Creating a Role Playing Game with XNA Game Studio 3.0
Part 30
Tile Map Editor – Part 4

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/New2DRPG29.zip You can download the graphics from this link: Graphics.zip

This is the fourth tutorial about creating a tile map editor for the game. There will be another part in the near future on being able to write out maps with multiple layers. The editor does work great and does write out maps but I haven't incorporated being able to create multiple layers yet. The next tutorial I will be working on is reading in a map but with out the Content Pipeline.

There is one thing that I want to add to the editor before going much farther. I would like to be able to display what tile the cursor is in. I want to add a **Label** and **Textbox** to the form. Drag a **Label** onto the form and set the **Autosize** property to **false** first then set to following properties: **(Name)** is **lblCursorLocation**, **Location** is **880, 552**, **Size** is **128, 13**, **Text** is **Cursor Location** and **TextAlign** is **MiddleCenter**. Drag a **Textbox** control on the form and set the following properties: **(Name)** is **tbCursorLocation**, **BackColor** is **White**, **Location** is **820, 568**, **ReadOnly** is T**rue**, **Size** is **125, 20** and **TextAlign** is **Center**.

I ended up having to redesign the **Camera** class. The reason was I am using it to scroll the map in the **TileDisplayControl**. I could have done this all in the code for the form but I thought separating it would have been better. This is the new **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();
        }
```

```
        public void LockToVector(Vector2 position)
        {
            Position.X = position.X + Engine.TileWidth
                         - (Engine.ViewPortWidth / 2);
            Position.Y = position.Y + Engine.TileHeight
                         - (Engine.ViewPortHeight / 2);
        }

        public void LockCamera()
        {
            Position.X = MathHelper.Clamp(
                Position.X,
                0,
                TileMap.WidthInPixels - Engine.ViewPortWidth);
            Position.Y = MathHelper.Clamp(
                Position.Y,
                0,
                TileMap.HeightInPixels - Engine.ViewPortHeight);
        }
    }
}
```

The difference from the other class is that there are two methods **LockToVector** and **LockCamera**. The first method, **LockToVector**, is like the **LockToSprite** method. It sets the **X** value of the camera's postion to the **X** value of the vector passed in, plus the width of a tile on the screen minus half the width of the display. For the **Y** value it takes the **Y** value of the vector passed in, adds the height of a tile on the screen and subtracts half the hieght of the display. The LockCamera method just makes sure that the position of the camera can never be negative and it will never cause the map to scroll off the display, just like in the **Camera** class in the game.

I also made a few small changes to the **TileMap** class to make getting the width and height of the map in pixels for the camera easier. I made the **mapWidth** and **mapHeight** fields static as well as the **MapWidthInPixels** and **MapHeightInPixels** properties. This is the updated **TileMap** 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>();

        static int mapWidth;
        static int mapHeight;

        public TileMap(int mapWidth, int mapHeight)
        {
            TileMap.mapWidth = mapWidth;
            TileMap.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 static int WidthInPixels
        {
            get { return mapWidth * Engine.TileWidth; }
        }

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

Right click **Form1.cs** and choose **View code** to bring up the code for the form. Add these two using statements to the code of **Form1.cs**.

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

I will be using XNA for the handling the input for the editor so I will the using statement for the **Input** namespace for XNA. The second one is because there is already an enum named **Keys** in the **System.Windows.Forms** namespace. So when I want to use the **Keys** enum from XNA I can use **XKeys** for it.

I will need four more fields in the class. I will be handling moving the cursor with the keyboard for now. Later I will add in mouse support where you can scroll the map by moving the map to the edges of map like in many popular strategy games. I also need a way to know which layer of the map we are working on. I have field to hold that. Add these fields to **Form1.cs**. I will be using a **Vector2** to keep track of the position of the cursor.

```
KeyboardState keyState;
KeyboardState oldKeyState;
```

```
Vector2 position = new Vector2();
TileMapLayer currentLayer = null;
```

There were a few changes to the constructor of **Form1**. When **Form1** I disabled being able to save a map when the form is first created because there is nothing to save. I also added in the event handler for when the user clicks the **Save Map** option in the menu as well as the **Exit** option. I also set the **InterceptArrowKeys** property of **nudCurrentTile** to false so that when you press an arrow key the control will not move the selected tile up or down. This is the code for the updated constructor of **Form1**.

```
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);

    exitToolStripMenuItem.Click +=
        new EventHandler(exitToolStripMenuItem_Click);

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

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

I made a small change to the **Render** method. At the end of the method I call a method I created called **DrawDisplay**. The **DrawDisplay** method draws a grid and it draws the cursor. This is the code for **Render** method and the **DrawDisplay** method. I will explain how the **DrawDisplay** method works after I have shown you the code.

```
private void Render()
{
    GraphicsDevice.Clear(Color.Black);
    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();
        }
        DrawDisplay();
    }
}

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);
    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();
}
```

The **DrawDisplay** method first draws a grid of white rectangles to show the tiles. To find out how many tiles to draw across the screen I get the width of the display and then divide it by the width of the tiles on the display. To find out how many tiles fit on the display vertically I take the height of the display area and divide that by the height of the tiles on the display. To draw the rectangle I find the **X** value by taking **x** and multiplying it by the width of the tile on the screen. I do something similar for

the **Y** value. I take **y** and multiply it by the height of the tile on the screen. For the **Width** and **Height** I use the width and height of the tiles on the screen.

Drawing the cursor is a lot like drawing the sprite for the player character. It is drawn relative to its position and the position of the camera. The way that I find the **X** and **Y** coordinates of the cursor may look a little strange. The **position** of the cursor is measured in pixels, not tiles. To make sure that the cursor is always aligned to the grid I take the **X** value of **position** and divide it by the width of the tiles on the display and then multiply it by the width of the tiles on the display. That just makes sure that if when **position** is updated that it will always be aligned to the grid. I then subtract the **X** value of the camera's position. The **Y** value is calculated in a similar way. I take the **Y** value of the position, divide it by the height of the tiles on the display, multiply it by the height of the tiles on the display and subtract the **Y** value of the camera's position. For the **Width** of the rectangle I use the width of the tiles on the display and for the **Height** of the rectangle I use the height of the tiles on the display.

Before I get to the **Logic** method I made a change to the **newMapToolStripMenuItem_Click** method. What I did was assign **currentLayer** to be the first layer in the map and set a few properties of **clbLayers**. I set the **SelectedIndex** to 0, the first index, and **SelectionMode** to **SelectionMode.One** which means only one item will be able to be selected. This is the code for the method.

```
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;
        saveMapToolStripMenuItem.Enabled = true;

        clbLayers.Items.Add(frmNewMap.TileLayerName, true);
        clbLayers.SelectedIndex = 0;
        clbLayers.SelectionMode = SelectionMode.One;
        currentLayer = tileMap.layers[0];

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

Now I will update the **Logic** method. The **Logic** method is where I will handle the input from the user. For the **Logic** method I will need two helper methods. The first one is **LockCursor** that will lock the cursor to the display and lock the camera to the cursor. The second one you will be familiar

with **CheckKey**. This method will be used to determine if a key has been pressed and released. This is all new code so I will explain it all.

```
private void Logic()
{
    tileDisplay1.Focus();
    if (tileMap != null)
    {
        keyState = Keyboard.GetState();
        if (CheckKey(XKeys.Right))
        {
            position.X += Engine.TileWidth;
        }
        else if (CheckKey(XKeys.Down))
        {
            position.Y += Engine.TileHeight;
        }
        else if (CheckKey(XKeys.Left))
        {
            position.X -= Engine.TileWidth;
        }
        else if (CheckKey(XKeys.Up))
        {
            position.Y -= Engine.TileHeight;
        }
        LockCursor();
        int tileX = (int)position.X / Engine.TileWidth;
        int tileY = (int)position.Y / Engine.TileHeight;

        tbCursorPosition.Text = "( " + tileX.ToString() + ", ";
        tbCursorPosition.Text += tileY.ToString() + " )";
        tbCursorPosition.Invalidate();

        if (CheckKey(XKeys.Space))
        {
            if (rbDraw.Checked == true)
                currentLayer.SetTile(
                    tileX,
                    tileY,
                    (int)nudCurrentTile.Value);
            if (rbErase.Checked == true)
                currentLayer.SetTile(
                    tileX,
                    tileY,
                    -1);
        }
        oldKeyState = keyState;
    }
}

private void LockCursor()
{
    if (position.X < 0)
        position.X = 0;
```

```
    if (position.Y < 0)
        position.Y = 0;
    if (position.X >= TileMap.WidthInPixels)
        position.X = TileMap.WidthInPixels - Engine.TileWidth;
    if (position.Y >= TileMap.HeightInPixels)
        position.Y = TileMap.HeightInPixels - Engine.TileHeight;
    camera.LockToVector(position);
    camera.LockCamera();
}


private bool CheckKey(XKeys theKey)
{
    return (oldKeyState.IsKeyDown(theKey) && keyState.IsKeyUp(theKey));
}
```

The first line in the **Logic** method gives focus to **tileDisplay1**. The reason I did this is because if the control does not have focus everytime you press a key Windows will make a sound. It's annoying so I wanted to keep it from happening. There is an if statement to make sure that there is a map to work on. Inside of that if statement is where I handle input form the user.

The first thing I do inside of the if statement is get the current state of the keyboard. Next there is a series of if-else-if statements. Unlike with the tile engine in the game I only want the editor to scroll one tile at a time in caridnal directions. In the if statements I call the **CheckKey** method passing in the key that I want to check. After checking which key is being pressed I call the **LockCursor** method. The **LockCursor** method is responsible for keeping the cursor on the map. I will explain how it works in a moment.

I then find out which tile the cursor is in. To find the **X** position I took the **X** value of the **position** field and divided it by the width of the tiles. For the **Y** I took the **Y** value of the **position** field and divided it by the height of the tiles. Next I set the **Text** property of **tbCursorPosition**. I then check to see if the **Space** key has been pressed. In that if statement there are two other if statements. The first one checks to see if the radio button for drawing a tile is checked. If it has been checked I use the **SetTile** method of **currentLayer** passing in **tileX**, **tileY** and **nudCurrentTile.Value**. The last value has to be cast to an integer because it is a decimal value. If the radio button for erasing a tile is selected I pass in **tileX**, **tileY** and **-1**. If you remember **-1** means that there is no tile present and it is skipped.

The **LockCursor** method is responsible for keeping the cursor on the map and in the display. The firs if statement makes sure that the cursor does not move off the left edge of the map. The next one makes sure the cursor does not move off the top of the map. The following two check for the right and bottom. The next two lines call the **LockToVector** and the **LockCamera** methods of **camera**.

To handle saving the map I created an event handler **saveMapToolStripMenuItem_Click** and a helper method called **WriteMap** that will actually write the map to disk. This is the code for the **saveMapToolStripMenuItem_Click** method.

```
void saveMapToolStripMenuItem_Click(object sender, EventArgs e)
{
    SaveFileDialog sfdSaveMap = new SaveFileDialog();
    sfdSaveMap.AddExtension = true;
    sfdSaveMap.CheckPathExists = true;
    sfdSaveMap.DefaultExt = "tmap";
    sfdSaveMap.Filter = "(Tile Maps *.tmap)|*.tmap";
    sfdSaveMap.ValidateNames = true;

    DialogResult result = sfdSaveMap.ShowDialog();
    if (result == DialogResult.OK)
    {
        try
        {
            WriteMap(sfdSaveMap.FileName);
            MessageBox.Show(
                "Map saved successfully",
                "Success",
                MessageBoxButtons.OK);
        }
        catch
        {
            MessageBox.Show(
                "Error trying to save the map!",
                "Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }
    }
}
```

The **saveMapToolStripMenuItem_Click** method first creates a **SaveFileDialog** object that will be used to get the file name that user wants to save the file as. I set the **AddExtension** property to true so if the user does not give the extension it will be added. The next property **CheckPathExists** makes sure if the user types in a path that the path exists. I decided to use **tmap** as the extension for the map files so I set the **DefaultExtension** property to **tmap**. I set the **Filter** property to **(Tile Maps *.tmap)|*.tmap** which means that the **SaveFileDialog** will only show files with the extension **tmap**. I also set **ValidateNames** to true which will make sure that the file name entered is in the proper format.

I then call the **ShowDialog** method to display the dialog and capture the result in the **result** variable. If the result is **DialogResult.OK**, meaning that everything worked, inside a try catch block I try to save the map. If saving the map works I show a message box saying that the map was saved successfully. If it fails I show a message box saying that saving the map failed.

In the **WriteMap** method I actually write the map out as an XML document. I chose to use an XML document because they are easy to create and read in. The format of the XML document is as follows.

```
<TileMap Width="10" Height="10">
  <Layers>
```

```
    <Layer>
      <Row>0 0 0 0 0 0 0 0 0 0 </Row>
      <Row>2 2 2 2 2 2 2 2 2 2 </Row>
      <Row>4 4 4 4 4 4 4 4 4 4 </Row>
      <Row>4 4 4 4 4 4 4 4 4 4 </Row>
      <Row>5 5 5 5 5 5 5 5 5 5 </Row>
      <Row>5 5 5 5 5 5 5 5 5 5 </Row>
      <Row>6 6 6 6 6 6 6 6 6 6 </Row>
      <Row>6 6 6 6 6 6 6 6 6 6 </Row>
      <Row>15 15 15 15 15 15 15 15 15 15 </Row>
      <Row>1 1 1 1 1 1 1 1 1 1 </Row>
    </Layer>
  </Layers>
</TileMap>
```

The root node is **TileMap** which has as attributes **Width** and **Height** which are the width and height of the map in tiles. There is a node **Layers** which will hold all of the layers in the map. The layers are written inside **Layer**. To make is easier to read and write maps I use nodes named **Row** for each of the rows in the map and set the inner text of the element to be the tiles in the row separated by spaces. This is the code for the **WriteMap** method.

```
private void WriteMap(string fileName)
{
    XmlDocument xmlDoc = new XmlDocument();

    XmlElement root = xmlDoc.CreateElement("TileMap");
    xmlDoc.AppendChild(root);

    XmlAttribute attrib = xmlDoc.CreateAttribute("Width");
    attrib.Value = tileMap.MapWidth.ToString();
    root.Attributes.Append(attrib);

    attrib = xmlDoc.CreateAttribute("Height");
    attrib.Value = tileMap.MapHeight.ToString();
    root.Attributes.Append(attrib);
    XmlElement layerElement = xmlDoc.CreateElement("Layers");
    root.AppendChild(layerElement);
    foreach (TileMapLayer layer in tileMap.layers)
    {
        XmlElement element = xmlDoc.CreateElement("Layer");
        for (int y = 0; y < tileMap.MapHeight; y++)
        {
            XmlElement row = xmlDoc.CreateElement("Row");
            string innerText = "";
            for (int x = 0; x < tileMap.MapWidth; x++)
                innerText += layer.GetTile(x, y).ToString() + " ";
            row.InnerText = innerText;
            element.AppendChild(row);
        }
        layerElement.AppendChild(element);
    }
    xmlDoc.Save(fileName);
}
```

The first thing that I do in the **WriteMap** method is create a new **XmlDocument** called **xmlDoc**. I then create an **XmlElement** called **root** and with the value **TileMap**. This is the root node of the **XmlDocument** and I append it as a child to **xmlDoc**. I then created an **XmlAttribute** with the name **Width** and set its value to **tileMap.MapWidth**. I had to use the **ToString** method to convert the integer to a string because XML only writes strings. I then append the attribute to **root**. I then create another attribute with the name **Height** and set its value to **tileMap.MapHeight** and append the attribute to **root**. I then create an **XmlElement** called **layerElement** that has as a name **Layers**. This element will hold all of the layers of the map. I then append **layerElement** to **root** as a child.

Next there is a foreach loop that will loop through all of the layers in the map. At the moment there is only one layer as I have not coded adding multiple layers to the editor. It will however when I do add in multiple layers to the editor will write out all of the layers. Unlike the **TilesetGenerator** order in this document is important. Things need to written in the right order. The layers will have to be written in the order you want them to appear on the screen. The first layer will be the base layer and each layer added on top of it will be drawn in the proper order.

Inside the foreach loop I create an **XmlElement** called **element** with the name **Layer**. Then there is a for loop that will loop through the rows in the map ranging from 0 to **tileMap.MapHeight**. Each pass through that loop I create another **XmlElement** called **row** with the name **Row**. I then create a string called **innerText** that will hold the tile number for each of the tiles in the row. I then loop through the columns of the map ranging from 0 to **tileMap.MapWidth**. I then use the **GetTile** method of the **TileMapLayer** class to get the tile at the **x** and **y** coordinates, convert it to a string and add it plus a space to **innerText**. After the inner for loop I set **row.InnerText** to **innerText**. I then append **row** to **element** as a child. After the outer for loop I append **element** to **layerElement** as a child. Once I am out of the foreach loop I save the XML document using the **Save** method of the **XmlDocument** class using the **fileName** parameter passed to the method.

There is one last method. The **exitToolStripMenuItem_Click** method that will handle when the user clicks the **Exit** option in the menu. At the moment the code is really simple. All it does is call the **Close** method of the form to tell it to close. I will eventually add in that if there were changes made to the map the user will be asked if they want to save the map. This is the **exitToolStripMenuItem_Click** method.

```
void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.Close();
}
```

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 my tutorials.