Creating a Role Playing Game with XNA Game Studio 3.0
Part 28
Tile Map Editor – Part 2

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/New2DRPG27.zip You can download the graphics from this link: Graphics.zip

The **Tile Map Editor** is all set up but it doesn't do anything at the moment. Now it is time to add functionality to the editor. The first thing that I will work on is getting a tile set loaded so it can be used to render the map.

I am going to use a class to hold the tile set information, similar to the class I made for the **Content Pipeline** project for importing, processing and writing a **Tileset** object. Right click the **TileMapEditor** project and add a new class to the project called **Tileset**. This is the code for the **Tileset** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;

namespace TileMapEditor
{
    class Tileset
    {
        public string textureName;
        public int tileWidth, tileHeight;
        public List<Rectangle> tiles;

        public Tileset()
        {
            textureName = "";
            tileWidth = 0;
            tileHeight = 0;
            tiles = new List<Rectangle>();
        }

    }
}
```

There is a using statement for the XNA framework in this class because I need the **Rectangle** class for the **List<Rectangle>** that will hold the rectangles for the tiles in the tile set. I usually don't like making fields in classes public but in this case I decided to do that to allow easy access to the fields of the class. Another reason I did is because this class isn't meant to be used outside of the editor. There is a string field, **textureName**, that holds the name of the texture. There are two integers, **tileWidth** and **tileHeight** that hold the width and height of the tiles in the tileset image. Finally there is a

List<Rectangle> called **tiles** that will hold the rectangles for the tiles in the tile set. There is a constructor for the class that just sets **textureName** to the empty string, **tileWidth** and **tileHeight** to 0 and creates a new **List<Rectangle>** for the **tiles** field. This is just a basic class to separate the tile set from the rest of the project.

Before I get much further there are a few using statements that I need to add to the code for **Form1**. I will be doing file I/O so I will need the **System.IO** namespace. I will be working with XML documents so I will need the **System.Xml** namespace. I will be using a few qualfied using statements for classes **System.Drawing** namespace. Instead of having using statement for a namespace that is referenced in yourr project you can use the full name of the class. For example if you wanted to use the **Rectangle** class in the **System.Drawing** namespace you can just use **System.Drawing.Rectangle** for the class instead of just **Rectangle**. You can use qualified name spaces to shorten that. Say you wanted to use **Rect** instead of **System.Drawing.Rectangle**. You can just add the following using statement to your class **using Rect = System.Drawing.Rectangle;** so now you can use **Rect** instead of the full part **System.Drawing.Rectangle**. Add these using statements to the code for **Form1**.

```
using System.IO;
using System.Xml;

using Image = System.Drawing.Image;
using Bitmap = System.Drawing.Bitmap;
using Graphics = System.Drawing.Graphics;
using Rect = System.Drawing.Rectangle;
using GraphicsUnit = System.Drawing.GraphicsUnit;
```

The first two are of course for the **System.IO** and **System.Xml** namespaces. The other ones are probably not as obviuos. They will be used for taking a part of the whole tile set image and just placing the tile we are using in the **PictureBox** control. You will also need to add a few fields to the **Form1** class.

```
XmlDocument input;
Tileset tileset = null;
Texture2D texture;
Image tilesetBitmap;
```

The first one is an **XmlDocument** object **input**. It will be used to load the tileset. I called it input because I will just copy the codeThe next one **tileset** will hold the an object of the above **Tileset** class. The third will hold the **Texture2D** of the tile set. The last one **tilesetBitmap** will hold the image of the tile set in a form that can be used on the form for drawing.

There are a few ways to create the events that go with the various controls on a form. To create the default event for a control you can double click it in design view and Visual C# will generate the necessary code for you. You can also click the little lightning bolt icon in the **Properties** window of the control and find the event that you want to create by double clicking the event name. I will be doing it manually like I did for the **TileDisplay** control in the last tutorial to make it easier for you to follow what I'm doing. If you remember from the last tutorial I created a **MenuStrip** for the form. Under the **Tile Set** menu there is an option **Open Tile Set.** There is also a **NumericUpDown** control that will control which tile we are drawing. Change the constructor of the **Form1** class to the following:

```
public Form1()
{
```

```
    InitializeComponent();

    tileDisplay1.OnInitialize +=
        new EventHandler(tileDisplay1_OnInitialize);
    tileDisplay1.OnDraw += new EventHandler(tileDisplay1_OnDraw);

    openTileSetToolStripMenuItem.Click +=
        new System.EventHandler(openTileSetToolStripMenuItem_Click);
    nudCurrentTile.ValueChanged +=
        new EventHandler(nudCurrentTile_ValueChanged);
}
```

The first line will handle when the user selects the **Open Tile Set** menu option. The second will handle when the user changes the value in the **NumericUpDown** control. Again, when you type the += you can just hit the tab key twice to fill in the necessary code for the event handlers. This is the code for those two methods and two helper methods called: **ProcessTileSet** and **FillPictureBox**. I will explain the code after you have read it.

```
private void openTileSetToolStripMenuItem_Click(object sender, EventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();

    openFileDialog.Filter = "(Tileset *.tset)|*.tset";
    openFileDialog.CheckFileExists = true;
    openFileDialog.CheckPathExists = true;
    openFileDialog.Multiselect = false;

    input = new XmlDocument();

    DialogResult dialogResult = openFileDialog.ShowDialog();

    if (dialogResult == DialogResult.OK)
    {
        try
        {
            input.Load(openFileDialog.FileName);
        }
        catch
        {
            MessageBox.Show(
                "Error reading tileset.",
                "Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }
        ProcessTileSet(Path.GetFullPath(openFileDialog.FileName));
    }
}

void ProcessTileSet(string filePath)
{
    tileset = new Tileset();
    foreach (XmlNode node in input.DocumentElement.ChildNodes)
    {
        if (node.Name == "TextureElement")
        {
            tileset.textureName = node.Attributes["TextureName"].Value;
        }
```

```
    if (node.Name == "TilesetDefinitions")
    {
        tileset.tileWidth = Int32.Parse(node.Attributes["TileWidth"].Value);
        tileset.tileHeight = Int32.Parse(node.Attributes["TileHeight"].Value);
    }

    if (node.Name == "TilesetRectangles")
    {
        List<Rectangle> rectangles = new List<Rectangle>();

        foreach (XmlNode rectNode in node.ChildNodes)
        {
            if (rectNode.Name == "Rectangle")
            {
                Rectangle rect;
                rect = new Rectangle(
                    Int32.Parse(rectNode.Attributes["X"].Value),
                    Int32.Parse(rectNode.Attributes["Y"].Value),
                    Int32.Parse(rectNode.Attributes["Width"].Value),
                    Int32.Parse(rectNode.Attributes["Height"].Value));
                rectangles.Add(rect);
            }
        }

        tileset.tiles = rectangles;
    }
}
string fileName = Path.GetDirectoryName(filePath);

fileName += @"\" + tileset.textureName;

if (File.Exists(fileName + ".png"))
{
    texture = Texture2D.FromFile(GraphicsDevice, fileName + ".png");
    tilesetBitmap = Bitmap.FromFile(fileName + ".png");
}
else if (File.Exists(fileName + ".jpg"))
{
    texture = Texture2D.FromFile(GraphicsDevice, fileName + ".jpg");
    tilesetBitmap = Bitmap.FromFile(fileName + ".jpg");
}
else if (File.Exists(fileName + ".tga"))
{
    texture = Texture2D.FromFile(GraphicsDevice, fileName + ".tga");
    tilesetBitmap = Bitmap.FromFile(fileName + ".tga");
}
else if (File.Exists(fileName + ".bmp"))
{
    texture = Texture2D.FromFile(GraphicsDevice, fileName + ".bmp");
    tilesetBitmap = Bitmap.FromFile(fileName + ".bmp");
}
else if (File.Exists(fileName + ".gif"))
{
    texture = Texture2D.FromFile(GraphicsDevice, fileName + ".gif");
    tilesetBitmap = Bitmap.FromFile(fileName + ".gif");
}
else
{
    MessageBox.Show("Unrecognized image format in the tile set.",
```

```
                "Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            pbCurrentTile.Image = null;
            tileset = null;

            nudCurrentTile.Value = 0;
            nudCurrentTile.Maximum = 0;

            this.Invalidate();
            return;
        }

        FillPictureBox(0);
        nudCurrentTile.Value = 0;
        nudCurrentTile.Maximum = tileset.tiles.Count - 1;
    }

    private void FillPictureBox(int index)
    {
        Image image = (Image)new Bitmap(128, 128);

        Rect destRectangle = new Rect(0, 0, 128, 128);
        Rect sourceRectangle = new Rect(
                tileset.tiles[index].X,
                tileset.tiles[index].Y,
                tileset.tiles[index].Width,
                tileset.tiles[index].Height);

        Graphics gi = Graphics.FromImage(image);
        gi.DrawImage(tilesetBitmap,
            destRectangle,
            sourceRectangle,
            GraphicsUnit.Pixel);
        pbCurrentTile.Image = image;
        this.Invalidate();
    }

    private void nudCurrentTile_ValueChanged(object sender, EventArgs e)
    {
        if (tileset != null)
        {
            int index = (int)nudCurrentTile.Value;
            FillPictureBox(index);
        }
    }
```

In the **openTileSetToolStripMenuItem_Click** method is where I will handle opening a tile set. To be able to select which file to open I created a **OpenFileDialog** control dynamically. That control is used to display a dialog box that can be used to select a file to open. I set a few of the properties for the **OpenFileDialog** next. The first property is the **Filter** property. The **Filter** property is used to limit the types of files you would like listed in the dialog box. We only want files that end with **tset**. The format of the **Filter** property may be a little confusing. There are two parts to it and they are separated by the | character. The first part is what is displayed in the **File Types** box in the dialog. The second part is what limits the type of files you want to open. The next three properties are all bool properties. The first one **CheckFileExists** keeps the user from selecting a file that does not exist. The second property is

CheckPathExists this is like the first. It checks to make sure the directory the user selected exists. These are more for if the user types in a value in the **File name** box. The last one, **Multiselect**, makes it so that the user can only select 1 file.

Next I create a new **XmlDocument** for the **input** field. To actually display the **OpenFileDialog** you use the **ShowDialog** method of the **OpenFileDialog** class. This method returns a **DialogResult** enum and I captured that result into the **dialogResult** variable.

What I do next is check to make sure that **dialogResult** is **DialogResult.OK**. This result will mean that the user has selected a file. Inside that if statement there is a **try-catch** block. The reason I used a **try-catch** block is to keep the program from crashing if something unexpected happens. First I try and load in the file using the **Load** method like I did in the **TilesetImporter**. I then try and process the file by passing the full path to the file to the **ProcessTileSet** method.

The **ProcessTileSet** method has as a parameter a string **filePath**. This parameter is the file name of the tile set, with the full path. The first thing I do in the **ProcessTileSet** method is create a new instance of the **Tileset** class to hold the tile set. The next block of code will look very familiar to you as it was copied out of the **TilesetProcessor** class. I have pasted how this block of code works from tutorial 21.

Inside a **foreach** loop I loop through all of the child nodes of the root element. You can get the root element of an XML document using the **DocumentElement** property. The **DocumentElement** property has a collection called **ChildNodes**. This collection has all of the children of the root element. I used a foreach loop instead of reading them in order in case you want to create your own XML file and don't put the elements in order.

There are three if statements in side the foreach loop. The first one checks to see if the current node is the **TextureElement**. If it is the **TextureElement** I get the value of the **TextureName** attribute using the Value property of the Attributes property for the node. The Attributes property is a collection of attributes for the node.

In the second if statement I check to see if the current node is **TilesetDefinitions**. I then use the **Parse** method the **Int32** to convert the sting values of the attributes to integers. I set **tileWidth** and **tileHeight** to the value of the attributes **TileWidth** and **TileHeight.**

In the third if statement checks to see if the current node is **TilesetRectangles**. Inside that if statement I create a **List<Rectangle>** to hold all of the rectangles. Then I have another foreach loop. That loop will loop through all of the children of the current node. There is an if statement inside that loop to make sure that the name of the child is **Rectangle**. If the child is a rectangle I create using the attribute of the node.

There is a small problem that needs to be resolved. The tile set only has the name of the texture with out the extension. To load in the texture for the tile set with out the **ContentManager** class and to be able to use it to display the current tile we are drawing with we will need to know the extension of the texture. What I did was create a variable **fileName** that I first set to the directory we are working in using the **GetDirectoryName** method of the **Path** class. Then I append the name of the texture to the string **fileName**.

There is now a series of if-else if statements. I first check to see if **fileName** plus **.png** exists

using the **Exists** method of the **File** class. If that file exists then the texture for the tile set is a **.png** file. I then use the **FromFile** method of the **Texture2D** class to load in the texture. The overload of the method takes as a parameter a **GraphicsDevice** object and the file name of the texture with the extension. Then I use the **FromFile** method of the **Bitmap** class to load in the image. The **Bitmap** class is part of the **System.Drawing** namespace and is for handling images in **GDI+**, which is what Windows uses for rendering normally.

In the four following else if statements I check for other image file formats: **.jpg**, **.tga**, **.bmp** and **.gif**. If any of those exist I load in the **Texture2D** and the **Bitmap** using the correct extension. There is an else clause for if the program can not find an image to go with the tile set. What happens is I display a **MessageBox** that says that an image could not be found, set the **Image** property of **pbCurrentTile** to **null** so nothing will be displayed in the control and set **tileset** to null. The reason I set **tileset** to **null** is when I try to do any rendering into the **TileDisplay** control I will check to make sure that there is a **Tileset** object that can be used to do the rendering. If there isn't I won't try to render anything. I then set the **Value** property of **nudCurrentTile** to 0 and the **Maximum** property to 0 as well. I then call the **Invalidate** method of the form to tell the form to redraw itself to have the changes I made to the form visible. Then I use the **return** statement to exit the method.

If everything so far has worked I call the **FillPictureBox** method passing in 0 as the argument. I then set the **Value** property of **nudCurrentTile** to 0 and the **Maximum** property to the number of tiles in the tileset minus 1 using the **Count** property of **tiles**. The reason I subtract 1 is because **tiles** is 0 based which means the first tile starts at 0 and the last tile will be the number of tiles minus 1. I again call the **Invalidate** method of the form to tell it that it needs to redraw itself.

The next method **FillPictureBox** took me a little while to get it right. The **PictureBox** control has an **Image** property which takes an **Image** object so I created an **Image** using the **Bitmap** class that is 128 pixels by 128 pixels, the size of the **pbCurrentTile PictureBox**. I create a **Rectangle** using the **System.Drawing.Rectangle** class called **destRectangle** that is the size of the **PictureBox** to hold the tile from the tile set. I then create a **Rectangle** again using the **System.Drawing.Rectangle** class that will be the source rectangle in the tile set image. I then create a **Graphics** object, **gi**, from **image** using the **FromImage** method of the **Graphics** class. This class is used like **SpriteBatch** to do rendering in Windows. I then use the **DrawImage** method of the **Graphics** class to draw the current tile in the tile set using the **tilesetBitmap** as the source image. I pass in the destination and source rectangles much like you would do when rendering in XNA. The final parameter **GraphicsUnit.Pixel** says that we are drawing in pixels. I then set the **Image** property of **pbCurrentTile** to this image and finally tell the form to redraw itself by calling **this.Invalidate**.

The last method is the **nudCurrentTile_ValueChanged** method. This method is responsible for updating **pbCurrentTile** to show the current tile you are drawing with. All this method does is make sure that there is a valid tile set to draw with. If there is it gets the current tile using the **Value** property of **pbCurrentTile**. This property is of type decimal so you need to cast it to int. It then calls **FillPictureBox** passing in the appropriate value.

That is a lot to digest. But I will do one more thing in this tutorial. I will show you that the **TileDisplay** control does actual rendering by rendering a simple tile map. What I will do is when you change the **nudCurrentTile** control I will create an array of int inside the **OnDraw** method and render a map with it. First you will want to tell the **TileDisplay** to redraw itself when you change the current tile. Modify the **nudCurrentTile_ValueChanged** method to the following.

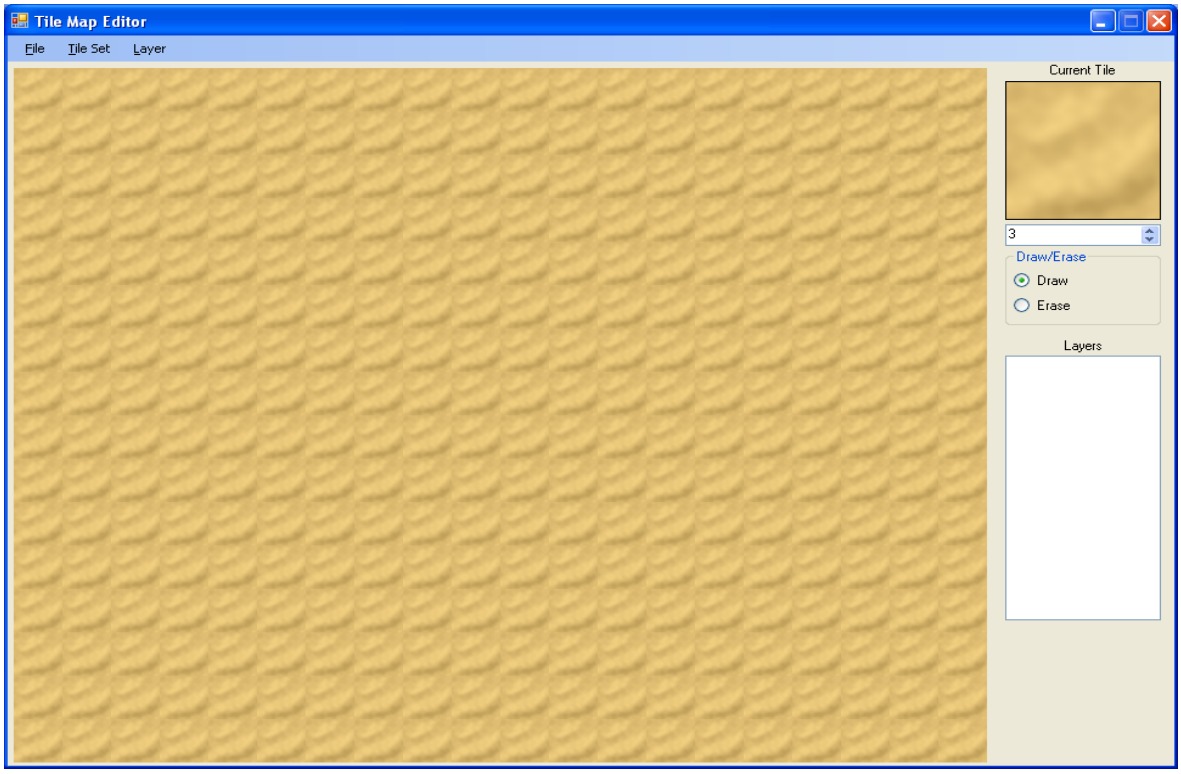
```
private void nudCurrentTile_ValueChanged(object sender, EventArgs e)
{
    if (tileset != null)
    {
        int index = (int)nudCurrentTile.Value;
        FillPictureBox(index);
        tileDisplay1.Invalidate();
    }
}
```

Now you will want to modify the **tile1Display_OnDraw** method on **Form1** to render a simple map. Just change the method to the following.

```
void tileDisplay1_OnDraw(object sender, EventArgs e)
{
    int[,] map = new int[50, 50];

    GraphicsDevice.Clear(Color.CornflowerBlue);
    if (tileset != null)
    {
        for (int y = 0; y < 50; y++)
            for (int x = 0; x < 50; x++)
                map[y, x] = (int)nudCurrentTile.Value;
        spriteBatch.Begin();
        for (int y = 0; y < 50; y++)
            for (int x = 0; x < 50; x++)
                spriteBatch.Draw(
                    texture,
                    new Rectangle(
                        x * 40,
                        y * 40,
                        40,
                        40),
                    tileset.tiles[map[y, x]],
                    Color.White);
        spriteBatch.End();
    }
}
```

All the above code does is create a 2D array of int called **map**. If there is a valid tileset inside a set of nested for loops I fill **map** to the **Value** property of **nudCurrentTile**. Then inside of a call to **Begin** and **End**. I render the map using 40 as the witdh and height of the tiles. That code should be pretty familiar to you by now. If you run the program and click the **Open Tile Set** menu option and navigate to the **Tileset** folder in the **Content** folder and open the tileset you should be able to get a result like this.



Well that is it for this tutorial. I am already working on coding the next part of Eyes of the Dragon so 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 these tutorials.