Creating a Role Playing Game with XNA Game Studio 3.0
Part 35
Tile Map Editor - Part 7
Minor Fix to Tile Engine

To follow along with this tutorial you will have to have read the previous tutorials to understand much of what it going on. You can find a list of tutorials here: XNA 3.0 Role Playing Game Tutorials You will also find the latest version of the project on the web site on that page. If you want to follow along and type in the code from this PDF as you go, you can find the previous project at this link: http://www.jtmbooks.com/rpgtutorials/New2DRPG34.zip You can download the graphics from this link: Graphics.zip

I will be working on the editor again in this tutorial. I also said on my blog that I will try and fix an issue with the tile engine displaying lines around the tiles. The reason you are getting lines is an issue with the tiles that I am using. If you use better tiles then there are no lines when the tile engine displays the tiles. There is a way to minimize this. What you can do is click the **tilesetTexture1.png** file in the **TileSets** folder in the **Content** folder. In the **Properties** window there is an entry **Content Processor** with a + beside it. Click the + to bring up other options. Set the **Color Key Enabled** to **False** and **Generate Mipmaps** to **True**. There is just one more thing to do. In the **Draw** method of the **TileMapLayer** class what you want to do is change the call to **sBatch.Begin**. You want to change the second parameter to **SpriteSortMode.BackToFront**. This will help the tiles being drawn on the screen. This is the complete code for the **Draw** method.

```
public void Draw(SpriteBatch sBatch, Texture2D texture, Tileset tileset)
{
    Point cameraPoint = VectorToCell(Game1.Camera.Position);
    Point viewPoint = VectorToCell(Game1.Camera.Position +
                        TileEngine.ViewPortVector);
    Point min = new Point();
    Point max = new Point();
    min.X = cameraPoint.X;
    min.Y = cameraPoint.Y;
    max.X = (int)Math.Min(viewPoint.X, map.GetLength(1));
    max.Y = (int)Math.Min(viewPoint.Y, map.GetLength(0));

    Rectangle tileRectangle = new Rectangle(
            0,
            0,
            TileEngine.TileWidth,
            TileEngine.TileHeight);

    sBatch.Begin(SpriteBlendMode.AlphaBlend,
        SpriteSortMode.BackToFront,
        SaveStateMode.None,
        Game1.Camera.TransformMatrix);
    for (int x = min.X; x < max.X; x++)
    {
        for (int y = min.Y; y < max.Y; y++)
        {
            if (map[y, x] != -1)
            {
```

```
                tileRectangle.X = x * TileEngine.TileWidth;
                tileRectangle.Y = y * TileEngine.TileHeight;

                sBatch.Draw(texture,
                    tileRectangle,
                    tileset.Tiles[map[y, x]],
                    Color.White);
            }
        }
    }
    sBatch.End();
}
```

For the rest of this tutorial I will be working on the editor so right click the **TileMapEditor** project and select **Set As Start Up Project**. What I will be doing in this tutorial is switching to using just the mouse to draw the tiles. The way that it is going to work is if you move the mouse to the edges of the map display the map will scroll. If you press and hold down and move the mouse it will draw the tile in the location you move the mouse into. The tracking isn't 100% perfect. You may have to at some points go back and fill in a few tiles. I needed a few new fields to do this. You can also get rid of a few. You can delete both of the **KeyboardState** fields as the will not be used any more. Add these fields around the **currentLayer** field.

```
int lastTick = Environment.TickCount;
int mouseX;
int mouseY;
bool isMouseDown = false;
bool trackMouse = false;
```

The first field is an int called **lastTick**. This is set to the value of **Environment.TickCount**. What this does is get the number of milliseconds, approximately, since the computer started. I will use this to slow the scrolling of the map when the mouse cursor is at the edges of the map. The next two **mouseX** and **mouseY** will hold the **X** and **Y** coordinates of the mouse on the display. The next one, **isMouseDown**, is use to tell if a mouse button is being pressed inside the display. The last one **trackMouse** will be used in the **Logic** method to determine if we need to track the mouse's movements.

Instead of using XNA to track the mouse I will be using the methods in C# to do this. It just simplifies things in my mind. We are going to need a few event handlers to do this. Controls in C# have several event handlers that we can pay attention to. The ones I'm interested in is when the mouse is over the control and it moves, if the left mouse is up or down and when the mouse enters or leaves the control. You will add the event handlers in the constructor for the form. Remember when you type the += you can press tab twice to generate the method stubs. The code for the event handlers is all pretty straight forward. This is the code for the constructor and the event handlers. I will explain the event handlers after you have seen the code.

```
public Form1()
{
    InitializeComponent();
    layerToolStripMenuItem.Enabled = false;

    tileDisplay1.OnInitialize +=
        new EventHandler(tileDisplay1_OnInitialize);

    tileDisplay1.OnDraw +=
        new EventHandler(tileDisplay1_OnDraw);
```

```
    newMapToolStripMenuItem.Click +=
        new EventHandler(newMapToolStripMenuItem_Click);

    saveMapToolStripMenuItem.Enabled = false;
    saveMapToolStripMenuItem.Click +=
        new EventHandler(saveMapToolStripMenuItem_Click);

    openMapToolStripMenuItem.Click +=
        new EventHandler(openMapToolStripMenuItem_Click);

    exitToolStripMenuItem.Click +=
        new EventHandler(exitToolStripMenuItem_Click);

    newLayerToolStripMenuItem.Click +=
        new EventHandler(newLayerToolStripMenuItem_Click);

    clbLayers.SelectedIndexChanged +=
        new EventHandler(clbLayers_SelectedIndexChanged);

    nudCurrentTile.ValueChanged +=
        new EventHandler(nudCurrentTile_ValueChanged);
    nudCurrentTile.InterceptArrowKeys = false;

    Application.Idle += new EventHandler(Application_Idle);

    this.FormClosing +=
        new FormClosingEventHandler(Form1_FormClosing);

    tileDisplay1.MouseMove +=
        new MouseEventHandler(tileDisplay1_MouseMove);

    tileDisplay1.MouseDown +=
        new MouseEventHandler(tileDisplay1_MouseDown);

    tileDisplay1.MouseUp +=
        new MouseEventHandler(tileDisplay1_MouseUp);

    tileDisplay1.MouseEnter +=
        new EventHandler(tileDisplay1_MouseEnter);

    tileDisplay1.MouseLeave +=
        new EventHandler(tileDisplay1_MouseLeave);
}
void tileDisplay1_MouseMove(object sender, MouseEventArgs e)
{
    mouseX = e.X;
    mouseY = e.Y;
}

void tileDisplay1_MouseDown(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left)
        isMouseDown = true;
}

void tileDisplay1_MouseUp(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left)
```

```
        isMouseDown = false;
}

void tileDisplay1_MouseEnter(object sender, EventArgs e)
{
    trackMouse = true;
}

void tileDisplay1_MouseLeave(object sender, EventArgs e)
{
    trackMouse = false;
}
```

The first event handler **tileDisplay1_MouseMove** is fired when the mouse moves inside of **tileDisplay1**. All event handlers have two parameters. The first parameter is an **object** and is called **sender**. This parameter tells what had triggered the event. The second parameter is always **e** but can have many different types. In the **MouseMove** and **MouseDown** handlers it is of type **MouseEventArgs** which holds information about the mouse. **MouseEventArgs** has two properties **X** and **Y** that hold the **X** and **Y** location of where the mouse was in the control. I capture these value into the **mouseX** and **mouseY** fields.

The **tileDisplay1_MouseDown** and **tileDisplay1_MouseUp** method are very similar. There is also a property of **MouseEventArgs** called **Button** and it is an enum that holds the different mouse buttons. I check to see if the left button is being pressed. In the **MouseDown** handler if it is the left button I set **isMouseDown** to true. In the **MouseUp** handler if the left mouse button is up I set **isMouseDown** to false.

The code for the last two event handlers is also very similar. In the **MouseEnter** event the mouse is over the control so we are interested in tracking it so I set **trackMouse** to true. In the **MouseLeave** event we are not interested in the mouse anymore so I set **trackMouse** to false.

I made a really big error in the **Form1_FormClosing** event handler. If you load the program and try to exit with out doing anything the program will not exit. There was an easy fix to this problem. All I did was set **e.Cancel** to be false at the start of the method. That way the rest of the code will only be triggered it there has been a change to the map. This is the new method.

```
void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = false;
    if (isChanged)
    {
        DialogResult result = MessageBox.Show(
            "The map has changed. Are you sure you want to close?",
            "Save map?",
            MessageBoxButtons.YesNoCancel,
            MessageBoxIcon.Question);

        if (result == DialogResult.Yes)
            e.Cancel = false;
        else
            e.Cancel = true;
    }
}
```

I added in the timing of scrolling the map to the **tileDisplay1_OnDraw** method. What this

method does is take the value of **lastTick** and subtract the current tick count. If that value is less than 50, which is approximately 50 milliseconds, it exits the method and not calling the **Logic** or **Render** methods. If the value was greater than 50 **lastTick** is set to the current tick and the **Logic** and **Render** methods are called. If this runs too fast on your computer just increase 50 to a larger number. If it runs too slowly you can try decreasing 50. This is the code for the **tileDisplay1_OnDraw** method.

```
void tileDisplay1_OnDraw(object sender, EventArgs e)
{
    if (Environment.TickCount - lastTick < 50)
        return;
    lastTick = Environment.TickCount;
    Logic();
    Render();
}
```

If you have read my blog there was a comment that requested that the current selected tile be drawn on the map be displayed in the cursor. This was not really a hard thing to do so I added it in. This will be taken care of in the **DrawDisplay** method. I don't want the tile to appear when I'm erasing a tile so I check to make sure that **rbErase** is not checked. To actually draw the tile I just cast the value of **nudCurrentTile** to an integer and draw the tile using the **tiles** field of the **Tileset** class. After drawing the tile I draw the cursor. This is the updated **DrawDisplay** method.

```
private void DrawDisplay()
{
    spriteBatch.Begin();
    for (int x = 0; x < tileDisplay1.Width / Engine.TileWidth; x++)
        for (int y = 0; y < tileDisplay1.Height / Engine.TileHeight; y++)
            spriteBatch.Draw(cursor,
                new Rectangle(x * Engine.TileWidth,
                y * Engine.TileHeight,
                Engine.TileWidth,
                Engine.TileHeight),
                Color.White);
    spriteBatch.End();

    spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
    if (!rbErase.Checked)
    {
        spriteBatch.Draw(texture,
            new Rectangle(
                (int)(position.X / Engine.TileWidth) * Engine.TileWidth
                - (int)camera.Position.X,
                (int)(position.Y / Engine.TileHeight) * Engine.TileHeight
                - (int)camera.Position.Y,
                Engine.TileWidth,
                Engine.TileHeight),
                tileset.tiles[(int)nudCurrentTile.Value],
                Color.White);
    }
    spriteBatch.Draw(cursor,
        new Rectangle(
            (int)(position.X / Engine.TileWidth) * Engine.TileWidth
             - (int)camera.Position.X,
            (int)(position.Y / Engine.TileHeight) * Engine.TileHeight
             - (int)camera.Position.Y,
            Engine.TileWidth,
            Engine.TileHeight),
```

```
            Color.Red);
    spriteBatch.End();
}
```

I also made some changes to the **Logic** method. They are a little extensive so I will show you the code and then go over it. This is the updated **Logic** method.

```
private void Logic()
{
    if (tileMap != null)
    {
        if (trackMouse)
        {
            if (mouseX <= Engine.TileWidth)
                camera.Position.X -= Engine.TileWidth;
            if (mouseX >= tileDisplay1.Width - Engine.TileWidth)
                camera.Position.X += Engine.TileWidth;
            if (mouseY <= Engine.TileHeight)
                camera.Position.Y -= Engine.TileHeight;
            if (mouseY >= tileDisplay1.Height - Engine.TileHeight)
                camera.Position.Y += Engine.TileHeight;
            camera.LockCamera();

            position.X = mouseX + camera.Position.X;
            position.Y = mouseY + camera.Position.Y;
        }
        int tileX = (int)position.X / Engine.TileWidth;
        int tileY = (int)position.Y / Engine.TileHeight;

        if (isMouseDown)
        {
            isChanged = true;
            if (rbDraw.Checked)
            {
                currentLayer.SetTile(
                    tileX,
                    tileY,
                    (int)nudCurrentTile.Value);
            }
            if (rbErase.Checked)
            {
                currentLayer.SetTile(
                    tileX,
                    tileY,
                    -1);
            }
        }
        tbCursorPosition.Text = "( " + tileX.ToString() + ", ";
        tbCursorPosition.Text += tileY.ToString() + " )";
        tbCursorPosition.Invalidate();
    }
}
```

I have removed all support for the keyboard. What I do now is if there is a map to work on I check to see if the mouse is in the display by checking the **trackMouse** field. If it is I then check to see if the mouse is in any of the tiles on the edges of the display. To find out if the tile is in the first column I check to see if **mouseX** is less than or equal to the width of the tiles on the screen. If it is I then subtract the width of a tile from the **X** value of the camera's position. The next test is to see if the

mouse is in the last row of tiles. I test that by checking if **mouseX** is greater than or equal to the width of the display minus the width of a tile. If it is I add the width of a tile to **X** value of the camera's position. I do something similar for scrolling the map up or down. To see if the mouse is in the top row of tiles I compare **mouseY** to the height of a tile on the display. If it is less than or equal that I subtract the height of a tile from the **Y** value of the camera's position. To check if the mouse is in the bottom row I compare **mouseY** to the height of the display minus the height of a tile on the screen. If it is I add the height of a tile to the **Y** value of the camera's position. I then call the **LockCamera** method of the camera class that will lock the camera so the cursor will not scroll off the map.

I set **position.X**, which is the position of the cursor, to **mouseX** plus the **X** value of the camera's position. I do something similar for **position.Y**. I take **mouseY** and add **camera.Position.Y**. This gets the position of the cursor, in pixels, on the map and not just the screen. I then get which tile the cursor. To find the **X** coordinate you divide **position.X** by the width of the tiles on the screen. To find the **Y** coordinate you divide **position.Y** by the height of the tiles on the screen. Instead of checking to see if the space bar has been pressed and released I check to see if the mouse button is down using **isMouseDown**. If the mouse is down the code is as before. I set **isChanged** to true and if **rbDraw** is checked I draw the tile and if **rbErase** is checked I erase the tile. I finally set the text in **tbCursorPosition** to be the tile the cursor is in.

There is one more thing that you can do. You no longer need the **LockCursor** method. All of that is now done in the **Logic** method. You can also remove these using statements because I am no longer using the XNA Input classes.

```
using Microsoft.Xna.Framework.Input;
using XKeys = Microsoft.Xna.Framework.Input.Keys;
```

That is it for this tutorial. In the next tutorial I will be working on the game again and I have already started coding the next part of Eyes of the Dragon. I encourage you to keep either visiting my site http://xna.jtmbooks.com or my blog, http://xna-rpg.blogspot.com for the latest news on my tutorials.