DEVELOPMENT OF A LARGE SCALE, SMART, MULTI-PURPOSE VIRTUAL ENVIRONMENT (SCIENCES BUILDING OF TUC) USING GAME ENGINE

By

Stefan Petrovski

Submitted to the

Department of Electronics and Computer Engineering

Technical University of Crete

Examining Committee

Dr Aikaterini Mania, Advisor
Dr Aggelos Bletsas
Dr Antonios Deligiannakis

Table of Contents

List Of Figures

Abstract

This thesis explores a method of three-dimensional fully interactive architectural visualization based on a game engine and presents a framework for creating such virtual environments. Through an examination of the tools and the components of a game engine, this thesis shows how a game engine can be used for creating large, smart and interactive Virtual Environments (VEs). The final product of this thesis, along with the research presented here, is a 3D computer application which simulates a real-time Virtual Environment (The Sciences building of TUC). The created application simulates a real-time 3D navigable environment which the user can interact with, navigate in it and view information related to spatial location of the faculty offices, labs or classrooms. Moreover, the user can view data from external sensors which could be streamed in the interactive application dynamically. The goal of the application is to show the results of modern 3D architectural visualization using a game engine, as well as to build functionality that helps the users in navigation and getting to know the university building simulated. In addition to this, many of the classes created can be used by developers to add interesting functionalities to their VEs.

Acknowledgements

I owe sincere thankfulness to my research advisor, Katerina Mania, who made me believe in myself and guided me through the whole process of thesis writing. I am sure that this thesis would not have been possible without her support, understanding and encouragement I felt when working on my project.

I would like to express my gratitude to Prof. Aggelos Bletsas and to Eleftherios Kampianakhs and Kostantinos Tountas for their invaluable help constructing and installing the sensors as well as for the implementation of the software used to send data to the web server.

Finaly, i would like to show my gratitude to my parents, friends and colleagues for their help and moral support. I would like to thank my sister for her patience and her help improving the readability of my thesis.

1 Introduction

The history of video games goes back to 1962 when a young computer programmer from MIT, Steve Russell created the first popular computer game Starwar. Starwar was almost the first computer game ever written, however, they were at least two far-lesser known predecessors: OXO (1952) and Tennis for Two (1958). Since then, video games have become so popular that over the past 30 years, they have become an integral part of our culture, and eventually led to more sophisticated technology for their further development. Their popularity was a major cause for continuously increasing budgets in that sector and consequently for the development of sophisticated technology (Game Engines) for creating these games. This has resulted in the development of tool sets integrated with programming interfaces we now call game engines that also support the development of real-time virtual environments.

Virtual Environments (VEs) allow us to explore an ancient historical site, visit a new home led by a virtual estate agent, or fly through the twisting corridors of a space station in pursuit of alien prey. They simulate the visual experience of immersion in a 3D environment by rendering images of a computer model as seen from an observer viewpoint moving under interactive control by the user. If the rendered images are visually compelling, and they are refreshed quickly enough, the user feels a sense of presence in a virtual world, enabling applications in education, training simulation, computer-aided design, electronic commerce, entertainment and medicine.

This thesis explores a method of three-dimensional fully interactive architectural visualization based on a game engine. Through an examination of the tools and the components of a game engine, this thesis shows how a game engine can be used for creating large, smart and interactive Virtual Environments (VEs).

3D Architectural visualization has been traditionally limited to static animation in the sense that although the images display movement, the movement is always from the same point-of-view. Such animations are frequently pre-rendered and not interactive - [Kevin Conway, 2011].

The final product of this thesis, along with the research presented here, is a 3D interactive computer application which simulates an existing university building named 'The Building of Sciences' of the Technical University of Crete (TUC), in Chania, Crete, Greece.

Application

The created application simulates a real-time 3D navigable environment which the user can interact with and view information related to spatial location of faculty offices, labs or classrooms. Moreover, the user can view data from external sensors, which could be streamed in the interactive application dynamically

This virtual 3D environment represents the Sciences Building of Technical the University of Crete (TUC) and it is built using the Unreal Development Kit game engine. In order to match the real building (scale and position of the walls, doors, windows etc.) as close as possible, the building's architectural floor pans, as well as photographs, were used. Figure x.x shows the virtual environment.

The goal of the application is to show the results of modern 3D architectural visualization using a game engine, as well as to build functionality that helps the users in navigation and getting to know the building. In addition to this, many of the classes created can be used by developers to add interesting functionalities to their VEs.

The main features provided by this application are listed below:

- Navigation: The user navigates through the world interacting with the controlable parts such as doors, elevators, menu system, popups interfaces.

- Display: A Simple and minimalistic yet intuitive and fully functional HUD is displayed while using the application.

- Menu System: A User-Friendly menu from which various functionallities can be initiated, such as, Path Finding, Interactive map, Cinematics etc.

- 3D User interfaces: A 3-layer, rotating user interface is implemented to display information related to faculty loaded from external files.

- Interactive Map: A 2D map of the building with buttons and images on it, that depicts the building's structure.

- Data from external sensors such as, current classroom temperature and presence detection are streamed in the application.

- Mini Map Functionality.

- Path Finding Functionality.

Navigation System
The user navigation system used in the application is the widely adapted WASD kebord layout. WASD is controlled as follows: W and S control the user's forward/back movement and S and D control left and right strafing. The mouse is used to look around the environment and the left mouse button (LMB) is controlling the user's interactions with the Menu System.

1.1 Thesis Outline

This thesis is divided into seven chapters, which will be outlined below.

Chapter 2 - Technical Background: This chapter reviews the process of creating 3D computer graphics and presents the basic tools for creating interactive VEs. It first shows two basic techniques for creating 3D objects and then introduces the reader to texture mapping. Furthermore, this chapter reviews the main components of a game engine, and at the end compares three state-of-art engines.

Chapter 3 – Software Architecture and Development Framework: This chapter describes the main tools used for creating the 3D application. It reviews the core components and tools of the Unreal Development Kit (UDK) game engine and it presents the scripting language - Unrealscript that is used for adding functionality to the VE. Apart from this, it introduces the technology used for creating User Interfaces (UIs) and how it can be connected to the application.

Chapter 4 - Virtual Environment Creation: This chapter desribes the whole process of creating the virtual environment, starting from creating the needed assets (static meshes, textures) to their asseble in the UDK and adding collision and materials to them.

Chapter 5 - Implementation (Scripting): This chapter thoroughly describes every aspect of creating the functionalities that the application supports through code, explaining and images.

Chapter 6 - Conclusion and Future Work: The final chapter presents conclusi-
ons resulting from the thesis, as well as hints about future work.

2 Technical Background

2.1 Introduction

A Virtual Environment (VE) is a computer simulated scene which can be interactively manipulated by users. Typical scenes used in such environments generally comprise of geometric 3D models, shades, images and lights which are converted into final images through the rendering process. The rendering process must be conducted in real time in order to provide scenes which are updated reacting to user interaction.

Figure 2.1. Virtual Environment [Team Bunraku]

In this chapter the technologies and the methods used for creating 3D Virtual Environments (VEs) are discussed.

2.2 3D computer graphics

3D computer graphics are those that use a three-dimensional representation of geometric data. This data is stored in the computer so that performing calculations and rendering 2D images is assured.

3D computer graphics creation is a three step process:

- First comes 3D modeling , which is the process of forming a computer model of an object's shape

- The second step includes layout and animation, that is the motion and placement of objects within a scene
- The third step is 3D rendering . This basically includes the computer calculations that, based on light placement, surface types, and other qualities, generate the image.

2.2.1 Modeling

The model actually describes the process of forming the shape of an object. The two most common sources of 3D models are those that an artist or engineer originates on the computer with some kind of 3D modeling tool, and models scanned into a computer from real-world objects. Basically, a 3D model is formed from points called vertices that define the shape and form polygons. A polygon is an area formed from at least three vertexes (a triangle). The overall integrity of the model and its suitability to use in animation depend on the structure of the polygons.

2.2.2 Layout and animation

Before rendering into an image, objects must be placed in a scene. This defines spatial relationships between objects, including location and size. Animation refers to the temporal description of an object, i.e., how it moves and deforms over time. Popular methods include keyframing, inverse kinematics, and motion capture. A combination of these techniques is often used in practice.

2.2.3 Rendering

Rendering is the process of generating an image from a model (or models in what collectively could be called a scene file), by means of computer programs. A scene file contains objects in a strictly defined language or data structure; it would contain geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The data contained in the scene file is then passed to a rendering program to be processed and output to a digital image or raster graphics image file. The rendering program is usually built into the computer graphics software, though others are available as plug-ins or entirely separate programs.

2.3 3D Models

3D models represent a 3D object using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, etc. Being a

collection of data (points and other information), 3D models can be created by hand, algorithmically (procedural modeling), or scanned.

Almost all 3D models can be divided into two categories:

- The first category is the so-called solid models. These models define the volume of the object they represent (like a rock). These are more realistic, but more difficult to build. Solid models are generally used for nonvisual simulations such as medical and engineering simulations, for CAD and specialized visual applications such as ray tracing and constructive solid geometry
- The second category includes the shell/boundary models , which represent the surface, for instance the boundary of the object, not its volume (like an infinitesimally thin eggshell). These are easier to work with than solid models.

2.3.1 Modelling Process

There are several modeling techniques, but the two most used are Polygonal and Curve Modeling. These are briefly described in this section.

Polygonal modeling - Points in 3D space, called vertices, are connected by line segments to form a polygonal mesh. The vast majority of 3D models today are built as textured polygonal models, because they are flexible and because computers can render them so quickly. However, polygons are planar and can only approximate curved surfaces using many polygons.

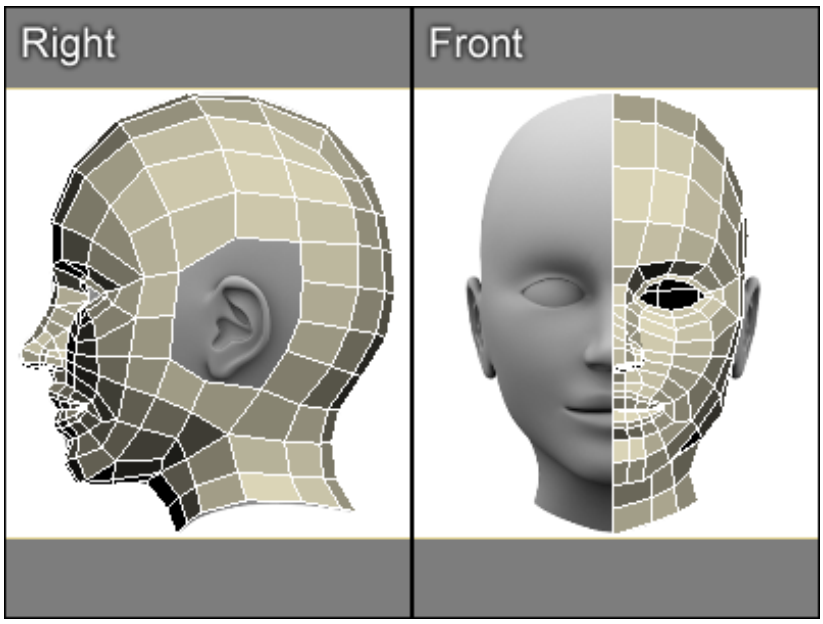


Figure 2.2. 3D polygonal modeling of a human face

1. **Curve modeling** - Surfaces are defined by curves, which are influenced by weighted control points. The curve follows (but does not necessarily interpolate) the points. Increasing the weight for a point will pull the curve closer to that point.

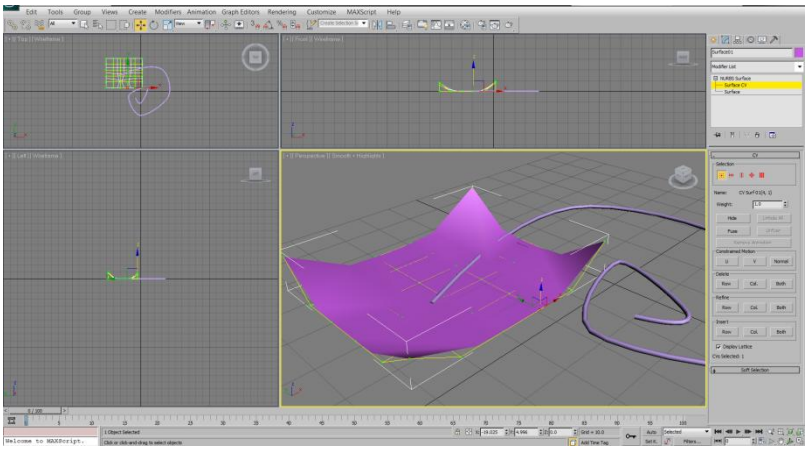


Figure 2.3. An example of Curve Modeling

The most popular programs for creating 3D models are: 3DS Max, Blender, Cinema 4D, Maya, Lightwave, Modo and solidThinking. Modeling can be performed also by using some scene description language such as X3D, VRML, and POV-Ray.

2.4 Texturing – Importance of texture maps

Texture mapping is a method for adding detail, surface texture (a bitmap or raster image), or color to a computer-generated graphic or 3D model.

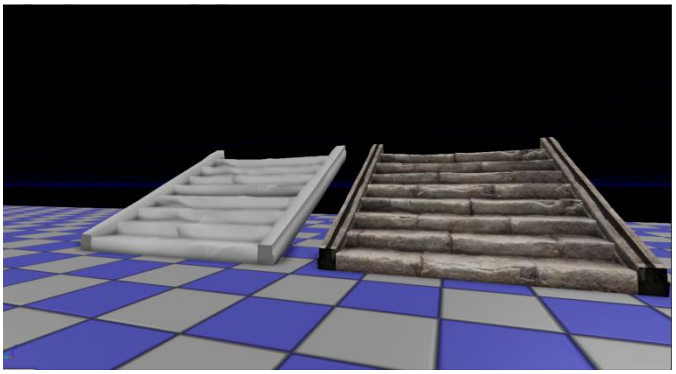


Figure 2.4. 3D model without textures (left), The same model with textures (right)

A texture map is applied (mapped) to the surface of a shape or polygon. This process is akin to applying patterned paper to a plain white box. Every vertex in a polygon is assigned a texture coordinate (which in the 2d case is also known as a UV coordinate) either via explicit assignment or by procedural definition. Image sampling locations are then interpolated across the face of a polygon to produce a visual result that seems to have more richness than could otherwise be achieved with a limited number of polygons. Multitexturing is the use of more than one texture at a time on a polygon. For instance, a light map texture may be used to light a surface as an alternative to recalculating that lighting every time the surface is rendered. Another multitexture technique is bump mapping, which allows a texture to directly control the facing direction of a surface for the purposes of its lighting calculations; it can give a very good appearance of a complex surface, such as tree bark or rough concrete that takes on lighting detail in addition to the usual detailed coloring. Bump mapping has become popular in recent video games as graphics hardware has become powerful enough to accommodate it in real-time.

The way the resulting pixels on the screen are calculated from the texels (texture pixels) is governed by texture filtering. The fastest method is to use the nearest-neighbour interpolation, but bilinear interpolation or trilinear interpolations between mipmaps are two commonly used alternatives which reduce aliasing or jaggies. In the event of a texture coordinatebeing outside the texture, it is either clamped or wrapped.

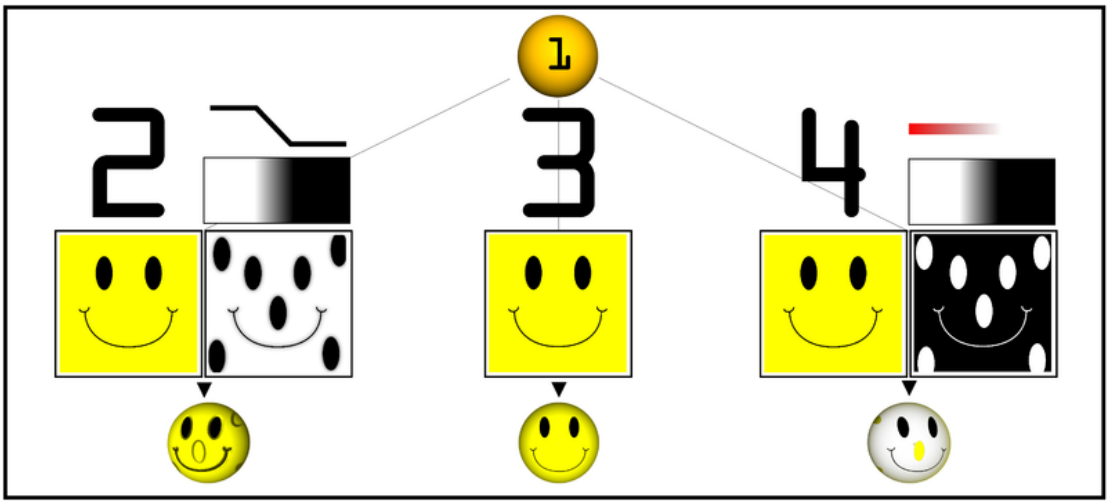


Figure 2.5. Examples of multitexturing. 1) Untextured sphere, 2) Texture and bump maps, 3) Texture map only, 4) Opacity and texture maps.

The essential map types are described below

Color (or Diffuse) Maps: As the name would imply, the first and most obvious use for a texture map is to add color or texture to the surface of a model. This could be as simple as applying a wood grain texture to a table surface, or as complex as a color map for an entire game character (including armor and accessories). However, the term texture map, as it's often used is a bit of a misnomer—surface maps play a huge role in computer graphics beyond just color and texture. In a production setting, a character or environment's color map is usually just one of three maps that will be used for almost every single 3D model.

Specular Map: A specular map tells the software which parts of a model should be shiny or glossy, and also the magnitude of the glossiness. Specular maps are named for the fact that shiny surfaces, like metals, ceramics, and some plastics show a strong specular highlight (a direct reflection from a strong light source). Specular highlights are the white reflection on the rim of a coffee mug. Another common example of specular reflection is the tiny white glimmer in someone's eye, just above the pupil.

A specular map is typically a grayscale image, and is absolutely essential for surfaces that aren't uniformly features. An armored vehicle, for example, requires a specular map in order for scratches, dents, and imperfections in the armor to come across convincingly. Similarly, a game character made of multiple materials would need a specular map to convey the different levels of glossiness between the character's skin, metal belt buckle, and clothing material.

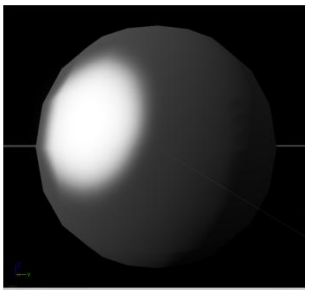


Figure 2.6. Specular Map applied on a 3D ball

Bump (Normal) Map: A bit more complex than either of the two previous examples, bump maps are a form of texture map that can help give a more realistic indication of bumps or depressions on the surface of a model.

To increase the impression of realism, a bump or normal map would be added to more accurately recreate the coarse, grainy surface of, for instance, a brick, and heighten the illusion that the cracks between bricks are actually receding in space. Of course, it would be possible to achieve the same effect by modeling each and

every brick by hand, but a normal mapped plane is much more computationally efficient.

Bump, displacement, and normal maps are a discussion in their own right, and are absolutely essential for achieving photo-realism in a render.

Figure 2.7. Wall with Normal Map applied

Transparency (or Opacity) Map: Exactly like a reflection map, except it tells the software which portions of the model should be transparent. A common use for a transparency map would be a surface that would otherwise be very difficult, or too computationally expensive to duplicate, like a chain link fence. Using a transparency, instead of modeling the links individually can be quite convincing as long as the model doesn't feature too close to the foreground, and uses far fewer polygons.

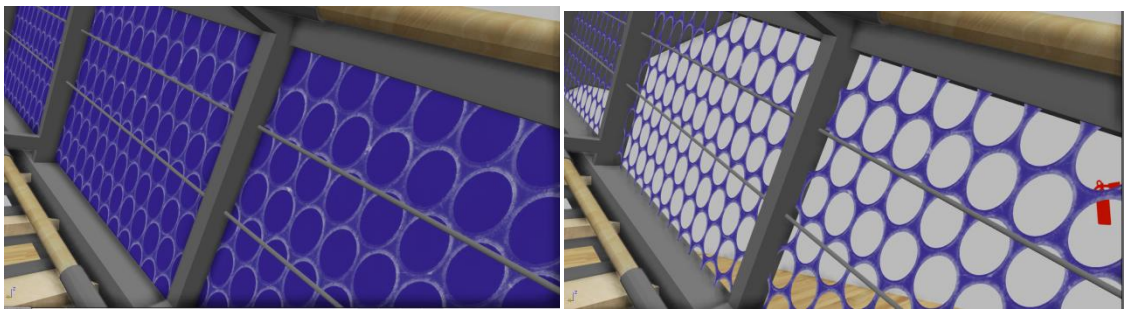


Figure 2.8. Mesh with diffuse map only (left). Opacity texture applied on the mesh (right)

Texture maps are crucial for the design of the virtual scene. They facilitate the reduction of the polygonal complexity of the 3d models used, which in any other way would hinder rendering performance. In particular, the normal maps of some

low polygon count models used in our scene were acquired from their high polygon versions to keep all the fine details of the models, thus maintaining an acceptable lighting quality with low computational cost. An example of this method can be seen on the following picture. Another use of texture maps, are opacity maps which allow for the otherwise opaque window in our scene to become transparent.

2.5 Game Engines

A game engine is a system designed primarily for the creation and development of video games. The leading game engines provide a software framework that developers use to create games for video game consoles and personal computers.

A game engine provides the framework and the Application User Interface (API) for the developer to use and communicate with the hardware. It consists of separate autonomous systems, each handling a specific process, e.g. the graphics system, the sound system, the physics system, etc. Figure 2.9 shows the standard architecture of game engines.

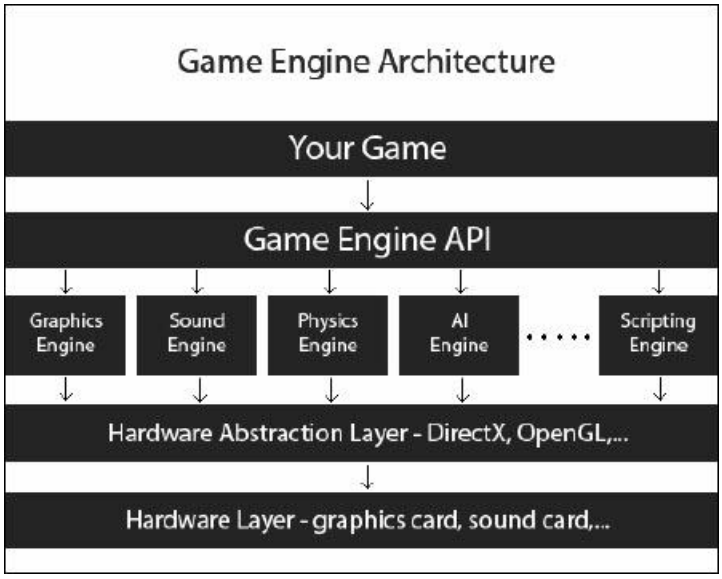


Figure 2.9. Architecture of a game engine

In recent years, the game engine technology has matured and became very user-friendly. This has led the fame engines to be used in variety of different applications such as architectural visualization, training, medical and military simulation applications. In other words, game engines can be used in development of any 3D

interactive environment. It was decided early on in this project that the implementation of the 3D interactive environment would be based on a game engine for this reason. Although the final application is not a computer game, a game engine offers the required tools and programming platform in order to create an interactive photorealistic environment.

2.5.1 Game Engine Components

Game Engines contains several components. The structure of a game engine is modularized. This means that each component work independently, but all the modules are controled by a central core engine. The main components that appart a game engine are: the Rendering (Graphic) Engine, the Physics Engine and the Sound Engine.

Rendering Engine

The job of the Rendering engine is to control what the user sees when using the 3D application. Rendering includes all the processes that take 3D geometry and transforms it into a representation of a world. The renderer does the rendering in real time with constantly changing variables and input to create the illusion for the player that the player is occupying another world that he or she can interact.

Physics Engine

In virtual environments, objects and characters must respond to collisions, gravity, acceleration, and inertia accurately. The component of a game engine that simulates physics models is called Physics Engine. The Physics Engine is basicaly a computer program that manages physics variables such as mass, velocity, friction etc. In most game engines, the physics engine is a middleware software module that is licensed from an independent software developer. Physics engines currently available for game engines include PhysX(Nvidia), Havok, Open Dynamics Engine, Vortex and many more.

Sound Engine

The Sound Engine is responsible for taking the sounds that we have created and playing them based on certain events in the application (for examlple when a door opens, when the user walks, when braking glass etc.)

2.6 Three Modern Game Engines

In this section, the game engines that were considered but rejected will be presented, as well as UDK, which is the game engine that was selected for the implementation of the 3D lighting system.

2.6.1 Unity 3D

Unity 3D is a fully featured application suite, providing tools to create 3D games or other applications, such as architectural visualization. It provides support for editing object geometry, surfaces, lights and sounds. It uses the Ageia physics engine, provided by nVidia. A lightmapping system, called Beast, is included. In terms of programming, Unity 3D supports three programming languages: JavaScript, C# and Boo, which is a python variation. All three languages are fast and can be interconnected. The game's logic runs in the open-source platform "Mono", offering speed and flexibility. Required for the development process a debugger is also included, allowing pausing the game at any time and resuming it step-by-step.

Unity 3D is widely used, utilized by a large community offering help. It is free for non-commercial use and it is targeted to all platforms, such as PC, MAC, Android, iOS and web. This game engine targets at offering increased rendering speed, even including machines with low memory and computational power, such as iOS and Android smartphones, and not at creating interactive photorealistic environments, which need a lot of memory and very fast CPU and GPU to render at acceptable speed. Unity 3D was rejected, because it was necessary to have the ability to create photorealistic VEs and render them in real-time.

2.6.2 Torque 3D

Torque 3D is a sophisticated game engine for creating networked games. It includes advanced rendering technology, a Graphical User Interface (GUI) building tool and a World Editor, providing an entire suit of WYSIWYG (What-You-See-Is-What-You-Get) tools to create the game or simulation application.

The programming language used is "TorqueScript", which resembles C/C++. It is targeted for both Windows and MacOS platforms, as well as the web. The main disadvantage of Torque 3D is that it is not free, but it needs to be licensed for $100. For this reason, Torque 3D was also rejected.

2.6.3 Unreal Development Kit (UDK)

The Unreal Development Kit (UDK) is a fully functional version of the Unreal Engine 3 that is available for anyone to download from Epic. UDK is currently one of the leading game engines. It became free on November 2009 for non-commercial use

and it is used by the world's largest development studios. The UDK community includes thousands of people from around the world, providing help and advice.

UDK's core is written in C++, making it very fast. It offers the ability to use both "UnrealScript", UDK's object-oriented scripting language, and C/C++ programming languages. It provides many different tools for the creation and the rendering of a virtual scene. The Unreal Development Kit includes the Unreal Lightmass, which is an advanced global illumination solver. The Unreal Lightmass supports the illumination with a single sun, giving off soft shadows and automatically computing the diffuse interreflection (color bleeding). It also offers a variety of options to optimize the illumination solution. It can provide detailed shadows by using directional light mapping, static shadowing and diffuse normal-mapped lighting. An unlimited number of lights can be pre-computed and stored in a single set of texture maps. The complete software architecture of UDK will be presented in Chapter 3 – Software Architecture and Development Framework.

3 Software Architecture and Development Framework

In this chapter, the software architecture and the framework used in the development process will be described in detail.

3.1 Unreal Development Kit

For the purposes of this project, the power of building and extending upon a framework was preferred to building from scratch. As already discussed, UDK is a powerful framework used mostly in creating computer games and visualization.

UDK consists of different parts, making it act both like a game engine and a 3D authoring environment. It provides the necessary tools to import 3D objects, create and assign materials on objects that affect the lighting distribution, precompute lighting effects and import and use sounds and sound effects. It, also, allows the designed application to seemingly attach to Flash-based User Interfaces (UI).

UDK can also be used to render the created VEs, as well as create and respond to events while navigating the synthetic scenes. UDK offers the ability to use both C/C++ and UnrealScript, which provides the developers with a built-in object-oriented programming language that maps the needs of game programming and allows easy manipulation of the actors in a synthetic scene.

The core components of UDK are briefly described next.

3.1.1 Rendering Engine

The Rendering (Graphic) engine monitors whatever appears on the screen during the game. It determines which items will be displayed in front of others or what would stay hidden.

The Unreal Development Kit comes along with Gemini, a flexible and highly optimized multi-threaded rendering system, which creates lush computer graphics scenes and provides the power necessary for photorealistic simulations. UDK features a 64-bit color High Dynamic Range (HDR) rendering pipeline. The gamma-corrected, linear color space renderer provides for immaculate color precision while supporting a wide range of post-processing effects such as motion blur, depth of field, bloom, ambient occlusion and user-defined materials.

UDK supports all modern per-pixel lighting and rendering techniques, including normal mapped, parameterized Phong lighting, custom user-controlled per material lighting models including anisotropic effects, virtual displacement mapping, light attenuation functions, pre-computed shadow masks and directional light maps. UDK provides volumetric environmental effects that integrate seamlessly into any environment. Camera, volume and opaque object interactions are all handled per-pixel. Worlds created with UDK can easily feature multi-layered, global fog height and fog volumes of multiple densities.

It also supports a high-performance texture streaming system. Additionally, UDK's scalability settings ensure that the application will run on a wide range of PC configurations, supporting both Direct3D 9 and Direct3D 11.

3.1.2 Sound Engine

An important element of 3D applications is sound. The UDK audio engine enables the designer to add sounds to the user's walking, to breaking a glass, water flowing in a river etc.

3.1.3 Physics Engine

UDK's physics engine is powered by NVIDIA's PhysX, providing unparalleled control over character movement, dynamic fluid simulation and even soft body physics. Tweak and modify your physics using the Unreal PhAT visual modeling tool. Create destructible worlds and physically simulated cloth and clothing using NVIDIA's APEX modules, which are tightly integrated into UE3.

UDK Tools

The main tools inside UDK are the Unreal Editor, which is used to create and edit VEs, handling all the actors and their properties located in the VEs, the Unreal Kismet, which allows for the creation of sequences of events and corresponding actions, and Unreal Matinee, responsible for the animation of actors or real-time changes in the actors' properties.

3.1.4 Unreal Editor

The Unreal Editor (UnrealEd) is a fully integrated editing environment inside UDK used to create and edit VEs. The engine's tools can be accessed from within Unreal Editor, as well as 3D objects, sounds, videos, textures and images can be imported to the Content Browser library and inserted in the VEs. Moreover, the Unreal Editor

can create and assign materials to 3D objects, as well as alter lighting and rendering configurations. Figure 3.1 presents the Unreal Editor.

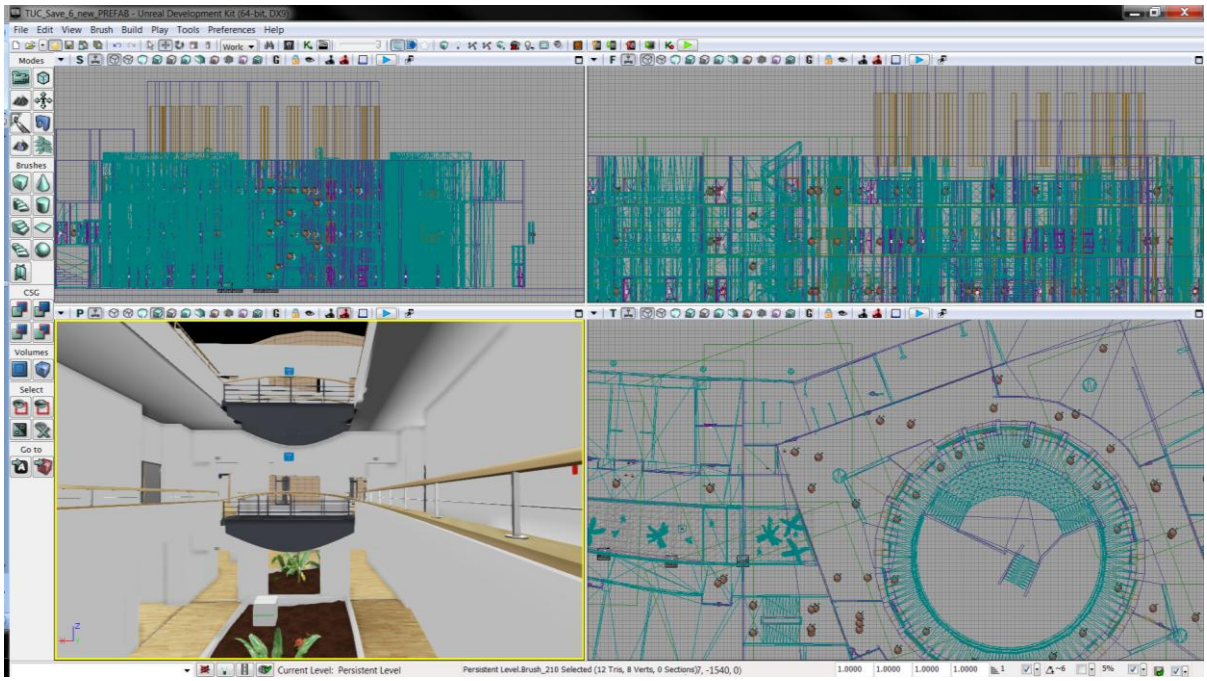


Figure 3.1. The Unreal Editor with a virtual scene loaded

Actors and their properties

Everything inside the virtual scene created in the Unreal Editor is considered from UDK to be an "Actor", from 3d objects to lights and sounds. This is in accordance with Unreal Script (see below), which is an Object-Oriented Programming language and every object is assigned to a class that extends from Actor. So, 3d objects are assigned to StaticMeshActor class, lights can be variedly assigned to PointLight, PointLightToggleable, DominantDirectionalLight classes according to their function, sounds are assigned to Sound class, while all these classes extend from the Actor class.

The 3D objects imported into Unreal Editor can be assigned to Static Mesh, used for static objects, or Skeletal Mesh, used for character bodies. After an object is imported through the Content Browser, we can change its main attributes, like the collision box, materials, light map UVs and polygon count with the Static Mesh Editor. These changes will affect all the instances of this object that will be inserted in the virtual scene, unless they are overridden.

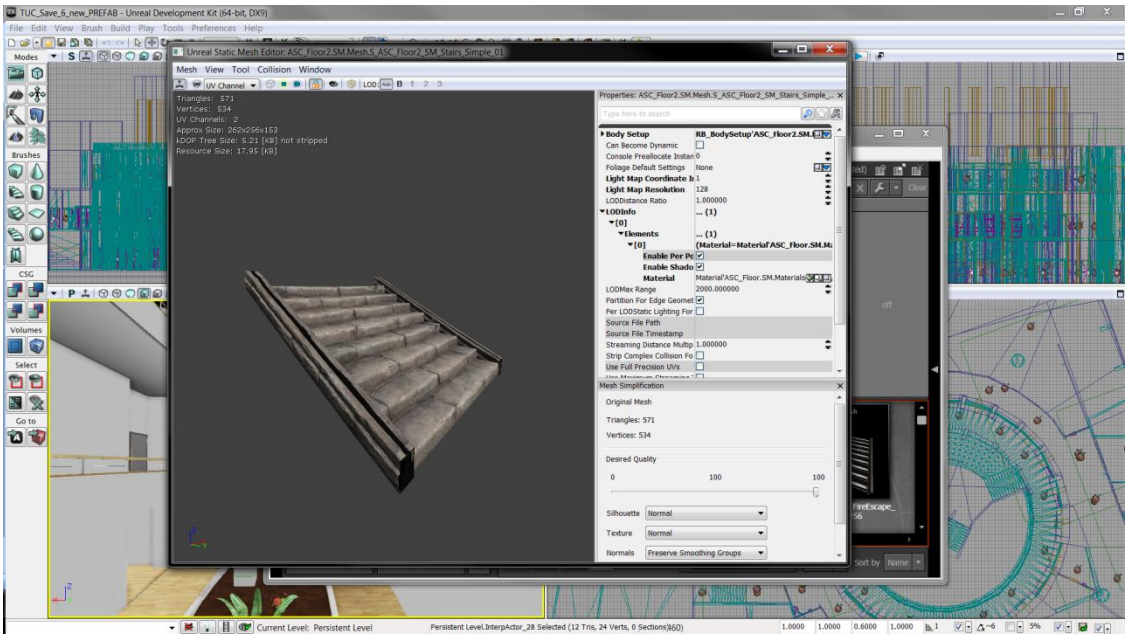


Figure 3.2 Static Mesh Editor

Once an object is inserted in the editor from the Content Browser library, an instance of its predefined Actor class is created and the editor offers the option to change the configuration of the specific instance, without affecting the other instances. This is true for all kinds of actors loaded in a scene, either being a light or any other possible Actor. The options that can be changed include the object's position, draw scale, properties for the lighting system, materials, collision, components, etc.

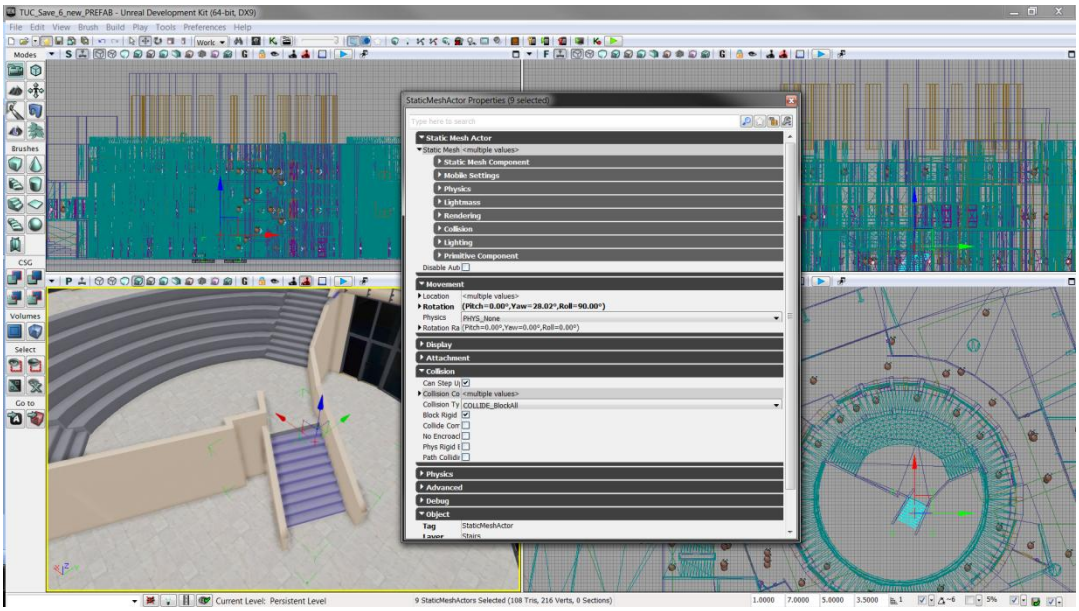


Figure 3.3. Properties of a Static Mesh Actor instance

Some engine's tools that are accessible through the UnrealEd are the following:

Unreal Content Browser, a powerful asset browser for organizing game content, including instant game-wide asset search, a robust content tagging system and drag-and-drop actor placement.

Unreal Kismet Editor, a visual scripting tool for the UDK. It enables level designers to create scripts in UnrealScript for gameplay events using a visual interface.

Unreal Matinee Editor, a tool within Unreal Editor that is used to change properties of actors over time, or to animate them.
Material Editor, an intuitive tool for visually designing materials and shaders using a graph network.

Unreal Cascade, an advanced particle physics and environmental effects editor.
Level Construction tools, including geometry editing features and BSP-based model creation.

Mesh Editor, for previewing meshes and adjusting physics properties. Also includes support for simplification of meshes, generating LODs and texture coordinates and fracturing objects.

3.1.5 Unreal Kismet

Unreal Kismet is a visual scripting system. It allows level designers or even artists to create scripting behaviour it the virtual environment in a quick intuitive maner, without touching a single line of code.The behaviour is created by connecting nodes (sequence object) with wires. The nodes have inputs and outputs and in order to use them, they are dragged and dropped on a canvas. These nodes represent events, actions or variables and their interconnection allows the change of the program flow. There are some predefined events and actions, however, more can be created through Unreal Script. The nodes form a network called sequence. These sequences are managed in a separate window.

An example of kismet sequence is presented in Figure 3.4. This example shows a sequence which is responsible for opening a door. The most left node is an event, in particular a trigger event. This event is listening out for the player to make use of the trigger placed in the world (triggers are used by pressing the E button). As soon as the trigger is used, the event output is activated which will cause activation of the switch node because of the connection between them (the arrow from the event output to the input of the switch node). The node on the right is a special one and it

represents the animation of the door. Two if its inputs, one for playing the opening animation and one for the closing, are used. These are activated in correspondence to the switch outputs.

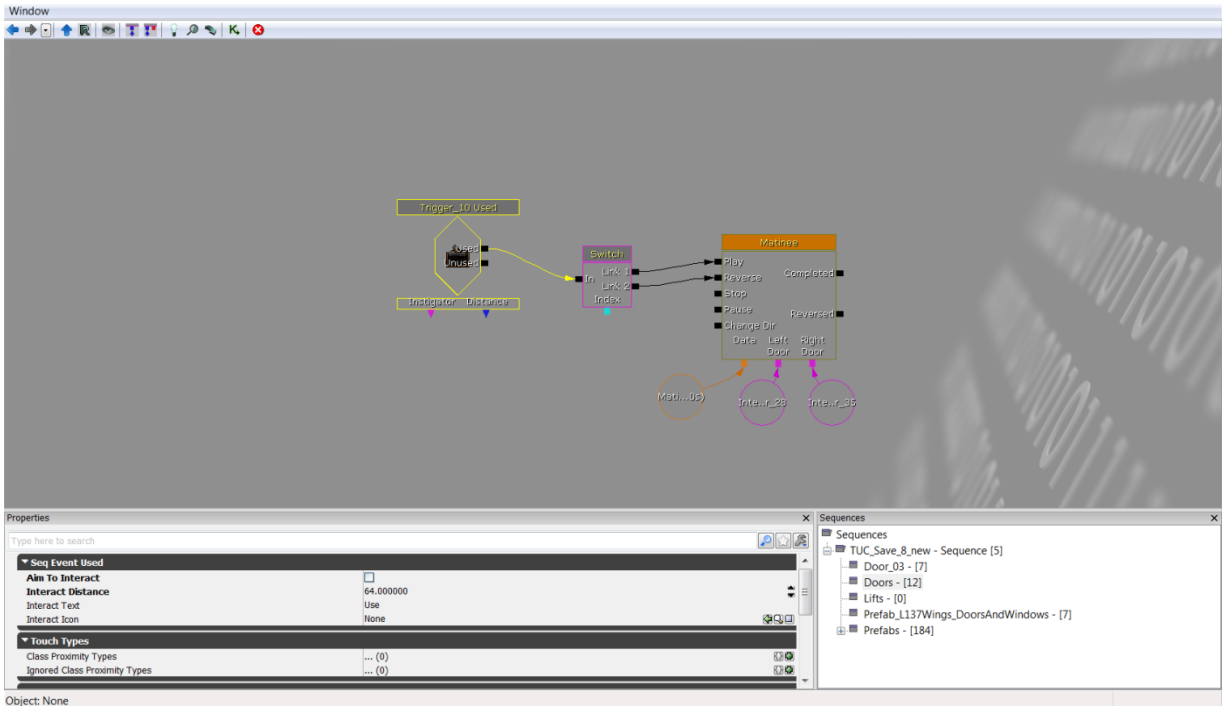


Figure 3.4. Unreal Kismet with a simple sequence

3.1.6 Material Editor

A useful tool within Unreal Editor, necessary for creating realistic environments, is the Material Editor. This tool handles the creation and editing of different materials that can be assigned to objects inside the scenes created. Materials affect the objects they are assigned to in a variety of ways, mainly in terms of their texture and their interaction with the light.

The Material Editor is node-based, much like the Unreal Kismet. However, its nodes do not represent events or actions, but textures, colors and several processing filters, such as addition of two different textures. The Material Editor provides the main node which has all the supported properties of the material, such as the diffuse, emissive and specular properties and each property can receive the output from a node or from a sequence of nodes.

Figure 3.5 displays the Material Editor displaying material for tiles. It can be seen that a texture node, containing a tile texture, is connected to the *diffuse* chanel of the main node and a normal map texture is connected to the *normal map* chanel.

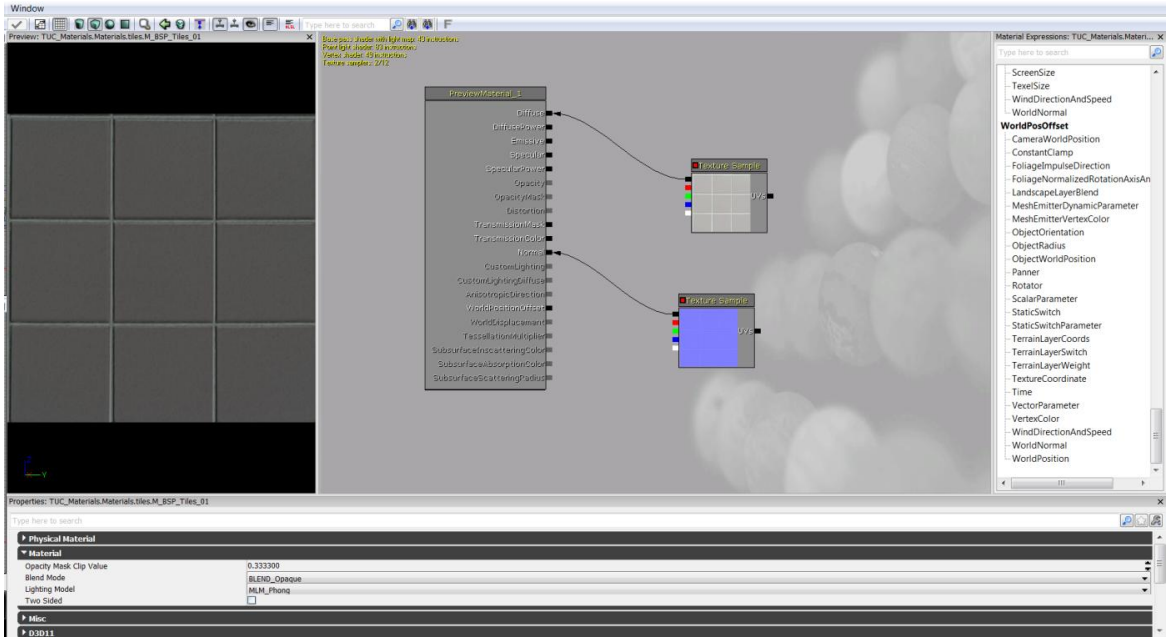


Figure 3.5 The Material Editor with a sample material

3.1.7 Sound Editor

The Unreal Editor supports its own sound engine and a Sound Editor provides the necessary tools, in order to create various sound effects. It supports immersive 3D location-based sounds and gives complete control over pitch, levels, looping, filtering, modulation and randomization.

As the rest of the Unreal Editor's tools, the Sound Editor provides a node-based User Interface to import and use several sound cues, change their properties, mix them together and channel the resulting output as a new sound effect.

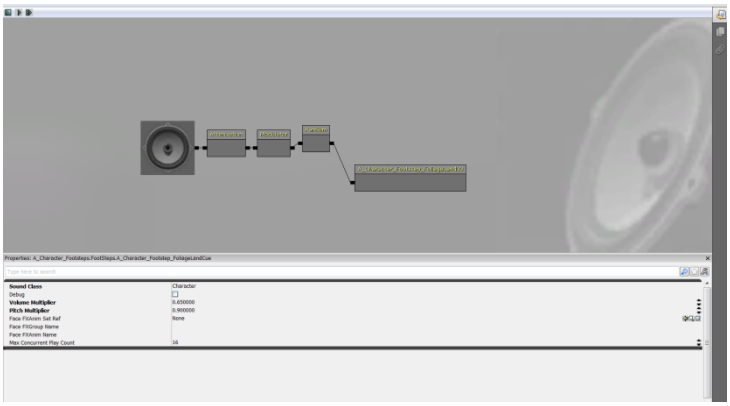


Figure 3.6 The Sound Editor with a simple sound sequence loaded

3.1.8 Unreal Matinee

The Matinee animation tool provides the ability to animate the properties of actors over time, either to create dynamic gameplay or cinematic in-game sequences. The system is based on the use of specialized animation tracks which can have keyframes placed to set the values of certain properties of the actors in the level. The Matinee editor is similar to the non-linear editors used for video editing, making it familiar to video professionals.

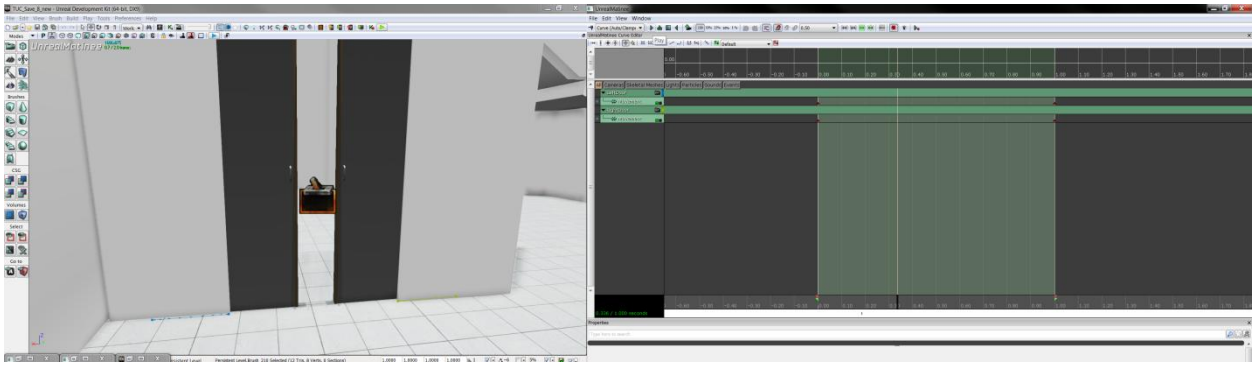


Figure 3.7. Animating pocket door (left) using the Unreal Matinee Editor (right)

3.1.9 Unreal Lightmass

Unreal Lightmass is an advanced global illumination solver. It uses a refined version of the radiosity algorithm, storing the information in each illuminated 3D object's light map, while providing ray-tracing capabilities, by supporting Billboard reflections, which allows complex reflections even with static and dynamic shadows with minimal CPU overhead. Unreal Lightmass is provided as part of the Unreal

Development Kit (UDK) and it can only work on scenes created through it. Its performance depends on the complexity of the scenes created and the types of light emitting sources that exist in the scene. It is optimized to increase the renderer's performance. Its main features include:

- Area lights and shadows: Within Lightmass, all lights are area lights by default. The shape used by Point and Spot light sources is a sphere. The radius of the sphere is defined by LightSourceRadius under LightmassSettings. Directional light sources use a disk, positioned at the edge of the scene. Light source size is one of the two factors controlling shadow softness, as larger light sources will create softer shadows. The other factor is distance from the receiving location to the shadow caster. Area shadows get softer as this distance increases, just like in real life.

- Diffuse interreflection: Diffuse Interreflection is by far the most visually important global illumination lighting effect. Light bounces by default with Lightmass, and the Diffuse term of each material controls how much light (and what color) bounces in all directions. This effect is sometimes called color bleeding. It's important to remember that diffuse interreflection is incoming light reflecting equally in all directions, which means that it is not affected by the viewing direction or position.

- Mesh Area Lights from Emissive: The emissive input of any material applied to a static object can be used to create mesh area lights. Mesh area lights are similar to point lights, but they can have arbitrary shape and intensity across the surface of the light. Each positive emissive texel emits light in the hemisphere around the texel's normal based on the intensity of that texel. Each neighboring group of emissive texels will be treated as one mesh area light that emits one color.

- Translucent shadows: Light passing through a translucent material that is applied to a static shadow casting mesh will lose some energy, resulting in a translucent shadow.

3.2 UnrealScript

UnrealScript is designed to provide the developers with a powerful, built-in programming language that maps the needs of game programming. The major design goals of UnrealScript are:

- Enabling time, state and network programming, which traditional programming languages do not address but are needed in game programming. UnrealScript includes native support for time state and network programming which not only simplifies game programming, but also results in low execution time, due to the native code written in C/C++;

- Programming simplicity, object-orientation and compile-time error checking, helpful attributes met in Java are also met in UnrealScript. More specifically, deriving from Java UnrealScript offers:
    - A pointerless environment with automatic garbage collection;
    - A simple single-inheritance class graph;
    - Strong compile-time type checking;
    - A safe client-side execution "sandbox";
    - The familiar look and feel of C/C++/Java code.

Often during the implementation, design trade-offs had to be made, choosing between execution speed and development simplicity. Execution speed was then sacrificed, since all the native code in UnrealScript is written in C/C++ and resulted performance outweighs the added complexity. UnrealScript has very slow execution speed compared to C, but since a large portion of the engine's native code is in C only the 10%-20% of code in UnrealScript that is executed when called has low performance.

3.2.1 The Unreal Virtual Machine

The Unreal Virtual Machine consists of several components: The server, the client, the rendering engine, and the engine's support code. The Unreal server controls all the gameplay and interaction between players and actors (3D objects, lights or sounds that can be inserted in a synthetic scene). A listen server is able to host both a game and a client on the same computer, whereas the dedicated server allows a host to run on the computer without a client. All players connect to this machine and are considered clients.

The gameplay takes place inside a level, containing geometry actors and players. Many levels can be running simultaneously, each being independent and shielded from the other.

This helps in cases where pre-rendered levels need to be fast-loaded one after another. Every actor on a map can be either player-controlled or script-controlled. The script controls the actor's movement, behavior and interaction with other actors. Actor's control can change in game from player to script and vice versa.

Time management is done by dividing each time second of gameplay into Ticks. Each Tick is only limited by CPU power, and typically lasts 1/100th of a second. Functions that manage time are really helpful for gameplay design. Latent functions, such as Sleep, MoveTo and more cannot be called from within a function but only within a state.

When latent functions are executing in an actor, the actor's state execution does not continue until the latent functions are completed. However, other actors may call functions from the specific actor that handles the latent function. The result is that all functions can be called, even with latent functions pending.

In UnrealScript, every actor is as if executed on its own thread. Windows threads are not efficient in handling thousands at once, so UnrealScript simulates threads instead. This means that 100 spawned actors will be executed independently of each other each Tick.

3.2.2 Class Hierarchy

UnrealScript is purely object-oriented and comprises of a well-defined object model with support for high level object-oriented concepts such as serialization and polymorphism. This design differs from the monolithical one that classic games adopted, having their major functionality hardcoded and being non-expandable at the object level. Before working with UnrealScript, understanding the classes hierarchy within Unreal is crucial in relation to the programming part.

The main gain from this design type is that object types can be added to Unreal at runtime. This form of extensibility is extremely powerful, as it encourages the Unreal community to create Unreal enhancements that all interoperate. The five main classes one should start with are `Object`, `Actor`, `Pawn`, `Controller` and `Info`.

`Object` is the parent class of all objects in Unreal. All of the functions in the Object class are accessible everywhere, because everything derives from Object. `Object` is an abstract base class, in that it doesn't do anything useful. All functionality is provided by subclasses, such as Texture (a texture map), TextBuffer (a chunk of text), and Class (which describes the class of other objects).

`Actor` (extends Object) is the parent class of all standalone game objects in Unreal. The Actor class contains all of the functionality needed for an actor to be placed inside a scene, move around, interact with other actors, affect the environment, and complete other useful game-related actions.

`Pawn` (extends Actor) is the parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.

`Controller` (extends Actor) is the class that defines the logic of the pawn. If Pawn resembles the body, Controller is the brain commanding the body. Timers and executable functions can be called from this type of class.

`Info` (extends Actor) is the class that sets the rules of gameplay. Players joining will be handled in this class, which decides which Pawn will be created for the player in the scene and which Controller will handle the behavior of the pawn.

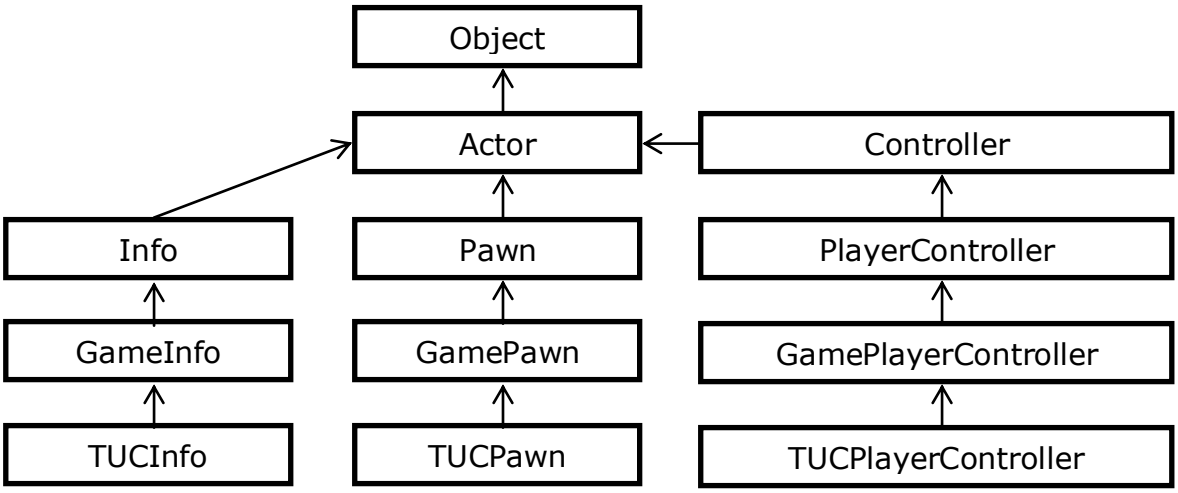


Figure 3.8 UnrealScript Class Hierarchy

3.2.3 Timers

Timers are a mechanism used for scheduling an event to occur. Time management is important both for gameplay issues and for programming tricks. All Actors can have more than one timer implemented as an array of structs. The native code involving timers is written in C++, so using many timers per tick is safe, unless hundreds expire simultaneously. This would require the execution of UnrealScript code for each one of the timers and the heavy computational demands would lead to unwanted delay to the handling of the timers.

The following function fires the function Foo after 10 seconds:

```
SetTimer(10, false, 'Foo');
```

3.2.4 States

States are known from Hardware engineering where it is common to see finite state machines managing the behaviour of a complex object. The same management is needed in game programming, allowing each actor to behave differently, according to its state. Usually, when implementing states in C/C++ there are many switch cases used, based on the object's state. This method, however, is not efficient, since most applications require many states, resulting to difficulties in developing and maintaining the application.

UnrealScript supports states at the language level. Each actor can include many different states, however, only one can be active at any time. The state the actor is in reflects the actions it wants to perform. Attacking, Wandering, Dying are potential states a Pawn may acquire. Each state can have several functions, which can be the same as another state's functions. However, only the functions in the active state can be called. For example, if an application dictates that an action should only be performed in a specific stage, then this stage could be encapsulated in a different state that implements the function corresponding to that action differently than other states.

States provide a simple way to write state-specific functions, so that the same function can be handled in different ways depending on which state the actor is in when the function is called. Within a state, one can write special "state code", using the regular UnrealScript commands plus several special functions known as "latent functions". A latent function is a function that executes slowly (i.e. non-blocking), and may return after a certain amount of "game time" has passed. Time-based programming is enabled which is a major benefit that neither C/C++, nor Java offer. Namely, code can be written in the same way it is conceptualized. For example, a script can support the action of "turn the TV on; show video for 2 seconds; turn the TV off". This can be done with simple, linear code, and the Unreal engine takes care of the details of managing the time-based execution of the code.

3.2.5 Interfaces

UnrealEngine3's UnrealScript has support for interface classes that resembles much of the Java implementation. As with other programming languages, interfaces can only contain function declarations and no function bodies. The implementation for these declared methods must be conducted in the class that actually implements the interface. All function types, as well as events, are allowed. Even delegates can be defined in interfaces.

An interface can only contain declarations which do not affect the memory layout of the class: enums, structs and constants can be declared. Variables cannot be declared for this reason.

3.2.6 UnrealScript Compiler

The UnrealScript compiler is three-pass. Unlike C++, UnrealScript is compiled in three distinct passes. In the first pass, variable, struct, enum, const, state and function declarations are parsed and remembered, e.g. the skeleton of each class is built. In the second pass, the script code is compiled to byte codes. This enables complex script hierarchies with circular dependencies to be completely compiled and linked in two passes, without a separate link phase. The third phase parses and imports default properties for the class using the values specified in the default properties block in the .uc file.

3.2.7 UnrealScript Programming Strategy

UnrealScript is a slow programming language when compared to C/C++. A program in UnrealScript runs about 20x slower than C. However, script programs written in UnrealScript are executed only 5-10% of the time with the rest of the 95% being handled in the native code written in C/C++. This means that only the 'interesting' events will be handled in UnrealScript. For example, when writing a projectile script, what is typically written is a HitWall, Bounce, and Touch function describing what to do when key events happen. Thus, 95% of the time, a projectile script is not executing any code and is just waiting for the physics code to notify it of an event. This is inherently very efficient.

The Unreal log may provide useful information while testing scripts. The UnrealScript runtime often generates warnings in the log that notify the programmer of non-fatal problems that may have occurred.

UnrealScript's object-oriented capabilities should be exploited as much as possible. Creating new functionality by overriding existing functions and states leads to clean code that is easy to modify and easy to integrate with other peoples' work. Traditional C techniques should be avoided, such as writing a switch statement based on the class of an actor or the state because code like this tends to clutter as new classes and states are added or modified.

3.2.8 Configuration Files

Unreal Development Kit relies on configuration files to dictate how it will function and initialize. These *Configuration* files allow low-level file interaction such as reading user data from external files or writing the output of a program in order for files to be preserved after it has terminated. The advantage of using configuration files is that they allow us to change some settings of the application without having to recompile the source code.

Configuration files are just plain text files that are saved with the extension ".ini" and they hold information that is used by the UDK and/or the user application. .Ini files are read when the class they are associated with is first loaded. After that, they can only be written to, but the values from their reading remains in memory. In order to associate or bind the class with some configuration .ini file, one has to add the *config* modifier in the class definition. For instance:

```
class ClassName extends SomeOtherClass config(ConfigName);
```

This example class declares that its configuration variables are saved/loaded to/From the '`ConfigName` ' configuration file:

Every UnrealScript class that binds to a configuration file can define which of its variables should be loaded from and/or saved in the respective configuration file. So, when the Unreal Editor loads up and initializes its components, it uses the configuration files to recall their properties. For instance, if one has a string that is a config variable, and they change its value in the .Ini file, its default value for that string will be updated when that class's package is next loaded. In order to use configuration variable it first has to be declared as presented below:

```
var config int X;
```

Writing to an .ini file is a two-step process, but it is something that is simple to be done. First, the value of the variables that we wish to write to disk must be set. Once that is done, the native final *SaveConfig*() or native static final *StaticSaveConfig*(), defined in Object is called in the class. *SaveConfig*() writes the contents of all the config variables in the instance of the class to the .ini file assigned to that class. An Example of writiing to a configuration file is given below:

```
X=15;
SaveConfig();
```

Typical configuration files consist of sections of key-value pairs, arranged as follows:

```
[Section]
Key=Value

[TUCProject.TUC_FrontEnd_Hud]
WorldWidth=9320
```

3.2.9 DLL Files

UDK provides the option for an UnrealScript class to bind to an external DLL file, by declaring it in the class definition. For example, the following line declares that *ExampleClass* binds to the *ExampleDLL*:

```
class ExampleClass extends GameInfo DLLBind (ExampleDLL);
```

By binding to a DLL file, an UnrealScript class can call the declared in that DLL file methods or functions, which are written in C/C++. This proves to be an easy and efficient way to implement functions that either UDK does not support at all, such as I/O operations, or it would slow down the application, due to the fact that UnrealScript is slow.

A function residing inside the DLL must be declared in the UnrealScript class file and then it can be called exactly like it would be if it was an original UnrealScript function. Following the previous example and assuming that the *ExampleDLL* contained a function called *ExampleDLLFunction*, the code inside the UnrealScript class would be:

```
dllimport final function ExampleDLLFunction();    // function declaration
…
ExampleDLLFunction ();            // function call
```

3.2.10 Input Manager

The input manager is responsible to handle the communication between the input hardware, such as keyboard, mouse, joystick or button boxes, and the application. The input manager examines a configuration file based on DefaultInput.ini and according to it binds each input action, such as the joystick/mouse movement or key press to a specific method designated to perform the selected action. The Unreal Editor comes with a default configuration file including a limited set of predefined bindings between buttons and methods. However, this file can be altered to match the needs of each application.
In order to create a new binding between a button press and the performed action or to change an already defined binding, this change must be reflected in the

configuration file. Also, the method defined in the configuration file must exist in the UnrealScript code of the new application being developed.
For example, if we wanted to add or change the action performed when the Left Mouse Button is pressed, we would add or change these lines in the "DefaultInput.ini" configuration file:

```
.Bindings = (Name="LeftMouseButton", Command="GBA_Fire")

.Bindings = (Name="GBA_Fire", Command="JProcessItem");
```

In the first line, it is defined that the press of the left mouse button corresponds to the game bindable action named *GBA_Fire*. In the next line, we define that if a game bindable action named *GBA_Fire* occurs, then the Input Manager should start the *JProcessItem* method, located in the application's UnrealScript file. Inside that method we could effectively develop the application to perform whatever action is necessary to correspond to the press of the left mouse button.

3.2.11 Materials in UDK

A Material is an asset which can be applied to a mesh to control the visual look of the scene. In general, when light from the scene hits the surface, the lighting model of the material is used to calculate how that light interacts with the surface. There are several different lighting models available within Unreal Engine 3's material system, but the Phong lighting model is use for what might be called "regular" lighting.

When the lighting calculation is being performed, several material properties are used to come up with the final color of the surface. These material properties are also called material inputs in the material editor and are things like DiffuseColor, EmissiveColor, SpecularColor, SpecularPower, etc.

Diffuse

The diffuse color of a material represents how much of the incoming light reflects equally in all directions. A value of (1,1,1) means that 100% of the incoming light reflects in all directions. Diffuse is not view dependent so it will look the same from any angle, but it does depend on the normal at each pixel. Diffuse color will only show up when affected by some un-shadowed lighting, because it scales the incoming lighting.

Emissive

The emissive color of a material represents how much light the material emits, as if it were a light source, although it does not actually light up other surfaces in the

scene. Since emissive is acting like a light source, it is not affected by lights or shadows. Emissive is sometimes used to represent the ambient lighting term.

Specular
Specular color represents how much of the incoming light reflects in a directional manner. The specular response is brightest when your eye lines up with the direction of the reflected incoming light, so specular is view dependent. The normal also affects specular because it affects the reflected light direction. Specular power controls how shiny or glossy the surface is. A very high specular power represents a mirror-like surface while a lower power represents a rougher surface.

Opacity
Opacity controls how much light passes through the surface for translucent materials. This means a translucent surface acts like a filter on the scene that you see through it. A material with an opacity of 1 is fully opaque, while an opacity of 0 means it lets all light through.

Normal
The normal property represents the surface orientation. It can be specified per-pixel with the use of a normal map, and it has an effect on both diffuse and specular lighting.

3.3 Scaleform GFx

The way in which the application interacts with the user is extremely important. There are three type of user interfaces used in virtual environments. These are: Heads-Up Display (HUD), Menus and In-game Interfaces.

Unreal Devolopment Kit (UDK) offers two very different systems for building GUIs: Canvas, which is the Unreal Engine's native GUI support, and Scaleform GFx, which is flash-based UI. This application uses the Scaleform GFx system for the creation of the GUIs. The reason that it is preffered to Canvas, is that the process of creating the interfaces is visual and not by writing code. When using Canvas, one needs to guess the image coordinates and the objects placed on the screen, then to see the result they have to compile the project and start the application. That is a time consuming process that has to be avoided. When using Scaleform, on the other hand, visual tools are provided objects to be placed and animated. The Only downfall of the Scaleform process is that it requires learning Flash and ActionScript.

Scaleform GFx is a piece of middleware developed by Scaleform Corporation that enables modern game engines to render a Flash-based interfaces. These interfaces can be rendered directly to the screen or rendered as textures which can be used in materials and applied to geometry within the world. Being integrated in UDK, Scaleform allows UnrealScript to communicate with Flash Actionscript, which means one can build their interface's visual design and animation using some Flash IDE, add minimal Actionscript for controlling the display, load it into their game, and let Unrealscript take over all the heavy lifting and calculations.

In order to improve the interface development, Scaleform provides a variety of different tools. These are:

- Scaleform 3Di - 3D interface rendering system that can be used to tilt and rotate any 2D Flash element in 3D space, including changing the Z and X/Y rotation. Adds stereoscopic 3D UI support to give the appearance that they are floating.
- Scaleform CLIK (Common Lightweight Interface Kit) - easily customizable Flash UI component framework, including buttons, list boxes, drop down menus, sliders, trees and windows, designed by gskinner.com.
- Scaleform UI Kits - prebuilt customizable Flash-based UI templates for high-performance HUD, front end Menu and Multiplayer Lobby that are used to get started quickly.
- Scaleform AMP (Analyzer for Memory and Performance) - profiler tool used to analyze memory and performance of Flash content inside a game or 3D application while running on PC, console or mobiles. AMP gives detailed stats for CPU usage, rendering, and memory, and includes a complete ActionScript profiler with function and per-line timing.

The User Interface creation is a four-step process. The first thing to do is to create UI assets, such as, images and audio clips. Next, these assets are imported and a flash file is created at this point, some interactivity can be implemented using action script. Then the flash file is exported using the Scaleform launcher. The last step is to import the flash file into udk from where it can be used in an unrealscript class. The Figure below shows these steps graphicaly:

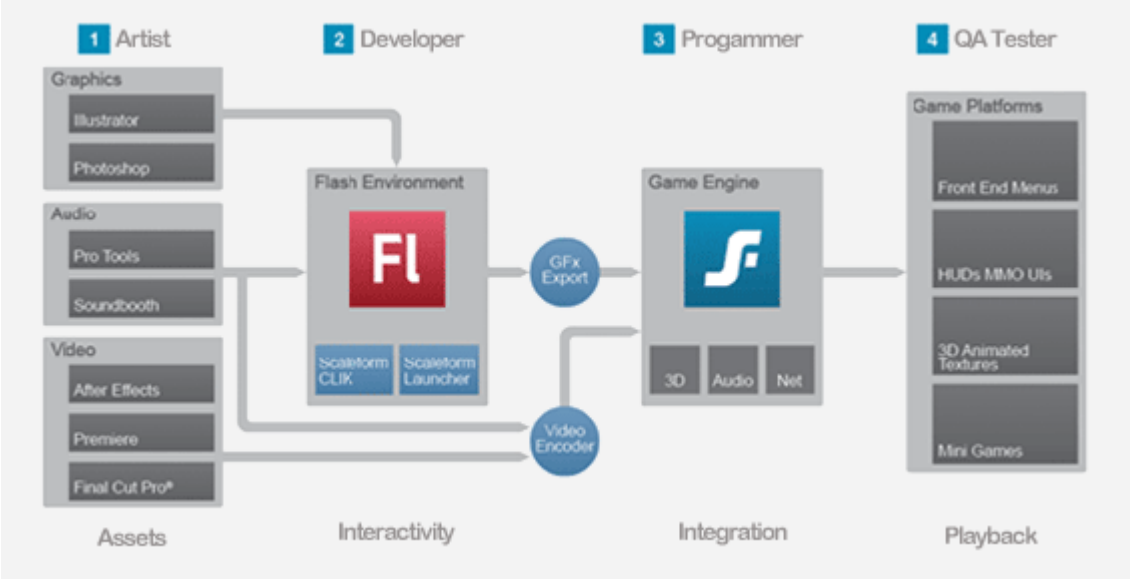


Figure 3.9. Scaleform design workflow

3.3.1 Flash Applications

As it was mentioned above, Scaleform GFx process is Flash-based, meaning that Flash files have to be created. A Flash file consists of many frames, placed in the main timeline. The flow of the frames being displayed can be changed through ActionScript - Flash's scripting language, thus allowing control over which frame will be displayed next and when this is going to happen. Each frame can have its own set of graphics, texts, movie clips and buttons and a script controlling the behaviour of the frame's components.

Adobe Flash uses a movie-making metaphor in how they define their concepts and areas of their interface. The basic terms used in flash files are:

- **Movie -** A Flash movie is the generic term for content created in Flash. It is also the overall container for Flash content, e.g. the .fla file created in Adobe Flash Professional.

- **Scene -** A scene is a "clip" of a movie. A Flash movie can have multiple scenes, each of which are self-contained units.

- **Stage -** The stage is the area that will be visible in the published Flash movie. It defines the work area within Flash where elements are placed and manipulated.

- **MovieClip -** A movieclip is a reusable piece of Flash animation. Essentially, they are self-contained Flash movies that can be used within other Flash movies. These can contain graphics, shapes, ActionScript, etc.
- **Graphic -** A bitmap graphic.
- **Button -** A button is an object that responds to interactivity such as clicks, key presses, etc. and performs actions based on those interactions.

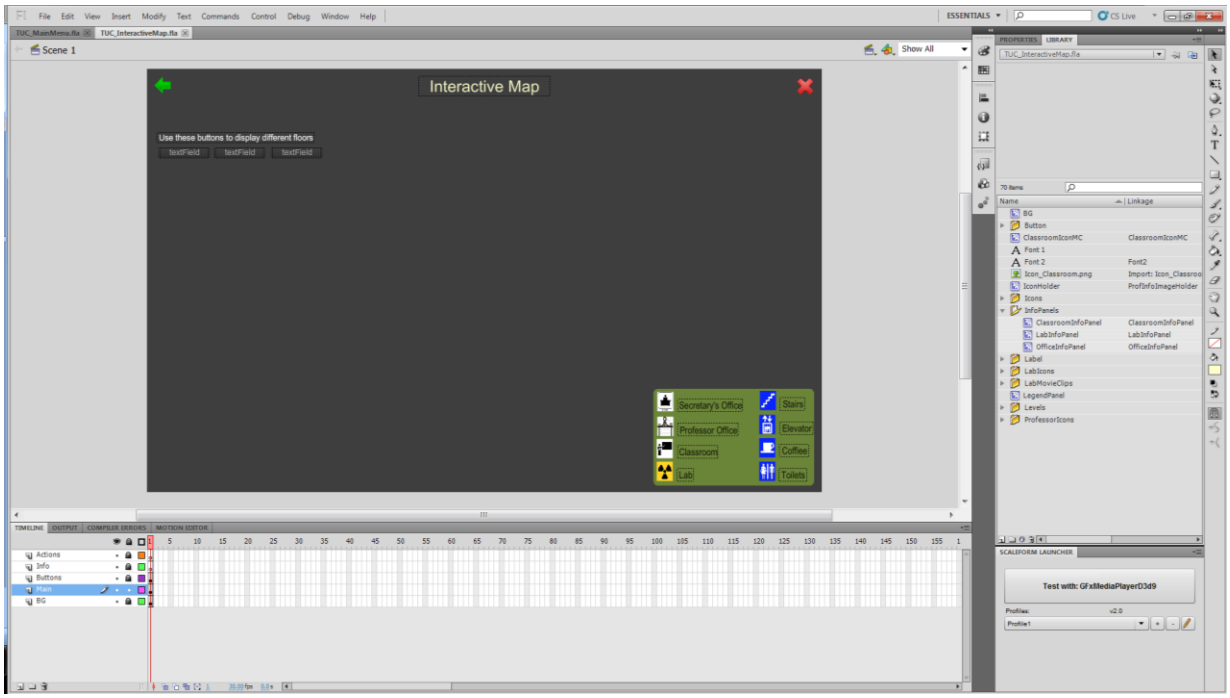


Figure 3.10. Flash Authoring environment with a Flash User Interface loaded

3.3.2 Authoring Environment for Interactive Content

In order to create a Flash application, a Flash authoring environment is necessary. There are many different Flash authoring environments available, yet, the most powerful is Adobe Flash Professional. Although it is not free, a 30-day trial version is available for download.

A freshly created Flash application is equiped with an empty stage and an empty timeline. Objects, such as movie clips, graphics, buttons, text, sounds or other Flash components, can be inserted into the application's library, or directly into the scene in the currently selected frame. Various different frames can be created, carrying different components inserted into each frame and the control of the application can be handled through ActionScript.

When the Flash application is fully developed and working, the authoring environment can compile the assets and the ActionScript, comprising the application into an executable file in SWF format. Such files can be directly executed by a Flash player and also this is the format that UDK supports.

3.3.3 ActionScript 3.0

The scripting language behind Flash is the ActionScript. UDK supports the integration of Flash applications with ActionScript version 2.0 (AS2) and version 3.0 (AS3). As AS3 is faster and more powerfull than AS2 it was preferred for scritipting the flash applications.

ActionScript 3.0 is an object-oriented language for creating applications and media-content that can then be played back in Flash client runtimes. It is a dialect of ECMAScript, meaning it has a superset of the syntax and semantics of the more widely known JavaScript. It is suited to the development of Flash applications. The language itself is open-source in that its specification is offered free of charge and both an open source compiler and open source virtual machine are available. It is often possible to save time by scripting something rather than animating it, which usually also enables a higher level of flexibility when editing.

ActionScript 3.0 executes on the AVM2 virtual machine which is available in Flash Player 9 and up.

ActionScript 3.0 consists of two parts: the core language and the Flash Player API. The core language defines the basic building blocks of the programming language, such as statements, expressions, conditions, loops, and types. The Flash Player API is made up of classes that represent and provide access to Flash Player–specific functionality.

Two main aspects of the language:

- AS3 is a strongly typed language. This means that whenever you use a variable, you must provide information about the type of data that this variable is expected to hold. If you want to use a counter and keep track of the counter progress in a variable, the type of data to be held in this variable will be of integer type (non negative numbers). To define that counter variable in AS3, you will type something like

  ```
  var counter:int = 0;
  ```

- AS3 is an object oriented language. This means that you have the possibility to split your code into specialized classes rather than writing a single program of 3,000 lines of code. This contributes to making the code easier to maintain and easier to re-use. One can then design specialized components that will be re-used across different applications. A typical example of such a component would be a calendar object that pops up to let you specify a date.

3.3.3.1 ActionScript Data Types

The primitive data types supported by ActionScript 3.0 are:

- **Boolean** - The Boolean data type has only two possible values: true and false or 1 and 0. No other values are valid.
- **int** - The int data type is a 32-bit integer between -2,147,483,648 and 2,147,483,647.
- **Null** - The Null data type contains only one value, null. This is the default value for the String data type and all classes that define complex data types, including the Object class.
- **Number** - The Number data type can represent integers, unsigned integers, and floating-point numbers. The Number data type uses the 64-bit double-precision format as specified by the IEEE Standard for Binary Floating-Point Arithmetic (IEEE-754). values between -9,007,199,254,740,992 (-2^{53}) to 9,007,199,254,740,992 (2^{53}) can be stored.
- **String** - The String data type represents a sequence of 16-bit characters. Strings are stored internally as Unicode characters, using the UTF-16 format. Previous versions of Flash used the UTF-8 format.
- **uint** - The uint (Unsigned Integer) data type is a 32-bit unsigned integer between 0 and 4,294,967,295.
- **void** - The void data type contains only one value, undefined. In previous versions of ActionScript, undefined was the default value for instances of the Object class. In ActionScript 3.0, the default value for Object instances is null.

There are additional "complex" data types. These are more processor and memory intensive and consist of many "simple" data types. For AS3, some of these data types are:

- **Object** - The Object data type is defined by the Object class. The Object class serves as the base class for all class definitions in ActionScript. Objects

in their basic form can be used as associative arrays that contain key-value pairs, where keys are Strings and values may be any type.

- **Array** - Contains a list of data. Though ActionScript 3 is a strongly typed language, the contents of an Array may be of any type and values must be cast back to their original type after retrieval. (Support for typed Arrays has recently been added with the Vector class.)
- **Vector** - A variant of array supported only when publishing for Flash Player 10 or above. Vectors are typed, dense Arrays (values must be defined or null) which may be fixed-length, and are bounds-checked during retrieval. Vectors are not just more typesafe than Arrays but also perform faster.
- **flash.utils:Dictionary** - Dictionaries are a variant of Object that may contain keys of any data type (whereas Object always uses strings for its keys).
- **flash.display:Sprite** - A display object container without a timeline.
- **flash.display:MovieClip** - Animated movie clip display object; Flash timeline is, by default, a MovieClip.
- **flash.display:Bitmap** - A non-animated bitmap display object.
- **flash.display:Shape** - A non-animated vector shape object.

3.4 Connection of the Graphical User Interfaces (GUIs) with the Application

The Scaleform GFx integration in UDK enables the use of flash-based interfaces and menus built in Adobe Flash Professional to be used as GUIs in a UDK application. The integration of a Scaleform UI inside a UDK application requires that the flash file that represents the UI is first compiled into an swf file and imported imported inside the UDK asset library. Afterwards, the imported swf file can be connected to the UDK application using one of the two possible methods. The first method is to use Unrealscript and the second is to connect the UI through the visual scripting system Kismet.

3.4.1 Connecting using Kismet:

An example can be seen in Figure 3.11, which shows the action "Open GFx Movie" firing up when it receives the "Level loaded" event. This action performs the required operations to start the Flash Movie that is inserted as an argument in its properties. It can also be set whether this movie can capture user input, as well as where to project this movie, either in the screen or on an Actor's surface.

Also, Unreal Kismet provides the action "Close GFx Movie", which handles the termination of the selected flash application.

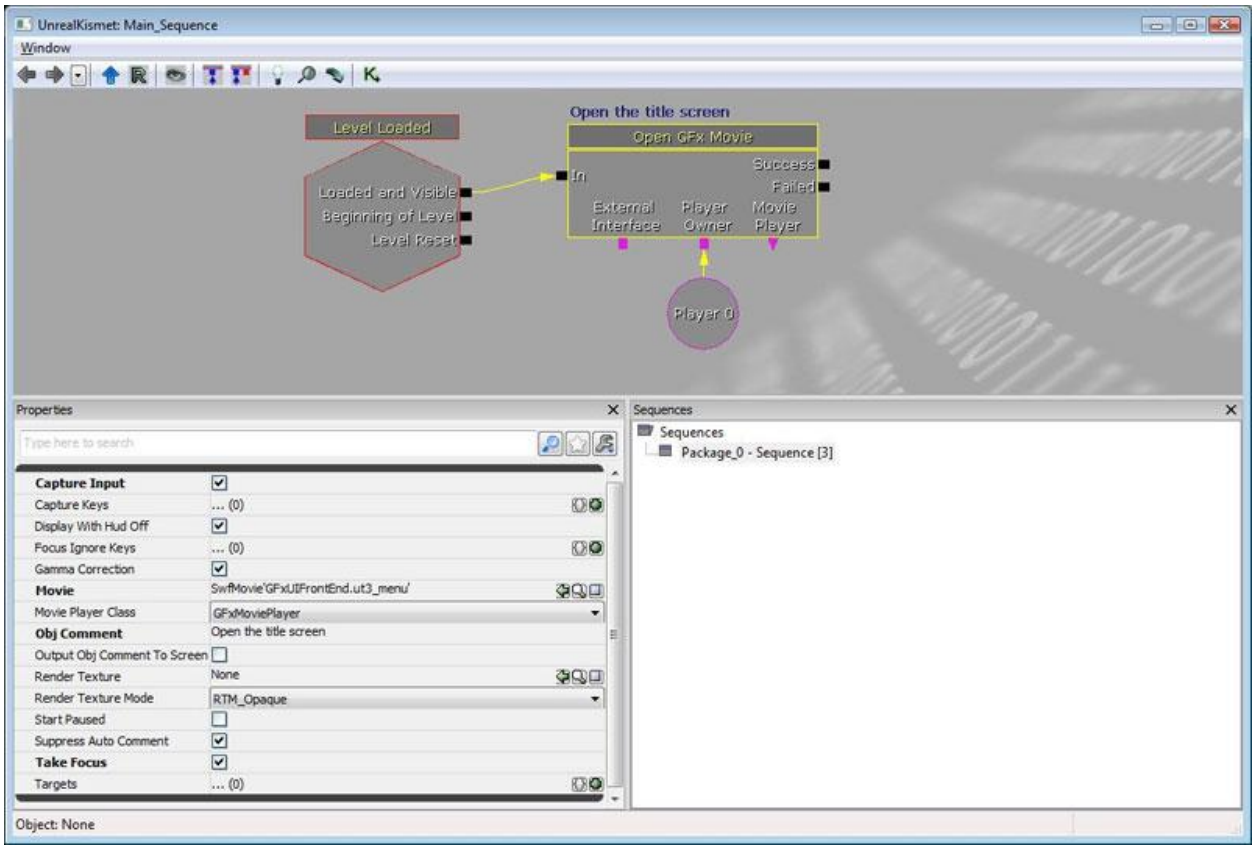


Figure 3.11. Initiation of a Flash User Interface Application through Unreal Kismet

3.4.2 Connecting using - Unrealscript:

The Scaleform UIs can be integrated in UDK application using Unrealscript. To embed a swf file into Unrealscript, an Unrealscript class that is extension of the `GFxMoviePlayer` class, has to be created. The `GFxMoviePlayer` class is the base class of all classes responsible for initializing and playing a Scaleform GFx movie. This

class will be subclassed in order to implement specialized functionality unique to the individual movie for which the player is responsible.

For example, the bellow line of code shows a custom class which extends the `GFxMoviePlayer` class:

```
class TUC_FrontEnd_ProfInfo extends GFxMoviePlayer;
```

The actual swf file is binded to the Unrealscript class in the defaultproperties section of the class using the command:

```
MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_ProfInfo'
```

Usually, in the custom class there are variables declared for caching a reference to the actual flash buttons and/or movie clips. After instantiating the class and initializing the newly created object, the above variables can be used to populate some text fields, to listen for an event etc.

The Scaleform flash files often contains some predefined components, such as buttons, lists, indication bars etc. These components are called widgets and they have to be binded to the Unrealscrip class in order to access them. Their binding is performed in the defaultproperties section of the Unrealscrip class by adding the widgets to the `WidgetBindings` array. An example of widget binding is shown below.

```
WidgetBindings.Add((WidgetName="dm",WidgetClass=class'GFxCLIKWidget'))
```

While a Flash application is playing inside the application, the UnrealScript can initiate a call of an ActionScript function and vice versa. This feature allows full interaction between the Flash interface and the application. Consequently, it is easy and efficient to create an application that initiates a Flash interface whenever it is required and then to receive the user's response and order it to stop playing.

3.5 Autodesk 3ds Max

All of the 3D models that constitute the virtual environment are created with the help of the industry-standard 3D modeling software, namely, 3ds Max.

Autodesk 3ds Max is 3D computer graphics software for making 3D models, animations, and images. It is developed and produced by Autodesk Media and

Entertainment. Although it is not free, a 30-day trial version is available for download.

Autodesk 3ds Max has modeling capabilities, a flexible plugin architecture and it can be used on the Microsoft Windows platform. It is frequently used by video game developers, TV commercial studios and architectural visualization studios.

Figure 3.12 shows the 3ds Max 2010 IDE.

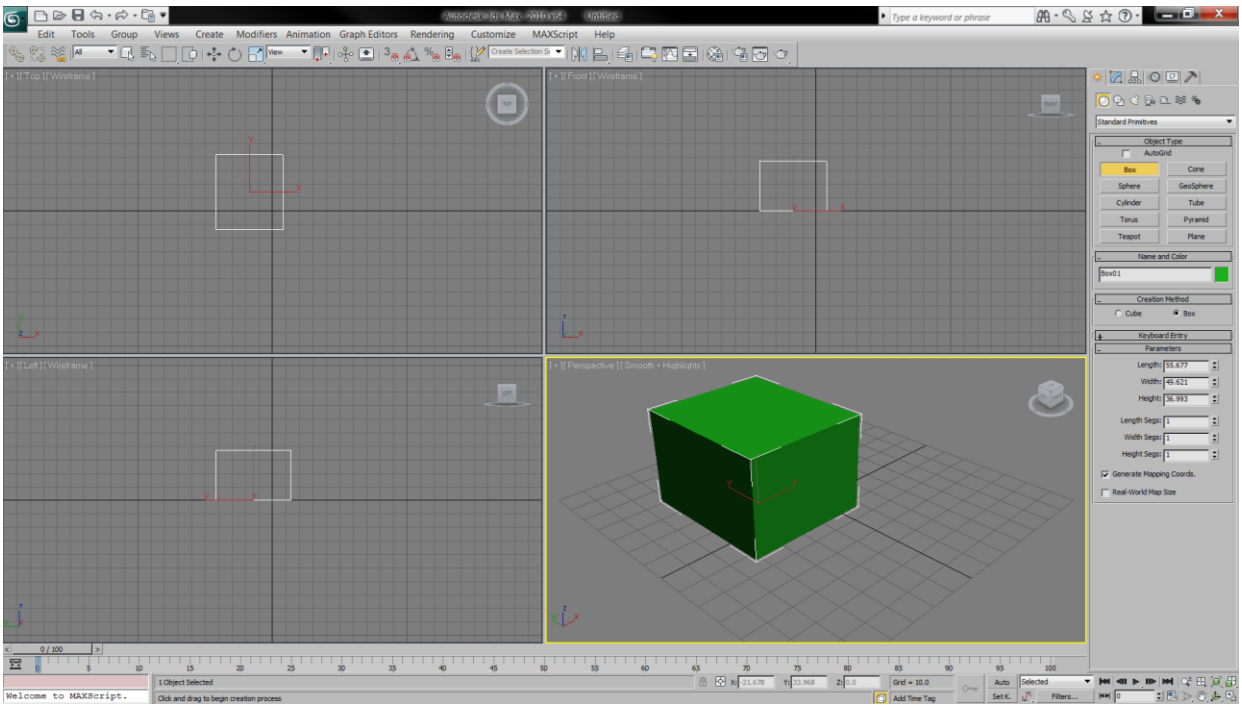


Figure 3.12. 3DS Max's User Interface

3.5.1 Predefined primitives

This is a basic method, in which one models something using only boxes, spheres, cones, cylinders and other predefined objects from the list of Predefined Standard Primitives or a list of Predefined Extended Primitives. One may also apply boolean operations, including subtract, cut and connect. For example, one can make two spheres which could work as blobs that will connect with each other.

Some of the 3ds Max Primitives as they appear in the perspective view of 3ds Max 2010:

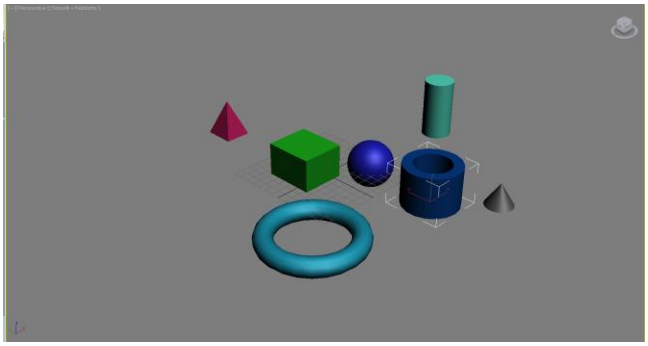


Figure 3.13. 3ds Max Standard Primitives: Box, Cylinder, Cone, Pyramid, Sphere, Tube and Torus.

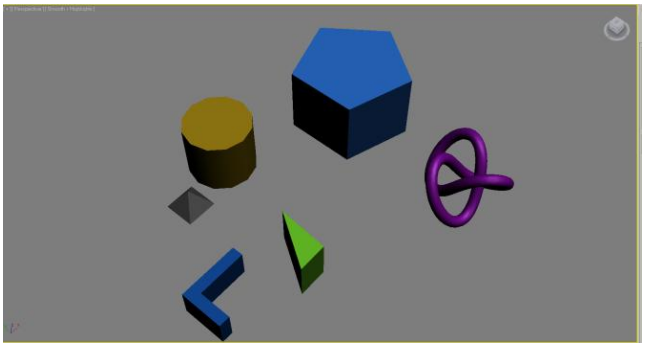


Figure 3.14. 3ds Max Extended Primitives: Torus Knot ChamferCyl, Hose, Capsule, Gengon and Prism

After creating the models (model creation is described in the section x.x), they are exported to a FBX file format. FBX is a file format owned by Autodesk and - it is used to provide interoperability between digital content creation applications and it is also the file format that UDK supports for importing 3D models.

3.6 Setting up the Environment

3.6.1 Setting up Scaleform GFx

In order to use Scalform and its visual components in Adobe Flash Professional, first it has to be set up. That includes installing the Scaleform GFx launcher - and the CLIK library. The Scaleform GFx Launcher allows us to test scenes created in Adobe Flash Professional without needing to import them into UDK and run the game.

To install the GFx launcher, we go to the Manage Extension options and install from there the extension that is found in the C:\UDK\UDK-2012-07\Binaries\GFx folder. The launcher is then accessd form the window->other panels options but it is not ready to be used yet. It first needs to be set up to access a GFx player. This is done adding the path to the player in the launcher panel.

To install the visual components that Scaleform provides we have to install the CLIK library. This library is found in the C:\UDK\UDK-2012-07\Development\Flash\AS3\CLIK folder and is added in the Source Path options of the ActionScript settings.

3.6.2 Setting up Microsoft Visual Studio with nFringe

UDK comes with a compiler for UnrealScript, however, it lacks a proper IDE. The IDE that was prefered is MS Visual Studio with nFringe module for unrealscript language support installed. Primary components of nFringe are a text editor and a source level script debugger. It provides context-sensitive IntelliSense, and support for standard tool panes like Class View and the Code Definition Window.

After installing nFringe, the first thing to do is to obtain license. In our case a non-comercial licnse was used. Next, new project is set up. This is done by creating new Unrealscript project named TUCProject in the directory C:\UDK\UDK-2012-07\Development\Src. As a final step, the path of unrealscript compiler is specified in the projct properties.

4 Virtual Environment Creation

The Virtual Environmenet represents an existent building called 'The Building of Sciences' of the Technical University of Crete (TUC). The building "is placed" on thousends of square meters, has four floors and complex structure which consists of many curved walls. Figure 4.1 shows the architectural beauty and complexity of this building.

Figure 4.1. The Sciences Building of TUC

The Virutal Environment consists of lot of different objects. These objects are called Actors in UDK and these can be: Geometry, Materials, Triggers, Ligths, Sounds etc. In the next paragraphs the creation of the geometry and the materials is thoroughly described.

4.1 World Geometry

The main structures in the virtual environment, as well as the decorations are considered *world geometry*. There are two types of Actors in UDK that are used to create the world geometry: BSP Brushes and Static meshes.

In order to precisely reconstruct the Science Building of TUC, architectural plans and photographs of the building were used. The following two figures show the complexity and the building's structure through architerctural floor plan and elevation (face).

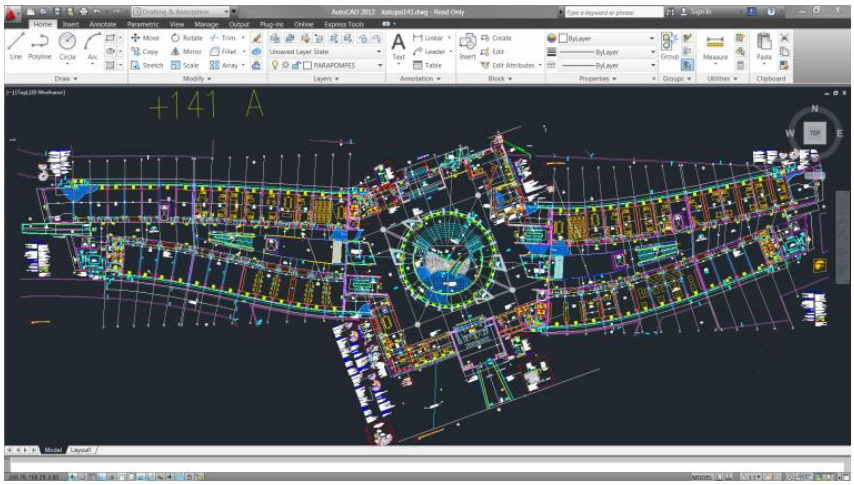


Figure 4.2. Floor plan of the first floor

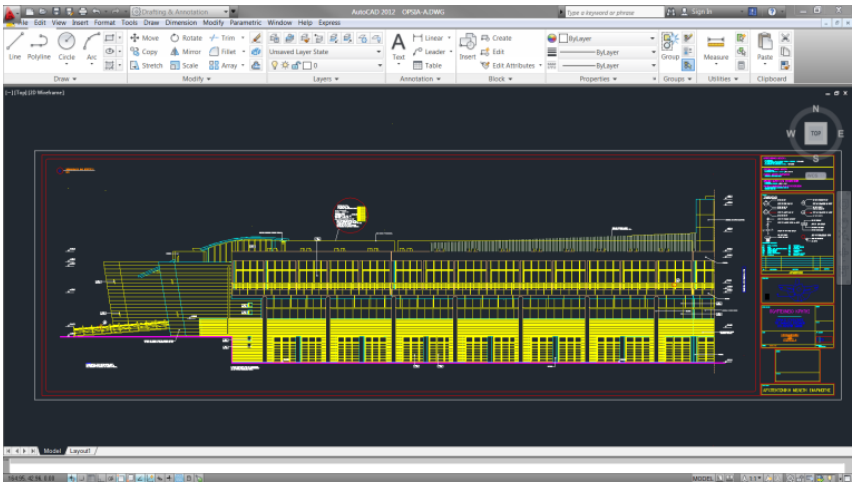


Figure 4.3. South elevation of the ECE department

The purpose of using the architectural drawings in building a huge VE from an actual building is to determine the shape, the location and the size of the main structures that appart that building.

At this stage creating small geometry objects and assembling them in the UDK without knowing their correct position is almost imposible task if we wanted precise design. In order to overcome that problem, a big 3D object which was used as placeholder for the geometry pieces, was created. The procedure of creating this place holder and the final results are presented next.

The first step was to remove the unimportant, redundant lines from one of the drawings (say floor 1) and to leave the lines that represent the main building structures.

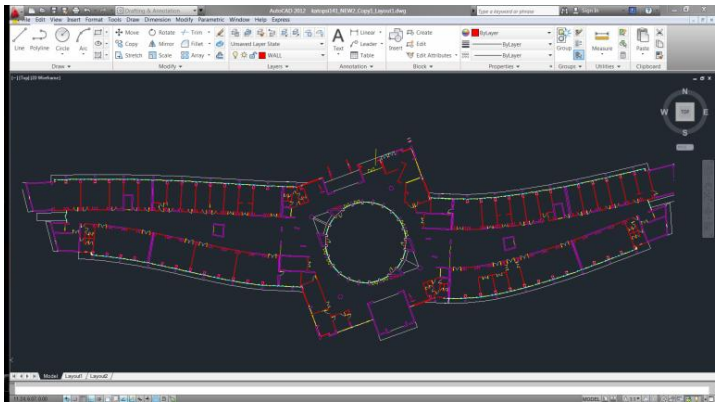


Figure 4.4. The redundant lines are removed

The next step was to import the drawing in the 3DS Max and to scale it correctly. To be more precise, the drawing had to be enlarged 52,5 times.

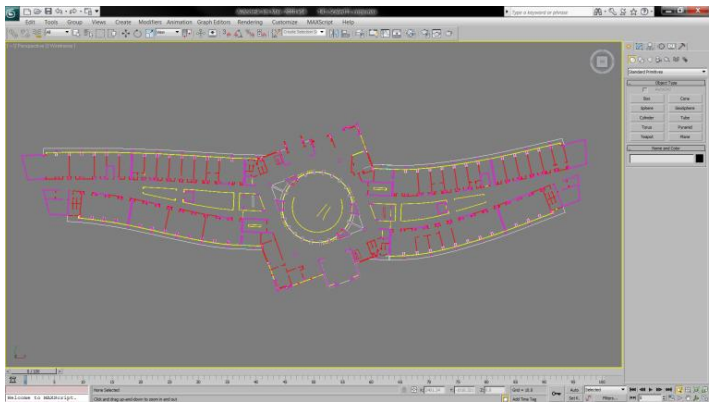


Figure 4.5. Floor plan imported into Max

In order to extrude walls from the drawing the lines have to be closed. Because that wasn't the case we had to create new closed lines on the top of the already existing. Inside the 3D modeling studio, the lines were created and attached one to another and the result was a complex shape as shown in Figure 4.6.

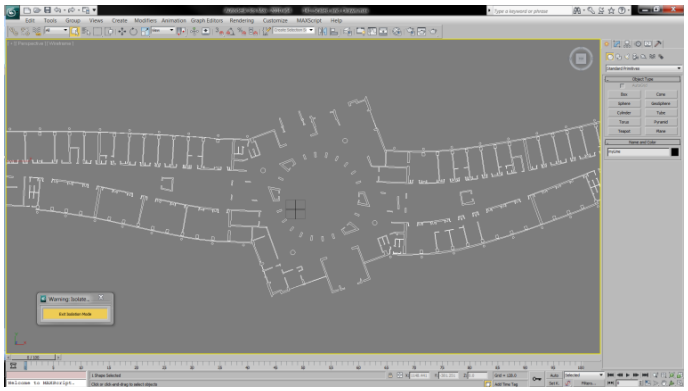


Figure 4.6. The new closed line

When the new closed shape was created, it had to be converted to editable polygon in order to make a 3D object out of it. The 3D object was created by extruding the shape usning the Extrude modifier.

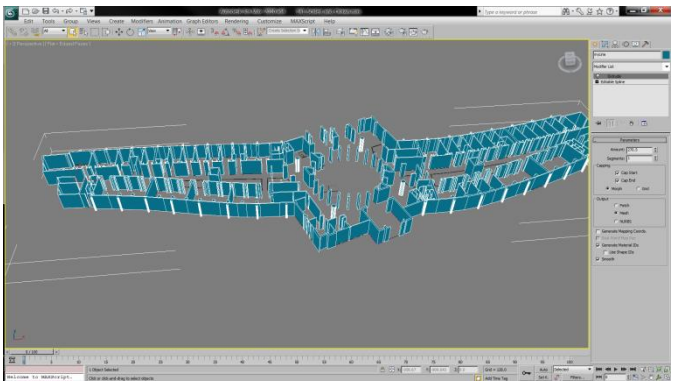


Figure 4.7. The final 3D extruded object

The finial step was to import this place holder in UDK as shown in Figure 4.8.

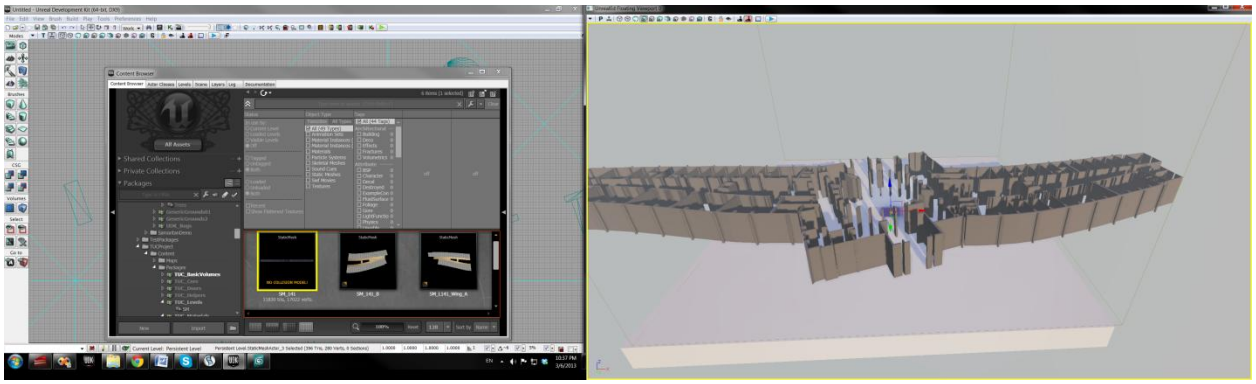


Figure 4.8. The place holder imported into UDK

Note that this 3D object can't be used for modeling the world for many reasons. These are: the object is static - neither doors nor windows can be created, the object is too complex for applaying materials to it, the object can't have correct UVW map etc. This place holder helped us to position the created geometry in the world correctly.

The Virtual Environment consists of more than 1100 objects, 2.000.000+ polygons and 1.500.000+ vertices. Next, the creation of the general geometry is explained.

4.1.1 BSP

BSP brushes are the most basic building blocks in Unreal Engine. They are used to carve out or fill in volumes of space in the world. These used to be the primary method of creating world geometry in previous versions of the engine, but they have been replaced by more specialized and efficient types of geometry, namely Static Meshes.

The BSP brushes (volumes) are created using the builder brush tool. The builder brush is the template out of which BSP brushes are created inside UDK. One can use this tool to crate additive, subtractive, blocking and other volumes when creating their level. The builder brush tool show up only in the Unreal Editor and not in the created application. To see the created volume the level has to be rebuilded.

The geometry created using BSP brushes is shown in the next two figures.

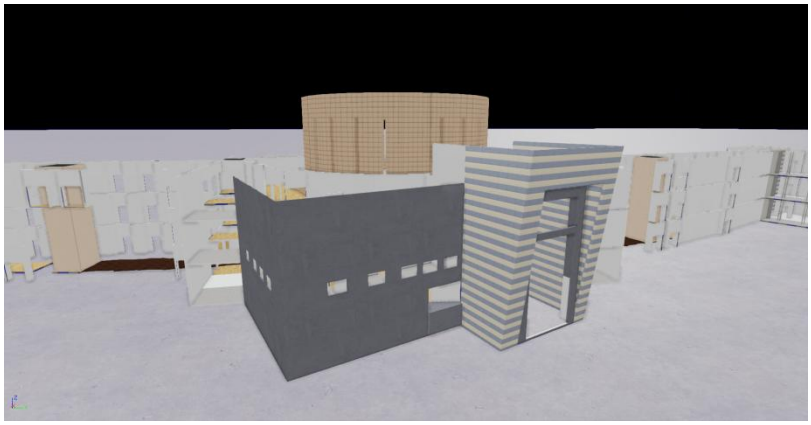


Figure 4.9. Application's BSP geometry (lit mode)

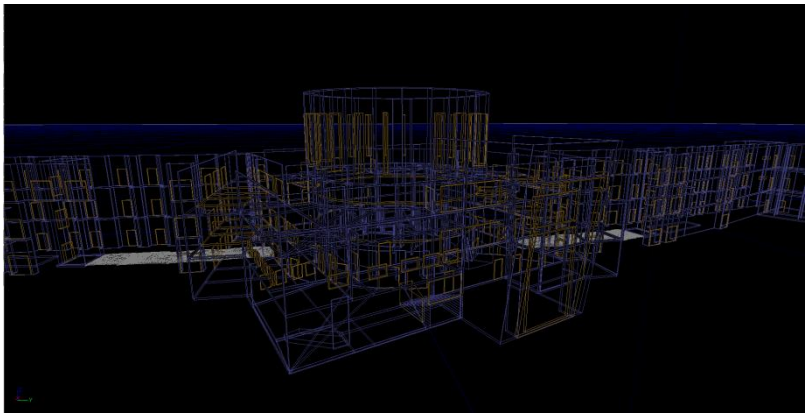


Figure 4.10. Application's BSP geometry (wireframe mode) - Additive Brushes (blue), Subtructive Brushes (yellow)

4.1.2 Static Meshes

A Static Mesh is a set of polygons that is drawn by the hardware of the graphics card. They are much faster, can handle many more polygons, are invulnerable to BSP holes and look better than BSP brushes. Static Meshes get their name since they are drawn as meshes that never change, so they can be cached in the video memory which leads to better efficiency and performance. [UDN]

All of the static meshes populating the virtual environment are created manually with the help of an industry-standard 3D modeling software - 3DS Max 2010. The method that is used for the asset creation is described next.

The first thing that is done, is setting up the 3DS max to use generic units. This is done in order to have one-to- one unit mapping with UDK. The shape and the size of the assets is determined with the help of the provided floor pans and the taken pictures.

The geometry for the most of the assets apparting the virtual environment is fairly simple and thus, the most 3D models are created out of standard primitives (box, sphere, cylinder...). After giving shape to the model, two UV channels, one that contains the model's texture data and the other containing its lightmap data, are set. The first one is set up by creating a material in the material editor and adding it to the created asset as the Figure 4.11 depicts.

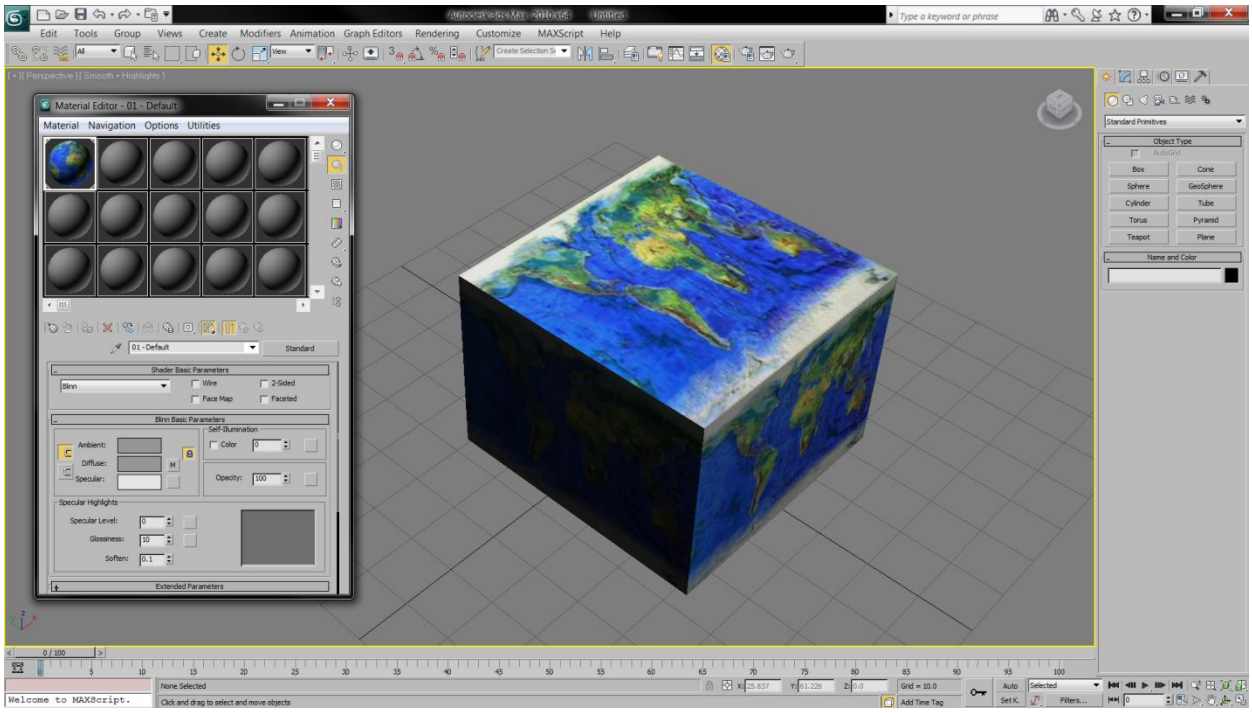


Figure 4.11. Simple 3D object with material applied to it

The second UV channel is needed by UDK so that illumination and shading effects are correctly applied on the object. This channel is created in 3ds Max by applying a UVW unwrap modifier on the editable mesh that is exported and then, in the editor that pops up select to automatically flatten the UVW mapping. An example of this can be seen in Figure 4.12.

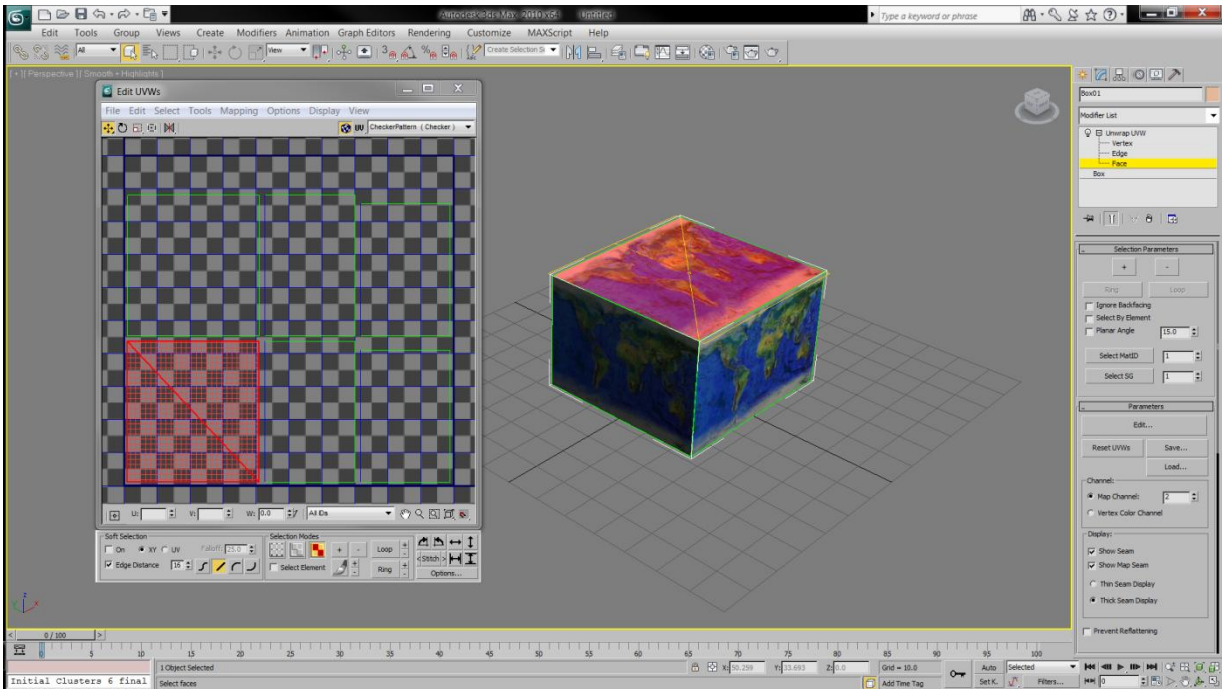


Figure 4.12. The UVW unwrapping of an object in 3ds Max. This UV channel is used by UDK in order to realistically illuminate an object

Before the Static Mesh is exported, it's pivot point is set at the world origin (0,0,0). In the most cases the pivot point is placed on the corner of the mesh to allow for proper grid alignment once in UDK. The 3DS Max's Snap to Grid function and the alignement tool are used for setting correct pivot point.

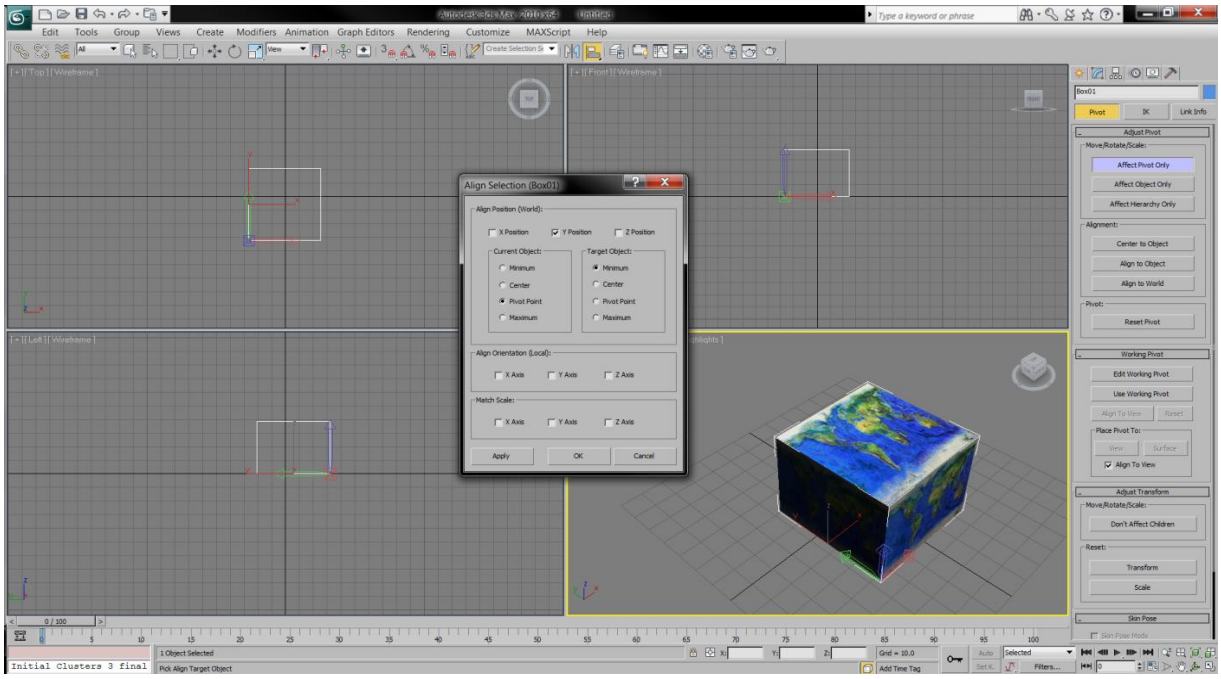


Figure 4.13. Setting a pivot point of a 3D model

The static mesh is exported as an FBX file. An FBX is a file format owned and developed by Autodesk. It is used to provide interoperability between digital content creation applications such as Autodesk MotionBuilder, Autodesk Maya and Autodesk 3ds Max [wikipedia]. UDK features an FBX import pipeline which allows simple transfer of content from any number of digital content creation applications that support the format.

The above methodology was used for creating simple box-shaped Static Meshes. Some of the needed assets are curved shape and those were created using different techics such as, Line Extruding and Loft Shaping.

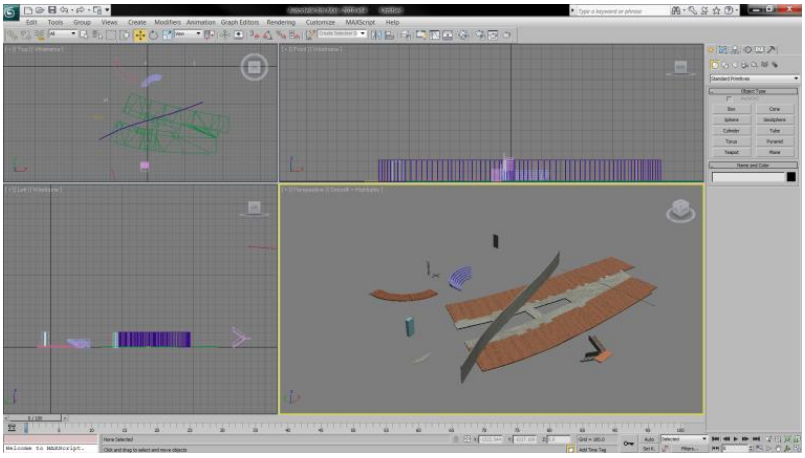


Figure 4.14. Some of the created static meshes

4.1.3 Importing the Static Meshes

Once created, the static mesh is ready for import into UDK. Inside UDK, the created 3D objects are imported by selecting the import option in the Content Browser of the Unreal Editor. UDK reads the file containing the exported 3D objects and recreates the geometry of the objects. It also creates the different material slots for each part of the 3D object and initializes the UV channels of the object.

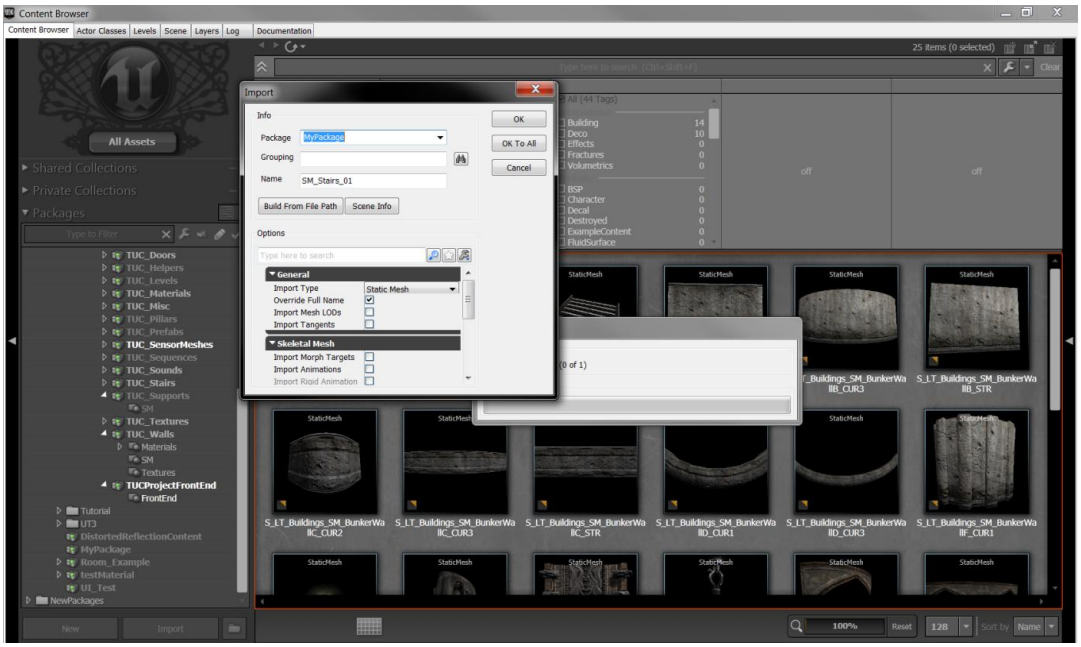


Figure 4.15 Importing external 3D model into UDK

The most of the collision detection requirements of the application are automatically performed by UDK, after assigning a collision data to each 3D object, according to its geometry. UDK provides several different ways of creating a collision data for a given geometry.The more complex collision data produce, the more realistic collision detection is achived. However, they suffer from increased need for computation. For the purposes of the application, the collision detection mechanism was required to simply prevent the user from intersecting other objects, or passing through walls. So, simple collision data is chosen, thus, decreasing the computational needs.

An example of the creation of a collision is presented in Figure 4.16, which shows the – imported in the UDK – Stair Rails 3D object opened in the Unreal Editor. The green box surrounding the geometry of the object is the collision vector that is created for that object. Although the geometry of the actual 3D object is quite complex in terms of polygon count, the collision vector is extremely simple, represented by a simple box. This does not offer great accuracy in collision

detection, but nevertheless, it serves the purpose of the application being computationally efficient.

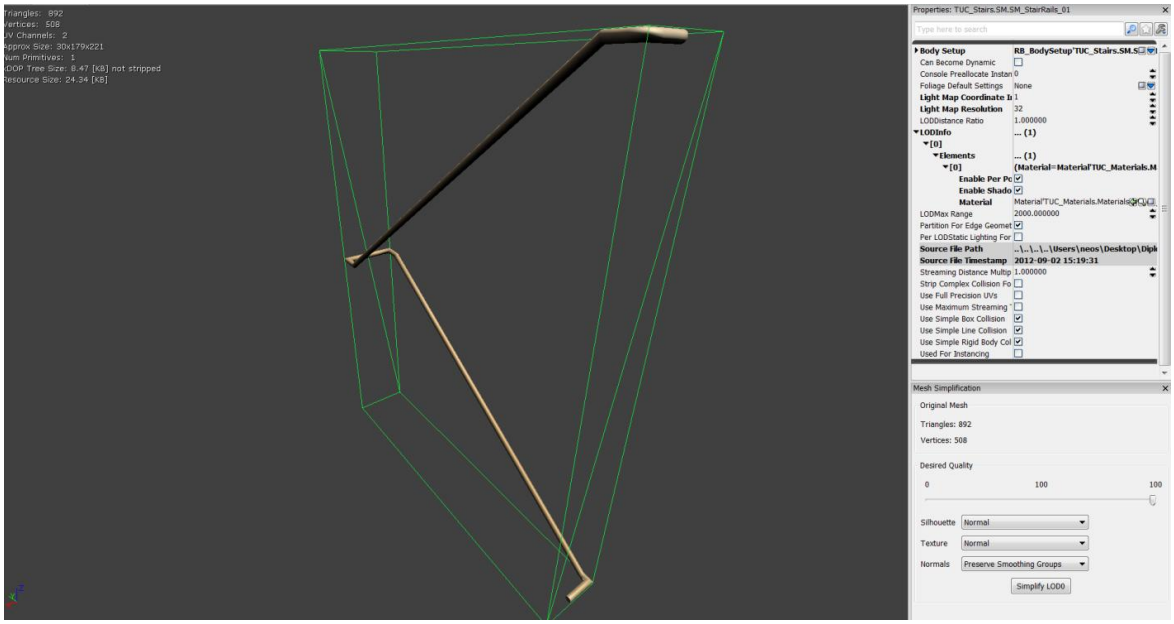


Figure 4.16. Stair rails 3D model with a simple collision applied on it

For some of the static meshes, the automatic collision creation is either too complex or not functional at all. In such cases, collision data is manually set up in the Unreal Editor by addind blocking volumes. Blocking volumes are created using the brush builder tool. In order to create the volume, the shape of the builder brush is made first. Then, the collision is created by right clicking on the Volume button in Unreal Editor and selecting BlockingVolume from the list. The collison data for the stairs which is manually created is shown below:

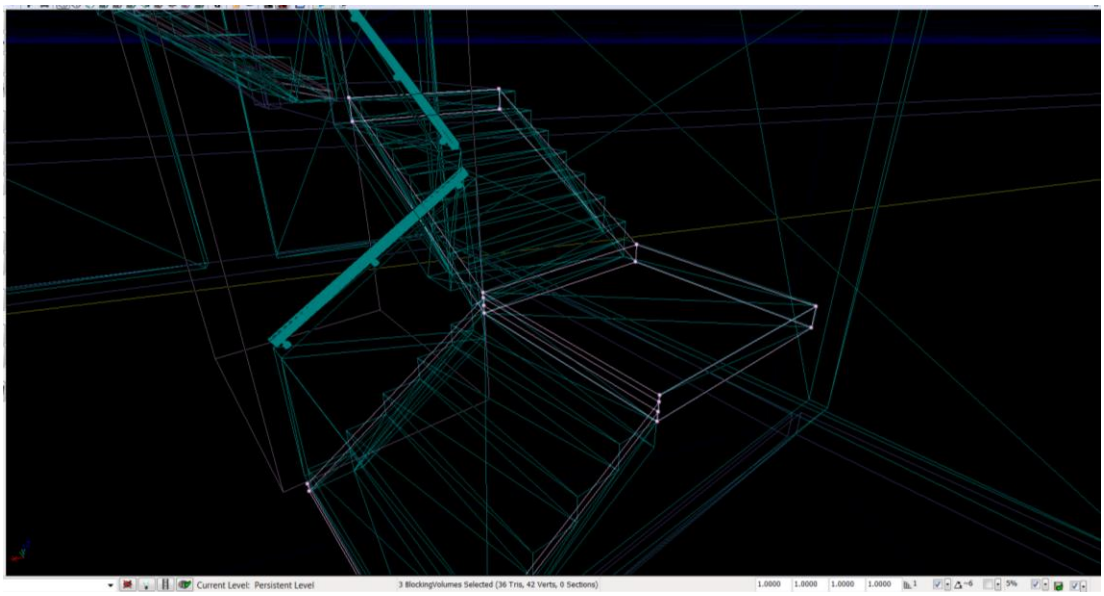


Figure 4.17. Collision data created from blocking volumes (pink lines)

4.2 Materials

Materials are the most important factor in the final look of a 3D virtual environment. They control the look of every object created - Static Mesh or BSP volume. A material, in essence, is a small computer program that describes how a surface appears. Materials can be used for adding color to the objects, making them shiny, reflective, and even transparent.

4.2.1 Textures Vs Materials

There is a difference between textures and materials in UDK.

Textures are a single file. It is usually a diffuse, specular or normal map file that is created in an image editing software such as Adobe Photoshop, GIMP or other software.

Materials are made up of various textures combined together inside the Material Editor in UDK. Material includes various textures and material expressions that create a network of nodes. The final result is a material that can be used for applying one's BSP geometry or Static Meshes.

4.2.2 Textures

Some of the textures are downloaded from image repositories while the others are created externally within the image editing application - Photoshop, and then imported into Unreal Editor through the Content Browser.

Before importing, all of the textures are scaled to be power of 2, as the UDK supports only those dimmensions. The textures were saved as either .bmp file either .png, as jpeg is not supported by UDK.

The normal maps are created using the Nvidia's normal map filter. Nvidia normal map filter is a plug-in for photoshop and it is used to create normal maps from grayscale images. Hence, the image that is used for creating the normal map is first grayscaled using the Hue/Saturation options. Next, a highpass filter is run on the image to sharp it a little bit. The last step is to run the Normal Map Filter on the image. In the most cases the default settings are used.

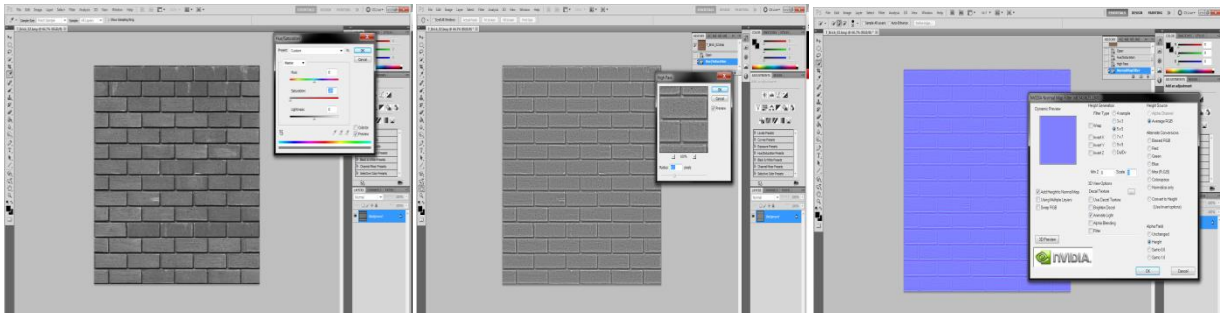


Figure 4.18. Normal Map creation workflow

4.2.3 Material Creation

The materials used for coating the 3D objects are created in the UDK's Material Editor. The first thing in creating materials is to import texture images. New material is in the content browser by clicking the right mouse button and selecting the New Material option from the contex menu. In the pop-up panel a package name is specified, as well as a group name and the name of the material following some name convention. Figure 4.19 shows creating new material

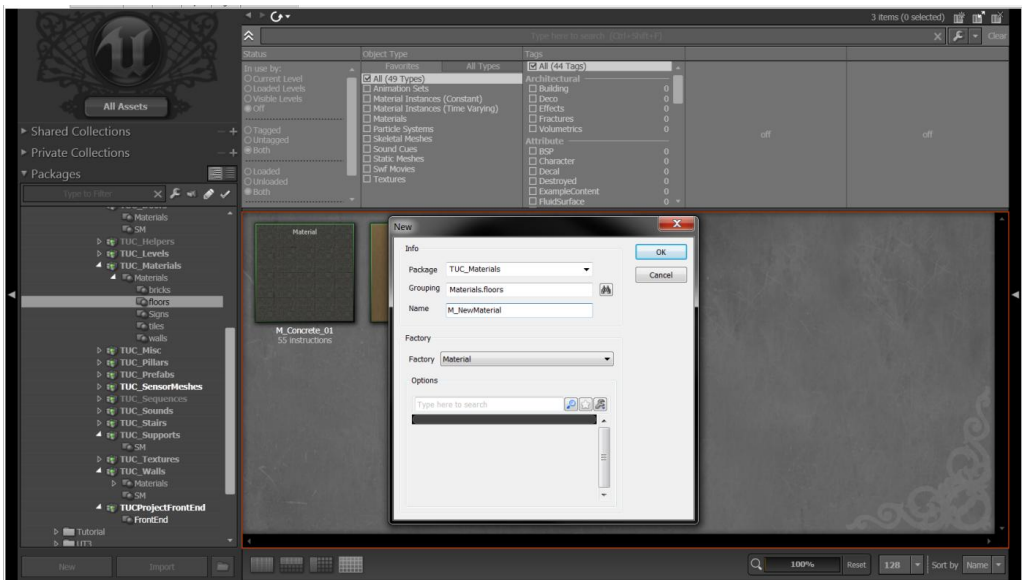


Figure 4.19. Creating new material in UDK

After creating a brand new material, the UDK's Material Editor is used for modifying it. As already mentioned, UDK's Material Editor offers the ability to not only set a diffuse texture or expression for a material, but also alter its properties, such as setting the specular color and power of the material, or defining a normal map, which can realistically differentiate the lighting of a rough (not smooth) surface. For example, Figure 4.20 shows two different materials in the Material Editor, applied on a sample sphere object. The first material on the left is a matte grey one, with no specular properties and consequently does not produce any lighting effects on the object's surface. The material on the right has a white color connected to its specular property, with high specular power (20.0), so it produces a specular effect due to the light that the object receives.

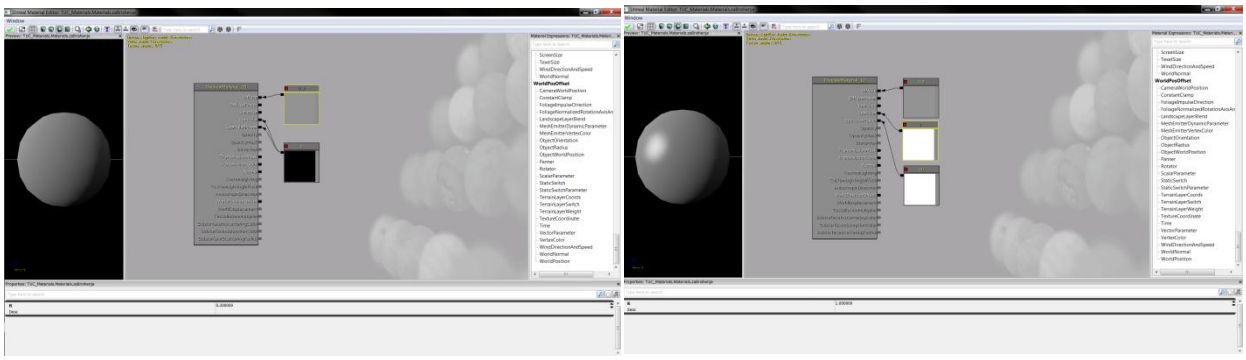


Figure 4.20. On the left is a grey matte material, with no specular color. On the right is the same grey material with white specular color

There is, also, the option provided to set up a normal map for a material, which helps Lightmass to calculate the light that bounces on the object when each specific

material is applied and scatter it according to the supplied normal map. This is very helpful in order to create the illusion of rough surfaces on objects with smooth geometry, such as the wooden surface of a table, or a carpet.

The geometry of a wooden floor is very complex. When creating a normal map for the wooden material, although the geometry's surface is smooth, the light should scatter as though bouncing on a rough surface, as it can be seen in Figure 4.21.

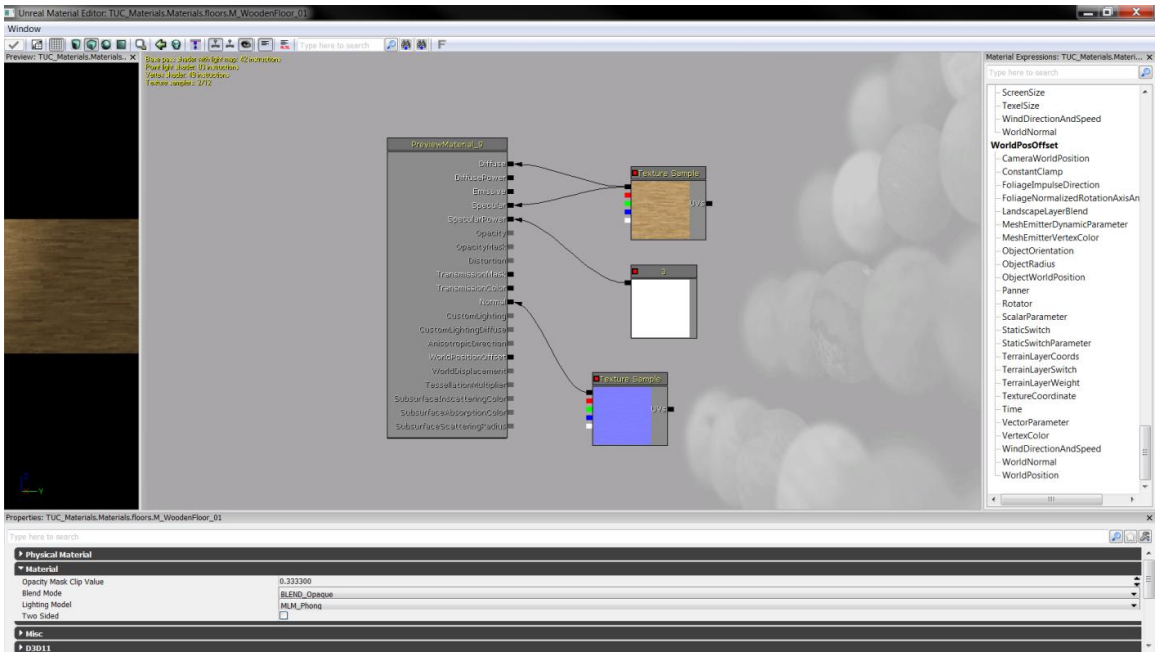


Figure 4.21. A material for the wooden floor surface with a normal map defined. As can be seen from the sample cube with the material applied on it, the lighting effects seem as though falling on a rough surface

4.3 Doors

This section deals with creating a door opening and closing as the player interacts with it. The doors are created using Static Meshes, Matinee Animations, Prefabs and Kismet sequences. The method used for creating a simple single door is described below. This method is general and it is used for creating all the buildings' (facility) doors.

Door Geometry

First, the static meshes that apart the door are created in 3ds MAX. These include: a door frame, a door handle, a door lock, door hinges and the door panel. Creating static meshes is thoroughly described in the section x.x. After importing the static meshes in UDK, they are, all except the door frame, placed in the virtual scene from where they are converted to InterpActor objects, to enable interpolation of the door parts, in this case rotation.

The InterpActor (interpolating actor) is a special dynamic version of the StaticMesh Actor. It supports a variety of physics types, it uses dynamic lighting, it can be controlled in a variety of ways with Unreal Kismet and Matinee, and it can play a number of sounds for specific actor states. When the static mesh actors are converted to InterpActors they lose the collision properties, which means that they do not block the player, so they are added manually. This is done by setting the collision type to "COLLIDE_BlockAll".

In order to enable user interaction with the door, a trigger actor is added to the level and attached to the door panel. The trigger actor is listening when the player gets to the proximity of the door. For the user to activate/use the trigger from specified distance, the trigger's collision radius is scaled from its properties. The next step is to wire up this trigger to the Door Panel in Kismet, but before this is done, a Matinee object for the door animation has to be set up.

Animating the door in Matinee

The following step is animating the door. Only the door panel is animated in Matinee and the other door parts that have to move together with the door panel are attached to it. The attaching is done in the Unreal Editor by selecting the other parts that include the door and that have to move together with the door panel, and by setting their Attachment properties. To be more precise, the door panel is added to the "Base" property of the selected actors and it is ensured that the "Hard Attached" property is checked.

In Kismet, a new Matinee object is selected in the context menu. After the Matinee object appears, the UnrealMatinee Editor is brought up and/which allows us to animate the Door. With the InterpActor Door Panel selected, a new empty group is created in the Matinee editor. This action creates InterpActor variable that will hold the animation. What is done next is creating a Movement Track which represents the animation of the door opening. The movement track comes with a keyframe that is established with the initial position of the selected InterpActor (the door panel). In the UnrelEd, the Door Panel is rotated 90 degrees, and a keyframe that represents the end of the animation, is added to the timeline at 1sec in the matinee editor. This completes the door open sequence.

Kismet

In order to achieve interaction, the trigger has to be wired with the Matinee Object. Once the trigger is selected, in Kismet, a new trigger used event is created from the context menu. Trigger allows us to interact with the trigger by pressing the 'E' button. The door is toggleable, that is, when the button E is pressed, if the door is open we want to close it, and when the door is closed, pressing the e key will open the door. In order to allow multiple door openings and closings, a switch action is added between the trigger event and the matinee object.

The switch action has a definable number of outputs, and increments to the next one every time it is fired. When the switch is activated it looks at the Index variable to determine which outputs should be activated. Two output links are added and the looping property is checked so that the switch loops back to the first output once all outputs have fired.

At this stage the "Used" output of Trigger Event is connected to the Input of the Switch. Next, the first Switch output (link 1)is hooked up to the Play input of Matinee Object and the second output is hooked up to the Reverse input of the Matinee Object. This means that when the trigger is pressed once, it opens the door. On the second time it will close (reverse) the door. And the sequence will loop because the switch is set to Looping.

Prefabs

Prefabs are collections of actors and associated Kismet that can be stored in a package and easily instanced into a level. Prefabs can have changes made to them in a level, and those changes can then be propagated to all existing instances of that Prefab.

5 Functionality Creation (Scripting)

The Unreal Engine contains a large amount of UnrealScript classes already implemented in several different packages which the user can put in action to extend its own classes. The first thing to do is to understand what these classes are and how they can be used in one's own application. The existing classes mainly provide basic and generic functionality. They are essentially a part of the engine itself and not part of "the game".

The typical approach to class design in UnrealScript is to make a new class which extends an existing class that has most of the functionality that one needs

Some of the more important class packages that the programmer extend from are: `Engine`, `GameFramework`, `UDKBase`, and `UTGame`. While the `UTGame` classes implement a lot of First Person Shooter Game (FPS) functionalities, such as AI, Networking, Scorebords, Menus etc., in this case it is decided to extend from the lightweigth package `UDKBase`. `UDKBase` is, in essence, the bare bones of a project, meaning that there is much more freedom with functionalities, but still there are many issues left to deal with.

5.1 The Three Fundamental Classes

The three fundamental classes that form the framework skeleton of this project are `TUCInfo`, `TUCPawn` and `TUCPlayerController`. All three classes extend from `UDKBase` and their purpose is to to create a working, yet generic gameplay experience.

The `TUCPawn` and `TUCPlayerController` construct the character that the user controls while navigating in the virtual scene. The `TUCPawn` is the physical representation of the player in the world. It has a mesh, collision, and physics enabling it to handle all the functionality involved with the physical interaction between the character and the world. In unrealscript each `Pawn` can have a single `Controller` at any time. In this application the `TUCPlayerController` is the `Controller` associated with the `TUCPawn`. As its name suggests, `TUCPlayerController` takes care of telling the `TUCPawn` what to do and how to behave. It is essentially the brains behind it. The main purpose of `Controllers` is to accept input from the player or other stimuli in the world, to process that input and act accordingly.

The "main" class of the application is the `TUCInfo` class. It is responsible for telling the engine which classes to use for `Controller`, `Pawn`, `HUD`, etc. Generally, this kind of classes (gametype classes) do much more than assigning `PlayerControllers`, `Pawns` and `HUDs` used in the application. They usually determine the rules of the game, the conditions under which the game progresses or ends etc. These three classes are presented below.

TUCPawn

The visual representation of the character and the logic for determining how it interacts with the physical world is encapsulated in this class.

```
class TUCPawn extends UDKPawn;
```

When a virtual scene is started, a new `TUCPawn` is instantiated, as instructed from the `TUCInfo` class. It extends from the `UDKPawn` class which means that the logic for interacting with the environment does not need to be implemented. This class defines some characteristic properties for the Pawn, such as the height, the speed, the acceleration and the body of the Pawn which is represented by a Collision Cylinder.

An important aspect that is configured through the `Pawn` class is the collision cylinder of the `Pawn`, which is taken into account by UDK to detect collisions between the `Pawn` and the objects in the synthetic scenes. A primitive cylinder object is with radius of 15 units and height of 20 units, is created in the default properties and then it is added as a component of this `Pawn`. Next, The eyes of the `Pawn` are set to 50 unreal units and the ground speed is set to 500 uu. The acceleration rate is modified to 400 uu.

```
DefaultProperties
{
      Begin Object Name=CollisionCylinder
            CollisionRadius=+0015.000000
            CollisionHeight=+0020.000000
      End Object
      CylinderComponent=CollisionCylinder

      BaseEyeHeight=50.0
      GroundSpeed=+00500.0000
      AccelRate=+000400.0000
}
```

TUCPlayerController

The `TUCPlayerController` class is an extension of the `UDKPlayerController` class. It is used as the `Controller` of the `TUCPawn` and it takes control of the created `Pawn` when the application starts, as instructed by `TUCInfo`. This class controls the main flow of the aplication. It handles all aspects of the application, such as navigation, interactions etc. The menu manager that is responsible for the flow of the user interaction is implemented here. The most of the other classes are instanciated and controlled in this class. All the functionalities that the application supports are started and ended here. It is understood from the above statements that the `TUCPlayerController` class is quite large. For this reason, only the member variables and function names are presented here. The function implementations are described in the sections where the related functionality is thorougly explained.

This class extends from the `UDKPlayerController` class.

```
class TUCPlayerController extends UDKPlayerController;
```

The variables used in this class are:

```
/* Navigation related variables */
var TUC_NavArrow NavArrow; //arrow that shows us the path to the destination
var String NavTargetString; // take values in TUC_FrontEnd_ShowPathMenu
var TUC_NavTargetNode NavTargetObject; // holds the target of our search
var bool bBtnSearchPressed_FE_SPM; // the search button
var bool bNavFindPath;  // flag for activating the pathfinding
var Vector NavArrowLocation; // the location of the arrow

var TUC_TCPLink TCPLink;

/* FrontEnd varibles */
var TUC_FrontEnd_MainMenu FE_MainMenu; // the main menu

var TUC_InfoData Info;  // data for professors, classrooms, labs
```

The functions that this class implements are given below, together with a simple description:

`PostBeginPlay()` - Initialization

`TUC_ShowMenu()` - This is a exec function that is called when the M button is pressed by the user.

`MenuManager()` - Handles all user input from the menu.

`PlayerWalking` – State in which the user is found while moving.

`FindPath()` – The actual path finding occurs in this function.

`SpawnAndAttachNavArrow()` – Spawns the navigation arrow and attaches it to the `Pawn`.

`NavFindTarget()` – Iterates through `TUC_NavTargetNode Actors` to find the target object and returns it.

`GetTriggerUseList()` – Enable use command for our custom triggers.

`Teleport()` – Teleports the user to the desired location.

TUCInfo

This is the class of the application that defines the main properties, such as the `Pawn` that is used in the application and the `Controller` that handles the `Pawn`. This class extends the `GameInfo` class, as it can be seen in its declaration:

```
class TUCInfo extends UDKGame;
```

When the user starts the application, the `TUCInfo` class creates a controller for this user. In order for the `TUCInfo` class to create the correct class, the code shown below is added to the end of the source file. In the below code chunk we see that the TUCPlayerController class is used as the Controller class The `Pawn` used in this application is instanciated from the `TUCPawn` class and the `HUD` is created from the `FrontEnd_HudWrapper` class.

```
DefaultProperties
{
      PlayerControllerClass=class'TUCProject.TUCPlayerController'
      DefaultPawnClass=class'TUCProject.TUCPawn'
      HUDType=class'TUCProject.TUC_FrontEnd_HudWrapper'
}
```

5.2 Application Data

Configuration File

The information about the level height or width, or the names of the professors, or the number of classroom seats, are stored in an .ini configuration file named TUC.ini. This configuration file consists of two sections of key-value pairs. These are:

```
[TUCProject.TUC_FrontEnd_Hud]
[TUCProject.TUC_InfoData]
```

The first section `[TUCProject.TUC_FrontEnd_Hud]`, refers / (is associated) to the `TUC_FrontEnd_Hud` class which is responsible for displaying objects on the `HUD`. The information about the level/world width and height is added after the line `[TUCProject.TUC_FrontEnd_Hud]:`

```
WorldWidth=9320
WorldHeight=2900
```

These two variables, which are used in the calculation of the position of the user on the minimap, are declared in `the TUC_FrontEnd_Hud` class, as follows:

```
var config float WorldWidth, WorldHeight;
```

The second section is associated with the `TUC_InfoData` class. This class uses the configuration file for storing information for the professors, the labs and the classrooms. An example of the stored data is shown below:

```
Professors=(ID="KaterinaMania",PName="Katerina Mania",Position="Assistant
Professor",Department="Department of Electronic & Computer
Engineering",Sector="Computer Science Division",Speciality="3D Graphics,
Virtual Reality, Simulation Engineering, Human Computer Interaction,
Perception-based Computer Graphics",Edu="- Ph.D in Computer Science,
Department of Computer Science, University of Bristol, 2001 '\n'- Master of
Science (M.Sc.) in Advanced Computing (Global Computing and Multimedia),
University of Bristol, UK, 1996 '\n'- Bachelor of Science (B.Sc.) in
Mathematics, Department of Mathematics, University of Crete, Greece,
1994",CV="Dr Katerina Mania completed a B.Sc. in Mathematics at the
University of Crete, Greece, an M.Sc. in Advanced Computing (Global Computing
and Multimedia) and a Ph.D in Computer Science at the Univeristy of Bristol,
UK, Department of Computer Science. Her Ph.D studies were fully funded by
Hewlett Packard Laboratories. Prior to her Ph.D, she was a full-time
researcher and consultant at Hewlett Packard Laboratories in Bristol, UK
working on architectural visualisation, 3D user interfaces and 3D graphics
for the web, from 1996 till 1998. After completion of her Ph.D in 2001 she
```

```
was appointed as an Assistant Professor in the Department of Informatics, University of Sussex, UK, achieving tenure in 2004",ResearchAreas="Perception-based Computer Graphics, Simulation Engineering, Human factors/HCI and Virtual Environments, Human factors Engineering, Fidelity Metrics for Computer Graphics Simulations, 3D Spatial cognition",Courses="Introduction to Computer Programming, Computer Graphics",Phone="+30 28210 37222",Email="k.mania@ced.tuc.gr",OfficeID="145.A14",Location=(X=63.6,Y=-2267.29,Z=279.4))
Professors=(ID="AggelosBletsas",PName="AggelosBletsas",Department="Telecommunications",OfficeID="141.A28",Edu="Edu1,Edu2,Edu3",Bio="BioBioBio",Location=(X=304.0,Y=-2960.0,Z=-66.0))

Classrooms=(ID="137P39",NumOfSeats="90",Location=(X=-504.0,Y=-5.0,Z=-190.0))

Labs=(ID="E01",LName="Electronics",OfficeID="A158",Stuff="Giwrgos A, Mitsos B",Location=(X=100.0,Y=210.0,Z=-250.0)))
```

The `Professors`, `Classrooms` and The `Labs` variables are arrays of structs declared in the `TUC_InfoData` class and are populated with the data found in the configuration file when the the class is instanciated. Then the arrays can be used in various classes that have need of this data.

TUC_InfoData Class

It was already seen in the previous paragraph that the configuration data for the application is stored in an external configuration file with .ini extension. In this paragraph, the class that is associated with that file, in particular with the section `[TUCProject.TUC_InfoData]` of the configuration file, is described. The class is declared in the *TUC_InfoData.uc* file as follows:

```
class TUC_InfoData extends Object
      config(TUC);
```

It extends from the Object class as specialized functionalities are not needed here. In order to use the configuration file, the config modifier is added in the declaration. When the class is instanciated in the program, the variables that are prefixed with the config modifier will be populated with data from the configuration file.

The data that defines a professor is constructed using a structure. The struct is called (obviously) Professor and it has fields for the name of the professor, the department he belongs to etc. Using this struct, an array of Professor is declared. This array stores all the data for the professors and it is populated on class instanciation. The array has the keyword "config" added after the "var" keyword to

indicate that the data for this array is found in a config file. These data structures are declared as follows:

```
struct Professor
{
	var string ID;
	var string PName;
	var string Position;
	var string Department;
	var string Sector;
	var string Speciality;
	var string Edu;
	var string CV;
	var string ResearchAreas;
	var string Courses;
	var string Phone;
	var string Email;
	var string OfficeID;
	var Vector Location;
};
var config array<Professor> Professors; //array loaded with Professors from
the config file
```

The data that defines a Lab is constructed using a struct. This struct holdes data for the lab id, the staff etc. An array of Lab is then declared. This array stores all the data for the professors and it is populated on class instanciation. These variables are declared as follows:

```
struct Lab
{
	var string ID;
	var string LName;
	var string Staff;
	var Vector Location;
};
var config array<Lab> Labs;    // array loaded with Labs from the config file
```

In a similar manner, classrooms are defined as follows:

```
struct Classroom
{
	var string ID;
	var string NumOfSeats ;
	var Vector Location;
};
var config array<Classroom> Classrooms; //array loaded with Classrooms from
the config file
```

When the user wants to make a search for a specific professor or classroom he/she can use one of the Find functions that are implemented and shown below. These three functions, one for each of the entities, accept a string which is used in the function body to compare with the id of the stored data. The search is done using the Find() function. This function allows to search for values in an array. It returnes the index of the first element containing the specified value.

```
/**
 * FindProfessor
 * @return Professor - the professor with the ProfessorID ID.
 */
function Professor FindProfessor(string ProfessorID)
{
    return Professors[Professors.Find('ID', ProfessorID)];
}

/**
 * FindLab
 * @return Lab - the lab that has id equal to LabID string
 */
function Lab FindLab(string LabID)
{
      return Labs[Labs.Find('ID', LabID)];
}

/**
 * FindClassroom
 * @return Classroom - the lab that has id equal to ClassroomID string
 */
function Classroom FindClassroom(string ClassroomID)
{
      return Classrooms[Classrooms.Find('ID', ClassroomID)];
}
```

5.3 Detecting Focused Actors

Some of the in-game interfaces are displayed on the screen when the user's look is focused on certain actors in the virtual environment. For example, when the user is looking at a lift button, a panel with instructions for calling the lift is shown on the screen. Another example is when the user focuses on a professor's office door; in such case an interface is displayed with some basic info (name and picture) for the

specific professor and a text message telling him/her to press the 'Q' button if they want to see more information.

The below Figures ilustrate these pop-up interfaces.

The logic behind focused actor detection is implemented in the `GetFocusedActor` function of the `TUC_FocusedActor` class:

```
class TUC_FocusedActor extends Actor;
```

The built-in function `Trace` does all the heavy lifting. It casts a ray into the world and returns what it collides with first. Trace takes into account both this actor's collision properties and the collision properties of the objects `Trace` may hit. If Trace does not hit anything it returns `NONE`; if Trace hits level geometry (BSP) it returns the current `LevelInfo`; otherwise `Trace` returns a reference to the `Actor` it hits.

If `Trace` hits something, the two local variables `HitLocation` and `HitNormal` declared at the beginning of the function will be filled in with the location of the hit and the normal of the surface.

The `Trace` function takes the location of the beginning and the ending of the line that we want to trace. The two vectors that are declared to hold these locations are `StartTrace` and `EndTrace`. The user's location is used as starting location. In order to calculate the trace from the user's eyes, the z (height) variable of the starting location is filled with the pawn's `EyeHeight` variable. The ending position is defined to be the starting positon of the user, plus the rotation of its head multiplied by some coefficient.

```
var PlayerController PlayerOwner;
var int SearchDistance;

simulated function PostBeginPlay()
{
```

```
        PlayerOwner = GetALocalPlayerController();
}


function Actor GetFocusedActor()
{
    local vector HitLocation, HitNormal, StartTrace, EndTrace;
    local Actor HitActor;

    StartTrace = PlayerOwner.Pawn.Location;
    StartTrace.Z += PlayerOwner.Pawn.EyeHeight;
    EndTrace = StartTrace+SearchDistance*vector(PlayerOwner.Pawn.Rotation);

    HitActor = PlayerOwner.Pawn.Trace(HitLocation, HitNormal, EndTrace,
                                      StartTrace, true, vect(0,0,0),,
                                      TRACEFLAG_Bullet);

    return HitActor;
}
```

The `GetFocusedActor` is called every tick (frame) from the `HudWrapper` class.

5.4 Lifts

In order to resemble the real building, lifts are implemented in Unrealscript and then placed in the virtual environment.

The Lifts consist of the following five classes:

- `TUC_LiftPlatform`
- `TUC_LiftControlPanel`
- `TUC_LiftButton`
- `TUC_FrontEnd_LiftControlPanel`
- `TUC_LiftDoor`

These classes can be used by level designers for fast and easy lift creation. One of the created lifts in the virtual environment is shown in the figure below:

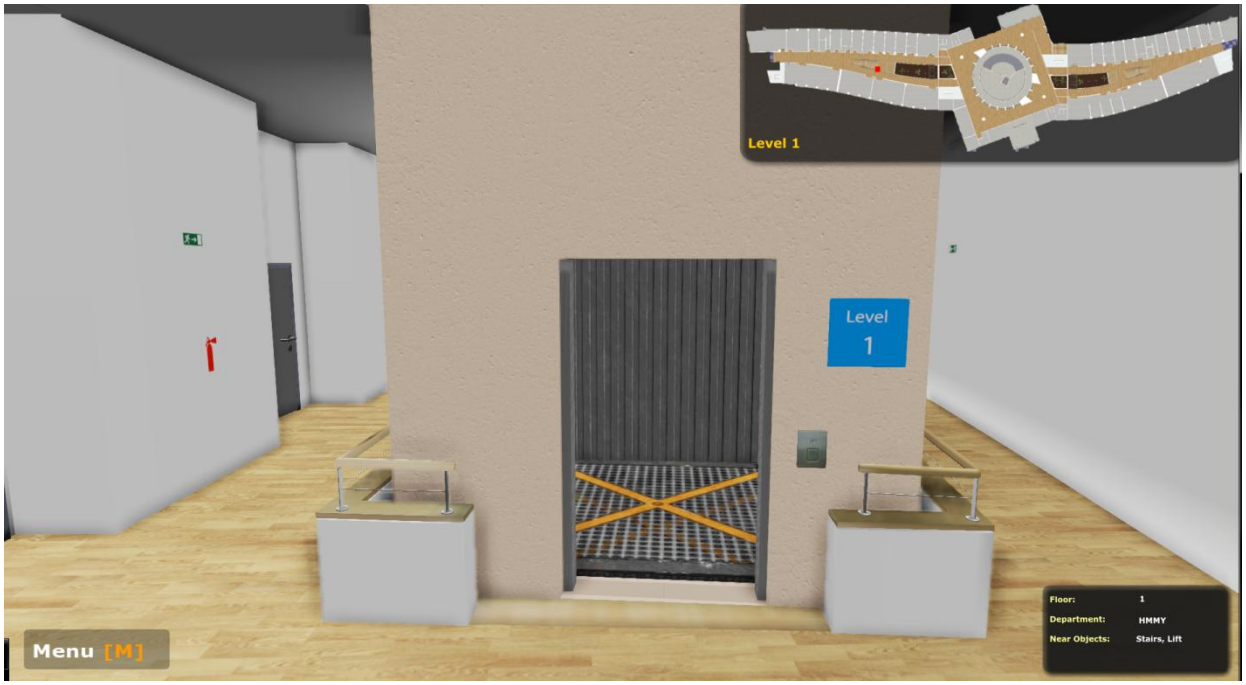


Figure 5.1 TUC Lift

The first two classes, `TUC_LiftPlatform` and `TUC_LiftDoor`, represent the platform that the user steps on and goes up and down, and, the doors of the lift.
The `TUC_LiftPlatform` is the more important class. All the other classes are associated with this class and instantiated in it. They are placeable, which means that they can be placed in the virtual scene.

The next two classes, `TUC_LiftButton` and `TUC_LiftControlPanel`, are used for the interaction between the user and the lift. These classes extend the trigger class and they are not the actual user interfaces. These are simply Trigger actors which listen when the player is entering the proximity of the lift. The first one, `TUC_LiftButton`, represents the outside/external button used for calling the lift. The second one, `TUC_LiftControlPanel`, is used inside the lift as a controller panel.

The last class, `TUC_FrontEnd_LiftControlPanel`, is the actual user interface for the control panel which is located inside the lift and it is used for controlling it.

Next, the above classes are described in details.

TUC_LiftButton

The external buttons used for calling the lift are implemented using this class. `TUC_LiftButton` extends the `Trigger` class, which means that it listens when the

user enters the proximity area of the lift button, in which the user can call the lift. The class declaration is as follows.

```
class TUC_LiftButton extends Trigger
      ClassGroup(TUC);
```

The next figure shows the placement of a trigger and its interaction area in which the user has to be located in order to call the lift:

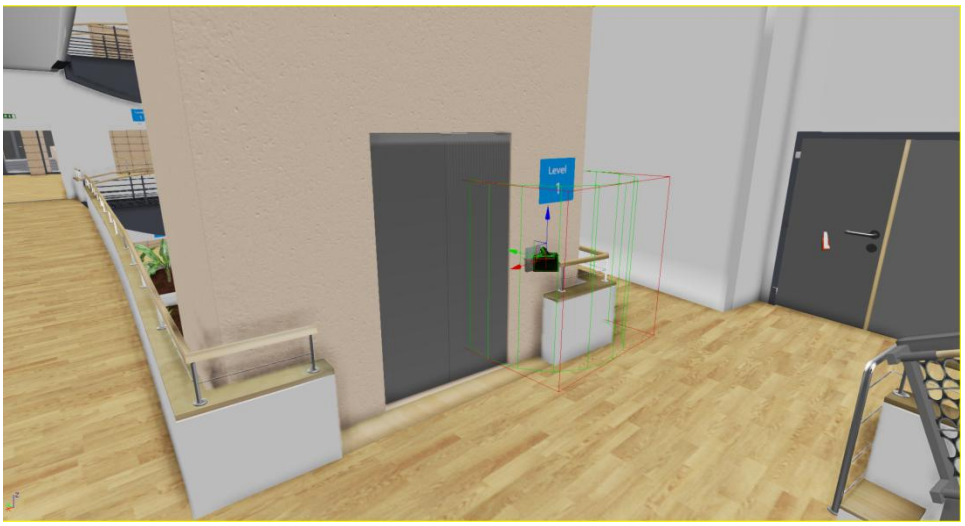


Figure 5.2. An example of external Lift Button. The user has to be located in green cylinder in order to interact with the lift

When the user enters the trigger's area (in other words he/she is in the interaction range), an information string "Call Lift [E]" is displayed on the HUD. Inside this area, the user can call the lift by pressing the 'E' button.

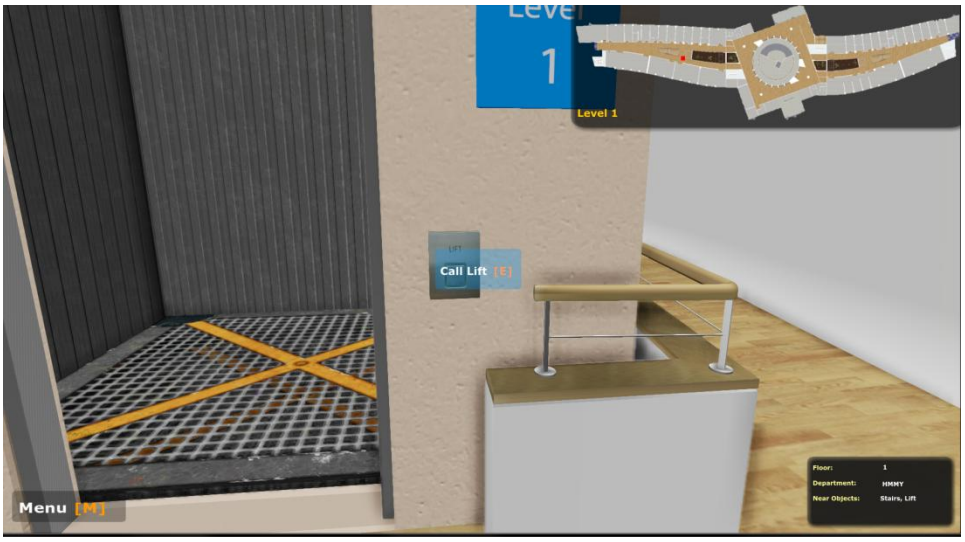


Figure 5.3. When the user looks at the lift button informative box is displayed

A lift button is associated with one `LiftPlatform`. Its more important variables are `LiftID` and `FloorID`. The first one holds information for the lift that is associated with this button. For example, in this application the lifts are given names (in the editor) such as: Lift1, Lift2 etc. The second one is holding information about the floor on which this button is placed. This variable also takes values in the Unreal Editor. The actual reference to the associated lift platform is held in the variable `LiftPlatform`. The last instance variable is `IsInInteractionRange`. This Boolean is true when the user enters the triggers' radius and further on it is checked in `UsedBy` function.

```
var TUC_LiftPlatform LiftPlatform;// LiftPlatform associated with this button
var bool IsInInteractionRange;//true when user has entered in the triggers'
var() string LiftID;    //lift assosiated with this button
var() int FloorID;      // the floor on wich is placed this button (
```

A reference to the lift platform associated with this button is found when the `PostBeginPlay` event is fired after all other actor has been initialized.

```
simulated event PostBeginPlay()
{
      LiftPlatform = FindLift();     // reference to the platform
}
```

Next, The `trigger`-related events and functions are overridden. When the user enters the `Triggers'` radius, the `Touch` event is fired. Here, not much effort is envolved, to be more precise, the parent implementation of the event is called and

IsInInteractionRange is set to true. Similar actions are taken in the Untouched event.

```
event Touch(Actor Other, PrimitiveComponent OtherComp, Vector HitLocation,
Vector HitNormal)
{
    super.Touch(Other, OtherComp, HitLocation, HitNormal);
    if( Pawn(Other) != none )
            IsInInteractionRange = true;
}

event Untouch(Actor Other)
{
    super.Untouch(Other);
    if( Pawn(Other) != none )
            IsInInteractionRange = false;
}
```

When the user finds himself/herself in the trigger's area and they press the E button, the UsedBy function is fired. This function checks if the user is within the interaction range and if so, it first plays a sound which announces that the button is pressed and then tells the lift platform to execute its MovePlatform function passing to it the FloorID variable(the floor on which the button is placed).

```
function bool UsedBy(Pawn User)
{
      local bool used;

      used = super.UsedBy(User);

      if( IsInInteractionRange )
      {
            PlaySound(SoundCue'a_interface.menu.UT3MenuCheckboxSelectCue');
            LiftPlatform.MovePlatform( FloorID );
            return true;
      }
      return used;
}
```

The function that performs the finding of the lift platform is the FinfLift. It simply iterates through all the actors that are instances of the TUC_LiftPlatform class, and when the correct platform that is associated with this button is found, it stops the searching and returns the platform to the caller.

```
function TUC_LiftPlatform FindLift()
{
      local TUC_LiftPlatform p;
```

```
        foreach AllActors( class 'TUC_LiftPlatform', p )
        {
                if( p.LiftID == LiftID )
                {
                        break;    // found
                }
        }
        return p;
}
```

TUC_LiftControlPanel

This class is not the actual user interface for controling the lift, but a trigger that is listening when the user is in interaction range with the control panel. It simply allows the user to interact with the control panel by pressing the 'E' button. An instance of this actor is attached to one moving platform in the world because one lift controller is allways associated with one lift.

```
class TUC_LiftControlPanel extends Trigger
        ClassGroup(TUC);
```

A lift control panel is associated with one lift platform and the reference to this platform is held in the `TUC_LiftPlatform` variable called `LiftPlatform`. The variable `LiftID` holds information for the lift platform that is associated with control panel. A reference to the flash movie that is played when the user presses the 'E' button is held in the `LiftControlPanelMovie` variable. The last instance variable is `IsInInteractionRange`. This `Boolean` is set to true when the user enters the triggers' radius and it is checked in `UsedBy` function.

```
var TUC_LiftPlatform LiftPlatform;    //reference to the assosiated platform
var TUC_FrontEnd_LiftControlPanel LiftControlPanelMovie; //the flash ui movie
var bool IsInInteractionRange;
var() string LiftID;        // lift assosiated with this door
```

A reference to the lift platform associated with this control panel is found when the `PostBeginPlay` event is fired after all other actor has been initialized.

```
simulated event PostBeginPlay()
{
        LiftPlatform = FindLift();    // take reference to the platform
}
```

When the user enters the `Triggers'` radius the `Touch` event is fired. The parent implementation of the event is called and `IsInInteractionRange` is set to true. Similar actions are taken in the Untouched event.

```
event Touch(Actor Other, PrimitiveComponent OtherComp, Vector HitLocation,
Vector HitNormal)
{
    super.Touch(Other, OtherComp, HitLocation, HitNormal);
    if( Pawn(Other) != none )
        IsInInteractionRange = true;
}

event Untouch(Actor Other)
{
    super.Untouch(Other);
    if( Pawn(Other) != none )
        IsInInteractionRange = false;
}
```

When the user is in the triggers' area and presses the 'E' button, the `UsedBy` function is fired. This function checks the `IsInteractionRange` variable, and, when true, it first plays a sound which announces that the button is pressed and then calls the `OpenMovie` function which instantiates the movie class for the control panel.

```
function bool UsedBy(Pawn User)
{
    local bool used;
    used = super.UsedBy(User);

    if( IsInInteractionRange )
    {
        PlaySound(SoundCue'a_interface.menu.UT3MenuCheckboxSelectCue');
        OpenMovie();    // Show the control panel to the user
        return true;
    }
    return used;
}
```

When the user chooses destination floor from the control panel, which is instance of `TUC_FrontEnd_LiftControlPanel`, the CalculateLiftMovement function is called. This function closes the lift control panel and calls the `LiftPlatform`'s `MovePlatform` function passing which button was pressed. `MovePlatform` function based on the pressed button and on which floor is the platform calculates the lift movement.

```
function CalculateLiftMovement(int PressedFloorButton)
{
	CloseMovie();	// first off, close the panel then do the calculations

	LiftPlatform.MovePlatform( PressedFloorButton );
}
```

The function that performs the finding of the lift platform is the `FinfLift`. It simply iterates through all the actors that are instances of the `TUC_LiftPlatform` class and, when the correct platform that is associated with this button is found, it stops the searching and returns the platform to the caller.

```
function TUC_LiftPlatform FindLift()
{
	local TUC_LiftPlatform p;

	foreach AllActors( class 'TUC_LiftPlatform', p )
	{
		if( p.LiftID == LiftID )
			return p;			// we found it
	}
}
```

After pressing the 'E' button, the program calls the `OpenMovie` function. A new movie that represents what the user interacts with is created and then initialized by calling its `InitMovie` function.

```
function OpenMovie()
{
    LiftControlPanelMovie = new class'TUC_FrontEnd_LiftControlPanel';
    LiftControlPanelMovie.InitMovie(self);
}
```

When the user selects the destination floor, the first thing to do is to close the associated flash movie which represents the control panel that the user interacts with.

```
function CloseMovie()
{
	LiftControlPanelMovie.Close();
	LiftControlPanelMovie = none;
}
```

TUC_FrontEnd_LiftControlPanel

The interface that is displayed on the `HUD`, and with which the user can actually interact in order to controll the lift, is implemented in this class. This class is a wrapper for the scaleform flash file that is created for the user interface and represents a panel with three buttons on it, one for each of the buildings' floors. This panel is displayed on the `HUD` when the user presses the 'E' button while it is located in the associated `Triggers'` area.

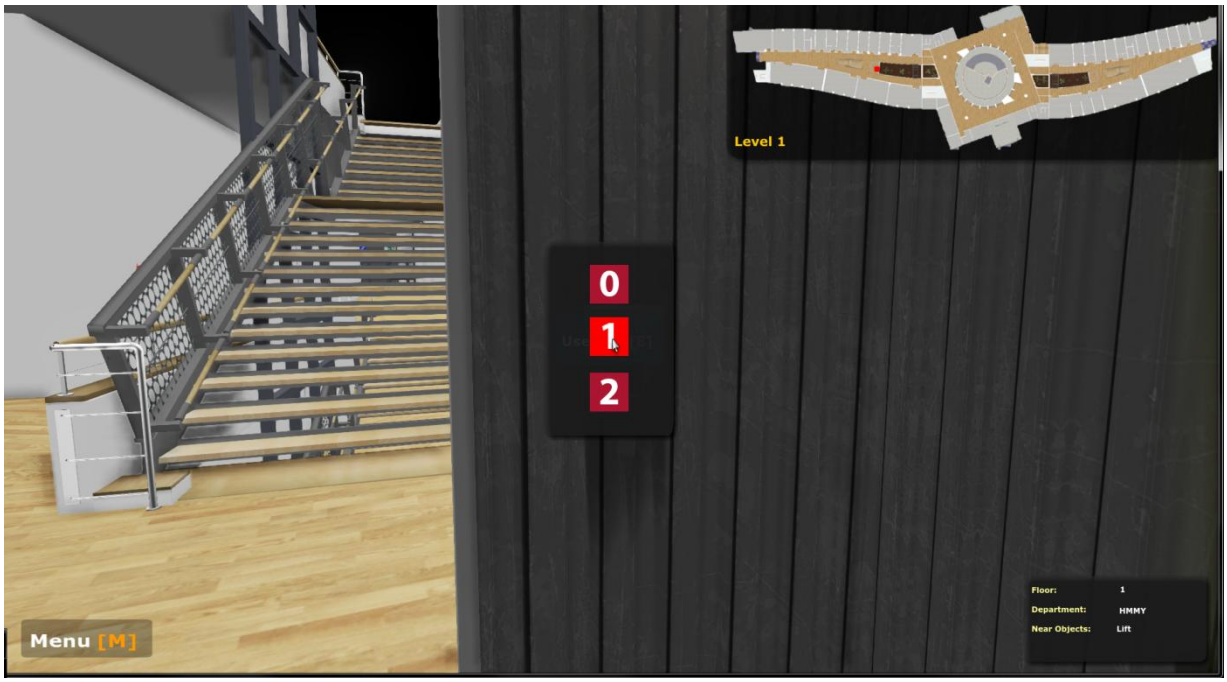


Figure 5.4 Lift Control Panel

The same as the other classes used for displaying flash content, this class also extends from the `GFxMoviePlayer` as it can be seen bellow:

```
class TUC_FrontEnd_LiftControlPanel extends GFxMoviePlayer;
```

References to the button components are stored in the `GFxClikWidget` variables: `BtnFloor0`, `BtnFloor1`, `BtnFloor2` . A variable of integer type is used to hold information which button is pressed. The last member variable is `LiftControlPanel` and it is used as a pointer to the associated `trigger`.

```
var GFxClikWidget BtnFloor0, BtnFloor1, BtnFloor2;// buttons
var int PressedFloorButton;                       //the selected floor button
var TUC_LiftControlPanel LiftControlPanel;        // the associated trigger
```

The movie is initialized in the `InitMovie` function. `InitMovie` takes reference to the `trigger` that is associated with this class, playes the movie and tells the program not to scale the control panel when it is rendered.

```
function InitMovie(TUC_LiftControlPanel Panel)
{
	LiftControlPanel = Panel;		// reference to the trigger
	Start();				//
	Advance(0);
	SetViewScaleMode(SM_NoScale);
}
```

The attaching event listener function to the buttons is done in the WidgetInitialized evet. This event is fired when the CLIK components in the associated Flash file call this function after they are done with all the ActionScript initialization. The name of the button is checked in a switch statement.

```
event bool WidgetInitialized(name WidgetName, name WidgetPath, GFxObject
Widget)
{
	local bool bWasHandled;

	bWasHandled = false;

	switch(WidgetName)
	{
		case ('btn_Floor0'):
			BtnFloor0 = GFxClikWidget(Widget);
			BtnFloor0.AddEventListener('CLIK_click', InputFloor0);
			bWasHandled = true;
			break;
		case ('btn_Floor1'):
			BtnFloor1 = GFxClikWidget(Widget);
			BtnFloor1.AddEventListener('CLIK_click', InputFloor1);
			bWasHandled = true;
			break;
		case ('btn_Floor2'):
			BtnFloor2 = GFxClikWidget(Widget);
			BtnFloor2.AddEventListener('CLIK_click', InputFloor2);
			bWasHandled = true;
			break;
		default:
			break;
	}
	return bWasHandled;
}
```

When one of the buttons is pressed, its corresponding attached event listener function is fired. The function sets the destination floor by assigning the correct value to the `PressedFloorButton` variable. It then calls the function `CalculateLiftMovement` passing it the variable for the destination floor. `CalculateLiftMovement` function is found in the control panel and based on the `PressedFloorButton` variable, it calculates how many floors the platform has to be displaced.

```
function InputFloor0(EventData data)
{
	PressedFloorButton = 0;
	LiftControlPanel.CalculateLiftMovement(PressedFloorButton);
}
```

The event listeners for the other buttons are similar to above function and therefore , they are ommited here.

The binding of the flash file as well as the widget binding with this class is done in the default properties section.

```
DefaultProperties
{
	MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_LiftControlPanel'
	WidgetBindings.Add((WidgetName="btn_Floor0",WidgetClass=class'GFxCLIKWi
	                                                              dget'))
	WidgetBindings.Add((WidgetName="btn_Floor1",WidgetClass=class'GFxCLIKWi
	                                                              dget'))
	WidgetBindings.Add((WidgetName="btn_Floor2",WidgetClass=class'GFxCLIKWi
	                                                              dget'))

	TimingMode=TM_Real
	bPauseGameWhileActive=true
	bCaptureInput=true
}
```

Lift Doors

The lift doors and their open-closing functionality is implemented in this class. This class is extension of the `Actor` class, it is placeable and can be found in the content's browser 'Actor Classes' options, under the TUC group.

```
class TUC_LiftDoor extends Actor
	placeable
	ClassGroup(TUC);
```

In order to use the doors, an instance of this class actor it first has to be added to the virtual environment. One lift door is associated with one lift platform. This is set in the actor properties by typing the name of the lift platform, for example Lift1. Other information that is added in the editor and that is needed for correct using of the lift doors is, the floor on which the door is placed, the static meshes that appart the door, the light environment used for correct lightning of the moving doors and the sounds played when opening or closing. The variables used for holding such information are declared using parenthesis and are shown below:

```
var() int FloorID;              // the floor on wich is placed this door
var() string LiftID;            // lift assosiated with this door
var() StaticMeshComponent LeftDoor, RightDoor;     // reference to the left
var() const editconst DynamicLightEnvironmentComponent LightEnvironment;
var() SoundCue OpenDoor, CloseDoor;                      // open/close sound
```

In order to see this Actor in the world, two StaticMeshComponent representing the right and the left door are added in the default properties of the class. The StaticMeshComponents consists of static mesh and a LightEnvironment. In order to form a two-part sliding door, one of the static meshes is translated in its x axis by 36 unreal units.

```
begin object class=StaticMeshComponent name=LeftDoorMesh
    StaticMesh=StaticMesh'TUC_BasicVolumes.SM.SM_Box_01'
    LightEnvironment=MyLightEnvironment
end object
LeftDoor=LeftDoorMesh
Components.Add(LeftDoorMesh)

begin object class=StaticMeshComponent name=RightDoorMesh
    StaticMesh=StaticMesh'TUC_BasicVolumes.SM.SM_Box_01'
    Translation=(X=36.0,Y=0.0,Z=0.0)
    LightEnvironment=MyLightEnvironment
end object
RightDoor=RightDoorMesh
Components.Add(RightDoorMesh)

// dynamic light environment component
begin object class=DynamicLightEnvironmentComponent
      Name=MyLightEnvironment
      bEnabled=TRUE
end object
LightEnvironment=MyLightEnvironment
Components.Add(MyLightEnvironment)
```

The other instance variables which are explained in the comments are shown below:

```
var bool bIsOpen;              // the door is open or closed
var float MovementTime;        // the time needed the doors to full open/close
var() int FullOpen;     // number of unreal units needed for full door opening
var float SleepTime;                //
var float RightTranslation, LeftTranslation;  RightTranslation=0 and
LeftTranslation=36
var float DeltaDistance;        // uu/sec
var vector L, R;               // left and right translation vectors
var int i;                     // counter used in the state and right door
```

The opening and closing of the doors is performed when the Door actor enters in its `Moving` state. This state uses the ignores keyword to ignore callings to the `Toggle` function which practically means that when the doors start to open or close they will not be interrupted by(od strana na korisnikot, taka?) the user. The actual door movement is performed in a loop which executes `FullOpen` times. The `FullOpen` variable holds the distance (in unreal units) between the start(closed) end the end(opened) point of the door's movement. In every loop execution, the left door is translated in one direction and the right door in(?) another direction. The translating is done by adding or subtracting one unit (`DeltaDistance`) depending on whether it is performed opening or closing doors. Due to the need for the opening or closing to take some time, the latent function `Sleep` is called on every cycle. `Sleep` pauses the state execution for a certain amount of time, and then continues. This function call does not return immediately, but it returns after a certain amount of game time elapses. When the doors' opening/closing terminates, the program exits this state and goes to the idle state.

```
state Moving
{
   ignores Toggle;
Begin:

    PlaySound(SoundCue'TUC_Sounds.Sounds.S_LiftOpenDoor');

      SleepTime = MovementTime/FullOpen;

      /* Opening/Closing the Doors */
      for( i=0; i<FullOpen; i++)
      {
            R.X = LeftTranslation += DeltaDistance;
            RightDoor.SetTranslation(R);
```

```
            L.X = RightTranslation += -DeltaDistance;
            LeftDoor.SetTranslation(L);

            Sleep((SleepTime));
      }

   GotoState('Idle');
}
```

The logic for toggling the door is imlemented in the `Toggle` function. This function checks the boolean `bIsOpen` variable to see if the door is opened or closed. If open, (`bIsOpen=true`) it inverses the `DeltaDistance` variable and continues to the moving state. The `DeltaDistance` variable is an integer and represents one unreal unit. When positive, it is used for opening the doors and when negative, for closing them.

```
function Toggle()
{
      if( !bIsOpen )
            DeltaDistance = default.DeltaDistance;
      else
            DeltaDistance = -default.DeltaDistance;

      bIsOpen = !bIsOpen;

      GotoState('Moving');
}
```

TUC_LiftPlatform

The platform that the user steps on and goes up and down is implemented in the `LiftPlatform`. While all the classes that are described above are used in the lift creation, this class is somewhat more important due to its direct association with all the other classes. As it is expected, it extends from the Actor class and it can be placed in the virtual environment.

```
class TUC_LiftPlatform extends Actor
      placeable
      ClassGroup(TUC);
```

Every lift platform has: liftID - the name which is given in the Unreal Editor, a static mesh component - which is used for holding the specified static mesh from the library, a sound component - for playing sound when it moves and a light environment.

```
var() string LiftID;
var() StaticMeshComponent LiftPlatform;    // the static mesh used for the
var() AudioComponent LiftMovementSound; // holds references to the lift
var const editconst DynamicLightEnvironmentComponent LightEnvironment;
```

The other lift platform variables are given below:

```
var TUC_LiftDoor Doors[3];                 // references to the lift doors
var int Floor;                             // current floor
var int NextFloor;                         // next floor
var() float DeltaDistance;                 // uu/sec
var() float MovementTime;                  // the time needed for the
platform to move from one floor to another
var float SleepTime;
var vector DeltaDistanceVector;            // [0, 0, DeltaDistance]
var int FloorsDisplacement;                // num of floors we should move
var int FloorHeight;                       // floor height
var int i;                            // counter used in the 'Moving' state

enum LiftDirection
{
      Stationary,       // not used
      Up,
      Down
};
var LiftDirection Direction;
```

When this class is instantiated, the `PostBeginPlay` function is fired. The `PostBeginPlay` function is often used as a `constructor`. In this case, a `timer` is set except the parent implementation. The `timer` waits for two seconds and calls the `InitDoors` function which is used for finding the associated doors and opening them. There is a waiting time in order to get the doors opened, after this class is first initialized by the engine. The opened doors are those found on the floor 0, the same as the lift platform. This is done so that the doors at the floor on which the lift platform is, are always opened.

```
simulated event PostBeginPlay()
{
      super.PostBeginPlay();
      SetTimer(2, false, 'InitDoors');
}

function InitDoors()
{
      FindLiftDoors();      // load the assosiated doors
```

```
	Doors[0].Toggle();	// open the doors on the first floor
}
```

When the lift platform has to find the associated doors, it uses the `FindLiftDoors` function. This function itterates trough all the lift doors that are placed in the virtual scene and when it finds that the `LiftID` of the door equals to itself, it captures a reference in the Doors array.

```
function FindLiftDoors()
{
	local TUC_LiftDoor D;

	foreach AllActors( class'TUC_LiftDoor', D )
	{
		if( D.LiftID == LiftID )
		{
			Doors[D.FloorID] = D;	// capture reference
		}
	}
}
```

When the user presses the buttons in the lift or calls the lift, the program flow is transfered eventually to the `MovePlatform` function. `MovePlatform` function accepts a integer parameter which represents the pressed lift button (basically the floor) or the chosen floor, and based on the variable value, it calculates how many floors does the lift platform has to move up or down, or stay where it is. This calculation is done by taking the diference of the passed variable and its current floor. The difference is stored in the `Move` variable. If the `Move` is 0, the function returns and no lift movement is performed. If the value is negative (which means that the lift has to move down), the `Direction` variable is set to `Down`, and if positive, it is set to `Up`. Before going to the `Moving` state, which is done by the platform movement, this function takes the absolute value of the Move variable and assigns it to the `FloorDisplacement` variable. This variable is used in the movement loop as an upper bound of the loop.

```
function MovePlatform( int CallingFloor )
{
	local int Move;

	NextFloor = CallingFloor;
	Move = CallingFloor - Floor;	// the diference gives how many floors
the lift has to move up or down.

	if( Move == 0 )
	    return;
```

```
	else if( Move < 0 )
		Direction = Down;
	else
		Direction = Up;

	FloorsDisplacement = abs(Move);
	GotoState('Moving');
}
```

The movement of the lift platform is implemented in the state named `Moving`. Upon entering the state code, the doors are closed and the state execution is paused for 2.1 seconds, in order the doors to close properly. Next, depending on the `Direction` variable, the `DeltaDistance` variable is set with the default value or the inverse value. The `DeltaDistance` variable is used for translating the platform. Its default value is 1, and represents one unreal unit. Next, the movement sound starts to play. The loop that does the platform movement is next defined, it is executed `FloorsDisplacement * FloorHeight` times. The floors in the virtual environment are 220 unreal units high and it was already seen that the `FloorsDisplacement` variable takes values such as 1, 2, 3 etc. At every loop execution, the platform is moved one unreal unit up or down referring to the `DeltaDistanceVector` vector, which is passed to the `Move` function.
The move function moves the actor in the 3d space according to its vector parameter. The vector's x and y variables are set to 0 and the z variable is set to `DeltaDistance` (-1 or 1). When the loop terminates the execution, which means that the lift has stopped moving, the sound is also stopped, the current floor of the lift platform is updated with the nextfloor variable, the doors are opened and finally, the state is turned to Idle.

```
state Moving
{
    ignores MovePlatform, FindLiftDoors;

Begin:
	Doors[Floor].Toggle();  // first close the lift doors
	Sleep(2.1);

	if( Direction == Up )                              // go up
		DeltaDistance = default.DeltaDistance;
	else                                               // go down
		DeltaDistance = -default.DeltaDistance;

	LiftMovementSound.Play();    // plays the movement sound

	DeltaDistanceVector.z = DeltaDistance;
	SleepTime = MovementTime/FloorHeight;
```

```
        /* Move the platform */
        for( i=0; i<(FloorsDisplacement*FloorHeight); i++)
        {
              Move(DeltaDistanceVector);
              Sleep(SleepTime);
        }

        LiftMovementSound.Stop();
        Floor = NextFloor;

        Doors[Floor].Toggle();     // open the doors

        GotoState('Idle');
}
```

The `LightEnvironment`, the `StaticMeshComponent`, the `AudioComponent` and some of the other variables are set up in the `defaultProperties` section. As For the static mesh, the `SM_Box_01` mesh is loaded from the library and then scaled properly in order to fit in the lift shaft.

```
DefaultProperties
{
        //
    begin object class=DynamicLightEnvironmentComponent
      Name=MyLightEnvironment
      bEnabled=TRUE
    end object
    LightEnvironment=MyLightEnvironment

    Components.Add(MyLightEnvironment)

    begin object class=StaticMeshComponent name=Platform
        StaticMesh=StaticMesh'TUC_BasicVolumes.SM.SM_Box_01'
        Scale3D=(X=4.5, Y=20.0, Z=0.25)
        LightEnvironment=MyLightEnvironment
    end object
    LiftPlatform=Platform
    Components.Add(Platform)

    Begin Object Class=AudioComponent Name=LiftSound
        SoundCue=SoundCue'a_movers.Movers.Elevator01_LoopCue'
    End Object
    LiftMovementSound = LiftSound
    Components.Add( LiftSound )

    CollisionComponent=Platform
    bCollideActors=true
```

```
    bBlockActors=true

    Floor=0
    Direction=Stationary
    DeltaDistance=1
    FloorHeight=220
    MovementTime=3
}
```

An example of lift creation

The steps taken in creation of a lift involve: First, a lift platform is added in the world and it is given a name (say Lift1). Next, A control panel is attached to it. In it LiftID propery, the name of the platform is typed. The buttons for calling the lift are added next, one button for every floor. They are also associated with the lift platform and in their Floor ID property we have given int values representing the floor on which they are placed.

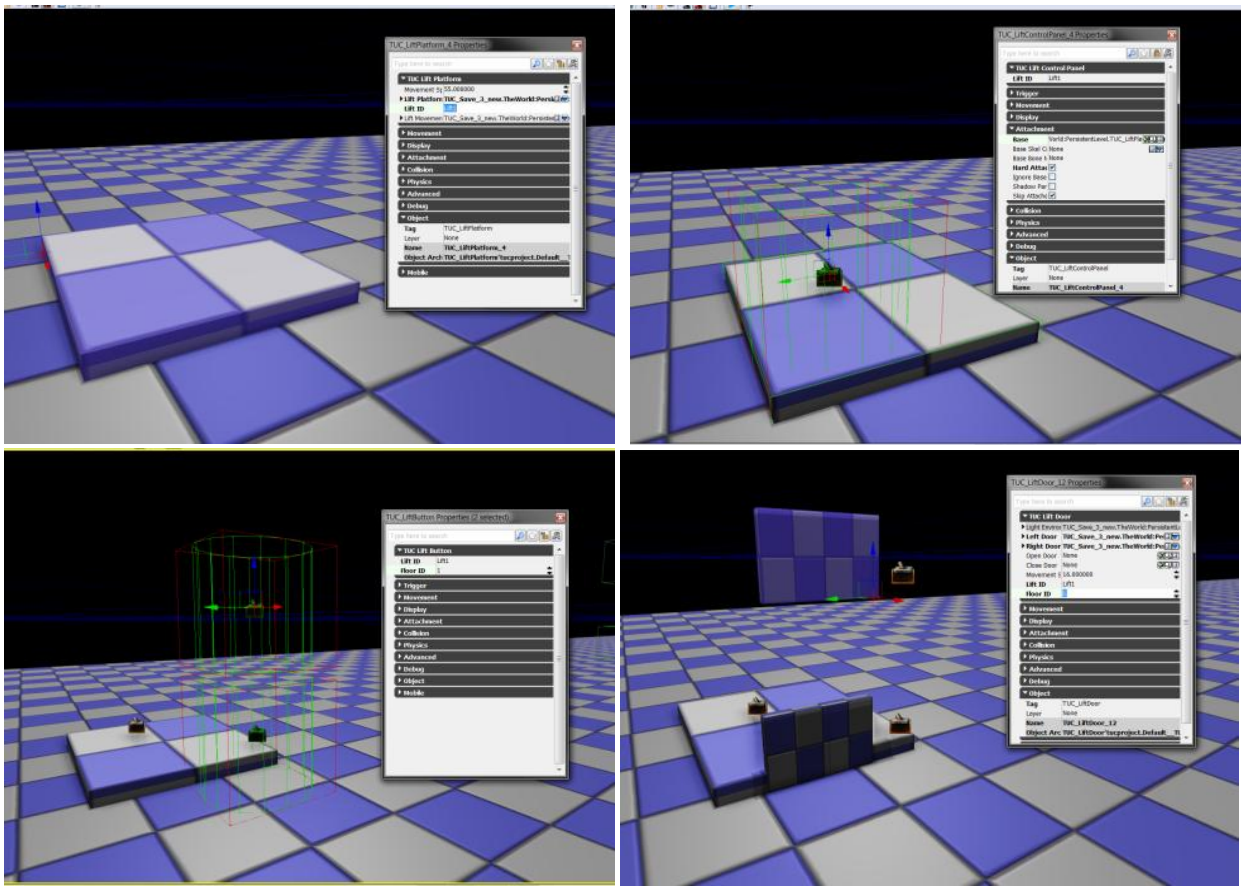


Figure 5.5 An example of lift creation

5.5 Menu System

The application's menu is accessed when the user presses the 'M' keybord key. The menu consists of black transparent background and buttons that are used to access some of the application's functionalities. The functionality of these buttons is briefly explained next.

Find Path: The Find Path functionality helps the user in finding a specific Professor office, lab, classroom and even department secretary. When this functionality is active, a transparent green arrow is created in front of the user which shows him/her the way to the destination.

- Toggle Mini Map: The "Toggle Mini Map" button toggles a miniature map of the building and it is placed in the right top corner of the screen.

- Toggle Extended Info: This button is used to show and hide a panel which is placed at the right bottom corner of the screen. The panel is populated with information from Unrealscript with information about the floor on which the user stands as well as about the department. Nearby objects are also dynamically added to the panel.

- Interactive Map: The Interactive Map button allows the user to see information for the staff, classrooms, labs, coffee shops, stairs, secretaries and lift, all of them placed on a 2D map.

- Cinematics: When the user press the "Cinematics" button, a menu page with a list of cinematics (film-style scenes) is presented.

- Settings: The "Settings" button opens a page from where the user can change some application parameters, such as, video resolution an anti-aliasing.

The interface is built in Adobe Flash and the logic behind it is implemented in Unrealscript.

A black transparent background is first set in Flash, and Scaleform CLIK buttons are added on top of it. All of the buttons are then given instance name in order to be accessible from Unrealscript. After placing them on the stage, the buttons are skinned as needed. After the menu system is completed, it is then imported in UDK and accessed from Unrealscript. The Unrealscript side of the menu system is analyzed in the following section.

The Unrealscript wrapper class for the swf file is called `TUC_FrontEnd_MainMenu` and extends from `GFxMoviePlayer`.

```
class TUC_FrontEnd_MainMenu extends GFxMoviePlayer;
```

References to the flash file buttons are declared at the begining of the class:

```
var GFxClikWidget btnShowPath, btnToggleMiniMap, btnToggleExtendedInfo,
BtnInteractiveMap, BtnSettings, BtnCinematics, BtnExitMainMenu, btnExitApp;
```

The same goes for the reference to the next menu page:

```
var GFxMoviePlayer NextMenuPage;
```

The sfw file is initialized and the movie started playing using the following function:

```
function Init(optional LocalPlayer player)
{
	Start();
	Advance(0.f);
	SetViewScaleMode(SM_ExactFit);
}
```

References to the swf's file buttons are taken in the next function. At this function, event listeners are also assigned to the buttons.

```
event bool WidgetInitialized(name WidgetName, name WidgetPath, GFxObject
Widget)
{
	local bool bWasHandled;

	bWasHandled = false;

	switch(WidgetName)
	{
		case ('btnShowPath'):
			btnShowPath = GFxClikWidget(Widget);
			btnShowPath.AddEventListener('CLIK_click', ShowPath);
			bWasHandled = true;
			break;
		case ('btnToggleMiniMap'):
			btnToggleMiniMap = GFxClikWidget(Widget);
			btnToggleMiniMap.AddEventListener('CLIK_click',
ToggleMiniMap);
			bWasHandled = true;
			break;
		case ('btnToggleExtendedInfo'):
			btnToggleExtendedInfo = GFxClikWidget(Widget);
			btnToggleExtendedInfo.AddEventListener('CLIK_click',
ToggleExtendedInfo);
			bWasHandled = true;
			break;
		case ('btnCinematics'):
			BtnCinematics = GFxClikWidget(Widget);
			BtnCinematics.AddEventListener('CLIK_click',
ShowCinematics);
			bWasHandled = true;
			break;
		case ('btnInteractiveMap'):
			BtnInteractiveMap = GFxClikWidget(Widget);
			BtnInteractiveMap.AddEventListener('CLIK_click',
ShowInteractiveMap);
			bWasHandled = true;
			break;
```

```
            case ('btnSettings'):
                  BtnSettings = GFxClikWidget(Widget);
                  BtnSettings.AddEventListener('CLIK_click', Settings);
                  bWasHandled = true;
                  break;
            case ('btnExitMainMenu'):
                  BtnExitMainMenu = GFxClikWidget(Widget);
                  BtnExitMainMenu.AddEventListener('CLIK_click',
ExitMainMenu);
                  bWasHandled = true;
                  break;
            case ('btnExitApp'):
                  btnExitApp = GFxClikWidget(Widget);
                  btnExitApp.AddEventListener('CLIK_click', ExitApp);
                  bWasHandled = true;
                  break;
            default:
                  break;
      }
      return bWasHandled;
}
```

When one of the `ShowPath`, `ShowCinematics`, `ShowInteractiveMap` or the `Settings` event listener is triggered, the function simply calls the `LoadNextMenuPage` function which takes care of loading a new movie.

The `LoadNextMenuPage` is declared as shown in the code chunk below. It first creates a new instance of the menu class, it initializes it and then closes itself - the current menu page. The false parameter in the `Close` function tells the program to not delete from the memory this page, because it might be accessed soon.

```
function LoadNextMenuPage(class<GFxMoviePlayer> NextMenuClass)
{
      NextMenuPage = new NextMenuClass;
      NextMenuPage.Init();
      Close(false);
}
```

The Mini Map and the Extended Info are components of the `HUD` - meaning that their showing and hiding is performed from functions defined in the `HUD` class. The toggling of the Mini Map and the Extended info is done in similar manner. After the `ToggleMiniMap` or the `ToggleExtendedInfo` function is triggered, the current movie is closed and the function responsible for the toggling is called. The last code line calls the `MenuMenager` function defined in the player controller.

```
*/
function ToggleMiniMap(EventData data)
{
	Close(false);
	TUC_FrontEnd_HudWrapper(TUCPlayerController(GetPC()).myHUD).HudMovie.To
ggleMiniMap();
	TUCPlayerController(GetPC()).MenuManager();
}

function ToggleExtendedInfo(EventData data)
{
	Close(false);
	TUC_FrontEnd_HudWrapper(TUCPlayerController(GetPC()).myHUD).HudMovie.To
ggleExtendedInfo();
	TUCPlayerController(GetPC()).MenuManager();
}
```

The user presses the red x button, the program closes the main menu and transfers the program flow to the Controller.

```
function ExitMainMenu(EventData data)
{
	Close(false);
	TUCPlayerController(GetPC()).MenuManager();	// PC will handle
}
```

The console command "exit" is used in order to terminate the application.

```
function ExitApp(EventData data)
{
	ConsoleCommand("exit");
}
```

The associated swf file, as well as the buttons found in it, are bounded to this class in the default properties as follows:

```
DefaultProperties
{
	MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_MainMenu'

	WidgetBindings.Add((WidgetName="btnShowPath",WidgetClass=class'GFxCLIKW
idget'))
	WidgetBindings.Add((WidgetName="btnToggleMiniMap",WidgetClass=class'GFx
CLIKWidget'))
	WidgetBindings.Add((WidgetName="btnToggleExtendedInfo",WidgetClass=clas
s'GFxCLIKWidget'))
```

```
        WidgetBindings.Add((WidgetName="btnInteractiveMap",WidgetClass=class'GF
xCLIKWidget'))
        WidgetBindings.Add((WidgetName="btnCinematics",WidgetClass=class'GFxCLI
KWidget'))
        WidgetBindings.Add((WidgetName="btnSettings",WidgetClass=class'GFxCLIKW
idget'))
        WidgetBindings.Add((WidgetName="btnExitMainMenu",WidgetClass=class'GFxC
LIKWidget'))
        WidgetBindings.Add((WidgetName="btnExitApp",WidgetClass=class'GFxCLIKWi
dget'))

        bShowHardwareMouseCursor=true

        TimingMode=TM_Real
        bPauseGameWhileActive=true
        bCaptureInput=true
        bDisplayWithHudOff=true
        bIgnoreMouseInput=false
}
```

The `bShowHardwareMouseCursor` command tells the `Movie` to use the system mouse cursor, the `TimingMode` tells the `Movie` to proceed at normal playback speed, disregarding slomo and pause, the `bPauseGameWhileActive` pauses the application when this movie is played and the other three commands tell the `movie` to capture keybord input, to hide the `HUD` and to take input from the mouse.

5.6 Heads-Up Display (HUD)

The HUD (heads-up display) is the method by which information is visually relayed to the player as part of a game's user interface. In video games, the HUD is frequently used to simultaneously display several pieces of information including the main character's health, items, and an indication of game progression - [wikipedia]. However, concerning this application, what is displayed on the HUD is a MiniMap (a map on which the player's location is shown), a panel showing navigation-related infromation, a panel that displays how to enter the main menu, and an informative label that is visible when the path finding functionality is activated. The HUD is non-interactive, meaning the user does not click elements of the HUD.

There are two basic methods that are used when creating HUDs in Unreal Development Kit. These are:

- Canvas - Utilizes the `Canvas` class to draw HUD graphics.

- Scaleform GFx (Flash) - Utilizes the Scaleform GFx integration to display HUD graphics and menus.

These two methods of HUDs are not mutually exclusive; it is possible to use a combination of both

This application utilizes the second method, the Scaleform GFx. The Scaleform GFx makes the use of interfaces built in Adobe Flash Professional to be able to be used as heads-up displays.

Figure 5.6 shows the HUD created for this application.

Figure 5.6. The HUD

The components that constitute the HUD are: Menu panel, Mini Map, a label that is set visible in the path finding and an ExtendedInfo panel.

The MiniMap and the ExtendedInfo components are independent objects that are created in separate files and classes from the HUD file and class, but they are instantiated and controlled by the HUD. These two components are thoroughly described later in this section.

Setting up the Scaleform/Flash scene

The HUD scene is first set up using Adobe Flash. It is fairly simple, it consists of a movie clip that shows the static text: "Menu [M]", and a label that is dynamically updated and displayed when the user uses the find path functionality. After setting up the scene, it is publlished as a swf flie and placed in the ..\UDK\UDKGame\Flash\TUCFrontEndProject folder. Then it is imported from the content browser in the application library. The resulting flash scene is shown in the Figure 5.7. Next, the unrealscript side of the HUD is examined.

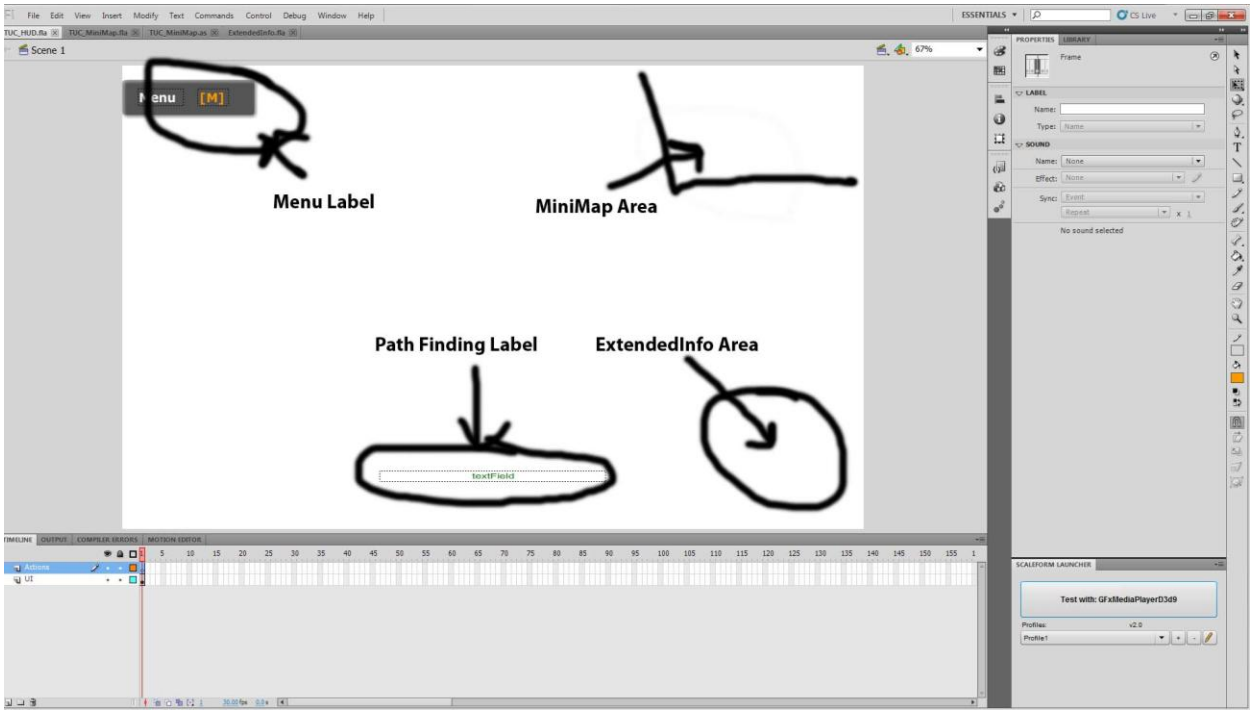


Figure 5.7. The HUD flash file

There are two unrealscript classes involved in the HUD creation. These are:

- `TUC_FrontEnd_HudWrapper`
- `TUC_FrontEnd_Hud`

The `TUC_FrontEnd_Hud` class is the Unrealscript class that represents the HUD. It is used as a wrapper for the flash file that was created before, and it is responsible for initializing and playing the movie file, as well as executing all Unrealscript

funtionality for communicationg with the swf file. It is an extension of the GFxMoviePlayer class.

```
class TUC_FrontEnd_Hud extends GFxMoviePlayer;
```

The other class, TUC_FrontEnd_HudWrapper, extends from the HUD class. The HUD class is the base class for displaying elements overlaid on the screen. Which HUD type is going to be used depends on the gametype being used. In the case presented, this is done by specifying the HUD class in the default properties of the TUCInfo class, as follows:

```
HUDType=class'TUCProject.TUC_FrontEnd_HudWrapper'
```

These two classes are described next.

TUC_FrontEnd_Hud

The first thing that is done is to bind the swf movie that represents the HUD to this class. This is done in the class's DefaultProperties area by specifying the path to the sfw file. Along with the swf binding, the HUD related properties are set up. These properties are used for disabling mouse input, enabling keybord input, disabling pausing the application while the HUD is active etc. These commands are shown in the below code chunk.

```
DefaultProperties
{
	MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_HUD'

	bIgnoreMouseInput = true
	bDisplayWithHudOff = false
	bEnableGammaCorrection = false
	bPauseGameWhileActive = false
	bCaptureInput = false;
}
```

The Menu label that can be seen in the figure x.x is static and therefore it is not accessed in this class. However, the info label found above the center of the HUD scene is dynamic. It is accessed from the unrealscript and thus, a variable holding a reference to the label is defined. The label in the flash file is created using a Scaleform component and therefore, the variable used here is of type GFxClikWidget.

```
var GFxClikWidget InfoLabel;
```

To bind these variable to the respective one in the swf file, the following code is used:

```
event bool WidgetInitialized(name WidgetName, name WidgetPath, GFxObject
Widget)
{
    switch (WidgetName) // WidgetName = instance name of a Flash object.
    {
            case ('infoLabel'):
                  InfoLabel = GFxClikWidget(Widget);
                  break;
        default:
            break;
    }
    return true;
}
```

The MiniMap and the ExtendedInfo panels are hold in the varibles:

```
var TUC_FrontEnd_MiniMap MiniMap;
var TUC_FrontEnd_ExtendedInfo ExtendedInfo;
```

Their implementation is explained later on, but for now let us see how they are used (instantiated, and toggled) here. When the user presses the button for toggling the `MiniMap`, the Main Menu object calls the function `ToggleMiniMap` which is implemented in this class. Its code is shown below, and what it does is: It first checks whether the `MiniMap` is created. If it is created, meaning that the mMiniMap is displayed on the HUD, it closes the `MiniMap` movie and assigns null value to it. In the other case, when the miniMap is not displayed, it creates new `MiniMap` object it initializes it, and sets it location on the HUD and disables the scaling of that movie.

```
function ToggleMiniMap()
{
      if( MiniMap != none && MiniMap.bMovieIsOpen )
      {
            MiniMap.Close();
            MiniMap = none;
      }
      else
      {
            if( MiniMap == none )
            {
                  MiniMap = new class'TUC_FrontEnd_MiniMap';
                  MiniMap.Init();
                  MiniMap.SetViewScaleMode(SM_NoScale);
                  MiniMap.SetAlignment(Align_TopRight);
            }
```

```
	}
}
```

The method for toggling the Extended Info panel is the same as the above, but for completeness, the code is provided here.

```
function ToggleExtendedInfo()
{
	if( ExtendedInfo != none && ExtendedInfo.bMovieIsOpen )
	{
		ExtendedInfo.Close();
		ExtendedInfo = none;
	}
	else
	{
		if( ExtendedInfo == none )
		{
			ExtendedInfo = new class'TUC_FrontEnd_ExtendedInfo';
			ExtendedInfo.Init();
			ExtendedInfo.SetViewScaleMode(SM_NoScale);
			ExtendedInfo.SetAlignment(Align_BottomRight);
		}
	}
}
```

The last function defined in this class is the `UpdateMiniMap`. This function is called on every engine tick by the HUD Wrapper class. It checks whether the MiniMap is displayed, and if yes, it makes a call to the MiniMap's function `CalculateSpotPosition` which does the calculation for finding the correct player's position.

```
function UpdateMiniMap(float DeltaTime)
{
	if( MiniMap != none )
	{
		MiniMap.CalculateSpotPosition();
	}
}
```

The binding of the swf file, as well as setting up some default properties, is shown below. The meaning of the properties is obvious by looking at their names.

```
DefaultProperties
{

	WidgetBindings(0)={(WidgetName="infoLabel",WidgetClass=class'GFxClikWid
get')}
```

```
        MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_HUD'

        bIgnoreMouseInput = true
        bDisplayWithHudOff = false
        bEnableGammaCorrection = false
        bPauseGameWhileActive = false
        bCaptureInput = false;
}
```

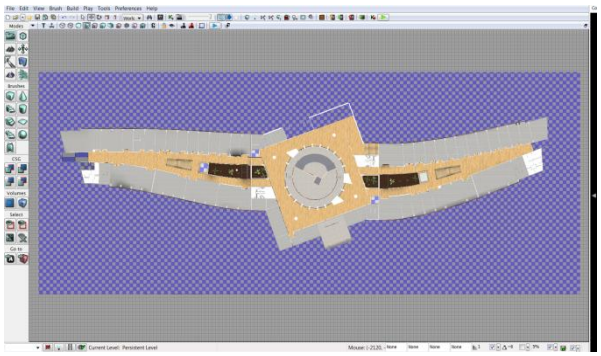
5.6.1 Mini Map

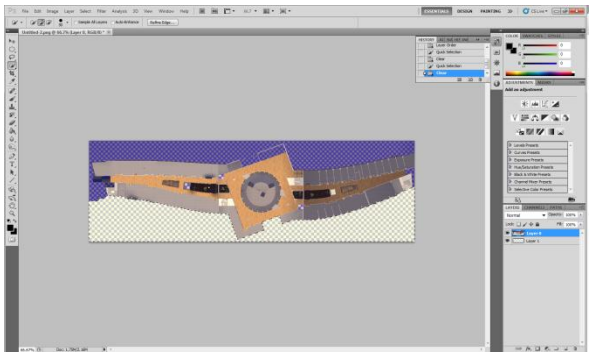
The mini-map is a miniature map of the virtual environment that is placed at the top right screen corner. On the top of the map there is a blip that shows the user's position. Its purpose is to aid users in having better orientation in the virtual world.

The mini-map is implemented using Scaleform/Flash and ActionScript for the interface that represents the mini-map and UnrealScript for updating the blip position on the mini-map.

Flash Setup

The images used for the world's map are screenshots of the facility that are taken in the UDK. In Adobe Photoshop, the image is resized, namely, its dimensions are set to be powers of two so that they are imported in udk (UDK)without problem. A new transparent layer is then added to the layer stack and it is placed at the bottom of the stack. Parts of the images are next converted to transparence in order to make visible only the facility. This is done by selecting the areas that are not needed with the 'Quick Selection Tool' and then removed by pressing the delete button. The procedure is illustrated in the next two figures:





The resulting images are imported to the flash library and then their properties are set up, namely, image smoothing is enabled and the compression is set to lossless PNG. At the end they are converted to movie clips.

Movie clips that represent the player blip and the semi trasparent black background are created from basic shapes.

All the movie clips that constitute the mini-map are added to the flash stage with the help of Actionscript. In the actionscript file that is associated with the flash file, the mini-map components are added by declaring them as instance variables. In the class constructor, these components are attached to the stage using the addChild function. The component that represents the first floor is set visible while the other two are hidden. This is done because the user starts using the application from the first floor. The class implementing the above stuff is as follows:

```
public class TUC_MiniMap extends MovieClip {

            /* instance variables */
            public var bg:Background = new Background();
            public var floor0:Floor0 = new Floor0();
            public var floor1:Floor1 = new Floor1();
            public var spot:SpotMC = new SpotMC();

            public function TUC_MiniMap() {
                // x and y coords of the minimap
                  x = 377;
                  y = 140;

                  rotationY = 35;     // rotate it

                  //attach the bacground
                  addChild(bg);

                  // attach the floor mc to the minimap
```

```
                addChild(floor0);
                addChild(floor1);

                // attach the player's spot
                addChild(spot);    // attach the user spot to the minimap

                floor0.visible = false;
                floor1.visible = true;
            }
        }
```

UnrealScript

In order to be used by the applicaton, the flash file created for the mini-map is first imported into UDK. The class that uses this file is named TUC_FrontEnd_MiniMap and it is an extension of the GFxMoviePlayer class. The purpose of the class is not only to display the file that was created in flash, but to map the player's position from the 3D virtual environment to the 2D mini-map. Note that the size of the two are different. The class is declared as follows:

```
class TUC_FrontEnd_MiniMap extends GFxMoviePlayer
      config(TUC);
```

The variables that are used for referencing the flash components are:

```
var GFxObject RootMC;                // the stage
var GFxObject Floor0MC, Floor1MC;    // current level map we are on
var GFxObject SpotMC;                // red dot on the minimap
```

The `RootMC` is the variable representing the root variable of the flash file, the `Floor0MC` and `Floor1MC` variables are the floor images and the `SpotMC` is the player's blip movie clip.

In order to map the player's position from the virtual environment to the 2D mini-map variables for holding the dimensions of the 3D world and the mini-map's dimensions are declared.

```
var float MiniMapWidth, MiniMapHeight;
var config float WorldWidth, WorldHeight;
```

The dimensions of the 3D world are larger than the dimensions of the mini-map, so variables are needed for storing the following information:how many times the world's width contains the mini-map's width and the same information for the heigth. Dividing the user's position by those ratios we can compute user's position on the mini-map. The variables used for the ratios are declared as follows:

```
var float xRatio, yRatio;
```

When objects of this class are instantiated, the first function that is called is the one shown below. The `Init` function initializes the flash movie, cashes a reference to the movie's components and calculates the width and height ratios. It is fired off by the `HUD` Wrapper when the `HUD` is instantiated.

```
function Init(optional LocalPlayer player)
{
	Start();
	Advance(0.f);

	RootMC = GetVariableObject("_root");	// caches a reference to the
	Floor0MC = RootMC.GetObject("floor0");	// level 0
	Floor1MC = RootMC.GetObject("floor1");	  // level 1
	Floor2MC = RootMC.GetObject("floor2");	  // level 2
	SpotMC = RootMC.GetObject("spot");	// reference to the User Spot

	MiniMapWidth = RootMC.GetFloat("width");
	MiniMapHeight = RootMC.GetFloat("height");

	xRatio = WorldHeight / MiniMapHeight;
	yRatio = WorldWidth / MiniMapWidth;
}
```

When the mini-map is displayed on the screen, the player's position on the mini-map is calculated on every tick. As presented above, this is done by dividing the players's position, namely, the x and y location, by the x and y ratio respectively. The 3D world and the mini-map have different coordinate system. The world's x axis is y axis in flash and also fash has inverted y axis. In order to overcome mapping problems, the function inverts the value of the calculated mini-map's y location. The function responsible for the calculation is shown below.

```
/**
 * CalculateSpotPosition
 * world width -> y, minimap width -> x
 * world height -> x, minimao height -> y
 */
function CalculateSpotPosition()
{
	local float x, y;

	x = GetPC().Pawn.Location.X;
	y = GetPC().Pawn.Location.Y;

	if( x >= 0 )
	{
	   x = -(x / xRatio);
	   y =  (y / yRatio);
	   SpotMC.SetPosition( y, x );
```

```
        }
        else
        {
            x = Abs(x / xRatio);
            y =  (y / yRatio);
            SpotMC.SetPosition( y, x );
        }
 }
```

When the user goes on another floor the mini-map is updated with the correct floor image:

```
function UpdateFloor(int l)
{
    if( l == 0 )
    {
            Floor0MC.SetVisible(true);
            Floor1MC.SetVisible(false);
    }
      else if( l == 1 )
      {
            Floor0MC.SetVisible(false);
            Floor1MC.SetVisible(true);
      }
}
```

5.6.2 Extended Info Panel

The extended info panel is a flash based interface that is placed at the bottom right corner of the screen. Its purpose is to aid users by providing information for the floor on which the user is at the current moment, for the department (HMMY, Geniko and Core) and the near "objects" such as Classrooms, Secretary office etc.

The extended info panel is implemented using Scaleform/Flash for the interface and UnrealScript for populating the panel with the information.

A populated info panel is shown in the figure below:

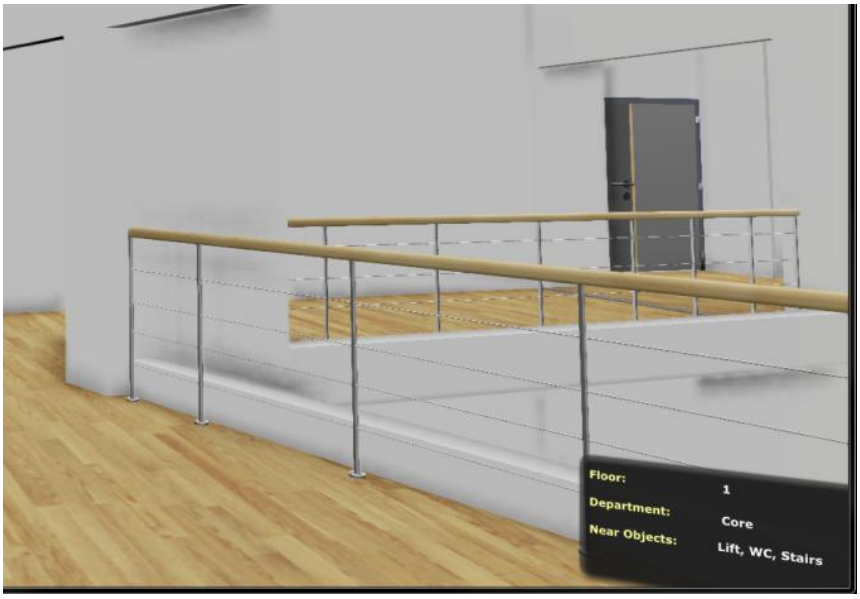


Figure 5.8. Info panel

Flash Setup

A flash file with dimmension of 310 x 160 is first created. It consists of three layers placed on the file's timeline. Thelayer on the top, named "Actions", is used for writing actionscript code. A single line of code is added here for the purpose of rotating the panel by 35 degrees. One layers(bez s) is used for placing the labels and the textfields, and one for the background. After creating the panel, it is published to swf file which is next imported in the UDK.

UnrealScript

Two classes are written in unrealscript in order to implement the Extenden Info functionality. One is for wrapping the flash file in unrealscript in order to access and populate its textfields, and the other one is used for detecting the user when it is near some objects of interest, such as stairs, classrooms, labs etc.

The floor and department detection, as well as the detection of the objects of interest, are represented by a class which stores information for the floor on which these objects are placed, the department in which they are placed and their name. This class is named `TUC_ExtendedInfo_TriggerVolume` and it is extension of the `TriggerVolume` class and allows the objects to detect user presence. The `TriggerVolume` is volume that is created using the builder brush and it is used as a non-interactive `Trigger`. When the user steps into the trigger volume, a touch event is fired. An untouched event is fired when the user steps out of the volume. Our class overrides those two events in order to implement the required behaviour.

```
class TUC_ExtendedInfo_TriggerVolume extends TriggerVolume
      ClassGroup(TUC);
```

The created volumes are shown in the following figure:

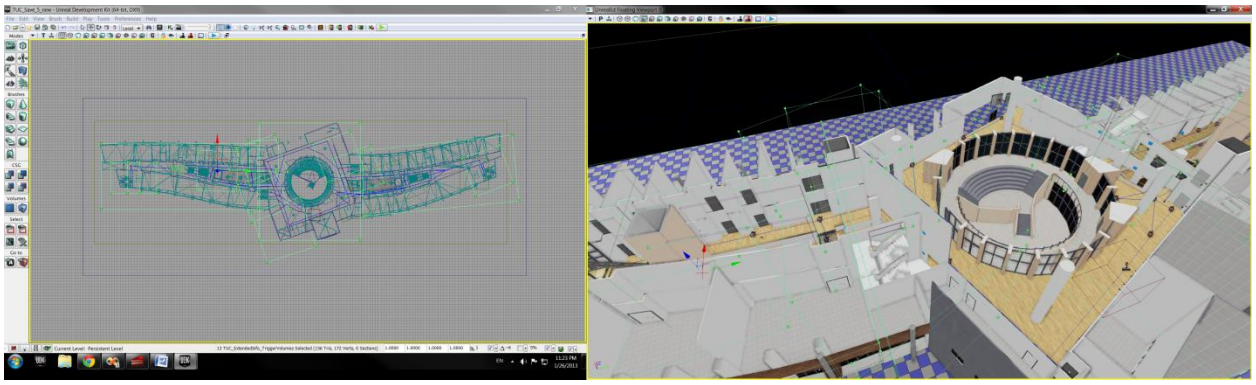


Figure 5.9. The detecting volumes

The stored information that was mentiond above is stored in the following variables:

```
// these values are modified in the editor
var() String Floor;
var() String Department;
var() String NearObject;

var TUC_FrontEnd_ExtendedInfo ExtendedInfoMovie;    // the movie that we will
write extended info data to
```

Some of the created volumes do not have assigned value to the `NearObject` variable. These volumes are used only for `Floor` and `Department` detection. The other volumes have assigned value only to the `NearObject`. Their values are used for updating the `NearObject` list in the `Extended Info Panel`.

When the user enters the trigger volume, the `Touch` event is fired. This event first checks if the actor that entered the volume is the player. Then it takes reference object that represents the flash file and checks to see if the volume is used for floor and department detection. If true, it passes the `Floor` and `Department` variables to the respectative `Setters` function. If false, it calls the `Setter` function for adding the name of the near object to the textfield of the flash file.

```
event Touch(Actor Other, PrimitiveComponent OtherComp, Vector HitLocation,
Vector HitNormal)
{
      super.Touch(Other, OtherComp, HitLocation, HitNormal);
```

```
      if( Pawn(Other) != none )
    {
            ExtendedInfoMovie =
TUC_FrontEnd_HudWrapper(WorldInfo.GetALocalPlayerController().myHUD).HudMovie
.ExtendedInfo;    // reference to the movie
            if( NearObject == "" )   // when used for floor and department
detection
            {
                  /* fill the movie with our data */

                  ExtendedInfoMovie.SetFloor(Floor);
                  ExtendedInfoMovie.SetDepartment(Department);
                  /* update the minimap with the correct floor */

      TUC_FrontEnd_HudWrapper(WorldInfo.GetALocalPlayerController().myHUD).Hu
dMovie.MiniMap.UpdateFloor(Floor);
            }
            else
            {
                  ExtendedInfoMovie.SetNearObjects(NearObject);
            }
    }
}
```

When the user steps out of the volumes, we want the name of the `nearObject` to be deleted from the panel. This is done in the `Untouch` event. The event checks if it was fired by the user, and when that is the case, it checks to see if this volume is used a `NearObject`. It then removes the name of the near object from the panel by calling the `RemoveNearObject` function.

```
event Untouch(Actor Other)
{
      super.Untouch(Other);

      if( Pawn(Other) != none )
      {
            if( NearObject != ""  )
                  ExtendedInfoMovie.RemoveNearObject(NearObject);
      }
}
```

TUC_FrontEnd_ExtendedInfo

In order to use the flash file, it has to be first wrapped in an unrealscript class. The class declaration is as follows:

```
class TUC_FrontEnd_ExtendedInfo extends GFxMoviePlayer;
```

The variables used for updating panel's texfields are:

```
var GFxClikWidget FloorLabel, DepartmentLabel;
var GFxObject NearObjects;     // textfiled polpulated with near bjects
var int NumOfNearObjects;      // how many near object are added to the list
```

The first two variables are of `GFxClikWidget` type and those are initialized in `WidgetInitialized` function in a manner that was seen many times, thus the function code is ommited here.

The `NearObjects` variable though, is of `GFxObject` type and holds reference to the flash `NearObjects` textfield. The reference is grabbed when the object is initialized using the `init` function.

```
function Init(optional LocalPlayer player)
{
	Start();
	Advance(0.f);

	NearObjects = GetVariableObject("_root.NearObjects");
}
```

The values of these instance variables are modified using setter functions. The `Floor` and the `Department` are variables that hold only one word, for example, HMMY, 1, Geniko etc, and these are simply assigned `String` values using the `SetText` function. As for the NearObjects, which can contain many words (because of their nature) we have to perform some checks and stick a comma between the different near objects.

```
/**
 * SetFloor
 */
function SetFloor(string Floor)
{
	FloorLabel.SetText(Floor);
}

/**
 * SetDepartment
 */
function SetDepartment(string Department)
{
	DepartmentLabel.SetText(Department);
}
```

```
/**
 * SetNearObjects
 * Add nearObject to the list
 */
function SetNearObjects(String no)
{
    local String s;
    s = NearObjects.GetText();

    if( s == "" )    // if empty textfield
    {
        NearObjects.SetText(no); // add it
    }
        else
    {
        s = s $ ", " $ no;  // stick comma and space at the begining of the
near objcet
        NearObjects.SetText(s);
    }
    NumOfNearObjects++;
}
```

When the user steps out of the `nearObject's trigger` volume that near object has to be deleted from the extended info panel. This is done in the `RemoveNearObject` function. The string representing a list of near objects is handled using string functions. These functions can not be used directly from this class because this class is not an extension of the `Actor` class. They are accessed using the `GetPC` helper function to get the owning player `controller` for this movie. If the list contains only one near object, it is simply removed by replacing the text of the string list with an empty string. If the list contains two or more near objects, a check is required to see if the object is the first in the list or no. If first, then it is removed from the list, as well as the succeeding coma. If not first, then it is removed together with the preceding comma.

```
function RemoveNearObject(String NearObject)
{
  local String s;
  s = NearObjects.GetText();

  if( s != "" )      // there are objects in the list
  {
    if( NumOfNearObjects > 1 )  // the list has two or more near objects
    {
      if( GetPC().InStr(s, NearObject) == 0 )    // if the string is at the
begining we need to remove and the succeeding comma
```

```
      {
         GetPC().ReplaceText( s, NearObject$", ", "");    // remove the
nearObject and the succeeding comma
         NearObjects.SetText(s);
      }
      else        // the string is not the first object
      {
        GetPC().ReplaceText( s, ", "$NearObject, "");  // remove the
nearObject and the preceeding comma
        NearObjects.SetText(s);
      }
    }
    else  // the list has exactly one near object
    {
      GetPC().ReplaceText( s, NearObject, "");
      NearObjects.SetText(s);
    }
    NumOfNearObjects--;
  }
```

5.7 Displaying Professor Information

While navigating in the virtual university environment, it is of great importance for the user to read screen-displayed information about the academic staff.
This application provides such information. There are two types of information provided: Basic and Advanced.

The Basic information is displayed on a small panel when the user simply looks at a professor office door while is located at a certain distance. The displayed information is the professor's name, its picture and a text message telling the user to press the Q button if he or she wants to see more information. Figure 5.10 shows the Basic information interface. The showing and hiding of this interface is part of the HUD system and for that reason it is explaind in the HUD section. The system that allows to detect the actor that the user is looking was explaind in section 5.3.
The rest of the section is focused on the Advanced information panel.

Figure 5.10. Basic professor information

The Advaced information is displayed on the screen when the user presses the 'Q' keybrord button while it is located near professor's office and is looking at the office door. Some of the displayed information are: Professor's name, its position at work, the department in which is employed, its specialities, research areas, education, the office number etc.

The interface on which the information are presented is created using flash technology. Characteristic of this interface is that it belongs to the desktop three dimensional family of interfaces. This in fact means, that, exept x and y axis, the component constituting the interface have z axis which gives them sense of depth. The 3D transformations are achived using the ActionScript 3.0 built-in 3D functionality. A flash object essentially has three 3D transforms that can be applied to it: X rotation, Y rotation, and Z translation The created flash file is imported into the UDK and wrapped in an unrealscript class. The unrealscript class loades the data about the professors from a configuration file. The implementation of this functionality is thoroughly explained in the next two sections.

5.7.1 Creation of the Professor Info Interface using Flash and ActionScript 3.0

The user interface for showing information for the professors is created using Scaleform-Flash and ActionScript 3.0. It consists of three main components which are placed on three different layers. In order to achieve feeling of 3D space, the layers are given different depth by translalating them on the z axis. This is done by giving values to the z variable on their timeline as shown below:

```
z = 200;
```

Note that negative values of the z translate the object toward the screen and positive values translate it away from the screen.

The first layer is used as a background and as a place where the field labels are placed. Movie clips that represent sort of 'shadows' used for enhancing the depth of the interface are placed on the second layer. Finally, the fields that contain the information about the professors are placed on the third layer. These fields are loaded with data from configuration file at the run time. The professor icons are imported from the library that is created for the project and dynamically loaded when needed. Figure 5.11 shows the final result.

Figure 5.11. Professor Information projected on a 3D UI

Now we are going to examine the ActionScript side of the interface.
The rotation of the interface is implemented in the associated actionscript file called TUC_ProfInfo.as as follows:

```
public class TUC_ProfInfo extends MovieClip
```

The rotation of the interface is enabled when the user has first pressed any of the mouse buttons. To check this condition a boolean flag called `isRotationEnabled` is declared. It is set to true when the user clicks on any of the mouse buttons. At the variable declaration section, variables that hold the coordinates of the mouse at the moment of the clicking and reference to the actual interface are declared. The last variables used in the script are those that store the x and y rotation of the interface.

```
var isRotationEnabled:Boolean = false;
var InitialMouseCoordX:Number = 0;
var InitialMouseCoordY:Number = 0;
var InfoPanelRotationY:Number = 0;
var InfoPanelRotationX:Number = 0;
public var InfoPanelMC:InfoPanel;
```

In the class constructor, the mouse related event listeners are added to the stage object of the flash file. The Stage in Adobe Flash Professional is the rectangular area where the graphic content is placed when creating documents. These listeners listen for events such as mouse pressing, mouse releasing and mouse movement.

```
public function TUC_ProfInfo()
{
    stage.addEventListener(MouseEvent.MOUSE_DOWN, enableRotation);
    stage.addEventListener(MouseEvent.MOUSE_UP, disableRotation);
    stage.addEventListener(MouseEvent.MOUSE_MOVE, MouseMove);
}
```

When the user presses any of the mouse buttons, the enableRotation function is fired. This function sets the isRotationEnabled flag to true and grabs the coordinates of the mouse cursor and the rotation of the interface.

```
function enableRotation(evt:MouseEvent):void
{
    isRotationEnabled = true;
    InitialMouseCoordX = root.mouseX;
    InitialMouseCoordY = root.mouseY;
    InfoPanelRotationX = InfoPanelMC.rotationX;
    InfoPanelRotationY = InfoPanelMC.rotationY;
```

```
}
```

Upon releasing the pressed button, the disableRotation function is called. What this function does is that it disables the interface rotation.

```
function disableRotation(evt:MouseEvent):void
{
    isRotationEnabled = false;
}
```

The next function does the actual rotation of the interface. It first checks if the rotation flag is true, and if true, it calculates the rotation of the interface.

In order to understand the computation of interface rotation the Flash coordinate system has to be explained first. The origin point of the stage (the main timeline) is always in its upper left corner. The main difference with regard to the classic system is that Flash has an inverted Y axis. Therefore in Flash, the negative Y values are above the origin point and the positive ones below it. Here is a visual representation:

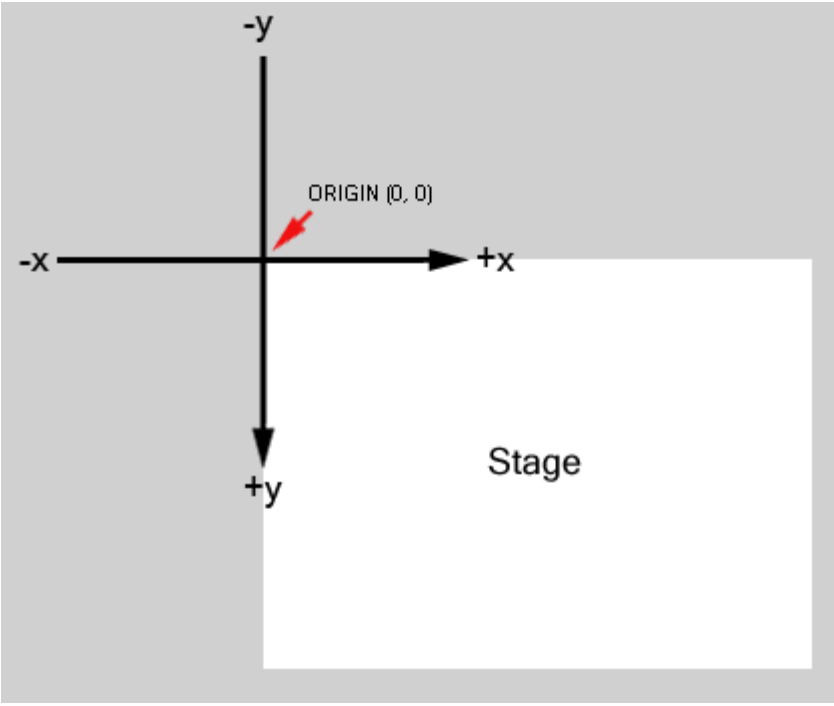


Figure 5.12. Flash Coordinate System

We can conclude from the above figure that when the user moves the mouse in the x axis, we want the interface to rotate in its y axis and vice versa.

```
function MouseMove(evt:MouseEvent):void
{
    if (isRotationEnabled)
    {
        InfoPanelMC.rotationY = InfoPanelRotationY + (root.mouseX -
InitialMouseCoordX);
        InfoPanelMC.rotationX = InfoPanelRotationX + (root.mouseY -
InitialMouseCoordY);
    }
}
```

In the next section we are going to describe how the flash file is connected with the application and how the interaction with interface is possible.

5.7.2 Connecting the User Interface with the application - UnrealScript

The file that is created in Flash is first imported into the UDK and then wrapped in an unrealscript class. The class that is discussed at this point is declared as follows:

```
class TUC_FrontEnd_ProfInfo extends GFxMoviePlayer;
```

Variables that hold references to the components of the flash file and that need to be populated are shown below. These are not explained for they are, all in all, self explanatory.

```
var GFxObject RootMC;                // the root object of the file
var GFxObject InfoPanelMC;           // the interface panel
var GFxObject Layer2;     // the layer on which are placed the info fields
var GFxObject ProfImageHolder;    // icon holder

/* Professor Data */
var GFxObject PName;
var GFxObject Position;
var GFxObject Department;
var GFxObject Sector;
var GFxObject Speciality;
var GFxObject Edu;
var GFxObject CV;
var GFxObject ResearchAreas;
var GFxObject Courses;
var GFxObject Phone;
var GFxObject Email;
var GFxObject OfficeID;

var String ProfessorName;
```

The class that handles loading the data from a configuration file is TUC_InfoData and it is explained in the section x.x

```
var TUC_InfoData Info;                // array with professors
var TUC_InfoData.Professor PData;     // structure that holds data for a
specified profssor
```

Having this class instantiated, it is initialized by callind the function `InitMovie`. `InitMovie` Initializes everything, takes references to the text filds that we want to populate with the porfessor data. It accepts a string which is the name of the specified professor.

```
 */
function InitMovie(String ProfName)
{
	Info = new class'TUC_InfoData';
	Start();
	Advance(0.f);

	ProfessorName = ProfName;

	RootMC = GetVariableObject("root");
	InfoPanelMC = RootMC.GetObject("InfoPanelMC");
	Layer2 = InfoPanelMC.GetObject("Layer2MC");
	ProfImageHolder = Layer2.GetObject("ProfImageHolder1");

	PName = Layer2.GetObject("ProfName");
	Position = Layer2.GetObject("Position");
	Department = Layer2.GetObject("Department");
	Sector = Layer2.GetObject("Sector");
	Speciality = Layer2.GetObject("Speciality");
	Edu = Layer2.GetObject("Education");
	CV = Layer2.GetObject("CV");
	ResearchAreas = Layer2.GetObject("ResearchAreas");
	Courses = Layer2.GetObject("Courses");
	Phone = Layer2.GetObject("Phone");
	Email = Layer2.GetObject("Email");
	OfficeID = Layer2.GetObject("OfficeID");

	PopulateProfInfo();
}
```

`PopulateProfInfo` function as as suggested by its name, populates the interface with the correct data. It first removes the space from the string containing the professor name. This is done because the icon names are stored without the space between the first name and the last name. Next, the icon is attached to to the icon holder. information for the professor, except the associated icon are stored in the

Info class and are accessed here by calling the `FindProfessor` function. `FindProfessor` accepts a string which is used for searching in the array of professor and returns a structure with the data for the specified professor. Using the data from the returned structure, we populate the text fields of the interface with help of the SetText function.

```
function PopulateProfInfo()
{

	local string ImageName;
	ImageName = ProfessorName;

	ImageName = Repl(ImageName, " ", "");   // remove the space in the name
	ProfImageHolder.AttachMovie(ImageName, "KM");  // attach the associated
icon for this prof
	PData = Info.FindProfessor(ImageName); // load the data for the
specified professor

	PName.SetText(PData.PName);
	Position.SetText(PData.Position);
	Department.SetText(PData.Department);
	Sector.SetText(PData.Sector);
	Speciality.SetText(PData.Speciality);
	Edu.SetText(PData.Edu);
	CV.SetText(PData.CV);
	ResearchAreas.SetText(PData.ResearchAreas);
	Courses.SetText(PData.Courses);
	Phone.SetText(PData.Phone);
	Email.SetText(PData.Email);
	OfficeID.SetText(PData.OfficeID);
}
```

As previously stated in the introduction, in order to activate the interface (to display it on the screen) the user has to be facing the office door not further than a predefined distance. This is achived using Triggers. Basicaly, we crated a class that is extension of the Trigger class.

```
class TUC_ProfInfo extends Trigger
	placeable
	ClassGroup(TUC);
```

This class does not implement anything at all nor overrides the functionality defined in the parent class. It is simply used because we want to detect it later on, as a different class.

The final peace that is missing for the completion of the functionality is binding the 'Q' keybord button with an exec function. The way that this is done is by adding the key binding into the DefaultInput.ini file:

```
.Bindings=(Name="Q",Command="TUC_ShowProfInfo")
```

The exec functions that is mapped to the 'Q' key, is implemented in the TUC_FrontEnd_HudWrapper class. Note that the job of TUC_FrontEnd_HudWrapper is to instantiate the HUD and to tell it to update after each frame is rendered. To allow the function to be exectuted by the user at runtime, the exec keyword is added in front of the function. Its purpose, however, is to show and hide the interface.

```
exec function TUC_ShowProfInfo()
{
    if( ProfInfoMovie != none && ProfInfoMovie.bMovieIsOpen )
      {
            ProfInfoMovie.Close();
            ProfInfoMovie = none;
            bShowHUD = true;
      }
      else
      {
            if( ProfInfoMovie == none && ActorTooltip.bUserIsLooking )
            {
                  ProfInfoMovie = new class'TUC_FrontEnd_ProfInfo';
                  ProfInfoMovie.InitMovie(ActorTag);
                  ProfInfoMovie.SetViewScaleMode(SM_NoScale);
                  bShowHUD = false;
            }
      }
}
```

5.8 Path Finding

The path finding functionality provides navigational aid for the application users and it assists them in locating professors' offices, classrooms, secretary office, cafeteria and other staff facilities.

To activate this functionality, the user has to choose the option "Find Path" in the Main Menu. After pressing that button, the "Find Path" menu page opens. This two main componetns of this menu page are one dropdown menu from which the user

can chose destination location and a button ("Find Path") to start the functionallity. At this screen the user can also use a back button for returning to the main menu or a exit button to completly exit the menu. When pressing the "Find Path" button, the menu page is closed and a 3D arrow is displayed in the virtual scene in front of the user. While navigating through the virtual building the arrow is showing the selected path by rotating left and right, and the user has to follow the arrow in order to reach the desired destination. While the player is in this "searching" state, a text is being displayed on the HUD informing the player of the current player's state. When the user reaches the destination the arrow disappears (is destroyed) and sound is played.

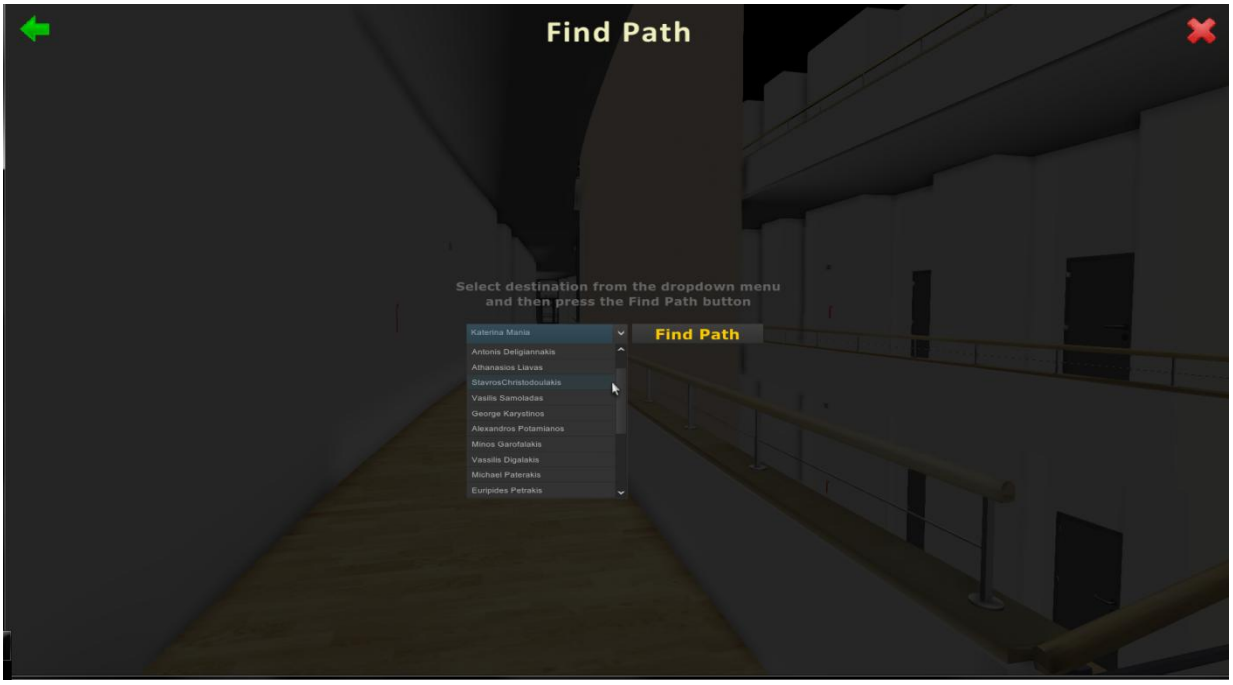


Figure 5.13. The user interface of the Find Path functionallity. In order to start path finding the user has to select one of the posible dropdown menu entries and then press the "Find Path" button

Figure 5.14. The green arow shows the way to the selected destination location

5.8.1 Setting Up the Navigation Network

The navigation in this application is based on a pre-generated path network. This is done in the virtual scene using the *Unreal Editor*. First and foremost, `PathNodes` (a subclass of `NavigationPoint`) were placed in the virtual scene on surfaces which the player can walk on. For the `PathNodes` to connect, they were placed less than 1200 Unreal units apart. The goal was to cover every area of the level excluding the offices, classrooms and the elevators. After placing the `PathNodes`, connections between the `NavigationPoints` are built from the Build menu by selecting Rebuild All option. Portion of the navigation network that was created for this application is shown in the next figure:

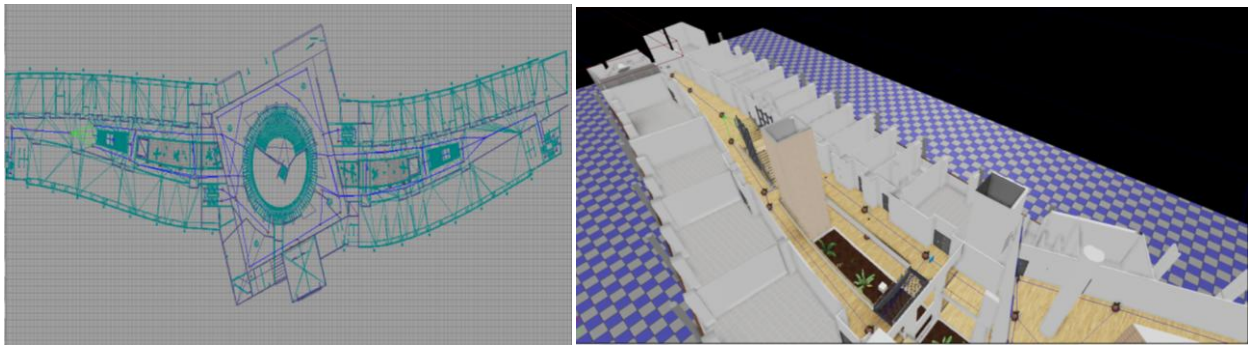


Figure 5.15. The Navigation Network consist of interconnected PathNodes (apples)

The implementation of this functionality is described below in details.

The menu page from which the user chooses destination location is created in Flash. The interface is shown in Figure 5.12. From the figure it can be seen that interface contains two labels, three buttons and one dropdown menu. All of the buttons are creted using the CLIK DefaultButton component. After adding the buttons on the stage they are skinned in order to take the desired shape. The dropdown menu is also a CLIK component that constitutes of three different subcomponents. These are: ListItemRender, ScrollingList and a ScrollingBar. Clicking on the dropdown menu component opens a list that contains the elements to be selected. The menu list is populated with professor names, classrooms and labs inside the associated Unrealscript class.

The Unrealscript classes that implement this functionality are:

- TUC_NavArrow
- TUC_NavTargetNode
- TUC_FrontEnd_ShowPathMenu
- TUCPlayerController

TUC_NavArrow

Instances of this class represent the arrow that shows the path. In the default properties of the class first we define Light Environments which allow a non-static actor to be lighted by static lights. Then the code defines a static mesh which represents the arrow. The static mesh is loaded from the library and the light environment is assigned to it. Eventually, the static mesh is added to the components of the class instance. The Implementation can be seen in the following code chunk.

```
class TUC_NavArrow extends Actor;

DefaultProperties
{
    Begin Object Class=DynamicLightEnvironmentComponent
Name=MyLightEnvironment
        bEnabled=TRUE
    End Object
    Components.Add(MyLightEnvironment)

    Begin Object class=StaticMeshComponent name=NavArrowMesh
        StaticMesh=StaticMesh'TUC_Misc.SM.SM_TUC_Arrow_01'
        LightEnvironment=MyLightEnvironment
      End Object
      CollisionComponent=NavArrowMesh
```

```
      CollisionType=COLLIDE_TouchAll
    Components.Add(NavArrowMesh)
}
```

5.8.2 Target Nodes

This class `TUC_NavTargetNode` represents the target object of the FindPath functionality. It is placed in the world at different locations that represent target objects. These target objects are in fact professors' offices, secretary office, classrooms etc. They are used in the FindPathToward algorithm as target. In the UDK editor, names are given for the actors in the tag field. Tags are compared with the user input string in TUC_FrontEnd_ShowPathMenu menu page when the user enters destination string.

The class extends the Trigger class, it is placeable and put in TUC group for better organization and finding in the editor.

```
class TUC_NavTargetNode extends Trigger
      placeable
      ClassGroup(TUC);
```

In order to implement the wanted behavior, the well known `Trigger's Actor` events and functions are overridden. To be more precise, these are `Touch` and `Untouch` event and the `UsedBy` function. When the user is in the "Path Finding" state and navigates in the world he/she eventually collides with this Node. When the collision ocures, the `Touch` function is fired. In the `Touch` event the parent implementation of the function is called first. Then the user checks if this `NavNode` is the target node that is found in the controller class. If this is true the flag `bNavFindPath` is set to false because the destination is reached and the finding path has to be stopped. At the same time, the arrow is destroyed, sound is played announcing that the destination is reached and the target object is set to none for the next search. At the same time the string "Destination Reached" is also printed on the `HUD`.

```
event Touch(Actor Other, PrimitiveComponent OtherComp, Vector HitLocation,
Vector HitNormal)
{
    super.Touch(Other, OtherComp, HitLocation, HitNormal);

    if( Pawn(Other) != none )
    {
       if(TUCPlayerController(Pawn(Other).Controller).NavTargetObject==self)
       {
```

```
            TUCPlayerController(Pawn(Other).Controller).bNavFindPath = false;
            TUCPlayerController(Pawn(Other).Controller).NavArrow.Destroy();
            PlaySound(SoundCue'TUC_Sounds.Sounds.S_NavDestinationReached');
      TUC_FrontEnd_HudWrapper(WorldInfo.GetALocalPlayerController().myHUD).Hu
            dMovie.InfoLabel.SetText("Destination Reached");
            TUCPlayerController(Pawn(Other).Controller).NavTargetObject =
none;
            }
            IsInInteractionRange = true;
    }
}
```

When the `Player` exits the `Trigger`, `Untouch` event is fired. In this event the text from the `HUD` that informs the player that he/she has reached the destination is removed.

```
event Untouch(Actor Other)
{
    super.Untouch(Other);

    if( Pawn(Other) != none )
    {

      TUC_FrontEnd_HudWrapper(WorldInfo.GetALocalPlayerController().myHUD).Hu
      dMovie.InfoLabel.SetText("");
      IsInInteractionRange = false;
    }
}
```

At the end of the class, in the default properties, the sprite of the `Trigger` is scaled to 0.2 because it is too big and annoying in the editor, and the radius is increased to 80.0 to meet the requirements.

```
DefaultProperties
{
      Begin Object Name=Sprite
            Scale=0.2
      End Object
      Begin Object NAME=CollisionCylinder
            CollisionRadius=+0080.000000
      End Object
}
```

5.8.3 Connection of the UI

The `TUC_FrontEnd_ShowPathMenu` class is a wrapper class for the TUC_ShowPathMenu Flash file.

```
class TUC_FrontEnd_ShowPathMenu extends GFxMoviePlayer
      dependson(TUC_InfoData);
```

The `dependson` modifier tells the compiler to process the `TUC_InfoData` class first because this class depends on a struct declared in that class. `TUC_InfoData` holds data for professors, classrooms and labs.

Four global variables of `GFxClickWidget` type are defined at the beginning of the class. These are: `BtnFindPath, BtnBack, BtnExitMainMenu, DropdownMenu`. They are used for holding references to the CLIK components in the flash file. An Unrealscript `GFxClikWidget` is like a wrapper for an ActionScript CLIK component. The `GFxClikWidget` class extends `GFxObject`, which is a wrapper for the ActionScript `Object` class (the root class for all other classes). Buttons' functionality is easily understood frome their names.

```
var GFxClikWidget BtnFindPath, BtnBack, BtnExitMainMenu, DropdownMenu;
```

Next step is write a function which is called after instances of this class are created. The initialization is performed in the `Init` function. The flash movie is initialized and stared by calling the `Start` function and the play head is placed to the first frame by calling the `Advanced` function.
Then, references to the `root` and `cursor` variables from the swf are grabbed and assigned to `RootMC` and `CursorMC GFxObject` variables.

```
function Init(optional LocalPlayer player)
{
    Start();
    Advance(0);
    SetViewScaleMode(SM_NoScale);
    RootMC = GetVariableObject("_root");
    CursorMC = RootMC.GetObject("cursor");
    SetCursorPosition();
}
```

The movie's buttons and dropdown menu are initialized using the `WidgetInitialized` mechanism. This event is fired when the CLIK components in the associated Flash file call this function after they are done with all the ActionScript initialization. This function simply checks the name of the ActionScript CLIK widget and uses a switch statement to assign it to the correct `GFxClikWidget` Unrealscript object after which an event listener function is added to it. These functions are fired when the buttons are clicked.

```
event bool WidgetInitialized(name WidgetName, name WidgetPath, GFxObject
Widget)
{
    local bool bWasHandled;
    bWasHandled = false;

      switch(WidgetName)
      {
            case ('btnFindPath'):
                  BtnFindPath = GFxClikWidget(Widget);
                  BtnFindPath.AddEventListener('CLIK_click', FindPath);
                  bWasHandled = true;
                  break;
            case ('dm'):
                  DropdownMenu = GFxClikWidget(Widget);
                  PopulateDropdownMenu();
                  bWasHandled = true;
                  break;
            case ('btnBack'):
                  BtnBack = GFxClikWidget(Widget);
                  BtnBack.AddEventListener('CLIK_click', Back);
                  bWasHandled = true;
                  break;
            case ('btnExitMainMenu'):
                  BtnExitMainMenu = GFxClikWidget(Widget);
                  BtnExitMainMenu.AddEventListener('CLIK_click',
ExitMainMenu);
                  bWasHandled = true;
                  break;
            default:
                  break;
      }
      return bWasHandled;
}
```

Furthermore, the event listener functions that are attached to the buttons are described next.

When the user presses the back button (green arrow), the `Back` function is fired. This function returns the user to the main menu page by calling the `LoadNextMenu` (implemented in the FrontEnd class) with the `TUC_FrontEnd_MainMenu` class as parameter.

```
function Back(EventData data)
{
      LoadNextMenu(data, class'TUC_FrontEnd_MainMenu');
}
```

When the user presses the exit main menu button (the red x) the function closes the flash movie and tells the TUCPlayerController to take the needed actions.

```
function ExitMainMenu(EventData data)
{
	Close();
	TUCPlayerController(GetPC()).MenuManager();		// PC will handle
}
```

After the dropdown menu is initialized, it is populated with destination locations. These locations are professor offices, clasrooms and labs. It was seen above that this data is found in instances of the `TUC_InfoData` class. The CLIK components are populated using the `DataProvider` object. `DataProvider` is basicaly a array of strings. So, first the `DataProvider` object is created and then populated in a foor loop. After that, the `DataProvider` is added to the dropdown menu. These step are shown next.

```
function PopulateDropdownMenu()
{
	local byte i;
	local GFxObject DataProvider;
	Info = new class'TUC_InfoData';
	DataProvider = CreateObject("scaleform.clik.data.DataProvider");

	for (i = 0; i < Info.Professors.Length; i++)
	{
		DataProvider.SetElementString(i, Info.Professors[i].PName);
	}

	DropdownMenu.SetObject("dataProvider", DataProvider);
}


function FindPath(EventData data)
{
	local int i;
	TUCPlayerController(GetPC()).btnFindPathPressed_FE_SPM = true;
	i = DropdownMenu.GetInt("selectedIndex");
	TUCPlayerController(GetPC()).NavTargetString=Info.Professors[i].PName;
	Close();
	TUCPlayerController(GetPC()).MenuManager();		// PC will handle
}
```

The last step in setting up the CLIK widget initialization is to add each `GFxClikWidget` variable to the `WidgetBindings` array. This is done in the default

properties of the class. This is also the place where the class is associated with the flash file.

```
DefaultProperties
{
MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_ShowPathMenu'
WidgetBindings.Add((WidgetName="textInput",WidgetClass=class'GFxCLIKWidget'))
WidgetBindings.Add((WidgetName="btnSearch",WidgetClass=class'GFxCLIKWidget'))
WidgetBindings.Add((WidgetName="btnCancel",WidgetClass=class'GFxCLIKWidget'))
}
```

TUCPlayerController

This is the class where, among many other things, navigation logic is implemented. In this paragraph, the code which is responsible for the FindPath functionality is thoroughly explained.

Variables of the above classes are defined at the beginning of the class among many other data types. These are:

`var TUC_NavArrow NavArrow` - which represents the 3D arrow that shows the path.
`var String NavTargetString` – Holds the user input in TUC_FrontEnd_ShowPathMenu class.
`var TUC_NavTargetNode NavTargetObject` - holds the target object of our search
`var bool bBtnSearchPressed_FE_SPM` – flag
`var bool bNavFindPath` - flag
`var Vector NavArrowLocation` – Vector that holds the location of the arrow

When the user enters a search string and presses the "Search" button in FindPath menu page, the program flow continues to the MenuMenager function as explained above. This function checks if the `bBtnSearchPressed_FE_SPM` flag is set to true which means that the "Search" button is pressed. If this flag is set to true, the function does the following steps:

- It finds the target object and assigns it to the NavTargetObject variable so that we have reference.
- Next, the arrow is destroyed so that we do not have two arrows if the user was previously searching.
- Next, the arrow is spawned and attached to the Pawn.
- The flag `bNavFindPath` is set to true. This flag is used in the search algorithm.
- "…Finding Path" string is displayed on the screen.

```
if( bBtnSearchPressed_FE_SPM )
{
	bBtnSearchPressed_FE_SPM = false;
	NavTargetObject = NavFindTarget(NavTargetString);
	NavArrow.Destroy();   // we don't want two arrows if the user
	makes another search without reaching the first destination
	SpawnAndAttachNavArrow();
	bNavFindPath = true;

	TUC_FrontEnd_HudWrapper(myHUD).HudMovie.InfoLabel.SetText("...Finding
                                                                Path");
	GotoState('PlayerWalking');
}
```

The two functions (`NavFindTarget` and `SpawnAndAttachNavArrow`) that are seen in the above code chunk are described below.

`NavFindTarget` iterates through all TUC_NavTargetNode Actors placed in the virtual scene to find the target object and return it to the caller. It accepts string parameter which is the string that the user enteres and represents the target of the search. A temporal `NavTargetNode` variable is created inside the function body. Then the function iterates through all the actors TUC_NavTargetNode Actors placed in the virtual scene and when the tag of the current node equals the Target string the function returns that node to the caller.

```
function TUC_NavTargetNode NavFindTarget(String Target)
{
	local TUC_NavTargetNode t;

	foreach AllActors( class 'TUC_NavTargetNode', t )
	{
		if( t.Tag == Name(Target) )
		{
			return t;
			break;
		}
	}
	return t;
}
```

Spawning and Attaching the Arrow

The `Spawning` of the arrow in the virtual environment and `Attaching` it to the user is handled by the `SpawnAndAttachNavArrow` function. The navigation arrow is a subclass of `Actor`. That means that the `Spawn` method had to be used in order to create the object instead of using the new keyword. The `Spawn` function can only be called from

a non-static function and can only create `Actors` of class that are specified in its parameter list in contrast with the `new` keyword which is used to create new object that are not a subclass of `Actor`. The returned object from the `Spawn` function is stored in the `NavArrow` variable.

We want the arrow to be set in front of the user and that is moving together with him. The relative (to the `Pawn`) location of the arrow is stored in the `NavArrowLocation` vector. The arrow is then attached to the pawn using the `SetBase` function after which the location is set using the `SetRelativeLocation` function.

```
function SpawnAndAttachNavArrow()
{
	/* spawn the navigation arrow that will show us the path to the
objective */
	NavArrow = Spawn(class'TUCProject.TUC_NavArrow', self);
	/* set the location of the arrow */
	NavArrowLocation.Z = 40;
	NavArrowLocation.X = 90;
	NavArrowLocation.Y = 0;
	if (NavArrow != none)
	{
		NavArrow.SetBase(Pawn);   // attach it to the pawn
		NavArrow.SetRelativeLocation(NavArrowLocation);
	}
}
```

The Actuall path finding

The actual path finding logic is implemented in the `FindPath` function. It was seen that when the level is built, the NavigationPoints placed in the virtual environment generate Navigation Network between them which has data used by path finding which allows the player to find the path from its position to the goal. The route from the starting position to the goal is created by the built-in native function called `FindPathToward`. This function executes a traversal of the navigation network generating a route from the `Player's` position to the passed goal, placing the generated route (An ordered list of `NavigationPoints`) in the `RouteCache` array of the `Controller`. This finction is called on every user's movement when the path find functionality is activated. The arrow direction (orientation) is calculated by, first, subtructing its position from the position of the next `NavigationPoint` and then using the resulting vector in a rotator that rotates the arrow to face the next NavigationPoint.

```
function FindPath()
{
```

```
	local Vector ArrowDirection;
	local Rotator ArrowRotation;
	local NavigationPoint tempNavPoint;
	local int Length;
	local float Distance;

	/* Create the Path - Searches the navigation network for a path to the
	   node closest to the given actor */
	FindPathToward(NavTargetObject);

	Length = RouteCache.Length;

	if( Length != 0 )
	{
		tempNavPoint = RouteCache[0];
		Distance = VSize(Pawn.Location - tempNavPoint.Location);

		if( (Length >= 2) && (Distance < 160) )
		{
			tempNavPoint = RouteCache[1];
			ArrowDirection = tempNavPoint.Location - NavArrow.Location;
			ArrowRotation = Rotator(ArrowDirection);
			ArrowRotation.Pitch = 0;
		}
		else if ( (Length == 1) && ( Distance <160 ) )
		{
			ArrowDirection = NavTargetObject.Location -
					NavArrow.Location;
			ArrowRotation = Rotator(ArrowDirection);
			ArrowRotation.Pitch = 0;
		}
		else
		{
			tempNavPoint = RouteCache[0];
			ArrowDirection = tempNavPoint.Location - NavArrow.Location;
			ArrowRotation = Rotator(ArrowDirection);
			ArrowRotation.Pitch = 0;
		}
	}
	else
	{
		ArrowDirection = NavTargetObject.Location - NavArrow.Location;
		ArrowRotation = Rotator(ArrowDirection);
		ArrowRotation.Pitch = 0;
	}

	NavArrow.SetRotation(ArrowRotation);
}
```

5.9 Interactive Map

The Interactive Map functionality allows the user to see VE's relative information in a map-based manner. The interface itself is implemented in Flash and it is accessed from the main menu. The interface consists of several parts. These are: Buttons for displaying one of the three floors, a Main area where a map with buttons is shown, buttons for closing the interface and returning to the main menu, next, there is a legend at the bottom right corner, and last, at the left bottom corner there is a panel for displaying information when the user presses one of the buttons on the map. Figure 5.16 presents the Interactive map.

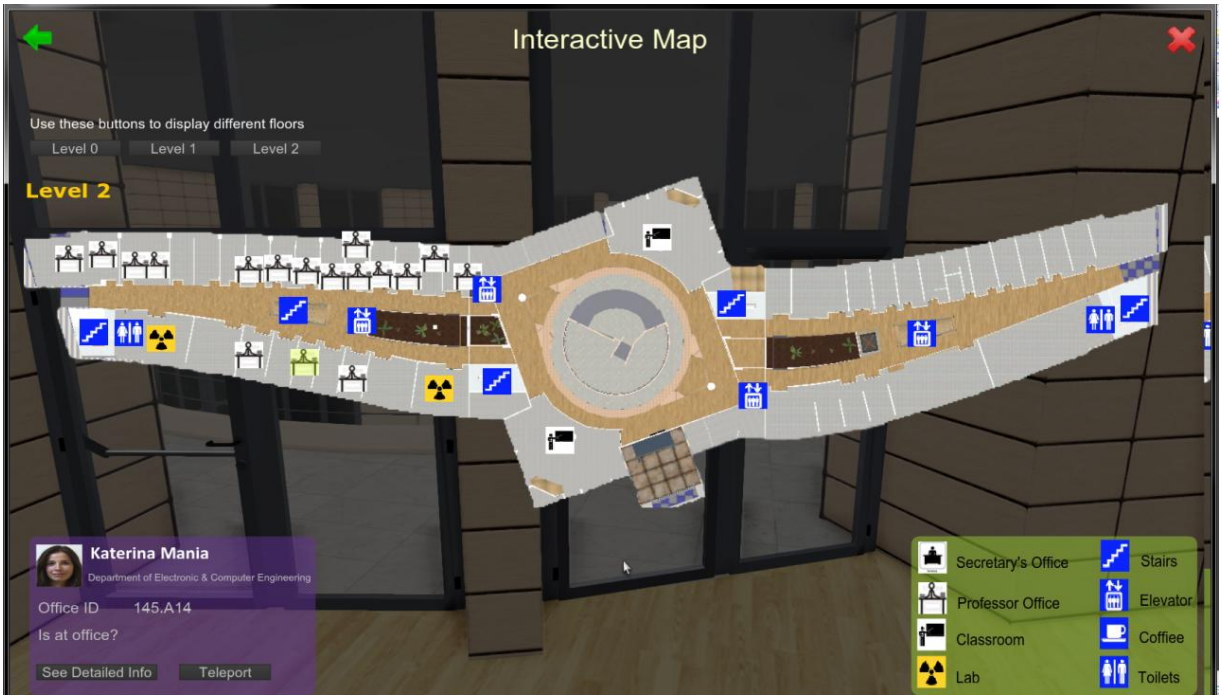


Figure 5.16. Interactive Map

The Interactive Map is accessed from the application's main menu. When the "Show Interactive Map" is selected from the menu, the program starts playing the associated flash file. The starting scene of the movie includes the name of the functionality, three buttons for showing the three floors and a legend showing the meaning of the symbols which are placed on the map (when activated). All of these movie clips and buttons are static objects that are placed in the flash scene. Figure 5.17 shows the flash scene loaded in the flash editor. The layers created for better organizing the work can be seen in the timeline window. The Buttons layer holds

the buttons for canceling the menu option and returning to the main menu (the green arrow and the red x), and three buttons used for dynamically loading one of the three floors. The BG layer is used for the dark transparent background, the Main layer is used for the map that is displayed in the center of the scene, when clicking on one of the related buttons, and last, the Info layer holds the legend panel (the green panel at the bottom).

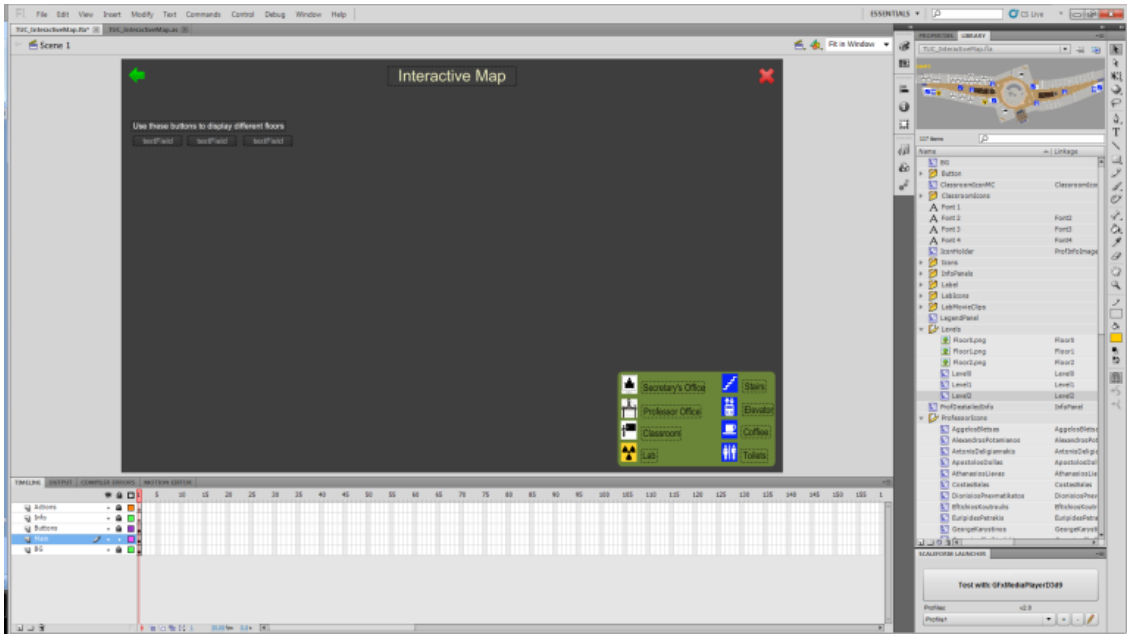


Figure 5.17. The IM flash scene loaded in Flash Editor

On the right side, in the library window, are shown all the assets used in this flash file. The assets are buttons, movie clips and images.

With the exception of the static content shown in Figure 5.17, the scene includes some other movie clips which are added to the scene using ActionScript. These movie clips represent the interactive map and the info panels. There are three different info panels. One is used for showing information for a professor, one for the classrooms and one for the labs.

Figure 5.18. Info Panels

The associated ActionScript class is called TUC_InteractiveMap. The class first declares the movie clips that have to be displayed dinamically.

```
/* Levels */
public var l0:Level0 = new Level0();     // Level 0
public var l1:Level1 = new Level1();     // Level 1
public var l2:Level2 = new Level2();     // Level 1

/* Info Panels */
public var OfficeIP:OfficeInfoPanel = new OfficeInfoPanel();
public var ClassroomIP:ClassroomInfoPanel = new ClassroomInfoPanel();
public var LabIP:LabInfoPanel = new LabInfoPanel();
```

In the class constructor, three functions are called. These function have initializing character and are used for populating the scene with the maps and info panels to set their location inside the scene and last to hide them. The further showing and hiding is implemented in Unrealscript.

```
public function TUC_InteractiveMap() {
      PopulateMap();
      SetLocation();
      HideMovieClips();
}
```

The ActionScript built-in function `addChild` allows adding movie clips to the scene. It is used in our PopulateMap function to add the maps and the info panels to the scene as follows:

```
/** Populate the map with diff movie clips */
function PopulateMap() {
      /* attach Levels */
      addChild(l0);
      addChild(l1);
      addChild(l2);

      /* attach info panels */
      addChild(OfficeIP);
      addChild(ClassroomIP);
      addChild(LabIP);
      addChild(ProfDeatailedInfoMC);
}
```

In the function SetLocatation, the location of the panels is defined using their x and y coordinates.

```
/** Set the location of the movie clips */
```

```
function SetLocation() {
	/*
	* Set Levels
	*/
	/* Level 0 */
	l0.x = LEVEL_XCOORD;
	l0.y = LEVEL_YCOORD;
	/* Level 1 */
	l1.x = LEVEL_XCOORD;
	l1.y = LEVEL_YCOORD;
	/* Level 2 */
	l2.x = LEVEL_XCOORD;
	l2.y = LEVEL_YCOORD;

	/*
	 * Set Info Panels
	 */
	OfficeIP.x = INFOPANEL_XCOORD;
	OfficeIP.y = INFOPANEL_YCOORD;

	ClassroomIP.x = INFOPANEL_XCOORD;
	ClassroomIP.y = INFOPANEL_YCOORD;

	LabIP.x = INFOPANEL_XCOORD;
	LabIP.y = INFOPANEL_YCOORD;

	ProfDeatailedInfoMC.x = 846.9;
	ProfDeatailedInfoMC.y = 455.25;
}
```

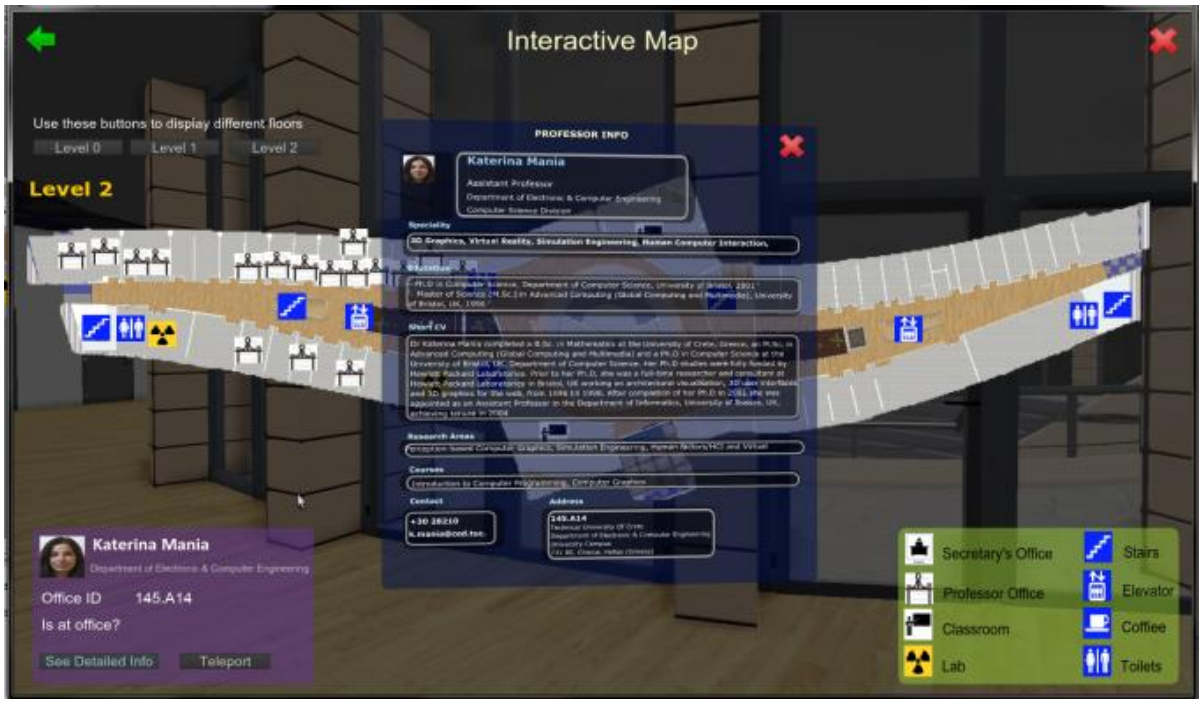
Next, all of the movie clips are hidden by setting their visible property to false.

```
/**
 * Hide the movie clips
 * we will display them in UnrealScript
 */
function HideMovieClips() {
    l0.visible = false;
    l1.visible = false;
    l2.visible = false;
    OfficeIP.visible = false;
    ClassroomIP.visible = false;
    LabIP.visible = false;
    ProfDeatailedInfoMC.visible = false;
}
```

On the top of the map that is currently displayed, there are placed some 2D graphics. The blue ones are only images that show information on the location of the coffee shops, the stairs, the lifts and the toilets. The other graphics are not only

images but buttons at the same time. These buttons are used for showing information about the professors' office, the labs and classrooms, in a panel placed at the right bottom corner (the purple panels).

The professors' office panel shows the name of the professor, his/her department and the office number. The panel also includes a button for showing additional information for the relevant professor, and a button to teleport the user directly to the professor's office.



The classroom panel shows the name of the classroom, the number of the seats and a teleport button.

The Lab panel shows the name of the lab, the staff and a teleport button.

The embedding of the flash file, the population of the info panels, as well as their showing and hiding, is implemented in the associated UnrealScript class. The class declaration is as follows:

```
class TUC_FrontEnd_InteractiveMap extends TUC_FrontEnd
      dependson(TUCPlayerController);
```

This class is quite large, and for that reason, only the most important lines of code are shown in this text. It can be seen from the class declaration that the class extends the `TUC_FrontEnd` class because it is part of the menu. The `dependson`

modifier tells the compiler to process the class in the parenthesis first, because this class depends on an enum or struct declared in that other class. In our case the `TUC_FrontEnd_InteractiveMap` class depends on `TUCPlayerController` class. That is done because the data for the professors, the labs and the classrooms is declared in a variable found in the `TUCPlayerController` class.

After instanciating this class the `Init` function has to be called. This function initializes and starts playing the movie. Moreover, it takes references to all the movie clips that have to be accessed from Unrealscript. A reference to the data structure that holds data for professors, labs and classrooms is also captured here.

```
function Init(optional LocalPlayer player)
{
	Start();
	Advance(0.f);

	RootMC = GetVariableObject("_root");
	CursorMC = RootMC.GetObject("cursor");
	SetCursorPosition();

	Level0 = RootMC.GetObject("l0");		// reference to the level 0
	Level1 = RootMC.GetObject("l1");		// reference to the level 1
	Level2 = RootMC.GetObject("l2");		// reference to the level 2

	OfficeInfoPanel = RootMC.GetObject("OfficeIP");
	LabInfoPanel = RootMC.GetObject("LabIP");
	ClassroomInfoPanel = RootMC.GetObject("ClassroomIP");

	InitProfDetailedInfo();

	Info = TUCPlayerController( GetPC() ).Info;
}
```

The functions used for populating and displaying panels with data for professors, labs and classrooms work in a similar manner, so therefore only the function for displaying the professor info panel is presented here.

It was seen above that all of the office buttons were assigned the same event listener function. This function is called `ShowProfessorInfoPanel` and it is fired every time one of the office buttons is pressed. In Flash those buttons were given instance name using professor names. The `ShowProfessorInfoPanel` function first displays the panel for the professors and it populates it with data which is loaded in structure, in an instance of the `Info` class.

```
function ShowProfessorInfoPanel(EventData data)
{
	local GFxObject Button;
```

```
        local string ButtonName;
        local string ProfID;

        ShowInfoPanel("Professor");

        Button = data._this.GetObject("target");
        ButtonName = Button.GetString("name");

        ProfID = GetRightMost(ButtonName);

        PData = Info.FindProfessor(ProfID);
        OfficeInfoPanel.GetObject("ProfInfoPanel_ImageHolder").AttachMovie(Prof
ID, "KM");      // attach prof image
        OfficeInfoPanel.GetObject("ProfInfoPanel_ProfessorNameLabel").SetText(P
Data.PName);    // display prof name
        OfficeInfoPanel.GetObject("ProfInfoPanel_OfficeIDLabel").SetText(PData.
OfficeID);     // display office id
}
```

When the user presses the teleport button, the program calls the `PTeleport` function. The actual teleporting of the user is done from the `Controller` class because from there we have reference to the `Pawn` - the body of the user in the VE. The teleporting basically consists of changing the user's location. The new location is retreived from the `PData` structure.

```
function PTeleport(EventData data)
{
    TUCPlayerController(GetPC()).Teleport(PData.Location);
    CloseInteractiveMap();
}
```

5.10 Reading Sensors Data From a Web Server

Data from a Temperature Sensor and a Switch Sensor is directly streamed in the application at runtime. The sensor is used for measuring current room temperature and the data from the switch is used as it was sensor for detecting presence - if a professor is at his/her office. The set up of these sensors, as well as sending their data to a web server, was done by the Telecommunication lab of TUC (should i include names- no names, good as it is). The data sent to the web server, as it can be viewed from a browser, is shown in the figure below:

Figure 5.19. The data from the sensors

The data from the sensors is sent to Arduino Ethernet is a microcontroller. Arduino can sense the environment by receiving input from a variety of sensors and can affect its surroundings by controlling lights, motors, and other actuators. It has 14 digital input/output pins, 6 analog inputs, a 16 MHz crystal oscillator, a RJ45 connection, a power jack, an ICSP header, and a reset button.

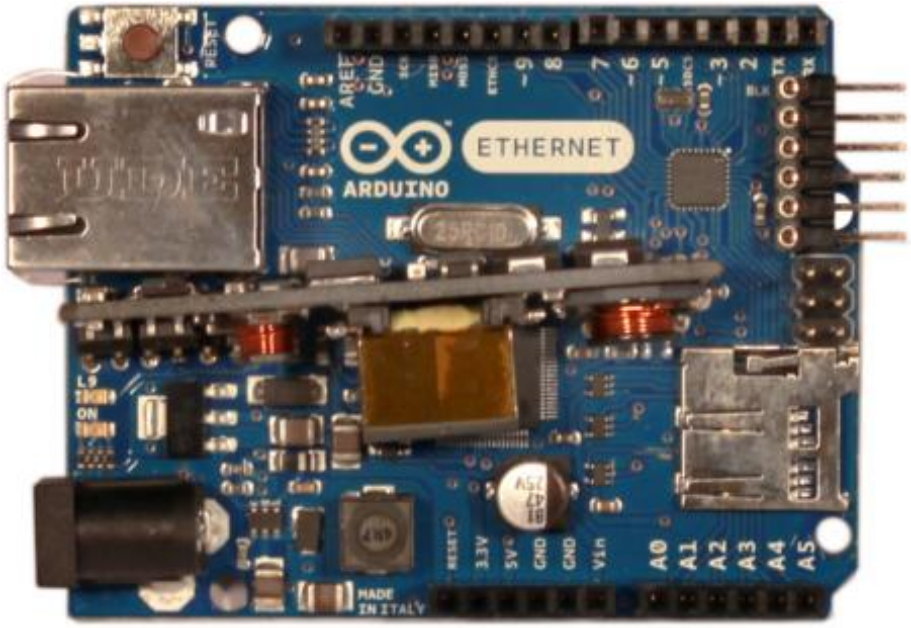


Figure 5.20. Arduino Ethernet board front view with optional PoE module

The Arduino Ethernet has a number of facilities for communicating with a computer, another Arduino, or other microcontrollers. A SoftwareSerial library allows for serial communication on any of the Uno's digital pins. Eventually, the data from the sensors is written to web server from where it is retreived.

The data from the temperature sensor is presented on display panels which are placed in front of the classroom entrances. In view of the demonstrative character of this functionality, the same readings are presented to all of the display panels. However, the other information presented on the panel is different for every panel and it is loaded and presented dynamically.

Figure 5.21. Display Panel: deactivated (left), activated(right)

The data from the switches is presented in three different ways. First, in the display panels, the data is used for showing if the classroom is occupied or not. Second, the data is used in the Interactive Map functionality that is described above. In the Interactive Map, specifically in the Office Info panel, the data from the switch is used for telling if a professor is at his/her office at the moment. Third, when a professor is at his/her office, a green light which is placed in front of the office is turned on.

5.10.1 Display Panels

The display panels are panels that are placed in front of the classroom entrances and are used for presenting information about the classroom, such as its id, number of seats, occupancy status and room temperature. The UI is implemented in Flash as all other interfaces. However, the difference is that it is not displayed on the user's screen, but instead it is displayed on a texture placed on a static mesh placed in the VE. A display panel is shown in Figure. The classroom id and the number of seats are loaded from an external configuration file. The data for the temperature and the occupancy is sent from sensors to a web server and then streamed dynamically in to the application.

The wrapper class for the swf file representing the UI is described next.

TUC_FrontEnd_SensorPanel
As allways this class is an extension of `GFxMoviePlayer`:

```
class TUC_FrontEnd_SensorPanel extends GFxMoviePlayer;
```

The variables that hold references to the UI's text labels that need to be populated are declared as:

```
var GFxObject RootMC, Title, NumOfSeats, IsItOcupied, CurrentTemperature;
```

Reference to the classroom loaded from a configuration file:

```
var TUC_InfoData.Classroom CData;
```

The Init function has to be called after instance of this class is created. This function initializes the movie, gets the movie started, and it also captures the references to the components of the swf file.

```
function Init(optional LocalPlayer player)
{
	Start();
	Advance(0.f);
	SetViewScaleMode(SM_ExactFit);

	RootMC = GetVariableObject("_root");

	/* References to the texfields */
	Title = RootMC.GetObject("title");
	NumOfSeats = RootMC.GetObject("numOfSeats");
```

```
	IsItOcupied = RootMC.GetObject("isItOcupied");
	CurrentTemperature = RootMC.GetObject("currentTemperature");
}
```

The UI's textfields are populated using the below setter functions:

```
function SetTitle(string Tag)
{
	Title.SetText("Classroom: "$Tag);
}
```

The information about the number of seats is taken from the info object found in our Player Controller class. The info object contains all the information about classrooms, professors and labs.

```
function SetNumOfSeats(String Tag)
{
	CData = TUCPlayerController( GetPC() ).Info.FindClassroom(Tag);
	NumOfSeats.SetText(CData.NumOfSeats);
}
```

The data streamed from the web server is held in an instance of the TUC_TCPLink class declared in the `PlayerController` class. This data is accessed by casting the player controller which is returned from the `GetPC` function to our `TUCPlayerController` and. The next two classes show how it is done:

```
function SetIsItOcupied()
{
	if( TUCPlayerController(GetPC()).newLink.Door == "o" )
		IsItOcupied.SetText("Yes");
	else
		IsItOcupied.SetText("No");
}

function SetCurrentTemperature()
{
	CurrentTemperature.SetText(TUCPlayerController(GetPC()).newLink.Current
                                                                Temperature);
}
```

In the default properties of the class reference to the associated swf file is taken, as well as reference to the texture used for projecting the movie.

```
DefaultProperties
{
	RenderTexture=TextureRenderTarget2D'TUC_SensorPanel.RT_SensorPanel'
	MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_SensorPanel'
}
```

TUC_SensorPanel

This class defines the static mesh used as a monitor on which the data is presented, as well as a `Trigger` that is used for activating and deactivating the display panel. In order to sense presence, the display panel is a subclass of `Trigger`.

```
class TUC_SensorPanel extends Trigger
      placeable
      ClassGroup(TUC);
```

The reference to the static mesh component used as display panel (modified in the editor) is declared as:

```
var() StaticMeshComponent Panel;
```

The variable below is used for creating the movie - the interface.

```
var TUC_FrontEnd_SensorPanel SensorPanelMovie;
```

When the user enters the proximity of the display panel, the `Touch` event is fired. What is done here is that the material of the panel is changed to the material (`M_SensorPanel`) used for projecting the movie. The material is created in the UDK's Content Browser and it uses `RenderTexture` as basic texture. It was seen above that the `Movie` is projected to exactly that texture. Consequently, after applying the material to the panel, the movie will be projected on the panel and not on the screen. The panel has no material by default (actually, it has assigned black material in the material editor. After that, the movie is created and started by calling the `OpenSensorPanelMovie` function.

```
event Touch(Actor Other, PrimitiveComponent OtherComp, Vector HitLocation,
Vector HitNormal)
{
    super.Touch(Other, OtherComp, HitLocation, HitNormal);

    if( Pawn(Other) != none )
    {
        Panel.SetMaterial(0, Material'TUC_SensorPanel.M_SensorPanel');
        OpenSensorPanelMovie( string(Tag) );

    }
}
```

When the user leaves the Trigger's area, the `Untouch` event is fired. The code of this event calls the `CloseSensorPanelMovie` used for closing the movie and then changes the material of the panel. The new material is set to none, and this will cause to use its default black color material.

```
event Untouch(Actor Other)
{
```

```
    super.Untouch(Other);

    if( Pawn(Other) != none )
    {
        CloseSensorPanelMovie();
        Panel.SetMaterial(0, none);
    }
}
```

Instances of the UI movie are created in the following function:

```
function OpenSensorPanelMovie(string ClassroomID)
{
      SensorPanelMovie = new class'TUC_FrontEnd_SensorPanel';
      SensorPanelMovie.Init();
      SensorPanelMovie.PopulateMovie(ClassroomID);
      SetTimer(1, true, 'SetSensorsData', SensorPanelMovie);
}
```

This function, except creating, it also initializes the new movie,then populates it and it sets a loop timer executed every second, which calls a function that populates the sensor textfields of the UI.

The movie is closed by the following function:

```
function CloseSensorPanelMovie()
{
      SensorPanelMovie.Close();
      SensorPanelMovie = none;
      ClearTimer('SetSensorsData', SensorPanelMovie);
}
```

The static mesh component is defined in the default properties.

```
DefaultProperties
{
      Components.Remove(Sprite)

      begin object class=StaticMeshComponent name=SM
            StaticMesh=StaticMesh'TUC_SensorPanel.SM.SM_DisplayPanel_01'
            Scale3D=(X=0.025,Y=0.25,Z=0.125)
      end object

      Components.add(SM)
      Panel=SM
      bHidden=false
}
```

5.10.2 Connecting to a web server

The data from the sensors is retreived from a web server. In order to connect to a web server, one can subclass the engine's TcpLink class which is used for establishing internet tcp/ip connection. The subclass that we wrote uses native functions to open and close a connection to a web server, and it also sends a http request message to the web server to read the data stored there.

```
class TUC_TCPLink extends TcpLink
       config(TUC);
```

This class constitutes most of event functions that are fired when some event, such as when established connection, occurs.

The domain name of the server or its ip address is held in the `Host` variable. The port is also needed by the class and it is held in the `HostPort`. These two variables are declared with the config modifier, which means that their values are set in a configuration file. That is done because we do not want to compile the project if their values have to be changed.

```
/** domain name or ip */
var config string Host;

/** remote port */
var config int HostPort;
```

The http requests that are sent to the web server contain a lot newline ('\r''\n' = carriage return + line feed) characters. The newline character is assigned to the CRLF variable.

```
/** newline: carriage return + line feed */
var string CRLF;
```

The data from the sensors is stored in the following variables:

```
/** data from the temperature sensor */
var string CurrentTemperature;

/** data from the switch sensor */
var string Door;
```

When a new instance of this class is created, the `PostBeginPlay` event is fired. In this event, the newline is defined and the receiving mode is set to `MODE_Line`. In `MODE_Line` the `ReceivedLine` event is executed. The line argument contains a single line, without the line terminator. The other two possible modes are `MODE_Text` and `MODE_Binary`.

```
event PostBeginPlay()
{
    super.PostBeginPlay();
    CRLF = chr(13)$chr(10);
    LinkMode = MODE_Line;
}
```

After an object of this class is created, in order to connect to the web server, the `Connect` function has to be called.

```
function Connect()
{
    // resolve the hostname
    Resolve(Host);
}
```

The `Resolve` function is native function that resolves a domain name or dotted ip. It is a nonblocking operation and triggers the `Resolved` event if the resolving is successful otherwise the `ResolveFail` event.

Upon success, the host port is set and tries are given to open connection using the ip address that is created for the specified host.

```
event Resolved( IpAddr Addr )
{
    Addr.Port = HostPort;
    BindPort();

    Open(Addr);
}
```

When the connection opens, the `Opened` event is triggered. In this event a super simple http request is written to communicate with the server. The request message consists of a request line using GET method. The GET method retreives whatever information is identified by the Resource path. The domain name of the server is defined in the Host header. The last command is to close the connection.

```
event Opened()
{
    // The HTTP request
    SendText("GET / HTTP/1.1"$CRLF);
    SendText("Host: "$Host$CRLF);
    SendText("Connection: Close"$CRLF);
    SendText(CRLF);     // end of request
}
```

When a response message is sent by the server, the `ReceivedLine` event is fired. The string parameter contains a line of text from which the newline character is removed. For every http request that is sent, 10 lines of text which constitute the response, are received. The first 7 lines and the last one are html tags, therefore they can be ignored. The 8th line contains the data from the switch sensor and it goes like this: Door = open
. It can be observed that by using the 7th character only it can be determined if the door is open or closed. The `Mid` was used to extract this character. The `Mid` function generates a substring of S by starting at character i and copying j characters. Using this function, the temperature from the 8th line is also extracted.

```
event ReceivedLine(string L)
{

	if(counter == 8)
	{
		Door = Mid(L, 7, 1);
		ToggleLight(Door);
	}
	else if(counter == 9)
	{
		CurrentTemperature = Mid(L, 15, 4);
	}
	else if(counter >10)
	{
		counter = 1;
	}


	counter++;
}

function ToggleLight(String s)
{
	local TUC_PresenceDetectionLight l;

	foreach AllActors( class'TUC_PresenceDetectionLight', l )
	{
		if(s == "o")
			l.TurnOn();
		else
			l.TurnOn();
	}
}

defaultproperties
{
```

```
        counter=1
}
```

6 Conclusions and Future Work

Since the first game was published in 1962, video games have become so popular that over the past 30 years, they have become an integral part of our culture, and eventually led to more sophisticated technology for their further development. Their popularity was a major cause for continuously increasing budgets in that sector and consequently for the development of sophisticated technology for creating these games. This has resulted in game engines that support development of real-time rendering, large virtual environments, global illumination, dynamic shadows, dynamic lighting, sophisticated audio and networking.

This thesis presents an interactive 3D application which is developed using the state-of-art game engine, namely, Unreal Development Kit (UDK). The application simulates a real-time large, smart and multi-purpose virtual environment (the science building of TUC). The VE's geometry is created using the UDK's native geometry objects and static meshes that was created externally with the help of the modeling tool Autodesk 3DS Max. The most important factor in the final appearance of the 3D virtual environment includes the materials created in UDK. These materials include textures that are created from the photographs are taken and also downloaded from texture repositories. All of the functionality that is created was implemented using the scripting language of UDK - the Unrealscript. The provided features are: path finding, mini map, reading data from external sensors via tcp link, interactive map and many more. The User Interfaces (UIs) are created using the Scaleform GFx technology. This technology allows importing Flash-based interfaces and rendering them on the screen or on a texture inside UDK. The scripting language for the flash applications used was ActionScript 3.0. The created application simulates a real-time 3D navigable environment which the user can interact with, navigate in it and see information for professors, labs or classrooms. Moreover, the user can see data from external sensors. This thesis explored a method of 3D architectural visualization based on a game engine, including dynamically updated information derived from the real-environment streamed in the 3D simulation.

6.1 Implications for Future Work

Some possible future extensions and improvements of this application may be the following:

- Interactive Virtual Environments on mobile platforms: Applications for mobile phones are widly used today. The Virtual Environment itself is very complex and heavy, so its geometry has to be simplifyed in order to be used on mobiles. However, Unrealscript provides a large set of classes that allows the development of mobile applications.

- Adding many sensors will allow building interesting applications. Audiovisual applications on top of the data collected from real-time sensors could be bult.

- Connecting the application with a Database: Data is often more important than programs and threrefore the safe, correct, and efficient management of those data is of crucial importance. The data stored by this application at the moment does not require such systems, but in the future, if more data is stored, for example, if many sensors are added, the way to organize the data would benefit by a database.

- Gameplay: An interesting extension of this application would be adding some gameplay. This application as is, allows adding First Person Shooter (FPS) gameplay easily.

7 Bibliography - References

Richard J. Moore, 2011 - Unreal Developement Kit 3: Beginner's Guide

Jason B, Zak P Jeff W, 2009 - Mastering Unreal Technology, Volume I: Introduction to Level Design with Unreal Engine 3

Jason B, Zak P Jeff W, 2009 - Mastering Unreal Technology, Volume II: Advanced Level Design Concepts with Unreal Engine 3

Rachel Cordone, 2011 - Unreal Development Kit Game Programming with UnrealScript: Beginner's Guide

Thomas Mooney, 2012 - Unreal Development Kit Game Design Cookbook

Tim Sweeney, 1998 - UnrealScript Language Reference

Texture Mapping - http://en.wikipedia.org/wiki/Texture_mapping

Unreal Engine - http://www.unrealengine.com/

Unreal Development Kit - http://www.unrealengine.com/udk/

Unreal Developer Network - http://udn.epicgames.com/Main/WebHome.html

UDK Community Links - udn.epicgames.com/Three/UDKCommunityLinks.html

UDK Forum - http://forums.epicgames.com/forums/366-UDK

3D Buzz - http://www.3dbuzz.com/Training

Hourences - http://www.hourences.com/tutorials/

Chris Albeluhn - http://www.chrisalbeluhn.com/3D_Tutorials.html

UnrealScript - http://www.moug-portfolio.info/

UncodeX - http://uncodex.elmuerte.com/

MS Visual Studio - http://www.microsoft.com/visualstudio/eng/team-foundation-service

Adobe Flash Professional - http://www.adobe.com/products/flash.html

Scaleform - http://gameware.autodesk.com/scaleform

Scaleform - http://udn.epicgames.com/Three/Scaleform.html

Settings Menu Page - http://www.mavrikgames.com/tutorials/udk-scaleform-menu

ActionScript 1:1 with Doug Winnie - http://tv.adobe.com/show/actionscript-11-with-doug-winnie/

Coordinate system in Flash - http://flashexplained.com/basics/understanding-how-the-coordinate-system-in-flash-works/

Scaling in AutoCAD - http://www.youtube.com/watch?v=vXZxKjpyupU

Loft Mapping - http://www.cadtutor.net/tutorials/3ds-max/loft-mapping.php

3DS Max UVW -www.youtube.com/watch?v=ZlZgH3O_vxw&feature=related

Textures Repository - http://www.cgtextures.com/

Normal Map Creation -

http://www.game-artist.net/forums/spotlight-articles/43-tutorial-introduction-normal-mapping.htmlhttps://developer.valvesoftware.com/wiki/Normal_Map_Creation_in_Photoshop_or_Paint_Shop_Pro

Christodoulou Ioannis, 2011 - An Interactive 3D Lighting System for fMRI Rendering Fidelity Experiments

8 Appendix A

8.1 Settings

This is the functionality that allows the user to change the resolution and the anti-aliasing settings of the application. The section below includes description of both the Scaleform/Flash side of the application and the Unrealscript side.

Flash Side
The Settings menu page is fairly simple – it contains two option steppers to change the resolution and the anti-aliasing settings, and two simple buttons, one of which to save the changes and one to cancel the current operation and return to the main menu. First, the file that represents the page and holds the buttons is created. It is named `TUC_SettingsMenu`. In the second place, three layers are created in the timeline: one layer that holds the page background, one for the buttons and one for the ActionScript timeline code named Actions. Then, we started laying out the `Scaleform CLIK` components. The default button from the shared library is used for the `Cancel` and `Save` buttons. The labels and the option steppers are imported from the CLIK component library. All the components are scaled a little. The buttons were given instance name because we want to access them in UnrealScript.

What follows is a description of the ActionScript side of the implementation and then the UnrealScript side.

ActionScript
The ActionScript class that is associated with the above flash file is `TUC_SettingsMenu` class. This class, as all the other Menu related classes, extends `TUC_FrontEnd`. The `DataProvider` class is imported at the beginning of the class.

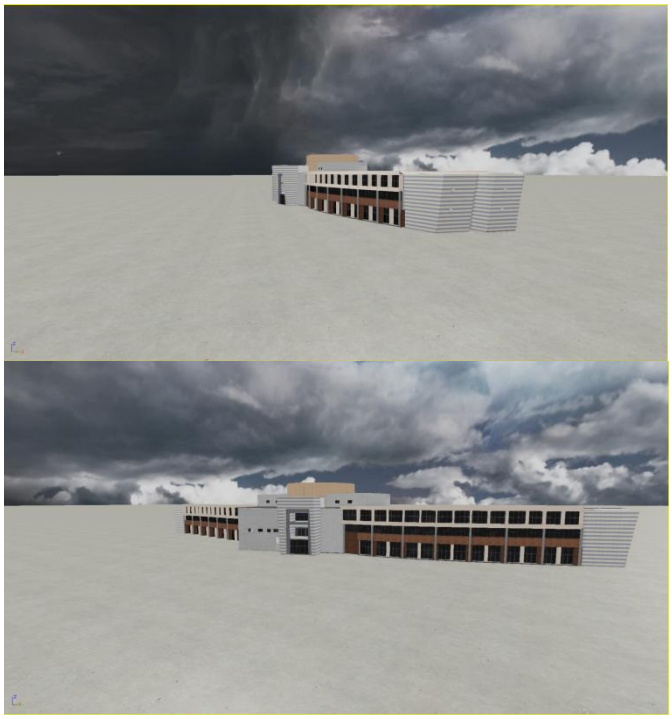
The data provider is basically an array of options that the user are going to want to step through. In this case, it will be an array of common video resolutions and anti-aliasing multipliers. In the class constructor after calling the super(), we populate the two the correct data.

```
package  {

	import scaleform.clik.data.DataProvider;

	public class TUC_SettingsMenu extends TUC_FrontEnd {

		public function TUC_SettingsMenu() {
			super();
			aliasingStepper.dataProvider = new DataProvider(["None",
"2x", "4x", "8x", "16x"]);
			resolutionStepper.dataProvider = new
DataProvider(["800x600", "1024x768", "1152x864", "1280x720", "1280x1024",
"1600x1200", "1680x1050", "1920x1080"]);
		}
	}
}



/**
 * @class TUC_FrontEnd_SettingsMenu
 * @author
 */
class TUC_FrontEnd_SettingsMenu extends GFxMoviePlayer config(TUC);

var config int ResolutionIndex, AliasingIndex;

/** reference to the root object of the flash file */
var GFxObject RootMC;

var GFxClikWidget ResolutionStepper, AliasingStepper, BtnBack,
BtnExitMainMenu, BtnSave;

/** the next menu page */
var GFxMoviePlayer NextMenuPage;

function Init(optional LocalPlayer player)
{
    Start();
    Advance(0);
	SetViewScaleMode(SM_ExactFit);

	RootMC = GetVariableObject("_root");
}

event bool WidgetInitialized(name WidgetName, name WidgetPath, GFxObject
Widget)
{
```

```
	local bool bWasHandled;

	bWasHandled = false;

	switch(WidgetName)
	{
		case ('resolutionStepper'):
			`log("Resolution Stepper Option");
			ResolutionStepper = GFxClikWidget(Widget);
			ResolutionStepper.SetInt("selectedIndex", ResolutionIndex);
			bWasHandled = true;
			break;
		case ('aliasingStepper'):
			AliasingStepper = GFxClikWidget(Widget);
			AliasingStepper.SetInt("selectedIndex", AliasingIndex);
			bWasHandled = true;
			break;
		case ('btnBack'):
			BtnBack = GFxClikWidget(Widget);
			BtnBack.AddEventListener('CLIK_click', Back);
			bWasHandled = true;
			break;
		case ('btnExitMainMenu'):
			BtnExitMainMenu = GFxClikWidget(Widget);
			BtnExitMainMenu.AddEventListener('CLIK_click',
ExitMainMenu);
			bWasHandled = true;
			break;
		case ('btnSave'):
			BtnSave = GFxClikWidget(Widget);
			BtnSave.AddEventListener('CLIK_click', SaveSettings);
			bWasHandled = true;
			break;
		default:
			break;
	}
	return bWasHandled;
}

function Back(EventData data)
{
	NextMenuPage = new class'TUC_FrontEnd_MainMenu';
	NextMenuPage.Init();
	Close(false);
}

function ExitMainMenu(EventData data)
{
	Close();
	TUCPlayerController(GetPC()).MenuManager();	// PC will handle
}

function SaveSettings(EventData data)
{
	local int ASI, RSI;

	ASI = AliasingStepper.GetInt("selectedIndex");
```

```
	RSI = ResolutionStepper.GetInt("selectedIndex");

	if (AliasingIndex != ASI)
	{
		AliasingIndex = ASI;
		ConsoleCommand("Scale Set MaxMultiSamples " $
AliasingStepper.GetString("selectedItem"));
	}

	if (ResolutionIndex != RSI)
	{
		ResolutionIndex = RSI;
		ConsoleCommand("Setres " $
ResolutionStepper.GetString("selectedItem") $ "w");
	}

	SaveConfig();

	NextMenuPage = new class'TUC_FrontEnd_MainMenu';
	NextMenuPage.Init();
	Close(false);
}

DefaultProperties
{
	MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_SettingsMenu'

	WidgetBindings.Add((WidgetName="resolutionStepper",WidgetClass=class'GF
xCLIKWidget'))
	WidgetBindings.Add((WidgetName="aliasingStepper",WidgetClass=class'GFxC
LIKWidget'))
	WidgetBindings.Add((WidgetName="btnSave",WidgetClass=class'GFxCLIKWidge
t'))
	WidgetBindings.Add((WidgetName="btnBack",WidgetClass=class'GFxCLIKWidge
t'))
	WidgetBindings.Add((WidgetName="btnExitMainMenu",WidgetClass=class'GFxC
LIKWidget'))

	bShowHardwareMouseCursor=true

	TimingMode=TM_Real
	bPauseGameWhileActive=true
	bCaptureInput=true
	bDisplayWithHudOff=true
	bIgnoreMouseInput=false
}
```

8.2 Cinematics



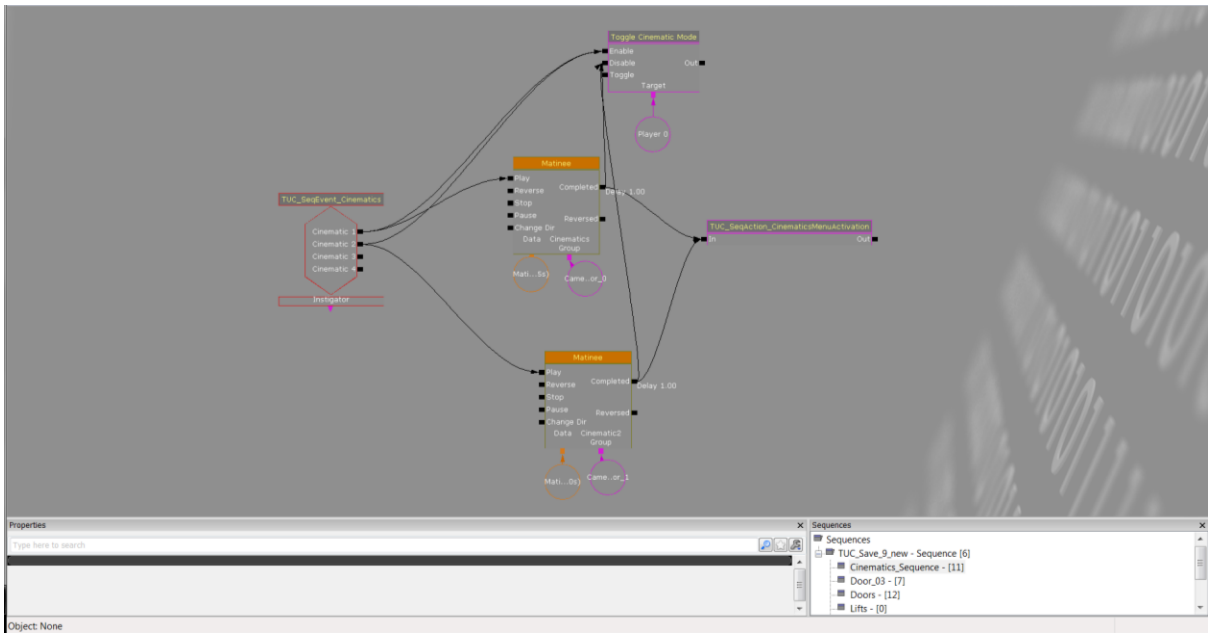
```
class TUC_FrontEnd_Cinematics extends GFxMoviePlayer;
```

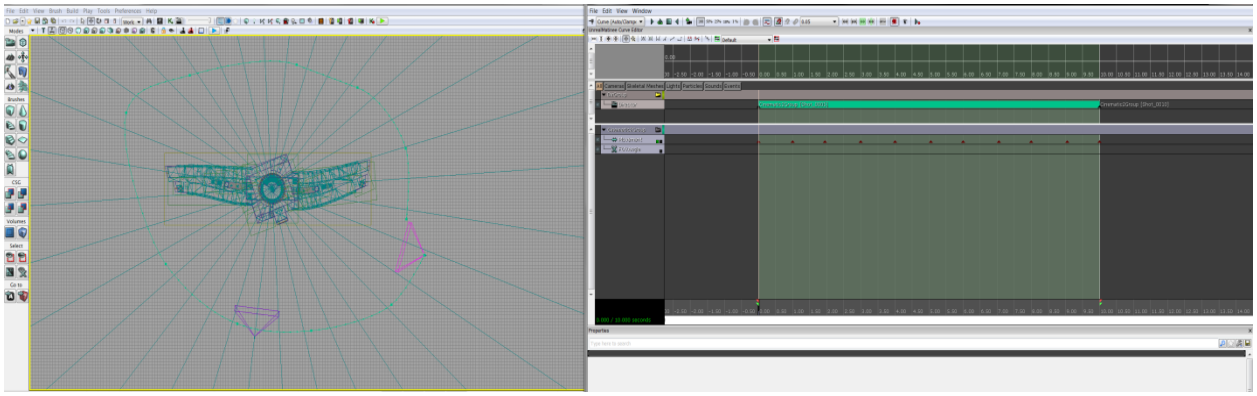
```
var GFxClikWidget Cinematic1, Cinematic2, Cinematic3, Cinematic4, BtnBack,
BtnExitMainMenu;

/** the next menu page */
var GFxMoviePlayer NextMenuPage;

function Init(optional LocalPlayer player)
{
	Start();
	Advance(0.f);
	SetViewScaleMode(SM_ExactFit);
}

event bool WidgetInitialized(name WidgetName, name WidgetPath, GFxObject
Widget)
{
	local bool bWasHandled;

	bWasHandled = false;

	switch(WidgetName)
	{
		case ('Cinematic1'):
			Cinematic1 = GFxClikWidget(Widget);
			Cinematic1.AddEventListener('CLIK_click',
ActivateCinematic1);
			bWasHandled = true;
			break;
		case ('Cinematic2'):
			Cinematic2 = GFxClikWidget(Widget);
			Cinematic2.AddEventListener('CLIK_click',
ActivateCinematic2);
			bWasHandled = true;
			break;
		case ('Cinematic3'):
			Cinematic3 = GFxClikWidget(Widget);
			Cinematic3.AddEventListener('CLIK_click',
ActivateCinematic3);
			bWasHandled = true;
			break;
		case ('Cinematic4'):
			Cinematic4 = GFxClikWidget(Widget);
			Cinematic4.AddEventListener('CLIK_click',
ActivateCinematic4);
			bWasHandled = true;
			break;
		case ('btnBack'):
			BtnBack = GFxClikWidget(Widget);
			BtnBack.AddEventListener('CLIK_click', Back);
```

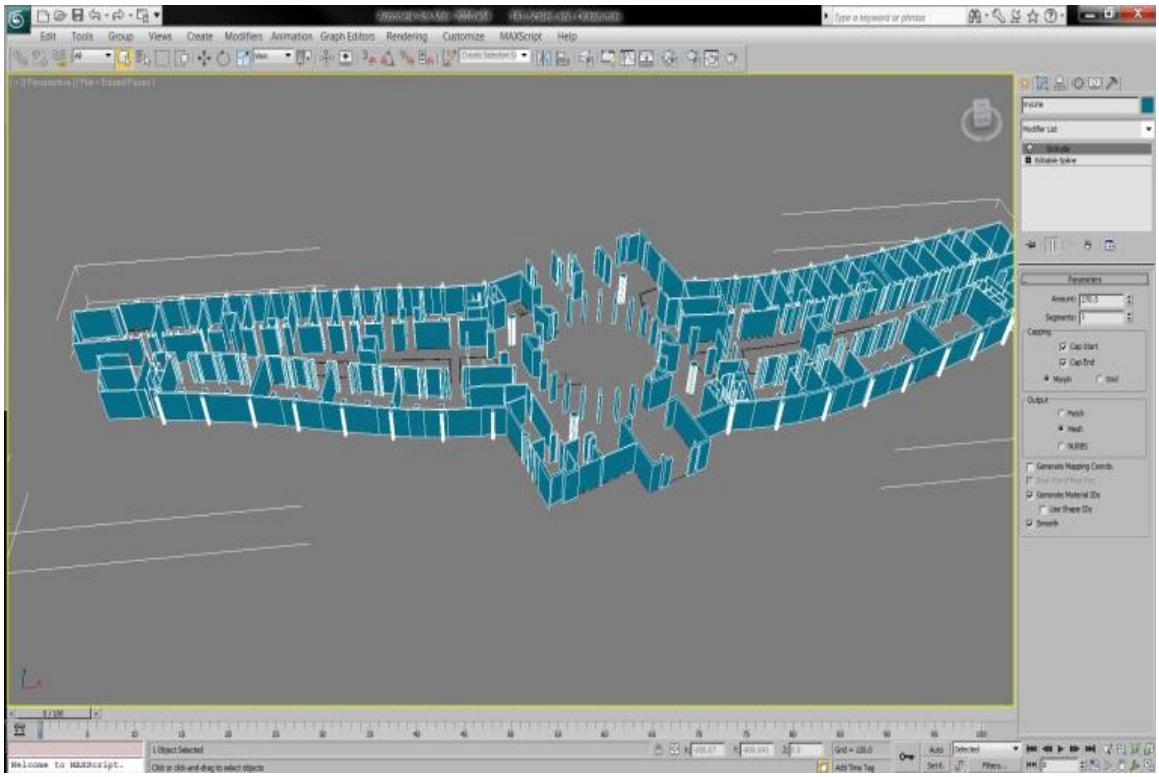
```
                  bWasHandled = true;
                  break;
            case ('btnExitMainMenu'):
                  BtnExitMainMenu = GFxClikWidget(Widget);
                  BtnExitMainMenu.AddEventListener('CLIK_click',
ExitMainMenu);
                  bWasHandled = true;
                  break;
            default:
                  break;
      }
      return bWasHandled;
}

function ActivateCinematic1(EventData e)
{
      `log("[TUC_FrontEnd_Cinematics] - ActivateCinematic1");
      Close(true);
      TUCInfo(GetPC().WorldInfo.Game).TriggerGlobalEventClass(class'TUC_SeqEv
ent_Cinematics', GetPC(), 0);
}

function ActivateCinematic2(EventData e)
{    `log("[TUC_FrontEnd_Cinematics] - ActivateCinematic2");
      Close(true);

TUCInfo(GetPC().WorldInfo.Game).TriggerGlobalEventClass(class'TUC_SeqEvent_Ci
nematics', GetPC(), 1);
}
function ActivateCinematic3(EventData e)
{    `log("[TUC_FrontEnd_Cinematics] - ActivateCinematic3");
      Close();

TUCInfo(GetPC().WorldInfo.Game).TriggerGlobalEventClass(class'TUC_SeqEvent_Ci
nematics', GetPC(), 2);
}
function ActivateCinematic4(EventData e)
{
      `log("[TUC_FrontEnd_Cinematics] - ActivateCinematic4");
      Close();

TUCInfo(GetPC().WorldInfo.Game).TriggerGlobalEventClass(class'TUC_SeqEvent_Ci
nematics', GetPC(), 3);
}

function Back(EventData data)
{
      NextMenuPage = new class'TUC_FrontEnd_MainMenu';
      NextMenuPage.Init();
```

```
	Close(false);
}

function ExitMainMenu(EventData data)
{
	Close();
	TUCPlayerController(GetPC()).MenuManager();		// PC will handle
}


DefaultProperties
{
	MovieInfo=SwfMovie'TUCProjectFrontEnd.FrontEnd.TUC_Cinematics'

	WidgetBindings.Add((WidgetName="Cinematic1",WidgetClass=class'GFxCLIKWi
dget'))
	WidgetBindings.Add((WidgetName="Cinematic2",WidgetClass=class'GFxCLIKWi
dget'))
	WidgetBindings.Add((WidgetName="Cinematic3",WidgetClass=class'GFxCLIKWi
dget'))
	WidgetBindings.Add((WidgetName="Cinematic4",WidgetClass=class'GFxCLIKWi
dget'))
	WidgetBindings.Add((WidgetName="btnBack",WidgetClass=class'GFxCLIKWidge
t'))
	WidgetBindings.Add((WidgetName="btnExitMainMenu",WidgetClass=class'GFxC
LIKWidget'))

	bShowHardwareMouseCursor=true

	TimingMode=TM_Real
	bPauseGameWhileActive=true
	bCaptureInput=true
	bDisplayWithHudOff=true
	bIgnoreMouseInput=false
}




class TUC_SeqEvent_Cinematics extends SequenceEvent;

event Activated()
{
	`log("***************ACTIVATED***************");
}

defaultproperties
{
	ObjName="TUC_SeqEvent_Cinematics"
```

```
	ObjCategory="TUCProject Events"

	OutputLinks[0]=(LinkDesc="Cinematic 1")
	OutputLinks[1]=(LinkDesc="Cinematic 2")
	OutputLinks[2]=(LinkDesc="Cinematic 3")
	OutputLinks[3]=(LinkDesc="Cinematic 4")
}



class TUC_SeqAction_CinematicsMenuActivation extends SequenceAction;

var TUC_FrontEnd_Cinematics C;

event Activated()
{
	C = new class'TUC_FrontEnd_Cinematics';
	C.Init();
}

defaultproperties
{
	ObjName="TUC_SeqAction_CinematicsMenuActivation"
	ObjCategory="TUCProject Actions"
	VariableLinks.Empty
}
```






Edit
Tools
Group
Views
Create
Modifiers
Animation
Graph Editors
Rendering
Customize
MAXScript
Help
Welcome to MAXScript.