XNA 4.0 RPG Tutorials

Part 23B

Level Editor

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 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.

This is part B in my tutorial on creating a level editor to use with your game. In this part I will start coding the actual level editor now that all of the basic pieces we need are in place. In this tutorial we will be working mostly in the code for **FormMain**, the actual level editor. Right click **FormMain** in the **XLevelEditor** project and select **View Code**. A lot of code is going to go into the code for the form so I'm going to do it in stages. To start with I'm going to set up the using statements, a few fields and properties, wire some event handlers, and set some properties of controls. Change the code for **FormMain** 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.IO;

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

using GDIBitmap = System.Drawing.Bitmap;
using GDIColor = System.Drawing.Color;
using GDIImage = System.Drawing.Image;
using GDIGraphics = System.Drawing.Graphics;
using GDIGraphicsUnit = System.Drawing.GraphicsUnit;
using GDIRectangle = System.Drawing.Rectangle;

using RpgLibrary.WorldClasses;

using XRpgLibrary.TileEngine;

namespace XLevelEditor
{
    public partial class FormMain : Form
    {
        #region Field Region

        SpriteBatch spriteBatch;
        LevelData levelData;

        TileMap map;
        List<Tileset> tileSets = new List<Tileset>();
        List<MapLayer> layers = new List<MapLayer>();

        #endregion
```

```
        #region Property Region

        public GraphicsDevice GraphicsDevice
        {
            get { return mapDisplay.GraphicsDevice; }
        }

        #endregion

        #region Constructor Region

        public FormMain()
        {
            InitializeComponent();

            this.Load += new EventHandler(FormMain_Load);
            this.FormClosing += new FormClosingEventHandler(FormMain_FormClosing);

            tilesetToolStripMenuItem.Enabled = false;
            mapLayerToolStripMenuItem.Enabled = false;
            charactersToolStripMenuItem.Enabled = false;
            chestsToolStripMenuItem.Enabled = false;
            keysToolStripMenuItem.Enabled = false;

            newLevelToolStripMenuItem.Click += new EventHandler(newLevelToolStripMenuItem_Click);
            newTilesetToolStripMenuItem.Click += new
EventHandler(newTilesetToolStripMenuItem_Click);
            newLayerToolStripMenuItem.Click += new EventHandler(newLayerToolStripMenuItem_Click);

            mapDisplay.OnInitialize += new EventHandler(mapDisplay_OnInitialize);
            mapDisplay.OnDraw += new EventHandler(mapDisplay_OnDraw);
        }

        #endregion

        #region Form Event Handler Region

        void FormMain_Load(object sender, EventArgs e)
        {
            Application.Idle += new EventHandler(Application_Idle);
        }

        void FormMain_FormClosing(object sender, FormClosingEventArgs e)
        {
        }

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

        #endregion

        #region New Menu Item Event Handler Region

        void newLevelToolStripMenuItem_Click(object sender, EventArgs e)
        {
            using (FormNewLevel frmNewLevel = new FormNewLevel())
            {
                frmNewLevel.ShowDialog();

                if (frmNewLevel.OKPressed)
                {
                    levelData = frmNewLevel.LevelData;
                    tilesetToolStripMenuItem.Enabled = true;
                }
            }
        }
```

```
        void newTilesetToolStripMenuItem_Click(object sender, EventArgs e)
        {
            using (FormNewTileset frmNewTileset = new FormNewTileset())
            {
                frmNewTileset.ShowDialog();

                if (frmNewTileset.OKPressed)
                {
                    TilesetData data = frmNewTileset.TilesetData;

                    mapLayerToolStripMenuItem.Enabled = true;
                }
            }
        }

        void newLayerToolStripMenuItem_Click(object sender, EventArgs e)
        {
            using (FormNewLayer frmNewLayer = new FormNewLayer(levelData.MapWidth,
levelData.MapHeight))
            {
                frmNewLayer.ShowDialog();

                if (frmNewLayer.OKPressed)
                {
                    MapLayerData data = frmNewLayer.MapLayerData;

                    charactersToolStripMenuItem.Enabled = true;
                    chestsToolStripMenuItem.Enabled = true;
                    keysToolStripMenuItem.Enabled = true;
                }
            }
        }

        #endregion

        #region Map Display Event Handler Region

        void mapDisplay_OnInitialize(object sender, EventArgs e)
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);

            mapDisplay.MouseEnter += new EventHandler(mapDisplay_MouseEnter);
            mapDisplay.MouseLeave += new EventHandler(mapDisplay_MouseLeave);
            mapDisplay.MouseMove += new MouseEventHandler(mapDisplay_MouseMove);
            mapDisplay.MouseDown += new MouseEventHandler(mapDisplay_MouseDown);
            mapDisplay.MouseUp += new MouseEventHandler(mapDisplay_MouseUp);
        }

        void mapDisplay_OnDraw(object sender, EventArgs e)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);
            Render();
            Logic();
        }

        void mapDisplay_MouseUp(object sender, MouseEventArgs e)
        {
        }

        void mapDisplay_MouseDown(object sender, MouseEventArgs e)
        {
        }

        void mapDisplay_MouseMove(object sender, MouseEventArgs e)
        {
        }

        void mapDisplay_MouseLeave(object sender, EventArgs e)
        {
        }
```

```
        void mapDisplay_MouseEnter(object sender, EventArgs e)
        {
        }

        #endregion

        #region Display Rendering and Logic Region

        private void Render()
        {
        }

        private void Logic()
        {
        }

        #endregion
    }
}
```

There was going to be confusion between some of the classes in the **System.Drawing** name space and the XNA Framework so I removed the using statement for **System.Drawing**. I added in a using statement for **System.IO** as we will be doing some file input and output. I also added in using statements for the basic XNA framework and the XNA framework **Graphics** classes. There is then several qualified using statements. A qualified using statement allows you to swap a fully qualified class with the qualifier. So, when you need to use the **Bitmap** class from **System.Drawing** you can use **GDIBitmap**. You may also hear it called aliasing, giving a class an alternative name.

I added in a few fields that I'm sure we are going to want to work with. There is a **SpriteBatch** field because we are going to be drawing maps and you need a **SpriteBatch** to draw the layers. There is a **LevelData** field that will hold information about the level currently being edited. I added in a **TileMap** field, **map**, to hold the map. I also have two **List<T>** fields. The first is for the tilesets and the second is the layers of the map.

At the moment there is just the one property, **GraphicsDevice**. I included that because you will need access to the **GraphicsDevice** associated with the **MapDisplay**. You need it to create a **SpriteBatch**, to clear the scene, and to read in content.

The constructor first wires event handlers for the **Load** event of the form and the **FormClosing** event. In the **Load** event I will do a little initialization. In the **FormClosing** event I will be adding checks that if the level has changed the user gets a warning that they haven't saved the level. I then set the **Enabled** property of most of the main menu items to false. I do that because you don't want to add items until there is a level to work with. I then wire the handlers for the **Click** event of the new level, new tileset, and new maplayer menu items.

The event handler for the **Load** event for the form wires an event handler for the **Idle** event of the application. The **Idle** event will be fired when your form isn't doing anything, or idle. When it is idle is a good time to let the map display know that it can redraw itself. The **FormClosing** event handler does nothing at the moment but will be used in the future so I added it in now. Next is the event handler for the **Idle** event of the application. All it does is call the **Invalidate** method of the map display letting it know it is time to redraw itself.

The next region of code is for the new menu items. The first is for the new level menu item. Inside of a using statement I create a new instance of **FormNewLevel**. Inside that statement I call the **ShowDialog**

method. I then check the **OKPressed** property of the form. If it was true I set the **levelData** field to be the **LevelData** property of **FormNewLevel**. I also set the **Enabled** property of the tileset menu item to true so tilesets can be added.

The handler for the **Click** event for the new tileset menu item is similar. It creates an instance of the form inside of a using statement. Inside that it displays the form using the **ShowDialog** method. It checks to see if the **OKPressed** property of the form is true as well. It is I set a **TilesetData** local variable to the **TilesetData** property. I also set the **Enabled** property of the map layer menu item to true.

The event handler for the **Click** event for a new layer has a similar form as the others. In a using statement I create a new **FormNewLayer**. I pass in the **MapWidth** and **MapHeight** from the **levelData** field to the constructor. Inside the using statement I call the **ShowDialog** method to display the form. I check to see if the **OKPressed** property of the form is true. If so, I create a **MapLayerData** object. I also set the **Enabled** property of the other main menu items to true.

There is a region of code that is dedicated to event handlers related to the **MapDisplay** control. The **OnInitialize** event will be triggered when the **MapDisplay** is first initialized. In the handler for that event I create a new **SpriteBatch** object to be used in rendering the map. I then wire handlers for many of the events related to the mouse.

The **OnDraw** event handler for the **MapDisplay** will be called when it is time to redraw the map. It will be triggered automatically everything the **Application.Idle** event handler is called by calling the **Invalidate** method of the **MapDisplay**. The method calls the **Clear** method of the **GraphicsDevice** for the **MapDisplay** control and sets the background to the familiar **CornflowerBlue** you see in XNA. It then calls a method **Render**. The **Render** method will be responsible for rendering the map. I also added a method called **Logic**. The **Logic** method will be responsible for the logic of the editor. Both methods live in a region dedicated to rendering and logic. At the moment they are method stubs that will be filled out as things progress.

That is the basic framework for the level editor. The next step will be reading in the image for a tileset so it can be used. To handle that I'm going to extend the event handler for the new tileset menu item. First, you are going to want a list of images you can use for the form. Add the following field and Change the **newTilesetToolStripMenuItem_Click** method to the following. I also added a new method to the **TileMap** class called **AddTileset** that will add a new tileset to the map.

```
List<GDIImage> tileSetImages = new List<GDIImage>();

void newTilesetToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (FormNewTileset frmNewTileset = new FormNewTileset())
    {
        frmNewTileset.ShowDialog();

        if (frmNewTileset.OKPressed)
        {
            TilesetData data = frmNewTileset.TilesetData;

            try
            {
                GDIImage image = (GDIImage)GDIBitmap.FromFile(data.TilesetImageName);
                tileSetImages.Add(image);

                Stream stream = new FileStream(data.TilesetImageName, FileMode.Open,
```

```
FileAccess.Read);

                Texture2D texture = Texture2D.FromStream(GraphicsDevice, stream);

                Tileset tileset = new Tileset(
                        texture,
                        data.TilesWide,
                        data.TilesHigh,
                        data.TileWidthInPixels,
                        data.TileHeightInPixels);

                tileSets.Add(tileset);

                if (map != null)
                    map.AddTileset(tileset);

                stream.Close();
                stream.Dispose();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error reading file.\n" + ex.Message, "Error reading image");
                return;
            }

            lbTileset.Items.Add(data.TilesetName);

            if (lbTileset.SelectedItem == null)
                lbTileset.SelectedIndex = 0;

            mapLayerToolStripMenuItem.Enabled = true;
        }
    }
}

public void AddTileset(Tileset tileset)
{
    tilesets.Add(tileset);
}
```

So, what is the new code doing. First I used a try-catch block when I try to read in the image for the tileset. Here is a spot where an exception could be thrown if there is a problem reading the the image so it is best to do this in a try-catch block. That way you can recover from an exception. I first try to read in the image in a format that can be assigned to the **Picture Box** control that will preview the tileset, an **Image** from GDI+. To do that I cast the return value of the **FromFile** method of the **Bitmap** class from GDI+ as well. I then add the image to the list of tileset images. A change in XNA 4.0 from previous versions of XNA is that there is no longer a **FromFile** method for the **Texture2D** class. You now have to use the **FromStream** method instead. A stream is data that is sent from a source, be it a file or over the internet, or other sources. I create a **FileStream** using the **TilesetImageName** property from the **TilesetData** object for the name of the file and use **FileMode.Open** and **FileAccess.Read** to read the file. If you don't specify you want to read the file you may get an access violation stating the file is being used by another process. I then use the **FromStream** method of the **Texture2D** class to read in the image. I then create a new **Tileset** object and add it to the list of tilesets using the **Texture2D** I just read in and the fields from the **TilesetData** object. If that **map** field is not null I call the new **AddTileset** method to add it to the map. I then close the stream and dispose it. I the catch for the try I display a message box stating there was an error and the error then exit the method. I then add the **TilesetName** property of the **TilesetData** object to **lbTileset**, the **List Box** that holds the names of the tile sets. If the **SelectedItem** of **lbTileset** is null I set the **SelectedIndex** property of **lbTileset** to zero so that the tileset will be selected. I also set the **Enabled** property of map layer menu item to true.

When the user clicks a tileset name in **lbTileset** you are going to want to perform a few actions. The best way to do that is to handle the **SelectedIndexChanged** event of the **List Box**. I wired the event in the **Load** event of the form and placed the handler in the same region. Change the **FormMain_Load** method to the following and add this new region below the **Form Event Handler Region**.

```
void FormMain_Load(object sender, EventArgs e)
{
    Application.Idle += new EventHandler(Application_Idle);
    lbTileset.SelectedIndexChanged += new EventHandler(lbTileset_SelectedIndexChanged);
}

#region Tile Tab Event Handler Region

void lbTileset_SelectedIndexChanged(object sender, EventArgs e)
{
    if (lbTileset.SelectedItem != null)
    {
        nudCurrentTile.Value = 0;
        nudCurrentTile.Maximum = tileSets[lbTileset.SelectedIndex].SourceRectangles.Length - 1;
        FillPreviews();
    }
}

private void FillPreviews()
{
    int selected = lbTileset.SelectedIndex;
    int tile = (int)nudCurrentTile.Value;

    GDIImage preview = (GDIImage)new GDIBitmap(pbTilePreview.Width, pbTilePreview.Height);

    GDIRectangle dest = new GDIRectangle(0, 0, preview.Width, preview.Height);
    GDIRectangle source = new GDIRectangle(
        tileSets[selected].SourceRectangles[tile].X,
        tileSets[selected].SourceRectangles[tile].Y,
        tileSets[selected].SourceRectangles[tile].Width,
        tileSets[selected].SourceRectangles[tile].Height);

    GDIGraphics g = GDIGraphics.FromImage(preview);
    g.DrawImage(tileSetImages[selected], dest, source, GDIGraphicsUnit.Pixel);

    pbTilesetPreview.Image = tileSetImages[selected];
    pbTilePreview.Image = preview;
}

#endregion
```

The new event handler for the **Load** event of the form wires a handler for the **SelectedIndexChanged** event of **lbTileset** that will be fired when a different tileset is selected from the list. The new region I added is called **Tile Tab Event Handler Region** and will house event handlers related to the tile tab in the left pane. The event handler makes sure that the **SelectedItem** is not null. If it is not I set the **Value** property of **nudCurrentTile** to 0, the first tile in the tile set. I also set the **Maximum** property to be the number of source rectangles in the tileset minus 1 because the source rectangles are zero based so the last rectangle is the length minus 1. I then call a method I wrote that will fill the **Picture Boxes** in the tab called **FillPreviews**.

The **FillPreviews** does a little GDI+ manipulation to get the selected tile in **nudCurrentTile** into the tile preview **Picture Box**. You first need to know what tileset is selected, the **SelectedIndex** property of **lbTileset**, and which tile is selected, the **Value** property of **nudCurrentTile**. It needs to be cast to an integer because it is a decimal. I first create a **GDIImage** that is the width and height of the preview **Picture Box** for the tile. I need a destination rectangle to draw the image to. I use zero for the **X** and **Y** coordinates and the **Width** and **Height** of the image for the width and height. I also need the source

rectangle of the tile in the tile set. I get that using the **SelectedIndex** and the **Value** properties that I captured and the **tileSets** collection. I then create a **Graphics** object using **FromImage** and the preview image that I created. I then using the **DrawImage** method passing in the image from **tileSetImages**, the destination rectangle, source rectangle, and **GDIGraphicsUnit.Pixel**. I then set the **Image** properties of the **Picture Boxes**.

You are also going to want to change the tile preview if the **Value** property of **nudCurrentTile** changes. I will wire that in the **Load** event handler of the form as well. Change that method to the following and add this handler to the **Tile Tab Event Handler Region**.

```
void FormMain_Load(object sender, EventArgs e)
{
    Application.Idle += new EventHandler(Application_Idle);
    lbTileset.SelectedIndexChanged += new EventHandler(lbTileset_SelectedIndexChanged);
    nudCurrentTile.ValueChanged += new EventHandler(nudCurrentTile_ValueChanged);
}

void nudCurrentTile_ValueChanged(object sender, EventArgs e)
{
    if (lbTileset.SelectedItem != null)
    {
        FillPreviews();
    }
}
```

The new handler checks to see if the **SelectedItem** property is not null. If it is null you don't want a preview. If it is not null I call the **FillPreviews** method to update the previews.

The next step it to handle creating a new layer. That takes place in the handler for the new layer menu item. Change that handler to the following.

```
void newLayerToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (FormNewLayer frmNewLayer = new FormNewLayer(levelData.MapWidth, levelData.MapHeight))
    {
        frmNewLayer.ShowDialog();

        if (frmNewLayer.OKPressed)
        {
            MapLayerData data = frmNewLayer.MapLayerData;

            if (clbLayers.Items.Contains(data.MapLayerName))
            {
                MessageBox.Show("Layer with name " + data.MapLayerName + " exists.", "Existing
layer");
                return;
            }

            MapLayer layer = MapLayer.FromMapLayerData(data);
            clbLayers.Items.Add(data.MapLayerName, true);
            clbLayers.SelectedIndex = clbLayers.Items.Count - 1;

            layers.Add(layer);

            if (map == null)
                map = new TileMap(tileSets, layers);

            charactersToolStripMenuItem.Enabled = true;
            chestsToolStripMenuItem.Enabled = true;
            keysToolStripMenuItem.Enabled = true;
        }
    }
}
```

What the new code is doing is checking to see if an entry in **clbLayers**, the **Checked List Box**, with the name **data.MapLayername** already exists. If it does I display an error message that a layer with that name already exists and exit the method. Otherwise I use the **FromMapLayerData** method I added to the **MapLayer** class to create a new layer and add it to the **List<MapLayer>**. I also add the **MapLayerName** to **clbLayers**. When adding the item I have it set to be checked initially, and thus drawn by default. I also set the **SelectedIndex** property of **clbLayers** to be the last layer that was added. If the **map** field is null I create a new **TileMap** using the **List<Tileset>** and **List<MapLayer>**. This is where you can see what the dangers of passing reference types around. The **map** field now has references to the **tileSets** and **layers** fields of the form. Changing one of them changes the other. I had originally included an else to the if statement that checked to see if **map** was null where I'd the new layer to **map** and suddenly there were 3 layers in the **layers** field, not 2. So, be careful when you are passing reference types around that this sort of thing does not happen.

The next step will be to actually draw the layers. For that we need two more things, a **Camera** and an **Engine**. Add the following fields to the **Fields** region of **FormMain**. Also, change the **Load** event handler of the form to the following to actually create the camera and the engine.

```
Camera camera;
Engine engine;

void FormMain_Load(object sender, EventArgs e)
{
    Application.Idle += new EventHandler(Application_Idle);

    lbTileset.SelectedIndexChanged += new EventHandler(lbTileset_SelectedIndexChanged);
    nudCurrentTile.ValueChanged += new EventHandler(nudCurrentTile_ValueChanged);

    Rectangle viewPort = new Rectangle(0, 0, mapDisplay.Width, mapDisplay.Height);
    camera = new Camera(viewPort);

    engine = new Engine(32, 32);
}
```

What I did was create a **Rectangle** that describes the **MapDisplay** control because the **Camera** class needs a rectangle that describes the view port it is using. I then create an **Engine** instance with pixel width and height of 32 pixels. Something I will be handling later, possibly not in this tutorial, is what to do if the size of the **MapDisplay** changes and how to recover from that. I will also add in options that you can set to change the width and height of the tiles on the screen.

So, the next step is to start drawing. That will take place in the **Render** method. Change the **Render** method to the following.

```
private void Render()
{
    for (int i = 0; i < layers.Count; i++)
    {
        spriteBatch.Begin(
            SpriteSortMode.Deferred,
            BlendState.AlphaBlend,
            SamplerState.PointClamp,
            null,
            null,
            null,
            camera.Transformation);

        if (clbLayers.GetItemChecked(i))
            layers[i].Draw(spriteBatch, camera, tileSets);
```

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

I loop through all of the layers in a for loop, not a foreach loop. I don't use a foreach loop because I will want to check if a layer should be drawn or not controlled by **clbLayers**. I then call the **Begin** method of **SpriteBatch** passing in parameters like I did in the game. There is an if statement after the call to **Begin** that gets if the item at the index in **clbLayers** is checked. If it is I call the **Draw** method of the layer in the list of layers passing in the **SpriteBatch** object, the **Camera** object and the **List<Tileset>** for the tilesets. I then call the **End** method of **SpriteBatch**.

The next thing I want to do is get the map scrolling and adding in tiles with the editor. My computer was scrolling my maps really fast so I ended up adding a **Timer** control to the form. Right click **FormMain** in the solution explorer and select **View Designer**. Now drag a new **Timer** control onto the form. Set the **(Name)** property of the timer to **controlTimer**. Now go back to the code for the form. I the **Load** event of the form I wired a handler for the **Tick** event of the timer and set a couple properties. I added the handler to the **Form Event Handler Region**. Change the **FormMain_Load** method to the following and add this handler to the region. I also removed the handler for **Application.Idle**. I wall call the **Invalidate** method from the handler for the timer's tick event.

```
void FormMain_Load(object sender, EventArgs e)
{
    lbTileset.SelectedIndexChanged += new EventHandler(lbTileset_SelectedIndexChanged);
    nudCurrentTile.ValueChanged += new EventHandler(nudCurrentTile_ValueChanged);

    Rectangle viewPort = new Rectangle(0, 0, mapDisplay.Width, mapDisplay.Height);
    camera = new Camera(viewPort);

    engine = new Engine(32, 32);

    controlTimer.Tick += new EventHandler(controlTimer_Tick);
    controlTimer.Enabled = true;
    controlTimer.Interval = 200;
}

void controlTimer_Tick(object sender, EventArgs e)
{
    mapDisplay.Invalidate();
    Logic();
}
```

So, all I did was wire the **Tick** event handler and set the **Enabled** property to true so that the event will be fired and I set the **Interval** value to 200 milliseconds, a fifth of a second. You may want to make the **Interval** smaller if your logic is sluggish or higher if it is too fast. Even where I set it my map scrolled rather quickly. Part of the reason being that I'm scrolling the map 1 tile at a time. The handler for the **Tick** event calls **Invalidate** on **mapDisplay** to redraw the display and it calls the **Logic** method.

For handling the logic of the editor I needed to add a few fields. A **Point** field for the position of the mouse over the **MapDisplay** and two bool fields that determines if the left mouse button is down and the other determines if we are interested in tracking the mouse. Add the following fields to **FormMain**.

```
Point mouse = new Point();

bool isMouseDown = false;
bool trackMouse = false;
```

Now I'm going to change the code for the mouse event handlers for the **MapDisplay**. Change them to the following.

```
void mapDisplay_MouseUp(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left)
        isMouseDown = false;
}

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

void mapDisplay_MouseMove(object sender, MouseEventArgs e)
{
    mouse.X = e.X;
    mouse.Y = e.Y;
}

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

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

The handler for **MouseUp** checks to see if the left mouse button triggered the event. If it did I set **isMouseDown** to false. The **MouseDown** handler works in reverse. The **MouseMove** handler sets the **X** and **Y** properties of **mouse** to the **X** and **Y** properties of the mouse. The handler for **MouseLeave** sets **trackMouse** to false because the mouse has moved outside of the **MapDisplay** and we aren't interested in processing it. The **MouseEnter** handler does the reverse, sets **trackMouse** to true because we are now interested in the mouse again.

Before I get to the logic of scrolling the map and drawing tiles I need to make two changes to the **Camera** class. I need to make the set part of the **Position** property public. I also need to make the **LockCamera** method public. Change the **Position** property and **LockCamera** method to the following.

```
public Vector2 Position
{
    get { return position; }
    set { position = value; }
}

public void LockCamera()
{
    position.X = MathHelper.Clamp(position.X,
        0,
        TileMap.WidthInPixels * zoom - viewportRectangle.Width);
    position.Y = MathHelper.Clamp(position.Y,
        0,
        TileMap.HeightInPixels * zoom - viewportRectangle.Height);
}
```

The next thing to do is to update the **Logic** method as that is where I will be handling scrolling the map and drawing the map. Change the **Logic** method to the following.

```
private void Logic()
{
```

```
    if (layers.Count == 0)
        return;

    Vector2 position = camera.Position;

    if (trackMouse)
    {
        if (mouse.X < Engine.TileWidth)
            position.X -= Engine.TileWidth;

        if (mouse.X > mapDisplay.Width - Engine.TileWidth)
            position.X += Engine.TileWidth;

        if (mouse.Y < Engine.TileHeight)
            position.Y -= Engine.TileHeight;

        if (mouse.Y > mapDisplay.Height - Engine.TileHeight)
            position.Y += Engine.TileHeight;

        camera.Position = position;
        camera.LockCamera();

        position.X = mouse.X + camera.Position.X;
        position.Y = mouse.Y + camera.Position.Y;

        Point tile = Engine.VectorToCell(position);

        if (isMouseDown)
        {
            if (rbDraw.Checked)
            {
                layers[clbLayers.SelectedIndex].SetTile(
                    tile.X,
                    tile.Y,
                    (int)nudCurrentTile.Value,
                    lbTileset.SelectedIndex);
            }
            if (rbErase.Checked)
            {
                layers[clbLayers.SelectedIndex].SetTile(
                    tile.X,
                    tile.Y,
                    -1,
                    -1);
            }
        }
    }
}
```

The first thing I do is check to see if the **Count** property of the **layers** field is zero. If it is you don't want to try and edit anything so I exit the method. I then save the **Position** property of the camera in a local variable **position**. I then check the **trackMouse** property. If it is true the mouse is in the map display. I check to see if the **X** value of the mouse's position is less than the width of a tile on the screen. If it is I subtract the width of a tile from the position of the camera, scrolling the map left. Then if the **X** value of the mouse's position is greater than or equal to the width of the map display minus the width of a tile I scroll the map one tile to the right. I do something similar for the **Y** component of the mouse's position. If it is less than the height of tile I scroll up and if it is greater than the height of the display minus a tile I scroll the map down. Since **Position** is a property in the **Camera** class and a structure you can't modify its **X** and **Y** value directly so I set the **Position** property of the camera to the **position** variable and call the **LockCamera** method to lock the camera. I then set the **position** variable to the position of the mouse plus the position of the camera. This tells us which tile the mouse is in on the map. I capture what tile the mouse is in using the **VectorToCell** method of the **Engine** class. I then check if the **isMouseDown** field is true. If it is I check if **rbDraw**'s **Checked** property is true. If it is

true you want to draw the tile. I call the **SetTile** method of the **MapLayer** class that takes the x and y coordinates of the tile and the tile index and tileset the tile belongs to. Similarly if **rbErase**'s **Checked** property is true you want to erase the tile. That is done by setting its tile index and tileset to -1.

So we have a working basic level editor. I'm going to stop this tutorial here though as I think you've had more than enough to digest. In a future tutorial I'm going to add writing out and reading in maps and adding some more options into the editor. The plan for this tutorial was to get started with the level editor and we are well under way.

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