WHAT MAKES TAMRIEL TICK
THE DATA STRUCTURES BEHIND
THE ELDER SCROLLS IV : OBLIVION
CHAPTER I - WORLD GENERATION

Introduction

Even if it was released more than 3 years ago, in March 2006, Oblivion still has one of the most immersive world in video games. This is not only due to its gorgeous graphics. A lot of work has been done on this episode to make the world more realistic, including day/night cycle, weather simulation, compelling physics engine, advanced "Radiant" AI. No need to say it's one of my reference games and as a game developer, and especially a RPG developer, I'm particularly interested in knowing how it is done. It's not always easy to guess how a game is done and reverse-engineering the data files or the compiled code is quite difficult. Fortunately, we don't have to do it because most of the data structures can be guessed through the editor released by Bethesda Softworks for the modding community : TES Construction Set.

Let me stress that this series of articles describes in no way the actual data structures used in Oblivion source code. These are just my understanding of how the game works. Feel free to report any mistake you might find.

I will use some vaguely javaish syntax to describe the data structures, but I won't use javac compilable code. For example, there are some places where I will use some imaginary intrinsic int16 type where I need 16 bit integers. I will also consider all fields public.

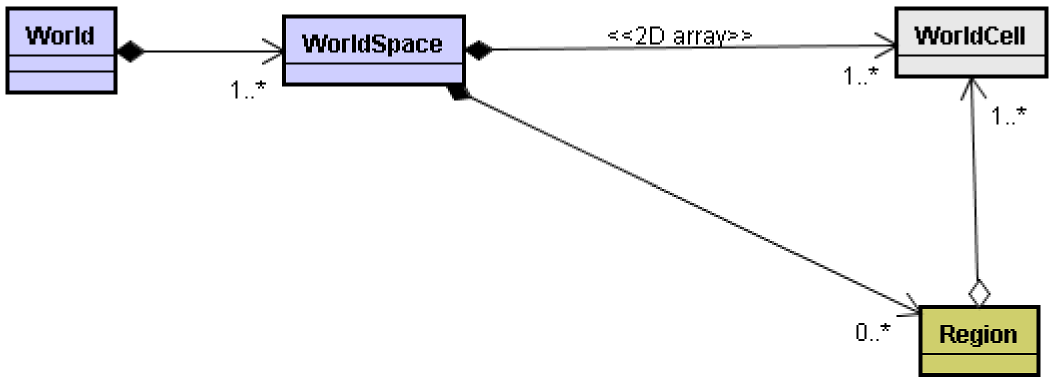
TES Construction Set

This is the almighty editor supposedly used by Bethesda designers to build Oblivion. In fact, it's probably a stripped down and polished version, but most of the game content can effectively be modified with this editor, including but not limited to :

- weather and climates
- world spaces, regions and cells (to be described in this article)
- races, classes, skills
- NPC, dialogs, quests
- items, spells,
- in-game scripts
- ...

In this chapter, the focus will be on the world map.

Oblivion's world is represented by a hierarchy of objects of increasing detail and decreasing size :



- the world contains at least one worldspace.
- each worldspace contains a 2D array of world cells.
- the worldspace can contain regions representing a part of the worldspace cells.

World spaces

Independent spaces, sub spaces

Oblivion's world is stored in a list of 2 levels trees of world spaces. Each tree root represent a complete independent world. You usually have only one in your game except if you want the player to be teleported in another world during the game. In Oblivion, you have Oblivion Gates worlds and some other independent worlds, like "the painted world" where you enter a magical painting to complete a quest.
All independent worlds use only a single world space in Oblivion. Only the main game world (the continent of Tamriel) uses a 2 levels tree. The root of the tree contains the complete continent. Each level 2 world space contains a detailed version of a part of the world. For example, when you look at a city from a far distance, you see only the low res version from the root world space. But when you get closer, the detailed world space for the city is loaded. It's a kind of very high level LOD mechanism.
In Oblivion, Tamriel contains 25 sub world spaces representing mainly the cities or some complex dungeons.

Ok, so long the data structure is quite simple. We only need some global list of world spaces to store all the game data.

```
class WorldSpace {
      String name;
      WorldSpace parent;
      ...
      static List<WorldSpace> list;
}
```

World space internal data

No let's see what's inside a world space.
First the world space defines the music that will be played when the player is inside it. This is not a single music but rather an intelligent playlist. For example, cities will have happy taverns musics while dungeon will have tense musics. Oblivion has only 3 "music types" : *public* (for cities), *dungeon* and *default*.
There are also a few obscure flags defining the world behaviour. The most important tells if this world space allows fast travel (traveling to a visited place in this worlspace by clicking on it on the world map).

The root world spaces (the one without parent) contains a few more very high level ambiance parameters :

- the world climate which defines the list of weathers, how the sky looks, sunrise and sunset hours and so on
- the water type (lava, standard water, swamp water, and so on) and height. These are the default water parameters for the worldspace but you'll be able to override them for each cell.
- the in-game map of this world. This map can be displayed by the player to find his bearing or to use fast travel when it's possible. The map is defined by a simple image and information about how to map the world space to this image.

A part of Tamriel's map

Ok, our WorldSpace class has grown up :

```
class WorldSpace {
      String name;
      WorldSpace parent;
      int musicType;
      boolean canFastTravel;
      int waterType;
      int waterHeight;
      Image worldMap;
      int topLeftCellX;
      int topLeftCellY;
      int bottomRightCellX;
      int bottomRightCellY;
      ...
}
```

Cells in space

You can see that the image is not mapped to world coordinates, but rather world *cell* numbers. The world space is indeed subdivided in cells. A cell is a squared part of the world space. In Oblivion, each cell is said to be 192x192 feet (about 58x58 meters). The Tamriel world space maps to cells coordinates -59,-58 to 59,47, or 119 x 106 cells => 12614 cells. This represent a map of 4.3x3.8 miles (16 square miles) or about 7x6 km (42 square kilometers).

Each cells represents 4096x4096 world units. That means the best coordinate precision in Oblivion is 58 meter / 4096 = 1,416 centimeter (0.5 inch). In the Tamriel world space, the coordinates 0,0 correspond to the top-left corner of cell 0,0 which is broadly the center of the continent. Coordinates can range from -241664,-237568 to 241664,192512. We definitely need 32 bits integers to store coordinates.

Ok enough numbers. Let's add a 2D array of cells to the world space. The WorldCell class will be defined later.

```
class WorldSpace {
      ...
      int nbCellsHoriz;
      int nbCellsVertic;
      WorldCell [ ] [ ] cells;
      ...
}
```

Heightmap

The first obivous thing to do when building a world is the heightmap. It will define where the coastlines are and give some basic data about the terrain types. The heightmap is stored in 16 1024x1024 16 bits grayscale textures. Each world cell uses a 32x32 part of a texture. Thus, a pixel from the heightmap represent a portion of 6x6 feet (1.8x1.8 meter). Of course, the engine interpolates the heights. The complete Tamriel heightmap requires 3808x 3392 pixels. Even with 16 bits precision, it still uses almost 25 megabytes of memory !

A part of Tamriel's heightmap (compare with the map above)

16 bits value gives a range of heights from 0 to 65536. Oblivion uses 4096 as default water level which means the highest mountain can have a height of 65536-4096 = 61440. One heightmap unit represents 2 units in the world cells. Remember the world cell unit represents 1.416 centimeter. Our biggest mountain's height is then : 61440 * 1.416 * 2 = 1739 meters. Not mount Everest, but high enough for a video game not dedicated to mountaineering. Moreover, having 4096 units (58x2 meters) for underwater is enough for any RPG except if it involves submarines !

As for the world map, the heightmap is only defined for root world spaces :

```
class WorldSpace {
      ...
      int16 [] [] heightmap;
      ...
}
```

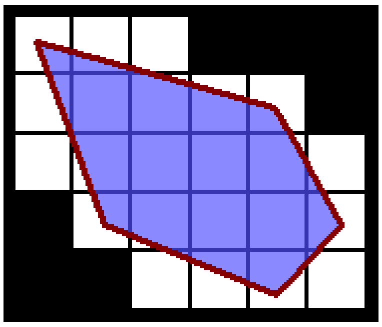
We could store the size of the heightmap but it can be easily computed :

```
heightmap width = nbCellsHoriz * 32
heightmap height = nbCellsVertic * 32
```

To sum up :
```
class WorldSpace {
      String name;
      WorldSpace parent;
      int musicType;
      boolean canFastTravel;
      Image worldMap;
      int topLeftCellX;
      int topLeftCellY;
      int bottomRightCellX;
      int bottomRightCellY;
      int nbCellsHoriz;
      int nbCellsVertic;
      WorldCell [] [] cells;
      int16 [] [] heightmap;
      static List<WorldSpace> list;
}
```

Regions

It's time to give some personality to the world cells. Instead of doing it one cell after another, Oblivion uses cells groups called *Regions*. A region is a polygonal part of the world. The polygon vertices are defined using the world units (4096x4096 for each cell).


Definition of a region boundaries

On this picture, the red lines represent the region boundaries. The blue surface is the region. The white squares are the world cells associated with this region.

```
class Region {
      int nbVertices;
      List<Point> vertices;
      ...
}
```

Point is a simple class with two x,y 32 bits integer fields.

Regions are used to generate procedural content. Note that a cell can be affected by more than one regions. Regions can even overlap. To keep procedural data from changing abruptly at region borders, the region defines a edge fall-off value. This represents the distance (in world units) from the region border where the region influence reaches 100%. For example, a tree generating region with a edge fall-off of 1000 (= 14 meters) will have its full tree density at 14 meters from the edge and a tree density of 0 at the edge. The density is linearly interpolated between those two values.

```
class Region {
      ...
      int edgeFallOff;
      ...
}
```

In Oblivion, there's a special region defining the border of the world space. It's an invisible fence around the world. I won't use it because I think it's better to put natural fences like cliffs or deep see rather than being suddenly stopped.by an invisible fence.

Regional objects

The first use of a region is to place some random objects on the map. This is mostly used for trees, rocks and hair in wilderness but it can be used for anything else, like ground textures. For each object type, you define some parameters :

- a density (the higher, the more objects there are)
- the min and max slope. For example, you don't want trees and grass on cliffs.
- the min and max height. For example, no trees in high mountains.
- the object radius : to keep objects from overlapping
- clustering factor : how much the object will shift to the closer object of the same type.

```
class RegionObject {
      int objectType;
      int density;
      int minSlope;
      int maxSlope;
      int minHeight;
      int maxHeight;
      int radius;
      int clustering;
}

class Region {
      ...
      List<RegionObject> objects;
      ...
}
```

Regional weather

The region can also alter its weather. It simply define which weather can occur and with which chance (1-100%).

```
class RegionWeather {
      int weatherType;
      int chance;
};

class Region {
      ...
      List<RegionWeather> weathers;
      ...
}
```

Regional sounds

A forest doesn't sound like a plain. You can define which background sounds will play in this region. This is pretty similar to the regional weather except that you can have sounds dependent on the weather. This part is quite basic and could be improved using an Atmosphere Deluxe approach (google for it and try the demo!). The region can also override the world space music type.

```
class RegionSound {
      int soundId;
      int chance;
      boolean whenPleasantWeather;
      boolean whenCloudyWeather;
      boolean whenRainyWeather;
      boolean whenSnowyWeather;
};

class Region {
      ...
      int musicType;
      List<RegionSound> sounds;
      ...
}
```

To sum up :

```
class Region {
      int nbVertices;
      List<Point> vertices;
      int edgeFallOff;
      List<RegionObject> objects;
      List<RegionWeather> weathers;
      int musicType;
      List<RegionSound> sounds;
}
```

World cells

The world cell is where the data displayed on the screen is stored. First, you can override some world space value like water type & height or music or the fast travel flag. This makes it possible to have a small mountain lake or a lava pit. Note that you cannot have a water lake and a lava pit on the same cell but since the cells are only 58 meters wide, this should not be an issue.

```
class WorldCell {
      int waterHeight;
      int waterType;
      int musicType;
      boolean canFastTravel;
      ...
}
```

Eventually, the cells contains the game objects displayed on screen. Some of those objects are generated procedurally from the regions overlapping the cell. In Oblivion, the generation is done in the editing phase. Then, generated object can be moved and altered by hand. In a RPG using a random world, you could generate the procedural objects at runtime during the initialization of the game. You can of course add objects by hand in the cell. Most cities and dungeons in Oblivion are completely hand crafted and use very few procedural content. The list of objects types is quite long :

- Actors (creature, NPC)
- Items
- Land textures
- Sounds
- Containers (barrels, furniture, ...)
- Doors
- Flora, grass and trees
- Lights
- Any other 3D scenery props
- Scripts triggers (called activators)

I won't detail all these objects in this article, but they all share at least a position in the world cell (in absolute world units) :

```
class WorldObject {
      int x,y,z;
}

class WorldCell {
      ...
      List<WorldObject> objects;
      ...
}
```

To sum up :

```
class WorldCell {
      int waterHeight;
      int waterType;
      int musicType;
      boolean canFastTravel;
      List<WorldObject> objects;
}
```

Appendice

Useful links

The best resource on TES : CS

http://cs.elderscrolls.com/constwiki/index.php/Main_Page

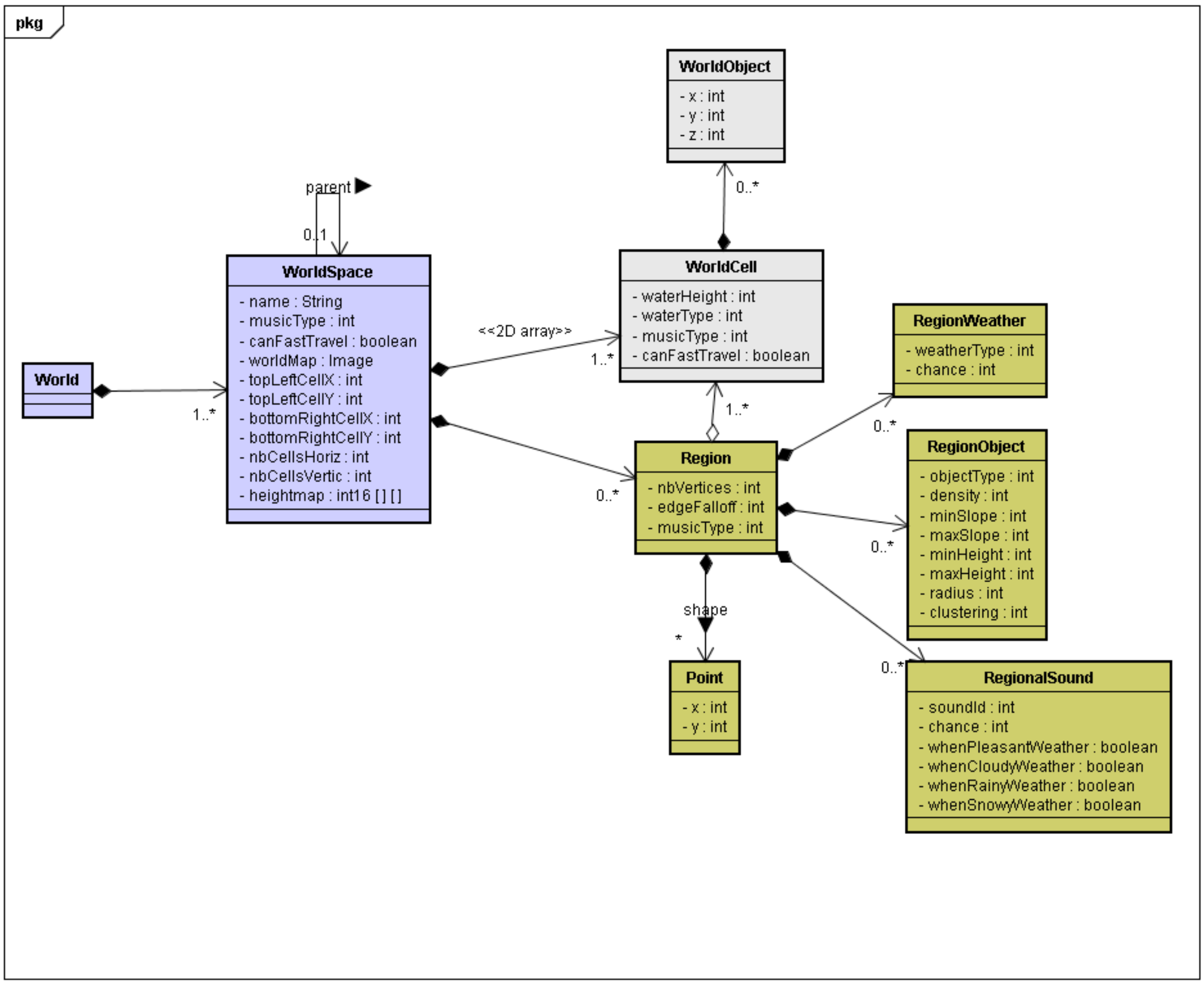
Full UML diagram

For UML fans ... :)

In blue : classes related to the world.

In green : classes related to a region.

In gray : classs related to a cell.


pkg
WorldObject
- x : int
- y : int
- z : int
0..*
parent
0..1
WorldSpace
- name : String
- musicType : int
- canFastTravel : boolean
- worldMap : Image
- topLeftCellX : int
- topLeftCellY : int
- bottomRightCellX : int
- bottomRightCellY : int
- nbCellsHoriz : int
- nbCellsVertic : int
- heightmap : int16 [] []
World
1..*
<<2D array>>
1..*
WorldCell
- waterHeight : int
- waterType : int
- musicType : int
- canFastTravel : boolean
1..*
RegionWeather
- weatherType : int
- chance : int
0..*
Region
- nbVertices : int
- edgeFalloff : int
- musicType : int
0..*
RegionObject
- objectType : int
- density : int
- minSlope : int
- maxSlope : int
- minHeight : int
- maxHeight : int
- radius : int
- clustering : int
0..*
shape
*
Point
- x : int
- y : int
0..*
RegionalSound
- soundId : int
- chance : int
- whenPleasantWeather : boolean
- whenCloudyWeather : boolean
- whenRainyWeather : boolean
- whenSnowyWeather : boolean