Writing your own 2D Map Editor

An introduction to how they work

Written by Rachel J. Morris

July 13th, 2009

http://www.moosader.com

Table of Contents

Introduction

The purpose of this mini-guide is to cover the basic theory on how map editors work. This guide is for 2D map editors in particular, generally being tile-based. You can either make the map editor be strictly on a grid, or have the player place tiles wherever.

Code License

Code used for the map editor is "gift-ware". According to the Allegro library (which I think coined the term):

[This] is given to you freely as a gift. You may use, modify, redistribute, and generally hack it about in any way you like, and you do not have to give us anything in return.

A lot of the code used as example in this guide is from my own map editors. The older versions are named "MusuGo" and the newer ones are just "Moose Tiles". You can get a version of the map editor from my site:

www.moosader.com

What a map file looks like

The first part of writing a map editor is knowing how it's going to be saved, and how it's going to be read by your game.

What will your map files store? One thing you want is what tile. Maybe you'll have the coordinates. Maybe you'll store whether it's solid or not as a boolean, but this isn't necessary. Maybe you'll have different layers, with a couple being under the player, one being above, and maybe a separate layer for the collision detection.

Grid Format

Secondly, what "format" do you want the map to be saved as? My earlier map editors would save a grid of numbers, like this:

```
05 05 05 05 05 05 05
05 01 02 03 05 05 05
05 01 02 03 05 05 05
05 05 05 05 05 05 05
```

What do these numbers symbolize? They are the tile number on the tile-sheet.

The numbers correspond to which tile it is. And, for a tile-sheet that in linear like the one above (there's only one row), the coordinate will be tileNumber * tileWidth.

If you want multiple layers, you do the entire grid for one, then the next, then the next.

Preferably, you would save these in Binary format to save space, but they'll just be saved as .txt (or whatever you want to use as a suffix) so they aren't going to take a ton of space.

Tile-Line Format

In this format, each tile will get it's own line. You can store a lot more information about a tile this way. It will look like this:

```
tile 00   layer 0   loc_x   00   loc_y 00   sheet_x 32   sheet_y 0   collision 1
tile 01   layer 0   loc_x   32   loc_y 00   sheet_x 64   sheet_y 32  collision 0
```

tile 02	layer 0	loc_x 64	loc_y 00	sheet_x 32	sheet_y 0	collision 1

And so on and so forth. This map file will end up being really long, and you can't "glance" and the number grid to figure things out, but you can store more information about each tile.

Also keep in mind these maps can be for side-scrolling platformers, or for top-down RPGs. Anything tile-based!

Saving and Loading the map

Level Structure

The type of map structure I use for my games is like this: I have a 3D array of Tile objects that belongs to the Level class:

```
Tile tile[LAYER_AMT][MAP_W][MAP_H];
```

Tile contains the following:

- x and y coordinates plus width and height, or a "Rectangle" class/struct that handles these.
- Boolean for whether it's "solid" (collidable) or not
- x and y coordinates (or Rectangle) for the tile's coordinates on the tile-sheet

Your map editor will need to save and load maps, while your game will probably just load the map. There are different ways to parse your map files, but here's some basic functions.

Saving (Grid format)

```
ofstream outfile;
outfile.open(filename.c_str());

for (int k=0; k<LAYER_AMT; k++)
{
	for (int j=0; j<MAP_H; j++)
	{
		for (int i=0; i<MAP_W; i++)
		{
			if ( grid[k][i][j].fx/32 < 1000 )
			{
				if ( grid[k][i][j].fx/32 < 10 )
				{
					outfile<<"000"<<grid[k][i][j].fx/32<<" ";
				}
				else if ( grid[k][i][j].fx/32 < 100 )
				{
					outfile<<"00"<<grid[k][i][j].fx/32<<" ";
				}
				else if ( grid[k][i][j].fx/32 < 1000 )
				{
					outfile<<"0"<<grid[k][i][j].fx/32<<" ";
				}
			}
			else
			{
				outfile<<grid[k][i][j].fx/32<<" ";
			}
		}
		outfile<<endl;
	}
	outfile<<endl;
}
cout<<filename<<" written successfully."<<endl;
outfile.close();
}
```

Here, “fx” stands for the X coordinate on the tile-sheet. (Fx stands for “filmstrip-x”)

Loading (Grid format)

```
{
ifstream infile;
infile.open(filename.c_str());

for (int k=0; k<LAYER_AMT; k++)
{
      for (int j=0; j<MAP_H; j++)
      {
            for (int i=0; i<MAP_W; i++)
            {
                  int temp;
                  infile>>temp;
                  grid[k][i][j].fx = temp*32;
            }
      }
}
infile.close();
cout<<"Map loaded successfully";
}
```

Here, the x and y coordinates will be i*32 and j*32.

Saving (Tile-Line format)

```
void Level::SaveMap( string filename, int MAP_W, int MAP_H, int LAYER_AMT )
{
    int total = MAP_W*MAP_H*LAYER_AMT;
    int count = 0;

    cout<<"Saving to "<<filename<<"..."<<endl;
    ofstream outfile;
    outfile.open( filename.c_str() );
    outfile<<"W: "<<MAP_W<<"\tH: "<<MAP_H<<"\tv 2\t\tTileAmt: "<<total;
    outfile<<endl<<endl;

    for ( int k=0; k<LAYER_AMT; k++ )               //layer
    {
        for ( int j=0; j<MAP_H; j++ )       //y
        {
            for ( int i=0; i<MAP_W; i++ )   //x
            {
                outfile<<"tile "<<count<<" layer "<<k<<" x "<<tile[k][i][j].X();
                outfile<<" y "<<tile[k][i][j].Y()<<" image "<<tile[k][i][j].Fx();
                outfile<<" solid "<<tile[k][i][j].Solid()<<" end"<<endl;
                count++;
            }
            outfile<<endl;
        }
        outfile<<endl<<endl;
    }
    outfile.close();
    cout<<"Save complete"<<endl;
}
```

Since my tilesheets tend to be in a straight line, with only one row, I only save what the x-coordinate on the tilesheet is. I have this stored as "Fx()" (filmstrip-x).

The output for this map is:

```
W: 80   H: 60          v 2     TileAmt: 19200
tile 0  layer 0        x 0     y 0     image 0       solid 0       end
tile 1  layer 0        x 32    y 0     image 0       solid 0        end
tile 2  layer 0        x 64    y 0     image 0       solid 0       end
tile 3  layer 0        x 96    y 0     image 0       solid 0       end
tile 4  layer 0        x 128   y 0     image 0       solid 0       end
tile 5  layer 0        x 160   y 0     image 0       solid 0       end
```

I added "tile" and "end" as a beginning and ending, to help the parser.

Loading (Tile-Line format)

```
void Level::LoadMap( string filename, int MAP_W, int MAP_H )
{
    cout<<"Loading from "<<filename<<"..."<<endl;
    ifstream infile;
    infile.open( filename.c_str() );
    string temp;

    // If you want, you can load in the MAP_W, and MAP_H from the
    // map's header, but if all maps are the same size you don't need to.
    // More on dynamic arrays later.

    //temporary
    int x, y, layer, fx;
    bool solid;

    while ( infile>>temp )
    {
        // This will skip over the header info automatically.
        // It also ignores the "tile #", as that's just a counter for reference.

        if ( temp == "layer" )
        {
            infile>>layer;
        }
        else if ( temp == "x" )
        {
            infile>>x;
        }
        else if ( temp == "y" )
        {
            infile>>y;
        }
        else if ( temp == "solid" )
        {
            infile>>solid;
        }
        else if ( temp == "image")
        {
            infile>>fx;
        }
        else if ( temp == "end" )
        {
            //Store tile info once we hit "end"
            tile[layer][x/32][y/32].X( x );
            tile[layer][x/32][y/32].Y( y );
            tile[layer][x/32][y/32].Solid( solid );
            tile[layer][x/32][y/32].FX( fx );
        }
    }

    infile.close();
    cout<<"Load complete"<<endl;
}
```

Creating a 3-dimensional dynamic array

Here's some code on creating a 3D dynamic array for your map editor. If you don't want to do this, you could have the user input the MAP_W and MAP_H in the command prompt before everything's initialized, and create the array this way:

Tile tile[LAYER_AMT][MAP_W][MAP_H];

But some compilers might not like this.

```
void Level::AllocateTiles( int MAP_W, int MAP_H, int LAYER_AMT )
{
    tile = new Tile**[LAYER_AMT];
    for ( int i=0; i<LAYER_AMT; i++ )
    {
        tile[i]= new Tile*[MAX_X];
        for ( int j=0; j<MAX_X; j++ )
        {
            tile[i][j] = new Tile[MAX_Y];
        }
    }
}
```

The Map Editor

Now you actually have to design your interface to create the maps! You will have to write a basic GUI (graphical user interface). This can be as simple or complex as you want, mainly depending on if you're the only person using it or not. Of course, it's much easier to use keyboard commands to enable/disable things, or choose which layer to draw on, but if other people will be using it, this will be confusing for them.

I do not know a "good way" to write a GUI. You might use wxWidgets or something else if you're going to be really serious about it, but it's been a while since I've worked on one.

Elements

Here are some classes you'll want:

- Brush – stores what layer the user is drawing on, the size (1x1 tile area? 3x3 tile area?), a boolean for if the mouse button is being held down, and what the current tile is (maybe the tile-sheet x and y coordinates).

- Level class – This is going to be about the same as your in-game Level class, except having a "save" function. Includes a Load and Draw function, namely.

- Tile class – the Level class will have an array of Tiles. These will hold X and Y coordinates, Tile-sheet coordinates, a boolean for whether it's solid or not.

- GUI elements
  - Window
  - Button/Radio Button/Check Box

A graphical look at an editor...

A – The tileset is displayed across the top. When the user clicks one of the tiles, it is stored in the "Brush" class, as well as displayed at **H**.

B – Tileset scroller. I also have keyboard shortcuts to scroll through the tileset.

C – The grid numbers. Not necessary, but helps get an idea of the dimensions of the map, and where you're currently drawing.

D – The map. It can be scrolled with the arrow keys. It displays all layers at once

E – Main options, such as the Save, Load, Exit, and Toggle Fullscreen buttons

F – The layer options, to enable drawing on layer 1, 2, or 3. I also have a Collision layer, but was lazy and didn't add a button. With my editor, you enable collision layer with the "C" key.

G – The brush options. You can choose a 1x1 tile brush, 3x3, fill, or erase entire layer.

H – A display of the current tiel

I – A mini-map of the map drawn.

There are a few un-used buttons, such as the arrow for "event", and the two blank ones on the right.