**By David Michael**

**Tile/Map-Based Game Techniques: Base Data Structures**  
**by** [**David Michael**](http://www.samugames.com)

This article is the first in what will be a short series of articles about Tile/Map-Based Games Techniques. This first article will cover the primary data structures of such a game: the main map structure and the unit structures.

My primary assumption is that the "look" you are wanting is similar to Command & Conquer, WarCraft, Ultima Online, and so on, either in "straight overhead" or isometric representation. My secondary assumption is that you already have at least the basic concept for your game. These articles will not cover game design in any real depth. Instead we will be focusing on implementation issues.

Data structure and coding examples will mostly be taken from my currrent project, Artifact. Artifact is an Internet-based, client-server, multi-player, persistent-world, real-time strategy game. Artifact is currently in pre-release, with full release scheduled for Fall 1999. For more information and screen shots of the game, check out the Artifact web page: <http://www.samugames.com/artifact>

**The Map**

The map is probably the single most important data structure in a game of this type. The map serves as the central repository of nearly all game data and is the primary source of information used by the game logic. The various units, "fog of war", unit AI, and so on, all rely extensively on the map.

The most basic structure of the map is the **map cell**. A map cell is simply a single map location. The information you store in a map cell depends on the game you're creating, but there are a few common elements. Such common elements include an indicator of the type of map cell (for instance, the terrain), the structure (or structure segment) built on the cell, and a pointer to the first mobile unit on the map.

Artifact's map cell structure is shown in Figure 1. It's not the most space-efficient structure, but since it's used on the server side where plenty of RAM is available, that's not an issue.

**Figure 1: Artifact's Map Cell**

struct world\_struct

{/\* map cell \*/

unsigned char reg;

TRP\_TYPE \*trp;

long ore,crops,wood,stone; /\* raw materials \*/

char structure; /\* bdg is a: 0 city, 1 facility, 2 artifact STR\_\* \*/

BDG\_TYPE bdg; /\* pointer to union of city or facility \*/

};

Once you have your map cell defined, you have to decide how you want to structure the entire map. Here we have to refer back to your game design concept and how you want the finished game to look. There are 3 primary "looks" that are popular: straight overhead, angled isometric, and layered isometric. WarCraft is a good example of a straight overhead map, with Railroad Tycoon 2 and Civilization 2 demonstrating angled isometric and layered isometric respectively.

The straight overhead map and angled isometric map have the simplest possible scenario: a two-dimensional array of map cells. The layered isometric map can also use a two-dimensional array, but there are several complexities. The biggest difference concerns adjacencies in the map.

For overhead and angled isometric maps, adjacent cells are easily determined. If a map cell is at location (x,y), then it's adjacent cells are as seen in Table 1. Horizontal (east-west or left-right) and vertical (north-south or up-down) "wrapping" on the map are also very simple.

**Table 1: Overhead and Angled Isometric Adjacency**

|  |  |
| --- | --- |
| **Direction** | **Location** |
| North | (x,y-1) |
| Northeast | (x+1,y-1) |
| East | (x+1,y) |
| Southeast | (x+1,y+1) |
| South | (x,y+1) |
| Southwest | (x-1,y+1) |
| West | (x-1,y) |
| Northwest | (x-1,y-1) |

Layered isometric maps, however, are significantly more complex. This is because, while the map is still a two dimensional array, the adjacencies work differently because of their on-screen representation. How adjacency is calculated varies according to whether the map cell is on an even-numbered row or an odd-numered row. See Table 2 below. Also, while the X dimension can be either even or odd, if the map is to wrap vertically then the Y dimension must be an even number. This is so that bottom and top edges "mesh" correctly.

**Table 2: Layered Isometric Adjacency**

|  |  |
| --- | --- |
| **Direction** | **Location** |
| North | (x,y-2) |
| Northeast | (x,y-1) (even rows) (x+1,y-1) (odd rows) |
| East | (x+1,y) |
| Southeast | (x,y+1) (even rows) (x+1,y+1) (odd rows) |
| South | (x,y+2) |
| Southwest | (x-1,y+1) (even rows)  (x,y+1) (odd rows) |
| West | (x-1,y) |
| Northwest | (x-1,y-1) (even rows) (x,y-1) (odd rows) |

even rows = 0, 2, 4, 6, 8, ...  
odd rows = 1, 2, 3, 4, 5, ...

Figure 2 shows a small layered isometric grid, with the cells labelled (col,row) to show their position in the map array. You can use Figure 2 to verify that the adjacency calculations in Table 2 are correct. Cell "A" is on an even numbered row and cell "B" is on an odd numbered row.

**Figure 2: Layered Isometric Grid Adjacencies**

In my projects so far, we've always used a straight overhead map. We considered using a layered isometric map structure for Artifact, but decided not to because of the increased complexity. However, the layed isometric view is becoming more and more the "standard" for this kind of game.

Rendering these maps to the screen, whether overhead or isometric, is a subject for another article. Right now we're simply buildng the necessary data structures.

**Unit Data Structures**

Once you have the map structure determined, you can move on to your unit data structures. There are 2 primary types of unit: mobile and stationary. Mobile units are the warriors, tanks, aircraft, and so on, that actually move around the map. The stationary units, on the other hand, are the buildings or building segments that do not move.

The common elements of mobile and stationary units include an identifier, a map location, an owner, and so on. These common elements may allow you to use object-oriented techniques and create a common ancestor class for both kind of units, but that depends on the game and game logic needs.

Stationary units generally do not move around the map. Once a stationary unit is placed on the map, it stays there until the player removes it or an enemy unit destroys it. This can simplify how you handle stationary units. Since there can only be one stationary unit per map cell, the map cell needs only to have a pointer to such a unit. If the pointer is nil (or null), then there is no stationary unit occupying that map cell.

Mobile units, unlike stationary units, can move from one map location to another. They also have another significant difference from stationary units: there is technically no limit to how many mobile units can be in a single map location. Thus, you will need to maintain a list of all mobile units per map cell. Depending on your game, it's possible you may want to limit mobile units so that they cannot "stack" in a single location. Or you may want to allow stacking only in locations with a special stationary unit (like a barracks). In this case, the map cell no longer needs to maintain a list, though the special stationary unit would.

These lists I keep mentioning for mobile units in the same map location or "inside" the same stationary unit, do not need to be incredibly complex. A simple linked list would likely work fine, though you might want to include some form of sorting to support game logic such as determining which unit in a location takes damage first or which units can "leave" first. To further simplify the lists, you could embed the list information (usually just a simple "next" pointer) in the actual unit data structure. If the unit can simultaneously be in multiple lists, however, you may want to use an actual list "container" class.

It's important, however, no matter how many different lists a particular unit is in, that units be stored in a centralized data structure. A one-dimensional array works quite well for this and even allows for certain performance benefits when processing all units of a particular type. The unit's position in the array also provides a very handy identifier for that unit. Thus, the "57th Infantry" unit could be found in array location 57. Additionally, if you're in a crunch for RAM, a 16-bit ID/array index takes up half the space of a 32-bit pointer to a structure.

You may have noticed that the unit data structures include the map location, and that the map cells have a list of units in those cells. This may seem redundant, but is nearly always necessary. If you are processing the map cell by cell, you will need a fast way to know which units are in each cell. Conversely, if you are processing all units you will need to know their location without searching the entire map. This puts a mild burden on the programmer to make sure that the units always have the correct location and that the map cells they travel through are correctly updated.

To provide a more concrete example of a unit data structure, Figure 3 shows a cut-down version of the troop structure used in Artifact. As you can see from Figure 3, a single troop can be in more than one list at a time, with the actual list information maintained in the troop data structure. The ID of a troop is simply its position in the single-dimensioned "Troops" array. The ID is stored in the troop structure even though it equals the array index because often it's necessary to know the troop's ID and you only have a pointer to the troop.

**Figure 3: Artifact's Troop Structure**

struct troop\_struct

{

int id;

int owner\_id;

short int x,y;

...

int count; /\* number of members of this troop \*/

short int morale; /\* 0=worst, 255=best \*/

short int training;

short int experience;

short int fatigue; /\* 255=worst, 0=best \*/

...

TRP\_TYPE \*next,\*prev; /\* linked list for troop update \*/

TRP\_TYPE \*next\_here,\*prev\_here; /\* linked list of troops at an x,y \*/

TRP\_TYPE \*btl\_next,\*btl\_prev; /\* linked list for battles \*/

...

};

**RAM Considerations**

By now you should have a good idea of what you require for your map cell data structure and unit data structures, and how you're going to store them. Now we do a quick overview of much RAM this is all going to take.

Your map is probably going to require the most RAM. Even a relatively small map can require a suprisingly large amount of RAM. A 100x100 map requires 10,000 map cells. If each map cell is 20 bytes, that's 200K. That's not so bad, but once you move your map to 500x250 you suddenly have 125,000 map cells at 20 bytes each using up 2.5MB of RAM. And if those maps seem small to you and what you really want is something like a 1,000x1,000 map for your mega-massive C&C clone, you'll need 20MB of RAM just for the map! For that reason, you should consider making your map cell data structure as small as you possibly can, even it costs you a minor performance hit. For instance, if a particular type of data only requires 3-bits of information try to use a bitfield (or least only 1 8-bit byte or unsigned char) instead of just allocating an "int" (32-bits) to hold it. If you simply cannot make the map cells any smaller, then you're going to need to accept a smaller map size or figure out some sort of "paging" mechanism so that you don't need the whole map in memory all at once (though you might be able to rely on virtual memory if you don't mind the performance hit).

But the map isn't the only thing you need to have in memory. There are all those units. You need room for them, as well. Unit data structures are generally going to be larger than map cells, but you're going to need far fewer of them unless you expect your players to put one or more units on every map cell. Several hundred per unit type, probably not more than several thousand, should be sufficient.

As you consider the RAM your base data structures will require, you may find that you have to do a bit of redesign work on your game. If you've set your sights on a huge map, you might want to scale down to something for reasonable. Or you might have been overestimating your needs and can now afford to scale up.

In our specific case, Artifact uses an 800x400 map (wraps horizontally), with a maximum of 35,000 facilities, 10,000 troops, and 1,000 cities. Fortunately, Artifact is a client-server game so the client software running on the player's PC generally doesn't have to deal with all of this information. The server segments the main map into "groups" and only sends those groups to the player that she has units on. Also, the information that's necessary on the client side (for rendering the map display and so on) is significantly less than is required on the server side where the actual game logic resides. Even still, the Artifact client software uses about 12MB-14MB RAM for all internal data structures and graphics and sound data.

**Conclusion**

Hopefully this article helps you understand the basic data needs of a game like Command & Conquer, WarCraft, or Civilization. The map is going to be your "central" data structure, with your arrays of unit data structures providing the necessary details.

Copyright © 1999 by David Michael. All Rights Reserved.