I'm currently writing a game that uses Isometric tiles to display a landscape on the screen. The landscape having hills and valleys is part of the game's design.  After a search on the Internet for help in determining which tile I was on I was presented with the following options:

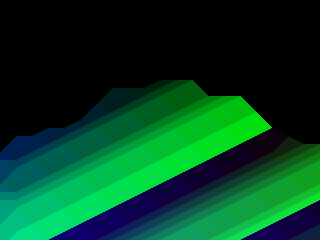
1. Use mouse maps as described in Ernest 'TANSTAAFL' Pazera's book: [Isometric Game Programming with DirectX 7](http://www.gamedev.net/columns/books/bookdetails.asp?productid=110).
2. Create a second screen and blit each tile on to this second screen in a sort of tile map (see the explanation & screen grab below)
3. Forget 2D/DirectX 7 and go 3D/DirectX 8 as everyone seems to be going this way

The first option is great if you have a flat landscape, but I'm not basing my game in Holland or any other area of flat land.  The second option, which I've tried and it does work, has drawbacks in that I've got to blit the map twice onto two different back buffers, and on one of the back buffers I've got to blit in one colour per tile - very bad for performance.  The final option at first seems good, but requires learning Direct 3D/DirectX Graphics.

I'll explain the theory behind my experiments and then I'll provide some sample code afterwards.

**Experiment 1: One colour per tile**

I first started with the second option and after a couple of days playing around had a simple program working that highlighted the correct tile.  The following screen grab is the tile map that I used to see where on the map I was:



This map was generated initially and then updated every time I moved the screen around the map.  This caused the system to lose a few frames/second as it had to draw the same map twice.  I also had problems finding where the mouse was as this involved a lock, read and finally an unlock which took time. Since this operation was being performed every time the mouse/screen moved then it started to add up to being a very slow process.

**Experiment 2: Adapting the mouse maps to be tile independent**

I started to think about the possibilities of using Mouse maps after trying to get better performance out of the second option.  Instead of using the normal isometric tile map:

http://web2.cyol.com/expo/xml/star/1000/56/img/wave/2010/04/img_1270168778_1.gif

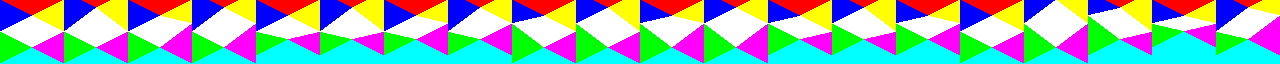
I extended the shape to be a square and produced the following new standard iso mouse map for flat terrain

.

After a little bit of a play with the new mouse map system, I discovered that the original mouse map had to be used to find the tile, but the new tile would allow me to use all of my tiles as a base for their own mouse map. Thus the following tile set:



became the following mouse map templates:

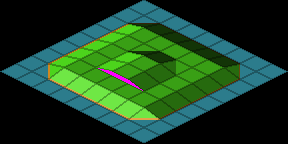


At first glance this seems strange, especially tiles 4-7, 12-14 and the last 3 where the cyan colour is not a normal isometric flat tile, but this is okay and it works perfectly with the algorithms I'm using.

By using the first 4 steps in Ernest's book to get to the rough, yet flat tile position, I then allowed my system to enter a while loop to get to the final tile.  The full steps in Ernest's book are:

1. Screen -> world coordinates
2. Subtract the world coordinates of map position (0,0)
3. Determine mouse map coordinates
4. Perform a course tile walk
5. Use mouse map lookup table

It was this final step that I changed into a while loop, and using the following test map I tried a little test:



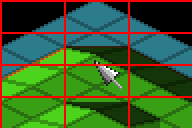
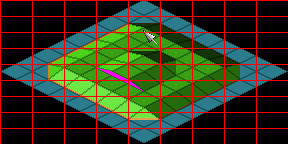
As can be seen in the simple map I was using, the game I'm working on can have cliffs. The purple area shows where a cliff would go. My mouse mapping algorithm was entering an infinite loop moving north east and south west whenever my mouse entered this purple area.  This was not what you would want to happen in a real game. So I've got this problem to sort out, and at the moment I think the easiest way would be to limit the number of moves the system can make in opposite directions.

**Explanation of how to use my method**

The first 4 steps of Ernest's Mouse -> Tile conversion are unchanged, except I'm adding a quarter tile height to the fine Y position so that the fine co-ordinates are sitting inside of the normal mouse map inside of my extended mouse map.  The final step (using the mouse map) has to be changed slightly.

In a normal mouse map lookup function you get the direction to travel and then exit after performing the movement. With my method you keep performing the lookup / movement loop until the lookup returns no movement, i.e. the colour is white.

Using the following image (this is my test map from before, but this time the red lines shows the coarse division of the map):



If you look at the mouse pointer (the enlarged view on the right) then you will notice that using a normal iso mouse map the tile returned would be 1,1, whereas the actual tile is 2,2.  After getting the initial tile number (1,1) we now switch over to the extended mouse maps (the following images have been enlarged to aid in the explinations):

The tile at map position 1,1 is:



and the mouse map for this is :



If you overlay the mouse pointer onto the mouse map you get :



and as you can see the mouse is in the cyan area, and this means to move south by one tile, so we do this and get the following Tile 2,2 image :



and mouse map :

.

We also need to update the fine coordinates by a full tile's height : tile dimensions are 32 x 16 (64 x 32 for larger tile set) so fy has 16 (32) subtracted from it, and the new position of the mouse on the new mouse map is :



just inside of the white area of the tile. Thus we have found the tile we wanted, but not quite yet.  Although this is the correct tile, the position of the mouse pointer is wrong, if you compare the two mouse maps you will notice that the mouse pointer on the first mouse map was about a quarter of a tile height down (this is the offset I use for the tile's height from ground level), so adding this onto the value of fy gives the following final position of the mouse map:



Here is a table of the changes to the fine X and Y co-ordinates, assuming that the tile width is 32, and thus the mouse maps are 32 x 32:

|  |  |  |  |
| --- | --- | --- | --- |
| Colour | Direction | Fine X change | Fine Y change |
| White | None | 0 | 0 |
| Blue | North West | +16 | +8 |
| Red | North | 0 | +16 |
| Yellow | North East | -16 | +8 |
| Green | South West | +16 | -8 |
| Cyan | South | 0 | -16 |
| Purple | South East | -16 | -8 |

**The code for the mouse maps**

In my game you can zoom in and out of the map. Actually there are two zoom levels, large and small, so in the following code segments m\_Size is the current tile size (0 = Large, 1 = Small).

First define the global mouse map array:

char g\_MouseMap[2][20][64][32];

[2] is for the two mouse maps  
[20] is for the 20 tiles that I use  
[64] is for the height of a large tile (64 x 64)  
[32] is for the width of a large tile (64 x 64). I'm using 7 colours so I'm compressing the data down, and saving approx 80K of memory.

Next the mouse map lookup constants:

#define MM\_NONE 0

#define MM\_NORTHWEST 1

#define MM\_NORTH 2

#define MM\_NORTHEAST 3

#define MM\_SOUTHWEST 4

#define MM\_SOUTH 5

#define MM\_SOUTHEAST 6

These are the same as the ones Ernest uses, but I've added North and South movement.

A helper function to load the mouse maps in to the array from 256 colour RAW files (produced from PSP without a header):

bool LoadMouseMaps()

{

FILE \*fin;

int x, y, o;

char c;

fin = fopen( "smtiles.raw", "rb" );

if( fin == NULL )

{

Log( \_\_FILE\_\_, \_\_LINE\_\_, "Unable to open smtiles.raw" );

return false;

}

else

{

for( y = 0; y < 32; y++ )

{

for( o = 0; o < 20; o ++ )

{

for( x = 0; x < 16; x ++ )

{

c = ((fgetc(fin) & 0x0F) << 4) + fgetc(fin) & 0x0F;

g\_MouseMap[1][o][y][x] = c;

}

}

}

fclose( fin );

fin = fopen( "lgtiles.raw", "rb" );

if( fin == NULL )

{

Log( \_\_FILE\_\_, \_\_LINE\_\_, "Unable to open lgtiles.raw" );

return false;

}

else

{

for( y = 0; y < 64; y++ )

{

for( o = 0; o < 20; o ++ )

{

for( x = 0; x < 32; x ++ )

{

c = ((fgetc(fin) & 0x0F) << 4) + fgetc(fin) & 0x0F;

g\_MouseMap[0][o][y][x] = c;

}

}

}

fclose( fin );

}

}

return true;

}

The Log( \_\_FILE\_\_, \_\_LINE\_\_, "\*\*\*" ) function is my error logging code, but the rest of the code is standard C / C++.  Please note that when exporting a raw file from PSP, reduce the colours down to 16 and then export, taking note of how the system has re-arranged your colours in the palette as this caused a problem in my code at first.

The following is a little helper to get the mouse map movement number:

int MM\_LookUp( int x, int y, int t )

{

int v;

x %= ( m\_Size == 1 ? 32 : 64 );

y %= ( m\_Size == 1 ? 32 : 64 );

v = g\_MouseMap[m\_Size][t][y][x >> 1];

if( ( x & 1 ) == 0 )

v = v & 0x0F;

else

v = ( v >> 4 ) & 0x0F;

return v;

}

And finally the code for the main mouse map function, please note this code still has the infinite loop in it:

void MouseMap( int xp, int yp, int &mx, int &my )

{

int m;

int cx, cy;

int fx, fy;

int md, mc;

// Adjustment for lower left corner offset.

yp += ( m\_Size == 1 ? 32 : 64 );

// Step 1 : Screen -> World Co-Ords

xp += m\_OffX;

yp += m\_OffY;

// Step 2 : Offset from world co-ords of tile (0,0)

xp -= ( ( 0 - 0 ) << ( 5 - m\_Size ) );

yp -= ( ( 0 + 0 ) << ( 4 - m\_Size ) );

// Step 3 : Determine MouseMap Co-ords

// Course co-ords

cx = xp / ( m\_Size == 1 ? 32 : 64 );

cy = yp / ( m\_Size == 1 ? 16 : 32 );

// Fine co-ords

fx = xp % ( m\_Size == 1 ? 32 : 64 );

fy = yp % ( m\_Size == 1 ? 16 : 32 );

// Adjust for my Unusual mouse maps...

fy += ( m\_Size == 1 ? 8 : 16 );

// Adjust for negative co-ords

if( fx < 0 )

{

fx += ( m\_Size == 1 ) ? 32 : 64;

cx --;

}

if( fy < 0 )

{

fy += ( m\_Size == 1 ) ? 32 : 64;

cy --;

}

// Step 4 : Perform Corse Tile walk

mx = 0;

my = 0;

while( cy < 0 )

{

TileWalk( mx, my, ISO\_NORTH );

cy ++;

}

while( cy > 0 )

{

TileWalk( mx, my, ISO\_SOUTH );

cy --;

}

while( cx < 0 )

{

TileWalk( mx, my, ISO\_WEST );

cx ++;

}

while( cx > 0 )

{

TileWalk( mx, my, ISO\_EAST );

cx --;

}

// Step 5 : Use the lookup table for final tile walk...

m = -1;

mc = 4;

while( ( m != MM\_NONE ) && ( mc > 0 ) )

{

m = MM\_LookUp( fx,

fy + (Map[( my >= 0) ? my : 0][(mx >= 0) ? mx : 0].Height << (3 - m\_Size)),

Map[(my >= 0) ? my : 0][(mx >= 0) ? mx : 0].Floor);

switch( m )

{

case MM\_NORTHWEST:

{

if( md == MM\_SOUTHEAST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_NORTHWEST );

fx += ( m\_Size == 1 ) ? 16 : 32; // Half x

fy += ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

case MM\_NORTH:

{

if( md == MM\_SOUTH )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_NORTH );

fy += ( m\_Size == 1 ) ? 16 : 32; // Half y

break;

}

case MM\_NORTHEAST:

{

if( md == MM\_SOUTHWEST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_NORTHEAST );

fx -= ( m\_Size == 1 ) ? 16 : 32; // Half x

fy += ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

case MM\_SOUTHWEST:

{

if( md == MM\_NORTHEAST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_SOUTHWEST );

fx += ( m\_Size == 1 ) ? 16 : 32; // Half x

fy -= ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

case MM\_SOUTH:

{

if( md == MM\_NORTH )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_SOUTH );

fy -= ( m\_Size == 1 ) ? 16 : 32; // Half y

break;

}

case MM\_SOUTHEAST:

{

if( md == MM\_NORTHWEST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_SOUTHEAST );

fx -= ( m\_Size == 1 ) ? 16 : 32; // Half x

fy -= ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

}

md = m;

}

}

**A slight problem with the above techniques**

Ok so the above would work, as long as you do not go above a height of 2.  To go above a height of 2, a small change to step 5 of the mouse map function is required. Add a new variable declaration (int tfy) and you're back in business:

// Step 5 : Use the lookup table for final tile walk...

m = -1;

mc = 4;

while( ( m != MM\_NONE ) && ( mc > 0 ) )

{

m = MM\_LookUp( fx,

fy + (Map[(my >= 0) ? my : 0][(mx >= 0) ? mx : 0].Height << (3 - m\_Size)),

Map[(my >= 0) ? my : 0][(mx >= 0) ? mx : 0].Floor);

tfy = (fy +

(Map[(my >= 0) ? my : 0][(mx >= 0) ? mx : 0].Height << (3 - m\_Size)));

if( ( fx >= 0 ) && ( tfy >= 0 ) && ( fx < ( m\_Size == 1 ? 32 : 64 ) ) &&

( tfy < ( m\_Size == 1 ? 32 : 64 ) ) )

{

switch( m )

{

case MM\_NORTHWEST:

{

if( md == MM\_SOUTHEAST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_NORTHWEST );

fx += ( m\_Size == 1 ) ? 16 : 32; // Half x

fy += ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

case MM\_NORTH:

{

if( md == MM\_SOUTH )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_NORTH );

fy += ( m\_Size == 1 ) ? 16 : 32; // Half y

break;

}

case MM\_NORTHEAST:

{

if( md == MM\_SOUTHWEST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_NORTHEAST );

fx -= ( m\_Size == 1 ) ? 16 : 32; // Half x

fy += ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

case MM\_SOUTHWEST:

{

if( md == MM\_NORTHEAST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_SOUTHWEST );

fx += ( m\_Size == 1 ) ? 16 : 32; // Half x

fy -= ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

case MM\_SOUTH:

{

if( md == MM\_NORTH )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_SOUTH );

fy -= ( m\_Size == 1 ) ? 16 : 32; // Half y

break;

}

case MM\_SOUTHEAST:

{

if( md == MM\_NORTHWEST )

mc --;

else

mc = 4;

TileWalk( mx, my, ISO\_SOUTHEAST );

fx -= ( m\_Size == 1 ) ? 16 : 32; // Half x

fy -= ( m\_Size == 1 ) ? 8 : 16; // Quarter y

break;

}

}

}

else

{

if( tfy >= ( m\_Size == 1 ? 32 : 64 ) )

{

m = MM\_SOUTH;

TileWalk( mx, my, ISO\_SOUTH );

fy -= ( m\_Size == 1 ) ? 16 : 32;

}

}

md = m;

}

I've [included a copy](http://downloads.gamedev.net/features/programming/mm4ihm/IsoTerrain.zip) of my test application that uses the above techniques.