The Unreal Editor as a Web 3D Authoring Environment

David Arendash
Quantum Leap Computing

Abstract

Epic Games provides a free game level editor with titles based on its Unreal engine. The editor provides a rich set of authoring tools that can be used to create fully interactive environments. This paper describes a tool that converts Unreal levels to web-ready environments in VRML and X3D. The paper also examines the similarities between first-person-shooter games and web 3D worlds, and discusses the implications of having a low-cost, fully featured virtual world authoring environment available for creating web 3D content.

CR Categories: I.7.2 [VRML]; K.8.0 [Games]; K.8.1 [Freeware/Shareware] and [Graphics]; I.3.4 [Graphics Utilities]; I.3.5 [Computational Geometry and Object Modeling]; I.3.6 [Methodology and Techniques]; I.3.7 [Three-Dimensional Graphics and Realism]; I.3.8 [Applications]

Keywords: VRML, X3D, Unreal, game, tool, 3D, authoring

1 Introduction

VRML has been around since 1995, and most 3D authoring packages have some VRML export capability. VRML's latest incarnation is X3D, which updates the specification to more modern standards, including additional graphics features and an XML encoding. Since X3D supports a VRML data encoding and implements a superset of the VRML97 standard, henceforth in this paper the term 'X3D' will comprise VRML97 as well as X3D.

Tools that have X3D export capabilities cost hundreds, some even thousands of dollars, and do not necessarily allow for authoring of interactivity, sounds, etc. Generally, 3D authoring packages focus on object creation and texturing, with less (if any) ability for lighting, compositing, etc. Having geometry and texturing is great, but for real fun, we want to be able to make things move and interact, to set up lights and cameras. There aren't many open-standard 3D file formats which provide for all this. But X3D does! For the dedicated X3D author, hand-editing (and custom scripting) has been the only real method of embedding objects into a scene and bringing them to life. The difficulty of this process has contributed a widespread lack of realistic content. An inexpensive, full featured, easy to use authoring environment which can provide for all of this - with no need for technical expertise - has been unavailable to X3D authors, until now.

It is possible to output Unreal geometry as X3D geometry, generally as indexed face sets. X3D also allows for specification of lights, as does Unreal. X3D also supports sounds, so does Unreal. And X3D allows for proximity sensors and time sensors. Of course, so does Unreal.

What has been developed here is a stand-alone Windows application that converts the scene files exported from Unreal-based editors to X3D, with great fidelity of content, including interactivity.

2 Background

Unreal technology comes from Epic Games. The games in the Unreal line are generally played as FPS (First Person Shooters), wherein an avatar ('you') roam about spaces (which are sometimes called 'levels' or 'maps'), often in pursuit of things at which to shoot various virtual weapons.

UnrealEd

Since the release of the original 'Unreal' in 1997, Epic Games has generously included the Unreal Editor (UnrealEd) with all Unreal titles. UnrealEd is an IDE (Integrated Design Environment) anyone can use to create new game spaces for Unreal-based games. Unreal Tournament includes UnrealEd 2.0, and Unreal Tournament 2003 and Unreal 2 include UnrealEd 3.0. Other titles based on the Unreal technology also ship with editors (Wheel of Time, Rune, etc.). (An exception is America's Army, which, although using Unreal technology, does not include an editor).

Thanks to the availability of UnrealEd, the Unreal communities have produced many thousands of add-ons for Unreal games. These include not only new levels, or maps, but other 'mods' (modifications) which can actually alter the entire play of the game. These are all made possible by the Unreal Editor's integrated script editor, UnrealScript, that is a Java-like language.

UnrealEd's UI looks much like most 3D authoring tools, having (by default) 3 orthogonal views and an arbitrary perspective view. Each window can be configured separately. Viewing options include wire-frame, shaded, BSP-cuts, and dynamic lighting. The latter shows a good real-time approximation of the scene as it would look in the game, including lighting and sounds.

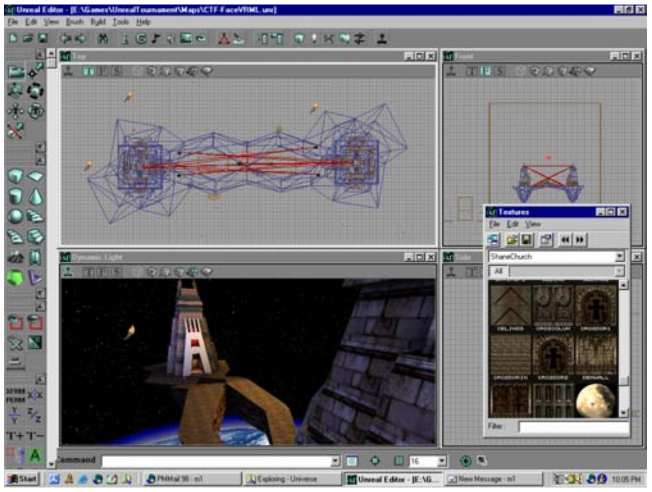


Figure 1. UnrealEd 2.0

All entities (objects, sounds, lights, etc) are called 'actors', as they are subclassed from the base class 'Actor'. Unreal levels are constructed using CSG (Constructive Solid Geometry). The Unreal universe is initially solid, so the first step in creating a level is to 'hollow out' or subtract a void into the world (this ensures no 'leaks' in the geometry). A host of primitives can be added, subtracted, intersected, unioned, and so forth to build up the world. Solid objects are referred to as 'Brushes', and brushes can be built up using the 'builder brush' and CSG operations. Bitmap, animation, and/or procedural textures can be applied to any surface (a procedural texture editor 'Fire Paint' is built-in). Lights of many types can be added, including a host of dynamic lighting effects. Sounds may also be added with a variety of behaviors, for ambient and/or positional audio. 'Triggers' of many types provide for proximity detection, timed events, event multiplexing, etc. Objects may be built as 'movers' who can then be assigned key-frame movement with parameterized behaviors. 'Meshes' (pre-defined object meshes) may be inserted as decorative objects. Fly-though camera positions may be supplied as well. A simple convention for connecting triggering events to other objects (movers, sounds, lights, other triggers, etc.) provides for complex interactivity. Using the integrated script editor, any actor may be subclassed to build out almost any kind of new actor an author can conceive.

UnrealEd builds game levels which are texture-baked, compiled binaries that the game engine can efficiently process when running the game. It also provides an import/export text format, T3D, which describes the level. This text format is fairly easy to parse, and contains almost all of the information needed to fully describe a game level (some texture info and custom scripts are stored separately).

4 The UnrealToX3D Converter

4.1 The Authoring Challenge

UnrealToX3D actually evolved from UnrealToOBJ. I had been experimenting with other authoring packages that imported Alias/Wavefront OBJ format. I was unimpressed with higher-end authoring tools, and found I could create objects much faster with UnrealEd. But with no direct export to OBJ that included all the texture info, I was stuck. I was already familiar with the T3D format, having worked on several utilities for importing 3DMF and building specialty objects into Unreal levels. It was easy enough to convert the solid geometry described in the T3D file to OBJ format. The real challenge was converting the texture coordinates. Unreal textures are described as tiling a plane in space coincident with the polygon on which they lie. (See Figure 2).

Each surface has a 3D vertex for U, another for V, and another for the texture normal (these are usually orthogonal). A pan value (in UV units) may also be supplied. This scheme allows for arbitrary scale, rotation, shear, and displacement of the texture without having to describe UV coordinates for each vertex on a polygon. To turn these various directionals into common UV coordinates, I find a transform that rotates the polygon from its arbitrary orientation onto the XY plane, and apply the same transform to the U and V texture components. I can now use the transformed U sand V component (which are now co-planar with the XY plane) to scale the texture. Using the pan components as offsets, I now use the polygon's coordinates (which are now all in the XY plane) to produce the UV coordinates.

The other challenge in conversion came with orientation. UnrealEd 1 and 2 use Euler angles (yaw, pitch, roll) to describe orientation. X3D uses quaternion-based orientation. That math was an entertaining exercise. UnrealEd 3.0 uses quaternions.

Having converted the geometry and textures, I found quite quickly that I had a need to add lights, cameras, and other elements to assemble a complete scene. There was really only one choice for an open-standard and widely accepted file format: X3D. That it also supports interactivity was a great bonus. I realized I now had the ability to design, light, and bring to life spaces, with a nearly-free full-featured design tool.

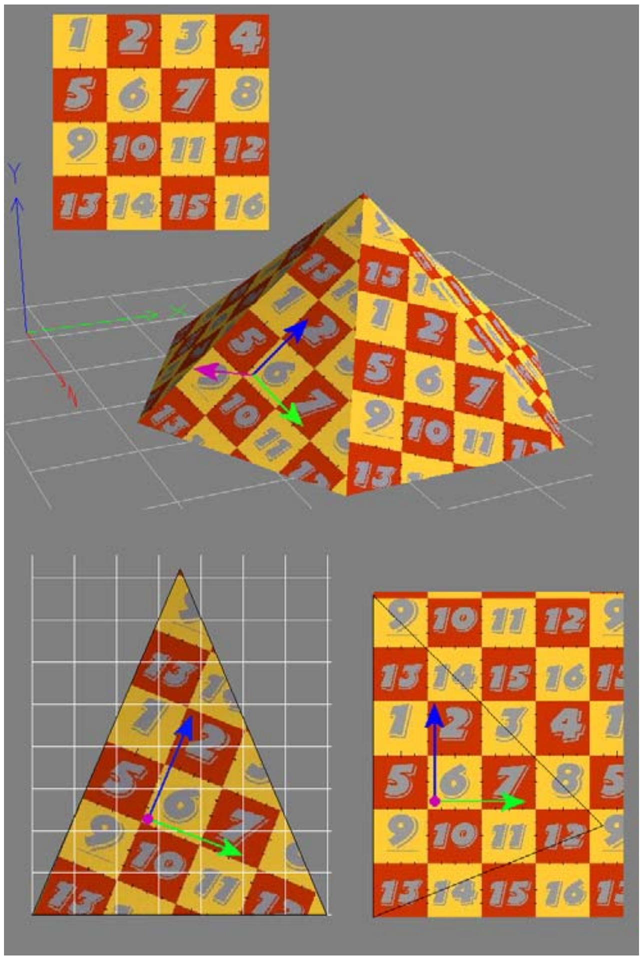


Figure 2 Unreal to UV mapping

4.2 The Converter Program: UnrealToX3D

UnrealToX3D is a dialog-based application that comes in its own installer. It uses few external components, most notably a 'tab' control for organizing the options. It runs in any 32-bit Microsoft Windows environment.

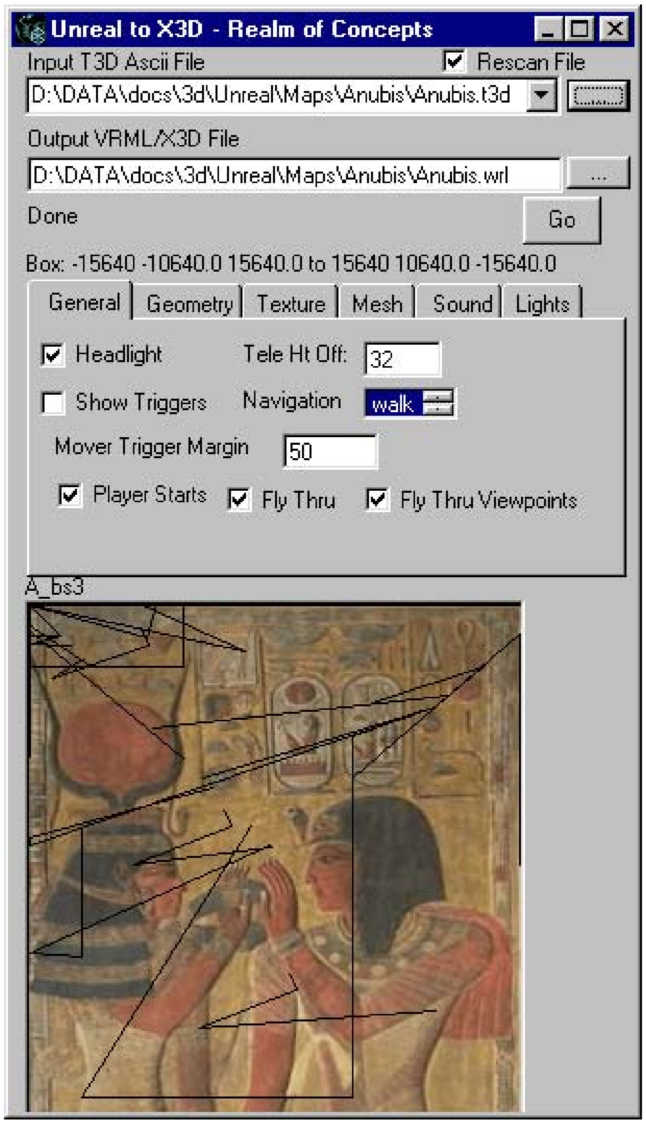
UnrealToX3D is currently implemented with Visual Basic. There is a parallel C++ effort, but it is lagging severely behind the VB code.

The UI for UnrealToX3D is pretty minimal. The tabbed sections provide options for the various components (geometry, textures, sounds, meshes, lights, etc.), and a picture display area shows bitmap textures as they are processed. There is also a display of the UV mapping, for added entertainment value.

The user can specify the base paths for the assets, which will be searched recursively when running. File extensions may be specified for the textures, which allows the user some flexibility with formats. The 'general' options include default navigation type, default 'headlight' setting, and whether to include fly-through viewpoints.

The user can also specify the margin around 'movers', which is a proximity sensor, which is attached to the moving object which tells the object to move when the avatar bumps into it. (Unreal has its own default, but the author can change it in this UI).

Additionally, the user can specify an extra margin above ‘teleporter’ entrances. This is needed because of the different ways Unreal and X3D specify avatars, and how they collide with things. In Unreal, the avatar is the shape of the player’s character, often times human. In X3D, the avatar is always a sphere. I found that because of this difference, teleporters needed extra room above them for the X3D player to catch the collision and trigger teleportation.


Unreal to X3D - Realm of Concepts
Input T3D Ascii File ☑ Rescan File
D:\DATA\docs\3d\Unreal\Maps\Anubis\Anubis.t3d
Output VRML/X3D File
D:\DATA\docs\3d\Unreal\Maps\Anubis\Anubis.wrl
Done
Box: -15640 -10640.0 15640.0 to 15640 10640.0 -15640.0

☑ Headlight Tele Ht Off: 32
☐ Show Triggers Navigation walk
Mover Trigger Margin 50
☑ Player Starts ☑ Fly Thru ☑ Fly Thru Viewpoints
A_bs3

Figure 3. The UnrealToX3d Application

UnrealToX3D remembers the last 10 files converted, and all options last used to make the authoring cycle faster. Also, if the file has been scanned, the user can opt to not re-scan it if the input file has not changed (as in the case where he is trying different options). This is handy when processing very large input files.

As an aid to ‘debugging’ the space, the user can also opt to ‘show triggers’. In this case, the proximity sensors are indicated with translucent non-colliding boxes of the same size. This lets the author see where his triggers are (see Figure 4).

During processing, the input file is parsed in several passes: one for geometry and textures, another for movers, another for lights, etc. If an asset file does not already exist in the destination folder, the paths specified are searched for suitable matches, and found files copied to the destination folder. Routes and scripts are generated as needed and put into a list which is attached toward the end of the output file. Both classic VRML97 and XML encodings are produced simultaneously.

After processing, the bounding box extents for the entire scene are displayed. If any assets were not found, list(s) of those will be displayed as well. Those lists are also saved as text files to the destination folder.

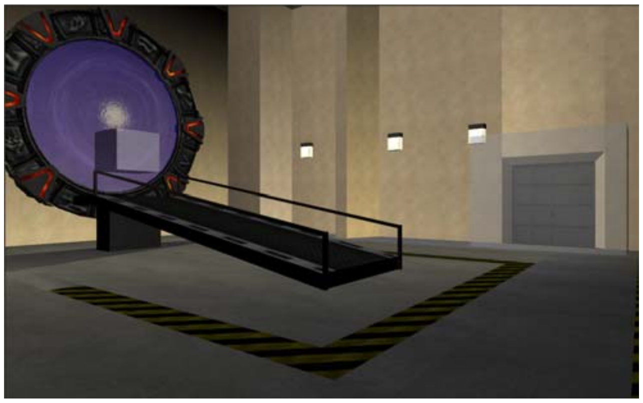


Figure 4. Visible ‘triggers’ (proximity sensors) in CTF-Karnack_SGC. One is the entrance to the Stargate’s invisible teleporter entrance. The others are around the doors exiting the Gate Room. Level design by Pat-BadKarma-Fitzsimons

4.3 Unreal Features Supported by UnrealToX3D

UnrealEd comes outfitted with thousands of components (‘actors’) for creating game levels. I have not implemented conversion for all of them. In fact, because Unreal is extensible, there is no way I could. Instead, I have started with the mostpopular components to allow authors to create compelling content.

- Objects – The gross geometry of the space. UnrealEd allows for additive and subtractive objects, but UnrealToX3D currently only supports additive geometry. However, geometry can be reduced from its component elements to a single object (using the ‘builder brush’) which is also exported with the T3D file. This allows the author to either build his entire space with additive geometry, or to build arbitrarily, and use the ‘builder brush’ to create a single object.
- Textures – Most textures used in Unreal are simple bitmaps. They are sized in power-of-2 on either edge. The surface editor allows the setting of various attributes such as UV panning, scale, rotation, transparency, etc. To the best of VRML97’s ability, these are supported in the current version of UnrealToX3D (X3D provides conveniences which VRML97 does not have, this will be addressed shortly). Currently not supported are animated textures or procedural textures (although screenshots may be cropped and used as a static version).
- Movers - These are objects that move. UnrealEd allows for key-frame animation of objects, as well as a few simple physical motions, like constant rotation (UnrealEd 3.0 includes more physics, but that’s for future work).
- Lights -The basic light is generally a point light, with a color, intensity, and radius. Other supported light types are SpotLights, and ‘flicker lights’ which have a strobing effect. Other lighting effects can be scripted as well, this is for future work. All lights are exported without regard to what the target

application will do with them. This allows the author to import the X3D to, say, 3D Studio Max, and get a nice ray-trace of the space.

If a lens flare texture is indicated, this is exported as a screen-facing billboard.

Curiously, X3D does not have an 'ambient light' node. Unreal allows for an ambient light. To get around this, the ambient light specified in the game LevelInfo structure is applied to all materials as an emissive color. It isn't dynamic, but it is better than nothing.

• Sounds - Tokens representing sound sources may be placed in the space. They may be looping, one-shot, and/or triggered. There is also support for 'dynamic ambient sounds' which is a list of sounds that play randomly given several parameters. These can be used to great effect for outdoor ambiance.

• Meshes - In Unreal technology, a Mesh is a reusable textured object. They may be animated. The T3D file does not include mesh information, only a reference to the mesh (and its location, orientation, etc.). No attempt is currently made to extract meshes with UnrealToX3D. A separate package, UTPT, can be used to export meshes to 3DS format, and several packages may be used to convert those to X3D. In practice, meshes are brought into the X3D space as 'inlines' (externally referenced reusable files).

• Triggers - A trigger may be a proximity sensor that sends an event when it is entered (or exited), based purely on time, or both. Special triggers, such as 'Dispatchers' will multiplex one event into many, with optional time delays. A Trigger can be used to fire a sound, mover, dynamic light, other triggers, etc.

A special circumstance is a 'bump mover' which begins to move when the avatar bumps into it. Examples might be doors or push-buttons. This is accomplished in X3D by creating a proximity sensor to enclose the object which will move, and routing that sensor the to the timer which drives the interpolators which move the object.

Triggers which use proximity detection are defined in Unreal as cylnders, but X3D currently only allows for box-shaped sensors. UnrealToX3D constructs a box which encloses the cylinder in these cases.

• Skyboxes - A skybox is some geometry that appears very large and very far away. Generally, they are used to display a sky with clouds, stars, planets, trees, mountains, etc. In X3D, this might be done with the 'Background' node, but that is just a simple static cube. An Unreal skybox can contain any geometry, lights, triggers, movers, etc. So the author may have birds, planes, clouds, all moving in the skybox. To do this with a panorama, one would need to project a rendition onto the bitmap(s) which are applied to the background, but resolution and animation would suffer, and interactivity would be completely lost. Instead, UnrealToX3D allows the author to convert the skybox as a separate file, with relatively large dimensions, and the 'main map' is loaded as an inlined file within that larger space.

• Viewpoints - In X3D, Viewpoints are basically cameras. In Unreal, the author can place 'PlayerStart' tokens (which are where players will spawn into the game). These PlayerStarts are repurposed to be Viewpoints.

• Teleporters - A common mode of 'transportation' in Unreal games is the Teleporter. This is an area that, when the avatar enters, is magically transported to a new destination. There are two types of Teleporters supported in UnrealToX3D, the VisibleTeleporter (which looks like a small cyclone), and a standard Teleporter, which may or may not be visible. If it is visible it generally uses a screen-facing billboard texture. These are implemented with proximity sensors at the teleporter entrance and a viewpoint at the teleporter exit. The proximity sensor is routed in such as way as to bind the avatar's viewpoint to the teleporter's exit viewpoint. In practice, the player walks into the teleporter and is suddenly somewhere else.

• Flythroughs – UnrealEd allows for fly-throughs. This is where the avatar is flown around the space in a guided tour. The author uses 'InterpolationPoints' to set the path, orientation, speed, and FOV. In-game, there is a smooth spline-like interpolation between these points, which appears to use the Chaikin algorithm.

• General options – UnrealToX3D by necessity includes several general options. These include the type of default navigation (walk, fly, etc.) to be used, the overall scale of the space, and so forth. If the Unreal map contains authoring information, it is brought into the X3D file as well.

4.4 Behavioral Features

Interactive components in Unreal are implemented via a simple tag/event naming convention. Any component can have a name (tag), and many can issue events. When an event is fired, (say by a proximity sensor), any other component having a tag that matches the event is fired. It was possible to employ Unreal's tag/event convention to connect events to triggered motion, lighting, and sounds. This is done in X3D through 'routes'.

Where X3D does not have built-in support for a behavior, it may be scripted through ECMAScript (or Javascript), and where behaviors seen in Unreal are not readily available, it is possible to provide ready-made scripts to emulate them. As of this writing, these scripts are embedded in the X3D file, but an external library of PROTOs is available as well, and will eventually be used optionally instead. Protos are a way to create new elements for X3D, or to sub-class existing ones.

For example, one such behavior is UV-animated textures. Unreal allows for time-based panning of textures, which work very well for emulating flowing water, moving clouds, lava, fire, etc. Until X3D, a script was necessary for animating the texture coordinates. The conversion tool computes the values for the timer and interpolation of the texture coordinates, and routes them automatically.

5 The Authoring Process

Using UnrealEd in conjunction with UnrealToX3D requires a few simple steps. In general, the author will have UnrealEd, UnrealToX3D, and an X3D viewer running at the same time. A small amount of set-up is involved, after which most of the process is straightforward.

1. Create asset libraries

If you are exporting an existing map, you can use the command line for UCC (which is already installed with the game) to export textures as BMP files. For example:

```
ucc BATCHEXPORT Ancient.utx TEXTURE BMP
c:\data\Textures\Ancient
```

This will extract all the textures from the binary texture package 'Ancient.utx' and put them into a folder called 'Ancient' in the indicated path.

Then use any paint package you like to batch convert them to JPEG, because that is what X3D likes. In the case of masked textures (those with one or more bits of alpha transparency), you should convert those to PNG format. Export sounds the same way:

```
ucc BATCHEXPORT AmbOutside.uax SOUND WAV c:\data\Sound\ AmbOutside
```

This produces WAV files, which are X3D-ready. Use UTPT to export meshes to 3DS format, then any package that can read 3DS and output VRML and/or X3D. You will likely have to link the texture file(s) to the mesh. I use 'Deep Exploration' from Right Hemisphere to assign those textures.

If you are creating new original content, you probably want to import your assets into UnrealEd so that you can design with them. Textures should be sized in powers of 2 (ex: 128x256, 512x512, etc.) WAV sounds can be imported pretty much as-is. Meshes can be imported from Lightwave and Maya (and others) using various free utilities.

In any case, remember that the assets in UnrealEd need to have external file representations.

I have created a folder structure that starts with 'Textures' and contains one folder for each Unreal texture package. I have done similar for 'Sounds' and the sound packages. UnrealToX3D will recursively search the folders for assets, and copy them to the destination folder (if they do not already exist there). Meshes currently need have their associated textures in the same folder. I dumped all my meshes into one folder.

2. Create the space.

There are many online resources for tutorials on using UnrealEd. If you are exporting an existing level, skip to step 3. If you are creating from scratch, you may want to hollow out the space, and use additive brushes only. If you do need to subtract geometry, you can use the 'Builder Brush' to reduce that geometry to additive-only by intersecting a 'Builder Brush' with the objects in question, and then adding the new 'Builder Brush' to your space (and delete the original components of course).

3. Check masked textures.

Fires, flares, etc. may sometimes have the masking indicated only in the texture package, so use texture options to set the 'masked' flag as needed.

4. Export.

If all your geometry is additive, just export the entire level to a T3D file. If your geometry is mixed additive and subtractive, surround level with 'Builder Brush', right-click on it, 'Transform Permanently' (which removes any transforms on the 'Builder Brush'), Intersect to create a 'Builder Brush' which contains all the geometry, and export.

5. Skyboxes:

If you are using a skybox, you should export the main part of the map to one T3D file, and the skybox to a separate one. You may want to save your map as two separate files, one for the main map, and one for the skybox, to clean out the non-geometric components. For example, lights, sounds, and viewpoints in the main map need not appear in the skybox, so those should be deleted before exporting the skybox.

6. Run UnrealToX3D.

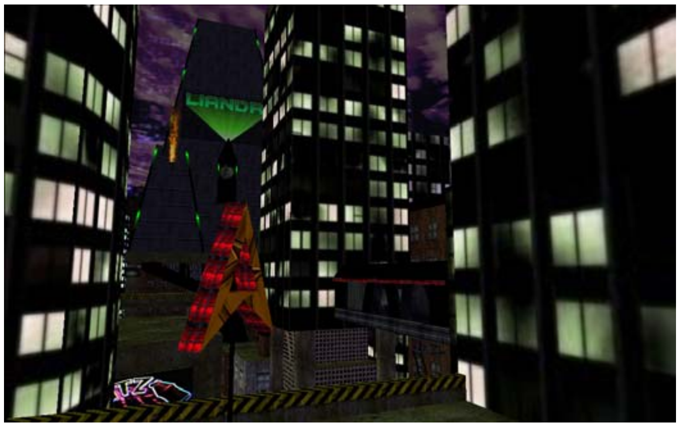
If the file contains a skybox, you should check that option on the Geometry tab. You may want to explore other options as you see fit. If you have not changed the T3D file and you are re-running the conversion process with different options, you don't need to have it re-scan the input file (which can take quite a while on large files). As of this writing, UnrealToX3D produces three files: a VRML97 WRL file, an X3D X3D file, and an X3D XML file. The last two are identical except for the extensions. XML files can be easily loaded into Internet Explorer for examination of the textual contents.

7. Test

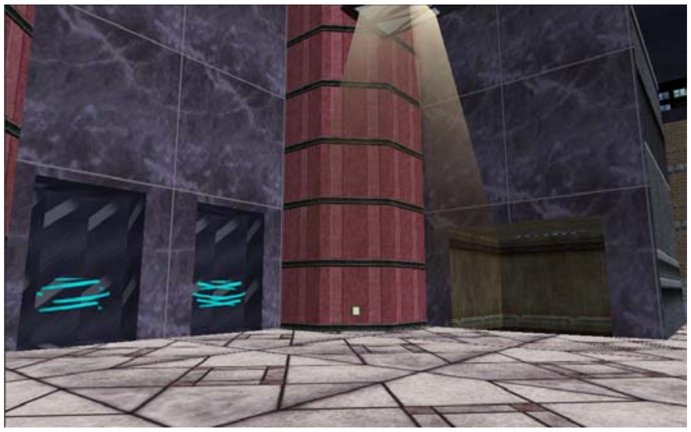
Use your favorite viewer. Currently, all the files needed for the resulting X3D are placed in one destination folder. You will probably just double-click the X3D file. Go to step 3 as needed.

6 Examples

The following are a few screen shots of Unreal-based game maps in X3D browsers. Of course, you can't hear or interact with any of these pictures, so use your imagination.



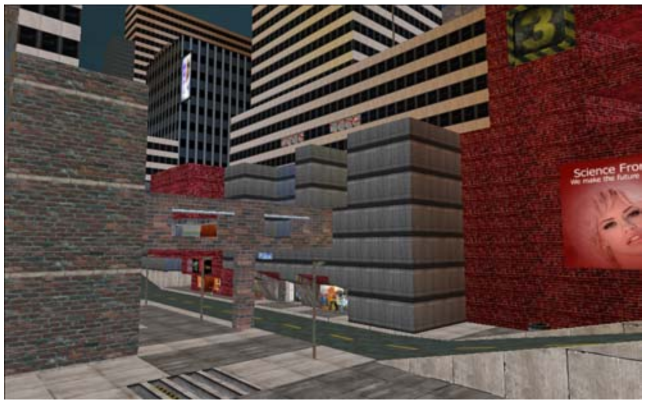
CityIntro – From the opening intro of Unreal Tournament. Features: clouds in the sky move, chaser lights on buildings, animated meshes, positional sounds, and interactive fly-through. Level design by Shane Caudle.



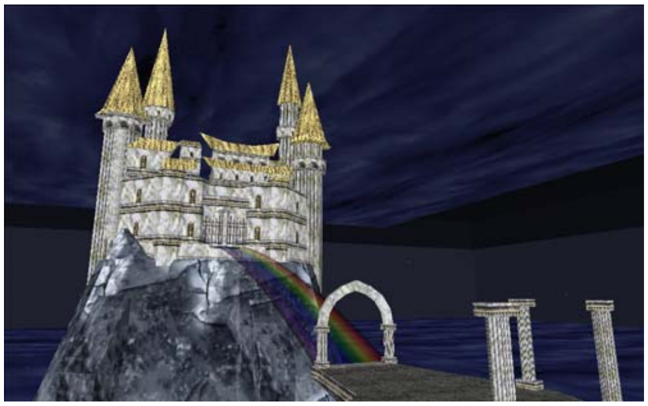
CTF-Midtown – A popular fan map. From inside the Red Base. The light-blue 'cyclones' are Visible Teleporters, and they swirl around. The shaded pyramid of light is a 'LightBox' mesh. Level design by Pavel 'NeSoLo' Bazylevich.

DM-Falkenstein – Arguably the most amazing Unreal space ever. It is a faithful rendition of the Falkenstein castle (never built) of Ludwig II of Bavaria. Masked textures make those chandeliers and railings very lightweight. A teleporter is barely visible here. The portraits are all custom textures. Level design by Robert (YoMammy) Wey.

DM-DuskToDawn – An amazing reproduction of the setting of vampire-ridden movie "From Dusk Till Dawn" directed by Robert Rodriguez. This space features many custom textures inspired by the movie, sounds (including the tune Salma Hayek danced to), doors that open when bumped, and one of the coolest skyboxes ever. Level design by Angelheart.



DM-ZeitkindPro – One of the best city simulations. This space features realistic dynamic ambient sounds, elevators, and various moving textures. Level design by Juergen Vierheilig.


Asgard (The Journeys End) - This is the final sequence from the game 'Rune' and was exported via RuneEd. Note the translucent Rainbow Bridge. There are several moving cloud layers, and swirling stars to boot. Level design by Mick "VerMoorD' Beard.

7 Open Issues and Further Work

Although functional and useful, there is much work that can still be done on this conversion process. The following is a short list.

7.1 UnrealToX3D Enhancements

There are plenty of opportunities to improve the conversion application:

Subtractives – It is unlikely that X3D will ever support subtractive geometry. UnrealToX3D could, however, provide a service to reduce the additive/subtractive objects into just additive objects. This would remove the step of intersecting the entire space with the builder brush and making one large object.

Using binaries – Currently UnrealToX3D only handles the text-based T3D file, which only describes the scene. If, instead, the compiled binary files that contain the light-baked textures, BSP cuts, and meshes, were utilized, a huge gain in quality and ease of use could be realized.

Animated textures, multitextures and shaders – Currently, UnrealToX3D only deals with simple static bitmaps. Unreal allows for page-flipped animations, which could be turned into GIF or MNG, or even AVI files. The 'FirePaint' procedural textures could probably be emulated with X3D shaders. UnrealEd 1 and 2 have limited texture options available in the T3D file, but some multi-texturing capabilities exist in the binary format. And UnrealEd 3.0 has moved most of the texture options, with much greater multi-texturing support, to the binaries.

Rotating skyboxes - Unreal allows for the skybox to rotate. This can give the illusion of day turning to night, or being on a spinning asteroid in space. With slight modification, UnrealToX3D could inline the skybox (rather than the other way around), and animate the transform on the skybox.

Smooth fly-throughs – As of this writing, UnrealToX3D just moves the fly-through viewpoint point-to-point between the control points on the interpolation path. A spline-like interpolator can be scripted and employed, using Chaikin's algorithm. UnrealEd 3.0 uses Bezier control points, this would also need to be scripted.

Fog – Both Unreal and X3D have fog. It just needs to be implemented in UnrealToX3D.

Better integration– It would be nice to have X3D as 'Import' and 'Export' options within UnrealEd itself. It may be possible to integrate conversion functionality into UnrealEd via UnrealScript, or it may require the cooperation of Epic Games.

Using X3D to its fullest – As of this writing, UnrealToX3D does not take advantage of any of the new features of the X3D specification. There's no reason it shouldn't. These would include the event utilities, 2D interpolators, and multitexturing.

Compression – X3D allows for the data to be compressed with GZIP. This is not currently integrated into UnrealToX3D.

Greater support for UnrealEd 3.0 – UnrealToX3D was developed before UnrealEd 3.0 hit the stands. Small changes will be needed to get textures to work the same, but UnrealEd 3.0 includes many new features which would be nice to support.

Multiplayer – No work has been done in the realm of multi-player (multi-avatar) spaces. It is unclear what, if anything, would need to be done.

7.2 X3D Enhancements

As rich as it is, there are still some things missing from the X3D specification that would make conversions complete, and the experience more real.

Global ambient light – Could be scoped to the entire space, or within zones.

Enhanced lighting model – Currently, X3D uses vertex shading only, and is specified only to a maximum of 8 lights. It could allow for viewers to auto-bake static lighting at load-time, or use pixel shaders.

GIF support – Unisys notwithstanding, masked (1-bit alpha) animated bitmaps are ubiquitous. They are probably the easiest 'movie' texture to create, they are light-weight, and can include transparency.

Arbitrary shapes for avatars, zones, and proximity sensors – The current specification allows only for boxes for proximity sensors, and a sphere for the avatar. This makes realistic collisions impossible in some cases. In Unreal, fog is specified per zone, which may be implemented in X3D as a cell. The specification would need to be extended to allow for arbitrary geometry for all these nodes.

Navigation modes including 'riding movers' – Bitmanagement Software has taken some initiative along these lines, allowing for 'game' navigation, which gives additional degrees of freedom while walking. But the specification would need to be amended to allow the avatar to 'ride' objects that move by jumping/stepping upon them. The avatar would then be attached to the object while it moves, while still allowing the avatar to move about freely upon it. This would allow for realistic elevators, cars, planes, etc. Actually, this may work in some viewers, but since the specification is non-specific in this regard, it's not guaranteed. The ability to crouch and jump would also add realism.

Particle systems – UnrealEd will let you place parameterized particle generators of many sorts for sparks, smoke, etc. These can be scripted, but native support would be more efficient.

Shadows – There is no provision in X3D for shadows.

Reflections – Real live reflective materials, not just environment mapping.

Doppler shift – When there is relative motion between a sound and an avatar, the pitch of that sound could be Doppler-shifted.

Gravity and velocity – Improved support for altering the direction and stength of gravity, and a 'velocity' component, would allow for more realistic simulation of non-Earth gravity spaces, flowing water, strong winds, etc. These are all supported in Unreal already.

7.3 Somewhere in between - Universal Behaviors?

Short of extending the X3D specification, a standardized set of external protos which define behavior can be stewarded. All the scripting that describes the behaviors would be in these external files. The references to these protos, used in the body of the X3D itself, could in theory be standardized and recognized by X3D tools, such that these behaviors could be imported and exported. This would be similar to the Universal Media project currently supported in the X3D community. Such an effort would mean an author could attach simple behaviors, such as 'move when bumped' and use another tools to make further modifications, without losing this behavior. In theory, conversion to/from other systems (Quake, Shockwave, Axel, Atmosphere, etc.) would also then be possible.

8 Conclusion

The free Unreal Editor can be used to create realistic game-like experiences in X3D, which can be experienced by anyone with an appropriate browser plug-in. With UnrealToX3d, anyone can now make rich, compelling, and very lively 3D experiences, for almost no cost, and with no need for programming expertise or knowledge of the internal representation of the spaces they create. Interactive elements are designed visually, rather than with a text editor, so the author need not be an engineer, but an artist. Unreal authors can publish their work online to advertise their work, whether as a game preview, or completely independent art. The price and demographic for Unreal makes it a great choice for education, both to teach the authoring of interactive 3D, and to publish student projects.

X3D is an excellent format for the interchange of interactive 3D spaces. As more authoring environments support the ability to use X3D, and as they become cheaper and easier to use, 3D spaces will likely proliferate on the Internet.

What has been describe here is only the beginning. Ongoing work in this area can bring added realism and increased simplicity to the creation of virtual realities.

To get your copy of UnrealEd, buy Unreal-based games, such as Unreal II and Unreal Tournament 2003 for Windows PC.

To get your copy of UnrealToX3D, visit www.unrealroc.com and download it for free.

References

Unreal Technology at unreal.epicgames.com
Web3D Consortium at www.web3d.org
Unreal Realm of Concepts at www.unrealroc.com
UTPT (Unreal Tournament Package Tool) by Antonio Cordero Balcázar at www.acordero.org/projects/utpt

Acknowledgements

Many thanks to Tony Parisi of Media Machines for providing guidance on this paper, and all manner of mentoring on VRML and X3D.

Thanks also to Alan Hudson at Yumetech, and Aaron E. Walsh of Mantis Development Corporation, for testing and guidance.

Thanks to all the Unreal mappers, only a few of which are mentioned here, for their spectacular efforts.
Most of all, thanks Epic Games and their support of Unreal Editing.