Unreal Model Importing: The 3ds2unr Converter

Developed by Legend Entertainment (Creators of Wheel of Time)
http://www.legendent.com/
http://www.wheeloftime.com/
Programmed by David Townsend
townsend@patriot.net

Last Updated: 07/01/98

Tools you need

To create animated models for Unreal mods, you will need the following tools:

1. A good 3D modelling/animation package, such as 3D Studio Max, Lightwave, SoftImage, or Alias PowerAnimator. Our converter currently only supports 3D Studio Max.
2. A model conversion tool which converts your models from their native format (for example, the converter available here supports 3D Studio Max) to Unreal's format (Aniv*.3d, Data*.3d).
3. UnrealEd. The beta version shipping with the game will do the job.

Required Reading

To be able to take advantage of this tool, you need to become familiar with:

1. The contents of this document.
2. How UnrealScript works (for writing code to control the models). See the UnrealScript Reference.
3. How Unreal packages work, specifically how to rebuild scripts from .uc files using "Unreal.exe -make". See the Package Reference.
4. The UnrealScript #exec commands for importing meshes, scaling them, and assigning names to animation sequences. These haven't been documented anywhere, but there are hundreds of examples in the code. See the Skaarj, Brute, and AutoMag scripts for examples.

Next, be aware that importing animations into Unreal has proven to be the trickiest aspect of Unreal content creation, because of the multiple tools that need to be used (a 3D modelling program, 3ds2unr, and UnrealEd), and because of the large number of steps that need to be followed carefully in order to successfully import and use a model in Unreal.

Files you'll deal with

For the sake of clarity, here are the files you'll be dealing with:

- *.3ds: Your original 3D Studio file, which includes: Your model. Your animation frames. Your texture coordinates.
- Aniv*.3d, Data*.3d: Two intermediate files generated by 3ds2unr.exe. Suitable for importing into UnrealEd.
- *.uc: Your UnrealScript class definition, which contains #exec commands for importing your mesh. A text file, generated either by UnrealEd (when exporting a class) or by any text editor you might use to author it.
- *.u: Your Unreal class file. A binary file containing all the data: the class and script code, the mesh, any sound effects, etc. Generated by "Unreal.exe -make"

Downloading the utility

Here is the 3D Studio to Unreal converter, including the utility (3ds2unr.exe), quick documentation (ImportingModels.doc), and the source code. The source code compiles with Microsoft Visual C++ 5.0 using Service Pack 2. We haven't tested with other compilers, versions, or service packs. The source code is unsupported and is provided as-is. Enthusiasts have permission to modify and redistribute the code on a non-commercial, not-for-profit basis.

- Download it

Here is an example of some of the cool things people could do by modifying the source. :)

- Make converters for other file formats, such as Lightwave, SoftImage, and Alias PowerAnimator.

Modeling

Models must be made up of a single 3D Studio object, which lives within the 256x256x256 coordinate space centered on the origin. (Setting the grid extents to –128..128 and turning on the grid is helpful here.) The model itself should be centered on the origin (0,0,0) because of how Unreal specifies the collision cylinders used to determine when objects are touching.

Models will lose some precision when imported. Specifically, X and Y coordinates are truncated to the nearest 1/8th (0.125) unit, and Z coordinates are truncated to the nearest 1/4th (0.25) unit.

One 3DS coordinate equals 6" in Unreal X and Y coordinates, or 12" in Unreal Z coordinates. You generally don’t have to worry about this scaling, as the model can be rescaled when it’s imported into Unreal.

Object polygons shouldn’t intersect. The hardware supported by Unreal gets confused when polygons intersect, resulting in an unpleasant shimmering.

If your object is a character that will be carrying a weapon that will be separately modeled, place an extra polygon (the weapon triangle) in or on the part of the character where the weapon should be. This polygon will be invisible in the game; it’s just used to sync the weapon’s position with the character’s animation.

The conversion tool will only convert objects with sequence numbers in their names (Obj01, Obj02, ...). The object name itself is meaningless; it's the sequence numbers that count. They must begin at 01 (even if that's the only frame!) and you can't have any "holes" in the sequence. (Under 3D Studio 4, it was easiest to use the "Snapshot" option; I'm not sure how 3DS MAX handles this.)

Objects that won't be animated (e.g. 3rd person weapon views) should be in their own .3DS file (e.g. save the angreal of healing as AngrealHeal.3DS).

Animation

Make sure that all animation frames stay entirely within the 256x256x256 coordinate space.

The Unreal animators are typically designing their animations for 30-35 fps playback. Then, to save memory, they shave the animations down to 15-17 fps and let Unreal do the tweening.

Objects that are animated should have a separate .3DS file for each animation sequence (Run.3DS, Walk.3DS, Shoot.3DS, etc.). You probably want a separate directory for each animating character, so that each can have it's own Run.3DS sequence.

Texture Mapping

You still use 3D Studio to do your texture mapping. However, there are some (pretty heavy) restrictions on what will actually show up in Unreal.

The only type of texturing that has any effect is using actual texture maps – bump maps, opacity maps, specular maps, etc. are meaningless. Furthermore, the bitmaps that you use as textures should be 256x256 pixels in size, and you can't have more than nine of them for any one model. To conserve memory, you should use as few texture maps as possible. 3DS Max V2 allows you to map part of a material to your geometry; using this feature lets you squeeze many smaller texture maps together into one big bitmap.

Our Wheel of Time characters currently use two texture maps. The angreal use only one.

Under most circumstances, the material names that you use in 3D Studio don't matter. However, there are several pre-defined material names that you can use to get special effects when the model is imported into Unreal. The special names are:

SKIN	This texture will be assigned as the model's Skin property in Unreal. Skin textures can be replaced easily at runtime (to show increasing damage, for example) and are the basis for Unreal's corona effects.
TRANSLUCENT	The polygons mapped with this texture will be marked as translucent, and can have their transparency set by the program. Wheel of Time uses this to get the transparent globe on the Shield angreal.
TWOSIDED	The polygons mapped with this texture will be marked as two-sided, so the texture mapping will show up on both sides of the polygon. Wheel of Time uses this for the characters' cloaks.
WEAPON	The polygon (there should be only one!) mapped with this texture will be marked as the weapon triangle. So it won't be visible

Note that this naming convention applies to the material name itself; the filename of whatever TGA file you use as the bitmap is irrelevant.

Converting to Unreal Format: Preparation

You'll need a copy of the 3ds2unr.exe conversion program, plus some way (Photoshop, Debabylizer, etc.) of converting TGA files to PCX format. A text editor (Notepad will do in a pinch) is also handy.

The conversion program expects an Unreal-like directory structure to exist before it will run. Specifically, you need:

1. a main Unreal directory (mine is c:\Unreal), below which is a
2. project-specific directory (for example, c:\Unreal\WOT), below which are
3. two directories, Models and Classes (e.g. c:\Unreal\WOT\Models and c:\Unreal\WOT\Classes)

This directory tree is what Unreal itself expects, so it may already be set up. If it isn't, you'll have to make it yourself before doing any conversions.

The converter is a Windows console application, meaning it runs by typing rather than pointing and clicking. Make sure that the 3ds2unr.exe program is somewhere in your DOS path before you begin.

Using the Converter

Once you're set up, using the converter is pretty simple. Just type 3ds2unr, followed by an optional class name and one or more .3DS files, like so:

```
3ds2unr Trolloc Walk.3ds Run.3ds Attack.3ds
```

The converter reads the given 3DS files, and produces a class file (Trolloc.uc) and model files (Trolloc_a.3d and Trolloc_d.3d) to be used by Unreal.

If this is the first time that you've run the converter, you'll be prompted to select your project directory (i.e. the directory from item #2 in the preparation list (for me, c:\Unreal\WOT). Once you've set this directory, you won't need to set it again. If you do need to change it, because you've began a new project, just type

```
3ds2unr –setproj
```

and you'll be able to select another project directory.

When specifying 3DS files, you can use standard DOS wildcard characters (`?' and `*'). So, if all of those Trolloc animation sequences were the only 3DS files in the current directory, you could type:

```
3ds2unr Trolloc *.3ds
```

The class name is just the name of the object that you're converting (Trolloc, AngrealHealing, Myrddraal, etc.) and is used as the base filename for the converter's output. If you're converting an object with no animation, you can omit the class name, in which case the base name of the sole 3DS file is used as the class name. In other words, this:

```
3ds2unr AngrealHeal.3ds
```

does what you probably want it to. However, if you try to convert more than one 3DS file at a time without supplying a class name, you'll be prompted to make sure that the class name inferred by the converter is correct. For example, omitting the "Trolloc" from the previous example is probably an error:

```
3ds2unr *.3ds

ClassName [Walk]?
```

You can just hit Enter to continue with that name, or type the correct name (Trolloc) first.

Be careful with your class names – the converter will happily overwrite existing files.

Converter Output

The converter will create three files:

1. ...\Models\ClassName_a.3d
2. ...\Models\ClassName_d.3d
3. ...\Classes\ClassName.uc

(where "..." is your project directory).

The .uc file will contain the appropriate UnrealScript commands to import your model, list its animation sequences, and set its textures.

Unreal needs all three files to use your model. The same three files are used by the Mesh Viewer (see below, and separate documentation) to view models.

Fixing Texture Maps

Since the converter doesn't read 3DS materials files, and Unreal requires PCX rather than TGA-format input, you have to manually help out the converter.

First, you need 8-bit PCX versions of all of the textures that you used in the model – use Photoshop, etc.

You then need to examine the .uc file in a text editor that can handle straight ASCII text – Notepad is fine. Look for the lines that begin with #exec TEXTURE IMPORT and note that each such line ends with a material name after a pair of slashes:

```
#exec TEXTURE IMPORT NAME=JSpider1 FILE=MODELS\Spider1.PCX GROUP=Skins FLAGS=2 // SpiderSkin
```

(The above should be a single line, but can't be shown here legibly that way.)

Notice the reference to a Spider1.PCX file. Both Unreal and MeshViewer will need this file to properly display your model. It corresponds to whatever TGA file you used in the given material, SpiderSkin. You need to have this file name match the name of the material's 8-bit PCX file, which you can do by either editing the .uc file directly, or renaming your PCX file to correspond with the name in the .uc file.

Using the above example, say that the SpiderSkin material was made with a SpiderSkin.TGA texture map. You'd convert that TGA file to and 8-bit SpiderSkin.PCX file. You could then either rename that file to Spider1.PCX (because that's what the script is expecting), or edit the script to look for SpiderSkin.PCX instead of Spider1.PCX.

Conversion Errors and Warnings

You might encounter the following problems when converting:

```
can't find file
```

You referred to a file that doesn't exist. Make sure you've typed the name and directory correctly.

```
No project directory found – exiting
```

The converter couldn't establish a project directory, either because you cancelled out or there's a problem in the system registry. Make sure that your project directory structure is correct, then try rerunning the converter with the –setproj option before attempting conversion again. The converter cannot run without a project directory.

```
read error [TAG]
```

The converter was expecting data that wasn't there. This probably means the 3DS file is corrupted.

```
too many textures
```

Unreal has a limit of ten materials that can be applied to an object, and the converter encountered an eleventh material. For memory and rendering performance reasons, you really should be using less than ten anyway.

```
Warning: out of sequence obj (Foo) skipped
```

The converter expects to find sequentially numbered objects in file, and it found an object either without a sequence number (e.g. Box instead of Box01) or with an out-of-sequence number (e.g. Box01, Box02, Box04). This is probably something you want to fix, although in some cases, like leaving a stray light source in the project, it might be OK to ignore (providing that the correct object was converted).

```
Warning: Filename.3DS: Bad coordinate x.xxxxx, y.yyyyy, z.zzzzz
```

The converter found a coordinate that was outside the allowable 256x256x256 coordinate space. Rescale your model or fix the animation sequence. To minimize warning lines, only one bad coordinate is reported per object in the file.

If you encounter errors other than those listed here, it probably indicates a converter bug.