**Comp 4002 Winter 2010 Assignment #1**

**(File I/O and Building Worlds)**

**Due: Wed Jan 19, 2010 (2 weeks) in the assignment box BEFORE class.**

**Basic goals**: To familiarize yourself with the existing game builder and game engine software (by reading it yourself and by making some small extensions), and to familiarize yourself with the worldcraft editor (by designing a small world) and the universal converter to convert its mostly undecipherable “.map” format to a readable “.uni” format. There are four parts to the assignment… They can be done almost independently.

**Where to start…**

Download the software from the course site <http://www.scs.carleton.ca/~lalonde/comp4002> or indirectly from my site <http://www.scs.carleton.ca/~lalonde/comp4002>. Follow the installation instructions in the file “Assign1 Step by Step.doc”.

**The major software development work required**…

In general, the game builder reads a “.uni” file and creates a “.wrl” file with ALMOST the same format. The only difference is that the first line of the “.wrl” file starts with “World” instead of “Universal object”, there is an extra line of the form “"Start position: 0.0, 0.0, 0.0;”, and the “.wrl” file does not contain “objects with sub-objects”… The routines to look at are called **read** and **write** (which create streams) and then **import** and **export** (which use streams). They are written in an object-oriented fashion; e.g., a world **import** method uses object **import** methods which in turn uses face **import** methods, etc.

The game engine by contrast is missing all its **import** routines although it does have the **read** routine (which currently does not invoke an **import** routine). It doesn’t need **write** and **export** routines, of course. It wouldn’t be difficult for you to COPY the builder’s import routines and make the small change required to deal with the fact that the first two lines are different and sub-objects are gone. If you did that, you would agree that the “.wrl” file format would be ALMOST identical to the “.uni” file format.

But we want to make 2 major changes to the format: (1) we want to move the information about an object’s type which is currently in the property section of an object to the beginning of the object and (2) we want to summarize all the textures used in the world at the beginning of the “.wrl” file.

To do that, you will have to change the builder’s export routines and this will in turn require corresponding changes to the import routines in the game engine. Keep this in mind as you read the next 4 sections. We also want to add some objects that can be used to implement windmills and elevators that turn and move autonomously so that our worlds appear more “alive”. The new classes we build will need a **tick** method to compute the motion changes and a **draw** method to draw the objects.

**1. Extend the Worldcraft “student.fgd” file to support 2 new kinds of objects…**

Rotator

name

axis (use default “0.0 1.0 0.0”; the y-axis)

rateInDegreesPerSecond (use default “90.0” degrees per second)

center (use default “0.0 0.0 0.0”; a position offset)

triggerable (use default 0 for false, 1 for true; not for this assignment)

angleWhenTis0 (use default 0.0 degrees; not for this assignment)

angleWhenTis1 (use default -140.0 degrees; not for this assignment)

Translator

name

offset (use default “0.0 1.0 0.0”; up 1 meter)

rateInMetersPerSecond (use default “1” meter per second)

triggerable (use default 0 for false, 1 for true; not for this assignment)

Use notepad to look at the following 3 files (search for “vehicle” in each file):

1. file “student.fgd” in directory “StudentFGDandWAD” (this is read in by Worldcraft),
2. file “SimpleRoom.map” in directory “Maps” (this is a file exported by Worldcraft), and
3. file “SimpleRoom.uni” also in directory “Maps” (created by the universal converter from (b)).

File “student.fgd” contains a description of a Vehicle class, file “SimpleRoom.map” contains 2 vehicle entities that were created in the Worldcraft editor, and file “SimpleRoom.uni” is the converted result. Notice that property information always comes through as 2 pairs (both strings), a property name and a property value. Also

Property name “**classname**” in (b) comes through as “**type**” in (c) and the value is converted  
 to LOWERCASE; e.g., “**Rotator**” comes through as “**rotator**”,

Property name “**targetname**” in (b) comes through as “**name**" in (c),

Every other property name like “**familyName**” comes through in LOWERCASE; e.g., “**familyname**”.

Every other property value comes through UNCHANGED.

You will also notice that there are currently 2 string variables for class Vehicle and 1 choice variable. The value that is associated with a string variable is the string you type when you browse the Vehicle entity in Worldcraft. The value associated with a choice variable is “0”, “1”, or “2”, … as opposed to the string “Car”, “Bike”, or “Tank” that actually displays in Worldcraft. So it’s not as convenient when it comes time for you to interpret this value in the builder or engine but it does give the artist the information he needs to choose a proper value.

Use the Vehicle class definition as a template to create corresponding Rotator and Translator definitions in the “student.fgd” file and save it. You will also be creating a **Rotator** class and a **Translator** class with corresponding instance variables in **the engine but not in the builder**... though you can delay doing this until much later…

**2. Extend the builder and engine to support rotators and translators**…

A major distinction between the builder and game engine **import** routines is that the game engine DOES NOT AND SHOULD NOT create property collections for the information it reads in (such collections are too COMPLEX, too SLOW, and UNNECESSARY). In the builder, we don’t care but in the game engine we do because we want it to use minimal space and run maximally fast. In the game engine, the property information that is passed through should be placed directly in variables associated with the appropriate type of object it is associated with rather than placed in a “slow to access” property dictionary.

For example, when a rotator object is read into the game engine, a rotator instance needs to be created and the string associated with the “**name**” property needs to be stored in its “**name**” instance variable (NOT IN A PROPERTY DICTIONARY). The same would have to be done with the “axis” (a Point) and “rateInDegreesPerSecond” (a double) which in turn additionally need to have the property value that is read in converted from a string to a point or double respectively. Don’t forget that the **transformation** IS NEEDED and dictates where the object will go in the world…

Note also that not all properties are useful. We can be more selective… For “**type**” "static geometry", only the type needs to be passed through. If you copied the approach used in the builder for importing its data, then your engine would not know what kind of object to make until AFTER it read in all the property information and it would therefore have to temporarily store it somewhere. There is a better way.

One approach is to have your builder change the output format so that the “type” is not in the list of properties but rather output separately ahead of the other information. So your “.wrl” output format could look like

Object: 0

“type” => “static geometry”

Transformation:

…

Faces: 5;

Face: 0;

etc.

Object: 1

“type” => “rotator”

Transformation:

…

Properties: 3;

“name” => “windmill”

“axis” => “0.0 1.0 0.0”

“rateindegreespersecond” => “90.0”

Faces: 20;

Face: 0;

etc.

Object: 2

etc.

Notice that static geometry has no properties whereas rotators and translators do. In your engine, to process “Object: 0”, for example, you will first read in the type information and either create an instance of Object, Rotator, or Translator and have the created object read in the rest. This means, of course, that your world which used to maintain a collection of objects now also has to maintain a collection of rotators and a collection of translators. A better alternative is to create an AbstractObject class with classes Object, Rotator, and Translator (below AbstractObject in the hierarchy) and maintain a collection of abstract objects instead (this way, if you add new kinds of objects, you don’t have to keep adding new kinds of collections)…

# 3. Extend the builder/game software to simplify texture processing…

In your builder, extend the **export** routines that output “.wrl” files so that they output a list of texture names immediately after the “**Start position**” information of the world. To do that, you will have to make a pass through all the faces gathering all the unique names in some sort of collection. One possibility is to use a string collection or a string dictionary (search for StringCollection or StringDictionary in the source code). Possible sample format

Textures: 3

Name1

Name2

Name3

**Aside**: The start position is the starting location for the camera when your game starts. The builder already extracts this information from an object of type “**info\_player\_start**” (search for info\_player\_start in the builder’s source code to find it). There is a special icon in Worldcraft to create such an object when you build a world… You should always make one. Otherwise, your camera will automatically start at the origin which may very well be under your world rather than above it. Because the important information has already been extracted from such an object (there should only be one), you should avoid exporting objects of type “**info\_player\_start**”.

By making the builder provide a complete list of texture names that the faces refer to, you can make sure that in the game engine, ONLY ONE TEXTURE is created per texture name no matter how many references there are to that name. Also, a face, for example, might refer to a texture with a short name such as ‘brick’ but the corresponding texture will have to be read in from a full path name such as “..\textures\brick.tga”. Take a look at texture method “**readUnknownTexture**” in the GAME ENGINE that does this. Note that it uses relative path names to make sure your game works no matter what directory it is moved to. In general, tga files support transparency (32 bits per pixel) whereas bmp files do not (24 bits per pixel). The face DOES NOT indicate what type to read in… So the reader must try to read in a “.tga” file first and if that fails, then try a “.bmp” version. If that in turn fails, it gives an error message…

Also add code in the game engine so that it can draw worlds. Additionally, add a help key (the character ‘?’) that draws (on the screen) the list of keys available to the player. **Hints**: Your objects know how to log themselves (see how they do it in an object-oriented way; “draw” and “tick” are similar). The builder knows how to draw the twirling “idling” face (see how it does it). The game (engine) knows how to display its frame rate (see how it does it). Routine “main.cpp” controls all the keys that are pressed; look there for a complete list. While you’re at it, replace “Wilf says hi…” by your own name or remove it entirely…

**Details that are meant to help rather than confuse**: In order to be able to someday handle files from other tools like Maya, 3D studio, Softimage, a universal converter was written (all source code provided; there is no need for you to recompile it) that currently can convert (a) Worldcraft files, (b) Rhino files, (c) a failed attempt at handling Google 3D files, and (d) Universal Files (it’s own format) and then write them all out as universal files. Anyone is welcome (in fact, encouraged) to make additions.

Keep in mind that classes World, Object, Face, and GamePoint in the builder are similar to but not identical to the corresponding classes in the game engine. The Face class in the builder for example keeps track of a texture name (e.g., “brick”) to be used for drawing whereas in the game engine, it keeps track of a pointer to a texture (a much more complex object). GamePoint in the builder has normals nx, ny, nz but they have been removed in the game engine.

**4. Create a simple but interesting map (or level)…**

Create a simple map with 2 houses, a windmill, a waterwheel, an escalator that goes back and forth between the roofs of the two houses, and an elevator that goes up and down from the ground to the roof of one of the houses.

Hint: After creating a windmill and waterwheel, you need to convert each one to a rotator entity. Similarly, you need to convert the escalator and elevator to a translator entity… If your artistic abilities are really poor, at least create a box to represent each of these objects…

**How do you know when you are done?**

When you are done, you will have built your own “.map” file in Worldcraft, you will have converted it to a “.uni” file using the universal converter, you will have run your builder which will read the “.uni” file and converted it to a “.wrl” file, and you will have run your engine. When in your engine, you will be able to move around using the “WASD” keys, you will see what you built in Worldcraft, you will see the windmill and waterwheels turning, you will see the escalator and elevators moving back and forth, you will be able to ask for help using the “?” key, you will be able to quit without crashing, and you will be able to run the game twice in a row without exiting. If you can’t, it means you have a memory leak; i.e., some “new” is missing the corresponding “delete” or more likely, you are trying to delete the same object more than once…

Come to the game lab and say “I don’t know where to start”…

**Marking check list**:

1. Builder doesn’t crash when building.
2. Engine doesn’t crash when loading and quitting.
3. Engine can reload the same or another level without having to be restarted.
4. There are rotators that rotate.
5. There are translators that translate.
6. There are no property collections in the engine.
7. The help key works.
8. The software is still path relative (no hardwired file names).