcx, png, psd, tga
- Peliculas SWF: swf, gfx (Flash)
- Sonidos: wav
- SpeedTree: srt
- Animaciones Unreal, secuencias Unreal, terrenos Unreal: t3d (propias del entorno UDK)
- Paquete Unreal: T3DPKG

Sobre la licencia de uso

Los términos de la licencia de uso se detallan tanto en la página de descarga de UDK, como en la ventana de instalación del programa.En resumen, UDK es de libre uso, y los proyectos creados en él son también de libre distribución.
Escuelas y facultades universitarias pueden utilizarlo libremente con fines educacionales pero no comerciales. En cualquiera que sea el caso, si el proyecto creado se utiliza como producto con fines comerciales, se deberá tener una licencia por 99$, mas un 25 % en calidad de *royalties* por sus ganancias, si es que estas superan los 50.000$. La compra de la licencia sólo se deberá realizar una vez, pudiéndose crear más proyectos y comercializarlos sin comprar nuevas licencias.

Estructura de carpetas de la instalación

Los pasos de descarga e instalación del UDK son triviales y sencillos y no merecen explicación. Si que debemos tener en mente cual es la carpeta de instalación de UDK. Es importante elegir bien esta ubicación, ya que a partir de ahora trabajaremos en ella, y desde ella accederemos a todos los elementos y programas del UDK.
Una vez instalado el UDK, podemos observar que se ha creado la estructura de carpetas de la figura 6.

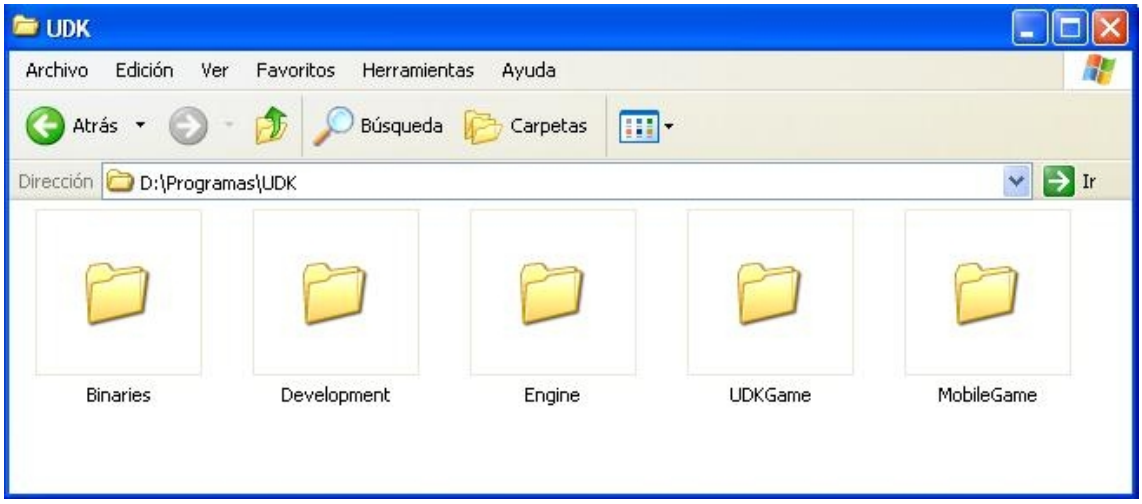


Figura 6

Estas son las carpetas básicas del UDK. Se explica brevemente la función de cada una de ellas:

Binaries: Contiene los ficheros ejecutables y binarios. En esta carpeta hay muchas subcarpetas, ya que el kit Unreal viene con multitud de aplicaciones que se ejecutan de forma independiente. Así pues, dentro de ella encontramos el ejecutable fundamental, UDK.exe, que lanza el editor Unreal; Unreal Frontend, muy útil aplicación para compilar nuestros proyectos y crear auto-extraibles del juego acabado; el SpeedTree, que sirve para modelar vegetación, y otros muchos ejecutables y paquetes para hacer que Unreal pueda trabajar con otros programas externos.

Development: Tiene dos subcarpetas: **Src**, es la ubicación para el código fuente del Motor Unreal. En ella se encuentran todos los paquetes, que a su vez contienen todas las clases que el *engine* usa. Dichas clases tienen un formato **.uc**.

La otra subcarpeta se llama **Flash**, y contiene todos los archivos relativos a la interfaz del juego, que están basados en el programa de diseño Adobe Flash.

Engine: Todo tipo de archivos, paquetes y binarios de los que se sirve el motor de Unreal. No es probable que tengamos que meternos en esta carpeta en el futuro.

UDKGame: Esta carpeta es la más importante para el desarrollador. En ella se depositan los archivos que se usarán en el juego. De sus subcarpetas, la carpeta **content** tiene todos los paquetes con los que luego podremos trabajar en el diseño del nivel. Se trata de objetos como modelos, texturas, materiales, etc.
Hay que destacar dos tipos de extensión importantes: .upk es para los paquetes de objetos para el nivel. .udk es la extensión para los mapas. Estos se encuentran en la subcarpeta **maps**, y en ella se guardarán los mapas que creemos en el Editor.

MobileGame: Esta carpeta es análoga a UDKGame, pero en ella se almacenan todos los contenidos relativos a los proyectos para dispositivos móviles. La plataforma de PC y la de un dispositivo móvil tienen grandes diferencias a nivel gráfico y de rendimiento, por lo que se asume tal diferenciación.

Comenzando con UnrealEditor – Fundamentos Básicos

El primer contacto con UDK será a través de su programa principal, el UnrealEditor, que es el archivo UDK.exe que se encuentra en la carpeta **Binaries.**

Nada más ejecutarlo, se nos presenta una ventana de bienvenida, y posiblemente la ventana *content browser*. De momento cerramos estas ventanas. Nos encontraremos con la pantalla del Unreal Editor (figura 7)

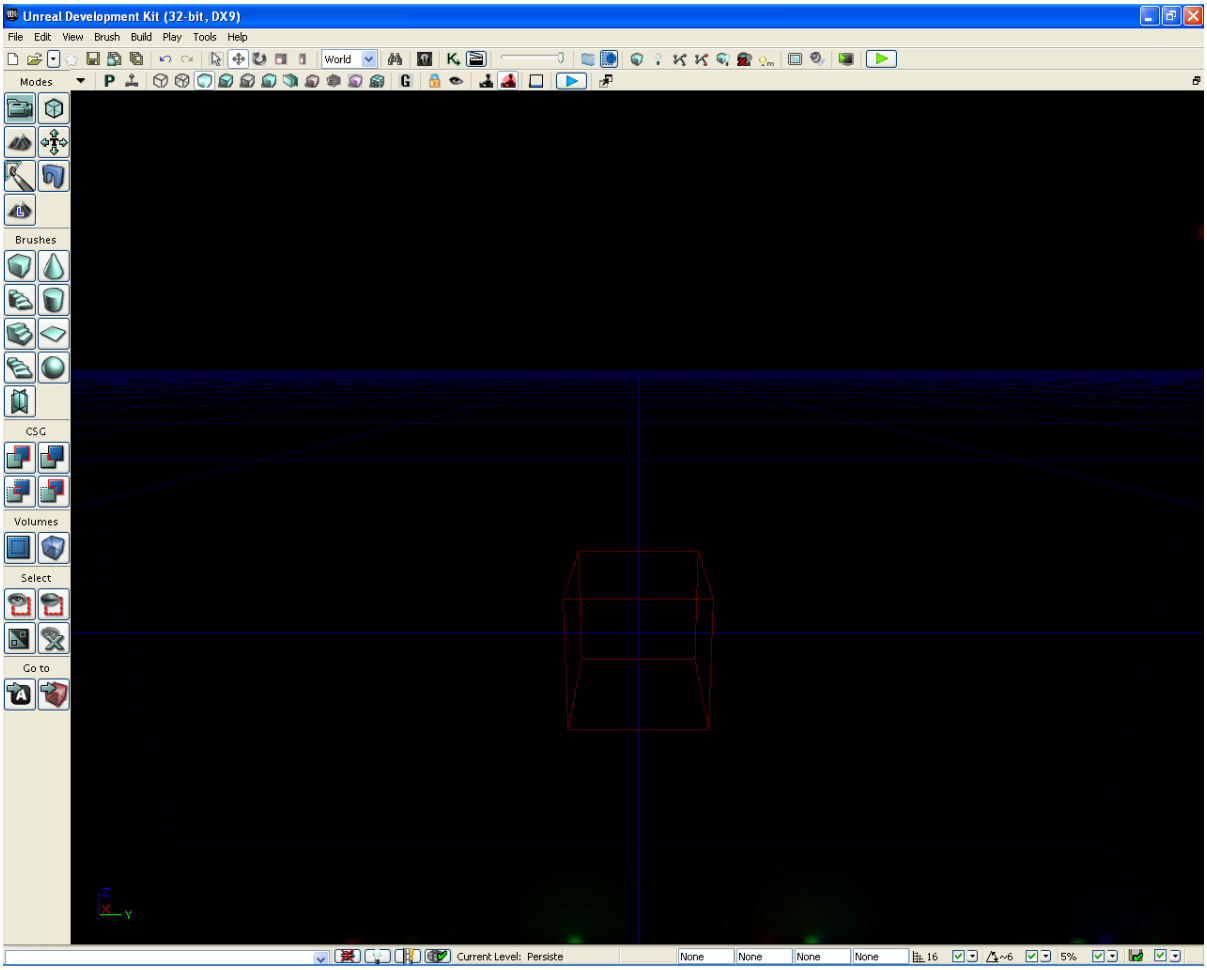


Figura 7

Estamos ante una interfaz para un editor 3D, pero si nos fijamos hay algo más que eso.

En la barra de herramientas de la izquierda, encontramos una serie de funciones dedicadas a la creación 3D; se explicarán más adelante.

En la barra de herramientas superior, podemos ver que existen varios botones que en realidad son accesos a otras utilidades del UDK, que en principio son ajenas al UnrealEditor. Hay que destacar, como botones importantes, el *Content Browser*,

representado por unos prismáticos, *Kismet*, que es la *K* que hay a su derecha y *Matinée*, que es la claqueta.

¿Qué es un mapa?

Vamos a introducir el concepto de mapa en videojuegos, por si acaso no se tiene claro. La idea es sencilla, los juegos en Unreal están basados en mapas. Un mapa es el escenario que se ha creado para que en él se desarrolle el juego.

Valga la redundancia, es un mapeado que se ha diseñado mediante herramientas de creación 3D que pueden ser muy diversas. Y la finalidad es movernos por este escenario.

Un mapa exige una carga previa independiente para ser visualizado y/o jugado. En un juego típico, nos referiremos al mapa como cada uno de los niveles en los que vamos a jugar (aunque varios mapas pueden importar unos mismos parámetros u objetos básicos.) En Unreal Tournament, cada mapa es un "mundo" diferente. Así pues, un nivel no necesariamente tiene que ser un mapa, ya que por motivos de economía de recursos, se puede descomponer dicho nivel en varios mapas. Los mapas de motor Unreal suelen requerir una cantidad considerable de recursos para su carga, por eso los juegos basados en Unreal típicamente presentan una mayor o menor modulación en mapas.

Esto es un aspecto fundamental que hace que los juegos basados en Unreal no se suelan utilizar para crear entornos inmensos propios de juegos de rol, por ejemplo. No obstante, para un proyecto como el que nos atañe, el Motor Unreal puede cumplir sin ningún problema (por no mencionar las diversas opciones de optimización que tiene implementadas).

Unreal editor es la herramienta con la que crear mapas, los cuales tienen como extensión el tipo de fichero .udk. En la figura 8 podemos ver el mapa FoliageMap.udk, cargado en UnrealEditor.

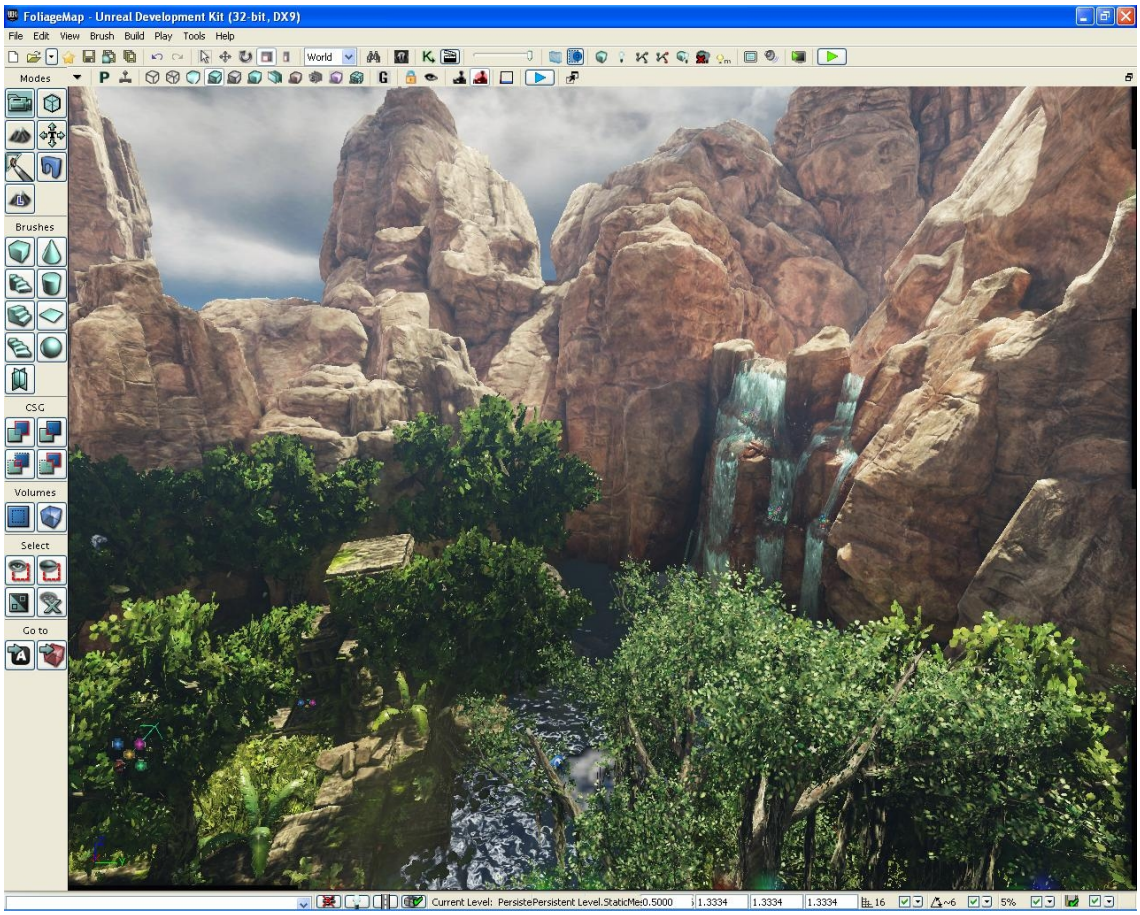


Figura 8

En Unreal, un mapa no tiene por qué existir exclusivamente como mundo del juego. Como se indica en el apartado de *Configurar el entorno de desarrollo*, El motor carga un mapa en el momento que se lanza el juego. Así pues, si queremos una interfaz a modo de menú principal, que no sea directamente el juego, debemos crear un mapa .udk, en el que se visualice una interfaz a modo de menú, que podremos crear con elementos de UnrealEditor, o lo que es más aconsejable, con la aplicación ScaleForm (disponible en el kit), que lo que hace es dibujar sobre la pantalla. Así podemos crear simplemente un mapa vacío pero con una interfaz ScaleForm que dibuja las opciones y un cursor para el ratón.

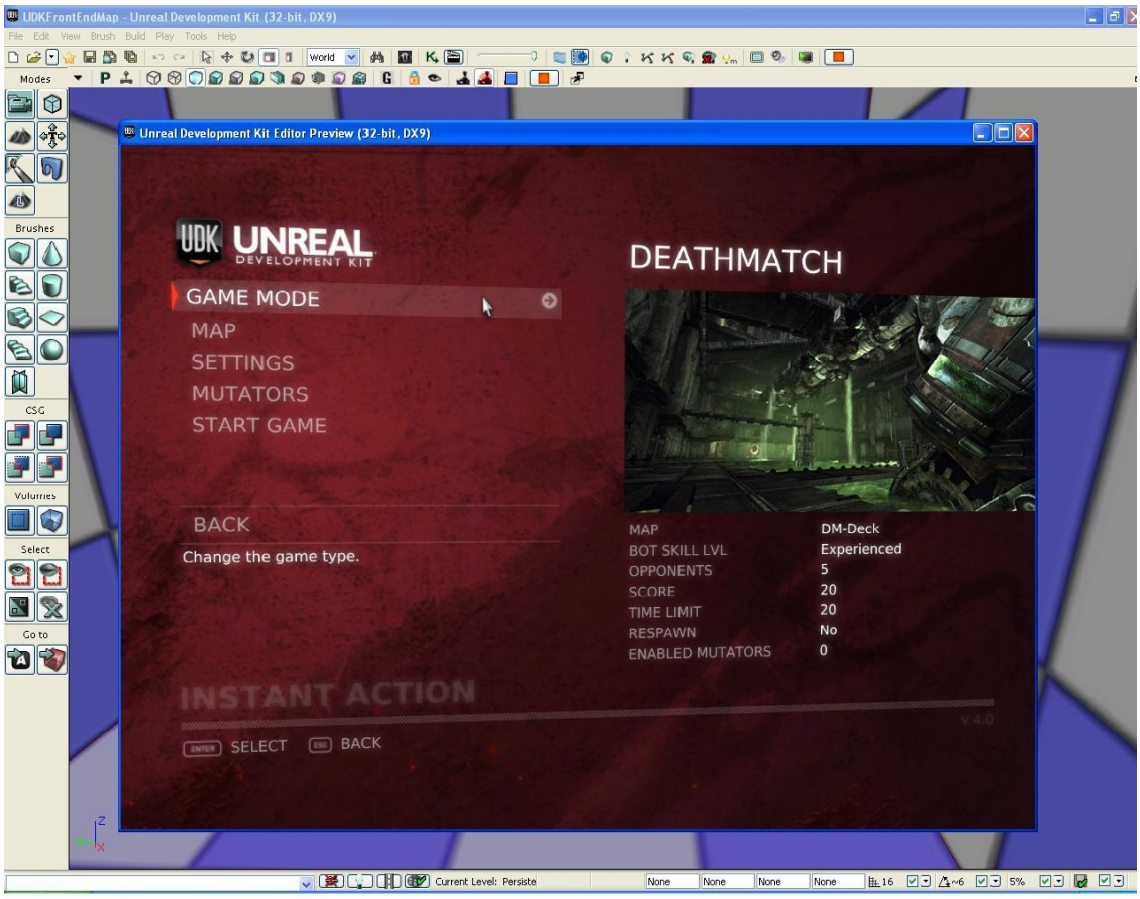


Figura 8

El mapa *UDKFrontEnd* no es mas que un espacio vacío. Lo que se ve en la figura 8 es un objeto *ScaleForm* programado para mostrarse delante al ejecutar el juego.

Los modos de Vista

En Unreal Editor, cada una de las ventanas muestra en su parte superior una barra de herramientas, entre las que se encuentra la configuración de vistas (Véase la figura 9.). Es importante entenderlas antes de comenzar a crear un mapa.

Figura 9

1: Define el tipo de perspectiva: Ortográfico: Base, perfil y alzado, o en modo Perspectiva.
2: Muestra el mapa en tiempo real, es decir, aquellos actores dotados de movimiento, se moverán durante la edición.
3: Muestra la malla de las brochas que estamos empleando en nuestro mapa. Es el modo aconsejado para trabajar con brochas.
4: Muestra las mallas de todos los polígonos. Puede ser útil, pero también puede recargar demasiado la vista.
5: Modo sin iluminar: muestra el mapa y todos los materiales o texturas tal cuales son, pero sin tener en cuenta ningún aspecto de iluminación que se haya podido configurar, es decir, no hay fuentes de luz ni sombras.
6: Modo con iluminación: muestra el mapa tal cual se verá durante el juego, con todos los detalles de luz y sombra.
7: Detallado de luz: Muestra los materiales tal cual están iluminados pero elimina los detalles de luz difusa y color especular.
8: Muestra las superficies sin aplicar los materiales, sólo la iluminación que tienen.
9: Mide la complejidad de luz. Útil para analizar el esfuerzo de nuestra GPU iluminando el mapa.
10: Densidad de texturas.
11: Complejidad de *shaders*.
12: Densidad del mapa de luces
13: Sólo iluminación usando densidad de *téxel*.

Los botones 7-13 ofrecen modos de vista destinados a la optimización de nuestro mapa, y son para usuarios más avanzados, por lo que no nos preocuparemos mucho por ellos.

Modelando con *brushes*

Las brochas tridimensionales, o *brushes*, que es como se denominan en el editor, y que es el término que usaremos por comodidad, son el fundamento del modelado 3D en UnrealEditor.
El *brush* actual es el volumen rojo que se nos presenta en la pantalla, y que al inicio tiene forma de cubo.
Podemos ver el juego de *brushes* disponible en la barra de herramientas de la derecha. Existen *brushes* para las figuras geométricas básicas. Cuando seleccionamos una, nuestro *brush* adquiere la forma que hemos seleccionado.
Este proceso lo único que hace es definir la forma que tendrá nuestra brocha. A continuación, nos interesa **crear** esa forma. En la barra de herramientas, debajo de los brushes, tenemos las funciones CSG, que son el tipo de operación que vamos a realizar con la brocha. Existen 4 tipos: Aditiva, substractiva, intersección, y desintersección.
Para emplazar un modelo con nuestro *brush*, simplemente pulsaremos en aditiva, o bien *ctrl-a*. Si lo que queremos es crear un hueco, es decir, realizar una operación de substracción, utilizaremos la función *substract*. Las operaciones *intersect* y *desintersect* simplemente cambian la forma del *brush* según esté combinada con el *brush* que esté superpuesto a él.

Aparte de poder emplazar *brushes* simples, siempre podemos después editarlos activando el modo geométrico o *Geometry Mode*. En este modo, se abre una interfaz nueva que nos permite editar el modelo con múltiples herramientas, como mover sus vértices, aristas, rotando, creando extrusiones, ect.

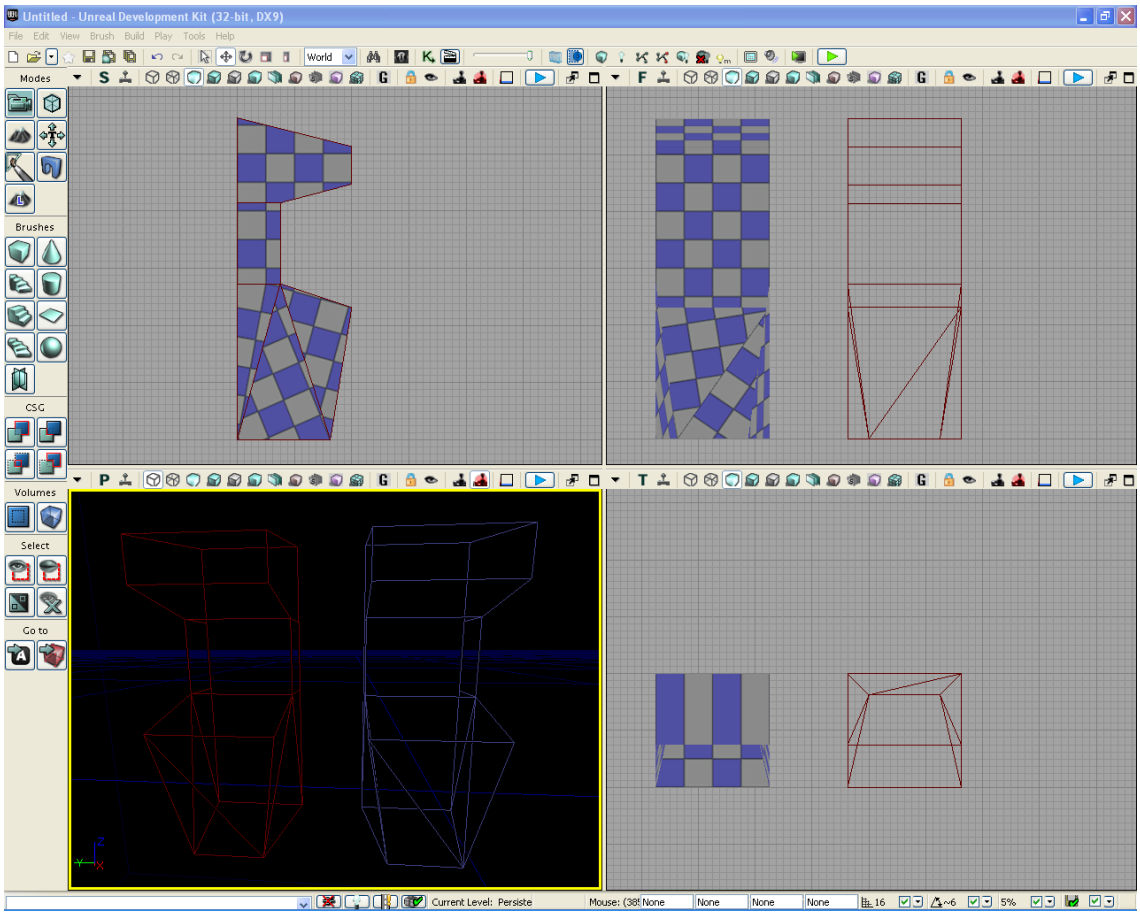


Figura 10

"Cabina de recreativa" modelada en *Geometry Mode*

En la figura 10 podemos ver un modelo improvisado en sus vistas de alzado, planta, perfil y perspectiva. La malla roja es el *brush,* el modelo que tiene al lado es el modelo Proyectado por dicho *brush*. Es aconsejable trabajar con este tipo de vistas cuando se esté trabajando en modo geométrico.

Por último, siempre podemos disponer de las típicas herramientas de traslación, rotación y reescalado, que todo iniciado en el 3D debe conocer.

Se recomienda ver la documentación disponible sobre el editor Unreal y edición con *brushes*, ya que la información es tan extensa que no tiene sentido intentar plasmarla en este documento. **[4]**

Content Browser (Navegador de contenidos)

Es una de las herramientas pilares dentro del UDK. Esta herramienta provee la necesaria gestión de todos los **bienes** (a partir de ahora los llamaremos *assets*, para estar en sintonía con la jerga habitual del UDK), y **clases** dentro de nuestro mapa.

Los assets son todos los bienes u elementos que de un modo u otro participan en un mapa. Desde modelos estáticos importados, a efectos de sonido, pasando por texturas, materiales, sistemas de partículas o animaciones ScaleForm.

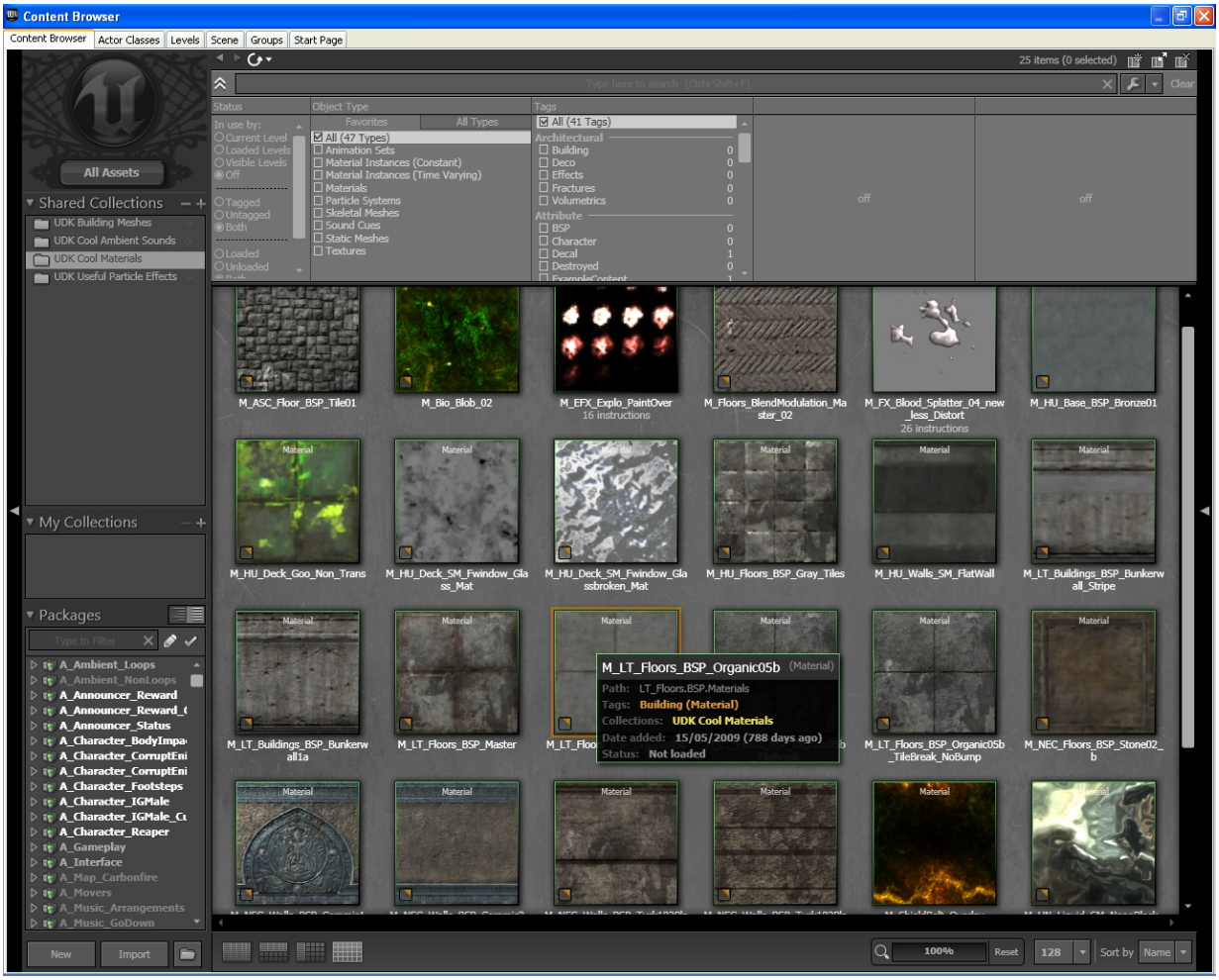


Figura 11

En todo momento a lo largo de la creación del mapa vamos a tener que pasar por aquí.

El kit UDK viene por defecto con una colección de *assets* que podemos usar libremente en nuestras creaciones. Sin embargo veremos que la variedad de estilos es limitada y vamos a tener que hacernos con nuestros propios recursos.

El Navegador de contenidos (figura 11) ofrece una interfaz muy intuitiva, y da un profundo sentido organizativo a los bienes. Podemos organizar los *assets* en **categorías**,

colecciones y **paquetes**. La estructura de paquetes es fundamental, ya que los *assets* se organizan dentro de la instalación de UDK por paquetes, que pueden englobar *assets* relativos a un mismo proyecto. Si decidimos crear nuestros propios *assets*, o bien importarlos, es aconsejable crear nuestro propio paquete donde los depositaremos. La organización por categorías es simplemente seleccionar los *assets* según de qué tipo son. Las colecciones son agrupaciones que nosotros mismos podremos realizar para mantener relacionados los *assets* que nos interesen. Se puede además añadir etiquetas a cada *asset* para facilitar su búsqueda, puesto que el navegador de contenidos ofrece un potente buscador. Hay que advertir que la cantidad de *assets* que se puede acumular es enorme, lo cual justifica tal esfuerzo en mantener esa organización.
Cada *asset* se representa por una pequeña ventana, con una previsualización del objeto. Para abrirlo, simplemente hacer doble clic sobre dicho objeto. Se abrirá el editor correspondiente según el tipo de formato que sea el *asset*. Por ejemplo, si el objeto es un material, se abrirá en Unreal Materia Editor. Si es un efecto de partículas, se abrirá con Unreal Cascade. Cada uno de estos editores permite tanto la visualización como la modificación del objeto.

Dentro del mismo content browser, existen otras pestañas que nos dan a acceso a mas facetas organizativas útiles:

Actor classes: Las clases, como todo programador puede intuir, hacen referencia a cualquier tipo de objeto, ya sea tangible en el juego, o bien de carácter abstracto, que pase a formar parte de nuestro mapa.
Todo elemento es una clase, y muchas de esas clases no se encuentran en el navegador de contenidos principal. Las clases pueden ser elementos de aspecto visual, como luces, niebla, destellos, modelos animados, filtros de postprocesado, etc. y pueden también ser elementos no visuales, como definidores de propiedades físicas, cámaras para hacer cinemáticas, patrones para IA, o elementos dotados de jugabilidades específicas.
Actor classes organiza todas las clases de forma jerárquica, según se estructuran por medio de los mecanismos de herencia. Como desarrolladores, podemos crear nuestras propias clases para implementarlas en nuestro juego, pero ello conlleva meterse de lleno en UnrealScrpt.
Levels: Gestiona el *level streaming*, es decir, podemos trabajar en varios mapas cargados al mismo tiempo, y movernos de uno a otro mediante esta función.

Scene: Muestra todas las primitivas que existen definidas en el nivel, y muestra sus parámetros relativos a su renderización.

Groups: Todos los objetos de un mapa, los *assets* utilizados, así como otros objetos del editor como brushes, luces, nieblas, efectos de postproceso, etc. pueden ser agrupados. Todo grupo puede ser ocultado/mostrado (al estilo diseño por capas).

Esto ofrece la ventaja de que podremos ir atravesando las diferentes etapas de creación del mapa, pudiendo quitar de nuestra vista aquellos elementos que nos interfieren en el trabajo por pertenecer a otra etapa. Por ejemplo, si queremos trabajar en la colocación de los muebles de una habitación, se nos hará más cómodo hacerlo si no aparecen los objetos de las habitaciones contiguas, ya que en el editor podremos ver a través de las paredes, y nos estorbarían.

Start page: Permite navegar por la Unreal Developer Network, página fundamental para desarrolladores Unreal, y que nos mostrará todo tipo de documentación y noticias relativas a UDK y, muy importante, información sobre actualizaciones.

CREANDO UN MAPA

Los siguientes aspectos importantes a conocer en UnrealEditor se explicarán mediante el seguimiento de cómo se construye un mapa sencillo.

Hay que entender que UnrealEditor provee una herramienta de diseño 3D limitada, es decir, que no se pueden crear objetos detallados o con alta cantidad de polígonos. En cambio, puede ser suficiente para crear un edificio básico creando bloques a modo de tabiques. Aún así, si en nuestro mapa queremos que nuestras construcciones gocen de un acabado decente, vamos a necesitar de forma muy aconsejable una herramienta tipo 3D Studio con la que crear modelos para embellecer y hacer realistas las superficies.

Construir *brushes*

Para empezar, aconsejo que se trabaje en la configuración de vista 2x2, que se puede poner en el menú: *View, Viewport configuration, 2x2 split.*

Crearemos una superficie que haga de suelo, igual que en la figura 12. Lo podemos hacer poniendo cualquier tipo de brocha y aplanándola, como un cubo, o bien seleccionando la brocha *sheet*, y lo aplicamos con la adición:

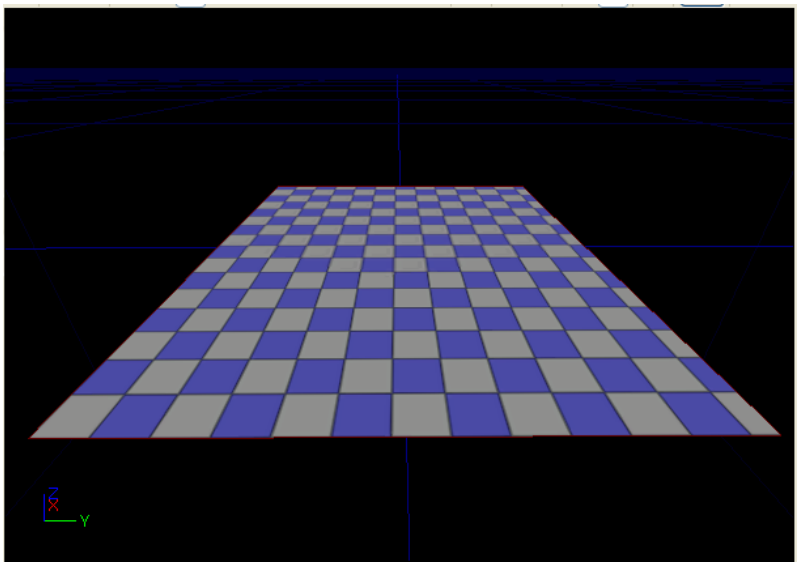


Figura 12

Hay que elegir bien la extensión del suelo, aun que en un futuro siempre podremos reescalarla según nuestra necesidad.

El siguiente proceso es el de añadir las paredes. Para ello, tenemos que elegir de antemano un grosor fijo para nuestros tabiques. Una buena opción es el de 16.

Seleccionamos la brocha cubo con el botón derecho y lo configuramos tal cual vemos en las figuras 13 y 14.

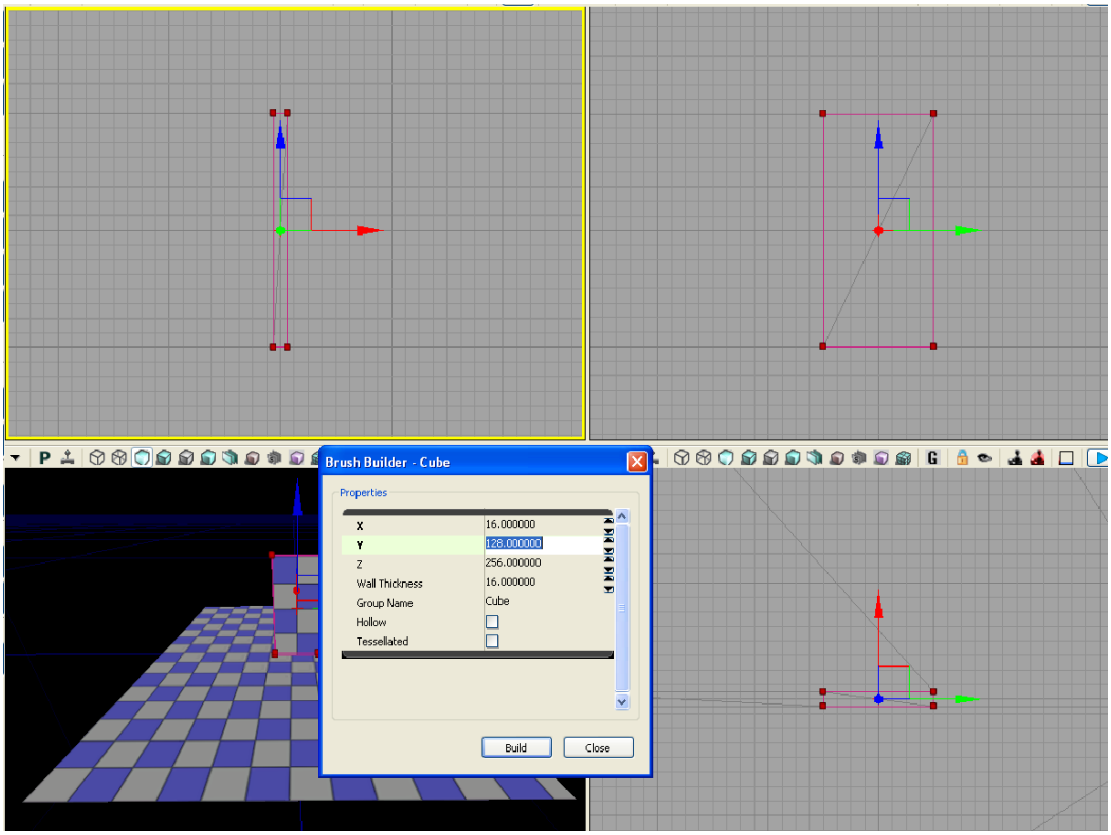


Figura 13

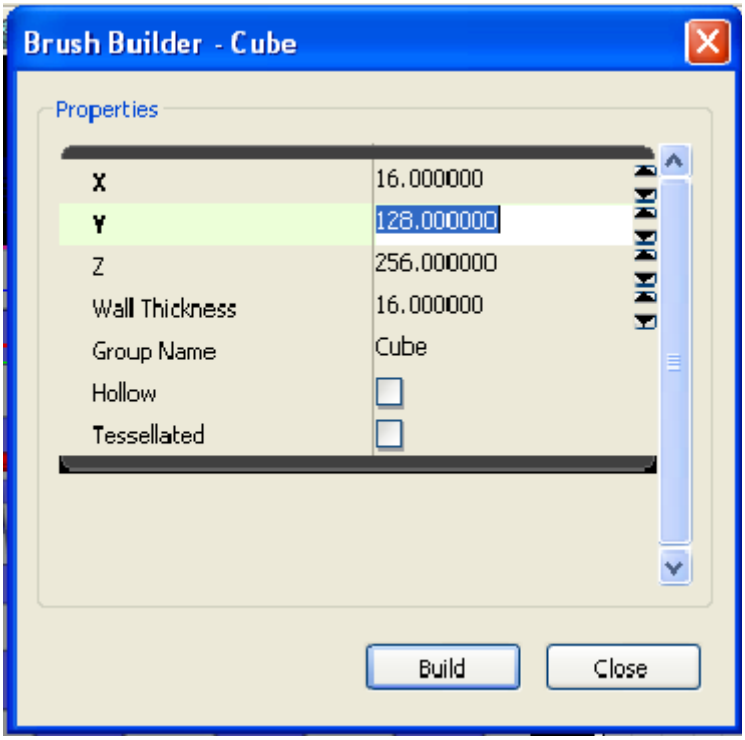


Figura 14: Detalle del *Brush Builder*

Donde Z será siempre la altura del muro, y X o Y será la anchura y longitud respectivamente. El apartado *Wall Thickness* sólo sirve si hemos seleccionado *Hollow* (hueco), y sirve para definir el grosor de las paredes del cubo en caso de que lo creemos hueco.

Así pues, con la herramienta de traslación vamos moviendo y rotando para colocar todos los muros que vamos creando. Para crear un techo, construiremos la brocha de forma que Z sea 16, y X e Y sean anchura y longitud del techo.

Muy importante: El editor 3D está sujeto a una **rejilla**, es decir, que los *brushes* que coloquemos sólo podrán disponerse de modo que se muevan entre múltiplos de la distancia espacial especificada. Esto ayuda enormemente a que nuestros muros queden perfectamente alineados y no haya problemas para colocar fácilmente cada muro donde le corresponde.

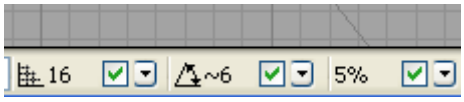


Figura 15

Las escalas de la rejilla se pueden modificar; en la esquina inferior derecha vemos la configuración de rejilla (figura 15). A la izquierda tenemos la escala de arrastre, en medio la escala de rotación, y a la derecha la proporción en la que se redimensionan los objetos. Cuanto más pequeño sean estos valores, más detallada será nuestra edición, pero cuidado, por que también se hará mas compleja y más fácil que acabemos arrastrando errores que nos obligue a reducir aún más la escala para compensar. Es por esto que es muy importante idear antes que nada las medidas de nuestros tabiques, con el fin de que encajen bien entre ellos.

En la figura 16 podemos ver dos tabiques que no se ajustan debido a una mala configuración de rejilla.

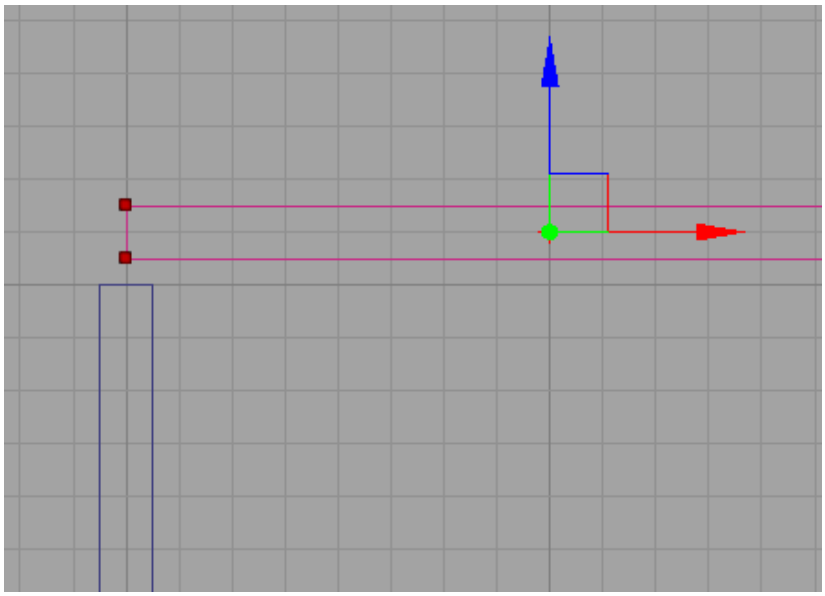


Figura 16

Una forma muy útil de ajustar esto es hacer clic derecho sobre el vértice del rectángulo (acción que se llama *snap to grid*) y que mueve el rectángulo de forma que se ajuste a la rejilla. Véase en la figura 17 cómo queda después de hacer *snap to grid* sobre las dos figuras anteriores:

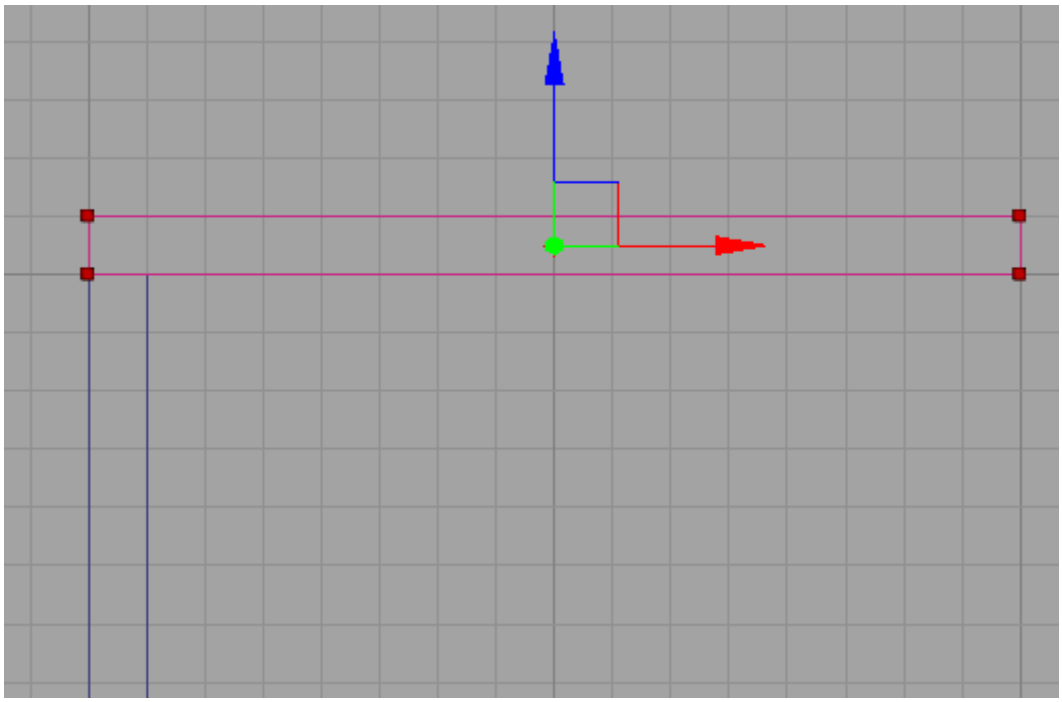


Figura 17

Una vez tenemos nuestros tabiques dispuestos, es lógico que queramos que haya huecos en éstos, como ventanas o puertas. Para esto utilizamos la herramienta CSG substractiva.

A diferencia de los *brushes* creados de forma aditiva, que se representan en azul en la vista *brush wireframe*, los que se crean de forma substractiva se representan de color amarillo (figura 18).

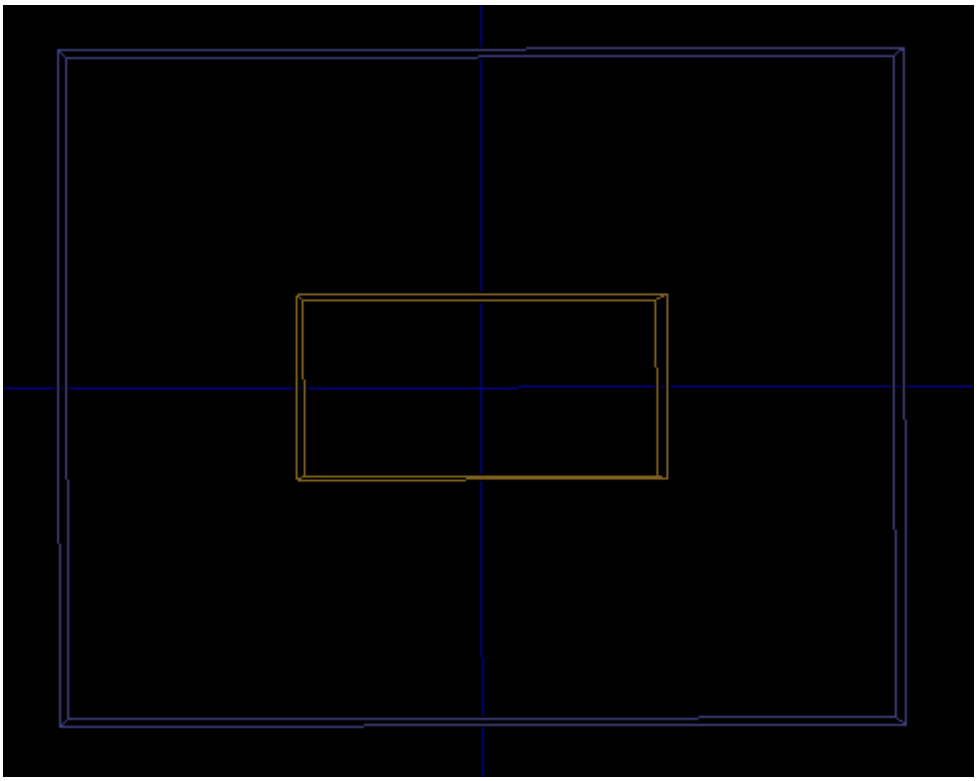


Figura 18: Un *hueco* creado dentro de un polígono rectangular

Tenemos además una herramienta para crear escaleras, que nos ahorra un gran trabajo, y nos permite elegir todo tipo de parámetros, como el alto o ancho del escalón, la cantidad de escalones, las dimensiones de la escalera…

Las formas cilíndricas son ideales para crear columnas, y las triangulares también para otro tipo de estructuras. Si a esto le añadimos un trabajo en *Geometry Mode*, podemos crear brochas de la forma que deseemos, (como se vio en el ejemplo). No obstante se aconseja que la cantidad de polígonos de nuestras brochas no sea muy grande.

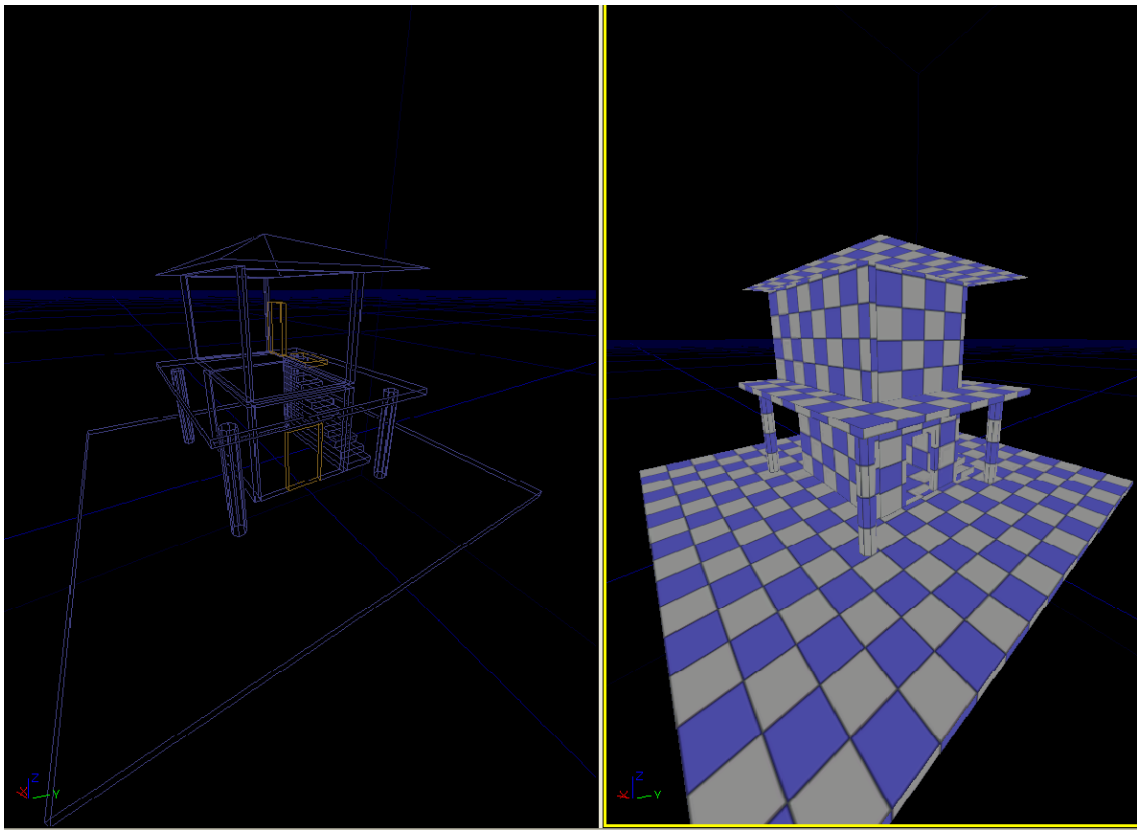


Figura 19: **Sencilla casa creada con brochas básicas**

Poniendo Materiales

Los materiales son un tipo de objetos que pueden llegar a ser complejos.

Un material se suele basar en una o varias texturas (aunque no necesariamente), son imágenes, y una serie de operaciones que se aplican sobre ellas. Estas operaciones determinan cómo se va a renderizar ese material, y hay una enorme cantidad de posibilidades, por ejemplo hacer que tenga luz emisiva, difusa, especular, o añadirle transparencia, o distorsión. UDK permite crear materiales con una herramienta llamada *Unreal Material Editor*, a la que se puede acceder fácilmente desde el navegador de contenidos.

En la figura 20 se muestra un material sencillo y su configuración. Se trata de una superficie empedrada que cuenta con una textura para la imagen de las piedras, y otra textura para el mapa de normales. En el *Material Editor*, configuramos el material de modo que tenga como resultado una superficie con brillo especular (que da aspecto de pulidez a las rocas), así como luz difusa (de modo que refleje la luz), y lo combinamos

con la textura de mapa de normal, que llevaremos a su canal correspondiente. Esto hará que las piedras reciban un sombreado que les de sensación de profundidad.

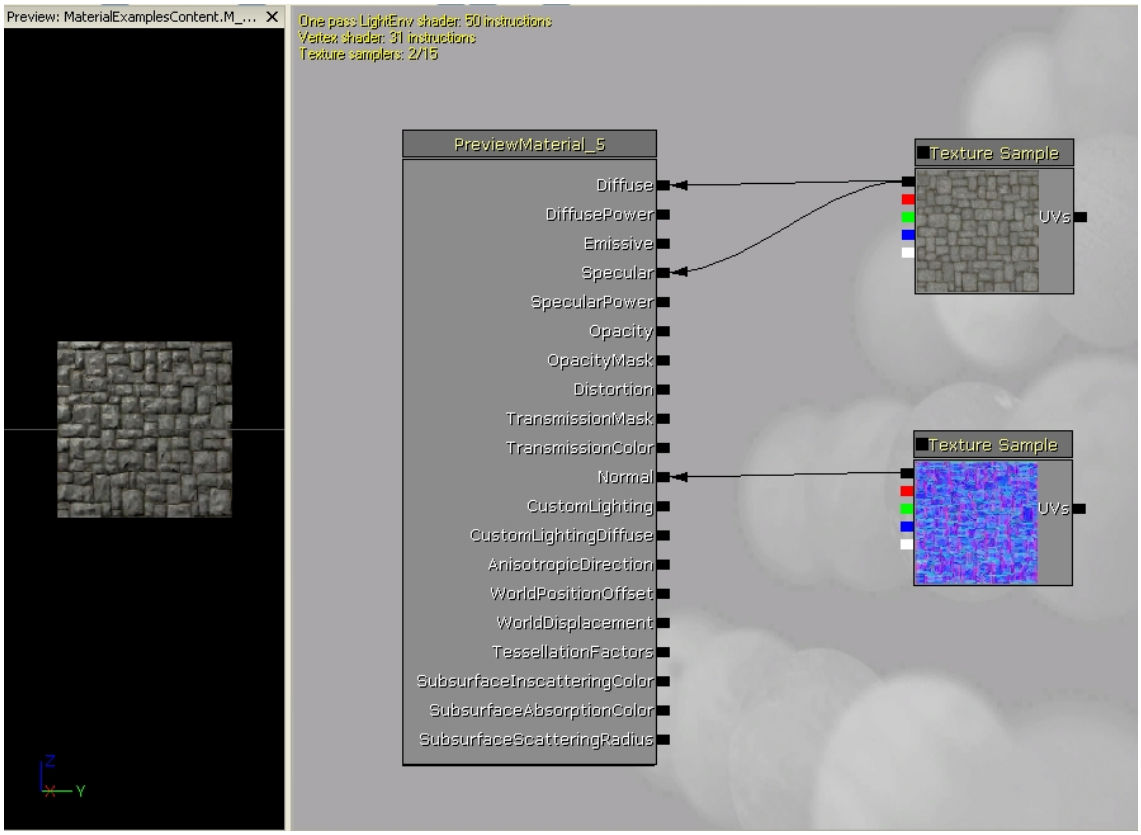


Figura 20

A nuestro material podríamos añadir además capas como superficies animadas, dando resultados cosas útiles como por ejemplo la posibilidad de crear un material de agua.

Para aplicar un material sobre una superficie, simplemente hacemos clic derecho sobre la superficie y elegimos *Apply Material:…,* o bien directamente arrastramos el icono del material desde el Navegador de Contenidos hasta la superficie deseada.

Por supuesto, para que se aprecie el material debemos poner alguna fuente de luz, y seleccionar el modo de vista *Lit Mode* (iluminado) o de lo contrario no apreciaremos el material correctamente.

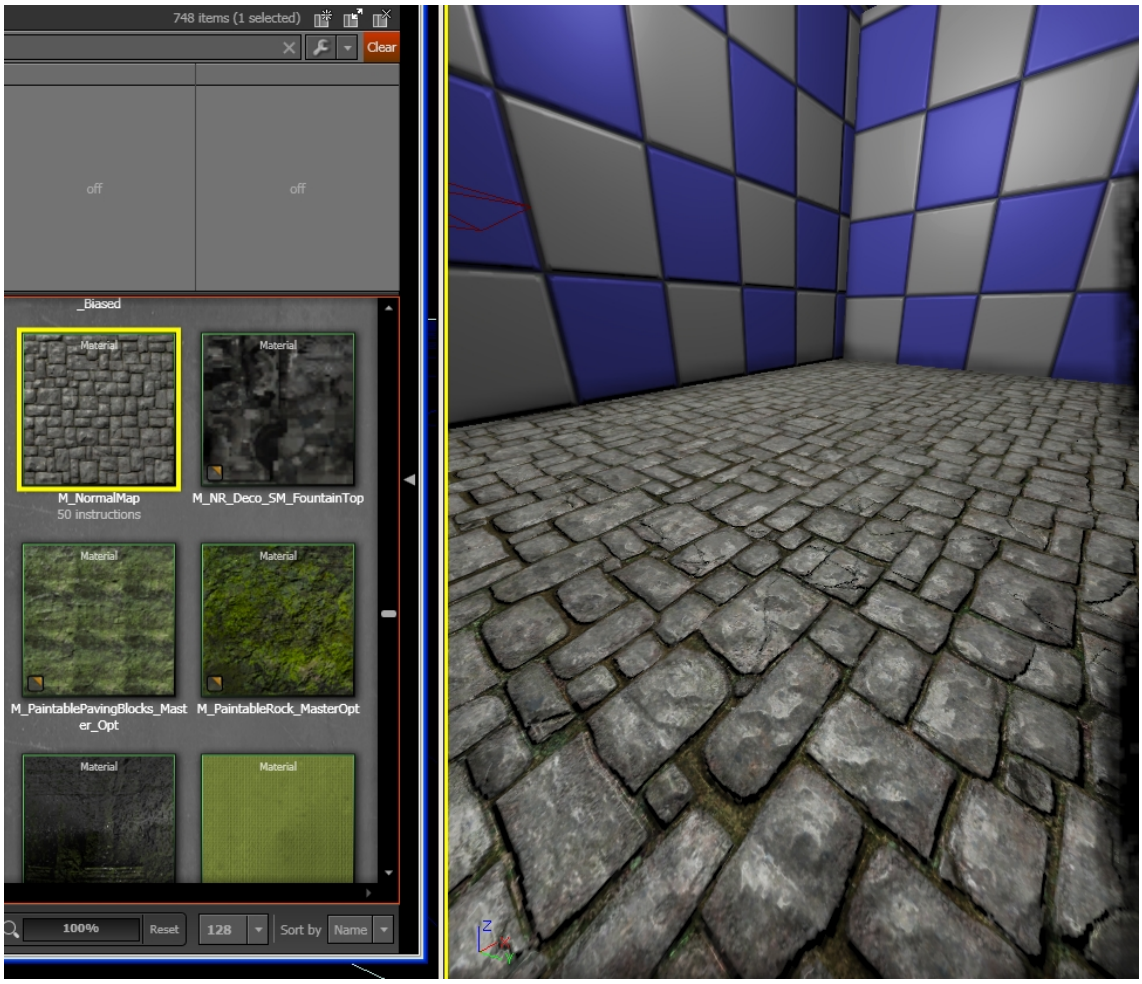


Figura 21

Como podemos ver en la figura 21, bajo una iluminación, las piedras tienen sensación de relieve gracias al mapa de normal, y un brillo especular.

Añadiendo modelos estáticos

Un paso que sucede al del modelado de las paredes y techos del edificio, es el de añadir modelos estáticos. Un modelo estático es un objeto 3D modelado e importado al UnrealEditor. Los mapas habitualmente están llenos de modelos estáticos (Véase la figura 22). Su función puede ser la de terminar de componer el mapa, o dar los adornos necesarios, como añadir muebles, lámparas para las fuentes de iluminación, vegetación, escaleras, pasarelas, cristales, y un largo etcétera.

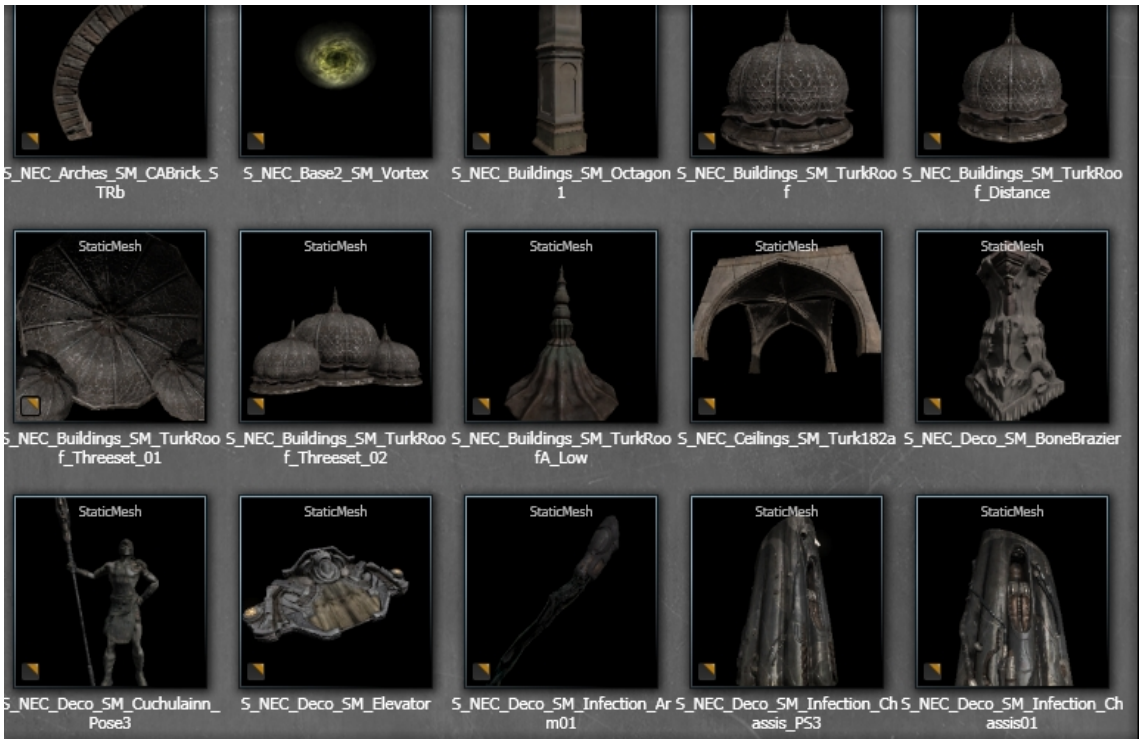


Figura 22

El *Content Browser*, es el que se ocupa de gestionar esto. Para colocar un modelo estático en un mapa, simplemente lo seleccionamos en el icono que lo represente en el *Content Browser*, y después hacemos clic derecho sobre el punto del mapa que deseemos y seleccionamos *addStacicMesh*…

Aparecerá el modelo en nuestro mapa. Una vez puesto, somos libres de moverlo, rotarlo, o redimensionarlo a nuestro gusto para que conforme la escena. No hay ningún problema en copiarlo todas las veces que sea necesario para tener réplicas de éste en nuestra escena. Para ello, simplemente seleccionamos el modelo, pulsamos *Alt*, y lo arrastramos. Lo que ocurrirá es que se creará una copia con los mismos atributos y proporciones. Usaremos esto cuando necesitemos un mismo objeto replicado muchas veces, como las farolas de una calle, o unas cajas apiladas.

Iluminación

Es uno de los aspectos más determinantes de nuestro mapa. En mi experiencia, puedo decir que obtener una iluminación lograda no es trivial ni es muy intuitivo, y la mejor forma de conseguirla no es más que haciendo muchas pruebas diferentes y experimentar con todas las combinaciones.

Si vamos al *Actor Classes*, y desplegamos el arbol *Lights*, encontraremos la lista de opciones de iluminación existentes (fig. 23)

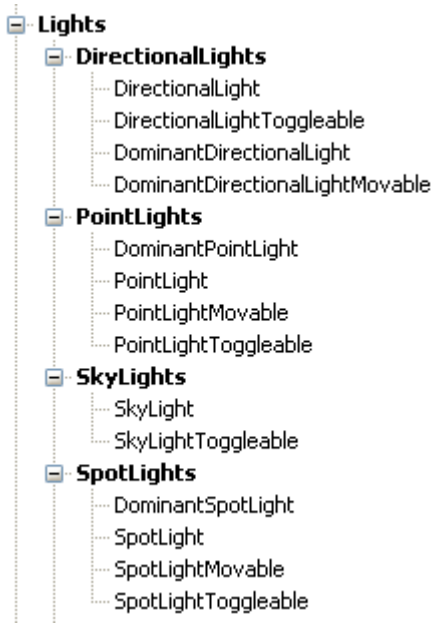


Figura 23

Estas opciones son fundamentalmente 4:

PointLights: Son las luces estándar de *UnrealEditor*. Consisten en una fuente de luz puntual que emite en todas direcciones. (fig. 24)

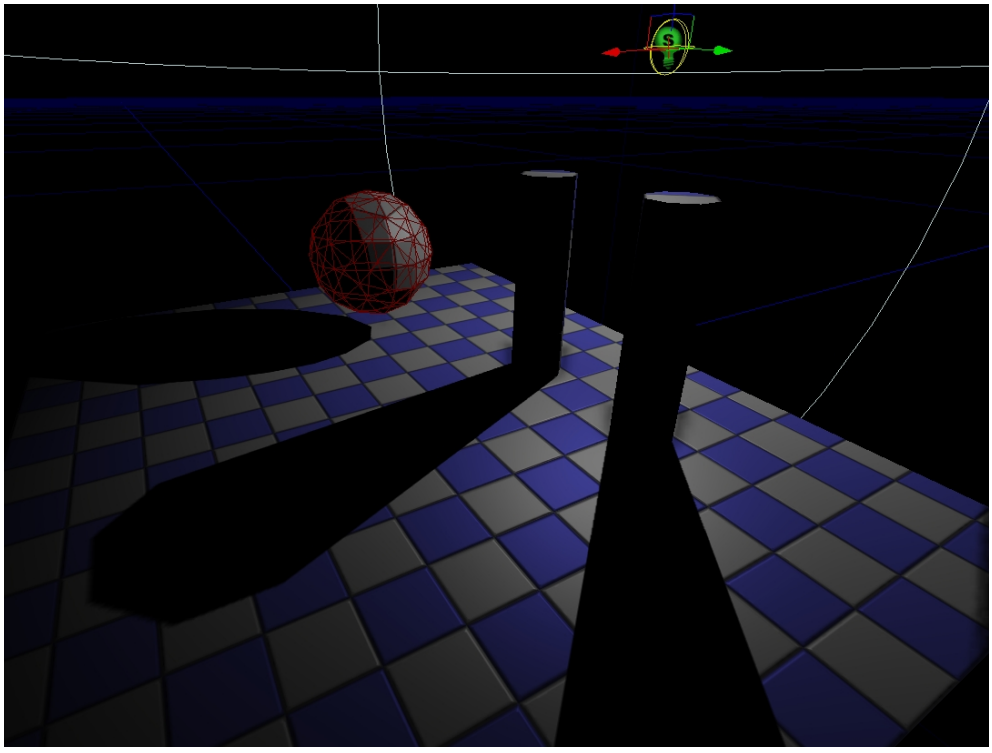


Figura 24

DireccionalLights: Son luces con fuente de radio es infinito, esto quiere decir que todas las sombras que proyecta un de estas luces son paralelas. Si queremos crear una luz solar, debemos usar una *DominantDirectionalLight*. (fig. 25)

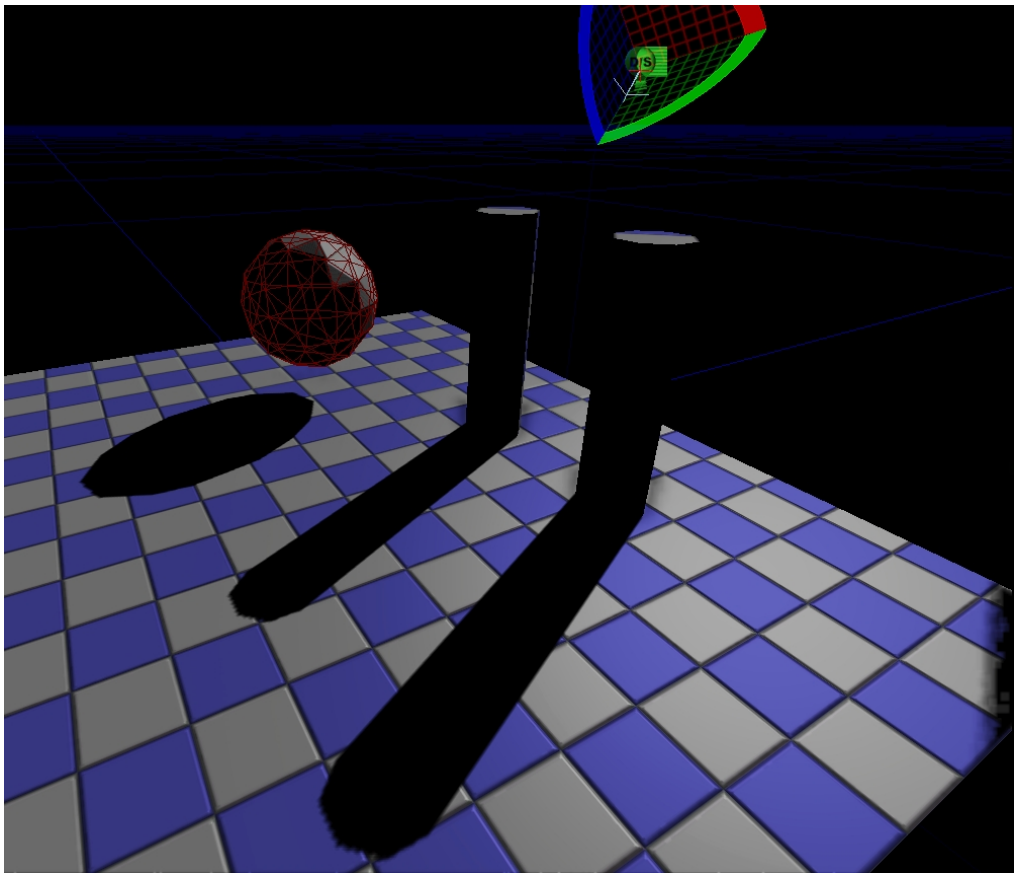


Figura 25

SkyLights: Es la luz de relleno o luz ambiental. Se debe usar esta luz para simular que existe luz en espacios no iluminados directamente, ya que de lo contrario toda zona que estuviese en sombra, se vería completamente negra. Se aconseja configurar **siempre** un *skylight* ya que es un elemento necesario para dotar el mapa de iluminación ambiental. Usaremos un nivel de brillo moderado para mapas diurnos, y un brillo más reducido para mapas nocturnos. (fig.26)

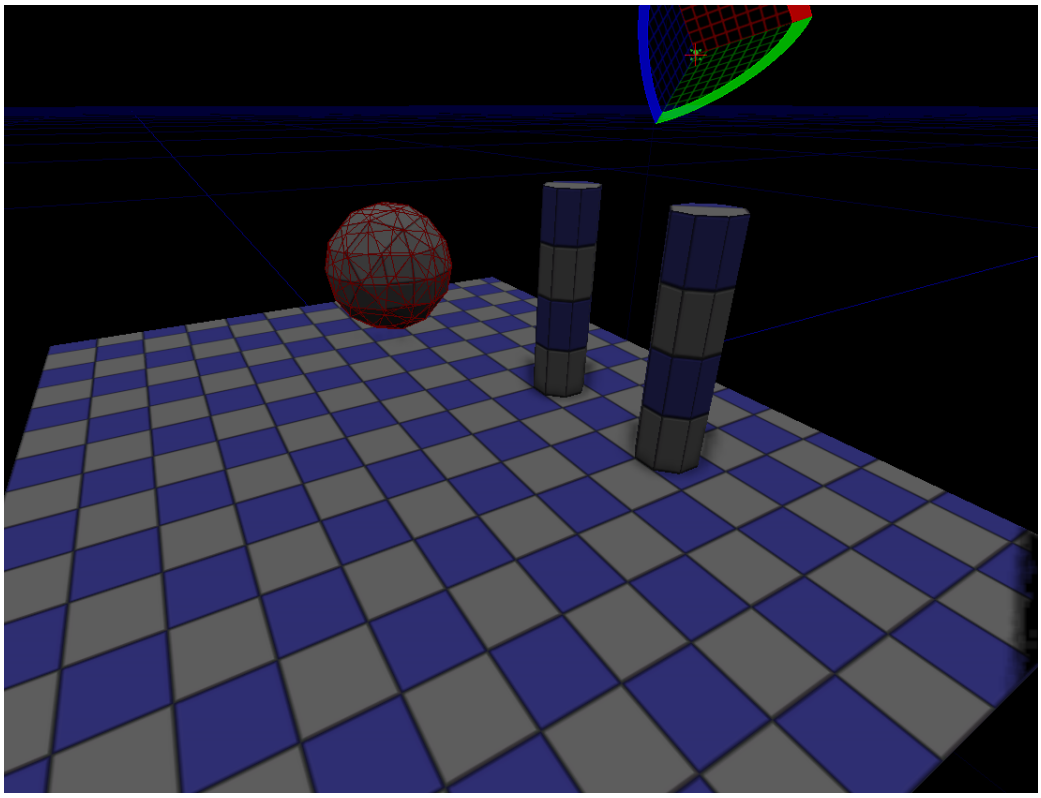


Figura 26

SpotLights: Son fuentes de luz cuya proyección es cónica (para simular focos o lámparas) (fig. 27).

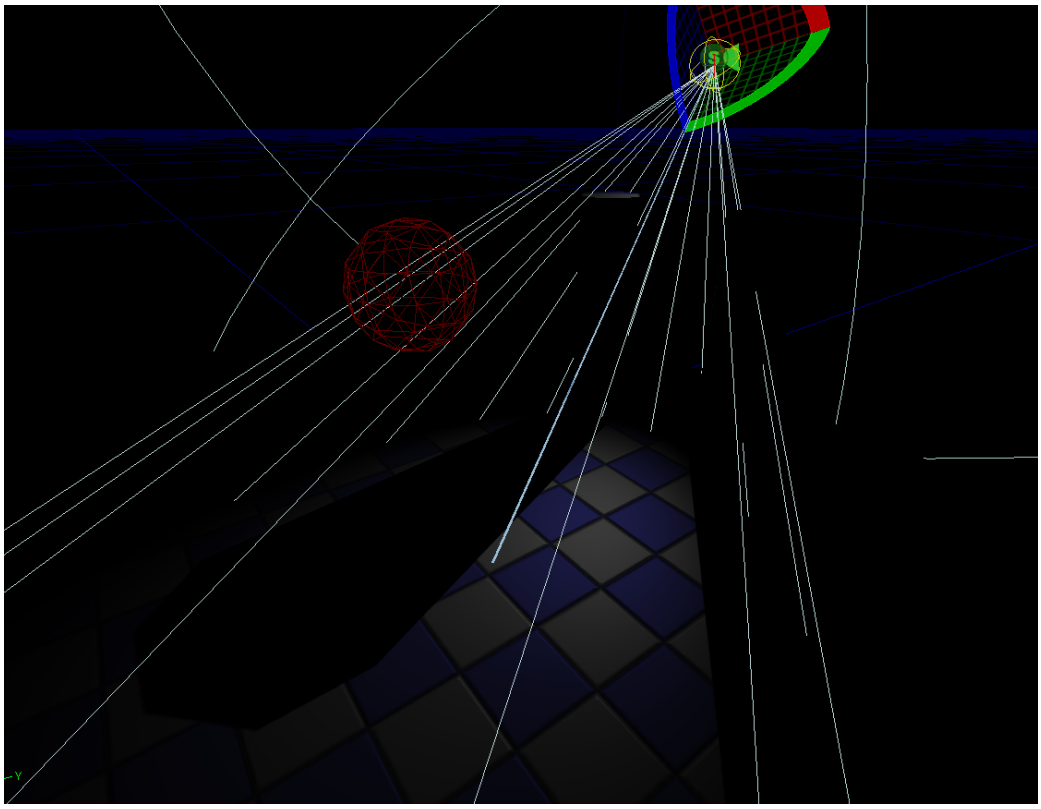


Figura 27

Las variantes *Toggleable* definen la luz para que pueda responder a un interruptor (via *Kismet*) y encenderse o apagarse, y las *movable* definen una luz que podrá moverse durante el juego en tiempo real (como una linterna unida al personaje).
Las precedidas por *Dominant*, quiere decir que su ámbito de iluminación se aplica a la totalidad del mapa.

Para poner una luz en nuestro mapa, primero la seleccionamos en *ActorClasses*, y después hacemos clic derecho donde nos interese en el mapa y seleccionamos *Add*... mas la opción de iluminación deseada.

Una vez colocada la luz, podemos editarla ampliamente accediendo a sus **propiedades** pulsando F4 (igual que con cualquier objeto en UnrealEditor). Ahí podremos configurar aspectos de todo tipo, como la función en la que se disipa la luz (*exponential falloff*), el color, la proyección de sombras, si queremos que dibuje rayos que pasen entre los objetos…

Para acabar, es importante anotar que cada vez que hacemos una modificación en la geometría o iluminación del mapa, tenemos que reconstruirlo. Haremos clic en este botón que representa un cubo, para construir la geometría, o bien clic en la bombilla, para reconstruir la luz, según la figura 28.

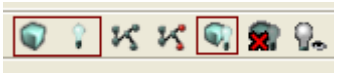


Figura 28

Esto puede llevar varios minutos, dependiendo de la complejidad de la escena, pero es necesario hacerlo para tener una vista real de lo que se verá en el juego.

Volúmenes especiales: *LightMassImportanceVolume* y *PostProcessVolume*

Los *brushes* 3D con los que trabajamos tienen mas usos aparte del de crear modelos para construir. Con los Volúmenes especiales, podemos dotar un área del mapeado de unas propiedades especiales. Por ejemplo, si hacemos una piscina, podemos poner un *brush* que coincida con el área de esta piscina y con él crear volúmenes especiales, como por ejemplo, uno volumen de postprocesado que haga que se vea borroso cuando el jugador se meta dentro de la piscina. Y no sólo hay aspectos visuales, también

podemos ponerle otro volumen que haga que cuando el jugador está dentro de la piscina, no caiga según lo dicta la gravedad, si no que caiga poco a poco, o incluso que flote. Los volúmenes ofrecen una profundidad de desarrollo enorme, y tal sólo explicaré dos de los más importantes.

LightmassImportanceVolume es un volumen aplicado a un área específica con el fin de mejorar considerablemente la calidad de iluminación. Lo que hace es definir el área en la que los fotones de la luz de los objetos se puede emitir, y simula una luz indirecta. (también conlleva un aumento en el tiempo de reconstrucción del mapa)

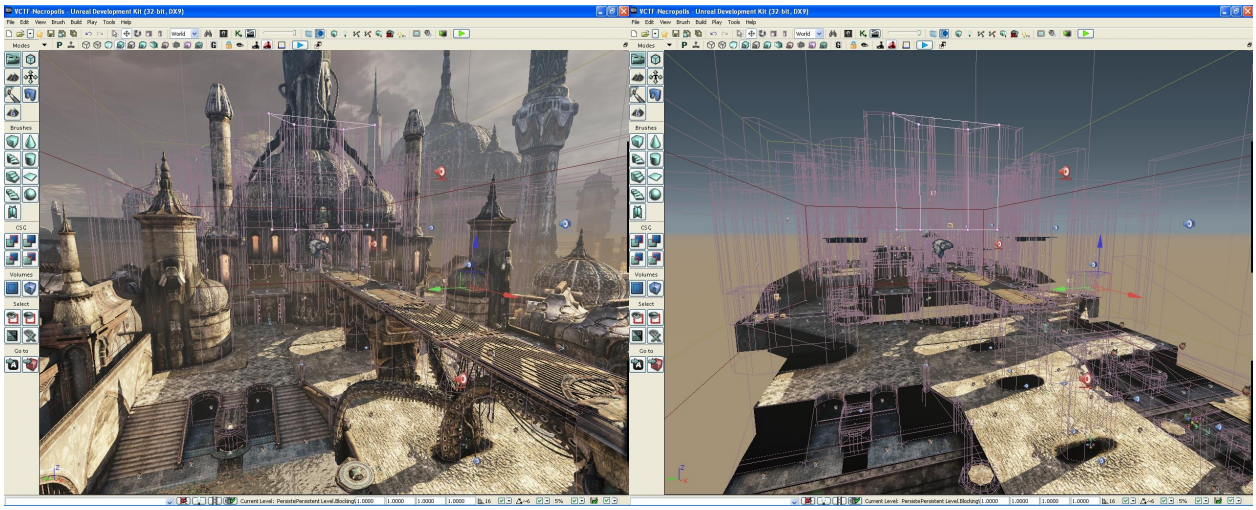
PostProcessVolume: Crea un área en la cual podremos realizar efectos visuales de postproceso, osea, cuando el jugador esté situedo dentro de ella, podremos hacer que vea la escena con efectos como correción de color, distorsión del movimiento *(motion blur)*, desenfoque óptico, etc.

Todo mapa profesional pasa por usar estos dos tipos de volúmenes, y utilizarlos es algo habitual en Unreal.

IMPORTAR MODELOS AL EDITOR UDK

Etapa fundamental en la creación de niveles, es la de reunir todos aquellos "bienes" (*assets* es el término usado en el UDK), para su implantación en el juego. El Editor Unreal nos da un entorno de modelado 3D que es muy básico, y en cierto modo bastante pobre. Hace un buen papel creando **estructuras fundamentales** para el nivel, simplemente techos, paredes, vigas, columnas, y escaleras sencillas. Pero de ninguna manera podemos esperar crear elementos artísticos, decorativos, o estructuras arquitectónicas detalladas con esta herramienta.

Entran en juego los programas de 3D externos, como el 3D Studio, Maya, COLADA o Blender. Es más, los artistas de videojuegos profesionales usarán lo mínimo posible las herramientas 3D que ofrece el Editor Unreal, y lo usarán más bien para componer el nivel, y los elementos que en el entran. Se podría estimar que en un mapa hecho por un profesional, el 90% de las superficies está formado por modelados estáticos creados por un programa externo, y el resto son hechas con el Editor Unreal (figura 29). Podemos comprobarlo nosotros mismos, cargando un mapa de los que vienen de muestra en el UDK, y en el modo de vista libre, presionar 'W', lo cual oculta/muestra todos los modelos estáticos del mapeado. Véanse como ejemplo estas imágenes:



Con modelos estáticos **Sin modelos estáticos**

Figura 29

Así pues, para el propósito de nuestro proyecto, si queremos un mínimo de detalle y buen acabado en nuestro edificio virtual, vamos a necesitar que muebles, escaleras detalles de la fachada, y otros muchos modelos se hagan por artistas 3D experimentados.

Importar modelados 3D a Unreal

Partiremos de la base de que se tienen conocimientos de modelado 3D, y de que disponemos de objetos 3D creados por nosotros o bien descargados de algún sitio Web especializado. No se va a entrar en detalles de cómo se modela. Además cada programa 3D utiliza interfaz y opciones propias.
Hay que tener en cuenta qué programa se ha utilizado para crear nuestros modelos. Para el entorno Unreal, el más aconsejado y utilizado por profesionales es el 3ds Max.

Exportando con 3ds Max:
Es el método más sencillo, ya que 3ds Max exporta directamente los modelos al **formato ASCII Scene Export (ASE), que es el formato aceptado por el Editor Unreal.** Seleccionando el modelo a exportar, vamos al menú y elegimos *Export Selected* Elegimos nombre y ubicación del archivo y aceptamos. A continuación aparecerá una ventana. Nos aseguramos de que las siguientes opciones estén marcadas: *Mesh Definition, Materials, Mesh Normals, Mapping Coordinates, Geometric*. Ninguna otra opción necesita ser modificada. Guardamos, y obtendremos así el nuevo archivo **.ASE**.
Solo nos queda un sencillo paso para importar el modelo. Una vez abierto el UDK, vamos al **Navegador de Contenido**, seleccionamos la opción *Import*, y seleccionamos el archivo exportado. Al ser un archivo .ASE, el nuevo objeto se creará como un modelo, nosotros debemos elegir si se trata de un modelo estático o esquelético (modelos esqueléticos son aquellos dotados de un esqueleto que servirá a modo de articulación ya que dicho modelo puede estar animado). Pasaremos por la ventana de asignación de paquete, en el que organizaremos qué paquete va a contener tal modelo y lo podremos agrupar. Acto seguido dispondremos de nuestro modelo listo para ser introducido en nuestro mapeado. **[5]**

Exportando con Blender:

El proceso que he elegido en un principio para mis pruebas, es el de la exportación por Blender, ya que es gratis y no dispongo del 3ds Max. Además es incesante explicar cómo hacer el proceso desde otras plataformas.

Existe un problema inicial con Blender, y es que no es capaz de exportar modelos al formato .ASE. Sin embargo hay alternativas a las que podremos recurrir.

Descubrí que Blender es capaz de exportar modelos al formato COLLADA (.DAE), y que resulta ser uno de los formatos aceptados por el Editor Unreal. Sin embargo, poco tardé en darme cuenta que los resultados no son siempre muy aceptables.

En primer lugar la textura o imagen perteneciente al objeto simplemente desaparece o se sustituye por una azulada. En otras ocasiones el objeto se deforma con respecto a uno o varios ejes, alterándose la proporción tal cual la tenía en el original

Así pues, el formato DAE no me parece el más apropiado para importar modelos con textura, pero sí tal vez para importar rápidamente modelos sencillos que se puedan utilizar como objeto simple en el mapa, aún estando por resolverse el motivo de que desaparezcan las texturas.

Para exportar un modelo hecho en Blender correctamente, necesitaremos servirnos de un **Script**. Para la versión actual de Blender (v2.58), existe un script llamado *ASE_Export_vMC-1-1.py* y que se puede descargar desde su enlace **[6]**. De todas formas se incluirá el contenido del fichero en el anexo por si fuese necesario. La instalación es sencilla; una vez en Blender, con el proyecto actual abierto, vamos a *file, user preferentes, install addon*; aparecerá un navegador. Buscamos y seleccionamos el archivo *ASE_Export_vMC-1-1.py*. Una vez instalado, tenemos que habilitarlo en la pestaña *addois*. Hecho esto, ahora podremos ver que en las opciones de exportación aparece la forma *ASCII Scene Export*

En la figura 30 podemos ver una muestra del script en Blender, cargado mediante la función *text editor*. A la izquierda un modelo a importar, a la derecha una ventana de texto con el script.

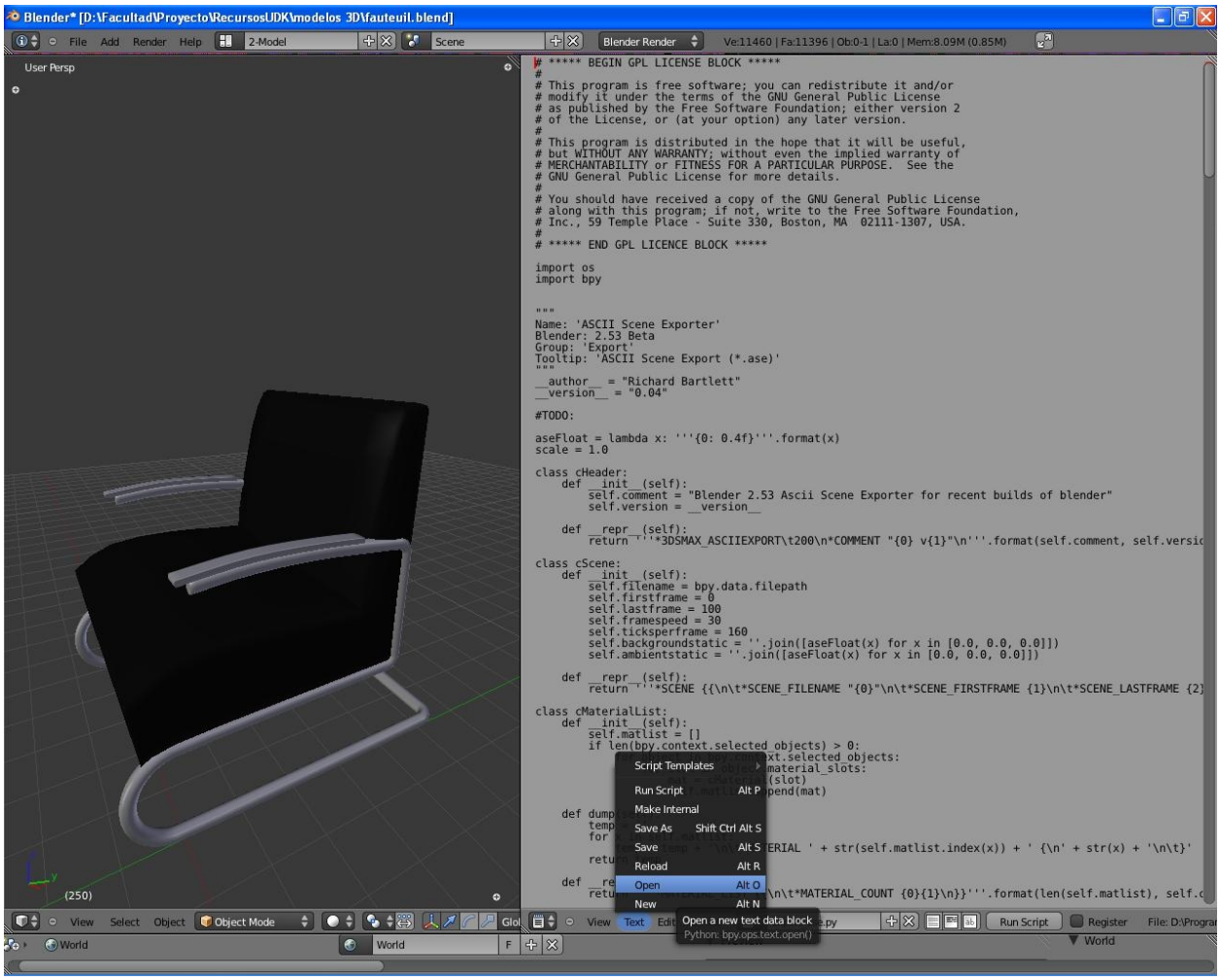


Figura 30

Importar al UDK se hace de forma análoga a como se describe anteriormente en el apartado de exportando con 3ds Max.

Importante: Antes de proceder a la exportación, es aconsejable aumentar considerablemente el tamaño del objeto (unas 10 veces). El motivo de esto es que Blender utiliza unas dimensiones para los objetos mucho menores que las que se usan para representar en el Editor Unreal, por lo que si exportamos un objeto desde Blender aparecerá en el Editor Unreal muy pequeño. Esto no supone un inconveniente grave ya que en el propio Editor Unreal se puede redimensionar, no obstante se aconseja hacerlo antes de la exportación para la comodidad de la reusabilidad de dicho modelo

También he descubierto que actualización tras actualización, el script puede fallar, y su instalación también puede cambiar. Hasta el punto que para cada versión de Blender existe una versión del script. Es por todos estos inconvenientes por lo que veo desaconsejable el uso de Blender como herramienta de modelado para modelos UDK, mientras no se implemente **oficialmente** una herramienta de exportación.

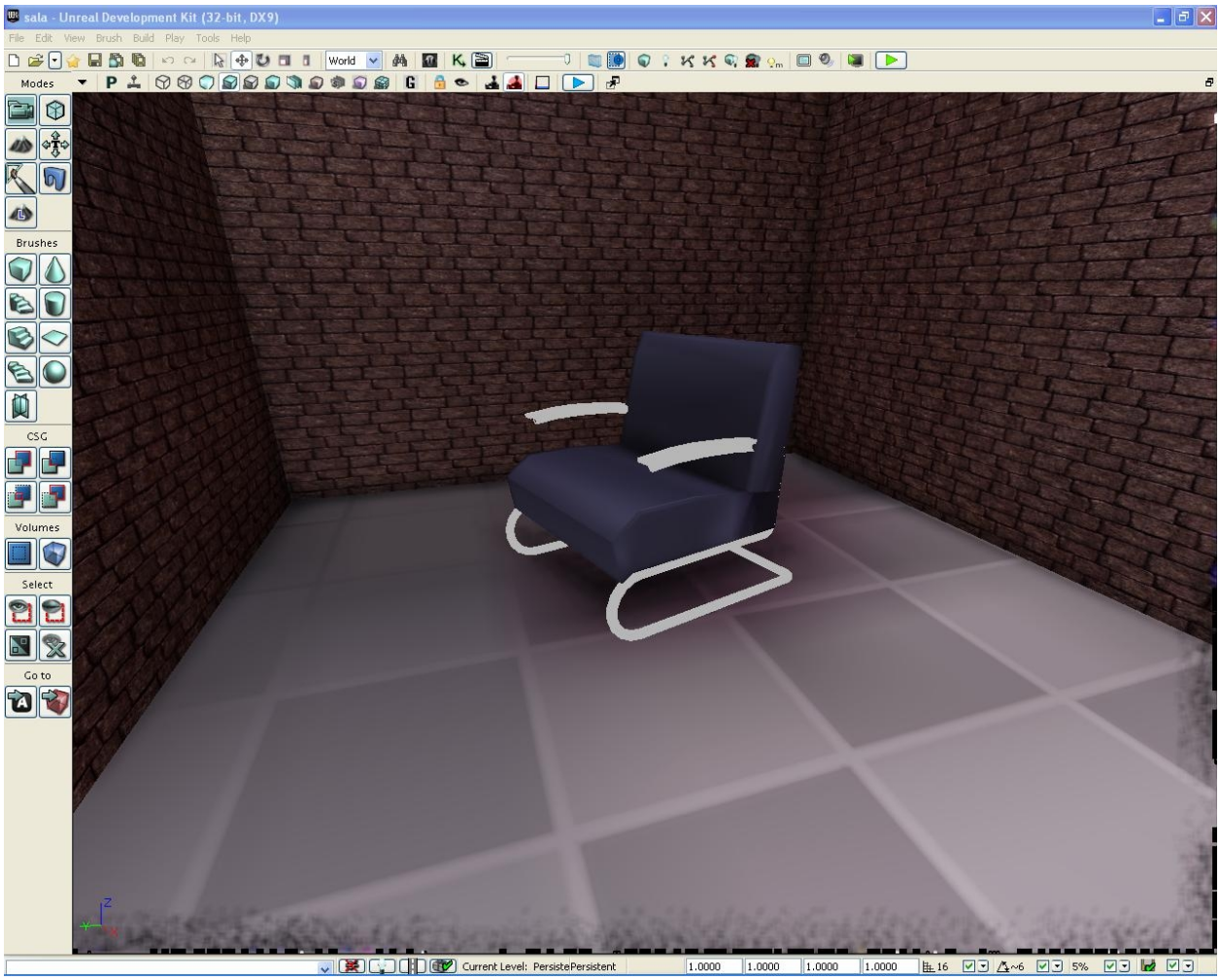


Figura 31

En la figura 31 podemos ver el mismo modelo importado a Unreal e integrado en un mapa.

Definir las colisiones: Una vez hemos importado nuestro modelo, nos daremos cuenta de que en el navegador de contenido, aparece una indicación con letras amarillas que dice: *NO COLLISION MODEL*. Esto significa que el objeto no tiene definido un modelo de colisiones y que una vez puesto en el mapa podremos atravesarlo sin que represente ningún obstáculo. Si queremos que esto no ocurra, existen varias formas de establecer un modelo de colisiones en nuestro objeto. Aun que se puede hacer directamente durante el modelado del objeto y antes de su exportación, voy a explicar la manera mas sencilla de hacerlo.

Desde el navegador de contenidos, editamos el objeto haciendo doble clic, de modo que se abrirá el Unreal Static Mesh Editor. Iremos al menú *collision* y elegiremos *auto convex collision*. Aceptaremos la ventana siguiente. Luego nos aseguramos, a la derecha que estén marcadas las opciones de *simple, line* y *rigid body collision*.

Importar modelos esqueléticos

Los modelos esqueléticos son muy importantes dentro de un juego, ya que todo modelo que necesite una articulación y una animación para un movimiento requiere un esqueleto. Sirva de ejemplo un personaje que camina por la escena. Como es obvio, querremos que tenga una animación adecuada de sus brazos y piernas que le dote de realismo. Otros ejemplos serían las ruedas de un coche, un brazo robótico, etc.

La exportación de modelos esqueléticos no es tan directa como la exportación de modelos estáticos explicada anteriormente. El UKD facilita una serie de plugins llamados **ActorX**, que sirven para esto.

Estos plugins se instalan en editores 3D ajenos, y actualmente existen para **3DStudio Max** y **Maya**. Los plugins se encuentran en la carpeta \Binaries\ActorX, y existen múltiples versiones para las distintas actualizaciones que podamos estar usando.

Puede ser interesante explicar también la existencia de los plugins **FaceFX,** que se encuentran en la carpeta \Binaries\FaceFXPlugins, y cuyo cometido es el de permitir exportaciones de animaciones faciales para usarlas en Unreal.

EDITOR DE TERRENO

Ésta útil aplicación nos facilitará las herramientas necesarias para crear paisajes naturales que tengan montañas o desniveles de terreno y de aspecto arbitrario. (fig.31 y 32).

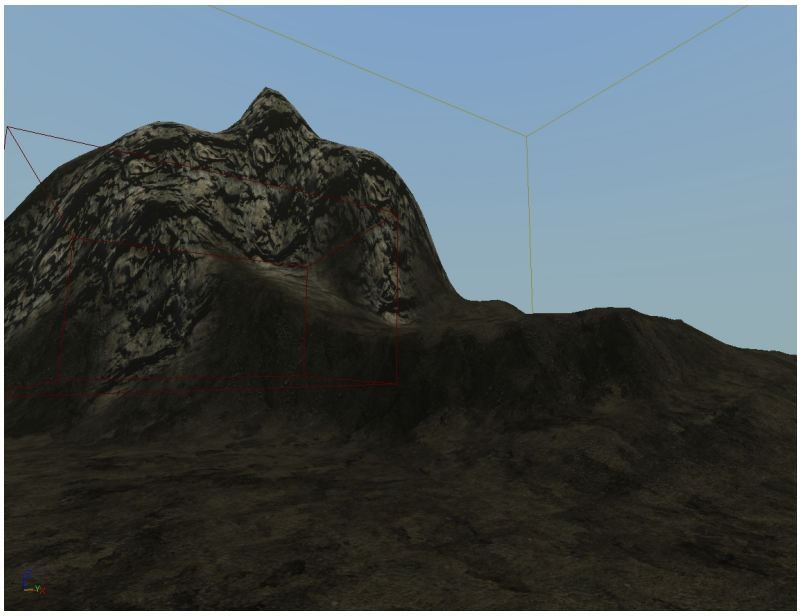


Figura 31: Muestra de terreno creado en *TerrainEdit*

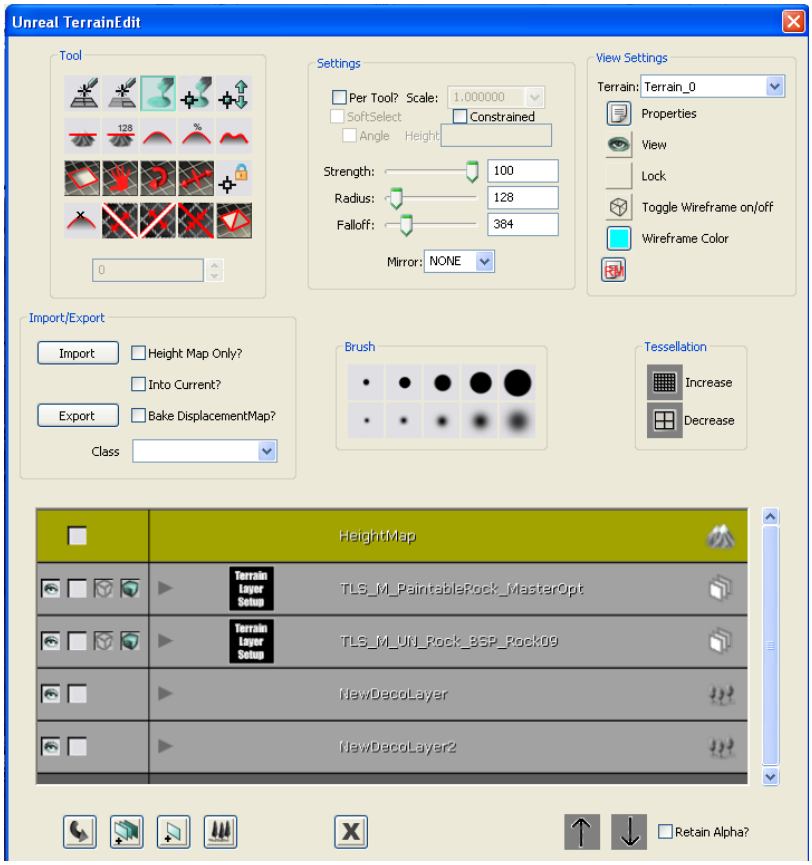


Figura 32

Funciona a partir de capas, y cuenta con tres tipos de ellas: *Heigh Map,* con la que podemos definir relieve sobre una plano que servirá de superficie, *Terrain Material*, con la que podremos pintar el relieve para dotarle de un material, y *Deco Layer,* que es una curiosa aplicación que nos permitirá crear agrupamientos de ciertos Modelos estáticos que definamos, ideal para crear bosques, como se puede ver en la figura 33.

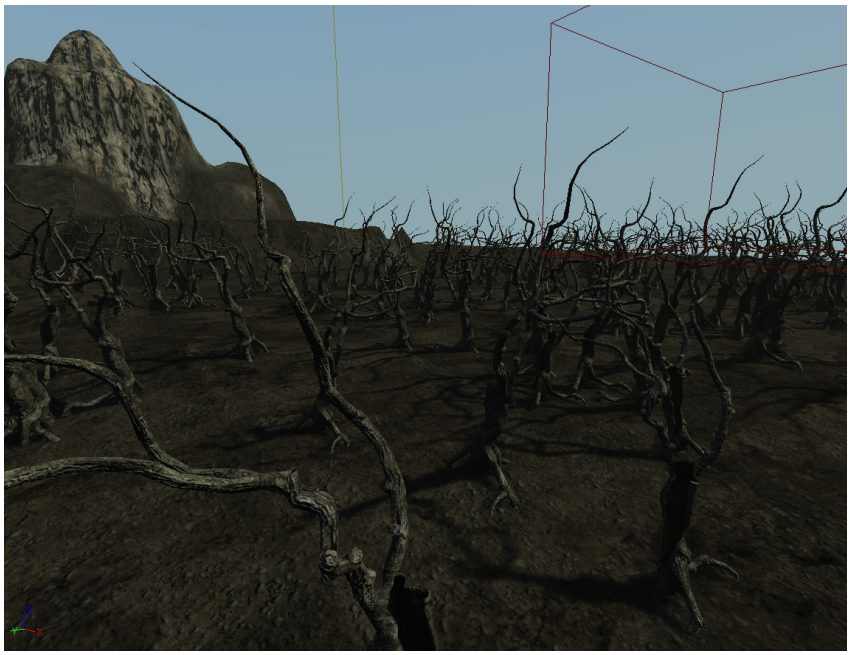


Figura 33: Bosque a partir del modelo: S_UN_Tree_SN_BurnTree01

Lo que hacen las *Deco Layer* es copiar un mismo modelo de forma aleatoriamente dispersa, e incluyendo también un componente aleatorio que diferencia la rotación y el tamaño de cada modelo individual del de los demás vecinos.

UNREAL KISMET

Kismet es una potente y flexible herramienta que permite dotar al mapa de un flujo complejo de jugabilidad, **sin necesidad de programar scripts**. Esta herramienta es una de las causas de que UDK sea accesible a todo tipo de creadores y no sólo a programadores. Kismet se basa en la creación de secuencias, las cuales están compuestas de Objetos de Secuencia. En la figura 34 podemos ver la interfaz de *Kismet*. Con *Kismet* podemos programar acciones como que se abra una puerta, que se activen efectos especiales, acciones sobre el jugador, y por supuesto activar acciones cinemáticas.

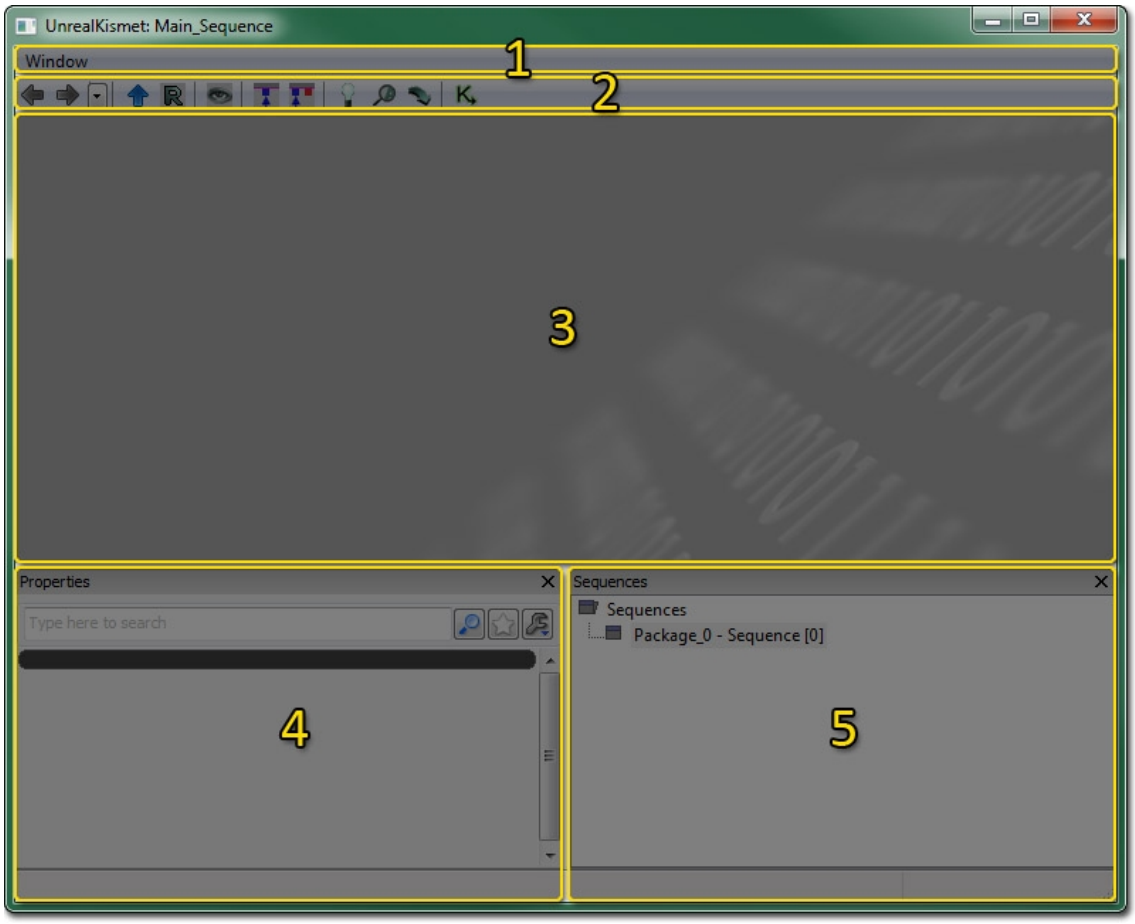


Figura 34

1: Barra de menú

2: Barra de herramientas

3: Panel gráfico (aquí se construyen las secuencias)

4: Panel de propiedades de los objetos

5: Panel de secuencias

Los tipos de objetos que existen son:

Eventos: Capturan algo que sucede durante el juego. Son la entrada de la secuencia, y habitualmente responden a un actor en el nivel. (fig 35).

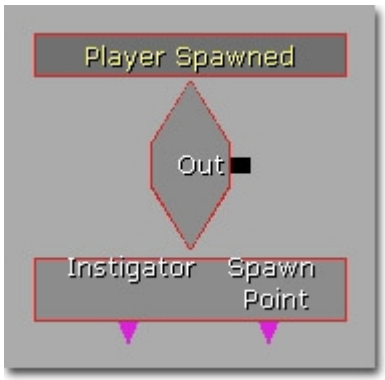


Figura 35

Acciones: Estos objetos llevan a cabo una acción sobre el nivel. Se activan con un disparador en la entrada a la izquierda, y tienen también una salida a la derecha cuando la acción se ha completado. (fig 36).

Figura 36

Condiciones: No afectan al nivel, pero ofrecen todo tipo de operaciones para programar el control de flujo de la secuencia. (fig 37).

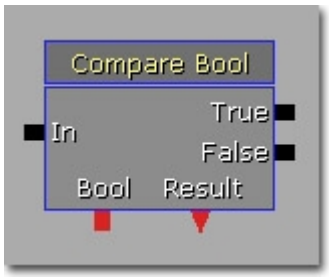


Figura 37

Variables: Almacena información de un tipo particular. Pueden ser referencias a objetos presentes en el nivel, y al propio jugador. (fig 38).

Figura 38

Ilustraremos un ejemplo básico de secuencia que se encarga de encender una luz.

Como se mencionó en el apartado de luces, existe un tipo de objeto que se corresponde con *ToggeableLights*, es decir, luces que pueden encenderse y apagarse. Para llevar a cabo esto debemos colocar una de estas luces en el mapa, y también un actor ***trigger***. Este tipo de actor es muy importante ya que se utiliza ampliamente para *disparar* acciones que serán manejadas en *Kismet*. Un *trigger* se representa en el mapa con el icono de la figura 39.

Figura 39

A efectos prácticos, un *trigger* es un volumen abstracto, de forma y dimensiones modificables, que tiene la capacidad de generar una respuesta cuando algún objeto específico entra en contacto con él, o cuando algún jugador **lo usa** (en Unreal, un jugador *usa* algo cuando centra su visión en este objeto y presiona la tecla "e"). Así es como haremos que funcione el interruptor de luz.

Una vez emplazados estos dos objetos, vamos a K*ismet* y configuramos la secuencia mostrada en la figura 40.

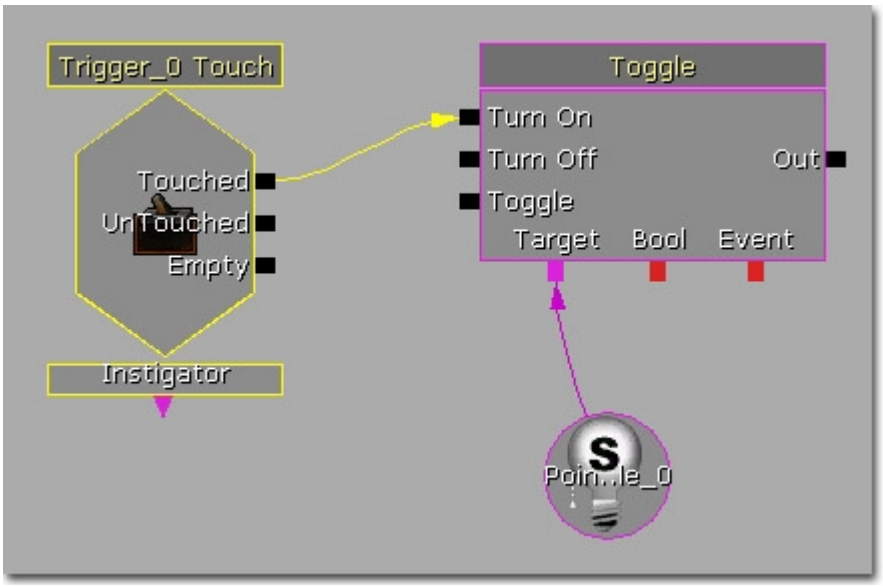


Figura 40

En la que emplazamos un objeto *trigger* que estará asociado al *trigger* colocado en el mapa. De la salida *touched* (o sea, cuando es tocado), sacamos una flecha que entrará en un objeto *toggle*. A la entrada *target* en *toggle*, colocamos una variable que consiste en el actor de luz que hemos colocado en el mapa. Así, cada vez que alguien entre en contacto con el *trigger*, la luz se encenderá o se apagará, según su estado actual.

Esto es un ejemplo muy básico, pero tenemos infinitas posibilidades, con cientos de objetos diferentes, con los que podremos incluso crear funcionalidades enteras, como hacer inventarios de objetos, controlar datos mediante operaciones, usar vectores… Y si además creamos los scripts adecuados, podemos incluso crear nuestros propios objetos *Kismet*.

Toda la documentación necesaria para tener una introducción a *Kismet* se encuentra en red UDN **[7]** y su consulta es muy aconsejable.

UNREAL MATINÉE

Matinée cumple la función de **animar** cualquier actor presente en el mapa mediante el uso de **fotogramas clave**. Así pues, funciona como cualquier entorno de animación 3D. Para quien no esté introducido en este tipo de programas, la animación por fotogramas clave consiste en definir las coordenadas de posición para un punto temporal concreto, y crear unas nuevas coordenadas en otro punto temporal. Mediante interpolación (que en Matinée puede ser curva, lineal o constante) se crea una traslación del actor con la progresión temporal. La posición del actor no es lo único que puede animarse, si no que puede animarse cualquier propiedad del actor, como el tamaño, en el caso de modelos estáticos, o el brillo en el caso de un foco de luz, o la densidad de una niebla, etc.
Por supuesto, *Matinée* es lo que tenemos que usar si queremos crear **secuencias cinemáticas**, es decir, secuencias de cámara no asociadas al jugador, que ocurren en algún momento durante el juego. Por ejemplo, hacer que la cámara sobrevuele previamente el mapeado para que el usuario pueda verlo todo antes de jugar.

Matinée está integrado en *Kismet*, y para hacer uso de él tenemos que crear un objeto *Matinée.* (figura 41).

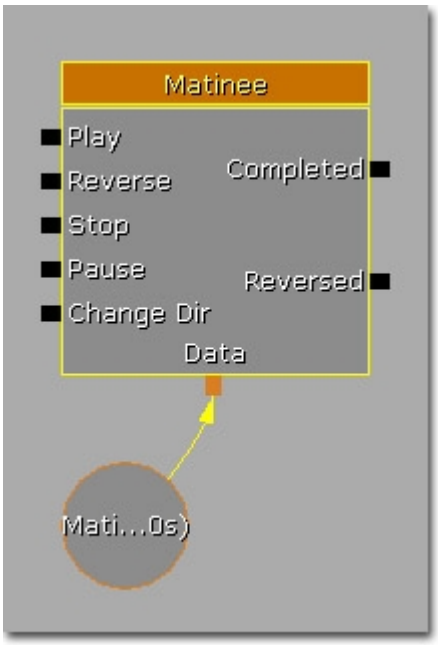


Figura 41

En las entradas podemos ver que el objeto se activa por una acción externa, que determina si la animación se reproduce, se detiene, se reproduce a la inversa, se pausa, o si cambia el sentido de la reproducción. Cuando la animación se completa, se puede utilizar una salida que crea la acción *completado*.

Haciendo doble clic sobre este objeto se abrirá la interfaz de *Matinée.*

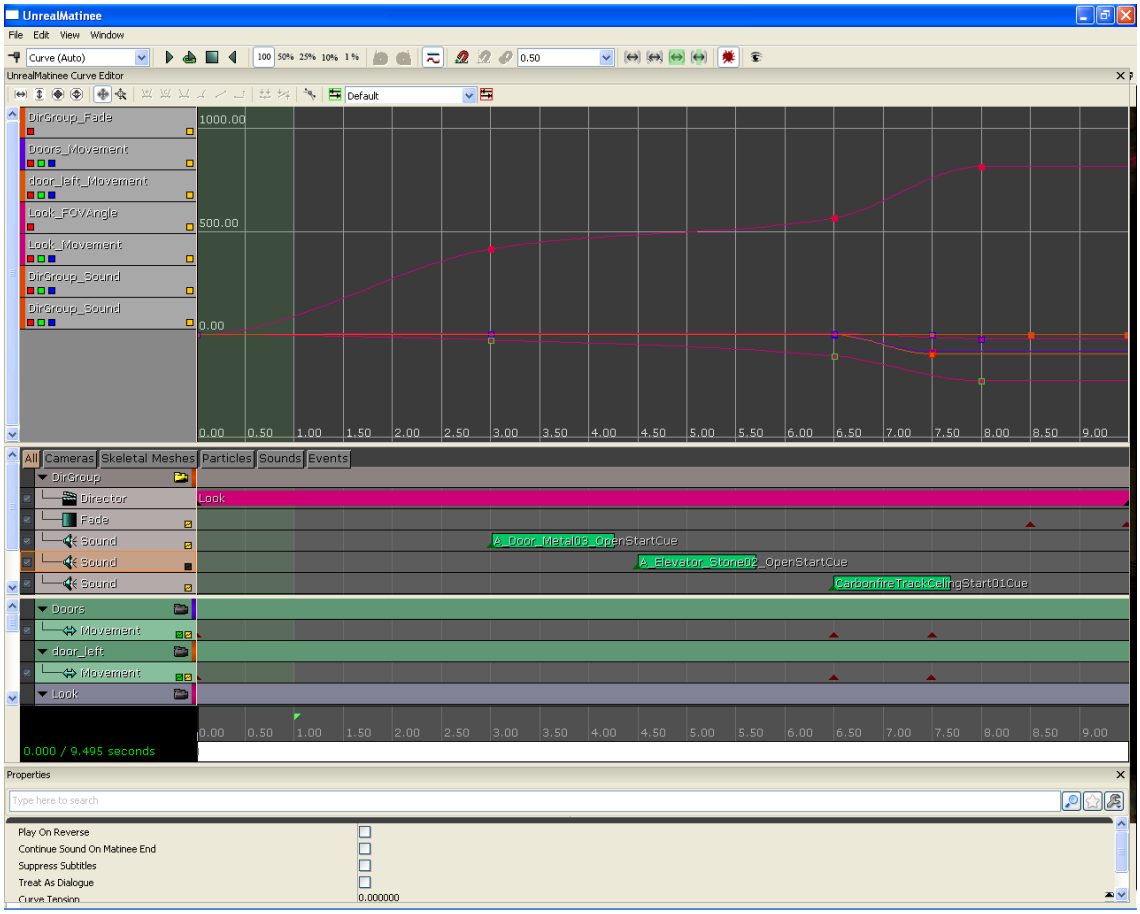


Figura 42

Aquellos familiarizados con la animación por fotogramas clave, verán que disponemos de un editor de curvas, en la parte superior, y de un panel de barras de tiempo (figura 42). En un mismo Matinée se pueden representar varias secuencias, cada una de ellas hace referencia a un actor ubicado en el mapa.

Nota importante: Para introducir un modelo estático en el mapa que sea animable mediante *Matinée*, no debemos insertarlo como un *static mesh*, sino como un *Interp Actor.*

Cabe añadir que *Matinée* puede manejar todo tipo de efectos cinematográficos, mediante la función *Director*, como fundidos, desenfoques, correcciones de color… y que además puede insertar música y sonidos ambientales a gusto del desarrollador.
Una guía a fondo para usar Matinée puede encontrarse en la red UDN. **[8]**

MENÚ Y HUD

En este capítulo se detallará lo necesario para crear elementos gráficos en pantalla durante el juego, que darán soporte a nuestro HUD o bien con los que podremos crear menús e interfaces de juego.
Un HUD se utiliza para mostrar información en tiempo real por pantalla. Dependiendo del juego, el HUD puede ser muy simple o muy complejo. Pero para nuestra aplicación, si queremos que sea intuitiva, rápida y accesible, se aconseja hacerla lo más sencilla posible.

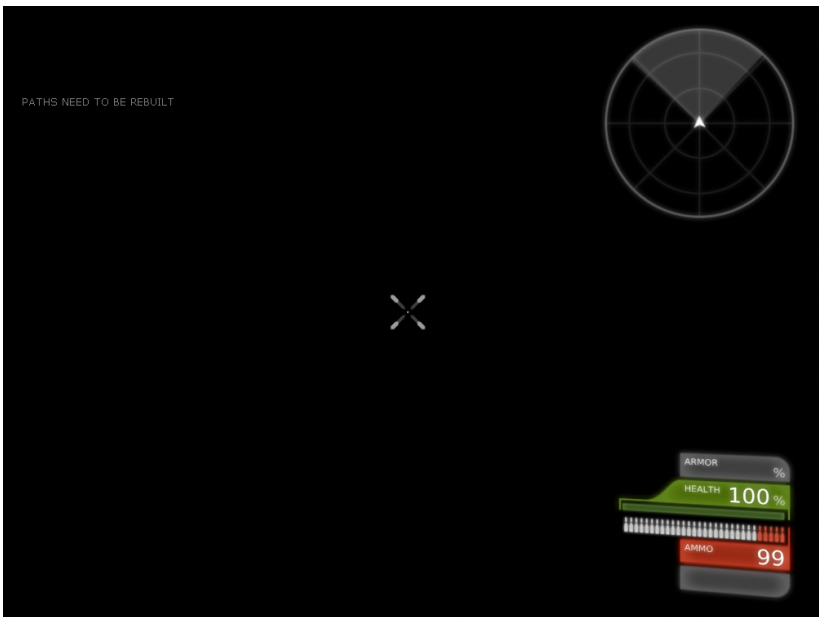


Figura 43

Por ejemplo, en la figura 43, vemos una pantalla de juego con HUD básico; en la esquina superior derecha hay dibujado un radar, en la inferior derecha un indicador de munición y de salud. En el centro, el punto de mira.

En Unreal hay dos formas de diseñar un HUD:

- Canvas: Se utiliza la clase Canvas para diseñar gráficos en pantalla.
- Scaleform GFx: Utiliza la aplicación Scaleform GFx, que es una extensión para Adobe Flash, para dibujar HUDs y menús. Es la más aconsejable y la que vamos a desarrollar en este texto.

Importante: Scaleform es una innovación dentro de Unreal que fue introducida en 2010. Hasta entonces, para crear interfaces se utilizaba un programa llamado **UIScene**, y que estaba integrado en el UnrealEditor. Actualmente, esta herramienta ha quedado obsoleta ya que parece ser que se ha contemplado que Scaleform supera con creces en calidad a UIScene. El Volumen 2 del libro **Mastering Unreal Technology** se basa en UIScene, en el capítulo 6: *Creating user interfaces*. Este capítulo es totalmente imprescindible en esta edición y debe ser ignorado.

Scaleform GFX

Instalación:

Scaleform requiere Adobe Flash CS4 o CS5 para su uso.

Lo primero que hay que hacer es instalar la extensión para nuestra versión de Adobe Flash. El paquete de instalación se encuentra en la carpeta \UDK\Binaries\GFx\CLIK Tools, y tiene por nombre: ***Scaleform CLIK.mxp***. Al ejecutarlo se abrirá una ventana similar a la de la figura 44:

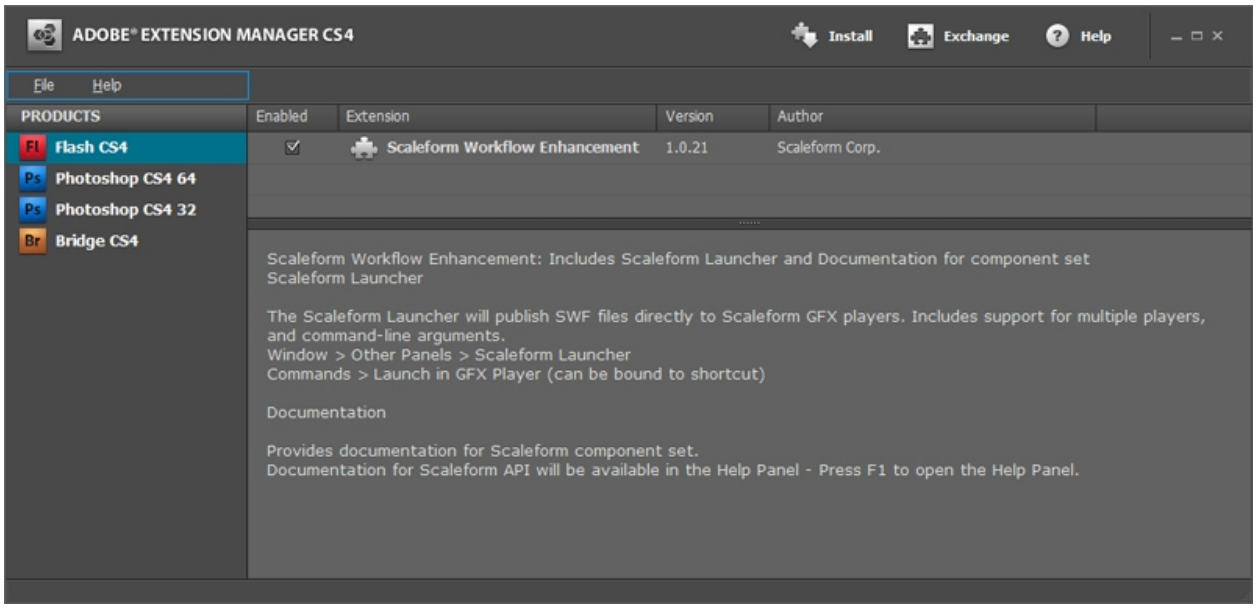


Figura 44

Tenemos que seleccionar nuestra versión de Flash en la columna *PRODUCTOS*, y activar la casilla correspondiente a la extensión Scaleform, y a continuación hacer clic en instalar. Se abrirá un explorador, en el que tenemos que seleccionar exactamente el

mismo archivo nombrado antes. Entonces se procederá a la instalación. Una vez hecho esto, ya podremos crear animaciones o gráficos Flash exportables a UDK.

Ejemplo de menú sencillo

Una vez lanzado Flash, creamos un nuevo documento de tipo **ActionScript 2.0** (Es probable que la versión 3.0 no sea soportada todavía. **[9]**)
Un documento Flash ActionScript tiene la peculiaridad de que en él podremos implementar funcionalidades mediante códigos de modo que nuestra animación Flash pueda manejar eventos, que le permitan comunicarse con un programa externo. Esto es lo que hace que Unreal pueda responder a nuestro manejo de la interfaz posteriormente. Se entenderá mejor conforme se desarrolle el ejemplo.

Cuando aparezca nuestro documento en blanco, será importante que en propiedades configuremos las dimensiones que va a tener. Ésto es importante por que se va a ver sobre nuestra pantalla del juego, y tiene que estar debidamente ajustado. En una pantalla típica LCD 5:4, las dimensiones serían 1280 x 1024, sin embargo, debemos contemplar la posibilidad de que el usuario esté ejecutando otras dimensiones de pantalla. Es cometido del desarrollador prevenir estos desajustes.
Otro factor importante son los FPS (*frames per second*), que definen la tasa de cuadros por segundo a la que se moverá nuestra animación Flash. Se recomienda usar un valor mínimo de 60, puesto que es la tasa habitual en la ejecución de un videojuego. Una tasa inferior podría dar una sensación de poca suavidad, sobre todo si vamos a implementar un cursor para el ratón. (figura 45).

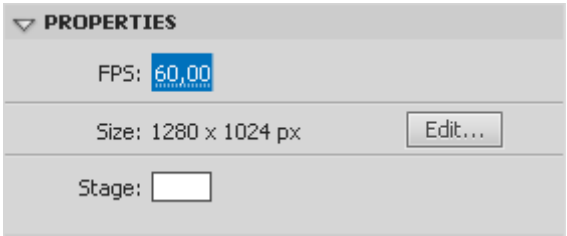


Figura 45

Nos queda realizar una configuración más. Vamos al menú *File, Publish Settings,* y configuramos la pestaña *Flash* tal cual se muestra en la figura 46.

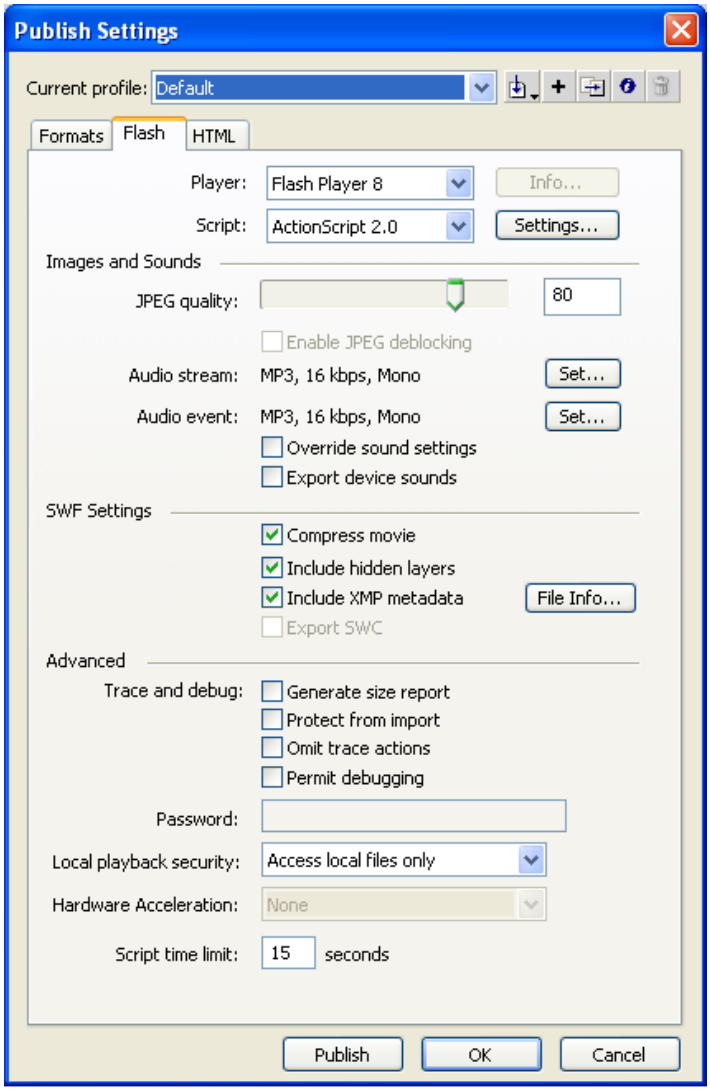


Figura 46

El motivo de seleccionar la versión Flash Player 8 es por incompatibilidad con versiones superiores. (Flash Player será el soporte de las animaciones que realicemos)

A continuación estaremos listos para comenzar a dibujar nuestro menú. Mediante las herramientas apropiadas, iremos creando en nuestra escena los elementos que deseemos. Si hace falta, podemos importar imágenes externas. La composición del ejemplo consta de varios textos de presentación, así como tres textos para nombrar los botones.
Además se ha importado una imagen, el logotipo de UDK, en formato PNG. En la figura 47 se puede ver el aspecto del menú.

Figura 47

Eso si, es importante hacer unos pasos cuando vayamos a usar imágenes externas en nuestra escena. Una vez importada la imagen, la seleccionamos en la librería, y abrimos la ventana *Bitmap Properties*. Aquí configuramos los parámetros según se muestra en la imagen. (figura 48) Es importante aclarar que hay que borrar la extensión en los campos correspondientes al nombre del archivo de imagen. Si no se hacen estos pasos, la imagen no se verá correctamente durante el juego.

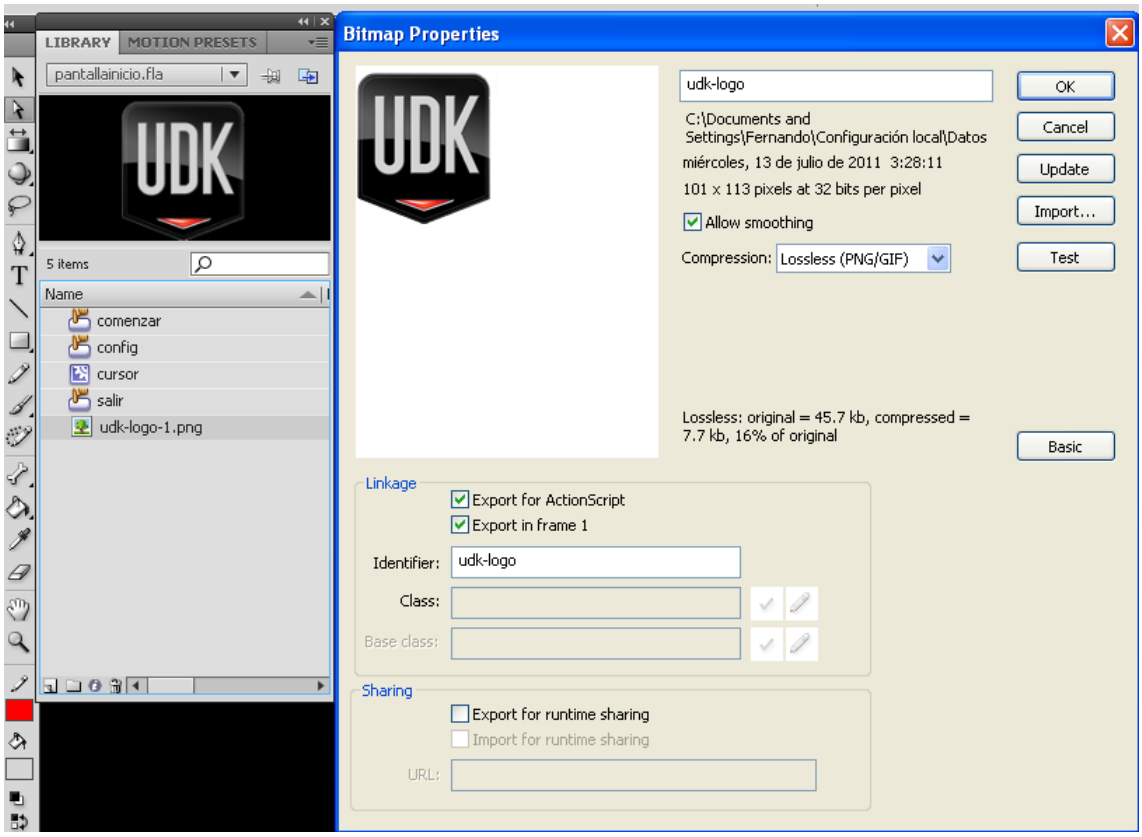


Figura 48

El fondo, aunque aparezca negro, en realidad no tiene definido color, será transparente, de modo que podamos ver el mapa de Unreal por detrás de nuestro menú.
Los botones serán las formas definidas a partir de rectángulos, las que tienen el borde amarillo.

Para que estas formas funcionen como botones, hay que definirlas como tales; para ello, seleccionamos la forma, y vamos al menú *Modidy, Convert to Symbol…*, seleccionamos como tipo: *Button*, y opcionalmente lo dotamos de un nombre. (figura 49).

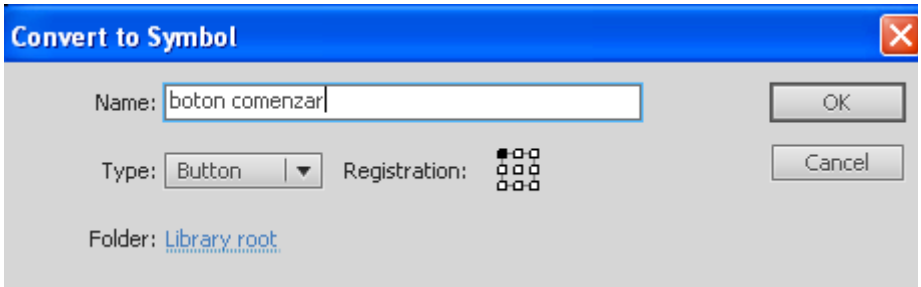


Figura 49

Ahora crearemos una nueva capa, en la que construiremos un cursor para poder manejarnos en el menú.

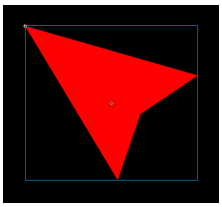


Figura 50

Se ha hecho con una forma sencilla como se puede comprobar en la figura 50.
Esta nueva forma también ha de ser convertida, pero en este caso no va a ser un botón, si no que va a ser del tipo *Movie Clip*. (figura 51).

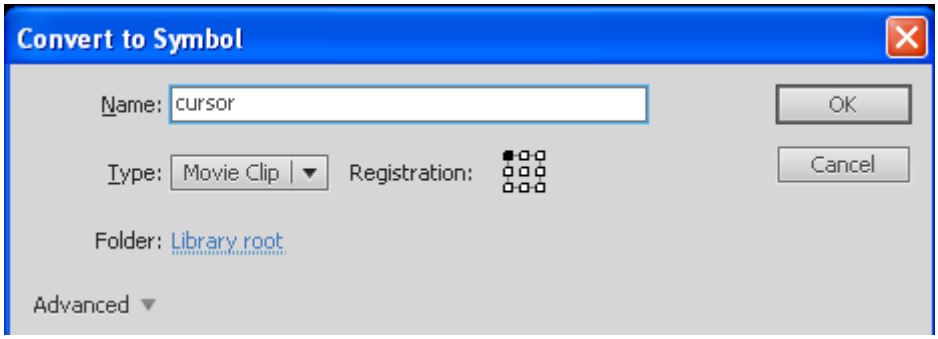


Figura 51

Ahora queda dar funcionalidad a este cursor. Aquí es donde vamos a utilizar *ActionScript.* Para ello, hacemos clic derecho en el cursor, y seleccionamos *actions*. Se abrirá un editor de *script*, tal cual se ve en la figura 52. Escribimos este código:

```
onClipEvent(enterFrame)
{
  _x = _root._xmouse;
  _y = _root._ymouse;
}
```

Lo que este simple *script* hace, es colocar nuestro cursor de forma que coincida con las coordenadas de posición del ratón.

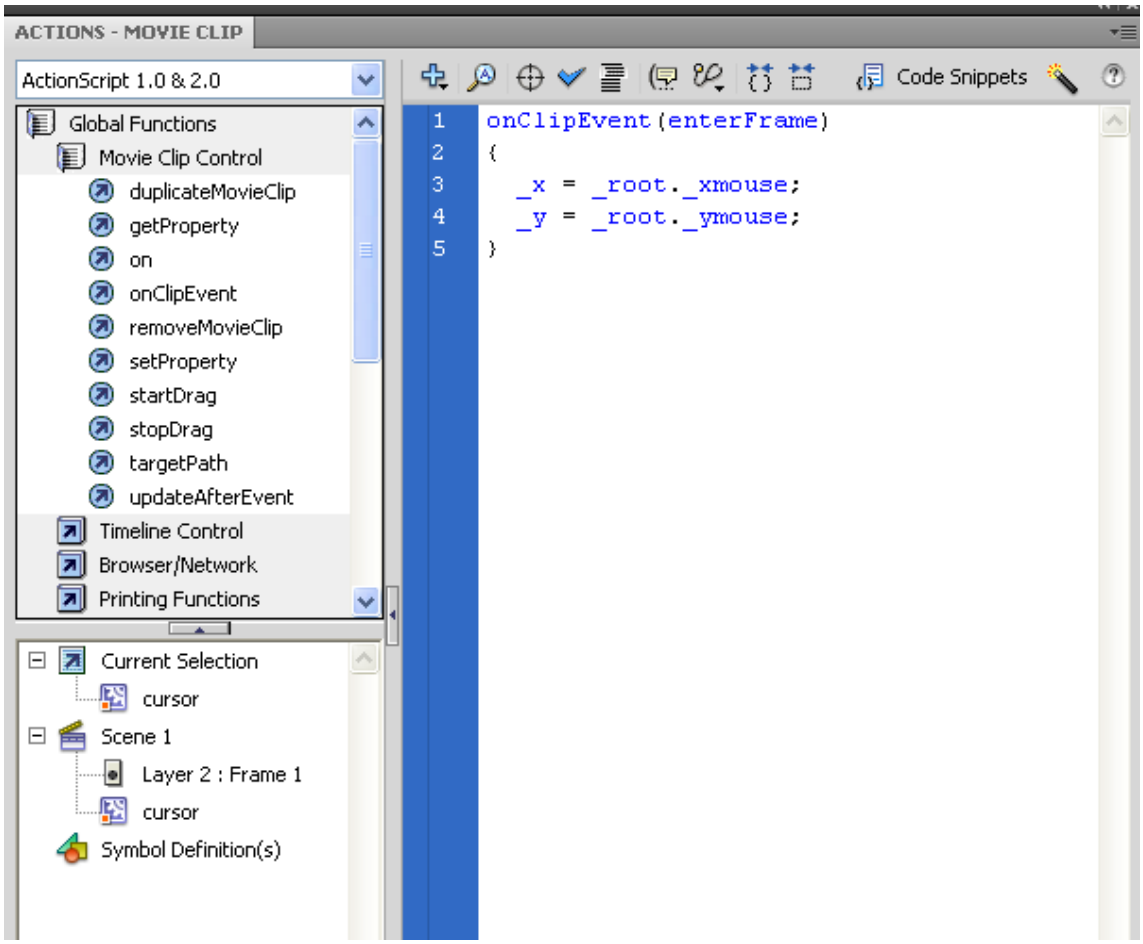


Figura 52

Por último, tenemos que hacer los scripts correspondientes para los botones. Para cada uno de los botones, respectivamente, usamos estos scripts:

```
on (press) {
     fscommand("comenzar");
     }

on (press) {
     fscommand("configuracion");
     }

on (press) {
     fscommand("salir");
     }
```

Estos comandos permiten que nuestro .sfw se comunique con Unreal, concretamente con Kismet. Lo único que hacen es lanzar una orden, que se traduce a un evento, que Kismet se ocupará de asociar a alguna acción.

Y hasta aquí; sólo nos queda ir a *File, Publish*, y publicar nuestra animación, que hará que se cree el archivo .sfw.

El siguiente paso es importarlo a nuestro mapa a través del *Content Browser*. Si es necesario, creamos un paquete nuevo para contenerlo. Si hemos incluido imágenes en nuestro sfw, se nos pedirá que localicemos la ruta de la imagen, y se cargará también.

Poniendo nuestro menú en nuestro mapa

Para hacer funcionar nuestro menú, tenemos que hacerlo a través de *Kismet.*
Necesitamos, fundamentalmente, un evento *Level Loaded*, para que nuestro menú aparezca desde el primer instante de ejecución, y una acción *Open GFx Movie*, que activará la animación sfw. El parámetro *Player Owner* tiene que ir conectado a todos los jugadores. Véase la figura 53.

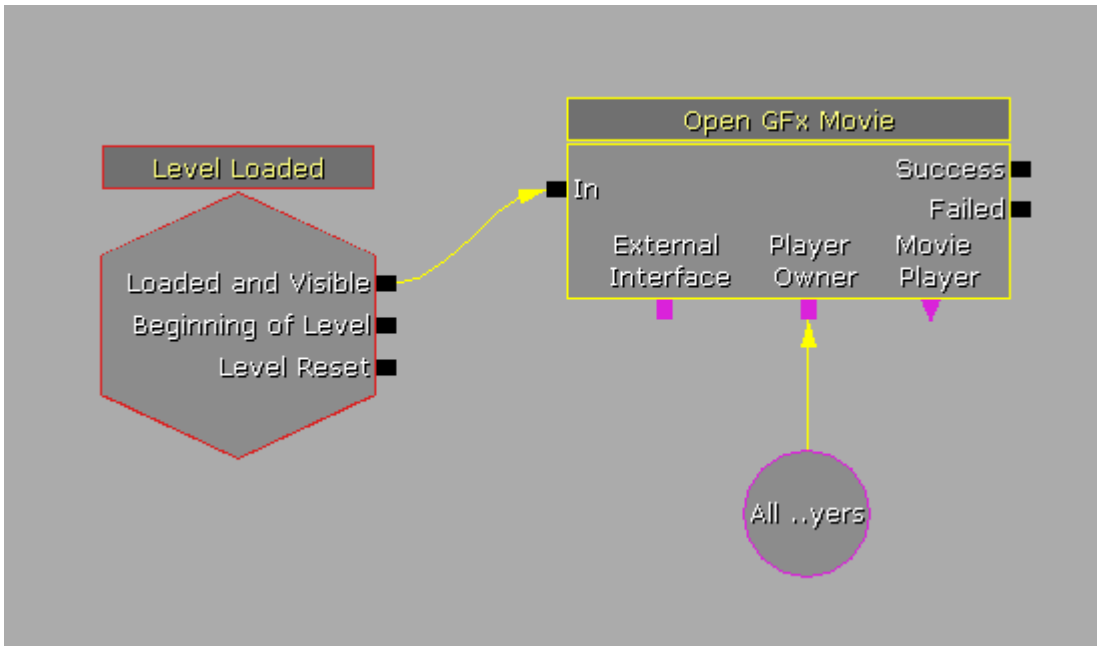


Figura 53

En las propiedades, seleccionamos nuestro archivo, en este caso se llama *pantallainicio.* (fig 54).

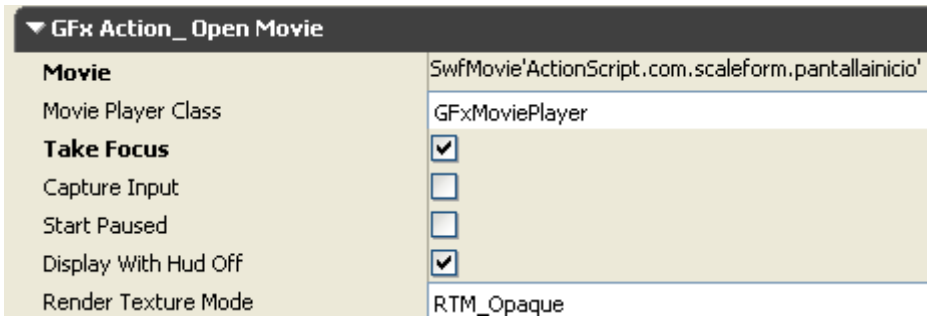


Figura 54

Esto es suficiente para mostrar el menú desde el momento que se lanza el mapa, como se ve en la figura 55.

Figura 55

Pero esto no es suficiente para que funcione correctamente. Los botones aún no tienen su funcionalidad. Además, el hecho de aparecer el menú en pantalla, no impide que el jugador pueda moverse y mirar libremente por el mapa, lo cual no resulta coherente, por que lo que nosotros deseamos de un menú, es que el jugador no pueda moverse ni hacer otra cosa que no sea usar el cursor para elegir las opciones.

Lo primero que hay que hacer es dar funciones a los botones. En este caso, haremos que cuando el jugador haga clic en *comenzar*, el menú desaparezca de la pantalla y pueda moverse libremente. Para hacer que se reconozcan los comandos de los botones, implementados anteriormente, se utiliza el evento *FsCommand.* En las propiedades de tal evento, definimos tal evento como en la figura 56. En este caso *comenzar.*

Figura 56

Para hacer que el jugador no pueda moverse mientras está el menú en pantalla, usaremos la acción *Toggle Cinematic Mode.* Véase la figura 57.

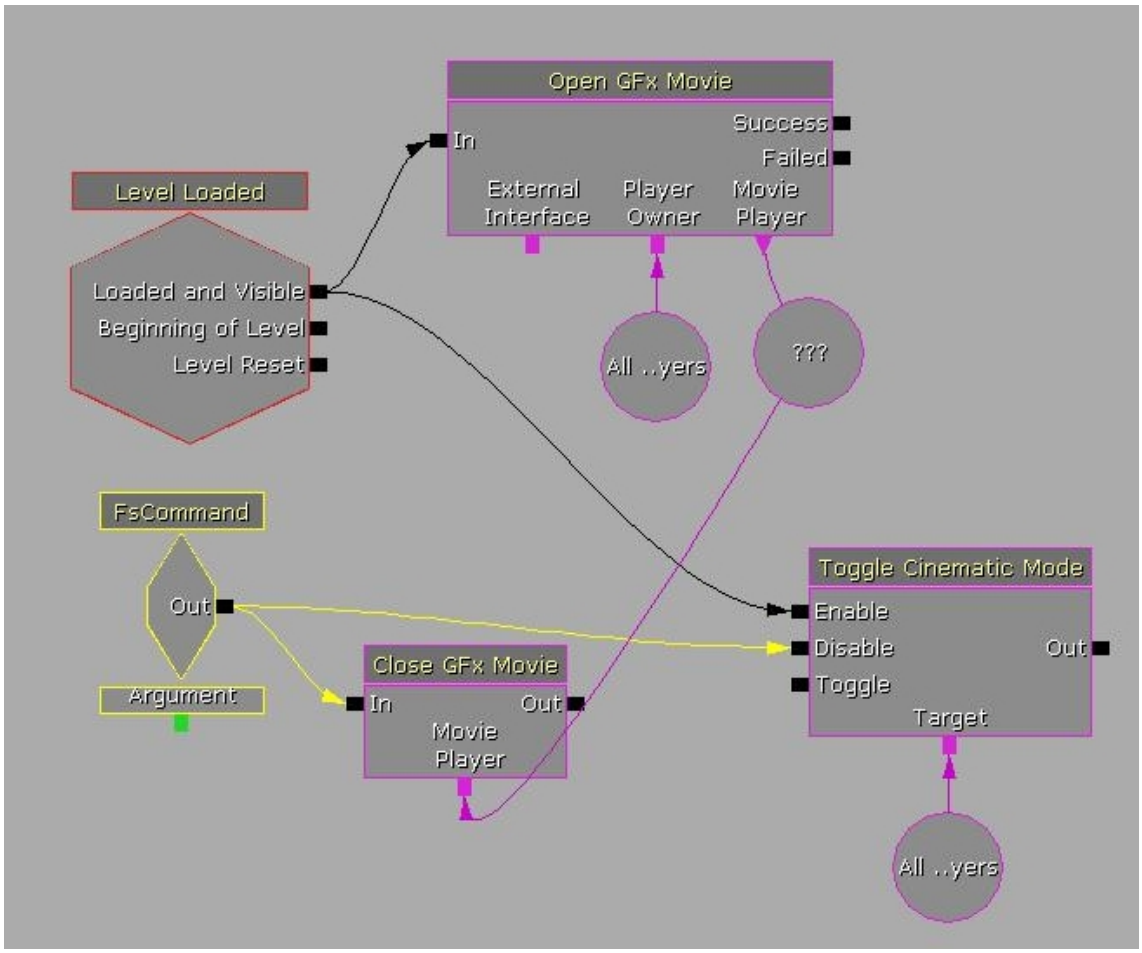


Figura 57

También haremos que cuando haga clic en *salir,* se cierre el juego (fig.58).

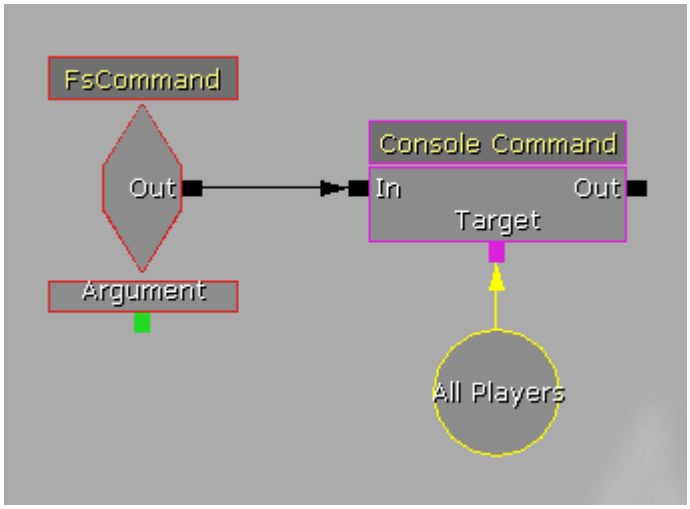


Figura 58

Donde *console command* tiene como parámetro en sus propiedades el comando *exit*, que finaliza el juego.

Por último, si queremos un acabado elegante, siempre podemos usar la imaginación. Por ejemplo podemos añadir una música que suene de fondo durante el menú, o hacer que de fondo se vea un *travelling* que mueva la cámara mostrando el escenario del mapa, añadir efectos de desenfoque o desaturación para resaltar el contraste entre el menú y nuestro mapa. Véase un ejemplo en la figura 59.

Figura 59

En realidad, este tipo de presentación es muy común en muchos juegos

UNREAL SCRIPT

La herramienta Kismet ofrece múltiples opciones de scripting de fácil implementación. Los resultados de usar esta herramienta pueden ser suficientes para que optemos que el desarrollo de nuestro juego no requiera scripting. La implantación de kismet tiene como propósito que usuarios poco introducidos en la programación puedan realizar sus propios juegos. Pero en realidad, dichos juegos no dejan de ser *mods* de Unreal Tournament. Así pues, se puede decir que crear scripts con UnrealScript permite control total sobre el motor del juego, y prácticamente infinitas posibilidades, mientras que utilizar Kismet limita su verdadero potencial.

Para un proyecto de visita virtual, en gran medida se puede configurar con Kismet, sin embargo, es contemplable en ultima instancia que haya un mínimo de scripting real.

Existe poca documentación en Internet, y nula en las bibliotecas, acerca de UnrealScript. La mayor parte son guías o referencias de ámbito general, y poco concisas. La iniciación en UnrealScript no es nada trivial, y además de eso, son frecuentes las actualizaciones por parte de Epic, que conllevan un cambio de proceder en muchos aspectos cada poco tiempo. Sirva de advertencia.

Consideraciones básicas – Juego y *Mod*

Antes de crear un juego en Unreal, hay que considerar si lo que se está desarrollando es un juego o un *mod*. Cuando estamos trabajando con el Kit de Desarrollo y creamos un mapa, (y aún si lo dotamos de secuencias *Kismet* y animaciones), cuando lo ejecutamos para jugarlo, lo que vamos a jugar es un modo **UTDeathmatch.** Nos daremos cuenta de eso por que apareceremos en el juego con todas las configuraciones propias de este modo, como son el arma, el modelo de nuestro personaje, el HUD (información en pantalla), así como que estarán definidas las reglas de este mismo modo, que son las de eliminar enemigos y ganar en caso de que se alcance un límite de muertes.

Esto es ni más ni menos que una partida de Unreal Tournament. La configuración del UDK le está diciendo al Motor que tiene que ejecutar los juegos bajo esas condiciones.

El modo UTDeathmatch no es el único modo que existe en Unreal Tournament, existen otros modos como capturar la bandera, o modos similares pero en los que intervienen vehículos. Cada uno de estos modos tiene unas reglas diferentes dentro de lo que es la jugabilidad del Unreal Tournament, pero en esencia todos ellos responden a las reglas

de comportamiento fundamentales del juego que se llama Unreal Tournament, y que se definen en la clase **UTGame**.

Ahí es donde viene la línea divisoria entre juego y *mod*: Si creamos un juego que herede de las reglas básicas de Unreal Tournament, el juego no dejará de ser Unreal Tournament, si no que será una versión modificada de dicho juego, o sea, un mod.

Por ejemplo, podemos crear un nuevo modo de juego que consista en una carrera de vehículos por un desierto. O bien podemos simplemente modificar algún modo de los ya existentes y simplemente cambiar aspectos como la altura que se alcanza al saltar, o el daño que cometen las armas en cada impacto.

La opción de crear un mod puede ser suficiente para llevar a cabo un proyecto sencillo. Unreal Tournament ya nos da muchas directrices que nos pueden servir para nuestro propósito, y que nosotros simplemente podemos modificar.

En una visita virtual a un edificio, por ejemplo, es lógico que no vamos a llevar armas con nosotros, ni vamos a ir corriendo de un lado a otro como si nos persiguiesen (una visita que pretenda ser seria). Así pues, se puede de manera sencilla modificar el modo UTDeathmatch para que no aparezcan armas, y para que se reduzca la velocidad de desplazamiento.

Por otra parte, si una compañía desarrolladora planea crear un juego, preferirá que tenga una identidad propia, y utilizar la tecnología Unreal a su manera. En tal caso, creará un juego basado en **GameInfo**, que es la clase básica que permite crear un juego "en crudo".

La estructura de clases

Para entender mejor lo dicho en el apartado anterior, hay que saber cómo UnrealScript está fundamentado en clases (figura 59). Se asume que se entiende perfectamente la mecánica de la programación en clases.

Existe una clase fundamental, Object, de la que heredan todas las demás clases. La estructura de clases está muy jerarquizada, en forma de árbol.

Toda clase posee como línea principal una definición de la clase tal que asi:

```
class ClaseActual extends Superclase
```

Seguida de la típica definición de variables. Las variables pueden ser de muchos tipos, las variables fundamentales son: **byte, int, bool, float, string, constant, enumeration**, además de existir las típicas agrupaciones de datos: **array** y **struct** (similar a C)
Las variables se definirán así:

```
var int a;
var byte Table[64];
var string PlayerName;
var actor Other;
var() float MaxTargetDist;
```

A continuación se expresan las funciones. Existen numerosas expresiones en el lenguaje UnrealScript, similares a las de C, con funciones y operadores de todo tipo.
Un aspecto a destacar es que en muchas ocasiones, al final de la clase se suele definir valores de propiedades por defecto, en el ámbito llamado *defaultproperties*, y que sirve para definir valores por defecto de ciertas variables de actores cuando son creados por una clase (un actor puede ser cualquier objeto que toma parte en el juego, aunque sea algo abstracto)

Explicar todo sobre UnrealScript conllevaria tanto material como para escribir un libro, asi que nos centraremos en las nociones fundamentales. Como ayuda, existe una completa guia de referencia para el lenguaje UnrealScript se puede encontrar en las páginas de UDN. **[10]**

Volviendo a las clases, existen varias clases que son las más importantes al hacer un juego.

- Para empezar tenemos la clase **GameInfo**. Esta clase define qué juego se está jugando, las reglas que usa, qué actores existirán en este juego. En nuestro kit UDK, esta clase define varios juegos: **MovileGame** y **UDKGame**. Movile game presenta un tipo de juego creado como demostración para ejecutarlo en dispositivos iOS como el iPad o el iPhone4. (Esta demostración se puede cargar desde el UnrealEditor o bién desde UnrealFrontend). UDKGame es la clase de la que heredan los juegos que vayamos a crear para PC, y de ésta clase heredan los juegos de UnrealTournament, concretamente, Deatchmatch, Deatchmatch por

equipos, y Capturar la bandera (estos 2 heredan de deatchmatch al ser una **extensión** de sus reglas de juego)

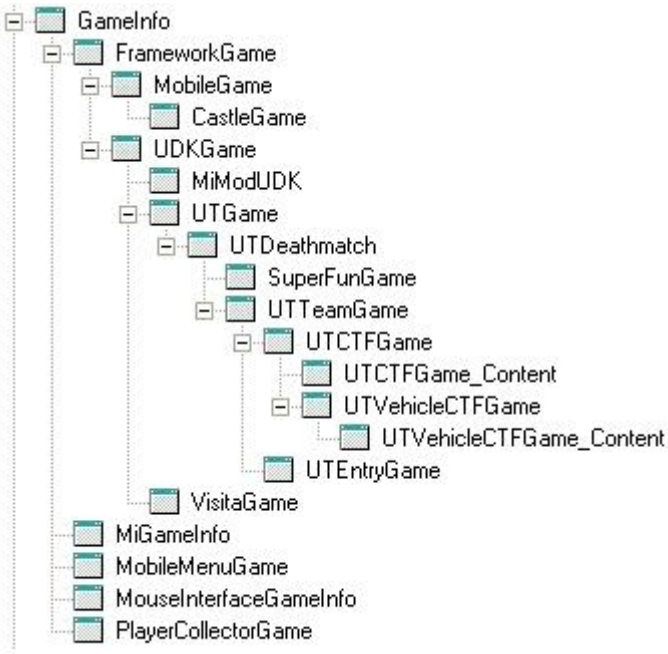


Figura 59: Vista de las subclases de GameInfo

- La clase **Pawn** define la representación física de los jugadores y criaturas del nivel, manejadas por jugadores o por I.A. Es decir, representa el "muñeco" que al que controlamos, desde su modelado, animación, a las interacciones físicas con el entorno; colisiones, físicas, sonidos que emiten, armas que llevan, daño que hacen, etc.En términos de programación, se denomina *pawn* este tipo de actores.

- **Controller:** Son actores abstractos, o no físcos, que sirven para controlar los *pawn*, osea, todo personaje jugable. De esta clase heredan dos clases: **PlayerController**, que define los controles de los jugadores humanos, y **AIController**, que define los controles de la inteligencia artificial sobre dichos *pawn*, es decir define un comportamiento que simula el de un humano para esos Pawn. A los personajes controlados por AI se les llama *bots*.

- **GameCamera:** Muchos desarrolladores encuentran interesante que se pueda modificar la cámara del juego, ya que esto es fundamental para definir el estilo

de juego. Asi pues, hay juegos en primera persona, como el propio Unreal Tournament, y otros en tercera persona, e incluso con camaras desde lejos, en isométrico, en plano zenital… Esta clase define este aspecto.

- **Input:** Un objeto input mapeta todas las asociaciones entre funciones que realiza el jugador y los botones correspondientes en el teclado o el mando.

- **HUD:** Es la superclase para el *Head Up Display*, que es todo aquello que se dibuja en pantalla para mostrar todo tipo de información relativa al juego en tiempo real, como por ejemplo, el estado del cargador de munición, el punto de mira o menús contextuales en el juego.

Flujo del juego en UnrealScript

Un perspectiva importante para acercar a los programadores a cómo funciona el Motor Unreal 3 a nivel de funciones llevadas a cabo dentro de un flujo de ejecución. Esto les facilita la configuración de sus juegos según lo necesite. Ver la figura 60.

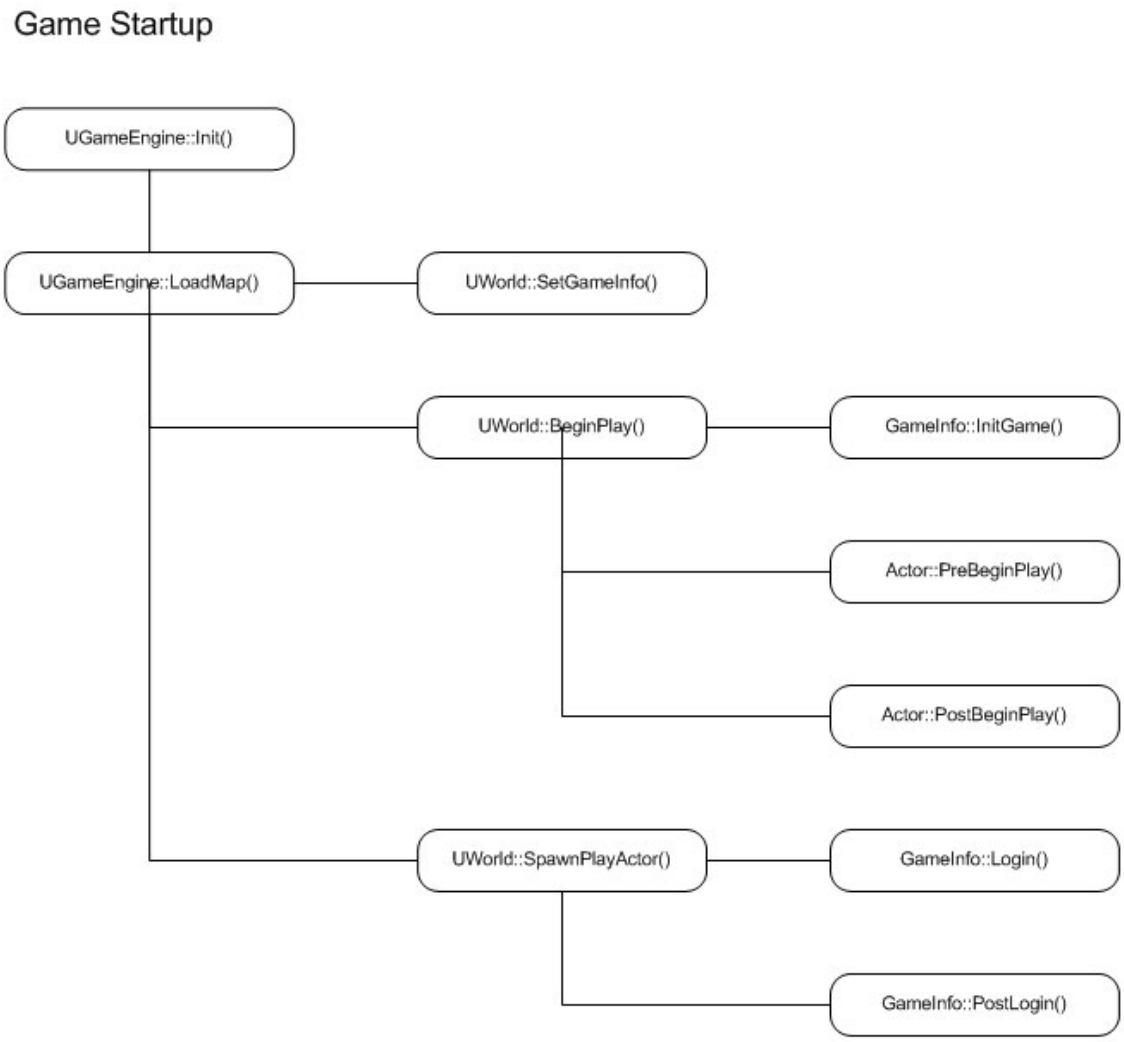


Figura 60

La secuencia se explica así:

Se inicializa el motor
El motor carga un mapa
Se establece el tipo de juego
Se inicializa el juego (comienza la partida jugable):
-GameInfo:InitGame() es llamado para inicializar el tipo de juego
-Actor::PreBeginPlay() inicializa actores
-Actor::PostBeginPlay() inicializa actores
Un jugador se une al juego
-GameInfo::Login() Maneja la aparición del jugador
-GameInfo::PostLogin() Maneja la aparición del jugador

Configurar el entorno de desarrollo

Antes de seguir explicando acerca del scripting, es importante dejar claro cómo se puede configurar el entorno. Existen varias alternativas para esto, a gusto del usuario.

En cualquier caso, hay un programa muy aconsejable que se llama UnCodex, que es más bien un navegador de paquetes y clases y que necesitaremos para movernos dentro de todas las clases. UnCodex proporciona una forma cómoda y directa de conocer la estrucuración de clases en UnrealScript
Aconsejo que sea el primer programa que se instale. Puede descargarse desde la página oficial. **[11]**

Instalación de UnCodex:
Ejecutar el instalador y seguir los pasos que indique.
A continuación ejecutar UnCodex. La primera vez que se ejecute, dará la opción de si se quiere cambiar la configuración.
Seleccionar *yes* Esto abrirá la ventana de configuración de la figura 61. En ella iremos a la opción *source paths*, y le daremos al botón *add*. Esto es simplemente para indicar a UnCodex cuál es la ruta del directorio en el que se encuentran todos los paquetes de las clases.

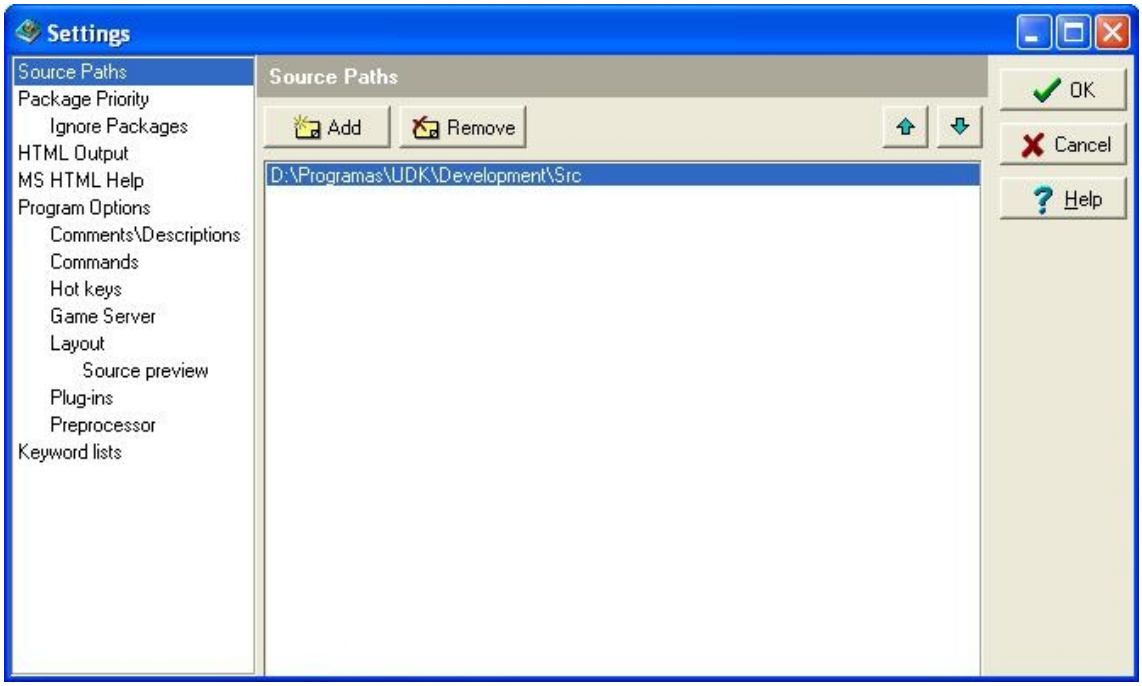


Figura 61

Seleccionamos la carpeta **\Development\Src** que estrá dentro de la carpeta de instalación de UDK. Aceptamos dándole a OK. A continuación UnCodex nos solicitará que se realice un escaneo y análisis de las todas las fuentes. Elegimos *yes*. El proceso durará unos segundos. Entonces UnCodex estará listo para usarse. Nos encontraremos la interfaz de UnCodex como la vemos en la figura 62.

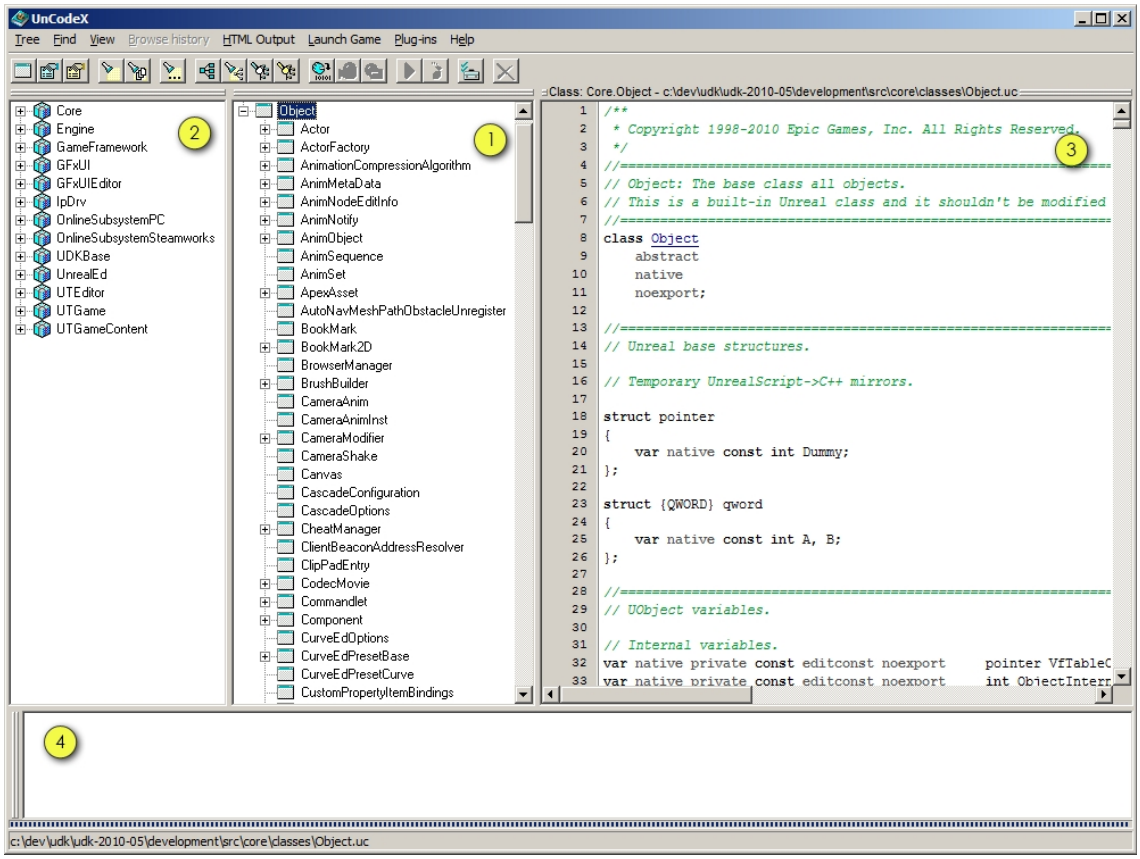


Figura 62

En la imagen podemos ver que UnCodex se divide en 4 campos:

1. El árbol de clases. Podemos crear fácilmente subclases de una clase haciendo clic derecho sobre ella y seleccionando *create subclase*.
2. El navegador de paquetes. Toda clase se almacena en un paquete con razones organizativas. Se aconseja que las nuevas clases se creen en un paquete nuevo que se identifique con el proyecto en concreto que se esté desarrollando.
3. Vista del código de la clase. Muchas de las clases de UDK, las más importantes, tienen abundantes comentarios explicativos. Se aconseja que se lean en muchos casos.
4. Ventana de eventos. Muestra avisos, errores e información.

UnCodex puede además ejecutar el Motor UDK para probar los juegos. Es decisión del desarrollador si quiere hacer esto desde este programa o bien prefiere usar otra de las múltiples opciones posibles.

Configurar Visual Studio 2008 como entorno de desarrollo. nFringe.

No existe un único procedimiento para crear scripts, ni una única herramienta. Un simple editor de texto como el block de notas nos permite hacer scripts para Unreal, siempre que lo guardemos como un archivo de extensión **.uc,** y esté dentro de la carpeta correspondiente a su clase.

Pero por supuesto, lo que nos interesa es poder disponer de un entorno de desarrollo conveniente para programar UnrealScript, que nos facilite el trabajo.

Para esto disponemos de muchas opciones. La opción que he escogido es la de usar el entorno **Visual Studio 2008**, aunque existen otras opciones que el desarrollador puede investigar y usar si es que le interesa un entorno de desarrollo diferente.

Se asume que el desarrollador posee una instalación de Visual Studio 2008. El primer paso que tenemos que hacer es descargar e instalar la extensión **nFringe**, que se puede la página oficial. **[12]**

nFringe es una extensión para Visual Studio que ofrece la posibilidad de crear proyectos UnrealScript, depurarlos, y proporciona un IDE que hace la implementación cómoda.

Nota: nFringe es gratis para uso no comercial. Para que funcione correctamente, el usuario debe estar registrado, tanto si lo ha comprado como si no. En el caso de que lo vaya a usar con fines no comerciales, deberá facilitar un nombre que le identifique y una dirección de correo electrónico. Una vez hecho esto, las características de nFringe funcionarán al 100%.

La instalación de nFringe es trivial. Una vez instalado, ya se puede crear un proyecto *UnrealEngine3*, tal como se muestra en la figura 63.

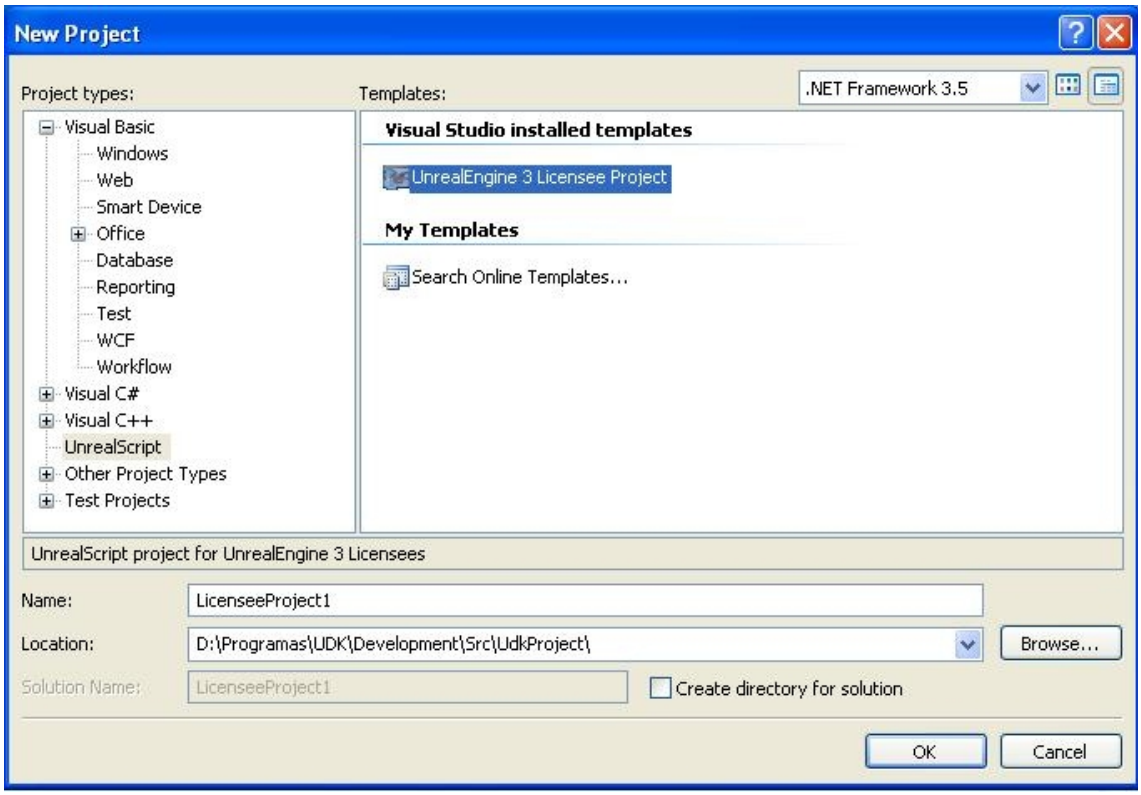


Figura 63

Una vez se ha creado el proyecto, hay varias configuraciones que debemos hacer en él: Abrimos la ventana de propiedades del proyecto actual, la pestaña General, y nos aseguramos de que se configure según la figura 64.

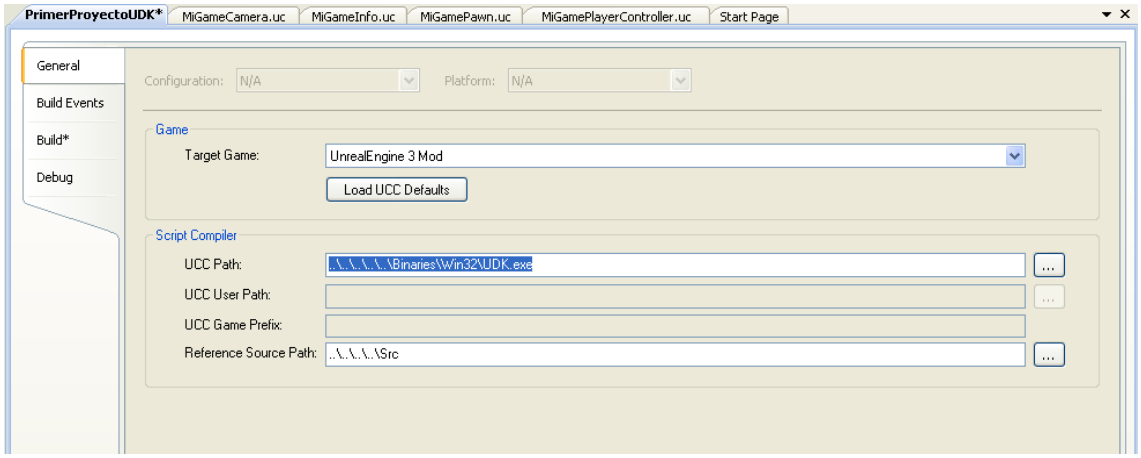


Figura 64

En "UCC Path" ponemos la ruta del ejecutable UDK.exe, y en "Referente Source Path", la ruta que se corresponde con las clases, o sea, **\Src**.

Una vez echo esto, cada vez que abramos un archivo de clase .uc, se abrirá un editor Visual Studio preparado para UnrealScript.

Podemos optar por trabajar desde el espacio de la solución del proyecto dentro de *Visual Studio*, aunque también hay quien prefiere, por comodidad, trabajar con *UnCodex* en combinación con Visual Studio.

Para crear un proyecto, simplemente podemos crear una carpeta dentro de \Development\Src. Al hacer esto estaremos creando un nuevo paquete dentro del Motor Unreal, y lo podremos visualizar en UnCodex una vez hayamos re-analizado el contenido de \Src. Veamos un ejemplo. Hemos creado una carpeta que se llama **MiJuego**. Dentro de *MiJuego*, crearemos una carpeta llamada *Classes*. Y ahí dentro iremos creando nuestras clases.

Para crear una clase nueva, nada mas fácil que hacerlo en UnCodex. Lo bueno de hacerlo en UnCodex, es que podemos crear la clase directamente debajo de aquella clase ya existente de la que queremos que herede, y se creará automáticamente un archivo .uc que tenga la cabecera de inicialización de clase conveniente.

Así pues, si queremos crear una clase llamada **MiModUdk,** que queramos que herede de UTDeathMatch, por que se trata de una pequeña modificación del modo DeathMatch, sólo tenemos que hacer clic derecho en la clase que vaya a ser padre, en este caso UTDeathMatch, seleccionar *Create Subclass*, y especificar a qué paquete pertenecerá esa clase, en este caso a *MiModUdk*. Al aceptar, aparecerá una ventana de Visual Studio con la clase, lista para ser editada.

Algunos ejemplos

Vamos a ilustrar un ejemplo de clase sencilla. Esta clase es una pequeña modificación del modo *DeathMatch*, que lo único que hace es que cuando el jugador mata a un enemigo, aparezca un mensaje por pantalla, y se escuche una voz de comentarista (un sonido asociado al objeto "*UTKillingSpreeMessage*"):

```
class SuperFunGame extends UTDeathmatch;

/**
  * Enviar un anuncio al jugador en el momento de matar a un enemigo
  */
function Killed( Controller Killer, Controller KilledPlayer, Pawn KilledPawn,
class<DamageType> DamageType )
{
```

```
        `log("SUPER FUN KILL by "$Killer); /esto muestra mensajes en pantalla,
útil para depuración
        if ( PlayerController(Killer) != None )
        {


PlayerController(Killer).ReceiveLocalizedMessage( class'UTKillingSpreeMessage'
, 5, Killer.PlayerReplicationInfo, None );
        }
        Super.Killed(Killer, KilledPlayer, KilledPawn, DamageType);
}

/**
  estas líneas sobreescriben la información acerca del tipo de juego
  */
static event class<GameInfo> SetGameType(string MapName, string Options,
string Portal)
{
        return default.class;
}
```

A continuación una clase que provoca modificaciones sencillas en el juego, suprimiendo el arma del jugador, evitando que pueda morir, y evitando que pueda agacharse y que se mueva demasiado deprisa. Esto podría ser un comienzo para un juego basado en una visita virtual:

```
class VisitaGame extends UDKGame;


/**
 * Aqui evitamos que se pueda morir en el juego Visita
 */
function bool PreventDeath(Pawn KilledPawn, Controller Killer,
class<DamageType> DamageType, vector HitLocation)
{
        return true;
}

static event class<GameInfo> SetGameType(string MapName, string Options,
string Portal)
{
                return default.class;
}



class MiUTPawn extends UTPawn;

defaultproperties
{
        GroundSpeed=+00300.000000          //velocidad reducida

        bCanCrouch=false                   // impide al jugador agacharse

        bWeaponAttachmentVisible=false     // oculta el arma
        JumpZ=+0.000000                    // impide que se pueda saltar
}
```

Por último se muestra el código de varias clases creadas siguiendo un tutorial. La finalidad de estas clases es la de conseguir que el juego se vea en 3ª persona, que la cámara siga al *pawn* desde detrás, y no solo eso, si no que se puede acercar o alejar al gusto mediante la rueda del ratón. El código es complejo, pero hay abundantes explicaciones, que se aconsejan leer para entender un poco más sobre *UnrealScript*.

Clase MiGameinfo

```
/*Este es el Gameinfo que configura el juego con la camara en 3ª persona*/


class MiGameInfo extends GameInfo; //This line tells UDK that you want to
inherit all of the functionality of GameInfo.uc, and add your own. The name
after "class" must match the file name.
DefaultProperties //Self explanatory
{
	bDelayedStart = false
	PlayerControllerClass = class 'MiJuego.MiGamePlayerController' //Setting
the Player Controller to your custom script
	DefaultPawnClass = class 'MiJuego.MiGamePawn' //Setting the Pawn to your
custom script}
```

Clase MiGameCamera

```
class MiGameCamera extends Camera; //La clase que redefine el uso de la cámara
para nuestro juego, y que se referencia desde "MiGamePlayerController"

	//Inicializando variables estáticas
var float Dist;  /*distancia. Por lo visto, esta variable se declara FUERA de
la función con el fin de que no se inicialize en cada ciclo del programa,
si no que dicho valor cambie solo cuando sea se le diga explicitamente*/

function UpdateViewTarget(out TViewTarget OutVT, float DeltaTime)
{
	//Declarando variables locales
	local Vector Loc, Pos, HitLocation, HitNormal;
	local Rotator Rot;
	local Actor HitActor;
	local CameraActor CamActor;
	local bool bDoNotApplyModifiers;
	local TPOV OrigPOV;

	OrigPOV = OutVT.POV;//almacenar el POV previo, en caso de que se
necesite después
	OutVT.POV.FOV = DefaultFOV;

	//Ver a través del actor cámara
	CamActor = CameraActor(OutVT.Target);
	if(CamActor != None) //este tipo de condiciones son extensamente usadas,
y aseguran que se preocede existiendo el actor clave que se va a usar.
	{
		CamActor.GetCameraView(DeltaTime, OutVT.POV);
		//Relacion de aspecto para el actor cámara
```

```
            bConstrainAspectRatio = bConstrainAspectRatio ||
AnimCameraActor.bConstrainAspectRatio;
            //Si el CameraActor quiere modificar la configuración de
postproceso usada:
            CamOverridePostProcessAlpha =
AnimCameraActor.CamOverridePostProcessAlpha;
            CamPostProcessSettings = AnimCameraActor.CamOverridePostProcess;
      }
      else
      {
            if( Pawn(OutVT.Target) == None || !
Pawn(OutVT.Target).CalcCamera(DeltaTime, OutVT.POV.Location,
OutVT.POV.Rotation, OutVT.POV.FOV) )
            {
                  bDoNotApplyModifiers = TRUE;

            /*A CONTINUACIÓN SE IMPLEMENTAN LOS ESTILOS DE CÁMARA*/

            switch(CameraStyle)
            {
            case 'Fixed': //No hay cambio, mantener la vista previa
                  OutVT.POV = OrigPOV;
            break;

            case 'ThirdPerson':

            case'FreeCam':
                  Loc = OutVT.Target.Location; //configurar la posición y la
rotación de la cámara en el objetivo.
                  Rot = OutVT.Target.Rotation;
                  if (CameraStyle == 'ThirdPerson')
                  {
                        Rot = PCOwner.Rotation; //configurar la posición de
la cámara a la posición del pawn
                  }
                  //OutVT.Target.GetActorEyesViewPoint(Loc, Rot);
                  if (CameraStyle == 'FreeCam')
                  {
                        Rot = PCOwner.Rotation;
                  }
                  Loc += FreeCamOffset >> Rot;
                  //Algoritmo de interpolación lineal. Esto "suaviza" el
moviento para que la cámara no salte entre los niveles de zoom
                  if (Dist != FreeCamDistance)
                  {
                        Dist = Lerp(Dist,FreeCamDistance,0.15); /*Incrementa
Dist hasta FreecamDistance, que es donde quieres la cámara. Incrementa un
porcentaje de la distancia entre ellos de acuerdo al tercer parámetro, en este
caso, 0.15, o 15%*/
                  }
                  Pos = Loc - Vector(Rot) * Dist;

                  HitActor = Trace(Hitlocation, HitNormal, Pos, Loc, FALSE,
vect(12,12,12)); //esto determina si la cámara atravesará un modelo
                  OutVT.POV.Location = (HitActor == None) ? Pos :
HitLocation;
                  OutVT.POV.Rotation = Rot;  //Aquí es donde se pone la
cámara en realidad cuando hay colisión
            break;

            case 'FirstPerson':
            default: OutVT.Target.GetActorEyesViewPoint(OutVT.POV.Location,
OutVT.POV.Rotation); //visión en primera persona, vista a través de los ojos
del jugador;
            break;
            }
      }
}
```

```
if (!bDonotApplyModifiers)
{
	ApplyCameraModifiers(DeltaTime, OutVT.POV);
}
//'log( WorldInfo.TimeSeconds  @ GetFuncName() @ OutVT.Target @
OutVT.POV.Location @ OutVT.POV.Rotation @ OutVT.POV.FOV ); //información de la
cámara
}

Defaultproperties
	{
		FreeCamDistance = 192.f //Distancia de la cámara al jugador por
defecto
	}
```

Clase MiGamePawn

```
class MiGamePawn extends GamePawn;  //Pawn hace referencia a todo lo relativo
a la presencia "fisica" del player actual

//This lets the pawn tell the PlayerController what Camera Style to set the
camera in initially (more on this later).
simulated function name GetDefaultCameraMode(PlayerController RequestedBy)
{
	return 'ThirdPerson';
}
Defaultproperties
{
	//Components.Remove(Sprite)
	Components.Remove(Sprite)

	//Configurando el componente de iluminación
	Begin Object Class=DynamicLightEnvironmentComponent
Name=MyLightEnvironment
		ModShadowFadeoutExponent=0.25
		MinTimeBetweenFullUpdates=0.2
		AmbientGlow=(R=.01,G=.01,B=.01,A=1)
		AmbientShadowColor=(R=0.15,G=0.15,B=0.15)
		LightShadowMode=LightShadow_ModulateBetter
		ShadowFilterQuality=SFQ_High
		bSynthesizeSHLight=true
	end object
	Components.Add(MyLightEnvironment)

	//Configurando el componente de modelo y animación
	begin object Class=SkeletalMeshComponent Name=InitialSkeletalMesh
		CastShadow=true
		bCastDynamicShadow=true
		bOwnerNoSee=false
		LightEnvironment=MyLightEnvironment;  //Aqui uso mi componente de
iluminación dinámica explicitamente creado
		BlockRigidBody=true;
		CollideActors=true;
		BlockZeroExtent=true;
		//cambiar si se quiere usar otros modelos o animaciones


PhysicsAsset=PhysicsAsset'CH_AnimCorrupt.Mesh.SK_CH_Corrupt_Male_Physics'
		AnimSets(0)=AnimSet'CH_AnimHuman.Anims.K_AnimHuman_AimOffset'
		AnimSets(1)=AnimSet'CH_AnimHuman.Anims.K_AnimHuman_BaseMale'
		AnimTreeTemplate=AnimTree'CH_AnimHuman_Tree.AT_CH_Human'

	SkeletalMesh=SkeletalMesh'CH_LIAM_Cathode.Mesh.SK_CH_LIAM_Cathode'
	end object
```

```
	//Configurando el cilindro de colisión
	Mesh=InitialSkeletalMesh;
	Components.Add(InitialSkeletalMesh);
	//bConsiderAllStaticMeshComponentsForStreaming.Add(InitialSkeletalMesh);
	begin object Name=CollisionCylinder
		CollisionRadius=+0023.000000
		CollisionHeight=+0050.000000
	end object
	CylinderComponent=CollisionCylinder
}
```

Clase MiGamePlayerController

```
class MiGamePlayerController extends GamePlayerController;  //En esta clase se definen aquellos controles que queremos añadir al juego, para poder hacer uso de las nuevas funciones

simulated event PostBeginPlay()
{
	super.PostBeginPlay();
	`Log("I am alive!"); //Esto envia un mensaje al log (se abre con el el prefijo -log al ejecutar UDK). Estos mensajes son útiles para supervisar en tiempo real qué es lo que ocurre
}

//Funciones para acercar-alejar zoom
exec function NextWeapon() /*el comando "exec" le dice a UDK que esta función puede ser llamada por consola o keybind (asignación de tecla)
	NextWeapon() no cambia el arma en nuestro juego, es la asignación que hace el motor a la acción de rodar la rueda del ratón abajo*/
	{
		if (PlayerCamera.FreeCamDistance < 512) //Comprueba que el valor de la distancia de la camara al objetivo no sea superior que el deseado.
		{
		`Log("MouseScrollDown");
		PlayerCamera.FreeCamDistance += 64*(PlayerCamera.FreeCamDistance/256); /*Esto incrementa la distancia de la camara en 64 en función de la distancia actual sobre una base de 256. (Cuanto mas lejos está la cámara, mas aunmenta el incremento de distancia) Es algo que mejora la jugabilidad y hace el zoom más cómodo.*/
		}
	}

	exec function PrevWeapon()
{
	if (PlayerCamera.FreeCamDistance > 64) //si la distancia de la camara se encuentra fuera de nuestra distancia mínima (64)
	{
		`Log("MouseScrollUp");
		PlayerCamera.FreeCamDistance -= 64*(PlayerCamera.FreeCamDistance/256); //orden análoga a la anterior, para acercar la cámara
	}
}

defaultproperties
{
	CameraClass=class 'MiGameCamera'//Decir a PlayerController que use la clase de Cámara deseada
	DefaultFOV=90.f  //Campo de visión 90°, para un juego en 3ª persona es más cómodo mas de los 70ª que se usan por defecto en Unreal
}
```

[13]

Modificar archivos de configuración

Un paso importante entre la creación de nuestros scripts y su testeo, es el de editar los archivos de configuración de UDK. Sencillamente por que el motor aún no sabe que tiene que cargar tu paquete a la hora de ejecutarse.
Existe una carpeta llamada \UDKGame\Config. Esta carpeta es fundamental, contiene los archivos ini, y sirven para decirle al motor Unreal qué configuraciones debe aplicar.
Existen varios tipos de archivos de configuración:

- Base*.ini: (están en \Engine\Config). Son las configuraciones de Epic para el motor. No deben manipularse.
- Default*.ini: son **nuestras** opciones de configuración. Sobrescribirán las opciones Base si son diferentes, y serán las opciones por defecto en nuestro juego.
- UDK*.ini: son las configuraciones del **usuario final.** Estas configuraciones podrán ser cambiadas por los usuarios a su gusto. Reescriben las opciones default. Si son borradas, se reemplazarán por estas también.

En esencia, nosotros vamos a configurar los archivos Default*.ini para nuestro propósito. En DefaultEngine.ini, por ejemplo, podemos configurar con qué mapa, modificando un par de lineas:
Arrancará nuestro juego:

```
[URL]
MapExt=udk
Map=MiMapa.udk
LocalMap=MiMapa.udk
TransitionMap=EnvyEntry.udk
EXEName=UTGame.exe
DebugEXEName=DEBUG-UTGame.exe
```

O también añadir una definición para cada uno de los paquetes que hayamos creado. Así se incluiría el paquete **MiPaquete**:

```
[UnrealEd.EditorEngine]
+EditPackages=UTGame
+EditPackages=UTGameContent
+EditPackages=MiPaquete
```

Ahora, las configuraciones que hay que hacer para que surta efecto los scripts para juego en tercera persona desarrollado anteriormente. Suponemos que nuestras cuatro clases están dentro de un paquete llamado MiJuego:

Fichero DefaultGameUdk.ini:

```
[Engine.GameInfo]
DefaultGame=MiJuego.MiGameInfo
DefaultServerGame=MiJuego.MiGameInfo
PlayerControllerClassName=MiJuego.MiGamePlayerController
DefaultGameType="UDKBase.UDKGame";
```

Como se puede ver, estamos configurando de modo que el motor usará las clases de nuestro juego por defecto, así como la clase MiGamePlayerController también por defecto.

Fuentes: **[14]**

He aquí una muestra del juego ejecutando con nuestro paquete cargado, en la figura 65 apreciamos cómo nuestro androide se maneja desde una perspectiva de 3ª persona:

Figura 65

Unreal Frontend

Una vez creadas nuestras clases, debemos compilarlas para que el motor Unreal las integre dentro de si.

Cuando modificamos la estructura de clases, estamos realizando cambios que presumiblemente afectarán a la ejecución del motor Unreal. Para que estos cambios surtan efecto, UDK debe recompilar los scripts. Esto se puede hacer automáticamente al ejecutar nuevamente UDK.exe. Si ha habido cambios en las clases, antes de abrirse el editor, UDK detecta estos cambios y pregunta al usuario si desea recompilar los scripts.

También disponemos de la herramienta **UnrealFrontend**. (figura 66) Esta herramienta pone a nuestra disposición una interfaz intuitiva con la que podemos compilar los scripts, probar nuestros juegos, o **empaquetarlos en archivos de instalación auto-extraíbles**.

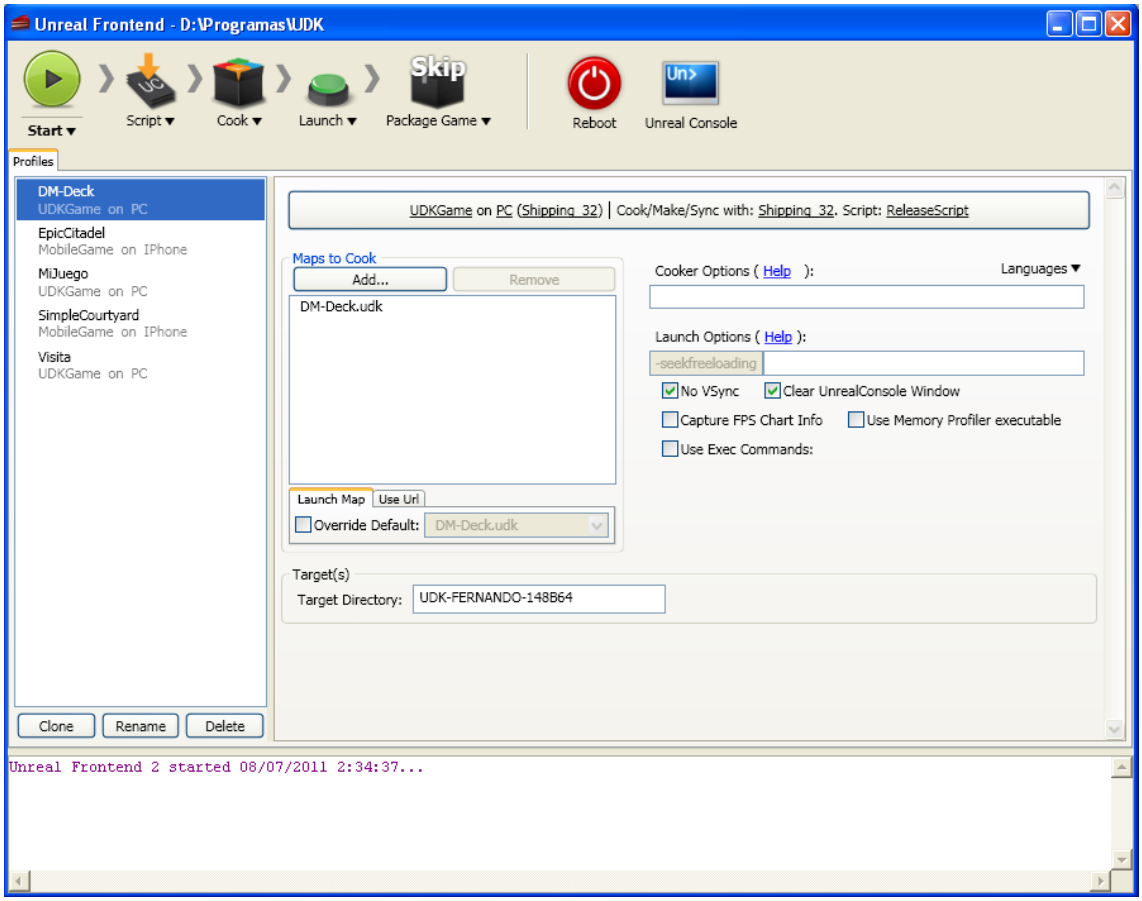


Figura 66

Cuando tengamos finalizado nuestro juego, deberemos crear un perfil para él. Para eso, hay que clonar uno de los que existen actualmente en la lista, y cambiar el nombre por el que queramos nosotros. A continuación, en el campo *maps to cook*, iremos añadiendo los archivos *udk* que conformen nuestro juego. Hay que aclarar que un archivo *udk* no es necesariamente un mapa, si no que puede ser también un menú principal del juego, que preceda a la carga de un mapa.

Antes de proceder al empaquetado, *UnrealFrontend* hace una compilación de todas las clases. Podemos elegir que se efectúe la compilación por nuestra cuenta, de forma manual, haciendo clic en *Script, Full Recompile*.

Por último creamos un paquete, tras las configuraciones oportunas, que será el producto acabado **listo para su distribución.**

Este paquete consiste en un fichero ejecutable (exe), que comprime en el nuestro juego, con todos los materiales, paquetes, mapas, etc. que se han empleado en nuestro juego, además de todas las configuraciones que hemos establecido.

El usuario final simplemente tiene que ejecutar este fichero, y el juego se auto extrae y se instala en su sistema.

Existe la posibilidad de crear proyectos para su instalación en sistemas iOS. Pero para esto es necesario estar registrado como desarrollador iOS, y validarse por Apple, el cual otorgará el certificado oportuno. Un cuadro de diálogo surgirá para guiar a través de los procesos necesarios para obtener dicho certificado.

Detalles necesarios sobre el uso de Unreal Frontend se pueden encontrar en UDN. **[15]**

ALGUNOS EJEMPLOS DE MAPAS

Ejemplo de tutorial: *SimpleLevel*

Existe una serie de video-tutoriales muy recomendable, por no decir imprescindible, para aprender de la forma más práctica posible las nociones de uso de Unreal Editor, desde el modelado hasta el uso de *Kismet* y *Matinée*. Pueden encontrarse en la sección de video tutoriales de la red UDN. **[16]**

Yo he seguido los pasos indicados en este tutorial, y en última instancia he añadido algunos elementos de creación propia. Las figuras mostradas a continuación se corresponden con imágenes tomadas durante su desarrollo.

Dicho desarrollo comienza por el modelado, usando brushes. (figura 67).

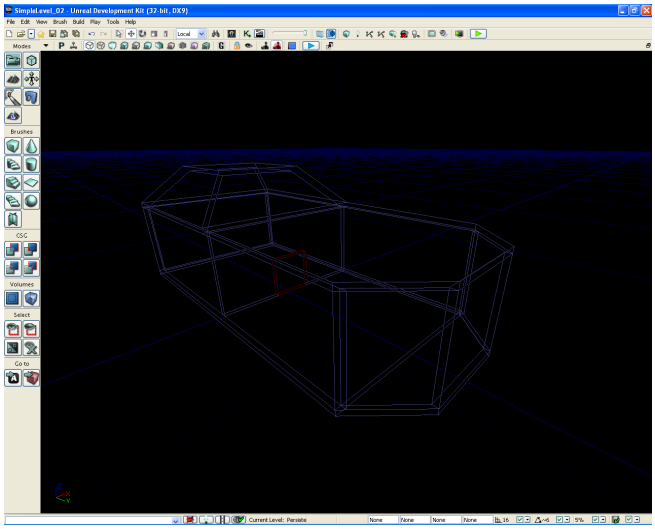


Figura 67

Pasando por la colocación de materiales y modelos estáticos (figura 68).

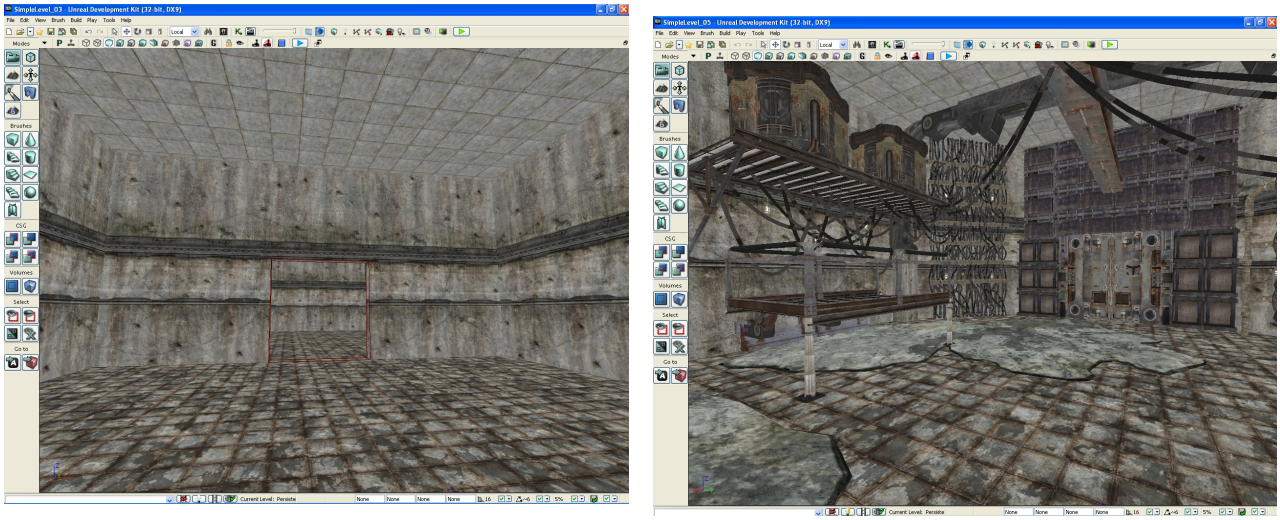


Figura 68

A continuación la aplicación de iluminación y postprocesado (figura 69).

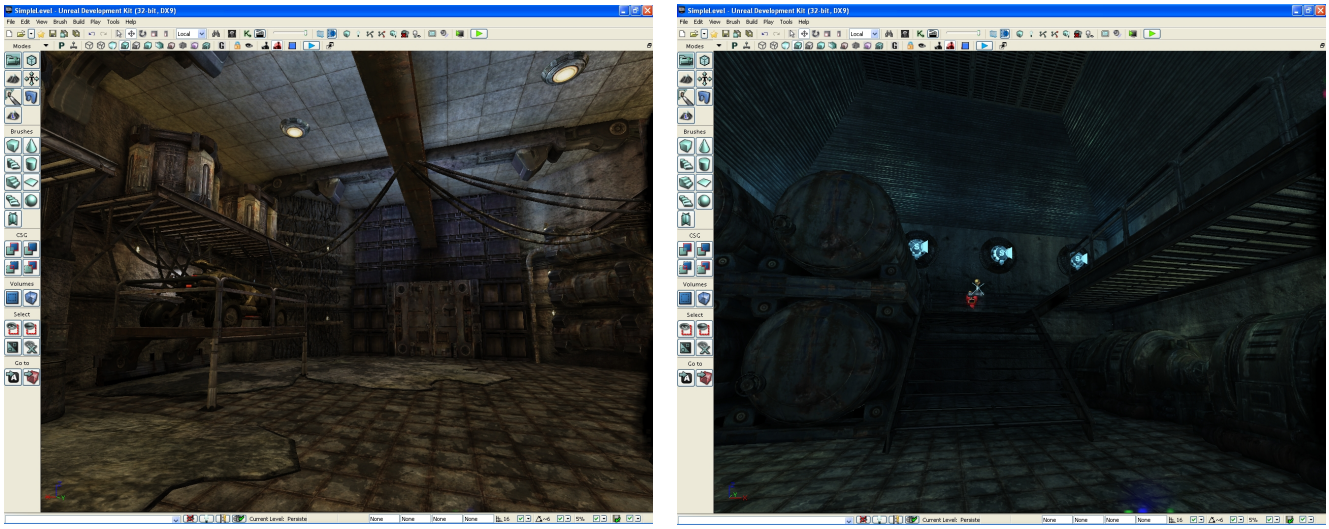


Figura 69

Por último, se han realizado secuencias *Kismet*, para programar una puerta que se abre al acercarse el jugador, y se cierra al alejarse, y se ha puesto un botón que al ser pulsado, realiza una secuencia de mensajes por pantalla, con una cuenta atrás para finalmente terminar con la destrucción del jugador. (figura 70).

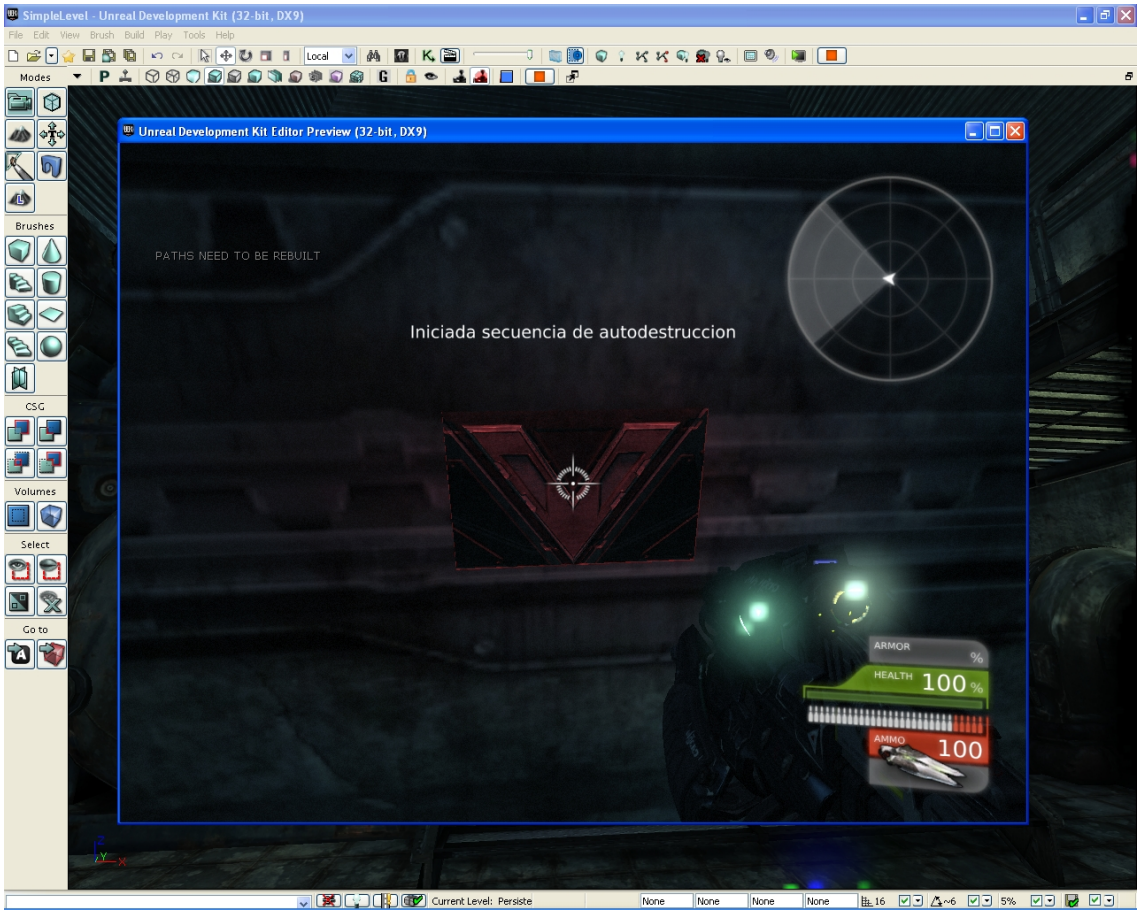


Figura 70

Ciclo día – noche

Este mapa está basado en el mapa *terreno*, que se creó para probar el editor de terreno. La intención es la de crear un ciclo de día-noche en el que el cielo cambia en función del tiempo, así como la iluminación. (figuras 71 y 72).

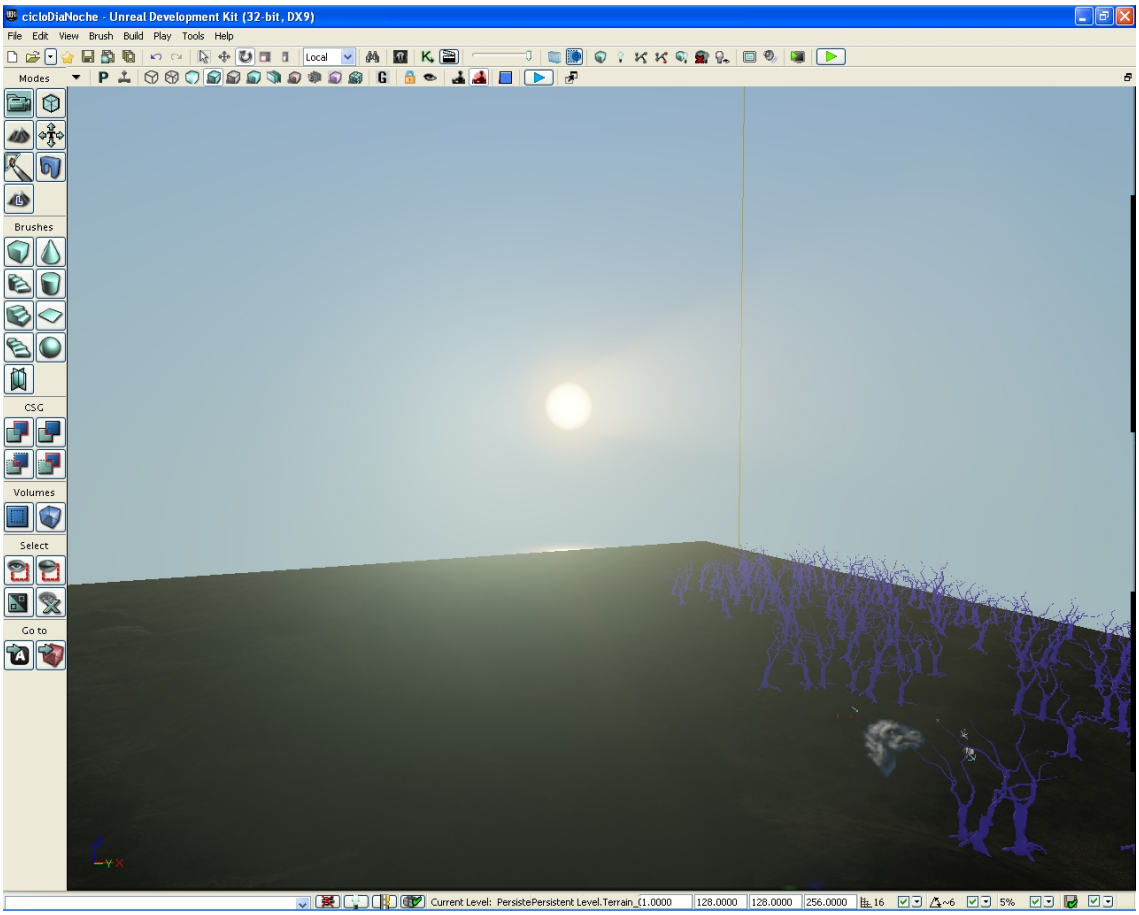


Figura 71

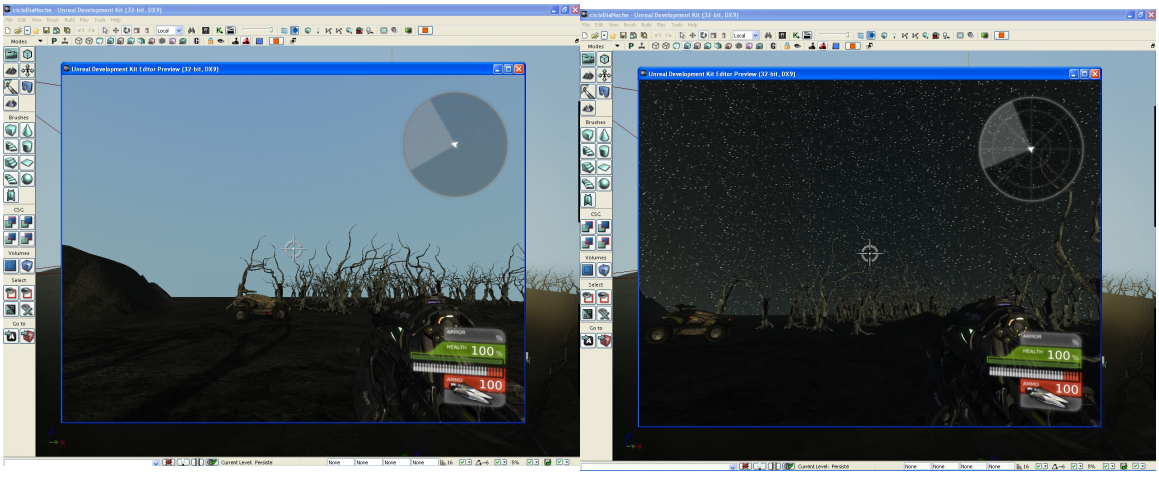


Figura 72

Para lograr esto, se ha creado un material especial, llamado **cielo procedural**, que consiste en una serie de capas de texturas y operaciones. Se puede ver la composición del material en la figura 73.

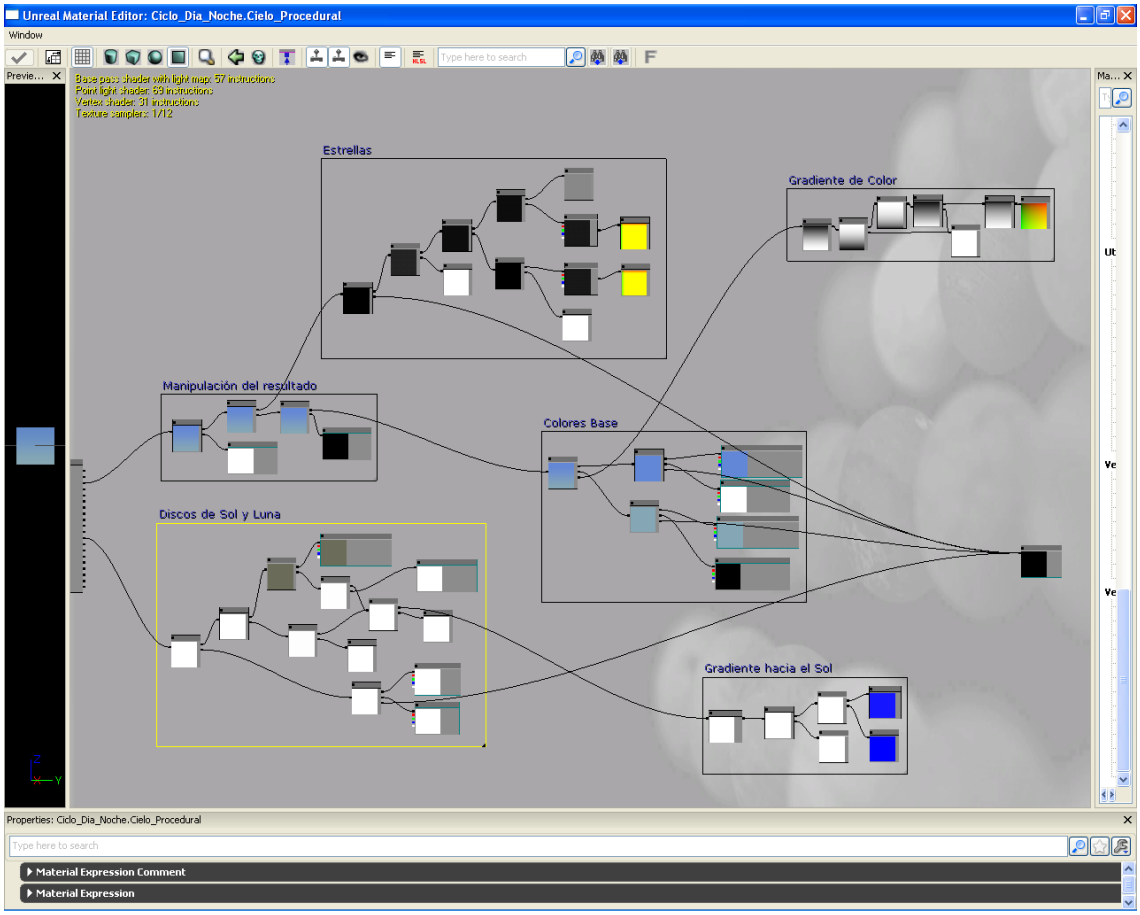


Figura 73

Por ejemplo, la capa de estrellas, tiene una transparencia. El azul del cielo tiene un factor de degradado. Todos estos parámetros se controlan en *Matinée* para ajustarlos a su posición según el tiempo. Una vez creado el material, se hace clic derecho en él en el navegador de contenidos, y se selecciona *create new material instante (time varying)*, lo que hace que este material pueda ser parametrizado en *Matinée.* Se coloca una esfera grande que cubra todo el mapa, y se aplica dicho material en la cara interna de la esfera.

Para la luz, se ha usado una luz ambiental, y una luz direccional dominante movible, que tiene un movimiento de rotación en *Matinée*. Esta luz ha sido configurada para que sea perpendicular al centro del material (disco solar).

Casualmente, en una de sus actualizaciones, UDK incorporó un material de cielo procedural, aunque más sofisticado, con nubes y varios efectos atmosféricos.

Mouse Interface

Es el nombre que he dotado al mapa que creé para probar un modo de juego creado en *UnrealScript*. Básicamente, en este modo, se ha creado una interfaz de pantalla que muestra un cursor del ratón. Habitualmente durante una partida de Unreal, el ratón debe estar oculto. Este *mod* permite usar el cursor durante tiempo de juego. Por otro lado, se ha implementado una nueva clase de objetos utilizables en el mapa. Este tipo de objetos pueden ser instanciados en *Kismet* (figura 74), y tienen la propiedad de que pueden crear acciones cuando interaccionan de algún modo con el ratón.

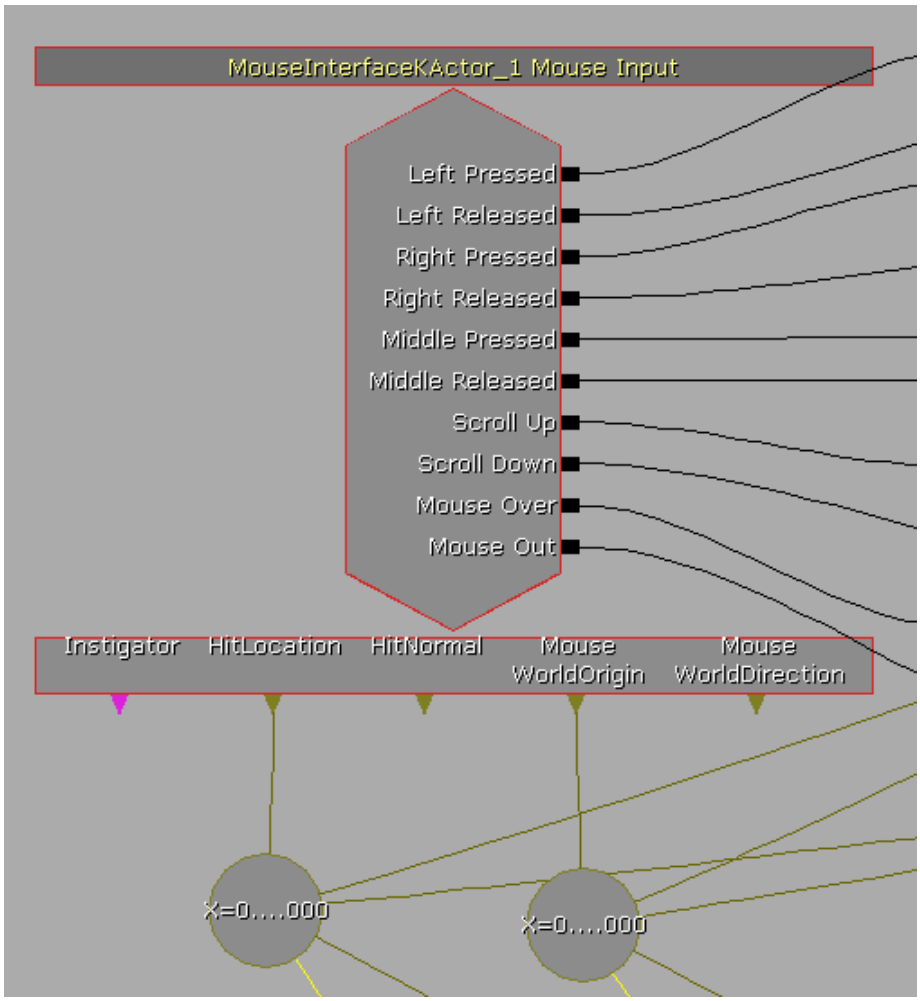


Figura 74

En el ejemplo, he colocado varios modelos estáticos con forma de estatuas, definidos como *MouseInterfaceKActor*. En *Kismet*, se asocian las coordenadas del cursor, y se puede crear una acción para cada posible evento de ratón realizado sobre la estatua. Así, cuando se pone el cursor encima, aparece un mensaje de consola indicándolo, así como

con todas las posibles acciones realizables, según se puede ver en la figura 75, y con más detalle en la figura 76.

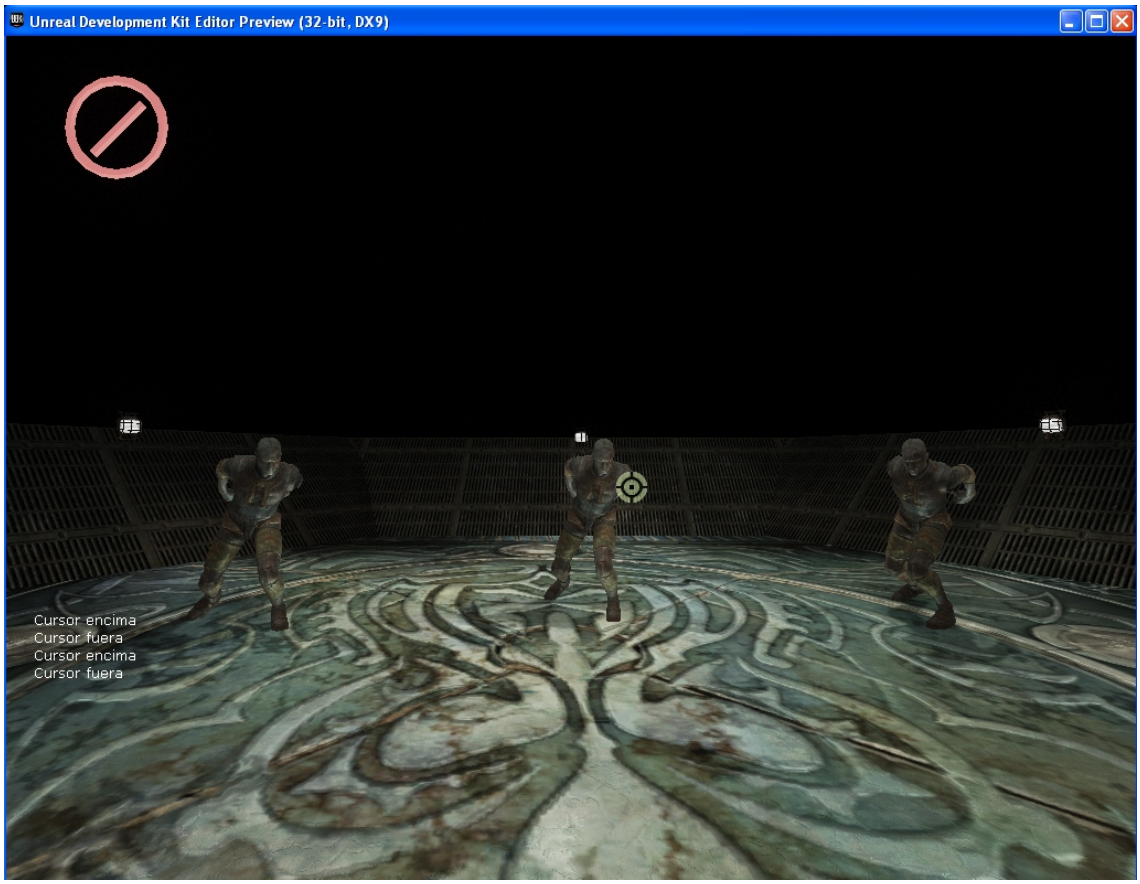


Figura 75

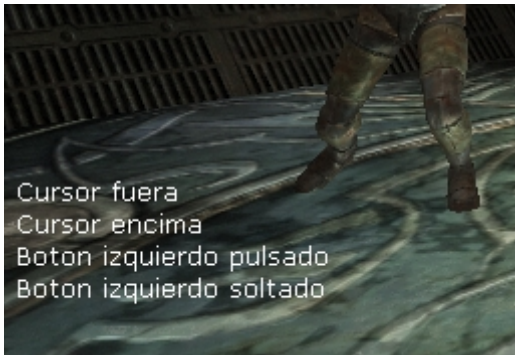


Figura 76

Considero que esto puede ser interesante en una visita virtual, como un servicio informativo contextual. Imaginemos que durante nuestra visita, habilitamos el cursor de ratón y al ponerlo encima de una puerta, vitrina, etc. nos aparece en pantalla un texto explicativo sobre qué es lo que estamos señalando.

La guía para hacer este modo de juego está aquí: **[17]**

Edificio

El propósito era crear un edificio para probar las capacidades arquitectónicas que ofrece UnrealEditor. Este edificio cuenta con 4 plantas, escaleras, columnas, y numerosas puertas y ventanas. En la figura 77 podemos ver el edificio en modo esquemático, por delante y por detrás.

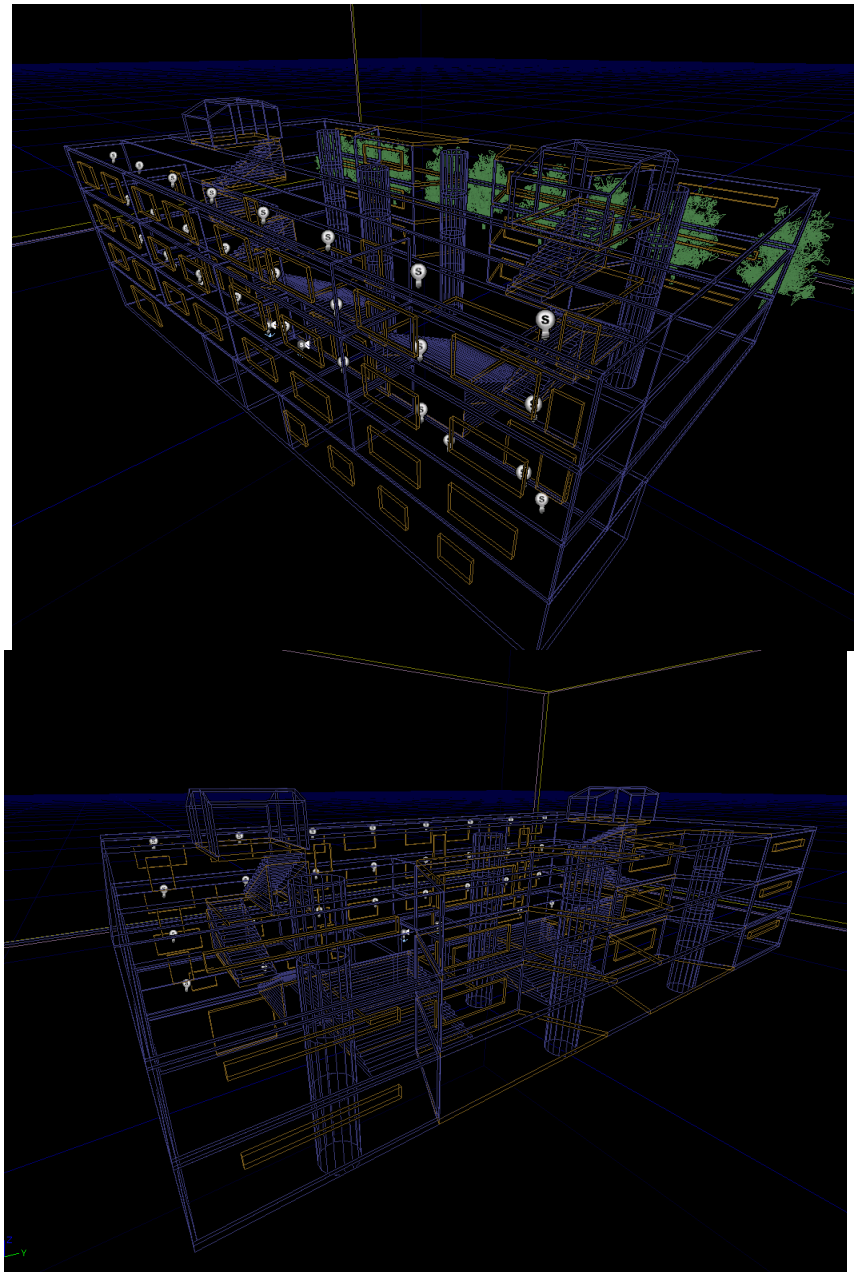


Figura 77

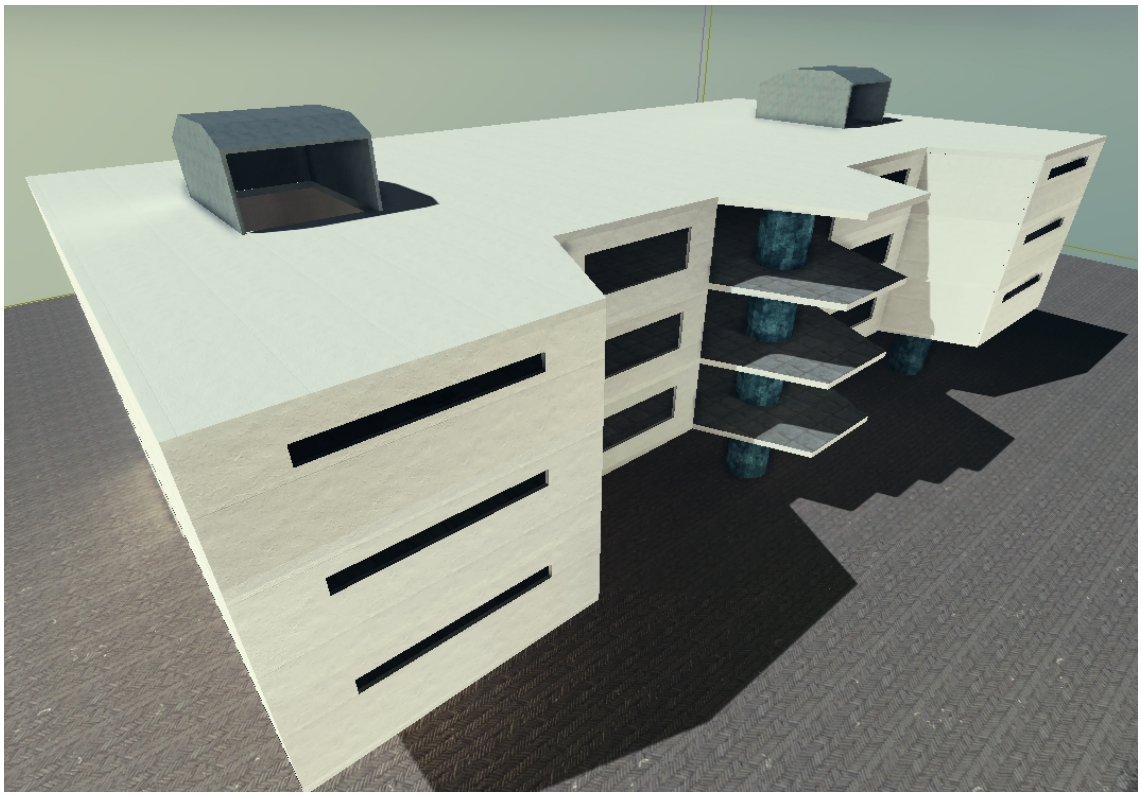


Figura 78

Figura 79

Se ha intentado que no sea un edificio demasiado sencillo, y que sus salas sean numerosas, ya que nos interesa el crear un edificio que tenga cierto grado de complejidad. He buscado una carga poligonal considerable. En las figuras 78 y 79 se aprecia el edificio con una iluminación y con materiales aplicados a las superficies.
En última instancia, decidí añadir modelos de árboles en el exterior, así como una iluminación trabajada y varios efectos atmosféricos y postproceso, para embellecer la escena (figura 80).

Figura 80

La creación de un edificio como este lleva unas 6 horas, y no es difícil habiéndose introducido en UnrealEditor. No obstante, no he añadido en el modelados estáticos para detallar, cosa que es importante para tener un edificio realista, (como puertas, cristales, barandillas, mesas, sillas, lámparas…) No he querido complicarme en ese aspecto pues la finalidad de este mapa era simplemente la de crear una estructura.
Si queremos hacer un edificio como los profesionales, se puede decir que podemos prescindir de las herramientas de modelado de UnrealEditor, que lo hagamos en 3DStudio o similar, y lo importemos a UnrealEditor. Siempre tenemos la opción de usar UnrealEditor para hacer estructuras básicas y *ensamblar* todos nuestros componentes. En cualquier caso, seguramente necesitaremos un artista 3D experimentado.

Todos estos mapas se anexan en el DVD de la memoria por si se quieren examinar con detalle.

ESCUELA DE INFORMÁTICA DE LA UPV

El propósito de este capítulo es dar a conocer cómo con el editor Unreal se puede construir algo que sea reconocible como un edificio real. Como se mencionó en la introducción, el UDK cuenta con posibilidades arquitectónicas. El usuario podrá reconocer claramente el edificio de la Escuela de Informática, y en última instancia sentir que camina virtualmente por su estructura. Además es interesante desde el punto de vista informativo. Por ejemplo, un estudiante que nunca ha estado o visto su futura facultad, pero planea hacerlo, puede tener una visita previa que le ayude a familiarizarse con el que va a ser su edificio.

Desarrollo

El proceso de creación de la escuela pasa por una exhaustiva etapa de planificación. Queremos un edificio lo más parecido posible, y lo fundamental por tanto es asegurarse de que hay un material del que sacar los datos necesarios para mantener en todo momento las **proporciones y medidas** del edificio. Además es importante planificar una estrategia de construcción, ya que cada edificio puede tener una estructura fundamental distinta. La Escuela de Informática (EI a partir de ahora) posee una estructura con cierta complejidad. No obstante, las distintas plantas poseen algunos patrones similares entre ellas. Así pues he decidido comenzar creando el contorno de la planta baja, para después replicarlo y modificarlo convenientemente, para copiarlo como plantas superiores. (Figura 81)

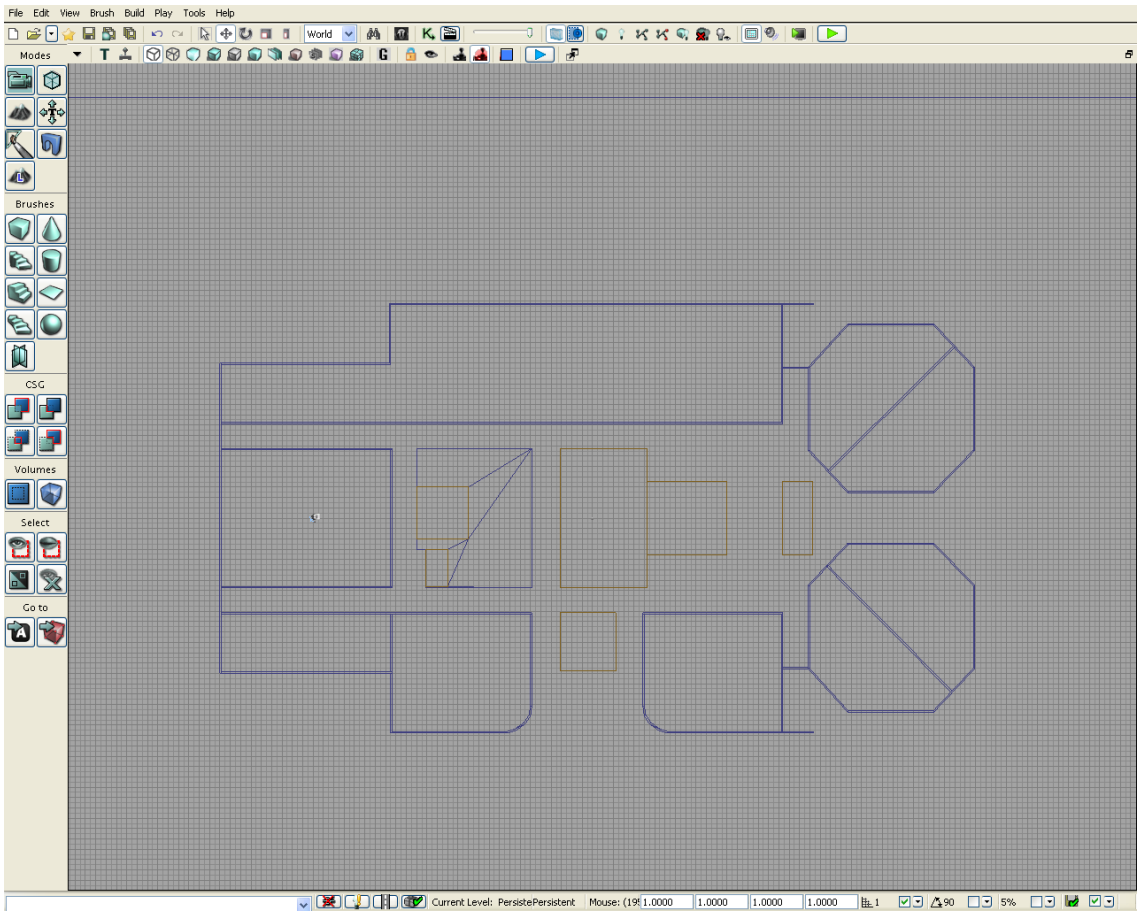


Figura 81

El contorno no se ha hecho a ojo, si no que se ha intentado estimar las medidas lo mejor posible. En este caso, no he podido disponer de material suficientemente preciso para modelar las proporciones con exactitud, si no que en ocasiones he tenido que recurrir a estimaciones. Me he basado en los planos disponibles en la Web de secretaria virtual de la EI. (Figuras 82, 83, 84 y 85)

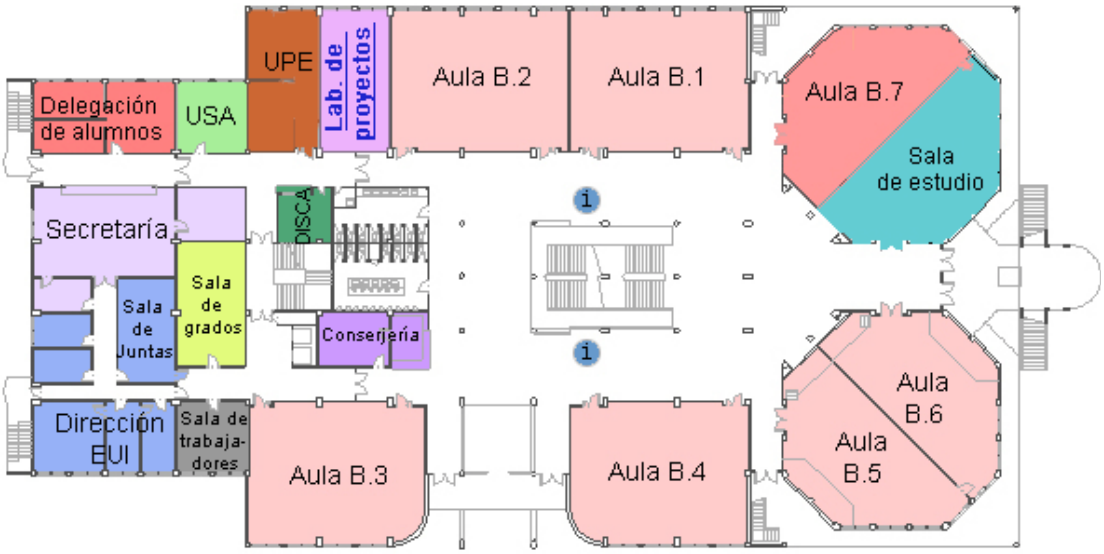


Figura 82

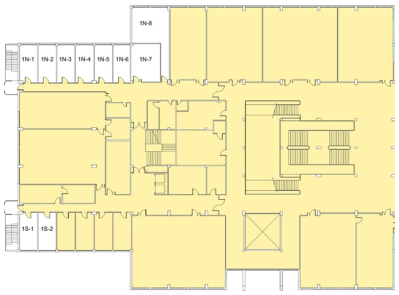


Figura 83

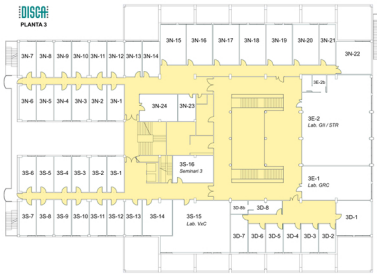


Figura 84

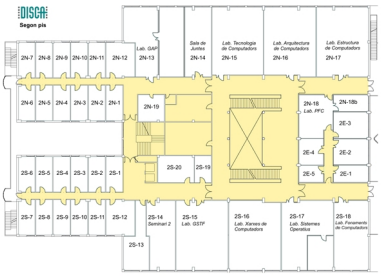


Figura 85

Estos planos ofrecen un esquema de la proyección ortográfica de la base de la escuela para cada planta. Sin embargo muestran una estructura interna ya desfasada debido a remodelaciones, por no mencionar que no describen las dimensiones reales de los muros. He tenido que deducirlas por mi mismo convirtiendo las dimensiones relativas del mapa a dimensiones reales. Por otro lado no he dispuesto de los mapas de las perspectivas de perfil, lo cual hubiese ayudado.

Una cosa que hay que considerar es el tamaño de las unidades en el Editor Unreal. En el editor se usan las UU (*Unreal Units*). Una unidad Unreal equivale a 2 centímetros. En base a esto es fácil estimar los tamaños iniciales de las estructuras. En mi caso, tuve que hacer un par de intentos previamente para ajustar las dimensiones de mi edificio a una escala realista. Estos intentos, así como todo ese trabajo de planificación supusieron una importante cantidad de horas que se invirtieron en pruebas de "ensayo y error". Hay que

tener en cuenta que si se comienza a construir con unas dimensiones mal proporcionadas, **el reescalado posterior como forma de corrección no es una solución**, debido a que el grosor de los muros crecería en la misma proporción, y se verían deformados de forma inconveniente, además que dada la gran cantidad de polígonos una operación masiva de reescalado sobre todos ellos afectarían enormemente a la estabilidad del programa.

Así pues, y tras varias pruebas, se estimaron estos parámetros fundamentales para el comienzo de la construcción de las estructuras:

-Altura de planta (sin techos): 240 u.

-Grosor de los techos: 56 u.

-Grosor de las paredes: 8 u.

Una vez creada la planta baja, se procede a crear su techo correspondiente, que se tiene que corresponder con el contorno de la propia planta. Después de esto, se copia y pega tal planta para crear una replica, que será la planta siguiente, y se modificará según convenga. Para ayudar a la edición, he coloreado los *brushes* correspondientes a cada planta de un color distinto. (Figura 86)

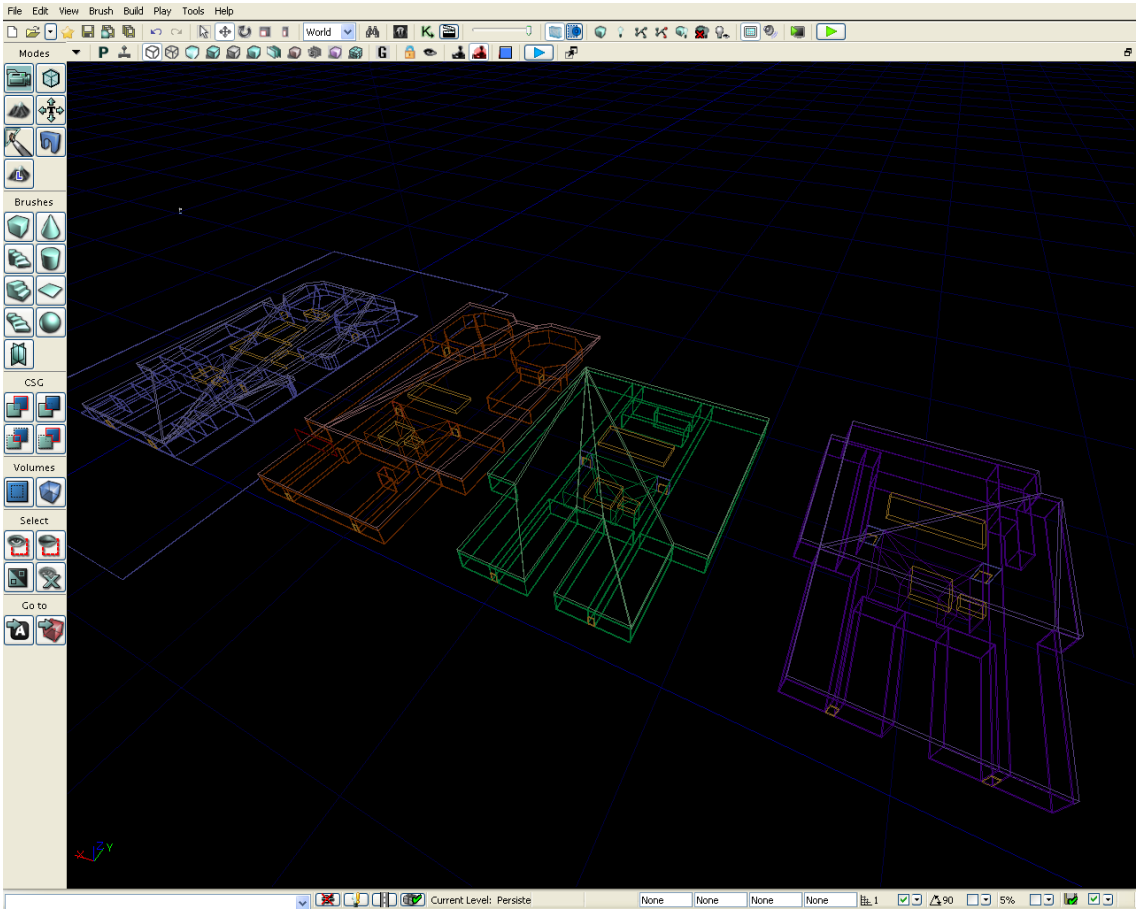


Figura 86

Cuando por fin acabamos la estructura fundamental, llegamos a la etapa de modelado de objetos mas detallados, que son las escaleras, las columnas y los muros de agujeros y escaleras (Figura 87). Es aquí donde hay que puntualizar de nuevo algo importante: En este proyecto, la EI se ha modelado íntegramente en el Editor de Unreal. El editor Unreal da resultados aceptables para objetos de poco detalle, pero si buscamos un acabado profesional en el edificio, aquellos objetos de mayor detalle convienen hacerlos en Maya o en 3DStudio, e integrarlos en el Editor Unreal. No solo por motivos estéticos, si no también de rendimiento, ya que he podido comprobar que al editor se le resienten las altas cargas de polígonos creados con los *brushes*. Además ciertos objetos requieren de una texturización especial mas detallada, algo que solo se puede conseguir con programas de edición 3D y exportándolos como *static meshes*.

Así pues, en mi opción, estos son los objetos que se deberían crear a parte e importar en UDK como *static meshes*: Escaleras (internas y externas) con sus barandillas, columnas (por lo menos aquellas que sean cilíndricas, necesitan una buena cantidad de polígonos para conseguir el aspecto de redondez), lámparas, mesas, plantas, taquillas, estufas, y un largo etcétera que termina donde deseemos que llegue el nivel de detalle de la escuela virtual.

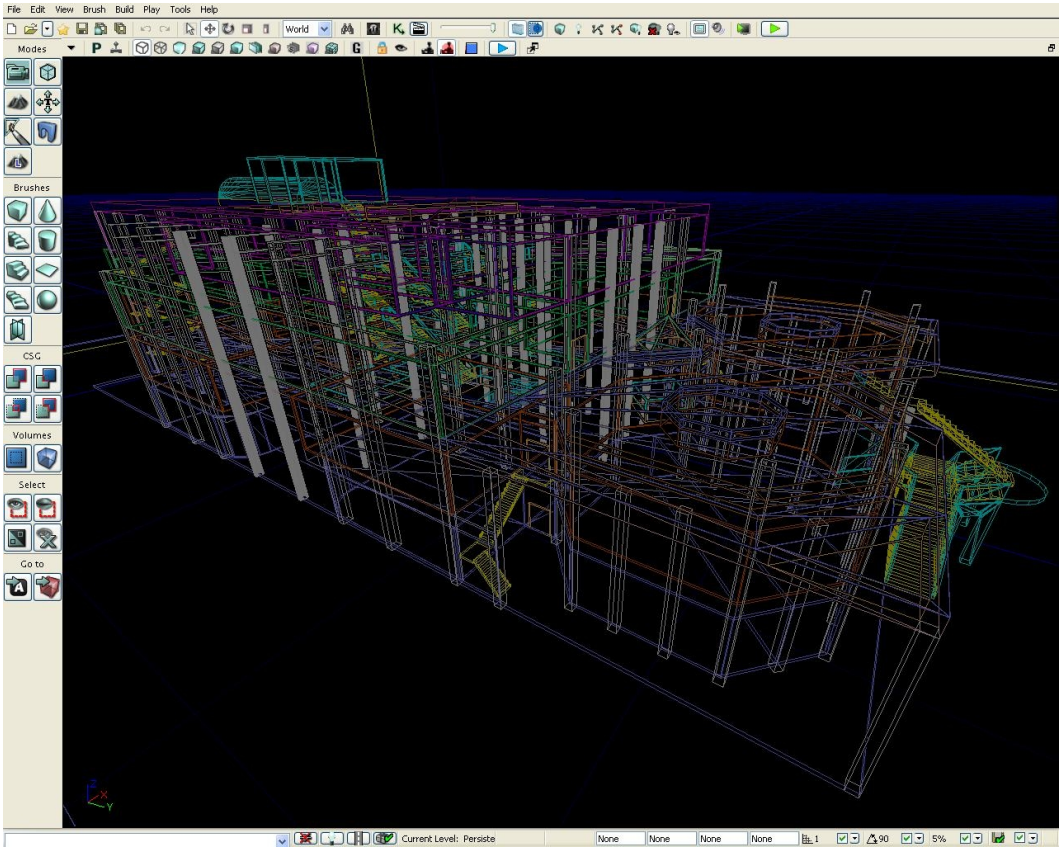


Figura 87

A continuación se han creado todos los materiales necesarios para dotar de las superficies del edificio del aspecto que deben tener. Las texturas usadas se han obtenido gracias al trabajo de dos alumnos de la asignatura de IMD, que realizaron un trabajo similar hace unos años.***[18]** Ellos se dedicaron a capturar fotografías de superficies del edificio, y trataron de que tuviesen una iluminación difusa (requisito necesario para crear un material). Sin embargo**, en UDK las texturas han de convertirse en materiales para poderse aplicar a una superficie**. La ventaja de los materiales es que permiten filtrar múltiples texturas a través de distintos canales. Así pues he aprovechado para crear un mapa de normales y un mapa de especulares de la mayoría de las texturas, y de esa manera crear unos materiales realistas que responden a la iluminación. Esto se puede conseguir con muchos programas, como *Phpotoshop*, pero yo, he utilizado un programa llamado *Crazybump* ***[19]**, que ofrece un uso gratuito durante el periodo de evaluación (30 días). El proceso es sencillo, el programa carga la textura y automáticamente genera distintas imágenes que son los mapeados. (Figura 88)

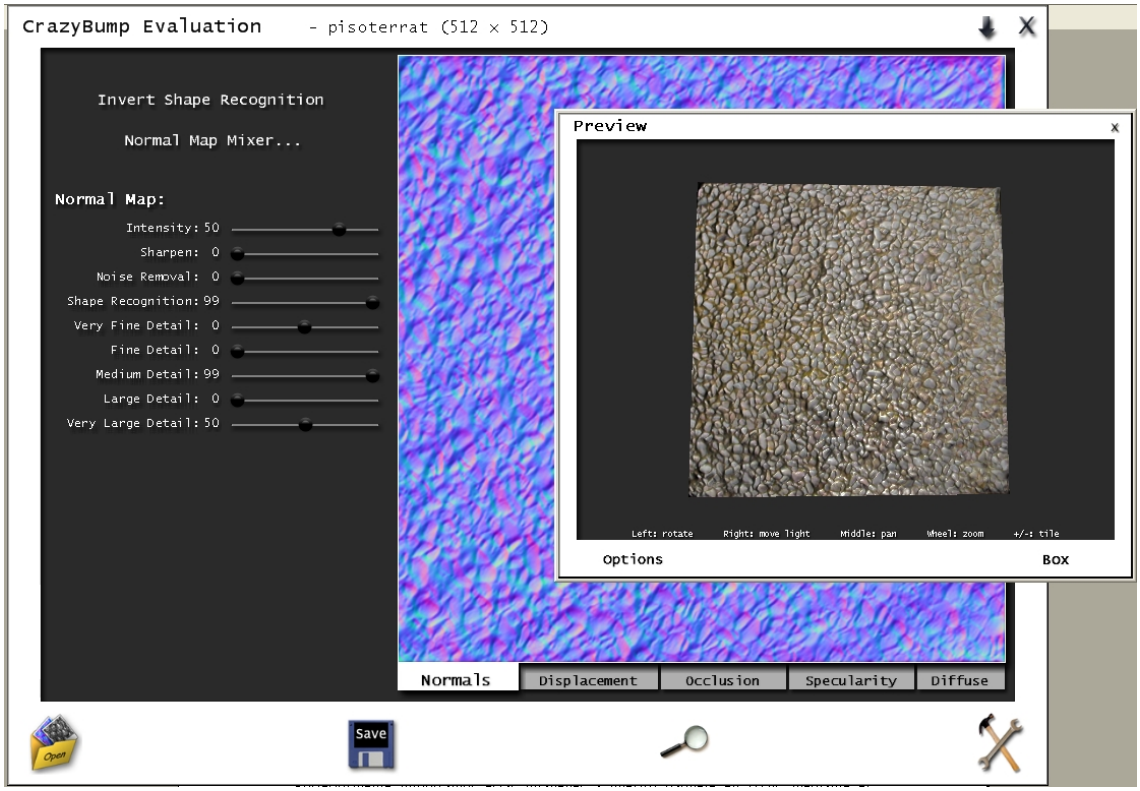


Figura 88

Posteriormente importamos estas imágenes a nuestro paquete en UDK mediante el *Content Browser* y con el *Material editor* configuramos el material. (Figura 89)

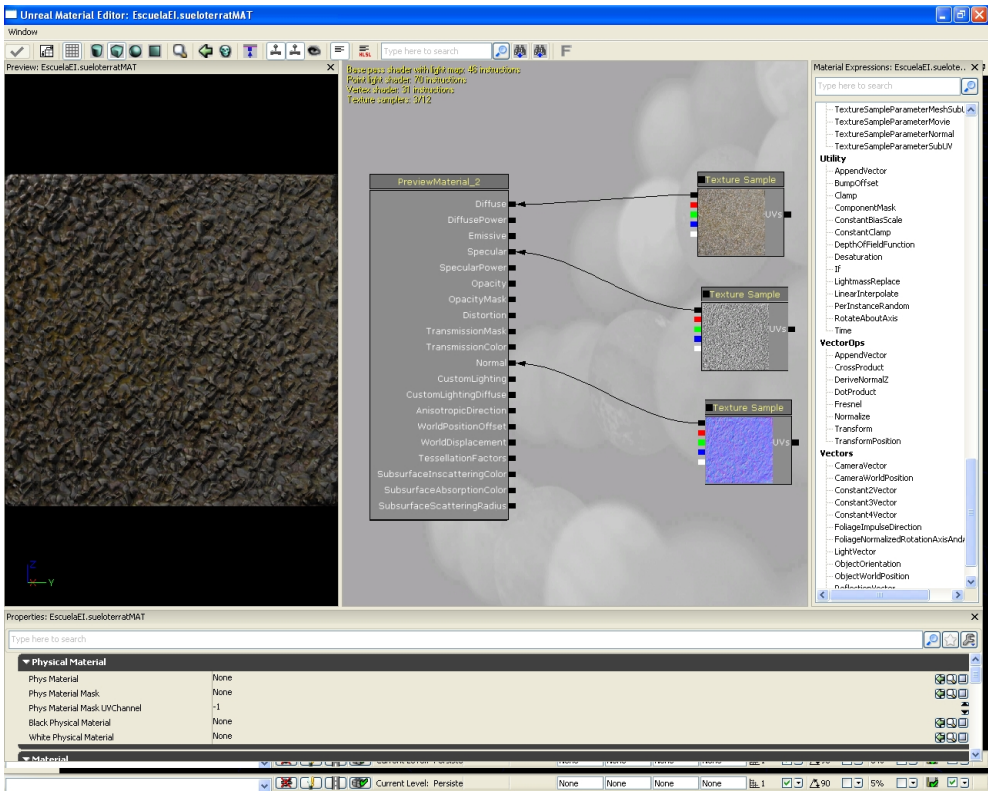


Figura 89

Así pues vamos aplicando los materiales a muros, suelos escaleras y techos. (Figura 90)

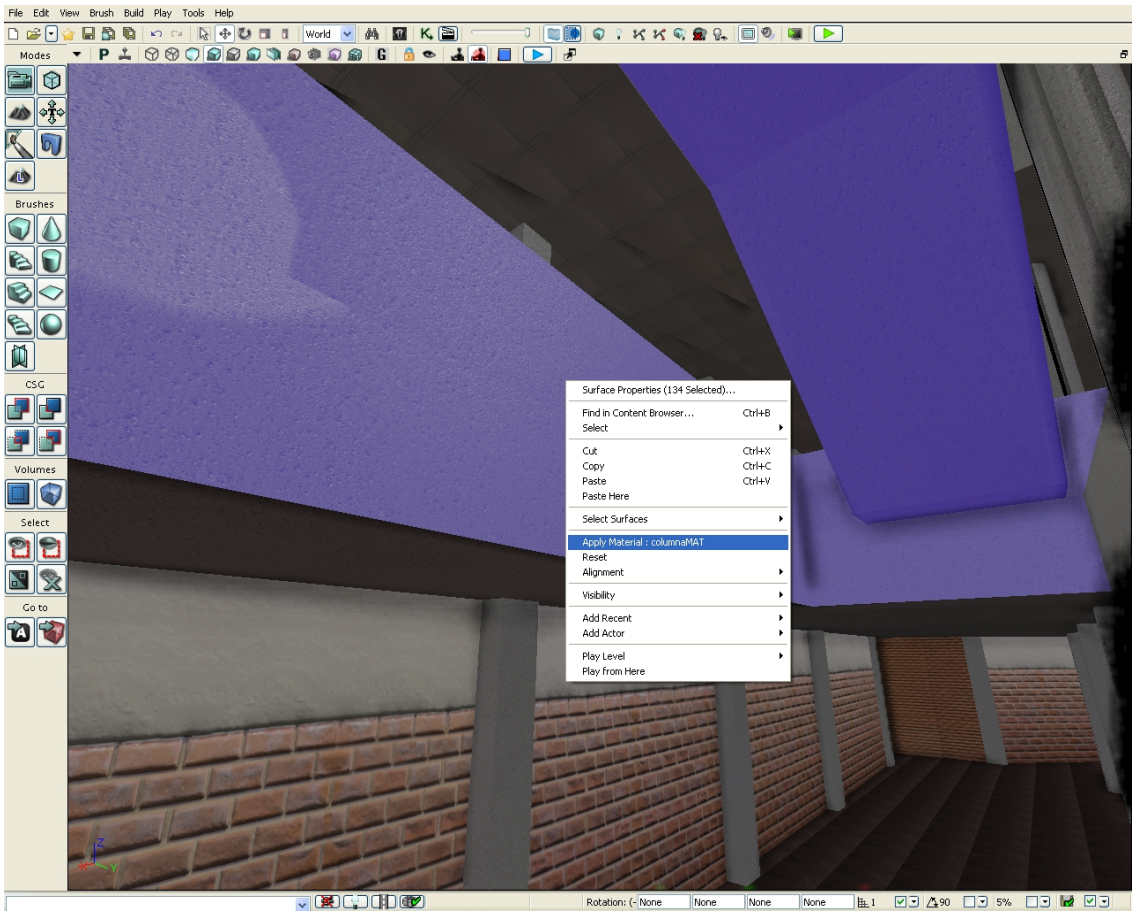
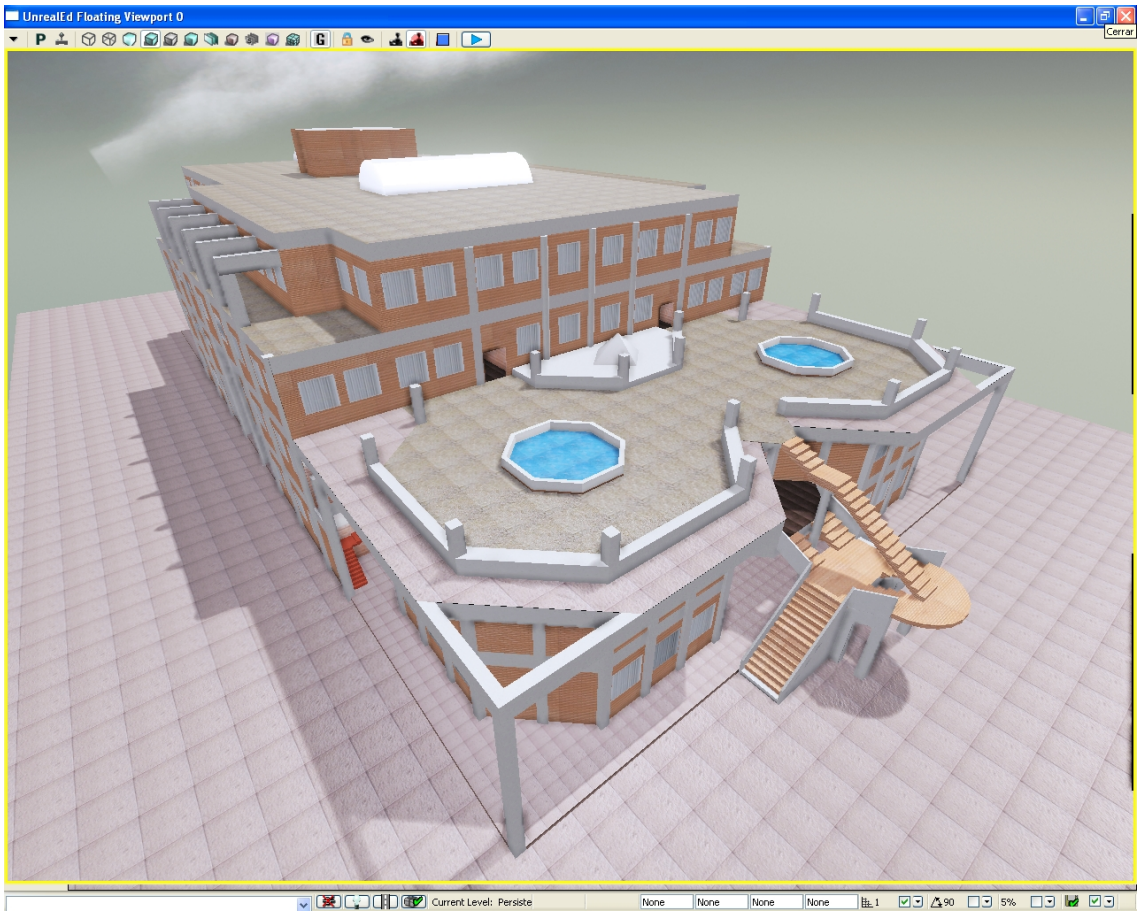
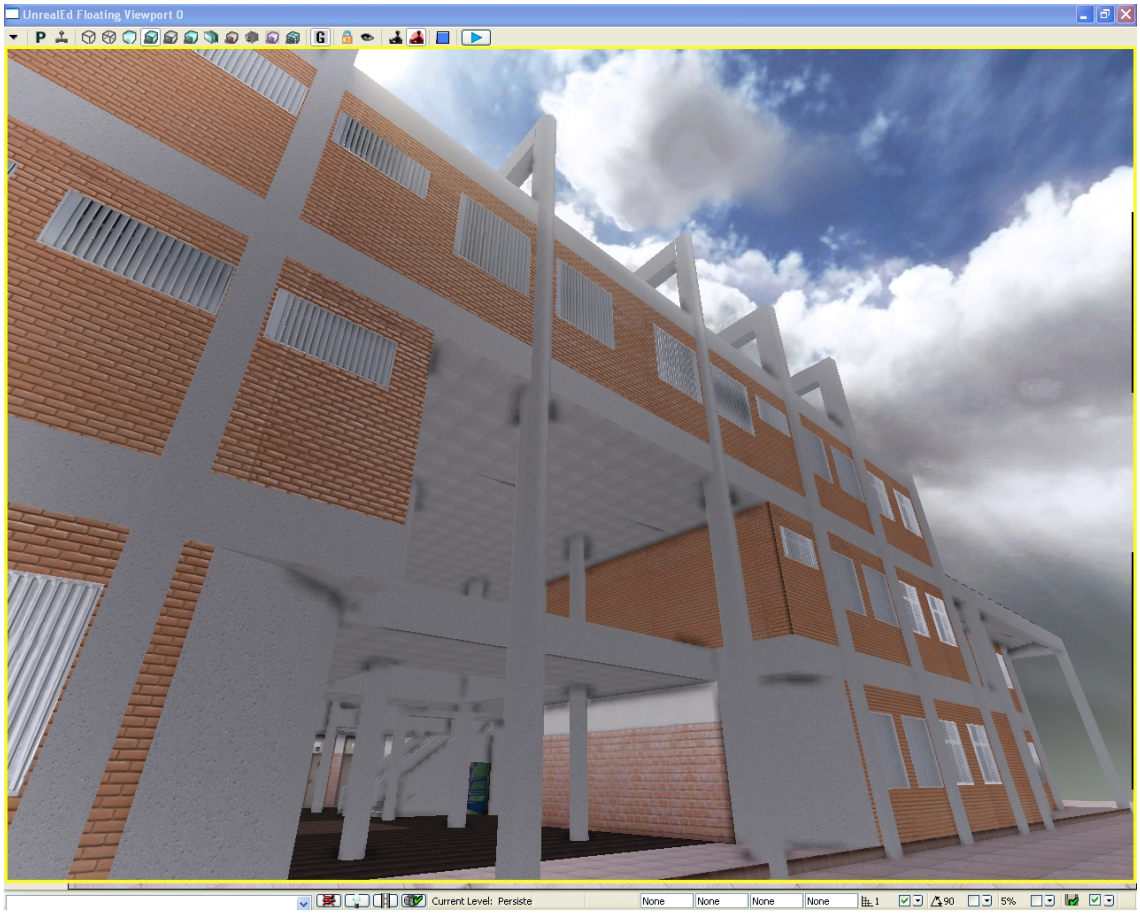


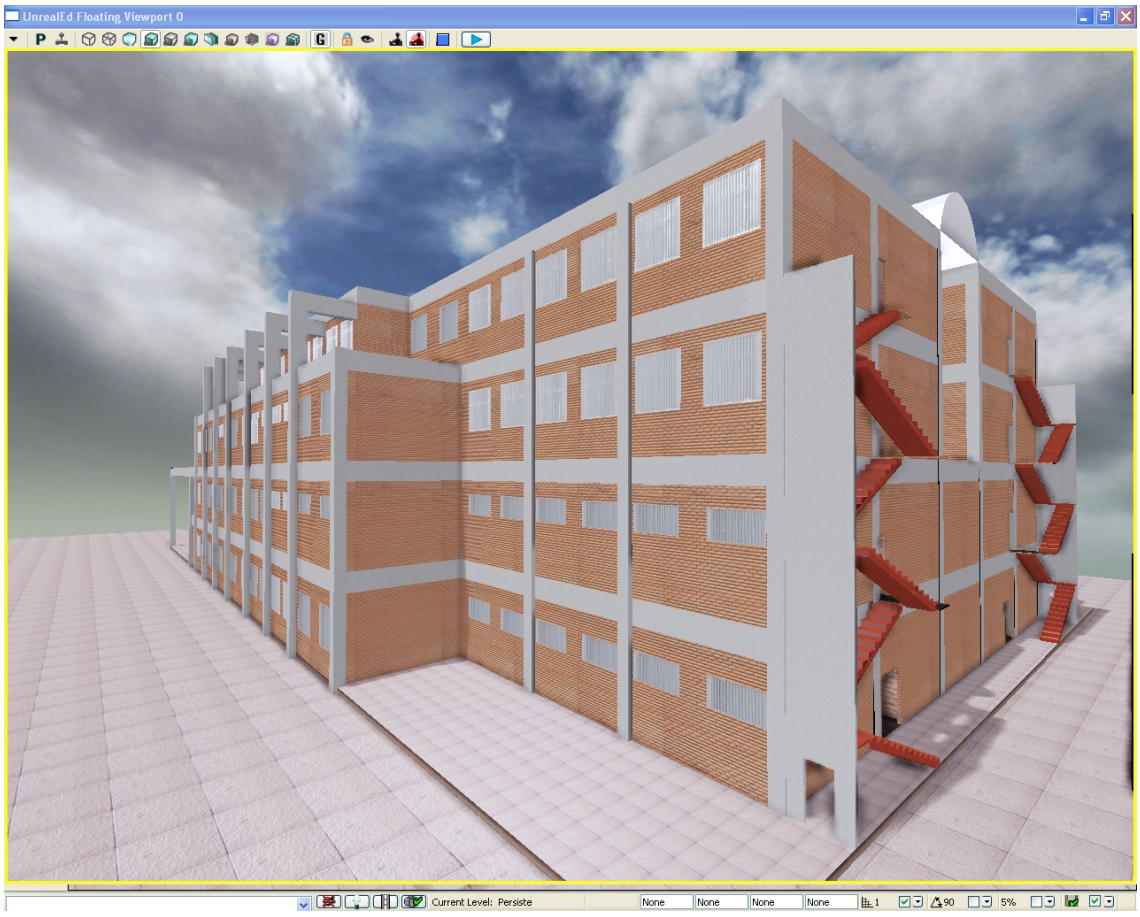
Figura 90

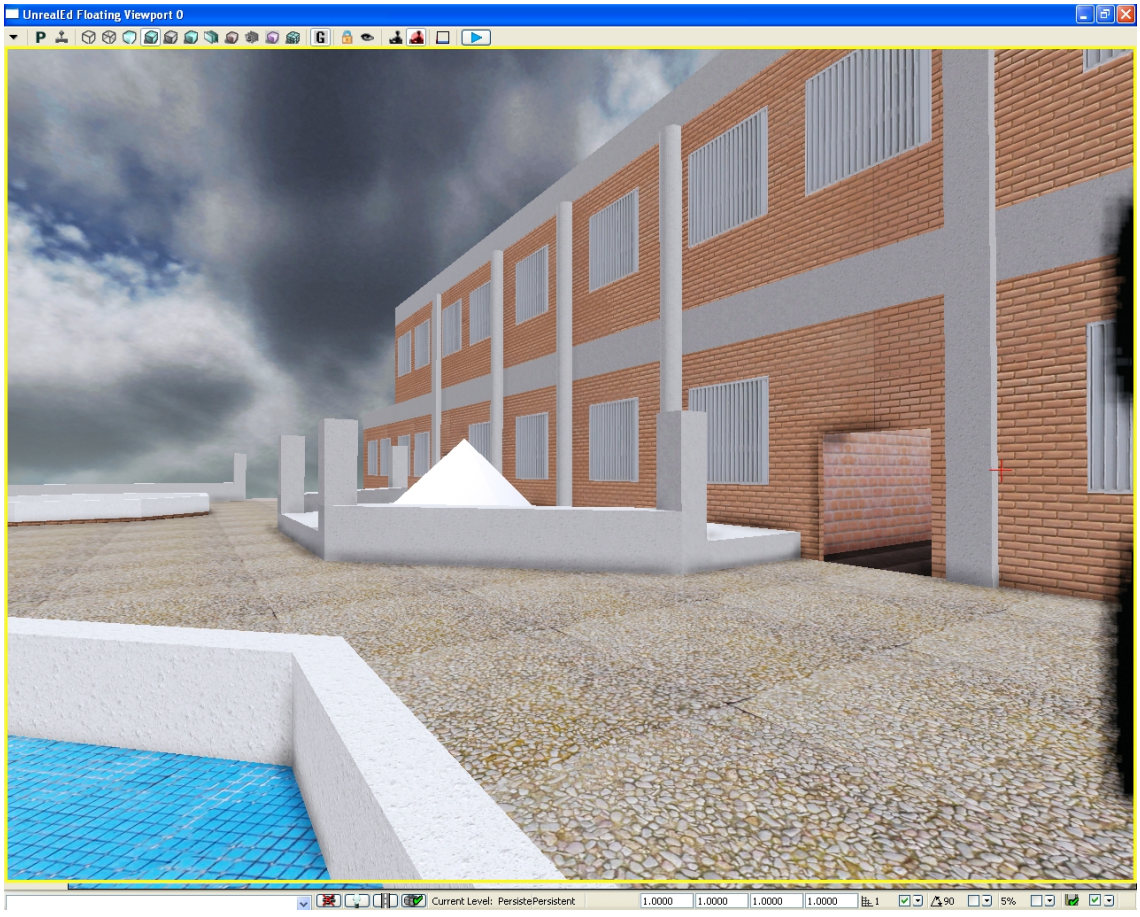
Por ultimo tenemos que colocar ventanas y puertas. Para esto, vamos a utilizar las texturas que tenemos, sin embargo, no las usaremos como materiales, si no como otro tipo de objetos llamados *decal material*. Esto se puede traducir como calcomanías. Estos materiales funcionan de forma distinta, ya que no se aplican ocupando toda la superficie, si no que proyectan el material solo en el área que éste ocupa. Esto hace que sea idóneo para puertas y ventanas. Muy importante: para construir una calcomanía, la imagen no debe ir al canal difuso, si no al canal emisivo, de lo contrario las calcomanías no se verán correctamente, pudiendo mostrarse con sombras o incluso negras.

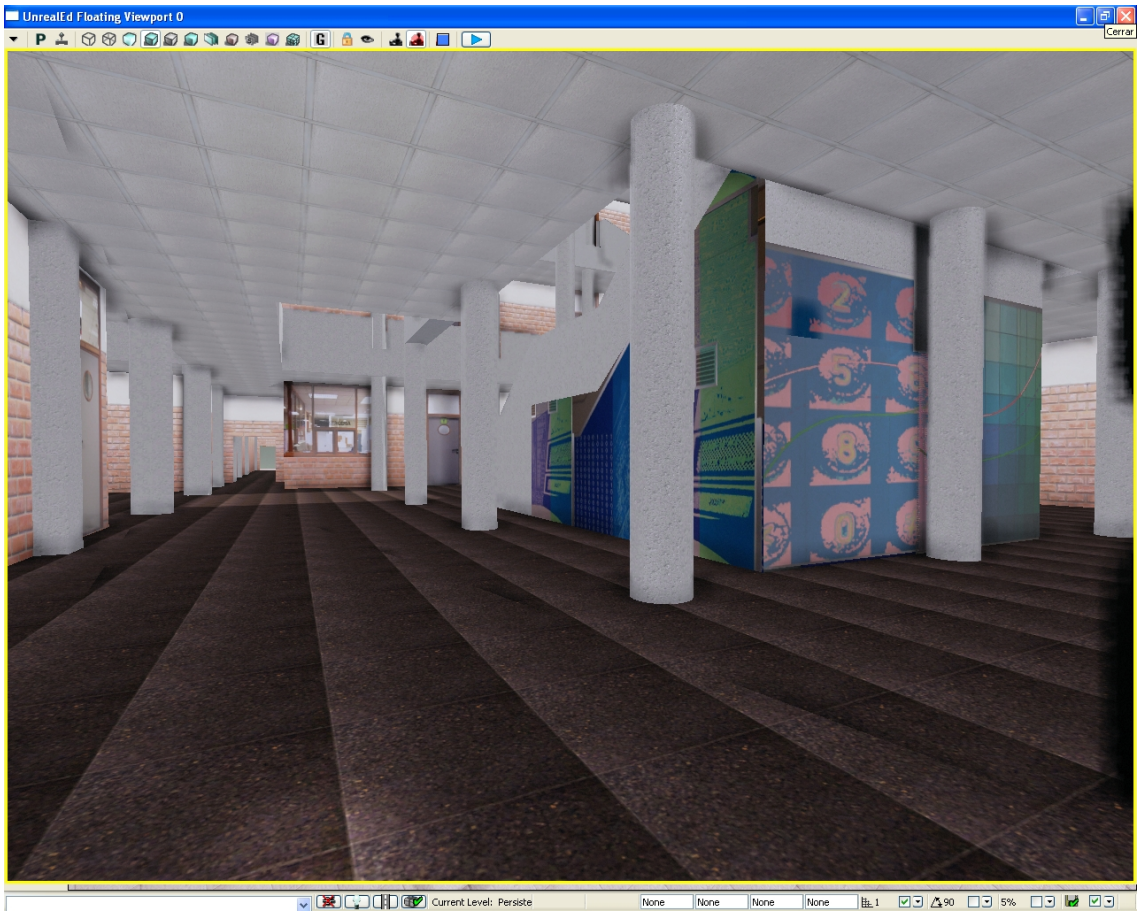
Solo falta ajustar la luz, crear un ambiente de post proceso para mejorar el color, y tenemos listo este prototipo de visita a la EI. Véase la siguiente colección de capturas de la EI virtual. (Figuras 91 - 98)

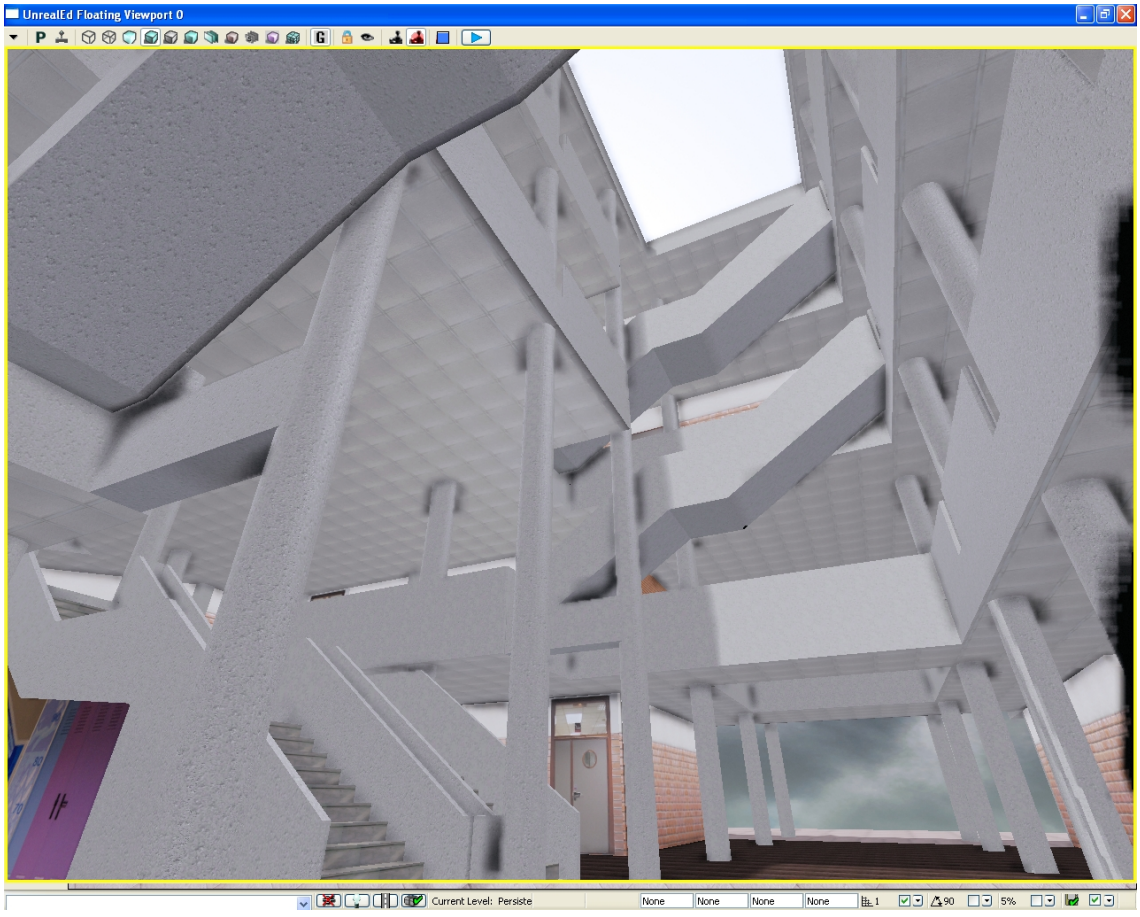

UnrealEd Floating Viewport 0
Cerrar
Current Level: Persiste
None
None
None
None

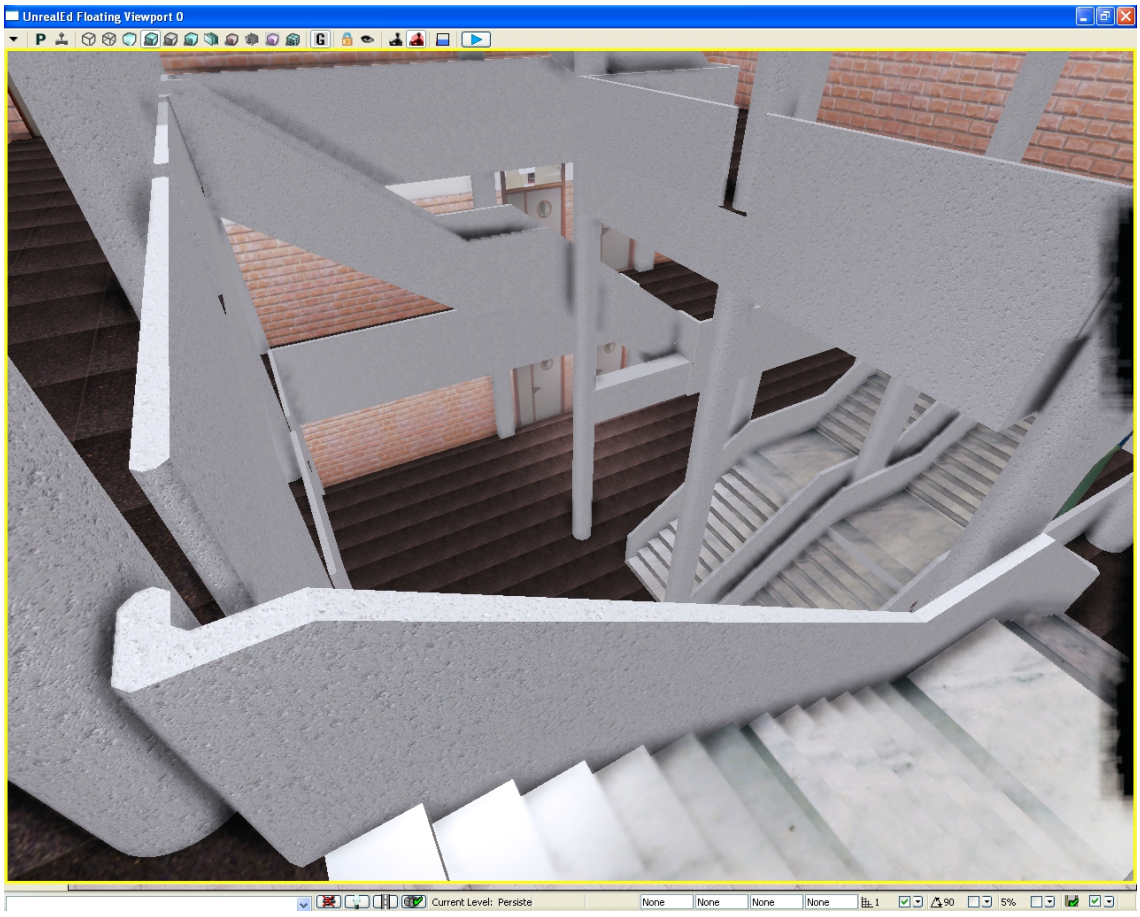

UnrealEd Floating Viewport 0
Current Level: Persiste
None
None
None
None

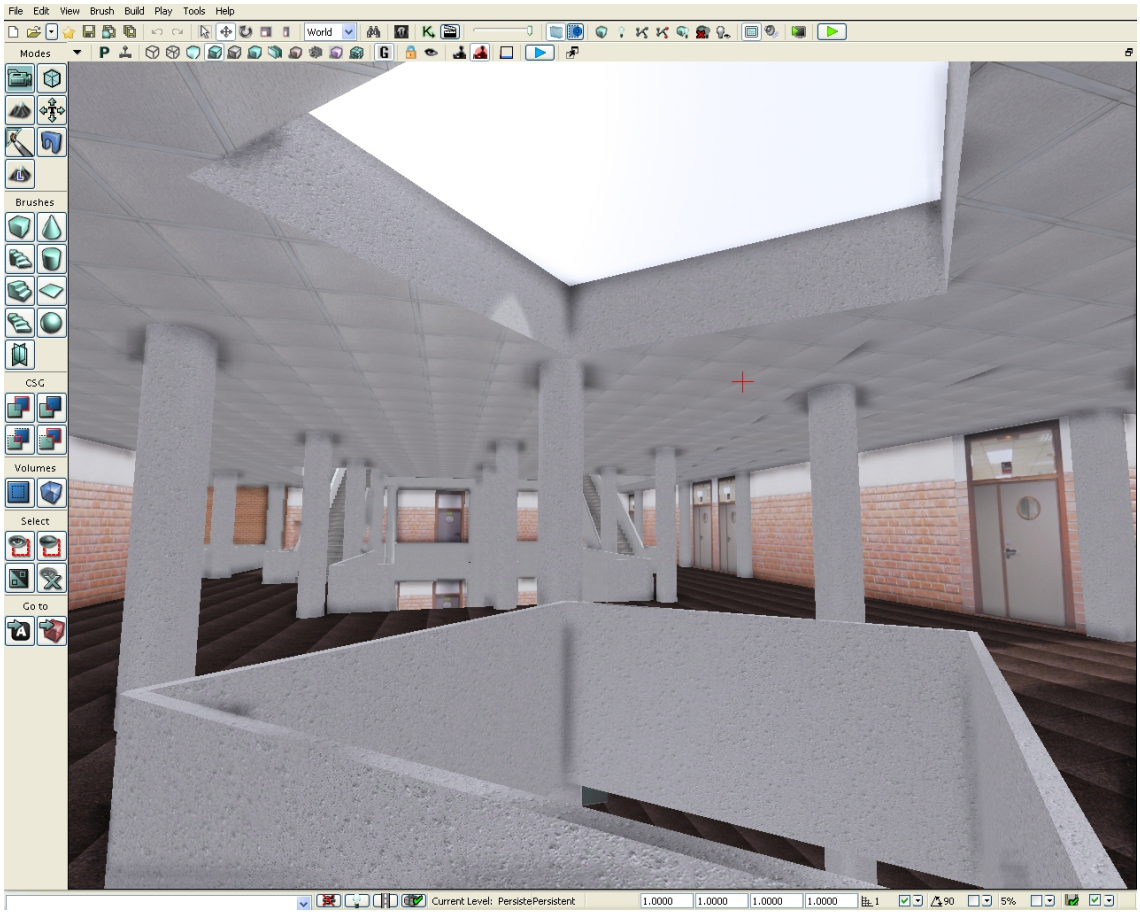

UnrealEd Floating Viewport 0
Current Level: Persiste
None
None
None
None


UnrealEd Floating Viewport 0
Current Level: PersistePersistent
1.0000
1.0000
1.0000
1.0000


UnrealEd Floating Viewport 0
Cerrar
Current Level: Persiste
None
None
None
None


UnrealEd Floating Viewport 0
Current Level: Persiste
None
None
None
None


UnrealEd Floating Viewport 0
Current Level: Persiste
None
None
None
None


File Edit View Brush Build Play Tools Help
World
Modes
Brushes
CSG
Volumes
Select
Go to
Current Level: PersistePersistent
1.0000
1.0000
1.0000
1.0000

Cabe añadir que al ser el editor Unreal una herramienta poco especializada en el modelado detallado, conforme se van añadiendo polígonos, si la escena llega a hacerse muy compleja, el programa se vuelve muy inestable, y en ocasiones puede fácilmente fallar y cerrarse. Es importante guardar a menudo. La estimación de tiempo de desarrollo de este mapa está en torno a las 50 - 60 horas, teniendo en cuenta factores como la planificación, las pruebas fallidas, con su consecuente recomienzo, y el hecho de que en el proceso he tenido que aprender numerosos atajos y técnicas de modelado. Ciertamente, la velocidad de producción al comienzo era muy inferior a la conseguida al final, ya que he comenzado con bastante inexperiencia. Es importante tener en cuenta que **alguien que trabaje para llevar a cabo este proyecto puede requerir muchas horas o pocas en función de su destreza con el modelado tridimensional.**

Trabajos futuros

En este proyecto, se ha diseñado la EI como prototipo y desde un punto de vista demostrativo. El desarrollo de niveles es muy multidisciplinar y conlleva un trabajo extenso. En algunos aspectos, el edificio no está acabado, o algunas partes deberían rehacerse.

En el modelado, como se ha anotado anteriormente, se aconseja el uso de una gran cantidad de modelos estáticos. Esto implica un importante trabajo en herramientas 3D como 3DStudio. No se han aplicado las lámparas ni los focos de luz interiores, la iluminación difusa en la EI es bastante grande. Tampoco se han modelado el interior de los despachos y las aulas. De todas formas, es un asunto discutible el hecho de incluir todos los detalles para todas las salas, o sólo recrear aquellas más importantes como las aulas y ciertas estancias.

También encuentro aconsejable rehacer muchos de los materiales, ya que aunque los actuales cumplen bien su función, algunos de ellos son de poca resolución o no muestran una buena iluminación difusa, como es el ejemplo de las baldosas del suelo, que reciben una iluminación irregular y esto se nota al replicarlas sobre la superficie.

Por último añadir que sería interesante trabajar en una interfaz de visita, esto es, crear un modo de juego UDK que esté adaptado a nuestros objetivos como programa de cara al usuario. Esto implica un diseño del *juego*, y un trabajo en scripts, crear menús de usuario, y hacer que pueda disponer de opciones de navegación e informativas. Este

seria el paso posiblemente más complejo o que más investigación requeriría debido a la profundidad de *UnrealScript*.

No puedo asegurar la cantidad de horas estimadas en este ultimo aspecto del desarrollo debido a mi poca experiencia en este campo, pero aun así puedo hacer una estimación en torno a las 200 - 300 horas, por supuesto siempre depende el factor de lo accesibles que sean los recursos y la experiencia.

CONCLUSIONES

Después de todas las bondades del UDK enumeradas hasta ahora, es preciso enfocar en qué aspectos nos va a resultar realmente práctica herramienta, o con qué conceptos de jugabilidad podemos explotarla, para llevar a cabo una visita virtual a un edificio.

En primer lugar, he de dejar claro que la recreación virtual de un escenario tridimensional que sea una **réplica** de uno real, no es nada sencilla. Mientras que crear un escenario inventado no es excesivamente difícil, crear una réplica conlleva un procedimiento y una metodología exhaustiva. De entrada, sólo con que el edificio tenga cierta complejidad, se hace bastante necesario contar con planos del edificio, tanto de planta como de los perfiles. Además hay que **fotografiar todos los puntos clave**, que son aquellos en los que el modelado de la estructura o la fachada puede ser especialmente complejo.

En cuanto a la captura de texturas para los materiales, he comprobado que si se quieren hacer a partir de fotografías, requieren de unas condiciones de iluminación especiales, una cámara de gran calidad, y, en cualquier caso, un importante retoque en algún programa de edición de imagen posteriormente. Una alternativa puede ser, con la suficiente dedicación, dibujar dichas texturas a partir de cero.

Por tanto, **si se desea un resultado detallado sobre un edificio más o menos complejo,** se asume que aquellos que tengan que trabajar en la creación del mapeado tienen que ser artistas gráficos experimentados. Tanto en el modelado como en la texturización y diseño de todos los elementos.

Y en vista de la envergadura que puede tener el proyecto, dada la cantidad de detalle y diferentes disciplinas que conlleva la creación 3D y todas sus etapas, se hace necesario posiblemente un equipo de artistas.

Sin embargo, no estamos necesariamente obligados a crear una réplica *exacta*, si nos conformamos con una aproximación reconocible de lo que es el edificio, recreando las estructuras básicas y colores y formas similares. Todo está en el grado en que queramos explotar la funcionalidad de nuestro proyecto o más bien hacer un alarde de sofisticación y realismo gráfico.

En mi experiencia personal, y con algunos conocimientos básicos previos en modelado y animación 3D (Blender), así como del entorno de los videojuegos, la puesta al día en

el uso de UDK no es precisamente rápida, si no que conlleva una larga etapa de documentación, adquisición de información y aprendizaje. Esto no es tanto por que UDK sea difícil de usar, si no por la enorme cantidad de aspectos y disciplinas que abarca el desarrollo de un proyecto UDK. Así pues, realizar unos tutoriales de iniciación a crear un mapa con *Unreal Editor* puede llevar varios días, o una semana, no obstante aún estaremos a años luz de aprender todo lo relativo a UDK.
Así pues podemos tener en cuenta:

-El esfuerzo de reunir documentación. La Web de udn.epicgames.com ofrece extensas guías y referencias de uso, así como videotutoriales bastante útiles, pero en muchas ocasiones no vamos a encontrar solución a problemas puntuales que nos puedan surgir, y es aquí donde tenemos que buscar en otras páginas y en foros. En muchos casos me ha costado solucionar un problema concreto. Y es aquí donde hay que mencionar un importante inconveniente: la rápida desactualización de la información. La red UDN intenta mantener la información actualizada todo lo posible, pero aún así es fácil que encontremos lugares donde la información está obsoleta por causa de una nueva versión de UDK (que recordemos está en fase BETA). Así pues, cuando creamos haber hallado una solución a un problema, puede ocurrir que ya no sea aplicable en la versión en la que estamos trabajando por ser más actualizada. Esto también se aplica a los libros. Los dos volúmenes de *Mastering Unreal Technology* resultan útiles en la mayoría de los capítulos, pero no nos van a ser útiles en todas aquellas adiciones y modificaciones que se hagan al Kit.
En cuanto al *UnrealScript*, debo decir que es la parte en la que más inconveniente se me han presentado. Mientras que generalmente UDK está bien provisto de tutoriales, no puedo decir lo mismo de *UnrealScript*. He tenido que buscar en webs y blogs particulares, para tener una introducción decente en cómo empezar a crear Scripts.

Cabe hacer una muy importante mención a que **es fundamental el conocimiento del inglés** a un nivel escrito, y muy aconsejable a nivel oral, por que podemos decir que casi la totalidad, por no decir toda la documentación para UDK está en ese idioma.

-La etapa de aprendizaje: Si el desarrollador no está mínimamente introducido en el ámbito de los videojuegos, eso va a implicar un esfuerzo extra en el aprendizaje para el uso del kit. Del mismo modo, si no sabe manejar herramientas de diseño 3D y relativas,

lidiará con el mismo problema. En cualquier caso, Unreal es un motor de videojuego con una arquitectura propia, y UDK cuenta con herramientas únicas y en abundancia, por lo que una profunda introducción en Unreal requerirá un tiempo importante de aprendizaje, aún tratándose de alguien introducido en el desarrollo multimedia.

-En cuanto al desarrollo, salvando que los desarrolladores no tengan problemas usando UDK y que los programadores de script estén al día, tal vez haya que recalcar en lo importante que es el tener un diseño del juego: organizar de principio a fin qué va a ver el usuario, qué elementos van a entrar en escena, y qué recursos vamos a utilizar.
Anotar que en el UDK, todo proyecto está organizado en carpetas, y cada tipo de trabajo que utiliza (podríamos llamarlo *subproyectos*) se encuentra en una carpeta distinta (por ejemplo, las interfaces se crean en una carpeta, los mapas va a otra carpeta, los scripts a otra…) A la hora de trabajar con UDK, se requieren permisos de escritura, así como de sobreescritura para aquellas carpetas correspondientes al tipo de trabajo que esté desempeñando el desarrollador.
Otro detalle a mencionar son algunos **problemas de inestabilidad** puntuales. Bajo algunas condiciones, Unreal Editor llega a dejar de responder y lanza al escritorio. Asumo que es consecuencia de estar trabajando con un programa en fase BETA y bajo continuo mantenimiento. UnrealEditor realiza guardados de seguridad cada cierto tiempo para prevenir perdidas de los mapas.

En otro orden de cosas, hay que abrir además el debate de hasta qué punto puede ser el uso de Unreal práctico para el propósito de nuestro proyecto. Un proyecto Unreal, terminado, y empaquetado, e instalado en el sistema del usuario, puede fácilmente ocupar varios cientos de megas. Tengamos en cuenta que el cliente debe instalar en su sistema el motor para que funcione (y tener privilegio de administrador). Unreal es un motor muy potente, con una gran carga gráfica, y esto lo hace ideal para crear videojuegos de gama alta, que pueden requerir un equipo muy potente para el usuario.
Por eso hay que preguntarse si merece la pena desarrollar un proyecto que en principio va a ser para tener un producto accesible, de uso espontáneo y cómodo.
Una interesante alternativa, Unity, es un kit de desarrollo reciente, que se está haciendo popular. Desde mi punto de vista, mientras que Unreal ofrece enorme potencia gráfica, Unity se presenta como un motor no tan sofisticado, pero sí cuenta con algo que

Unreal no tiene, la capacidad de ejecución online por medio de un navegador como plataforma (*Unity Web Player*), que instala un plugin de pocos megas en el navegador.
Además Unity es multiplataforma, y su kit funciona tanto en Windows como en Mac OS. Quizás es interesante decir que *Unity Web Placer* funciona en Internet Explorer, Firefox, Chrome, Safari, Opera para Sistemas Windows, y Safari, Firefox, Chrome, Camino para sistemas Mac OS.
He encontrado un ejemplo interesante de Unity funcionando en navegador, que consiste en una visita virtual a un museo. ***[20]**
Por supuesto, la posibilidad que ofrece Unity para nuestro proyecto requeriría un estudio propio aparte, que es algo que está fuera de lugar en este documento.

BIBLIOGRAFÍA Y REFERENCIAS

Libros:

-Mastering Unreal Technology. Vol. 1. Introduction to level design with Unreal Engine 3. Ed. SAMS.

-Mastering Unreal Technology. Vol. 2. Advanced Level Design Concepts with Unreal Engine 3. Ed. SAMS

Enlaces de interés:

Web oficial de UDK: http://www.udk.com/

Unreal Development Network: http://udn.epicgames.com/Three/WebHome.html

Una Web de tutoriales: http://udkc.info/index.php?title=Main_Page

Blog con guias UnrealScript, Scaleform y otros http://forecourse.com/unreal-tutorials/

Foros de UDK de Epic: http://forums.epicgames.com/forumdisplay.php?f=366

Referencias en el texto:

1: http://www.youtube.com/watch?v=XgS67BwPfFY

2: http://en.wikipedia.org/wiki/Comparison_of_OpenGL_and_Direct3D

3: http://www.devmaster.net/engines/engine_details.php?id=635

4: Mastering Unreal Technology, volumen 1, capítulos 3 y 4

5: Mastering Unreal Technology, volumen 1, capitulo 5, tutorial 5.6

6: http://campagnini.net/2011/04/16/ase-export-for-blender-2-57

7: http://udn.epicgames.com/Three/KismetTutorial.html

8: http://udn.epicgames.com/Three/MatineeUserGuide.html

9: http://www.youtube.com/watch?v=yTVznfRW2wY&feature=channel_video_title

10: http://udn.epicgames.com/Three/UnrealScriptReference.html

11: http://sourceforge.net/projects/uncodex/.

12: http://wiki.pixelminegames.com/index.php?title=Tools:nFringe

13:http://www.disturbedprogrammer.com/index.php?option=com_content&view=article&id=54:third-person-camera-and-gow-camera&catid=43:geek-tutorials&Itemid=63

14:http://www.disturbedprogrammer.com/index.php?option=com_content&view=article&id=54:third-person-camera-and-gow-camera&catid=43:geek-tutorials&Itemid=63

15: http://udn.epicgames.com/Three/UnrealFrontend.html
16: http://udn.epicgames.com/Three/UnrealFrontend.html
17:http://udn.epicgames.com/Three/DevelopmentKitGemsCreatingAMouseInterface.html
18:http://web-sisop.disca.upv.es/imd/cursosAnteriors/2k3-2k4/copiaTreballs/anrigar1/trabajoIMD.xml
19: http://www.crazybump.com/
20: http://obrasocial.bancaja.es/cultura/exposiciones/sorolla-visita.aspx

NOTA: Se desarrolló una tercera edición de *Mastering Unreal Technology*, que fue cancelada previamente a su publicación. Por lo tanto, a día de hoy, no existe forma de adquirir este libro, ni se han dado muestras de que se planee publicar en un futuro.