HYPERCOSM APPLETS AND SECURE ENCODING OF 3D MODELS

A White Paper prepared by Hypercosm LLC., December 2005

Introduction

When displaying interactive 3D information over the web, there is often a requirement to visually display an interactive view of a 3D model on the end user's machine without making the actual 3D model information available. This presents a dilemma because the 3D model must exist on the client machine in order to be displayed, but it can not be transmitted in a form where it can be extracted. This white paper describes Hypercosm's unique approach and solution to this problem.

Hypercosm's Solution

Hypercosm solution to this problem relies upon a technique known as "procedural modeling". Using procedural modeling, you describe not the geometry of the object or scene, but rather a series of instructions on how to create the object or scene that is desired. This approach relies upon a suitable "interpreter" to decode and execute the instructions in order to generate the final result.

Hypercosm Patent Information

For more information on this approach, see Hypercosm's Patent:

"Method and Apparatus for Data Compression for Three-Dimensional Graphics"

This patent is US Patent #6,426,748 B1, awarded on July 30, 2002.

This approach is desirable for three main reasons:

Benefits of Hypercosm's procedural modeling approach:

1. **Efficiency (file size)** - It is often much more efficient to describe how to create an object than to describe the object itself.

2. **Security** - In order to decode the object, you must be able to interpret the instructions to create the desired objects. This is more difficult than interpreting a static description of the object and it must be done correctly to yield the proper result.

3. **Flexibility** - since we are describing how to create the 3d scene, the initial creation parameters can be altered to yield new variations. For example, the tessellation parameter can be changed and the objects can be re-generated with a lower polygon count.

An Analogy

A useful analogy for this approach is to think about DNA.

1. **Efficiency** - DNA encoding is certainly compact and efficient since it allows the entire description of a person to fit inside of a single cell. In order to "decode" the DNA, the organism is grown. This is similar to what happens when the applet runs and the 3d geometry of the

scene is constructed according to the applet instructions.

2. **Security** - deciphering the meaning of DNA has proven to be a very difficult process because in order to understand it, we must understand not just the genome, but also what will be the result of the genome when it is expressed. A similar analogy is to think of a computer program. It is not too difficult to understand the meaning of a software program when looking at it line by line. However, predicting what will happen when that program actually runs can be a very difficult process, even for the programmer who wrote it.

3. **Flexibility** - DNA is responsible for encoding every form of life on Earth. One of DNA's main purposes is to allow flexibility and variation since that allows for evolution. Another analogy from software is to think of the difference between a computer language and a file format. A computer language can encode an unlimited set of ideas whereas a static file format is restricted by its definition to a specific set of functionality.

The Encoding Process

The encoding process relies upon a series of steps as described below.

Step 1: Convert 3ds max File to OMAR Script Code

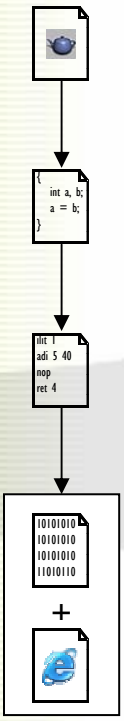
In the first step, the high level information in the 3ds max file is converted into script code written in Hypercosm's OMAR scripting language. This step is performed by Hypercosm's Teleporter software. The OMAR language is a Java-like language with extensions built in for 3d graphics. It is similar in structure to Java but has built in data types for vectors, transformations and scene graph management to make 3d graphics operations easier to code and more efficient to execute. 3ds max primitives, modifiers, animation controllers, materials, and other entities are written in OMAR script code that is built on top of very basic and general purpose 3d graphics subsystem. This allows the Hypercosm Player to only support a basic set of primitives such as 3d meshes while higher level entities such as lofts, extrusions, and other primitives can be encoded in the scripting language. This system allows the Hypercosm Player to support many different high level formats. It also allows the system to be maintained more easily because improvements to the export capability can be made by changing the script code that is exported rather than requiring changes to the Hypercosm Player. The ideas behind this approach are similar to the ideas used in the Postscript language that is used for 2d graphics and desktop publishing. The Postscript language is based upon a simple scripting language called Forth which is used like Hypercosm's OMAR, but for encoding descriptions of 2d objects.

Step 2: Convert Hypercosm Script Code to Bytecode Assembly Language

In the next step, the OMAR script language is compiled into a bytecode assembly language. This is similar to the process used by Java compilers to convert human readable text based code into an intermediate representation that can be interpreted by the computer. This step is performed by the OMAR compiler that is built in to the Hypercosm Teleporter.

Step 3: Convert Bytecode Assembly Language to Applet

In the last step, the OMAR Bytecodes are compressed and combined into a binary format along with other information about the applet which may include inline resources for sounds and textures, version information, and optional digital signature information (publishing keys) for restricting the playing of applets or tagging applets with a watermark.



3ds max file – File used by 3ds max containing geometry, modifiers, materials, and references to plug-ins to interpret this data

Hypercosm Teleporter

OMAR Script – Text readable script code written in Hypercosm's Object Oriented Modeling, Animation, and Rendering Language (OMAR)

Bytecode Assembly Language – Mnemonic representation of low level Hypercosm Bytecode instructions – this is used internally by the Hypercosm compiler and is never seen

Hypercosm Applet – Binary compressed representation of Hypercosm Bytecode instructions needed to generate 3d scene geometry, materials, etc

Standard HTML Web Page – This web page serves as a container for the Hypercosm Applet. The web page is viewed in a standard browser and the applet is displayed by a web browser plug-in

Figure 1: The Encoding Process

The Decoding Process

In the decoding process, we follow the instructions encoded in the applet to create and then display the desired shapes. This process is performed by the Hypercosm Player using a Hypercosm Applet as input.

Step 1: Interpret Hypercosm Applet Instructions

In order to decode a Hypercosm applet, we must have an appropriate interpreter to actually execute the Hypercosm applet.

Step 2. Create Scene Graph

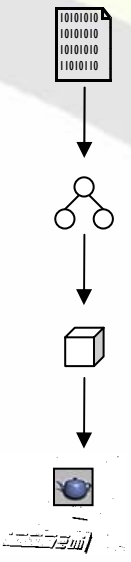
The scene graph is the underlying data structure that represents the high level organiation of the scene. This is constructed according to instructions executed by the interpreter.

Step 3. Create Meshes

The mesh level representation is a low level (vertices, edges, etc.) description of the surface geometry. These meshes are constructed by the interpreter. For high level objects such as lofts, extrusions, etc., the actual mesh geometry doesn't even exist in the original applet file and only comes into existence when the applet runs.

Step 4. Display

Once the scene graph and the meshes have been constructed, they are displayed by the viewer, in this case, the Hypercosm Player.



Hypercosm Applet – Binary compressed representation of Hypercosm Bytecode instructions that need to be executed in order to generate 3d scene geometry, materials, etc.

Hypercosm Player

Scene Graph Construction – The scene graph is constructed as the applet is run according to the instructions contained in the Hypercosm Bytecodes.

Mesh Generation – Code in the applet instructs the Hypercosm Player to build meshes based upon primitives, modifiers etc. that are encoded in the applet.

Display – After the object and its materials and geometry have been constructed, it is displayed by the Hypercosm Player in the web page

Figure 2: The Decoding Process

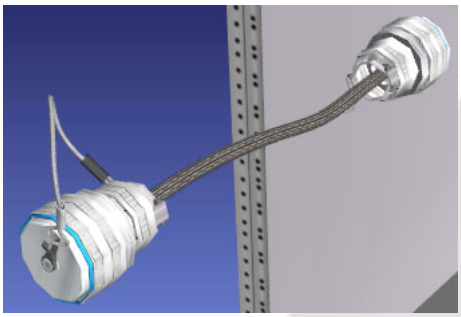
A Closer Look: The OMAR Script Language

To get an idea of what the OMAR script code looks like, the following page contains an example of a typical fragment of exported OMAR script code. From this code, you can see how the parameters that define objects are passed through the code to generate the geometry.

Figure 3: Sample Fragment of Exported OMAR Script Code

```
include "system/3d.ores";
include "system/3ds_translator.ores";


shape Pwr_Cbl01 with
        scalar tessellation = 1;
is
        translated_3ds_loft with
                path_steps = round (3 * tessellation);
                shape_steps = round (1 * tessellation);

                path is (new translated_3ds_line of [
                        translated_3ds_connected_spline through [
                                translated_3ds_smooth_vertex at <0.38353 -15.803 -1.926>
                                translated_3ds_smooth_vertex at <0.2513 -15.833 0.21628>
                                translated_3ds_smooth_vertex at <0.22166 -15.617 1.9216>
                                translated_3ds_smooth_vertex at <0.23124 -15.268 3.6237>
                                translated_3ds_smooth_vertex at <0.18649 -15.363 4.9775>
                                translated_3ds_smooth_vertex at <0.16111 -15.456 6.0127> ]
                        ]
                with
                        steps = round (6 * tessellation);
                        thickness = 1;
                        gen_mapping_coords is true;
                );

                cross_sections is [
                        (new translated_3ds_circle with
                                radius = 0.17168;
                                steps = round (6 * tessellation);
                                thickness = 1;
                        )

                ];

                path_offsets is [
                        0
                ];
        end;
end;
```

Figure 4: Power Cable Generated By The Script Code Shown in Figure 3



A Closer Look: The OMAR Bytecode Format

In the previous example, we demonstrated how parameter values are passed through various subroutines in order to generate the geometry that is eventually displayed in the scene. In this section, we will take a closer look at what happens to the code when it is compiled.

```
lid 1
fid 38
fid 4
nil
sca
lid 2
fml
vdp
sdr
sdr
lid 1
fid 38
fid 8
vli 1 0 0
fli -180
pro 0 1 0 16 1 0
trf 78
std 0 0
lid 1
nil
nil
std 0 0
lid 1
nil
scd 0 0
lid 2
nil
scd 0 0
lid 3
nil
scd 0 0
lid 4
nil
vcd 0 0
lid 5
nil
vcd 0 0
lid 8
nil
vcd 0 0
lid 11
```

Figure 4: Sample Fragment of Compiled OMAR Bytecodes

The compiled bytecodes are composed of a series of instructions followed by operands. In the bytecode assembly code, each type of instruction is represented by a three letter mnemonic. These text mnemonics are later converted into a binary form when the bytecodes are compressed. In the example shown in figure 4, we can identify a few instructions. For example, the instruction:

```
vli 1 0 0
```

is code to push a vector literal onto the operand stack. The next instruction:

```
fli -180
```

is code to push a floating point number onto the operand stack. By looking at the corresponding source code (not shown here), we can identify these instructions as a part of a procedure call to rotate an object by 180 degrees around the x axis.

In order to generate the geometry encoded by these instructions, it would be necessary to reverse engineer a virtual machine capable of executing these instructions in the exact same way as the Hypercosm Player. This would be a virtually impossible task, even if the complete set of information about the Hypercosm bytecode instruction set and the underlying virtual machine and 3d graphics subsystem and applet format were published, which is not the case.

Conclusion

By procedurally encoding 3d geometry into a series of low level instructions, we have simultaneously solved the problem of transmitting complex geometry in an efficient high level way and also solved the problem of finding an encoding mechanism that is virtually impossible to crack or to reverse engineer.