Shadow Monsters – MonoGame Tutorial Series
Chapter 14
Editor – Part Two

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called Shadow Monsters. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my blog: Shadow Monsters. The source code for each tutorial will be available as well. I will be using Visual Studio 2019 Community for the series. The code should compile on the 2013, 2015 and 2017 versions as well.

I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just give credit to Cynthia McMahon and add a link to my site, https://mygameprogrammingadventures.blogspot,com. Screenshots of your project and/or a video of game play would be appreciated.

I also want to mention that I assume you have a basic understanding of C# and MonoGame. If you don't I recommend that you learn basic C# and work with MonoGame a little. Enough to know the basics of fields, properties, methods, classes and the MonoGame framework.

In this tutorial we will be continuing with the editor. We need controls inside MonoGame to be able to select tiles, layers and eventually other things. I added in controls that I previously created in other projects.

To get started right click the ShadowMonsters project, select Add and then New Folder. Name this new folder Controls. Right click the Controls folder select Add and then Class. Name this new class Control. Here is the code for that class.

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

namespace ShadowMonsters.Controls
{
    public abstract class Control
    {
        #region Field Region

        protected string _name;
        protected string _text;
        protected Vector2 _size;
        protected Vector2 _position;
        protected object _value;
        protected bool _hasFocus;
        protected bool _enabled;
        protected bool _visible;
```

```
protected bool _tabStop;
protected SpriteFont _spriteFont;
protected Color _color;
protected string _type;
protected bool _mouseOver;

#endregion

#region Event Region

public event EventHandler Selected;

#endregion

#region Property Region

public string Name
{
    get { return _name; }
    set { _name = value; }
}

public string Text
{
    get { return _text; }
    set { _text = value; }
}

public Vector2 Size
{
    get { return _size; }
    set { _size = value; }
}

public Vector2 Position
{
    get { return _position; }
    set
    {
        _position = value;
        _position.Y = (int)_position.Y;
    }
}

public object Value
{
    get { return _value; }
    set { this._value = value; }
}

public virtual bool HasFocus
{
    get { return _hasFocus; }
    set { _hasFocus = value; }
}

public bool Enabled
{
    get { return _enabled; }
    set { _enabled = value; }
```

```
}

public bool Visible
{
    get { return _visible; }
    set { _visible = value; }
}

public bool TabStop
{
    get { return _tabStop; }
    set { _tabStop = value; }
}

public SpriteFont SpriteFont
{
    get { return _spriteFont; }
    set { _spriteFont = value; }
}

public Color Color
{
    get { return _color; }
    set { _color = value; }
}

public string Type
{
    get { return _type; }
    set { _type = value; }
}

#endregion

#region Constructor Region

public Control()
{
    Color = Color.White;
    Enabled = true;
    Visible = true;
    SpriteFont = ControlManager.SpriteFont;
    _mouseOver = false;
}

#endregion

#region Abstract Methods

public abstract void Update(GameTime gameTime);
public abstract void Draw(SpriteBatch spriteBatch);
public abstract void HandleInput();

#endregion

#region Virtual Methods

protected virtual void OnSelected(EventArgs e)
{
    Selected?.Invoke(this, e);
```

```
        }

        #endregion
    }
}
```

This is an abstract class. That means that it must be inherited and implemented similar to an interface. There are a number of protected fields so the controls can access them directly. The _name field is the name of the control. The _text field is the text associated with the control. The _size field is the size of the control. The _position field is the position of the control. The _value field is a value that is associated with the control. The _hasFocus field says if the control is the currently selected control. The _enabled field says if the control is enabled. The _visible field says if the control is drawn or not. The _tabStop field determines if the control will be selected if the user hits the tab key. The _spriteFont field is what font will be used when drawing text. The _color field is what color text will be drawn in. The _type field is what type the control is. Finally _mouseOver tells if the mouse is over the control. There are read/write properties to expose the fields. There is also an event that will fire if the control is selected.

The constructor initializes some of the fields. Color is initialized to white. Enabled and Visible are set to true. SpriteFont is set to the sprite font of the ControlManager that I will implement next. Finally _mouseOver is set to false.

There are three abstract methods that the inherited class must implement. The Update method is used to update the control. The Draw method draws the control and HandleInput handles the input for the control. There is a protected virtual method that will invoke the event handler if it is subscribed to.

Now I'm going to add the control manager that will call the Update, Draw and HandleInput methods of the controls and handle what control currently has focus. Right click the Controls folder, select Add and then Class. Name this new class ControlManager. Here is the code for that class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ShadowMonsters.Controls
{
    public class ControlManager : List<Control>
    {
        #region Fields and Properties

        int _selectedControl = 0;
        bool _acceptInput = true;

        static SpriteFont _spriteFont;

        public static SpriteFont SpriteFont
```

```
{
    get { return _spriteFont; }
}

public bool AcceptInput
{
    get { return _acceptInput; }
    set { _acceptInput = value; }
}

#endregion

#region Event Region

public event EventHandler FocusChanged;

#endregion

#region Constructors

public ControlManager(SpriteFont spriteFont)
    : base()
{
    ControlManager._spriteFont = spriteFont;
}

public ControlManager(SpriteFont spriteFont, int capacity)
    : base(capacity)
{
    ControlManager._spriteFont = spriteFont;
}

public ControlManager(SpriteFont spriteFont, IEnumerable<Control> collection) :
    base(collection)
{
    ControlManager._spriteFont = spriteFont;
}

#endregion

#region Methods

public void Update(GameTime gameTime)
{
    if (Count == 0)
    {
        return;
    }

    foreach (Control c in this)
    {
        if (c.Enabled)
        {
            c.Update(gameTime);
        }
    }

    foreach (Control c in this)
    {
        if (c.HasFocus)
```

```
        {
            c.HandleInput();
            break;
        }
    }

    if (!AcceptInput)
    {
        return;
    }

    if (Xin.CheckKeyReleased(Keys.Up))
    {
        PreviousControl();
    }

    if (Xin.CheckKeyReleased(Keys.Tab))
    {
        NextControl();
    }
}

public void Draw(SpriteBatch spriteBatch)
{
    foreach (Control c in this)
    {
        if (c.Visible)
        {
            c.Draw(spriteBatch);
        }
    }
}

public void NextControl()
{
    if (Count == 0)
    {
        return;
    }

    int currentControl = _selectedControl;

    this[_selectedControl].HasFocus = false;

    do
    {
        _selectedControl++;

        if (_selectedControl == Count)
        {
            _selectedControl = 0;
        }

        if (this[_selectedControl].TabStop && this[_selectedControl].Enabled)
        {
            FocusChanged?.Invoke(this[_selectedControl], null);

            break;
        }
```

```
            } while (currentControl != _selectedControl);

            this[_selectedControl].HasFocus = true;
        }

        public void PreviousControl()
        {
            if (Count == 0)
            {
                return;
            }

            int currentControl = _selectedControl;

            this[_selectedControl].HasFocus = false;

            do
            {
                _selectedControl--;

                if (_selectedControl < 0)
                {
                    _selectedControl = Count - 1;
                }

                if (this[_selectedControl].TabStop && this[_selectedControl].Enabled)
                {
                    FocusChanged?.Invoke(this[_selectedControl], null);

                    break;
                }
            } while (currentControl != _selectedControl);

            this[_selectedControl].HasFocus = true;
        }

        #endregion
    }
}
```

First, I want to mention that this class requires Xin, my input handler. It also inherits from List<Control> so that it can be used as a collection. This makes it a little easier to manage the controls.

There are three fields in this class. The _selectedControl field determines what the currently selected control is. The _acceptInput field says if the control manager accepts input or not. The static field _spriteFont is the font that can be used. There is a read only property that returns the _spriteFont field. There is also a read/write property for the _acceptInput field.

There are three constructors. Each of them takes a SpriteFont. The first takes no additional parameters and calls the constructor of the base class and sets the _spriteFont field. The second constructor also takes an integer that is the initial capacity of the collection. The last takes an IEnumerable<Control> that is a collection of controls.

The Update method checks if there are no controls and if there are none it exits the method. It then loops over all of the controls. If they are Enabled it calls their Update method. It then

loops over the controls again and if the control has focus it calls its HandleInput method then breaks out of the loop. If the AcceptInput property is false it returns. If the Up key has been released the PreviousControl method is called. If the Tab key has been released the NextControl method is called.

The Draw method loops over all of the controls. If the control is Visible it calls the Draw method of the control.

The NextControl method cycles through the controls. If there are no controls in the manager it returns. It then assigns a local variable to the _selectedControl field. There is next a do loop that will loop until the currentControl variable is the _selectControl. It increments the _selectedControl field. If it is the Count property it is set to 0. If the control is a TabStop and it is Enabled the FocusChanged event is fired if it is subscribed to and the loop is exited. The control's HasFocus method is set to true.

The PreviousControl method works in reverse. It decrements instead of increments and checks if the Count property is 0. If it is 0 the _selectedControl field is set to the Count property minus 1.

I'm going to implement two controls. The first control is a picture box that displays an image. The second is a list box for selecting items. Right click the Controls folder in the Solution Explorer, select Add and then Class. Name this new class PictureBox. Here is the code.

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

namespace ShadowMonsters.Controls
{
    public class PictureBox : Control
    {
        #region Field Region

        private Texture2D _image;
        private Rectangle _sourceRect;
        private Rectangle _destRect;

        #endregion

        #region Property Region

        public Texture2D Image
        {
            get { return _image; }
            set { _image = value; }
        }

        public Rectangle SourceRectangle
        {
            get { return _sourceRect; }
            set { _sourceRect = value; }
```

```
}

public Rectangle DestinationRectangle
{
    get { return _destRect; }
    set { _destRect = value; }
}

#endregion

#region Constructors

public PictureBox(Texture2D image, Rectangle destination)
{
    Image = image;
    DestinationRectangle = destination;

    if (image != null)
        SourceRectangle = new Rectangle(0, 0, image.Width, image.Height);
    else
        SourceRectangle = new Rectangle(0, 0, 0, 0);
    Color = Color.White;
}

public PictureBox(Texture2D image, Rectangle destination, Rectangle source)
{
    Image = image;
    DestinationRectangle = destination;
    SourceRectangle = source;
    Color = Color.White;
}

#endregion

#region Abstract Method Region

public override void Update(GameTime gameTime)
{
}

public override void Draw(SpriteBatch spriteBatch)
{
    if (_image != null && _destRect != null && _sourceRect != null)
    {
        spriteBatch.Draw(_image, _destRect, _sourceRect, _color);
    }
}

public override void HandleInput()
{
}

#endregion

#region Picture Box Methods

public void SetPosition(Vector2 newPosition)
{
    _destRect = new Rectangle(
        (int)newPosition.X,
```

```
                (int)newPosition.Y,
                _sourceRect.Width,
                _sourceRect.Height);
        }

        #endregion
    }
}
```

The class inherits from the Control class and implements the required abstract methods. It has three fields. The _image field is the image to be drawn. The _sourceRectangle field is the area of the image that will be drawn. The _destinationRectangle field is where the image will be drawn. There are properties to expose the fields.

There are two constructors. The first takes a Texture2D and a Rectangle parameter. The Texture2D is the image to be drawn and Rectangle is the destination of the image. It then assigns the corresponding field to the parameter passed in. It then creates a source rectangle that describes the entire image using the Width and Height of the image. It then assigns the Color property to white. The second constructor takes the Texture2D that is the image, a destination rectangle and a source rectangle. It assigns the fields to the values passed in and the Color property to white.

The Update method does nothing so it is empty. The Draw method checks to make sure that the Image, SourceRectangle and DestinationRectangle are not null. If they aren't I call the SpriteBatch Draw overload that requires a Texture2D, destination Rectangle, source Rectangle and tint Color using the fields. The HandleInput method does nothing as well. The SetPosition method creates new destination rectangle using the Vector2 that is passed in.

The next control that I want to add is the list box. Right click the Controls folder in the Solution Explorer, select Add and then Class. Name this new class ListBox. Here is the code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ShadowMonsters.Controls
{
    public class ListBox : Control
    {
        #region Event Region

        public event EventHandler SelectionChanged;
        public event EventHandler Enter;
        public event EventHandler Leave;

        #endregion

        #region Field Region

        private readonly List<string> _items = new List<string>();
```

```
private int _startItem;
private readonly int _lineCount;

private readonly Texture2D _image;
private readonly Texture2D _cursor;

private Color _selectedColor = Color.Red;
private int _selectedItem;

#endregion

#region Property Region

public Color SelectedColor
{
    get { return _selectedColor; }
    set { _selectedColor = value; }
}

public int SelectedIndex
{
    get { return _selectedItem; }
    set { _selectedItem = (int)MathHelper.Clamp(value, 0f, _items.Count); }
}

public string SelectedItem
{
    get { return Items[_selectedItem]; }
}

public List<string> Items
{
    get { return _items; }
}

public override bool HasFocus
{
    get { return _hasFocus; }
    set
    {
        _hasFocus = value;

        if (_hasFocus)
            OnEnter(null);
        else
            OnLeave(null);
    }
}

#endregion

#region Constructor Region

public ListBox(Texture2D background, Texture2D cursor)
    : base()
{
    _hasFocus = false;
    _tabStop = false;
```

```
            this._image = background;
            this.Size = new Vector2(_image.Width, _image.Height);
            this._cursor = cursor;

            _lineCount = _image.Height / SpriteFont.LineSpacing;
            _startItem = 0;
            Color = Color.Black;
        }

        #endregion

        #region Abstract Method Region

        public override void Update(GameTime gameTime)
        {
        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(_image, Position, Color.White);
            Point position = Xin.MouseAsPoint;
            Rectangle destination = new Rectangle(0, 0, 100, (int)SpriteFont.LineSpacing);
            _mouseOver = false;

            for (int i = 0; i < _lineCount; i++)
            {
                if (_startItem + i >= _items.Count)
                {
                    break;
                }

                destination.X = (int)Position.X;
                destination.Y = (int)(Position.Y + i * SpriteFont.LineSpacing);

                if (destination.Contains(position) && Xin.MouseState.LeftButton ==
ButtonState.Pressed)
                {
                    _mouseOver = true;
                    _selectedItem = _startItem + i;
                    OnSelectionChanged(null);
                }

                if (_startItem + i == _selectedItem)
                {
                    spriteBatch.DrawString(
                        SpriteFont,
                        _items[_startItem + i],
                        new Vector2(Position.X, Position.Y + i * SpriteFont.LineSpacing),
                        SelectedColor);
                }
                else
                {
                    spriteBatch.DrawString(
                        SpriteFont,
                        _items[_startItem + i],
                        new Vector2(Position.X, Position.Y + i * SpriteFont.LineSpacing),
                        Color);
                }
            }
        }
```

```
public override void HandleInput()
{
    if (!HasFocus)
    {
        return;
    }

    if (Xin.CheckKeyReleased(Keys.Down))
    {
        if (_selectedItem < _items.Count - 1)
        {
            _selectedItem++;

            if (_selectedItem >= _startItem + _lineCount)
            {
                _startItem = _selectedItem - _lineCount + 1;
            }

            OnSelectionChanged(null);
        }
    }
    else if (Xin.CheckKeyReleased(Keys.Up))
    {
        if (_selectedItem > 0)
        {
            _selectedItem--;

            if (_selectedItem < _startItem)
            {
                _startItem = _selectedItem;
            }

            OnSelectionChanged(null);
        }
    }
    if (Xin.CheckMouseReleased(MouseButtons.Left) && _mouseOver)
    {
        HasFocus = true;
        OnSelectionChanged(null);
    }
    if (Xin.CheckKeyReleased(Keys.Enter))
    {
        HasFocus = false;
        OnSelected(null);
    }

    if (Xin.CheckKeyReleased(Keys.Escape))
    {
        HasFocus = false;
        OnLeave(null);
    }
}

#endregion

#region Method Region

protected virtual void OnSelectionChanged(EventArgs e)
{
```

```
            SelectionChanged?.Invoke(this, e);
        }

        protected virtual void OnEnter(EventArgs e)
        {
            Enter?.Invoke(this, e);
        }

        protected virtual void OnLeave(EventArgs e)
        {
            Leave?.Invoke(this, e);
        }

        #endregion
    }
}
```

This class also inherits from Control. There are three events for this control. SelectionChanged is if the selected item changes. Enter occurs when the control gets focus and Leave occurs when the control loses focus.

There are a lot of fields in this class. The _items field holds the items that the list box contains. The_startItem is the index in the _items field that displays first in the list box. The _lineCount field is the number of lines that the list box can draw. The _image field is the background image for list box. The _cursor field is for a background image. The _selectedColor field is what color the selected item is drawn in. The _selectedItem field is what item is currently being drawn.

There are read/write properties to expose the _selectedColor and _selectedIndex fields. There are properties to return the _selectedItem and the _items fields. The HasFocus property is overridden. The get returns the _hasFocus field. The set sets the _hasFocus field then if the control has focus it calls a method OnEnter that we will get to later and if it does not have focus calls the OnLeave event handler.

The constructor takes two Texture2Ds and one for the background and one for the cursor. It sets the _hasFocus field and _tabStop field to false. The _image field is set to the background parameter. The Size property of list box is set to the width and height of the background. The _cursor field is set to the cursor parameter. To determine how many lines can be drawn you take the height of the background image and divide that by the line spacing of the sprite font. The _startItem field is set to 0 and the Color property is set to black.

There is nothing happening in the Update method but it has to be implemented. The Draw method draws the background. It assigns a local variable to the position of the mouse as a point. It then creates a Rectangle that is 100 pixels wide and the height of the line spacing for the font. _mouseOver is assigned false. It then loops _lineCount times to draw the items. If _startItem + i is greater than or equal to the number of items it breaks out of the loops. The position of the destination rectangle is the X coordinate of the control and the Y coordinate plus the loop index times the line spacing of the font. If the destination rectangle contains the position of mouse and the left mouse button is down _mouseOver is set to true, _selectedItem is set to _startItem plus i and OnSelectionChanged called. If _startItem + i is the _selectedItem field then we draw the item in SelectedColor property otherwise the Color property. To determine where to draw the item I take the X coordinate of the list box and for

the Y coordinate the Y coordinate of the list box plus i times the LineSpacing property of the font. What item to draw is done by using the _startItem plus i.

The HandleInput method checks to see if the control has focus. If not it exits the method. It checks to see if the Down key has been released. If the _selectedItem is less than the number of items minus one it increases the _selectedItem field. If that is greater than or equal to _startItem plus _lineCount then _startItem is _selectedItem minus _lineCount plus one. Then the OnSelectionChange method is called. If the Up key has been released I check to see if the _selectedItem is greater than zero _selectedItem is decremented. If _selectedItem is less than _startItem we are scrolling below what is currently being drawn and the _startItem needs to be changed. OnSelectionChanged needs to be called. Now if the left button has been released and the mouse is over a line focus is given to the control and OnSelectionChanged is called. If the Enter key is released HasFocus is set to false and OnSelected is called. If the Escape key been released HasFocus is set to false and OnLeave is called.

The OnSelectionChanged method is called when the selection inside the list box is changed. If the SelectionChanged event is subscribed to the delegate is invoked using the instance and the EventArgs passed in. The OnEnter method is called when the list box gets focus. If the Enter event is subscribed to the delegate is invoked using the instance and the EventArgs passed in. The OnLeave method is called when the list box loses focus.

Before I get to the code for the editor I need to add a SpriteFont to the editor project. Open the Content Pipeline Tool in the ShadowEditor project. Click the Add New Item button. Select Sprite Font from the list and name it InterfaceFont.

With these controls I can now update the editor to paint using the tile set images that are selected during the new map process. I will also include the code for saving and loading maps now. Since there were a lot of changes to the editor code I will give the code for the entire Editor class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using ShadowMonsters;
using ShadowMonsters.Controls;
using ShadowMonsters.TileEngine;
using System;
using System.IO;
using System.Security.Cryptography;
using WF = System.Windows.Forms;

namespace ShadowEditor
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    public class Editor : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        int frameCount = 0;
        TileMap map;
```

```
Camera camera;
Rectangle viewPort = new Rectangle(0, 0, 64 * 18, 1080);
ControlManager controls;

PictureBox pbTileset;
PictureBox pbPreview;
ListBox lbLayers;
ListBox lbTileSets;
int selectedTile;

private readonly byte[] IV = new byte[]
{
    067, 197, 032, 010, 211, 090, 192, 076,
    054, 154, 111, 023, 243, 071, 132, 090
};

private readonly byte[] Key = new byte[]
{
    067, 090, 197, 043, 049, 029, 178, 211,
    127, 255, 097, 233, 162, 067, 111, 022,
};

public Editor()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    new Engine(viewPort);
    camera = new Camera();

    graphics.PreferredBackBufferWidth = 1920;
    graphics.PreferredBackBufferHeight = 1080;
    graphics.ApplyChanges();

    IsMouseVisible = true;
}

/// <summary>
/// Allows the game to perform any initialization it needs to before starting to run.
/// This is where it can query for any required services and load any non-graphic
/// related content.  Calling base.Initialize will enumerate through any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
    // TODO: Add your initialization logic here
    Components.Add(new Xin(this));
    base.Initialize();
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    controls = new ControlManager(Content.Load<SpriteFont>(@"InterfaceFont"))
    {
```

```
    AcceptInput = true
};

Texture2D background = new Texture2D(
    graphics.GraphicsDevice,
    100,
    150);

Color[] buffer = new Color[100 * 150];

for (int i = 0; i < buffer.Length; i++)
{
    buffer[i] = Color.White;
}

background.SetData(buffer);

Texture2D cursor = new Texture2D(
    GraphicsDevice,
    100,
    12);

buffer = new Color[100 * 12];

for (int i = 0; i < buffer.Length; i++)
{
    buffer[i] = Color.Blue;
}

cursor.SetData(buffer);

pbTileset = new PictureBox(
    new Texture2D(
        graphics.GraphicsDevice,
        512,
        512),
    new Rectangle(
        64 * 18,
        128,
        1920 - 64 * 18,
        512));
pbPreview = new PictureBox(
    new Texture2D(
        graphics.GraphicsDevice,
        64,
        64),
    new Rectangle(
        64 * 18,
        25,
        64,
        64));

lbLayers = new ListBox(
    background,
    cursor)
{
    Position = new Vector2(64 * 18, 800),
    TabStop = false
};
```

```
            lbLayers.Items.Add("Ground");
            lbLayers.Items.Add("Edge");
            lbLayers.Items.Add("Decorations");
            lbLayers.Items.Add("Building");

            lbTileSets = new ListBox(
                background,
                cursor)
            {
                Position = new Vector2(64 * 18 + 150, 800),
                TabStop = false
            };

            lbTileSets.SelectionChanged += LbTileSets_SelectionChanged;

            controls.Add(lbTileSets);
            controls.Add(lbLayers);
            controls.Add(pbTileset);
            controls.Add(pbPreview);
        }

        private void LbTileSets_SelectionChanged(object sender, System.EventArgs e)
        {
            pbTileset.Image = map.TileSet.Textures[lbTileSets.SelectedIndex];
            pbPreview.Image = map.TileSet.Textures[lbTileSets.SelectedIndex];
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// game-specific content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            Point tile;

            if (!IsActive)
                return;

            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            frameCount++;

            if (Xin.CheckKeyReleased(Keys.N) && frameCount > 5)
            {
                NewMapForm form = new NewMapForm(GraphicsDevice);

                form.ShowDialog();
```

```
                if (form.OkPressed)
                {
                    map = form.TileMap;
                    pbTileset.Image = map.TileSet.Textures[0];
                    pbPreview.Image = map.TileSet.Textures[0];

                    lbTileSets.Items.Clear();
                    foreach (string s in map.TileSet.TextureNames)
                        lbTileSets.Items.Add(s);

                    lbTileSets.SelectedIndex = 0;
                    pbPreview.SourceRectangle =
                        map.TileSet.SourceRectangles[0];
                    pbTileset.SourceRectangle = new Rectangle(
                        0,
                        0,
                        map.TileSet.Textures[0].Width,
                        map.TileSet.Textures[0].Height);
                }

                frameCount = 0;
            }

            Vector2 position = new Vector2
            {
                X = Xin.MouseAsPoint.X + camera.Position.X,
                Y = Xin.MouseAsPoint.Y + camera.Position.Y
            };

            tile = Engine.VectorToCell(position);

            if (map != null && viewPort.Contains(Xin.MouseAsPoint))
            {
                if (Xin.MouseState.LeftButton == ButtonState.Pressed)
                {
                    switch (lbLayers.SelectedIndex)
                    {
                        case 0:
                            map.SetGroundTile(tile.X, tile.Y, lbTileSets.SelectedIndex,
selectedTile);
                            break;
                        case 1:
                            map.SetEdgeTile(tile.X, tile.Y, lbTileSets.SelectedIndex,
selectedTile);
                            break;
                        case 2:
                            map.SetDecorationTile(tile.X, tile.Y, lbTileSets.SelectedIndex,
selectedTile);
                            break;
                        case 3:
                            map.SetBuildingTile(tile.X, tile.Y, lbTileSets.SelectedIndex,
selectedTile);
                            break;
                    }
                }

                if (Xin.MouseState.RightButton == ButtonState.Pressed)
                {
                    switch (lbLayers.SelectedIndex)
                    {
```

```
            case 0:
                map.SetGroundTile(tile.X, tile.Y, -1, -1);
                break;
            case 1:
                map.SetEdgeTile(tile.X, tile.Y, -1, -1);
                break;
            case 2:
                map.SetDecorationTile(tile.X, tile.Y, -1, -1);
                break;
            case 3:
                map.SetBuildingTile(tile.X, tile.Y, -1, -1);
                break;
        }
    }
}

if (pbTileset != null &&
    pbTileset.DestinationRectangle.Contains(
    Xin.MouseAsPoint) &&
    Xin.CheckMouseReleased(MouseButtons.Left))
{
    Point previewPoint = new Point(
        Xin.MouseAsPoint.X - pbTileset.DestinationRectangle.X,
        Xin.MouseAsPoint.Y - pbTileset.DestinationRectangle.Y);
    float xScale = (float)map.TileSet.Textures[lbTileSets.SelectedIndex].Width /
        pbTileset.DestinationRectangle.Width;

    float yScale = (float)map.TileSet.Textures[lbTileSets.SelectedIndex].Height /
        pbTileset.DestinationRectangle.Height;

    Point tilesetPoint = new Point(
        (int)(previewPoint.X * xScale),
        (int)(previewPoint.Y * yScale));

    Point clickedTile = new Point(
        tilesetPoint.X / map.TileSet.TileWidth,
        tilesetPoint.Y / map.TileSet.TileHeight);

    selectedTile = clickedTile.Y * map.TileSet.TilesWide + clickedTile.X;
    pbPreview.SourceRectangle =
        map.TileSet.SourceRectangles[selectedTile];
}

if (Xin.CheckKeyReleased(Keys.F1) && frameCount > 5)
{
    WF.SaveFileDialog sfd = new WF.SaveFileDialog();
    sfd.Filter = "Tile Map (*.map)|*.map";
    WF.DialogResult result = sfd.ShowDialog();

    if (result == WF.DialogResult.OK)
    {
        SaveMap(sfd.FileName);
    }
}

if (Xin.CheckKeyReleased(Keys.F2) && frameCount > 5)
{
    WF.OpenFileDialog ofd = new WF.OpenFileDialog();
    ofd.Filter = "Tile Map (*.map)|*.map";
    WF.DialogResult result = ofd.ShowDialog();
```

```
        if (result == WF.DialogResult.OK)
        {
            LoadMap(ofd.FileName);

            pbTileset.Image = map.TileSet.Textures[0];
            pbPreview.Image = map.TileSet.Textures[0];

            lbTileSets.Items.Clear();
            foreach (string s in map.TileSet.TextureNames)
                lbTileSets.Items.Add(s);

            lbTileSets.SelectedIndex = 0;
            pbPreview.SourceRectangle =
                map.TileSet.SourceRectangles[0];
            pbTileset.SourceRectangle = new Rectangle(
                0,
                0,
                map.TileSet.Textures[0].Width,
                map.TileSet.Textures[0].Height);
        }
    }
    controls.Update(gameTime);
    base.Update(gameTime);
}

private void SaveMap(string fileName)
{
    using (Aes aes = Aes.Create())
    {
        aes.IV = IV;
        aes.Key = Key;

        try
        {
            ICryptoTransform encryptor = aes.CreateEncryptor(Key, IV);
            FileStream stream = new FileStream(
                fileName,
                FileMode.Create,
                FileAccess.Write);
            using (CryptoStream cryptoStream = new CryptoStream(
                stream,
                encryptor,
                CryptoStreamMode.Write))
            {
                BinaryWriter writer = new BinaryWriter(cryptoStream);
                map.Save(writer);
                writer.Close();
            }
            stream.Close();
            stream.Dispose();
        }
        catch (Exception exc)
        {
            WF.MessageBox.Show("There was an error saving the map." + exc.ToString());
        }
    }
}

private void LoadMap(string fileName)
```

```
        {
            using (Aes aes = Aes.Create())
            {
                aes.IV = IV;
                aes.Key = Key;

                try
                {
                    ICryptoTransform decryptor = aes.CreateDecryptor(Key, IV);
                    FileStream stream = new FileStream(
                        fileName,
                        FileMode.Open,
                        FileAccess.Read);

                    using (CryptoStream cryptoStream = new CryptoStream(
                        stream,
                        decryptor,
                        CryptoStreamMode.Read))
                    {
                        BinaryReader reader = new BinaryReader(cryptoStream);
                        map = TileMap.Load(Content, reader);
                        reader.Close();
                    }

                    stream.Close();
                    stream.Dispose();
                }
                catch (Exception exc)
                {
                    WF.MessageBox.Show("There was a problem loading the map. " +
exc.ToString());
                }
            }
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            if (map != null)
            {
                map.Draw(gameTime, spriteBatch, camera, true);
            }

            spriteBatch.Begin();
            controls.Draw(spriteBatch);
            spriteBatch.End();

            base.Draw(gameTime);
        }
    }
}
```

Just a quick comment on the using statements. There are conflicts between several classes

and enumerations in System.Windows.Forms and Microsoft.Xna.Framework namespaces. For that reason I added an aliased using statement for System.Windows.Forms shortening it to WF.

There are some new fields. There is a ControlManager, controls, that will of course hold the controls. There are two PictureBox fields. pbTileset displays the currently tile set image and pbPreview displays the currently selected tile in the tile set. There are two ListBoxes, lbLayers holds what layer is being painted and lbTileSets holds the names of the tile sets. The selectedTile field is the currently selected tile for painting. I copied the IV and Key fields from the GamePlayState.cs so when I'm saving the encryption works the same.

There was only one change to the constructor. I set the IsMouseVisible property to true so that we can see where we are painting.

In the LoadContent method I create the control manager instance passing in the font that we added. It sets the AcceptInput property to true so that the control manager will process key presses. I create a texture for the background of the list boxes passing in a white buffer. I do the same for the cursor of the list boxes but blue instead of white. I create the tile set picture box passing in a 512 by 512 background image. The X coordinate is 64 * 18 and the Y coordinate is 128. The Width of the picture box is 1920 - 64 * 18 pixels and the height is 512 pixels. The preview picture box is 64 pixels by 64 pixels. I position it at 64 * 18 and 25. The height and width is set to 64 pixels. The lbLayers list box is passed the background and cursor that were created earlier. The Position property is set to 64 * 18 and 800 and TabStop is set to false. I then add the layers to the items collection. Similarly I create the lbTileSets list box using the background and cursor textures. It is then positioned at 64 * 18 + 150 and 800 with TabStop false. I then add an event handler for the SelectionChanged event. The controls are then added to the control manager.

The handler for the SelectionChanged event sets the Image property of pbTileset to be the texture of the tile set at the selected index. It does the same thing for the pbPreview.

The Update method is where most of the changes occurred. I updated the code for creating a new map. After creating a new map the Image property of the pbTileset and pbPreview picture boxes to be the first image in the tile set. The lbTileSets Items collection is cleared. I loop over the TextureNames property of the tile set and add the name to the Items collection. The SelectedIndex property of the lbTileSets list box is set to 0. The SourceRectangle of the pbPreview is set to the first source rectangle of the tile set. The SourceRectangle of the pbTileSet picture box is set to the Width and Height of the first texture.

The code for painting was updated. I added a switch inside of the code that will run if the left or right mouse buttons are pressed. In case 0 I call the SetGroundTile method, in case 1 I call the SetEdgeTile method, in case 2 I call SetDecorationTile and in case 3 I call SetBuildingTile. For the left button I pass in tile.X, tile.Y, lbTileSets.SelectedIndex and selectedTile has parameters for each. For the right button I pass in tile.X, tile.Y, -1 and -1.

Next I handle selecting which tile to paint with. That is determined by clicking on pbTileset. There is a check to see if pbTileset is not null, the destination rectangle for the picture box contains the mouse as a point and that the left button was released. If all of those conditions are true I calculate where in the picture box the click occurred. That is calculated by taking

location of the click and subtracting the position of the picture box. Since the image for the tile set is not the same as I need to calculate what the scaling factor is between the two. That is done by taking the with of the tile set texture and dividing it by the width of the destination rectangle. I then calculate where the click occurred in the tile set by multiplying the preview coordinates by the scale. I then figure out which tile was clicked by dividing by the width and height of the tiles. I then set selectedTile to be the clickedTile.Y times the number of tiles wide the tile set is plus clickedTile.X. I set the SourceRectangle for pbPreview to be the source rectangle of the tile.

If the F1 key has been released and frameCount is greater than 5 I trigger the save process. I do that by creating a SaveFileDialog. It is preceded by the alias for System.Windows.Forms. I give the file the extension .map. I then show the dialog. If the result is OK I call the SaveMap method passing in the file name of the dialog.

If the F2 key has been released and frameCount is greater than 5 I trigger the load process. I do that by creating an OpenFileDialog. I give the file extension .map again. I show the dialog and if the result is OK I call the LoadMap method passing in the file name of the dialog. I then set the Image property of pbTileset to the first texture of the tile set as well as the pbPreview. The lbTileSets.Items is cleared and then the texture names are added to the Items. I then set the selected index to the first tile. The SourceRectangle of pbPreview is set to the first source rectangle of the tile set. The SourceRectangle of pbTileset is set to be the height and width of the texture.

The last thing to do in the Update method is to call the Update method of the control manager.

The SaveMap method is similar to the code for saving the game. You create an Aes instance using the Create method. The IV and Key are set to be the same IV and Key for saving and loading games. You then create an encryptor to encrypt the data. You need a file stream with write access. You then need a CryptoStream that needs the stream and the encryptor with write access. A BinaryWriter is created using the CryptoStream. The Save method of the TileMap class is called. If there is an error saving the map it is displayed in the catch block.

The LoadMap method works the reverse of the SaveMap method but is similar in structure. You need to create an Aes instance. Set the IV and Key properties. Create a decryptor and a stream with read access. A CryptoStream is created using the stream and the decryptor with read access. A BinaryReader is created with the CryptoStream. I then load the map passing in the ContentManager and the reader. I try the error and display an error message.

In the Draw method I call Begin on SpriteBatch. I then call the draw method of the ControlManager passing in the SpriteBatch. Finally I call the End method of the SpriteBatch.

I'm going to wrap this tutorial up here. I will be starting work on the next tutorial shortly. Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so.

I wish you the best in your MonoGame Programming Adventures!