I'm writing these tutorials for the new XNA 4.0 framework. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the [XNA 4.0 RPG tutorials page](http://xna4gpa.net/xna4/downloads/tilesets.zip) of my web site. I will be making my version of the project available for download at the end of each tutorial. It will be included on the page that links to the tutorials.

A major part of a role playing game is exploring the map of the world. This is a 2D role playing game and uses tiling. Tiling is the process of creating a larger image using many smaller images. Each of the smaller images is called a tile. Drawing the map is done using a tile engine. There are a few different types of tile engines, I will be using the most basic.

The first thing that you will need is tiles. There are a number of places that you can get tiles from on the web. I will be using tiles from [OpenGameArt.org](http://open GAMEART.ORG). This is a site that offers free graphics for open source games. This game is an Open Source game so I thought I would use the graphics from there.

Now that you have tiles you have a decision to make. You can either create tile sets or use separate images. I will use a tile set. A tile set is an image that is made up of the individual tiles. You can download the tile set from this link: [http://xnagpa.net/xna4/downloads/tilesets.zip](http://xnagpa.net/xna4/downloads/tilesets.zip).

After you have downloaded the tile set and unzipped it open up your game in Visual C#. Right click the `EyesOfTheDragonContent` project, select **Add** and then **New Folder**. You can name this new folder `Tilesets`. Next, right click the `Tilesets` folder select **Add** and then **Existing Item**. Navigate to where you extracted the tile set and add the `tileset1.png` file.

The next step is to create the tile engine. I will be making a layered tile engine. Your map will be made up layers. On the different layers you will have different elements of the map. I will also be making an editor for the maps to make your life easier. Right click your `XRpgLibrary` project in the solution explorer, select **Add** and then **New Folder**. Call this folder `TileEngine`. You will want a general class that holds information about the tiles and the map. Right click the `TileEngine` folder, select **Add** and then **Class**. Call this class `Engine`. This is the code for the `Engine` class.

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

namespace XRpgLibrary.TileEngine
{
    public class Engine
    {
        #region Field Region
        static int tileWidth;
```
The **Engine** class holds the width and the height of the tiles on the screen. It may be developed more down the road as the game progresses. The class uses the **Point** and **Vector2** classes of the XNA Framework so I added a using statement for that. There are static fields in the **Engine** class and static read only properties to get their values. The **tileWidth** and **TileWidth** field and property are for the width of the tiles on the screen. The **tileHeight** and **TileHeight** field and property are for the height of the tiles on the screen. The constructor of the class takes as parameters the width and height of the tiles. The **VectorToCell** method is used to get the position of a **Vector2** in tiles on the map.

Since I decided to go with a tile set over a list of tiles the next thing I added was a class for tile sets. Right click the **TileEngine** folder, select **Add**, and then **Class**. Name this class **Tileset**. This is the code for the **Tileset** class.

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

namespace XRpgLibrary.TileEngine
{
    public class Tileset
    {
        #region Fields and Properties
```
Texture2D image;
int tileWidthInPixels;
int tileHeightInPixels;
int tilesWide;
int tilesHigh;
Rectangle[] sourceRectangles;

#endregion
#region Property Region

public Texture2D Texture
{
    get { return image; }
    private set { image = value; }
}

public int TileWidth
{
    get { return tileWidthInPixels; }
    private set { tileWidthInPixels = value; }
}

public int TileHeight
{
    get { return tileHeightInPixels; }
    private set { tileHeightInPixels = value; }
}

public int TilesWide
{
    get { return tilesWide; }
    private set { tilesWide = value; }
}

public int TilesHigh
{
    get { return tilesHigh; }
    private set { tilesHigh = value; }
}

public Rectangle[] SourceRectangles
{
    get { return (Rectangle[])sourceRectangles.Clone(); }
}
#endregion
#region Constructor Region

public Tileset(Texture2D image, int tilesWide, int tilesHigh, int tileWidth, int tileHeight)
{
    Texture = image;
    TileWidth = tileWidth;
    TileHeight = tileHeight;
    TilesWide = tilesWide;
    TilesHigh = tilesHigh;

    int tiles = tilesWide * tilesHigh;

    sourceRectangles = new Rectangle[tiles];

    int tile = 0;

    for (int y = 0; y < tilesHigh; y++)
    for (int x = 0; x < tilesWide; x++)
    {
        sourceRectangles[tile] = new Rectangle(
This class just keeps everything that has to do with the tile set in one place. It holds the image, the width of the tiles in the image, and the source rectangles of the image. The source rectangles describe the tiles in the image. When I get to drawing the tiles they are being drawn using a source rectangle for the tile set they are being drawn from and a destination that describes where they are draw to on the screen. The class requires a `Texture2D` for the tile set and an array of `Rectangles` for the source rectangles so there are using statements for the XNA Framework and the XNA Framework graphics classes. The fields in the class are `image`, `tileWidthInPixels`, `tileHeightInPixels`, `tilesWide`, `tilesHigh`, and `sourceRectangles`. The first is the `Texture2D` for the tile set, the second and third are the width and height of the tiles in pixels, the third and forth are the number of tiles wide and high the image is, and the last is the source rectangles for the tile set. The properties for the class are `Texture`, `TileWidth`, `TileHieght`, `TilesWide`, `TilesHigh`, and `SourceRectangles`. They are public get and private set. The work for the fields list above in the same order. `SourceRectangles` returns a clone of the source rectangles. I do that because if you modify one of the rectangles that are returned you modify it inside the class as well. You are much better to work with a copy of the rectangle.

The constructor for this class takes five parameters. The `Texture2D` for the tile set, the number of tiles wide the tile set is, the number of tiles high the tile set is, the width of each tile in pixels, and the height of each tile in pixels. The constructor then sets the fields with the values passed in. To calculate the number of source rectangles for the tile set you multiply the number of tiles wide by the number of tiles high. The next step is to create an array for the source rectangles. The variable `tile` will be the index of the source rectangle that is being created. The next step is important. There is a set of nested for loops. The first loop loops through the tiles high the tile set is. The second loop loops through the tiles wide the tile set is. What this does is goes through all of the tiles in the image starting at the top left corner, moving left to right first and then top to bottom. This will be important when you are making your maps. The first tile will be index 0 in the map, the one just to the right will be index 1, the following just to the right of that one index 2, and so and so forth. After creating the tile I increment the `tile` variable.

The next thing you need to start tiling is the map. I've designed the map so that you can have multiple layers. I've also designed the map so that you can have multiple tile sets. To allow that instead of just using an integer to describe a tile, I created a class that will hold the tile information. For now it will have an index for the tile and an index for the tile set the tile belongs to. Right click the `TileEngine` folder, select Add and then Class. Name this class `Tile`. This is the code for the `Tile` class.
This is a very simple class. It has two fields. One for the index of the tile in the tile set and one for the index of the tile set. There are properties where the get is public and the set it private. This allows the value of the tile to be accessed outside of the class and set inside of the class. The constructor just assigns the fields with the values passed in.

Before I get much farther, I want to add some simple tiling to the game. That will be done in the GamePlayScreen. I will create a simple map and demonstrate how to tile the map. This is for those who have not read my other XNA RPG tutorials or my XNA simple tile engine tutorials. Open the code for the GamePlayScreen.

I won't use the Tile class quite yet. I will, instead, use an array of integers to represent the tiles. If you look at the image I made with the tiles the first tile is completely transparent. So to show that the map is being drawn properly I created an array of integers filled with ones. I also created a Tileset object and an Engine object. First, add a using statement to bring the TileEngine name space of the library into scope. Also, change the Fields region of the GamePlayScreen to the following.

```csharp
using XRpgLibrary.TileEngine;

Engine engine = new Engine(32, 32);
Tileset tileset;
int[,] map;
```
What I've done is add a field `engine` for the `Engine` class. The tile engine will require a `Tileset` to draw the tiles so there is a field for that as well. The last new field is `map` which is a 2D array of integers. When the tile engine is finished this will be a layer of the map and it will be a 2D array of `Tile` instead of integers. When I created the instance of the `Engine` class I passed in 32 for the width and height of the tiles. This is also the width and height of the tiles in the tile set I made.

You will need to create a `Tileset` object. For the `Tileset` object you will need the `Texture2D` with the tiles. The best thing to do here is to add in the `LoadContent` method in the `GamePlayScreen`. In the `XNA Methods` region of the `GamePlayScreen` class add this `LoadContent` method.

```csharp
protected override void LoadContent()
{
    Texture2D tilesetTexture = Game.Content.Load<Texture2D>("Tilesets\tileset1");
    tileset = new Tileset(tilesetTexture, 8, 8, 32, 32);
    map = new int[50, 50];
    base.LoadContent();
}
```

I use the content manager from our `Game1` class to load in the texture for the tile set. I then create a `Tileset`. I pass in the image of the tiles 8 for the tiles wide and high. For the width and height of the tiles I pass in 32. The reason for these values is the tile sets are eight tiles wide and high. The tiles are 32 pixels wide and high. I chose these values because it is a good idea to have your texture as a power of 2. The tile sets are 256 pixels by 256 pixels that will work out to a power of 2. I also create the array of integer to be 50 by 50.

Now we have everything we need to draw a tile map. Since this involves drawing to the screen that will happen in the `Draw` method. I will show you the code and then explain the code. Add this `Draw` method to the `GamePlayScreen` in the `XNA Methods` region.

```csharp
public override void Draw(GameTime gameTime)
{
    GameRef.SpriteBatch.Begin(
        SpriteSortMode.Immediate,
        BlendState.AlphaBlend,
        SamplerState.PointClamp,
        null,
        null,
        null,
        Matrix.Identity);
    for (int y = 0; y < map.GetLength(0); y++)
    {
        for (int x = 0; x < map.GetLength(1); x++)
        {
            GameRef.SpriteBatch.Draw(
                tileset.Texture,
                new Rectangle(
                    x * Engine.TileWidth,
                    y * Engine.TileHeight,
                    Engine.TileWidth,
                    Engine.TileHeight),
                tileset.SourceRectangles[map[y, x]],
                Color.White);
        }
    }
    base.Draw(gameTime);
    GameRef.SpriteBatch.End();
}
```
When Microsoft released XNA 4.0 they remodeled the SpriteBatch class entirely to make it much more robust. Instead of using enumerations for the way sprites are blended they created a class that controls it. They added in some other classes that give you much more control as well, like being able to sample textures. When they release the first version of XNA people were complaining about some issues with sprites. In particular, when you scale sprites in a sprite sheet, like the tile set, you can get lines around the sprites. This had to do with how the textures for the sprite were loaded and with how the images were sampled. The way some people got around it was to map meshes to vertex buffers like you do in 3D. Another was to read in textures in a different manner. The upgrades to SpriteBatch give you so much more control over the way things are rendered.

So, if you haven't worked much with SpriteBatch in XNA 4.0 the call to Begin, compared to earlier releases, is quite new. There are five overloads to the call. The one I used takes seven parameters. The first is the way sprites are sorted. I chose Immediate. This means as soon as you make a call to Draw the images is sent to the graphics card instead of waiting for the call to End. This mode works well for tile engines.

The next parameter is a BlendState object. There is a static property of the BlendState class that returns an alpha blending state. The tiles have transparency, an alpha channel of 0, so you want alpha blending. There are other states as well.

The third parameter is a SamplerState object. This object helps control the way textures are mapped to the output. The mode I chose, PointClamp, clamps the sampling to points of the texture. This type of sampling is what keeps the sprites from "bleeding" when scaled. I'm not going into the next three parameters that are set to null in this tutorial. The last parameter is why I used this overload over another. Later to control scrolling, and scaling, I will be using matrices. I passed in the identity matrix here. You control scrolling, scaling, rotation, etc. by multiplying matrices together. Using the identity matrix returns the original values with no moving, scaling, or rotation.

In the Draw method there is a nested for loop, much like the nested for loop that I used to create the source rectangles in the constructor of the Tileset class. This may seem a little backwards to you. In math, and drawing in 2D and 3D, the coordinates are (x, y) but when it comes to arrays in C# they are backwards [y, x]. If you tried to do it the other way your map will come out rotated 90 degrees. You can get the size of each dimension of the array using the GetLength method. Since the Y coordinate comes first you use 0 as the parameter to get its length. You pass in 1 to get the size of the X coordinate. These loops work like the loops in the constructor of the Tileset class. The outer loop loops from top to bottom and the inner loop loops from left to right.

Inside the inner loop is where you actually do the drawing. The overload of the Draw method that I used takes four parameters. The first is the Texture2D that you want to draw. The second is the destination rectangle that you want to draw to. The third is the source rectangle in the Texture2D that you want to draw. The last, and is always the last in the Draw method of the SpriteBatch class is the tint color. You get the Texture2D and source rectangle from the Tileset object. The destination rectangle is the interesting part. It has something to do with the x and y values in the map and the width and height of the tiles. The width and height of the destination rectangle is the width and height of the tiles defined in the Engine class. To find the X and Y coordinates you do a little math. For the tile at (0, 0) on the screen you would use (0, 0). For the tile (1, 0) you need to add the width of one tile on the screen to draw it in the right place. Similarly for tile (2, 0) you need to add the width of two tiles to get it to draw the same way. The same is true as you move down the screen. For tile (0, 1) you need to add the height of one tile to get it drawn in the same place. Again, for tile (0, 2) you need to add the height
of two tiles. If you think of the tile (1, 1) you need to add the width and height of the tile. The function for finding the X coordinate is \( x \times \text{tile width} \). The function for finding the Y coordinate is \( y \times \text{tile height} \).

The next step is to create a class to represent a layer of the map. Right click the \textbf{TileEngine} folder, select \textbf{Add} and then \textbf{Class}. Name this class \textbf{MapLayer}. This is the code for the \textbf{MapLayer} class.

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

namespace XRpgLibrary.TileEngine
{
    public class MapLayer
    {
        #region Field Region
        Tile[,] map;
        #endregion

        #region Property Region
        public int Width
        {
            get { return map.GetLength(1); }
        }

        public int Height
        {
            get { return map.GetLength(0); }
        }
        #endregion

        #region Constructor Region
        public MapLayer(Tile[,] map)
        {
            this.map = (Tile[,] )map.Clone();
        }

        public MapLayer(int width, int height)
        {
            map = new Tile[height, width];
            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    map[y, x] = new Tile(0, 0);
                }
            }
        }
        #endregion

        #region Method Region
        public Tile GetTile(int x, int y)
        {
            return map[y, x];
        }

        public void SetTile(int x, int y, Tile tile)
        {
```
The layers won't be responsible for drawing themselves. The map class will do all of the drawing. This just simplifies things and will make reading in maps easier. There is just the one field in the class, `map`, that is a 2D array of type `Tile`. There are also properties for returning the width and height of the layer. The use the same method I used above for drawing the map. You can get the width using the `GetLength` method passing in 1 and the height using the `GetLength` method passing in 0.

There are two constructors in this class. The first one takes a 2D array of `Tile`. This constructor makes a clone of the array. This is again because if you make a change to the array it will change the original array. The second constructor takes as parameters the width and height of the map. It then creates a new array and then initializes the array filled with `Tile` objects. There are three methods in this class. The first, `GetTile`, returns the `Tile` at the given coordinates in the map. The other two are overloads of the same method `SetTile`. The first takes as parameters the X and Y coordinates and a `Tile` object. The second takes as parameters the X and Y coordinates of the tile, the index of the tile in the tile set, and which tile set the tile belongs to.

The next step is to create a class for the map with the layers and tile sets. Right click the `TileEngine` folder in your game, select `Add` and then `Class`. Name this class `TileMap`. This is the code for that class.

```csharp
class TileMap
{
    #region
    List<Tileset> tilesets;
    List<MapLayer> mapLayers;
    #endregion
    #region
    #endregion
    #region
    public TileMap(List<Tileset> tilesets, List<MapLayer> layers)
    {
        this.tilesets = tilesets;
        this.mapLayers = layers;
    }
}
```
public TileMap(Tileset tileset, MapLayer layer)
{
    tilesets = new List<Tileset>();
    tilesets.Add(tileset);
    mapLayers = new List<MapLayer>();
    mapLayers.Add(layer);
}
#endregion
#region Method Region
public void Draw(SpriteBatch spriteBatch)
{
    Rectangle destination = new Rectangle(0, 0, Engine.TileWidth, Engine.TileHeight);
    Tile tile;
    foreach (MapLayer layer in mapLayers)
    {
        for (int y = 0; y < layer.Height; y++)
        {
            destination.Y = y * Engine.TileHeight;
            for (int x = 0; x < layer.Width; x++)
            {
                tile = layer.GetTile(x, y);
                destination.X = x * Engine.TileWidth;
                spriteBatch.Draw(
                    tilesets[tile.Tileset].Texture,
                    destination,
                    tilesets[tile.Tileset].SourceRectangles[tile.TileIndex],
                    Color.White);
            }
        }
    }
#endregion
}

This is class is the heart of the tile engine. This is where the tiling is done. There are two fields in this class. The first, tilesets, is a List<Tileset>. This is a list of tile sets that can grow and shrink as needed. It is better than using an array because you need to know the size of the array at the start. Using List<T> is a good idea when you don't know the size you will need. The second field, mapLayers, is a List<MapLayer> and will hold the layers of the map.

I added in two constructor for this class. One that takes a List<Tileset> and List<MapLayer> as its parameters. The other takes a Tileset and MapLayer as its parameters. The first constructor just sets the fields to the parameters passed in. The second creates a new List<Tileset> and adds the Tileset that is passed in to the list. It then creates a new List<MapLayer> and adds the MapLayer passed in to the list.

In the Draw method I draw the layers. The Draw method takes a SpriteBatch parameter that is the active SpriteBatch in between calls to Begin and End. To make it more efficient I have a Rectangle and Tile object that I will reuse rather than creating and destroying them each time through the loop. For the Rectangle only the X and Y values will change, the height and width remain constant. As well I'm get a new tile object each time so it is okay to reuse that as well, as long as it isn't changed.
In a for each loop I loop through all of the layers in the map. Inside of that there are the nested for loops that you should be familiar with. Inside the outer loop I calculate the Y coordinate. There is no point in recalculating it each time time through the inner loop as it only changes when you move to the next row. In the inner loop I get the tile using the GetTile method of the MapLayer class. I then calculate the X coordinate of the tile.

The complicated part here is deciphering the call do the Draw method. The first parameter is of the Draw method is the Texture2D of the image. I find that using the Tileset property of the tile and using that as the index for the List<Tileset> and use the Texture property. The destination rectangle has already been calculated. The source rectangle is found using the Tileset property of the tile to make sure we are in the right tile set and then using the TileIndex property to get the proper source rectangle.

The next thing you will want to do is to create a map and test these classes. I did that in the GamePlayScreen. What I did is replace the array of integers field with a TileMap field. Change the Field region of the GamePlayScreen to the following.

```
#region Field Region
Engine engine = new Engine(32, 32);
Tileset tileset;
TileMap map;
#endregion
```

The next thing that needs to be done is to create the map. I did that in the LoadContent method. Change the LoadContent to the following.

```
protected override void LoadContent()
{
    Texture2D tilesetTexture = Game.Content.Load<Texture2D>("Tilesets\tileset1");
tileset = new Tileset(tilesetTexture, 8, 8, 32, 32);
MapLayer layer = new MapLayer(40, 40);
for (int y = 0; y < layer.Height; y++)
{
    for (int x = 0; x < layer.Width; x++)
    {
        Tile tile = new Tile(0, 0);
        layer.SetTile(x, y, tile);
    }
}
map = new TileMap(tileset, layer);
base.LoadContent();
}
```

What I did here is first create a MapLayer called Layer using the constructor that takes the width and height of the layer. There is again the nest for loops. In the inner loop I create a new tile setting the tile index to 1 and the tile set index to 0. I then use the SetTile method passing in the x and y coordinates and the Tile object I created. I then create a new map passing in the tileset and layer.

The last thing to do is to draw the map. The Draw method is where I did that. All you need to do is call the Draw method of the TileMap class. Change the Draw method of the GamePlayScreen class to the
Now we have the beginnings of a nice little tile engine to work with. There is still more work to do but it is a good beginning. So, I think this is enough for this tutorial. I'd like to try and keep them to a reasonable length so that you don't have too much to digest at once. I encourage you to visit the news page of my site, XNA Game Programming Adventures, for the latest news on my tutorials.

Good luck in your game programming adventures!

Jamie McMahon