Modular Tank Simulation

Andy Oxfeld

June 7, 2004

Abstract

The objective of this project is to create a tank game in a modular fashion. The game will be a simple one in which you drive around using the keyboard and the mouse, shooting at other tanks. The project will be divided into several reusable modules.

This project will accomplish several things. First, it will create a game that will be fun to play. Second, since it will be coded in a modular fashion, parts of the code can easily be re-used for future games, or even non-game programs. The code will use sophisticated programming techniques that will allow for easy debugging, customization, and reusability.

Introduction and Background

The project will consist of programming individual modules. Once each module is finished and tested, I will write code to tie that module to the other modules, and then I will then begin work on the next module.

The only resource I will need is a computer to program and test the program on. No other materials will be needed. The languages used will be C++ and XML. The only data that will be needed will be randomly generated (such as terrain).

The first module will be a data storage module. The data will be stored in XML format. The module will contain code to load and manipulate this data. It will also contain data editor software that will allow the game programmer to easily organize and edit data.

The second part will be a 3D engine. Due to limitations in time, this engine will obviously be far simpler than the engines used in commercial games. Its job will be to render the models, the interface, and the terrain.

The third module will be a random terrain generator. This will generate a random landscape for the game.

The fourth module will be the movement engine. This engine will accept keyboard and mouse input and will move the user's tank accordingly.

The fifth and last module will be the main menu module. This module will display the pre-game menus that allow you to configure options, create characters, and maybe do things such as design custom tanks, etc.

Expected Results

The expected result will be a game that is fun to play. The result will also consist of reusable programming components which can be used for future projects in a fairly "plug and play" style.

The final results will be presented using screenshots from the game and benchmarks showing how long various processes take to complete.

The work will focus on programming techniques, random terrain generation, and input handling. If I have extra time, I will also expand into user interface handling and data editing software (which will let me create maps and change settings). The main limitation of this project will be time.

Experimentation with Programming Techniques

My first research was into programming techniques. Having chosen the language C++ to write my program in, I decided to experiment with more advanced C++ techniques than I have used before.

Duffs Device

The first experiment I tried was something called a Duff's Device. This is a way of writing faster code. It does something called "unrolling a loop". Duff's Device is a specialized type of loop unrolling.

Basic loop unrolling is simple. A loop, prior to unrolling, might look like this:

for (int i = 0; i ¡ 100; i++) dosomething();

That code calls the dosomething function 100 times. However, it also executes the i++ instruction 100 times too. You could optimize this by getting rid of the loop and just writing dosomething() 100 times, so you don't even need the i variable. However, that would be very tedious. Instead, you could write something like this:

```
for (int i = 0; i < 25; i++)

{

  dosomething(); dosomething(); dosomething(); dosomething();

}
```

This still calls dosomething 100 times, but only executes the i++ instruction 25 times. Thus, the i++ instruction has been optimized. This is a 4x loop unrolling.

The problem is when you want to execute something a variable number of times. The number might not be evenly divisible. For example, if in the above example, if I wanted to call dosomething 101 times instead of 100, I could not evenly divide 101 by 4. So, the code would have to call dosomething an extra time before it went into the loop.

Duff's device is an easy and fast way of overcoming this difficulty. It uses a switch statement to jump right into the middle of the loop. This way, it executes a part of the loop once, and then the rest of the unrolled loop. This is an example of 8x unrolling using Duff's Device:

```
int count = 101;

int n = (count + 7) / 8;

switch (count % 8)

{

  case 0: do { dosomething();
  case 7:      dosomething();

  case 6:      dosomething();

  case 5:      dosomething();

  case 4:      dosomething();

  case 3:      dosomething();

  case 2:      dosomething();

  case 1:      dosomething();

             } while (--n > 0);

}
```

This calls dosomething 101 times with 8x unrolling. It takes advantage of the ability to have a switch command in C++ without a break. Thus the execution falls through the entire switch statement.

Unions

There is a little-known C++ keyword called union. This looks similar to a struct, but it has one important difference.

The size of the struct is the size of the all of the items in it combined. However, the size of a union is the size of the largest item in it. This is because, in a union, all of its members share the same storage space.

The way I use unions is in my XML data module. As I will describe later, the XML file contains different tags. Different tags need different data. I could create a different struct for each tag (container, integer, string, etc.) However, instead, I created one generic struct that handles every tag. In it is a union for each type of tag. This way, the struct contains all of the data needed for EVERY type of tag. However, it only is the size of the largest possible tag. This saves a lot of memory.

An example of the usage of the union method in the XML parser:

```
union
{
  struct
  {
    TYPE_DATANODE** children;       // Array of pointers to children
    unsigned long numchildren;      // Num of children
  } dncontainer;
  struct
  {
    unsigned long* numberarray;     // Array of numbers
    unsigned long numnumbers;       // Num of numbers
  } dninteger;
}
```

Memory Allocation

I also experimented with memory allocation. C++ offers two ways of doing memory allocation. The first uses the functions malloc and free and is backwards-compatible with C. The second uses the operators new and delete and is not backwards-compatible with C.

Using defines, I abstracted all memory allocation so that I could use either method. This way, I can change one value, and it will cause the entire program to use a different method. This will allow me to experiment and find out which method works better for my program.

I also experimented with implementing a third type of allocation, which uses java-style ”smart pointers”. This type of allocation would detect any attempts to address unallocated memory and would report any memory that is ”leaked” (not freed). This way, during debug I could have a safe java-style mode, but when I was ready to release the program, I could switch to a faster mode. However, while I implemented this method successfully, I found the difficulties in using it were too great to make it reasonable, so I removed it.

The following are the defines used to implement the system:

```
#if defined(CPLUSPLUS_MEMORY_ALLOCATION)

#define ALLOCATEVARIABLE(type) new type

#define ALLOCATEARRAY(type, size) new type[(size)]

#define FREEVARIABLE(variable) delete variable

#define FREEARRAY(variable) delete [] variable

#else

#define ALLOCATEVARIABLE(type) ((type*)malloc(sizeof(type)))
```

```
#define ALLOCATEARRAY(type, size) ((type*)malloc(sizeof(type) * (size)))

#define FREEVARIABLE(variable) free(variable)

#define FREEARRAY(variable) free(variable)

#endif // defined(CPLUSPLUS_MEMORY_ALLOCATION)
```

XML Parser

After experimenting with programming techniques, I began to research XML for writing my data module. I decided on XML because it has a very nice tree-structure which is what I was looking for. It is also a format I was familiar with because of my experience with the TJ website.

I structured my XML parser as a recursive function. The function reads in a tag, then recurses multiple times to read in any children of that tag. I used a variety of file reading techniques, such as buffering, which is reading ahead multiple characters, and parsing, which performs a different action depending on what character has been read.

The core of the XML parser was basic brute-force code to extract data and store it in the data structure. The most difficult part was getting the recursion to work correctly with end tags and data storage. That is, at first the parser has to assume the end tag is actually a new tag; but then when it realizes it is not a new tag but is the end of an earlier tag, it must handle that appropriately.

I organized my data structure as a spanning tree. Here is the part of the structure that contains the tree:

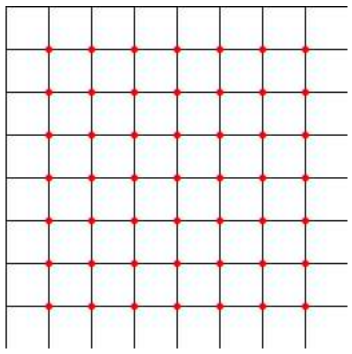
```
typedef struct TYPE_DATANODE
{
  NODE_TYPES type;                    // Type of node
  TYPE_DATANODE* parent;              // Pointer to parent node
  char* name;                         // Name of this node
  unsigned long uniqueid;             // Unique ID representing this node
  unsigned long fileoffset;           // Offset within data file
}
```

The rest of the data structure contains the data for each type of node. For example, integers store a number, strings store a string, etc. It uses the union method described above for greater efficiency.

Random Terrain Generation

Heightmaps

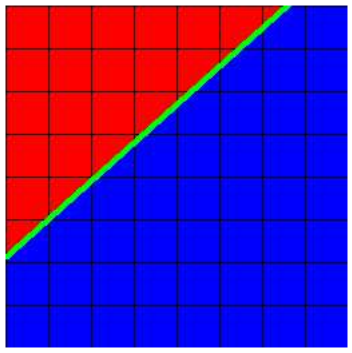
The most common method of describing terrain is using a heightmap. A heightmap starts as a flat grid of squares. Each point on the heightmap is then assigned a height, making it no longer flat. This is an easy and efficient method of describing terrain; only the heights at each point need to be stored. The terrain is regular and easy to render.



In this picture of a heightmap, the red points are the points which each will have a height assigned to them.

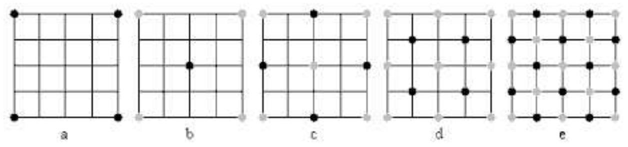
Fault Formation

Now that we have a way of describing terrain, we still need a way of generating the heights of each point. One method of doing that is fault formation. The fault formation method starts with a flat heightmap. Then, it draws a random line across the heightmap. All points on one side of the line are raised, and all points on the other are lowered. That line-drawing process is repeated a number of times, and then a smoothing algorithm is run to clean up the results. In this picture, the green line is a fault. The points in the red area will be raised and the points in the blue area will be lowered.



Diamond-Square Algorithm

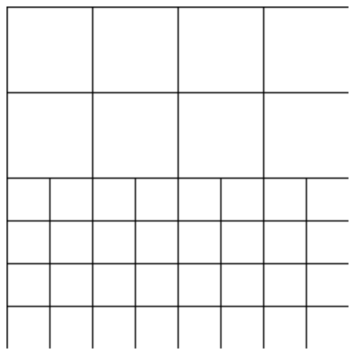
An alternate to fault formation is the diamond-square algorithm. This is a fractal algorithm that generates a terrain through subdivision. It starts by describing the entire world as one huge square. It then divides this one huge square into four squares (a 2x2 grid), adjusting the height of the points of the new squares. Each of the new squares then undergoes the same process, being each divided into 4 new ones. Each of those are again divided. The farther into the process, the smaller the height adjustments become; it becomes more fine-tuning. This process continues a set number of times until a fairly smooth terrain is generated.



This picture shows the two steps to the diamond-square process. The first step, the diamond step, starts with a square and averages the 4 corners with a random perturbation to produce a height for the point in the middle of the square. When looking at multiple of these squares side-by-side, this results in diamonds. The square step then occurs: each corner of the diamond is averaged with a perturbation to produce the height of the center of the diamond, which results in squares.

Terrain Rendering

Once a heightmap has been generated, to render it, the squares can simply be sent to the renderer, in this case OpenGL. That is called a brute-force method. It is high quality but is not the fastest method. Instead, a method called continuous level of detail, CLOD, uses adaptive degradation so that less detail is used for squares far away from the viewer. This degradation is hard to notice and puts a lot less strain on the video card, although it adds a little strain to the CPU.



As you can see from the picture, the heightmap density is larger for squares farther away from the viewer (the viewer is assumed to be at the bottom of that picture.) I am currently researching the ROAM algorithm to learn how to implement CLOD.

Results, Analysis, Conclsuion

In the end I finished some but not all of the modules. However, because of the modular design of the project, everything is still completely functional. The modules that were finished were the XML data loading module, the random terrain generation module, and the terrain rendering module.

The XML data module had a lot that went right and a few things that went wrong. The first flaw was in coding design. I didn't plan it out too well ahead of time. Thus, I had a lot of problems with the recursion. I designed the function so that once it read a tag in, it would then call it self to read more inner tags, but due to a lot of mis-conceptions I had to come close to rewriting a lot of it several times. The end function was a little too hacky, ungaily, and complex, and hard to manage.

I also didn't implement full support for all the XML tags I had designed. I implemented the container, integer, float, and string tags. These were by far the most important ones, but I didn't implement enum, pointer, or extref. I also didn't implement all of the parameters for the tags I did implement. Because of this, currently the XML data file is more restricted in what you can do than I had originally planned. However, with not too much additional time and work, the rest of the tags could be finished and it would be fully functional.

Overall the XML parser is fairly fast and fairly resistant to errors. Thus, if you make an error in your XML data file, it will probably tell you there is an error, instead of crashing or exhibiting unpredictable behavior. It won't do too good a job of helping you find WHERE the error is, but even many professional XML interpreters I have seen don't do a very good job of that either.

For the random terrain generation module, I chose to implement the diamond-square algorithm that I described earlier. I was also planning to implement a second module, the fault-formation method, but for time reasons I focused on only diamond-square. I did a very clean and basic implementation of it. I was going to include more advanced features, like more complex variance, and also possibly multiple types of terrain. Another problem is that I don't have much control over what is generated; sometimes the terrain is generated over the camera and thus the entire screen is black.

The terrain rendering module I didn't get time to do much with. I only implemented a simple, brute-force algorithm. I was planning on experimenting with CLOD, continuous level of detail, which is more complex but would allow for much faster rendering times. I did add some nice detail though; lighting and basic texture mapping. However as there is only one texture it doesn't look too impressive.

References

Books:

Focus on 3D Terrain Programming by Trent Polack

Websites:

http://www.xml.com/pub/a/98/10/guide0.html

http://www.xml.com/pub/a/2000/08/holman/index.html

http://www.gamedev.net/reference/articles/article944.asp

http://www.gamedev.net/reference/articles/article678.asp

http://www.gamedev.net/reference/articles/article971.asp

http://www.llnl.gov/graphics/ROAM/

http://www.gamedev.net/reference/articles/article1436.asp

http://www.gamedev.net/reference/articles/article1842.asp

`http://www.gameprogrammer.com/fractal.html`