Creating a Role Playing Game with XNA Game Studio 3.0
Part 29
Tile Map Editor – Part 3

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

Looking back on what I was doing with the Tile Map Editor I found that I made a bad design. For the maps you only want one tile set at a time. What I had done up until now would allow for more than one tile set. I found a good solution to only having one tile set available at a time.

Once you have the last version of the project loaded the first thing that you are going to want to do is right click the **Tile Set** menu option in the **MenuStrip** and select **Delete**. This will remove the menu from the form as well as all the items under it. This will make an error in your code. First find the **openTileSetToolStripMenuItem_Click** method and delete it. Now find the following lines of code in the constructor for **Form1** and remove them.

```
    openTileSetToolStripMenuItem.Click +=
        new System.EventHandler(openTileSetToolStripMenuItem_Click);
```

What I am going to do next is add a tile engine to the project. It will be based on the tile engine for the game but it will not be exact but it will function some what like the tile engine from the game. What you will want to do first is right click the **TileMapEditor** project and select **Add New Folder** and name the folder **TileEngine**. Instead of adding a **TileEngine** class I am just going to add a class called **Engine**. Right click the **TileEngine** folder and add a new class called **Engine** and add the following code to the class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TileMapEditor.TileEngine
{
    public static class Engine
    {
        static int tileWidth = 40;
        static int tileHeight = 40;

        static int viewPortWidth = 800;
        static int viewPortHeight = 640;

        public static int TileWidth
        {
            get { return tileWidth; }
        }
        public static int TileHeight
        {
            get { return tileHeight; }
        }

        public static int ViewPortWidth
        {
            get { return viewPortWidth; }
        }

        public static int ViewPortHeight
        {
            get { return viewPortHeight; }
        }
    }
}
```

As you can see the code is practically the same as the code as the **TileEngine** class in the game. The differences is that for the tile width and heigh I chose 40 so the tiles would fit nicely where the map is being rendered. The **viewPortWidth** and **viewPortHeight** values are the size of the **tileDisplay1** control 800 for **viewPortWidth** and 640 for **viewPortHeight**. I also removed the one property **ViewPortVector**. It is okay that this tile engine is not overly efficient and it will simplfy things.

What I am going to add next is a simple 2D camera. I'm not going to worry about the transformation matrix for this camera. Right click the **TileEngine** folder in the **TileMapEditor** project and add a new class called **Camera**. This is the code for the **Camera** class.

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

namespace TileMapEditor.TileEngine
{
    public class Camera
    {
        public Vector2 Position;

        public Camera()
        {
            this.Position = new Vector2();
        }
    }
}
```

Since I will be using a **Vector2** for the camera I need a using statement for the XNA framework. I will be doing all of the validation on the camera in the editor so I made the position of the camera public. I also didn't need a speed for the camera or a method to lock the camera.

The next class I will add is for the layers of the map. The reason I didn't add the class for the map first is that I need the class for the layers inside of the map class. Add a new class to the **TileEditor** folder called **TileMapLayer**. This is the code for the **TileMapLayer** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TileMapEditor.TileEngine
{
    public class TileMapLayer
    {
        int[,] map;

        public TileMapLayer(int width, int height)
        {
            map = new int[height, width];
        }

        public void SetTile(int x, int y, int tileIndex)
        {
            map[y, x] = tileIndex;
        }

        public int GetTile(int x, int y)
        {
            return map[y, x];
        }

    }
}
```

This is a stripped down version of the **TileMapLayer** class from the game. I will not be doing the rendering of the layers inside the class, I will be doing it in the form. For fields all there is in this class is the array of integers for the map. There is a constructor that takes as parameters the **width** and the **height** of the map. There is a method **SetTile** to set a tile in the map and another method **GetTile** to get a tile in the map.

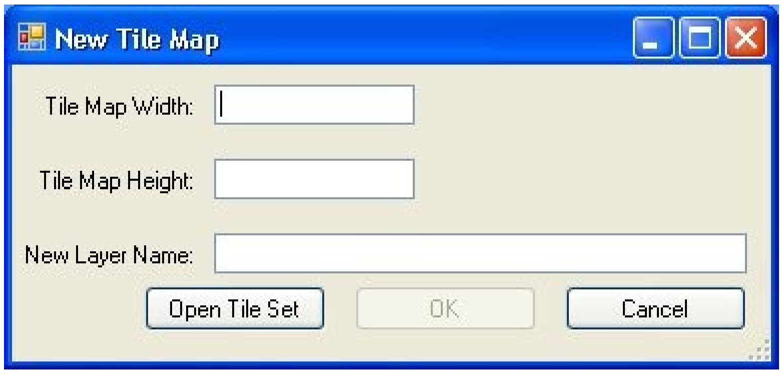
I will need a tile map class. I again will use a scaled down class for this. Right click the **TileEngine** folder in the **TileMapEditor** solution and add a new class called **TileMap**. This is the code for the class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TileMapEditor.TileEngine
{
    public class TileMap
    {
        public List<TileMapLayer> layers = new List<TileMapLayer>();
        int mapWidth;
        int mapHeight;

        public TileMap(int mapWidth, int mapHeight)
        {
            this.mapWidth = mapWidth;
            this.mapHeight = mapHeight;
            TileMapLayer layer = new TileMapLayer(mapWidth, mapHeight);

            for (int y = 0; y < mapHeight; y++)
                for (int x = 0; x < mapWidth; x++)
                    layer.SetTile(x, y, -1);

            layers.Add(layer);
        }

        public int MapWidth
        {
            get { return mapWidth; }
        }

        public int MapHeight
        {
            get { return mapHeight; }
        }

        public int WidthInPixels
        {
            get { return mapWidth * Engine.TileWidth; }
        }

        public int HeightInPixels
        {
            get { return mapHeight * Engine.TileHeight; }
        }
    }
}
```

There are three fields in this class: **layers**, **mapWidth** and **mapHeight**. **layers** is a **List** of **TileMapLayer**. The other two **mapWidth** and **mapHeight** are integers for the width and height of the map in tiles.

The constructor for the class takes as parameters the width and the height of the map in tiles. In the constructor I set the **mapWidth** and **mapHeight** fields. I create an instance of the **TileMapLayer** class. Then I set the tile index of all the tiles to -1 meaning that there is nothing to draw. I then add the layer to the list of layers for the map.

We need a way to create a new map and test out those classes. To do this I will create a second form. Click the **TileMapEditor** project. Then under the **Project** menu select **Add Windows Form** and call the new form **FrmNewMap**. This is what my finished form looks like.



These are the properties you need to change for **FrmNewMap**: **Size** will be **387, 182**,

StartPosition will be **CenterParent** and **Text** will be **New Tile Map**. There are two properties that will have to be set a little later.

For the form you will need three **Labels**, three **TextBoxes** and three **Buttons**. Start by dragging a **Label** onto the form and set the following properties for the **Label**: **(Name)** is **lblMapWith**, **Location** is **13, 14** and **Text** is **Tile Map Width:**. Drag another **Label** onto the form and set the following properties: **(Name)** is **lblMapHeight**, **Location** is **10, 51** and **Text** is **Tile Map Height:**. Drag one last **Label** on the form and set the following properties: **(Name)** is **lblLayerName**, **Location** is **3, 88** and **Text** is **New Layer Name:**.

Now drag a **Textbox** onto the form and set the properties as follows: **(Name)** is **tbMapWidth**, **Location** is **100, 10**, **Size** is **100, 20** and **TabIndex** is **0**. Drag a second **Textbox** onto the form and set the following properties: **(Name)** is **tbMapHeight**, **Location** is **100, 47**, **Size** is **100, 20** and **TabIndex** is **1**. Drag a third **Textbox** onto the form and set its properties as follows: **(Name)** is **tbLayerName**, **Location** is **100, 84** and **TabIndex** is **2**.

Drag a single **Button** onto the form and set these properties: **(Name)** is **btnOpenTileSet**, **Location** is **66, 110**, **Size** is **91, 23**, **TabIndex** is **3** and **Text** is **Open Tile Set**. Now click the **Button** on the form and while holding down the **Control** key drag it to a new location. This will replicate the **Button** on the form. Set the following properties of the new **Button**: **(Name)** is **btnOK**, **Enabled** is **False**, **Location** is **171, 110**, **Size** is **91, 23**, **TabIndex** is **4** and **Text** is **OK**. The reason I set **Enabled** to **False** is that I don't what the user to be able click **OK** until they have loaded a tile set. Repeat the process to add a third **Button** to the form and set these properties: **(Name)** is **btnCancel**, **Location** is **276, 110**, **Size** is **91, 23**, **TabIndex** is **5** and **Text** is **Cancel**.

We are almost done with with the designer. There are just two properties of the form to set. Click the title bar of the form to get its properties. I'm going to set the **AcceptButton** property to **btnOK** and the **CancelButton** property to **btnCancel**. What this will do is if the user presses the **Enter** key it will be the same as if the user clicked the **OK** button. Also, if they press the **Escape** key it will be like they pressed the **Cancel** button. This is just an interesting thing you can do with Windows forms that make life easier for the user.

Now it is time to add the code to the form. Right click **FrmNewMap.cs** in the solution explorer and select **View Code** to bring up the code window. Change the using statements to the following.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml;
using System.IO;
using Microsoft.Xna.Framework;
```

I had removed the using statement for **System.Drawing** and added in the using statement for **Microsoft.Xna.Framework**. There are also using statements for the **System.Xml** namespace because I will try and load the **Tileset** in this form and the **System.IO** namespace for the same reason. Now just after the curly brace for the class definition add the following fields.

```
public int MapWidth;
public int MapHeight;
public Tileset tileset;
public string TilesetTextureName;
public string TileLayerName;
public bool OkClicked = false;

XmlDocument input;
```

Most of the fields for this class are public. This is bad object-oriented programming but I decided to go for ease of use and shorter code over strict object-oriented programming. The reason the fields are public is after the user is done with the form and they have clicked the **OK** button or pressed **Enter** I need a way to get the information they entered into the form. **MapWidth** and **MapHeight** will of course hold the width and the height of the map they want to create. The **tileset** field will hold the tileset they chose to load. **TileSetTexture** will hold the name of the texture for the tile set, with its extension. The next one might not be so obvious. The reason I chose to have a **Textbox** hold the name of the layer is so that they can be differentiated in the **CheckedListBox** on the form. That is the reason why I added the **Textbox** and have the field **TileLayerName**. The next one will tell if the user has pressed the **OK** button. The last field **input** is from the previous tutorial. I will try and load in the tile set in this form and retrieve it from the other form.

The code for the constuctor for this class is below. Remember that after you type the += for the event hanlders you can press **Tab** twice to generate the base code for them. This is the code for the constructor.

```
public FrmNewMap()
{
    InitializeComponent();
    this.Load += new EventHandler(FrmNewMap_Load);
    btnOpenTileSet.Click += new EventHandler(btnOpenTileSet_Click);
    btnOK.Click += new EventHandler(btnOK_Click);
    btnCancel.Click += new EventHandler(btnCancel_Click);
}
```

Remember that when you are generating event handlers you need to do it after the method call **IntializeComponent** because until that method is called the controls don't exist. The first event handler is for when the form loads. In this event I will just be making sure that the **btnOK** is disabled and that **OkClicked** is set to **false**. The second event handler is for when **btnOpenTileset** has been clicked. The third and forth handlers are for when **btnOK** has been clicked, or if the user presses the **Enter** key, and when **btnCancel** has been clicked, or the user presses the **Escape** key, respectively. Next I will give you the code for the event handlers plus a helper method **ProcessTileSet**.

```
void FrmNewMap_Load(object sender, EventArgs e)
{
    btnOK.Enabled = false;
    OkClicked = false;
}

private void btnOpenTileSet_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);
            ProcessTileSet(Path.GetFullPath(openFileDialog.FileName));
        }
        catch
        {
            MessageBox.Show(
                "Error reading tileset.",
                "Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            tileset = null;
            btnOK.Enabled = false;
            return;
        }
    }
}
```

```
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;

    string[] extensions = { ".png", ".jpg", ".tga", ".bmp", ".gif" };
    bool found = false;

    foreach (string extension in extensions)
    {
        if (File.Exists(fileName + extension))
        {
            found = true;
            fileName += extension;
            break;
        }
    }

    if (found)
    {
        TilesetTextureName = fileName;
        btnOK.Enabled = true;
    }
```

```
        else
        {
            MessageBox.Show("Unrecognized image format in the tile set.",
                "Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            tileset = null;
            btnOK.Enabled = false;
        }
    }

    void btnCancel_Click(object sender, EventArgs e)
    {
        btnOK.Enabled = false;
        OkClicked = false;
        this.Close();
    }

    void btnOK_Click(object sender, EventArgs e)
    {
        if (!int.TryParse(tbMapWidth.Text, out MapWidth))
        {
            MessageBox.Show("You must enter a numeric value for Tile Map Width",
                "Tile Map Width Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            return;
        }

        if (!int.TryParse(tbMapHeight.Text, out MapHeight))
        {
            MessageBox.Show("You must enter a numeric value for Tile Map Height",
                "Tile Map Height Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            return;
        }

        if (MapWidth < 1)
        {
            MessageBox.Show("Tile Map Width can not be less than 1",
                "Tile Map Width Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            return;
        }

        if (MapHeight < 1)
        {
            MessageBox.Show("Tile Map Height can not be less than 1",
                "Tile Map Height Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            return;
        }
        if (string.IsNullOrEmpty(tbLayerName.Text))
        {
            MessageBox.Show("You must enter a name for the new layer.",
                "Layer Name Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            return;
        }

        btnOK.Enabled = false;
        OkClicked = true;
        TileLayerName = tbLayerName.Text;
        this.Close();
    }
```

As I mentioned earlier all that the method **FrmNewMap_Load** does is make sure that the **Enabled** property of **btnOK** is set to false so the user has to open a tile set. The code for the next event handler **btnOpenTileSet_Click** you saw in the last tutorial. The code was copied right out of the last tutorial. The only difference is in the else part of the if statement. If loading and processing the tile set fails I set **btnOK.Enabled** to false.

The code for the **ProcessTileSet** method, for the most part, is the same as the last tutorial as well. The difference being after I load and process the tile set I changed the way that I check for what the file name of the texture for the tile set is. Instead of all the if-esle statements I create a small array of all the extensions I want to check called **extensions**. The extensions I'm checking for are **.png**, **.jpg**, **.tga**, **.bmp** and **.gif**. If there is an extension that you would like to check for you can just add it to the list. I also have a boolean variable called **found** that is set to **false**. I next use a foreach loop to loop through the extensions. Inside the loop I check to see if a file with that extension exists if it does I set **found** to **true** meaning that I found the file I then add the extension to **fileName**. I then have a **break** statement. If you are not familiar with it what that statement does is exit the loop. Finally I check to see if the variable **found** is **true**. If it is I set **TilsetTextureName** to **fileName** and set **btnOK.Enabled** to **true** because there is now a valid **Tileset** object. Otherwise I display a message box saying the image for the tile set could not be found, set **tileset** to **null** meaning there is no **tileset** to work with and make sure that **btnOK.Enabled** is set to **false**.

The next method, **btnCancel_Click** handles if **btnCancel** has been clicked. This method sets **btnOK.Enabled** to **false**. It then sets **OkClicked** to **false** which means that the user did not click **btnOK**. Finally it class **this.Close** which closes the form.
The last method in this class is **btnOK_Click**. This method does a lot of validation to make sure that the user has entered correct values. You should get used to doing validation when you are writing code to make sure that unexpected values will not crash your program. You would for example want a web browser to crash if you typed something wrong for the URL of the web site you wish to load.

In the first if statement I used **int.TryParse** to try and parse **tbMapWidth.Text**. You might be unfamiliar with one part of this method. It the second arguement of the method there is the keyword **out**. Methods can only return one value. The **out** keyword allows you to change a value inside of the method to get around this restriction. The **TryParse** method returns true if the conversion from a string to a integer is successful. If the conversion fails an error message is shown and the method exits. The next if statement tries to convert **tbMapHeight.Text** to and integer. If it fails an error message is shown and the method exits. The next if statements checks to see if the **MapWidth** field is less than 1. You could impose a restriction that maps must be at least a certain width but less than 1 just makes sure that the program won't crash. If **MapWidth** is less than 1 a message is displayed and the method exits. The next statement performs a similar test on **MapHeight** and displays a message and exits. The last if statement checks to make sure there is something in **tbLayerName.Text** using the **IsNullOrEmpty** method. If that fails a message is displayed and the method exits. If all tests pass **btnOK.Enabled** is set to **false** meaning that the next time the form is shown the button will be disabled. **OkClicked** is set to **true** meaning that the form was processed correctly**.** The **TileLayerName** field is set to **tbLayerName.Text** and then the form closes.

There is something that you will need to do before we get back to the other form. You will need a cursor to display so the user knows which tile they are working on. What you will want to do is right click the **TileMapEditor** project and add a new folder called **Content**. Now you will need an image for the cursor. You can find an image I made at http://xna.jtmbooks.com/Cursor.zip. Download the file, extract the file and right click the **Content** folder in the **TileMapEditor** project and select **Add** and then **Existing item,** make sure that you select **Images** from **Objects of type** combo box. Navigate to where you extracted the file and select **cursor.png**. Now there is one thing else that you will need to do. Click the **cursor.png** file and then go to the **Properties** window. There will be an item **Copy To Output Directory**. You want to set that to **Copy always** to make sure that when you build the project the file will be there.

There is really no easy way to go over the changes to the code for **Form1**. What I will do is just give you the code and try to explain it. Right click **Form1** and choose **View Code**. This is the updated code for **Form1**.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

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;

using TileMapEditor.TileEngine;

namespace TileMapEditor
{
    public partial class Form1 : Form
    {
        SpriteBatch spriteBatch;
        Tileset tileset = null;
        Texture2D texture;
        Image tilesetBitmap;
        Texture2D cursor;
        TileMap tileMap = null;
        Camera camera = new Camera();
        FrmNewMap frmNewMap = new FrmNewMap();
```

```
public GraphicsDevice GraphicsDevice
{
    get { return tileDisplay1.GraphicsDevice; }
}

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

    tileDisplay1.OnInitialize +=
        new EventHandler(tileDisplay1_OnInitialize);

    tileDisplay1.OnDraw += new EventHandler(tileDisplay1_OnDraw);

    newMapToolStripMenuItem.Click +=
        new EventHandler(newMapToolStripMenuItem_Click);

    nudCurrentTile.ValueChanged +=
        new EventHandler(nudCurrentTile_ValueChanged);

    Application.Idle += new EventHandler(Application_Idle);
}

void Application_Idle(object sender, EventArgs e)
{
    tileDisplay1.Invalidate();
}

void tileDisplay1_OnInitialize(object sender, EventArgs e)
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    cursor = Texture2D.FromFile(GraphicsDevice, @"Content\cursor.png");
}

void tileDisplay1_OnDraw(object sender, EventArgs e)
{
    Logic();
    Render();
}

private void Render()
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    if (tileMap != null)
    {
        foreach (TileMapLayer layer in tileMap.layers)
        {
            spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
            for (int y = 0; y < tileMap.MapHeight; y++)
                for (int x = 0; x < tileMap.MapWidth; x++)
                    if (layer.GetTile(x, y) != -1)
                    {
                        spriteBatch.Draw(texture,
                            new Rectangle(
                                x * Engine.TileWidth
                                    - (int)camera.Position.X,
                                y * Engine.TileHeight
                                    - (int)camera.Position.Y,
                                Engine.TileWidth,
                                Engine.TileHeight),
                                tileset.tiles[layer.GetTile(x, y)],
                                Color.White);
                    }
            spriteBatch.End();
        }

        spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
        spriteBatch.Draw(cursor, camera.Position, Color.Red);
        spriteBatch.End();
    }
}

private void Logic()
{
}

void newMapToolStripMenuItem_Click(object sender, EventArgs e)
{
    frmNewMap.ShowDialog();

    if (frmNewMap.OkClicked)
    {
        tileset = frmNewMap.tileset;
        texture = Texture2D.FromFile(GraphicsDevice,
            frmNewMap.TilesetTextureName);
        tilesetBitmap = Bitmap.FromFile(frmNewMap.TilesetTextureName);
        tileMap = new TileMap(frmNewMap.MapWidth, frmNewMap.MapHeight);
        layerToolStripMenuItem.Enabled = true;

        clbLayers.Items.Add(frmNewMap.TileLayerName, true);

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

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);
        tileDisplay1.Invalidate();
    }
}
}
}
```

The first new thing is there is a using statement for the **TileMapEditor.TileEngine** namespace to allow us to use the classes in that namespace. There are three new fields in the class. The first one is **tileMap** which will hold the map you are working on. The next one is **camera** which will hold the **Canera** object. The last one may be new to you if you haven't work with Windows forms before. It is an object of the class **FrmNewMap** called **frmNewMap**. This field will be used for displaying the form for creating a new map.

The constructor has changed a little. After the call to **InitializeComponents** I set the **Enabled** property of **layerToolStripMenuItem** to **false** so that a layer can not be worked on until there is a map to work on. I then create the event handlers for **tileDisplay1** like before. Next I create an event handler for the **Click** event of **newMapToolStripMenuItem** so that when you click the **New Map** item in the menu the program will display the new form and create a new map if everything goes according to plan. Next I create the event handler for when the user changes the value of **nudCurrentTile**. The last line creates an event handler for **Application.Idle**. What this does is whenever the application isn't doing something import it will call this event. I will be using this event handler to tell **tileDisplay1** to call the **Invalidate** method and redraw itself. After the constructor comes the **Application.Idle** event handler. All that it does is call **tileDisplay1.Invalidate** which redraws the control.

In the **tileDisplay1.OnDraw** method I call two new method **Logic** and **Render**. The first method will preform the logic for the map and the second will do the drawing of the map. When you type **Render(** Intellisense is going to kick in and you will get **RenderRightToLeft(**. Sometimes Intellisense can be frustrating. Just remove the **RightToLeft** part.

Next is the **Render** method. The first thing the **Render** method does is call the **Clear** method to erase everything from the control. It then will only do rendering if there is a tile map available. The code for rendering the map should be familiar to you by now. It goes through all of the layers in the map and if the tile is not -1 it will draw the tile. After rendering the map it will draw the cursor tinted red. If you look at the image of the cursor it is just a white box with lots of transparency. If you like a color other than red go ahead and use it. At the moment there is no code in the **Logic** method. I will be adding that code in the next tutorial.

The next method is the **newMapToolStripMenuItem_Click** method. In this method I first display the form calling **ShowDialog**. What this does is make it so that the user can not do anything with the editor until they have finished with the form. It then checks to see if the form was closed by the user clicking the **OK** button on the form. If the user did click the **OK** button it sets the **tileset** field to **frmNewMap.tilset**. It then loads in the texture for the tile set using the **TilesetTextureName** field of **frmNewMap**. It also loads in the bitmap using the same field. It then creates the **TileMap** object using

frmNewMap.MapWidth and **frmNewMap.MapHeight**. It sets **layerToolStripMenuItem.Enabled** to **true** so the user can work with the layers. It then adds a new item to **clbLayers** with the **Checked** property set to **true**. The method then calls the **FillPictureBox** method passing in 0 as the index of the tile to be displayed. It sets **nudCurrentTile.Value** to 0, the first tile, and **nudCurrentTile.Maximum** to the number of tiles minus 1. It then calls **this.Invalidate** to let the form know it needs to be redrawn.

The last two methods are the same as before. This tutorial was a little longer than I thought it would be but I do believe that it needed to all be done. 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.