Creating a Role Playing Game with XNA Game Studio 3.0
Part 25
Adding a Sprite – Part 3

To follow along with this tutorial you will have to have read the previous tutorials to understand much of what it going on. You can find a list of tutorials here: XNA 3.0 Role Playing Game Tutorials You will also find the latest version of the project on the web site on that page. If you want to follow along and type in the code from this PDF as you go, you can find the previous project at this link: Eyes of the Dragon - Version 24 You can download the graphics from this link: Graphics.zip

In today's tutorial I will be finishing with adding the sprite for the player to control. If you recall from the last tutorial the sprite moved well but he could walk off the screen and the camera wouldn't follow the sprite. In this tutorial I will lock the camera to the sprite and make it so that the sprite can not walk off the map. Notice that I said the sprite could not walk off the map, not that the sprite could not walk off the screen. The reason that I said map and not screen is you want the sprite to be tied to the map and not the screen.

The first thing I will do is make a modification to the **Animation** class. To lock the camera and the sprite to the map it is important to know the width and height of the sprite. Fortunately when we created our animations we passed in the width and height of the frames. We will use that to set the width and height of the frames. The **Animation** class is still short so I will give you the new class and then explain the changes. This is the code for the new **Animation** class.

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

namespace New2DRPG.SpriteClasses
{
    public class Animation : ICloneable
    {
        Rectangle[] frames;
        int framesPerSecond;
        TimeSpan frameLength;
        TimeSpan frameTimer;
        int currentFrame;
        int frameWidth;
        int frameHeight;

        public Animation(int frameCount, int frameWidth,
                         int frameHeight, int xOffset, int yOffset)
        {
            frames = new Rectangle[frameCount];
            this.frameWidth = frameWidth;
            this.frameHeight = frameHeight;

            for (int i = 0; i < frameCount; i++)
            {
                frames[i] = new Rectangle(
```

```
                xOffset + (frameWidth * i),
                yOffset,
                frameWidth,
                frameHeight);
        }
        FramesPerSecond = 5;
        Reset();
    }

    private Animation()
    {
        FramesPerSecond = 5;
    }

    public int FramesPerSecond
    {
        get { return framesPerSecond; }
        set
        {
            if (value < 1)
                framesPerSecond = 1;
            else if (value > 60)
                framesPerSecond = 60;
            else
                framesPerSecond = value;
            frameLength = TimeSpan.FromSeconds(1 / (double)framesPerSecond);
        }
    }

    public Rectangle CurrentFrameRect
    {
        get { return frames[currentFrame]; }
    }

    public int CurrentFrame
    {
        get { return currentFrame; }
        set
        {
            currentFrame = (int)MathHelper.Clamp(value, 0, frames.Length - 1);
        }
    }

    public int FrameWidth
    {
        get { return frameWidth; }
    }

    public int FrameHeight
    {
        get { return frameHeight; }
    }

    public void Update(GameTime gameTime)
    {
        frameTimer += gameTime.ElapsedGameTime;

        if (frameTimer >= frameLength)
        {
```

```
                frameTimer = TimeSpan.Zero;
                currentFrame = (currentFrame + 1) % frames.Length;
            }
        }

        public void Reset()
        {
            currentFrame = 0;
            frameTimer = TimeSpan.Zero;
        }

        public object Clone()
        {
            Animation animation = new Animation();
            animation.frames = this.frames;
            animation.frameWidth = this.frameWidth;
            animation.frameHeight = this.frameHeight;
            animation.Reset();
            return animation;
        }
    }
}
```

In the public constructor I set the **frameWidth** field to the **frameWidth** passed in. I did the same thing with the **frameHeight** field. I set it to the **frameHeight** passed in. Since I'm a big believer in keeping data that is inside of a class private I decided to expose the **frameWidth** and **frameHeight** fields through properties. The get only property **FrameWidth** returns **frameWidth** and the get only property **FrameHeight** returns **frameHeight**. The only other change in this class was in the **Clone** method. Since in the private constructor there was no way to know what the width and height of the frames were I needed a way to set the **frameWidth** and **frameHeight** fields. Fortunately like I did for the **frames** field I can use the **this** keyword to assign the fields. If you recall the **this** keyword is used to reference the current object. The **frameWidth** field is set to **this.frameWidth** and **frameHeight** is set to **this.frameHeight**. Those are all of the changes to the **Animation** class.

Now I will turn my attention to the **Sprite** class. You will need to know the width and the height of the sprite to be able to make sure it does not walk off the map. There were no width and height fields in the **Sprite** class or the **AnimatedSprite** class. I decided to add protected fields to the **Sprite** class so all inherited classes will have access to them. The width and height of the sprite should never change so I decided to expose then through get only properties. I also needed to set them initially when a new sprite is created and that of course is done in the constructor. The first thing you will want to do is add these two fields to the **Sprite** class.

```
        protected int width;
        protected int height;
```

Now you will want to set these in the constructors. I will give you the code for the constructor that takes a **Game** object and a **Texture2D** first. For this one you can just set **width** to **texture.Width** and set **height** to **texture.Height**. This is the code for the new constructor.

```
public Sprite(Game game, Texture2D texture)
    : base(game)
{
    spriteBatch =
        (SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));
```

```
    this.texture = texture;
    width = texture.Width;
    height = texture.Height;
    position = Vector2.Zero;
    velocity = Vector2.Zero;
    center = new Vector2(texture.Width / 2,
        texture.Height / 2);

    scale = 1.0f;
    rotation = 0.0f;

    sourceRectangle = new Rectangle(0,
                            0,
                            texture.Width,
                            texture.Height);
}
```

The other constructor takes a third parameter, a source **Rectangle**. You can use the **Width** property of the rectangle to set the **width** field. You can use the **Height** property to set the **height** field. This is the code for the second constructor.

```
public Sprite(Game game, Texture2D texture, Rectangle sourceRectangle)
    : base(game)
{
    spriteBatch =
        (SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));

    this.texture = texture;
    this.sourceRectangle = sourceRectangle;

    position = Vector2.Zero;
    velocity = Vector2.Zero;
    center = new Vector2(sourceRectangle.Width / 2,
        sourceRectangle.Height / 2);

    width = sourceRectangle.Width;
    height = sourceRectangle.Height;

    scale = 1.0f;
    rotation = 0.0f;
}
```

Like I set I also added in get only properties to return the width and the height. **Width** will return **width** and **Height** will return **height**. This is the code for the properties.

```
public int Width
{
    get { return width; }
}

public int Height
{
    get { return height; }
}
```

In the constructor for the **AnimatedSprite** class. I need to set a few fields. I need to set the

width and **height** fields. I also need to set the **center** field. The **center** field will be used when I lock the camera to the sprite. I set the **width** field to the **FrameWidth** property of the first animation, since all of the animations are the same width I can do this. I set the **height** field to the **FrameHeight** property of the first animation. I can again do this because the height of the animations never changes. Finally I set the **center** field to be a new **Vector2** with the **X** property being **width / 2** and the **Y** property to being **height / 2**. This is the code for the new constructor of the **AnimatedSprite** class.

```
public AnimatedSprite(Game game, Texture2D texture, List<Animation> animations)
    : base(game, texture)
{
    spriteBatch = Game1.TileSpriteBatch;
    this.animations = animations;
    currentAnimation = AnimationKey.Down;
    isAnimating = false;
    width = animations[(int)currentAnimation].FrameWidth;
    height = animations[(int)currentAnimation].FrameHeight;
    center = new Vector2(width / 2, height / 2);
}
```

For the **AnimatedSprite** class I will also be adding a public method called **LockToMap**. This method will keep the sprite from walking off the map. Because the map is locked to the view port this will also keep the sprite from walking off the screen. You can add this method to the **AnimatedSprite** class.

```
public void LockToMap()
{
    if (position.X < 0)
        position.X = 0;

    if (position.Y < 0)
        position.Y = 0;

    if (position.X + width > TileMap.WidthInPixels)
        position.X = TileMap.WidthInPixels - width;

    if (position.Y + height > TileMap.HeightInPixels)
        position.Y = TileMap.HeightInPixels - height;
}
```

Keeping the sprite from walking of the left or top edge of the map is easy. If the **X** value of the sprite's position is less than zero it will go off the left edge of the map so you want to make sure it is never less than zero. If the **Y** value of the sprite's position is less than zero it will go off the top edge of the map so you want to make sure that it is never less than 0. That is what the first two if statements do. If **position.X** is less than 0 it sets **position.X** to 0. Similarily, if **position.Y** is less than 0 then **position.Y** is set to 0.

To keep the sprite from walking off the right edge of the map is a little more difficult. You need to add in the width of the sprite to the **X** value of the sprite's position to see if it will walk off the right edge of the map. If you recall the property **TileMap.WidthInPixels** holds the width of the map in pixels so I compared the **X** value of the sprite's position plus the width of the sprite to that property . If the above condition is true you want to set the position of the sprite to be the width of the map in pixels minus the width of the sprite.

To keep the sprite off the bottom of the map you need to do something similar. You check to see if the **Y** value of the sprite's position plus the height of the sprite is greater than the height of the map in pixels. If that condition is true then you set the **Y** value of the sprite's position to the height of the map in pixels minus the height of the sprite.

Now we need to lock the camera to the sprite. The way I'm going to do this is that the camera will start to scroll horizontally when the sprite, plus half of its width, reaches half the width of the map. When the sprite plus half of its width is at the center of the screen it will be perfectly centered horizontally. So when the sprite starts out in the top left corner the map will not start to scroll until the sprite reaches the middle of the screen. Once the camera reaches the width of the map minus the width of the viewport it will stop and the sprite will continue to walk on until it reaches the right edge of the map. The same is true in reverse when the sprite will walk from the right edge of the map toward the left edge of the map. To control the vertical scrolling a similar process is used. Once the sprite reaches half the height of the view port, plus half of the sprite's height, it will be perfectly centered vertically and the map will start to scroll vertically.

I will create a new method in the **Camera** class that will take a **Sprite** as a parameter to lock the camera to the sprite. Why am I using the **Sprite** class instead of the **AnimatedSprite** class? The reason I'm using **Sprite** instead of **AnimatedSprite** is so that you can pass in any class that inherits from **Sprite** to this method to lock the camera. This is another use of polymorphism, which I am a big fan of. Simply polymorphism is the ability of a base class to act as an inherited class. So I can pass any class that inherits from **Sprite** to the method. After I give you the code for the method I will explain it. This is the code for the method.

```
public void LockToSprite(Sprite sprite)
{
    Position.X = sprite.Position.X + sprite.Center.X
                 - (TileEngine.ViewPortWidth / 2);
    Position.Y = sprite.Position.Y + sprite.Center.Y
                 - (TileEngine.ViewPortHeight / 2);
}
```

So, what exactly does this method do? First it sets the **X** value of the camera's position to the **X** value of the sprite's postion and adds half of the width of the sprite to it and then subtracts half the width of the viewport. Similarily for the **Y** value of the camera's position it takes the **Y** position of the sprite and adds half the height of the sprite and then subtracts half the height of the view port.

All that is left to do is implement this into the game. This will be taken care of in the **Game1** class in the **HandleActionScreenInput** method. I will give you the code and then explain the changes that I made. This is the code for the new **HandleActionScreenInput** method.

```
private void HandleActionScreeenInput()
{
    if (CheckKey(Keys.Escape))
    {
        playerSprite.IsAnimating = false;
        activeScreen.Enabled = false;
        activeScreen = quitActionScreen;
        activeScreen.Show();
    }
    else if (CheckKey(Keys.V))
```

```
{
    playerSprite.IsAnimating = false;
    activeScreen.Enabled = false;
    activeScreen = viewCharacterScreen;
    viewCharacterScreen.SetPlayerCharacter(playerCharacter);
    activeScreen.Show();
}
else
{
    Vector2 motion = new Vector2();
    playerSprite.IsAnimating = true;
    if (newState.IsKeyDown(Keys.Up) || newState.IsKeyDown(Keys.NumPad8))
    {
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Up;
    }
    if (newState.IsKeyDown(Keys.Down) || newState.IsKeyDown(Keys.NumPad2))
    {
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Down;
    }
    if (newState.IsKeyDown(Keys.Right) || newState.IsKeyDown(Keys.NumPad6))
    {
        motion.X++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.Left) || newState.IsKeyDown(Keys.NumPad4))
    {
        motion.X--;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (newState.IsKeyDown(Keys.NumPad9))
    {
        motion.X++;
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad3))
    {
        motion.X++;
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad1))
    {
        motion.X--;
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (newState.IsKeyDown(Keys.NumPad7))
    {
        motion.X--;
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (motion != Vector2.Zero)
    {
        motion.Normalize();
    }
```

```
            else
            {
                playerSprite.IsAnimating = false;
            }

            playerSprite.Position += motion * playerSprite.Speed;
            playerSprite.LockToMap();

            camera.LockToSprite(playerSprite);
            camera.LockCamera();
        }
}
```

All of the changes take place at the bottom of the method. The order in which you do things is important. The first thing to do is to make sure the sprite will not walk off the map. So you will call the **LockToMap** method of the sprite. Then you want to lock the camera to the sprite so you call the **LockToSprite** method of the camera passing in **playerSprite**. The last step is to lock the camera to the view port by calling **LockCamera**.

I was going to add in how to create the **Tile Set Generator** to this tutorial. I decided against it in the end though. I will write up a separate tutorial on creating that and try and have it up on my web site in the very near future.

There is just one more thing that I would like to add to the game that you can remove later on. You do not have to follow this step. It is completely optional. When you are writing games it is often handy to debug the game in a window instead of in full screen mode. If you are writing a Windows game, this will not work for an XBOX or Zune game, there is a way that you can have a message box pop up and ask if you want to run the game in full screen mode and then use that information to pass a parameter to a constructor of the game of the and set the **IsFullScreen** property.

To do this you will need to add a reference to the project. Under the **Project** menu in visual studio there is an option **Add reference**. Click that option. Under the **.NET** tab you will want to select the **System.Windows.Forms** entry. This will give you access to the **MessageBox** class.

You will hardly ever do this in an XNA game but open the **Program.cs** file. You will want to change it to this. I will explain how this works in a moment.

```
using System;
using System.Windows.Forms;

namespace New2DRPG
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            bool fullScreen;

            DialogResult result;
            result = MessageBox.Show("Run the game in full screen mode?",
                    "Eyes of the Dragon",
```

```
                    MessageBoxButtons.YesNo);

            if (result == DialogResult.Yes)
                fullScreen = true;
            else
                fullScreen = false;

            using (Game1 game = new Game1(fullScreen))
            {
                game.Run();
            }
        }
    }
}
```

There is a using statement for **System.Windows.Forms**. That gives you access to the classes for the **MessageBox** and the **DialogResult** enum. The **Main** method is the first method that is called in a C# program. In the **Main** method there is a bool variable **fullScreen**. This variable will be set and passed to a new constructor for the **Game1** class, the main class for an XNA game. There is another variable of **DialogResult** called **result**. This variable will hold the result returned from the message box. The overload for the **Show** method of the **MessageBox** class that I used requires three parameters. The first on is the text of the message box. The second is the caption of the message box, what shows up in the title bar. The third requires an enum of **MessageBoxButtons**. This parameter defines which buttons will appear on the message box. I chose **MessageBoxButtons.YesNo** to have **Yes** and **No** buttons on the message box. In an if-else statement I check to see if the result returned from the message box is **DialogResult.Yes** which means the user pressed the **Yes** button. If the **Yes** button was pressed then I set **fullScreen** to true. In all other cases I set **fullScreen** to false. The last change is in the **using** statement where the **Game1** class is created. I put the variable **fullScreen** in as a parameter.

Now you will have to return to the **Game1** class and add in a new constructor that takes a bool as a parameter. Do not just modify the old constuctor. You will be able to use it later on when you are finished the game and want to always run it in full screen mode. This is the code for the second constuctor in the **Game1** class.

```
public Game1(bool fullScreen)
{
    graphics = new GraphicsDeviceManager(this);
    graphics.PreferredBackBufferWidth = 1024;
    graphics.PreferredBackBufferHeight = 768;
    graphics.PreferredBackBufferFormat = SurfaceFormat.Bgr32;
    graphics.IsFullScreen = fullScreen;
    this.Window.Title = "Eyes of the Dragon";
    Content.RootDirectory = "Content";
}
```

The only difference between this constructor than the other one is that **graphics.IsFullScreen** is set to the value of the parameter **fullScreen**. So if true is passed as the parameter, if the user presses the **Yes** button, the game will be run in full screen mode. Otherwise it will be run in windowed mode. This is an easy trick that you can do for when you are debugging your games so you don't have to always go into the code and set **graphics.IsFullScreen** to true or false everytime. You do not have to do this but I find it useful when debugging a game as when you are running the game in full screen mode sometimes you can not get back to the debugger to find out where your program crashed.

Well that is it for this tutorial. I am already working on coding the next part of Eyes of the Dragon so I encourage you to keep either visiting my site http://xna.jtmbooks.com or my blog, http://xna-rpg.blogspot.com for the latest news on these tutorials.