Creating a Role Playing Game with XNA Game Studio 3.0
Part 23
Adding a Sprite – Part 1

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

In the next few tutorials I will be working on adding a sprite for the player character. This will be a multistep process because there are many things that will need to be changed to accomplish what I am looking to do. I will start with adding in custom classes to work with animating a sprite and I will draw the sprite on the screen. I like Nick Gravelyn's method for animating sprites. I will, however, be adding in my own style of coding to accomplish this as well. I am using the sprites from the game Last Guardian that can be found on evilmana.com. These sprites are under the Creative Commons License which in the simplest terms means you can use them for non-commercial use. You can find the full license here: http://creativecommons.org/licenses/by-nc/2.5/

To get started you will want to load the last version of the project and download the sprite sheet that I will be using, the new tile set I created and a simple HUD. You can find them in the Graphics.zip file. Right click the **Content** folder and add a new folder called **Sprites**. Right click the new folder and select **Add existing item** and select the **amg1large.png** image. Next right click the **Tilesets** folder and add in the **tileset1.png** file. The tile set is not complete but there is enough in it to suit our purposes. Finally you will want to add in the updated HUD. So, right click the **Backgrounds** folder and add in **characterhud.png**.

I made a few small changes to the **TileEngine** class. All that the changes were is that I changed the **tileHeight** variable to be **64** and the **viewPortHeight** variable to **704**. This is the updated class. The reason I did this was I thought it would be nice if the tiles on the screen were 64 by 64 and the sprites were the same size, 64 by 64. I could have scaled the sprites to fit the 64 by 48 rectangles but I decided to go with simplicity.

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

namespace New2DRPG
{
    public static class TileEngine
    {
        static int tileWidth = 64;
        static int tileHeight = 64;

        static int viewPortWidth = 1024;
        static int viewPortHeight = 704;

        public static int TileWidth
```

```
        {
            get { return tileWidth; }
        }

        public static int TileHeight
        {
            get { return tileHeight; }
        }

        public static Vector2 ViewPortVector
        {
            get
            {
                return new Vector2(viewPortWidth + tileWidth,
                    viewPortHeight + tileHeight);
            }
        }

        public static int ViewPortWidth
        {
            get { return viewPortWidth; }
        }

        public static int ViewPortHeight
        {
            get { return viewPortHeight; }
        }

    }
}
```

There are a variety of ways to do animation. I will be using the idea of frame animation; which in my opinion is the simplest form of animation. Frame animation is the idea of having multiple frames for the animation of your sprite, much like how animation is done in movies. You can think of a frame like a snap shot from an animated movie. By slightly altering the next frame you give the illusion that the object is animating. In our case for each direction the sprite is facing we have two frames. At a fixed interval every second we will flip from one frame to the other. These are the frames for the down animation of the sprite I am using.

In the first frame we will draw the first image. In the next frame we will flip to the second. In the third we will go back to the first. Animation can be done using any number of frames though. You are not limited to just two frames. If you look at sprite for RPG Maker I believe that they use three frames per sprite. XNA has a great way of implementing this type of animation. You've already seen me use it in the tile engine. You can have a large rectangle and specify a source rectangle for the sprite. That means I can use an image, like the one above, and just specify the portion of that image that I want to draw.

I will be adding in two classes to the **SpriteClasses** folder. The first class is an animation class that handles the frame animation. The second class is for the sprite itself. So right click the **SpriteClasses** folder and add a new class called **Animation**. As I usually do I will give you the code

and then explain the way it works.

```
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;

        public Animation(int frameCount, int frameWidth,
                         int frameHeight, int xOffset, int yOffset)
        {
            frames = new Rectangle[frameCount];
            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 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.Reset();
            return animation;
        }
    }
}
```

This class uses the XNA **Rectangle** class so I required the using statement for the XNA framework in this class. The class declaration might seem strange to some. It looks like it is using inheritance but it is actually implementing an interface call **ICloneable**. Unlike inheritance with classes a class, or struct, can implement many different interfaces. Interfaces contain signatures for methods, events or delegates that the class is implementing must use. **ICloneable** is used to clone, or copy, an object. The reason this is important is that classes are reference types. What that means is if you have an object of a class called **object1** and assign it to **object2** any changes you make to **object2** effect **object1**. The reason why I used **ICloneable** is it is an easy way to make a copy of an object. After you create the animations for one sprite many other sprites in the game can use copies of the original animation because they all use the same number of frames and are in the same order. If you were to just assign the animation from the player character's sprite to an enemy sprite for example and changes the enemy sprite would make would effect the player character's sprite. This can be a hard topic for beginners to understand and many struggle with it until they see it in practice. At the end of the tutorial I will give you a quick sample program to demonstrate this.

There are six fields in this class. The first one **frames** is an array of **Rectangle** that will hold the frames for each **Rectangle** that defines a frame for animation. Like I mentioned earlier you can specify a smaller part of a whole image to draw. This array will hold the source rectangles to be drawn for the individual frames. The next field **framesPerSecond** holds how many frames per second the sprite will

animate. The next two fields are **TimeSpan** objects called **frameLength** and **frameTimer**. When it comes to dealing with timing I like to use the **TimeSpan** structure. **TimeSpan** is a great to measure changes in time. The first one, **frameLength**, holds the amount of time that should pass before moving on to the next frame. The second one, **frameTimer**, holds the elapsed time since the last frame was drawn. The last field **currentFrame** holds the current frame being drawn.

There are two constructors for this class. The first one is public and can be used outside of the class. The second one is private and can only be used by the class. I will use the second constructor when I implement the **ICloneable** interface.

The first constructor has five parameters. That is a lot of parameters but they are all important. The first parameter, **frameCount**, is for the number of frames for the animation. **frameWidth** and **frameHeight** hold the width and the height of the frames and will be used create the rectangles for each frame. The next two **xOffset** and **yOffset** may be a little more difficult to understand.

If you look at the image that I am using for the sprite all of the sprites are in one line. The reason for this is to make the process of animating them, and creating the frames easier. For the down animation to determine where to start the frames you would begin at 0. The next set of frames, the up animation would start at 128 because the sprites are 64 pixels wide. So it would have an x offset of 128 and offset for the left and right would be 256 and 384. If I was to stack all of the sprites in one vertical sheet I could specify a y offset just like the x offset.

In the public constructor I intialize all of the values. First I create an array of **Rectangle** with a length of **frameCount**. Then inside a for loop I create the rectangles for each frame. I use a method similar to the method I used to create the rectangles for the tiles in the tile set. There isn't a second loop because all of the sprites are in the same row. The only thing that changes is the **X** coordinate for each sprite. It is calculated by taking **xOffset** and adding the current frame times the width of the frame. So the first rectangle will be an **xOffset** and the next rectangle will be at **xOffset** plus the frame width and so on. I then use a property that I will explain in a moment called **FramesPerSecond** to set the number of frames to animate per second to 5 and the length of each frame. Then I call a method **Reset** that will reset the animation to the first frame and reset the frame timer.

The private constructor is much simpler. It just calls the property **FramesPerSecond** and set the number of frames to animate per second to 5. When I implement the cloning I will call the **Reset** method so there is no need to call it here.

The properties for this class are: **FramesPerSecond**, **CurrentFrameRect** and **CurrentFrame**. **FramesPerSecond** is a rather complicated property in the set part anyway. The other two are much simpler. **CurrentFrameRect** just returns the **Rectangle** for the current frame. **CurrentFrame** is used to get and set the **currentFrame** field. The get part just returns **currentFrame**. The set part uses the **MathHelper.Clamp** method to preform validation on the value passed in. **currentFrame** can never be negative and can never be greater than the total number of frames minus 1.

The get property just returns the field **framesPerSecond**. The set part is a little more complicated. There is a sequence of if and else statements. I didn't see a reason to set the number of frames to animate less than 1 frame per second and you definitely don't want that value to be negative so if the value passed to the property is less than 1 I set it to 1. In the else-if I check to see if the value is greater than 60. The reason I chose 60 is that XNA tries to call the **Update** and **Draw** methods 60 times per second so there is no point in trying to have a sprite trying to animate more than 60 frames per second. If all other cases fail the value is okay. The next part is where I calculate the length of the frames. I use the **FromSeconds** method of **TimeSpan** to calculate this. To find this value you take 1 and divide it by the number of frames you want to animate per second. I cast **framesPerSecond** to a double so there will be no rounding when the division takes place.

There is an **Update** method that takes a **GameTime** parameter. **gameTime** will be used to measure how much time has passed since the last call to **Update**. I use the **ElapsedGameTime** property and add that to the **frameTimer** variable. Then if **frameTimer** is greater than **frameLength** it is time to update the current frame. I reset **frameTimer** to **TimeSpan.Zero** and then calculate the next frame. I used Nick Gravelyn's trick here. If you take the current frame and add 1 and then get the remainder using the number of frames, using the % or modulus operator, the values will always be with in the proper range. So you can use this single statement instead of three.

The **Reset** method is used to reset the animation back to the starting point. **currentFrame** is set to 0, the first frame. Then **frameTimer** is set to **TimeSpan.Zero** which means no time has elapsed.

The last method in this class is the implementation of **ICloneable** and is called **Clone**. When you add an interface to a class definition you will get a little blue symbol under the **I**, which is usually used to denote an interface, you can press **shift+alt+F10** or hover your mouse of the symbol. You will be given the option to explicitly implement the interface. I suggest that you do that. This interface only has one method: **Clone**. The **Clone** method returns an **object** and you will have to cast the return type.

The **Clone** method creates a new **Animation** object called **animation**. Usually when you are cloning an object it is a good idea to use copies of the fields. There is no reason to make a copy of the **frames** field because it is never changed once set. If you did make a copy it would use up memory so I use the **this** keyword and set the **frames** field of the new animation to the **frames** field of the current animation. I call the **Reset** method just to make sure that all values are set to their initial position. Finally I return the new animation.

There is another new class to add to the game. Right click the **SpriteClasses** folder and to this folder add a new class called **AnimatedSprite**. This class will represent an animated sprite that implements the **Animation** class. This is the code for the **AnimatedSprite** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using New2DRPG.CoreComponents;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace New2DRPG.SpriteClasses
{
    public enum AnimationKey { Down, Up, Left, Right };
```

```
    public class AnimatedSprite : Sprite
    {
        List<Animation> animations = new List<Animation>();
        AnimationKey currentAnimation;
        bool isAnimating;
        float speed = 2.0f;

        public AnimatedSprite(Game game, Texture2D texture,
                              List<Animation> animations)
            : base(game, texture)
        {
            this.animations = animations;
            currentAnimation = AnimationKey.Down;
            isAnimating = false;
        }

        public float Speed
        {
            get { return speed; }
            set
            {
                speed = MathHelper.Clamp(value, 0.1f, 10f);
            }
        }
        public bool IsAnimating
        {
            get { return isAnimating; }
            set { isAnimating = value; }
        }

        public AnimationKey CurrentAnimation
        {
            get { return currentAnimation; }
            set { currentAnimation = value; }
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
            if (isAnimating)
                animations[(int)currentAnimation].Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            Vector2 spritePosition = position - Game1.Camera.Position;
            base.Draw(gameTime);
            spriteBatch.Draw(
                texture,
                spritePosition,
                animations[(int)currentAnimation].CurrentFrameRect,
                Color.White);
        }
    }
}
```

There are three using statements that had to be added to this class. This class inherits from the **Sprite** class so I needed an using statement for **New2DRPG.CoreComponents**. I also needed using

statements for the XNA framework and the XNA framework graphics namespaces. I decided to have a public enum at the namespace level. I remembered that enums do not have to be part of a class but can be part of a namespace. Having the enum outside the class but part of a namespace allows you to use it with out qualifying it with a class outside of the class. The enum, **AnimationKey**, has four members: **Down**, **Up**, **Left** and **Right**. Their order is important however. I will discuss that in a moment.

Because I am using the **Sprite** class as a base class there are not as many fields as there could be in this class. For example **texture, spriteBatch** and **position** are all inherited from the base class **Sprite**. If you have done Nick's tile engine tutorials you will see that this class is, in a lot of ways, different from his implementation. For example he uses a **Dictionary** for the animations and I decided to go with a **List** with an enum so there was no need to check if a string is in the **Dictionary** and the members of the enum can easily be used to select the animations from the **List**. It is important to note that to use my method you have to add the animations to the **List** in the same order that they are in the enum. The animation for the sprite moving down has to be first, followed by up, left and right. I could have went with a **Dictionary** for this but I decided that having this restriction out weighed the complexity of using a **Dictionary**.

The fields in this class are: **animations** which is a **List<Animation>** that will hold the animations for the sprite, **currentAnimation** is of type **AnimationKey** and will be used to determine which animation to use, **isAnimating** will be used to determine if the sprite is currently animating and **speed** is a field that will be used later on in the game but I added it in now. For the moment you don't really need to worry about. It is just important to note that it will be used in controlling the speed the sprite moves across the screen.

I chose to use the first constructor from the **Sprite** class that takes a **Game** object and a **Texture2D** as its parameters. It also takes a third parameter which is a **List<Animation>** that will hold the animations for the sprite. There is a call to the base constructor with the **Game** and **Texture2D** objects. The constructor just sets the fields for the class.

There are three properties in this class. The first one **Speed** is used to expose the **speed** field. The get part just returns the **speed** field. The set part uses the **MathHelper.Clamp** method to clamp to value passed in between 0.1f and 10f. **IsAnimating** is used to expose the **isAnimating** field that controls if the sprite is animating or not. The third property **CurrentAnimation** is used to control which animation to use.

There is an override of the **Update** method of the parent class **Sprite**. If the sprite is currently animating I call the **Update** method of the approriate animation. This is where the order becomes important. As you can see to find out which animation to use I cast **currentAnimation** to an int. Since the animations were added in the same order as the members of the enum **AnimationKey**. Since the values of the members of the enum will be 0 for **Down**, 1 for **Up**, 2 for **Left** and 3 for **Right** and the animations are added to the **List<Animation>** in the same order the first animation will be the down animation, the second up, the third left and the last right.

There is also an override of the **Draw** method. In this method I create a **Vector2** to hold the position of the sprite minus the position of the camera. Then I the overload of the **Draw** method I use takes four parameters. The first is the texture of the sprite, the second is a **Vector2** which is the position to draw the sprite, the third is a **Rectangle** that is the source rectangle in the texture and the last is the tint color. The source rectangle is found using the **CurrentFrameRect** property of the animation. I find which animation to use the same as in the **Update** method.

Now I will add a sprite to the game. There is only ever going to be one sprite for the player at a time so I decided to make it static. As well I added in a get only property to expose it to the rest of the game. Add this code to the **Game1** class near where the **Camera** object is. You will also want to add a using statement for the **New2DRPG.SpriteClasses** class to **Game1**.

```
using New2DRPG.SpriteClasses;

static AnimatedSprite playerSprite;

public static AnimatedSprite PlayerSprite
{
   get { return playerSprite; }
}
```

Now we have to actually create the sprite. I decided to do this in the **LoadContent** method. I will go over the code after I have shown it to you. This is the new **LoadContent** method.

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Services.AddService(typeof(SpriteBatch), spriteBatch);
    Services.AddService(typeof(ContentManager), Content);

    normalFont = Content.Load<SpriteFont>("normal");

    LoadCreatePCScreen();
    LoadStartScreen();
    LoadHelpScreen();
    LoadActionScreen();
    LoadQuitPopUpScreen();
    LoadGenderPopUpScreen();
    LoadClassPopUpScreen();
    LoadDifficultyPopUpScreen();
    LoadNameInputScreen();
    LoadCreditScreen();
    LoadIntroScreen();
    LoadViewCharacterScreen();

    creditScreen.Hide();
    startScreen.Hide();
    helpScreen.Hide();
    createPCScreen.Hide();

    activeScreen = introScreen;
    activeScreen.Show();

    List<Animation> animations = new List<Animation>();

    Animation tempAnimation;
    tempAnimation = new Animation(2, 64, 64, 0, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 128, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 256, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 384, 0);
```

```
    animations.Add(tempAnimation);

    Texture2D playerSpriteTexture = Content.Load<Texture2D>(@"Sprites\amg1large");

    playerSprite = new AnimatedSprite(this, playerSpriteTexture, animations);
    playerSprite.CurrentAnimation = AnimationKey.Down;
    playerSprite.IsAnimating = true;
}
```

All of the new code is at the end of the method. I created a **List<Animation> animations** that will hold the different animations for the sprite. I created a variable **tempAnimation** to hold the animations that I will add to **animations**. Then I created the various animations. I had explained earlier how this works but I will go over it again. The constructor for the **Animation** class requires 5 parameters. The firs it the number of animations. This will always be 2 in our case because each of the animations has two frames. Next is the width and height of the frames. This is also constant and 64 for each. The next parameter is the interesting one **xOffset**. For the first animation it is 0 because that is where the animation for the player moving down is. Since there are 2 frames for each animation and the frames are 64 pixels wide each the next animation starts at 128 pixels. The next will begin at 256 pixels and the last will begin at 384. Since the animations are all in a straight line the last parameter never changes as well. After loading in the texture for the sprite I create the sprite. I then set the current animation to the **Down** animation and set it to be animating so you can see that the sprite actually animates.

Since the sprite is related to the player's character I decided to take care of updating it and drawing it in the **PlayerCharacter** class. The **Draw** method of the **PlayerCharacter** class just calls the **Draw** method of the sprite. This is the code for the **Draw** method.

```
public override void Draw(GameTime gameTime)
{
    DrawHitPointsSpellPoints();
    Game1.PlayerSprite.Draw(gameTime);
    base.Draw(gameTime);
}
```

This is a minor thing and I will leave this up to you to decide which you like better. At the moment I have the sprite being drawn on to of the hit points and the spell points. If you like it the other way around just call the **Draw** method of the sprite before the **DrawHitPointsSpellPoints** method.

I also added in an overrid of the **Update** method in the **PlayerCharacter** class to control the animations. This method just calls the **Update** method of the sprite and **base.Update**. This is the new method.

```
public override void Update(GameTime gameTime)
{
    Game1.PlayerSprite.Update(gameTime);
    base.Update(gameTime);
}
```

There is just one thing left to do for this tutorial. I have to call the **Update** method of the **PlayerCharacter** class in the **ActionScreen** class to update the **PlayerCharacter** class and thus the sprite for the player character. This is the new **Update** method for the **ActionScreen** class.

```
public override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    playerCharacter.Update(gameTime);
}
```

If you compile and run the game now the sprite will be animating as if it was walking down the screen once you get to the **ActionScreen**. At the moment the sprite does not scroll with the camera. If you scroll the map to the right or down the sprite will disappear off the screen. I will get to moving the sprite with the screen in the next tutorial.

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.