Eyes of the Dragon Tutorials
Part 7
Animated Sprite

I'm writing these tutorials for the MonoGame 3.8 framework using Visual Studio 2019. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon page of my web blog. I will be making each version of the project available on GitHub here. It will be included on the page that links to the tutorials.

The first thing I want to do is fix a bug in the previous tutorial. I neglected to add the name selector to the control manager. Change the **CreateControls** method of the **CharacterGeneratorScreen** to the following.

```
private void CreateControls()
{
    Texture2D leftTexture = Game.Content.Load<Texture2D>(@"GUI\leftarrowUp");
    Texture2D rightTexture = Game.Content.Load<Texture2D>(@"GUI\rightarrowUp");
    Texture2D stopTexture = Game.Content.Load<Texture2D>(@"GUI\StopBar");

    backgroundImage = new PictureBox(
        Game.Content.Load<Texture2D>(@"Backgrounds\titlescreen"),
        GameRef.ScreenRectangle);

    ControlManager.Add(backgroundImage);

    Label label1 = new Label
    {
        Text = "Who will search for the Eyes of the Dragon?"
    };
    label1.Size = label1.SpriteFont.MeasureString(label1.Text);
    label1.Position = new Vector2(
        (GameRef.Window.ClientBounds.Width - label1.Size.X) /2, 150);

    ControlManager.Add(label1);

    nameSelector = new LeftRightSelector(leftTexture, rightTexture, stopTexture);
    nameSelector.SetItems(maleName, 125);
    nameSelector.Position = new Vector2(label1.Position.X, 200);

    ControlManager.Add(nameSelector);

    genderSelector = new LeftRightSelector(leftTexture, rightTexture, stopTexture);
    genderSelector.SetItems(genderItems, 125);
    genderSelector.Position = new Vector2(label1.Position.X, 250);
    genderSelector.SelectionChanged += GenderSelector_SelectionChanged;

    ControlManager.Add(genderSelector);

    classSelector = new LeftRightSelector(leftTexture, rightTexture, stopTexture);
    classSelector.SetItems(classItems, 125);
    classSelector.Position = new Vector2(label1.Position.X, 300);

    ControlManager.Add(classSelector);

    LinkLabel linkLabel1 = new LinkLabel
    {
```

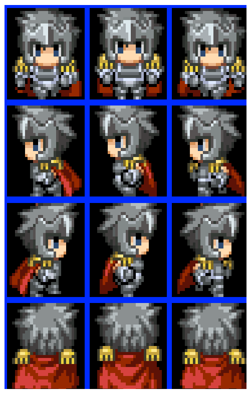
```
                Text = "Accept this character.",
                Position = new Vector2(label1.Position.X, 350)
            };
            linkLabel1.Selected += new EventHandler(LinkLabel1_Selected);

            ControlManager.Add(linkLabel1);
            ControlManager.NextControl();
        }
```

In this tutorial I will be adding in an animated sprite for the player to control. For that you will of course need sprites. I will be using the sprites I used from my XNA 3.x RPG series to use for this tutorial. You can download them from http://cynthiamcmahon.ca/blog/downloads/playersprites.zip.

After you have downloaded and decompressed the sprites you will need to add them to the content project. Open the **MonoGame Pipeline Tool** by double clicking the **Content.mgcb** file in the **Content** folder of the **EyesOfTheDragonContent** project. Right click the **Content** node select **Add** and then **New Folder**. Call this new folder **PlayerSprites**. Right click the **PlayerSprites** folder, select **Add** and then **Existing Item**. Navigate to where you decompressed the sprites. Select all eight sprites to add them. The names are of the format **genderclass.png**. Where **gender** is the gender of the character and **class** is the class of the character. So, **malewizard** is the sprite for a male wizard. I don't have non-binary sprite so we will use the female ones instead.

It will be best to have the sprite classes part of the **MGRpgLibrary** project. There will be more than just these animated sprites so I added in a folder for all sprite class. Right click the **MGRpgLibrary** project, select **Add** and then **New Folder**. Name this new folder **SpriteClasses**.

Animation in computer games is done much like it was done when the first animated cartoons came out. It is the process of repeatedly drawing one image after another to give the illusion that the image is changing. When done at an appropriate speed the person seeing the illusion will see the image changing. If you look at the image below the blue squares show the different images, or frames. There are three images in each row. The artist created the images so that you go from the first frame to the second frame to the third and then back to the first. That is the way I will be implementing the animation of the sprite. I will have a frame rate, the number of frames that will be drawn each second. I've found that five frames per second is a reasonable rate. I will be creating a class for this type of animation.



Right click the SpriteAnimations folder in the **MGRpgLibary**, select **Add** and then **Class**. Name this new class **Animation**. This is the code for the **Animation** class.

```
using System;
using Microsoft.Xna.Framework;

namespace MGRpgLibrary.SpriteClasses
{
    public enum AnimationKey { Down, Left, Right, Up }

    public class Animation : ICloneable
    {
        #region Field Region

        Rectangle[] frames;
        int framesPerSecond;
        TimeSpan frameLength;
        TimeSpan frameTimer;
        int currentFrame;
        int frameWidth;
        int frameHeight;

        #endregion

        #region Property Region

        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; }
}

#endregion

#region Constructor Region

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(Animation animation)
{
    this.frames = animation.frames;

    FramesPerSecond = 5;
}

#endregion

#region Method Region

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;
```

```
        }

        #endregion

        #region Interface Method Region

        public object Clone()
        {
            Animation animationClone = new Animation(this)
            {
                frameWidth = this.frameWidth,
                frameHeight = this.frameHeight
            };

            animationClone.Reset();

            return animationClone;
        }

        #endregion
    }
}
```

There is a using statement for the MonoGame Framework because this class uses the **Rectangle** class. There is an enumeration called **AnimationKey**. It describes the different types of animations the sprite has. It can be down, up, left and right. You could also have animations for moving on the diagonals. The sprite I used doesn't have these animations. This class implements the **ICloneable** interface. The reason is you may have multiple sprites in the game that have the same lay out as the player's sprite. It is better to store the animations in a master list of animations and return copies of them. This is a class so if you pass just the instance and you make changes it will affect the original.

There are quite a few fields in this class. The first one, **frames**, is an array of rectangles for the source rectangles of the animation. If you look back at the image all of the animations for one direction are in the same row. The reason will become clear when I get to the constructor. The next field is an integer and holds the number of frames to animate per second. The next two fields are of type **TimeSpan**. They hold the length of each frame of the animation and the time since the last animation started. Using the **TimeSpan** structure gives you a finer degree of control over the animations than using a double or floating point value. The next field, **currentFrame**, is the index of the current animation in the array of rectangles. The last two fields, **frameWidth** and **frameHeight**, hold the height and width of the frames. The one downfall of this method is that all of the frames must have the same width and height.

There is another method that you could use to animate the sprites but I believe this is the best approach. There are several properties in this class to expose the fields they represent. The first one is **FramesPerSecond** and it is used to get and set the number of frames the sprite will animate in one second. The get part is trivial, it just returns the value in the **framesPerSecond** field. The set part is a little more interesting. It does a little validation on the values passed in. The first check is to make sure that the value is not less than one. If it was you would get some pretty bizarre results. It also makes sure that the frame rate is not greater than sixty. This value is actually pretty high. Checking for a value of ten would probably be a better idea. After validating the values it sets the **frameLength** field. What is important here is you want to make sure you cast the **framesPerSecond** field as a double when you

do the division. If you don't then integer division will be preformed and you will get either one or zero, depending on if **framesPerSecond** was one or not. You will often need to know what the current rectangle of the animation is so there is a property **CurrentFrameRect** to retrieve that. You will not only want to be able to get what the current frame is but also set it. You will again have to make sure that the value passed in is valid. I did that using the **MathHelper.Clamp** method which makes sure that the value is with in the minimum and maximum values, inclusive, passed in. There are also properties that get the width and height of the frames, **FrameWidth** and **FrameHeight** respectively.
There are two constructors for this class, a public one and a private one. The public takes as parameters the number of frames for the animation, the width of each frame, the height of each frame, the X offset of the frame and the Y offset of the frame. The last two's purpose probably isn't obvious. If you go back to the image of the sprite sheet above you will see that it is split up into rows and columns. The first row of animations begins at coordinates (0, 0). The second row of animations begins at coordinates (0, 32). The third and forth at (0, 64) and (0, 96) respectively. As you can see as you move down in rows the Y value changes. This is the Y offset value. You could also have had the animations in two rows and two columns. In this case you would also have had an X offset. Later I will get to having more than just the walking animation and the X offset will be useful.

The constructor then sets the **frames**, **frameWidth**, and **frameHeight** first. Then, in a for loop, it creates each of the rectangles to describe the frames for the animation. This is where the X and Y offsets become important. When you create the first animation you will be passing in the values 3, 32, 32, 0, and 0. For the second you will be passing in 3, 32, 32, 0, and 32. In the for loop to find the X coordinate in the sprite sheet for the rectangle you take the X offset and add the width of the frame times the frame counter, which is **i** the loop index. In this case since all of the animations or on the same row you can just use the Y offset value. The **Width** and **Height** values are set using the **frameWidth** and **frameHeight** values passed in. I then use the **FramesPerSecond** field to set the number of frames for the sprite to be 5. I then call the **Reset** method of the class which sets the animation in its starting position.

The second private constructor is used when I implement the **ICloneable** interface. This constructor takes an **Animation** object as its parameter. The reason is because as I mentioned above about the **frames** field being marked readonly. This constructor sets the frames field of the new animation to the **frames** field of the old animation. It then sets the **FramesPerSecond** to 5 like the other constructor.
There are two methods, **Update** and **Reset**, that are for controlling the animation of the sprite. The third method, **Clone**, implements the **ICloneable** interface. The **Update** method takes a **GameTime** parameter. This parameter measures how much time has elapsed since the last call to **Update** in the **Game1** class. This is used to determine when it is time to move to the next frame of the animation. What I do is add the **ElapsedGameTime** property of the **GameTime** parameter to **frameTimer**. This is used to determine when to move to the next frame. It is time to move to the next frame when **frameTimer** is greater than or equal to **frameLength** so there is an if statement that checks for that. If it is you want to reset **frameTimer** to the zero position and move to the next frame. There is an nice mathematical formula next to calculate which frame to move to. In our case we have three frames that are at index 0, 1, and 2. If you add 1 to that you will have the values 1, 2, and 3. You then get the remainder of dividing those by the number of frames. Those values will be 1, 2, and 0. So if you are on frame 0 you will move to frame 1, from frame 1 to frame 2, and from frame 2 back to frame. The **Reset** method is very simple. It sets the current frame to be 0 and then sets the time back to 0 as well. The last method in this class is the **Clone** method. This method is from the **ICloneable** interface. This

method returns a copy of the current animation. The first step is to create a new **Animation** object using the private constructor, passing in the current **Animation** object. The next step is to set the **frameWidth** and **frameHeight** fields. I call the **Reset** method to reset the **Animation** to the first frame. I then return the new **Animation** as an **object**. The **Clone** method always returns an **object** as its return value. When you use the **Clone** method of the **Animation** class you will have to cast the return value to be of type **Animation**.

The next class I'm going to add is a class for an animated sprite. Right click the **SpriteClasses** folder, select **Add** and then **Class**. Name this new class **AnimatedSprite**. This is the code for that class.

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using MGRpgLibrary.TileEngine;

namespace MGRpgLibrary.SpriteClasses
{
    public class AnimatedSprite
    {
        #region Field Region

        Dictionary<AnimationKey, Animation> animations;
        AnimationKey currentAnimation;
        bool isAnimating;
        Texture2D texture;
        public Vector2 Position;
        Vector2 velocity;
        float speed = 2.0f;

        #endregion

        #region Property Region

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

        public bool IsAnimating
        {
            get { return isAnimating; }
            set { isAnimating = value; }
        }

        public int Width
        {
            get { return animations[currentAnimation].FrameWidth; }
        }

        public int Height
        {
            get { return animations[currentAnimation].FrameHeight; }
        }

        public float Speed
        {
            get { return speed; }
            set { speed = MathHelper.Clamp(speed, 1.0f, 16.0f); }
        }

        public Vector2 Velocity
        {
            get { return velocity; }
            set
            {
                velocity = value;
                if (velocity != Vector2.Zero)
                    velocity.Normalize();
            }
        }

        #endregion

        #region Constructor Region

        public AnimatedSprite(Texture2D sprite, Dictionary<AnimationKey, Animation> animation)
        {
            texture = sprite;

            animations = new Dictionary<AnimationKey, Animation>();

            foreach (AnimationKey key in animation.Keys)
                animations.Add(key, (Animation)animation[key].Clone());
        }

        #endregion

        #region Method Region

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

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
        {
            spriteBatch.Draw(
                texture,
                Position - camera.Position,
                animations[currentAnimation].CurrentFrameRect,

            Color.White);
        }

        public void LockToMap()
        {
            Position.X = MathHelper.Clamp(Position.X, 0, TileMap.WidthInPixels - Width);
            Position.Y = MathHelper.Clamp(Position.Y, 0, TileMap.HeightInPixels - Height);
        }

        #endregion
    }
}
```

This is a little different from previous **AnimatedSprite** classes that I made but the core class is the same. The difference is in the constructor of the class. There are using statements to bring a few classes of the MonoGame Framework into scope as well as our **MGRpgLibrary.TileEngine** name space. There are a number of fields in this class. The first is a **Dictionary<AnimationKey, Animation>** to hold the animations of the sprite. I used a dictionary because you can only have one entry per key and it is easier than having to remember which animation was added when if you were to use a **List<T>**. The **currentAnimation** field is for the current animation of the sprite. The bool field **isAnimating** is used to tell if the sprite is currently animating so the animation should be updated in the **Update** method.

The **texture** field is for the sprite sheet. The next two field are **Vector2** fields and are for the position and velocity of the sprite. I will be controlling the speed the sprite moves across the screen like the scrolling of the map. I will normalize the vector and then multiply it by a constant value, the **speed** field that holds the speed the sprite moves.

There are properties to expose information about the sprite. **CurrentAnimation** is a get and set property for the **currentAnimation** field. The **IsAnimating** property is also a get and set property and is for the **isAnimating** field. You will often need to know the height and width of the sprite. The **Height** and **Width** properties return the height and width of the current frame of the animation. The **Speed** property is also get and set and exposes the **speed** field. The set part clamps the speed between 1 and 16 using **MathHelper.Clamp**. The **Velocity** property exposed the **velocity** field. It is get/set as well. However, the set part checks to see if the value passed in is **Vector2.Zero** because it normalizes the vector and you can't normalize a vector that has zero length.

The constructor takes two parameters. A **Texture2D** for the sprite and a **Dictionary<AnimationKey, Animation>** for the animations. You don't have to pass a clone of the animations to the constructor as it will clone the animations passed in. The constructor sets the **texture** field to the texture passed in and then creates a new **Dictionary<AnimationKey, Animation>**. Then in a for each look it loops through all of the keys in the key collection of the dictionary passed in. Inside the loop it adds the key with a clone of the animation to the dictionary in the class.
The **Update** takes the **GameTime** parameter to be able to update the animation. It checks to see if the sprite is currently animating. If it is it calls the **Update** method of the current animation.

The **Draw** method takes three parameters. The **GameTime** parameter of our game's **Draw** method, a **SpriteBatch** between calls to **Begin** and **End**, a **Camera** for the map. The reason is the sprite, the map, and the camera are all related. In drawing the map you subtract the position of the camera. You will also need to subtract it from any object on the map, including sprites. Things are going to get more than a little complicated having to subtract the camera here and other things. I've got a good solution though.

To draw the sprite I use the overload that takes the texture for the sprite, a **Vector2** for its position, a source rectangle, and the tint colour.

The **LockToMap** method is used to keep the sprite from going off the map. You need to clamp the **X** coordinate between zero and the width of the map in pixels minus the width of the sprite. If you don't subtract the width of the sprite it will move off the screen to the right. Similarly, you clamp the **Y** coordinate between zero and the height of the map in pixels minus the height of the sprite.

For now I'm going to add a sprite to the game play screen. In the next tutorial I will move things around a bit. The first step for adding the sprite to the **GamePlayScreen** will be to add a using statement for the **SpriteClasses** name space of our **MGRpgLibrary**. and add a field for the sprite in the **Field** region.

```
using MGRpgLibrary.SpriteClasses;

        AnimatedSprite sprite;
```

In the **LoadContent** method you load in a sprite sheet and construct the animations and add them to a dictionary to pass to the constructor of the sprite. This is the new **LoadContent** method.

```
        protected override void LoadContent()
        {
            Texture2D spriteSheet = Game.Content.Load<Texture2D>(@"PlayerSprites\malefighter");
            Dictionary<AnimationKey, Animation> animations = new Dictionary<AnimationKey,
                Animation>();

            Animation animation = new Animation(3, 32, 32, 0, 0);
            animations.Add(AnimationKey.Down, animation);

            animation = new Animation(3, 32, 32, 0, 32);
            animations.Add(AnimationKey.Left, animation);

            animation = new Animation(3, 32, 32, 0, 64);
            animations.Add(AnimationKey.Right, animation);

            animation = new Animation(3, 32, 32, 0, 96);
            animations.Add(AnimationKey.Up, animation);

            sprite = new AnimatedSprite(spriteSheet, animations);

            base.LoadContent();

            Texture2D tilesetTexture = Game.Content.Load<Texture2D>(@"Tilesets\tileset1");

            Tileset tileset1 = new Tileset(tilesetTexture, 8, 8, 32, 32);
            tilesetTexture = Game.Content.Load<Texture2D>(@"Tilesets\tileset2");

            Tileset tileset2 = new Tileset(tilesetTexture, 8, 8, 32, 32);

            List<Tileset> tilesets = new List<Tileset>();

            tilesets.Add(tileset1);
            tilesets.Add(tileset2);

            MapLayer layer = new MapLayer(40, 40);

            for (int y = 0; y < layer.Height; y++)
            {
                for (int x = 0; x < layer.Width; x++)
                {
                    Tile tile = new Tile(0, 0);
                    layer.SetTile(x, y, tile);
                }
            }
            MapLayer splatter = new MapLayer(40, 40);

            Random random = new Random();

            for (int i = 0; i < 80; i++)
            {
                int x = random.Next(0, 40);
                int y = random.Next(0, 40);
                int index = random.Next(2, 14);

                Tile tile = new Tile(index, 0);

                splatter.SetTile(x, y, tile);
            }

            splatter.SetTile(1, 0, new Tile(0, 1));
            splatter.SetTile(2, 0, new Tile(2, 1));
            splatter.SetTile(3, 0, new Tile(0, 1));

            List<MapLayer> mapLayers = new List<MapLayer>();

            mapLayers.Add(layer);
            mapLayers.Add(splatter);

            map = new TileMap(tilesets, mapLayers);
        }
```

The new code just loads in a texture. It then creates the animations. I chose the male fighter sprite sheet for the sprite. To create the animations you need to know the offset values for X and Y. The X offsets are the same as each animation is on a row of its own. The thing that changes is the Y offset. Since the sprites are 32 pixels high the offsets will be multiples of 32. 0 times 32 for the top row, 1 times 32 for the second, 2 times 32 for the third and 4 time 32 for the last. After creating the animations and loading the sprite sheet I create the sprite.

In the **Update** method you will want to call the **Update** method of the sprite. In the **Draw** method you will want to call the **Draw** method of the sprite. Change the **Update** and **Draw** methods in the **GamePlayScreen** to the following.

```
        public override void Update(GameTime gameTime)
        {
            player.Update(gameTime);
            sprite.Update(gameTime);

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin(
                SpriteSortMode.Immediate,
                BlendState.AlphaBlend,
                SamplerState.PointClamp,
                null,
                null,
                null,
                Matrix.Identity);
            map.Draw(GameRef.SpriteBatch, player.Camera);
            sprite.Draw(gameTime, GameRef.SpriteBatch, player.Camera);

            base.Draw(gameTime);
            GameRef.SpriteBatch.End();
        }
```

The sprite isn't moving at the moment though. It just sits up in the top corner of the map. We want th sprite to move not the camera but at the same time we want the player to be able to explore the map with the camera. The way I'm going to implement it is to have two modes for the camera. In the one mode the camera will follow the sprite. In the other the camera has free movement.

The first step is to update the **Camera** class. I ended up making quite a few changes to the **Camera** class so I will give you the code for the entire class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MGRpgLibrary.SpriteClasses;
namespace MGRpgLibrary.TileEngine
{
    public enum CameraMode { Free, Follow }

    public class Camera
    {
        #region Field Region

        Vector2 position;
        float speed;
        float zoom;
        Rectangle viewportRectangle;
        CameraMode mode;

        #endregion

        #region Property Region

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

        public float Speed
        {
            get { return speed; }
            set
            {
                speed = (float)MathHelper.Clamp(speed, 1f, 16f);
            }
        }
```

```
public float Zoom
{
    get { return zoom; }
}

public CameraMode CameraMode
{
    get { return mode; }
}

#endregion

#region Constructor Region
public Camera(Rectangle viewportRect)
{
    speed = 4f;
    zoom = 1f;
    viewportRectangle = viewportRect;
    mode = CameraMode.Follow;
}
public Camera(Rectangle viewportRect, Vector2 position)
{
    speed = 4f;
    zoom = 1f;
    viewportRectangle = viewportRect;
    Position = position;
    mode = CameraMode.Follow;
}

#endregion

#region Method Region

public void Update(GameTime gameTime)
{
    if (mode == CameraMode.Follow)
        return;

    Vector2 motion = Vector2.Zero;

    if (InputHandler.KeyDown(Keys.Left) ||
        InputHandler.ButtonDown(Buttons.RightThumbstickLeft, PlayerIndex.One))
        motion.X = -speed;
    else if (InputHandler.KeyDown(Keys.Right) ||
        InputHandler.ButtonDown(Buttons.RightThumbstickRight, PlayerIndex.One))
        motion.X = speed;

    if (InputHandler.KeyDown(Keys.Up) ||
        InputHandler.ButtonDown(Buttons.RightThumbstickUp, PlayerIndex.One))
        motion.Y = -speed;
    else if (InputHandler.KeyDown(Keys.Down) ||
        InputHandler.ButtonDown(Buttons.RightThumbstickDown, PlayerIndex.One))
        motion.Y = speed;

    if (motion != Vector2.Zero)
    {
        motion.Normalize();
        position += motion * speed;
        LockCamera();
```

```
            }
        }

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

        public void LockToSprite(AnimatedSprite sprite)
        {
            position.X = sprite.Position.X + sprite.Width / 2
                - (viewportRectangle.Width / 2);
            position.Y = sprite.Position.Y + sprite.Height / 2
                - (viewportRectangle.Height / 2);

            LockCamera();
        }

        public void ToggleCameraMode()
        {
            if (mode == CameraMode.Follow)
                mode = CameraMode.Free;
            else if (mode == CameraMode.Free)
                mode = CameraMode.Follow;
        }

        #endregion
    }
}
```

I add in a using statement to bring the **SpriteClasses** name space into scope. I also added in a enum called **CameraMode** that has two values: **Free** and **Follow**. I went this route rather than using a bool as the bool could get confusing. The **Free** value allows for free movement and the **Follow** value has the camera follow the player's sprite. I add a field, **mode**, that holds the mode of the camera. There is also a read only property, **CameraMode**, that exposes the field's value. The constructors both set the **mode** field to **Follow** initially.

I also made many changes to the **Update** method. The first is at the start I check to see if the field **mode** is set to **Follow**. If it is the player's sprite is controlling the camera and you don't want free movement so exit the method. I added in support for the Xbox 360 controller for free movement as well. Cameras are usually controlled with the right thumb stick so I add in checks for the right thumb stick in the direction if the arrow keys. Since I use an or in the condition either the keyboard or the thumb stick will control movement. If both are down the camera behaves the same way as if either are down. I also moved the code for updating the camera's position and locking the camera inside the if that checks for motion. I did that because if there is no motion there is no need to update the position and lock the camera if needed.

I also added in a public method **LockToSprite** that will snap the camera to the sprite. The method takes an **AnimatedSprite** as a parameter. The camera is tied to the position of the sprite. The camera's

X coordinate is found by taking the **X** position of the sprite, adding half the width of the sprite and subtracting half the width of the view port. What this does is have the map not scroll horizontally until the center of the sprite is past the center of the view port. The camera's **Y** coordinate is found by taking the **Y** position of the sprite, adding half the height of the sprite and subtracting half the height of the view port. It then calls **LockCamera** to keep the map from scrolling off the view port.

There is one last method, **ToggleCameraMode**. This method just moves from one camera mode to another. If the mode of the camera was **Free** it is set to **Follow** and if it was **Follow** it is set to **Free**. The last thing is to implement the movement in the **GamePlayScreen**. I'm also going to add toggling of the view mode of the camera. The sprite will be controlled using the **W**, **A**, **S**, and **D** keys or left thumb stick of the controller. Pressing the **C** key or left thumb stick will snap the camera to the sprite. Finally, pressing the **F** key or right thumb stick toggles the camera mode. Change the **Update** method of the **GamePlayScreen** to the following.

```
public override void Update(GameTime gameTime)
        {
            player.Update(gameTime);
            sprite.Update(gameTime);

            Vector2 motion = new Vector2();

            if (InputHandler.KeyDown(Keys.W) ||
                InputHandler.ButtonDown(Buttons.LeftThumbstickUp, PlayerIndex.One))
            {
                sprite.CurrentAnimation = AnimationKey.Up;
                motion.Y = -1;
            }
            else if (InputHandler.KeyDown(Keys.S) ||
                InputHandler.ButtonDown(Buttons.LeftThumbstickDown, PlayerIndex.One))
            {
                sprite.CurrentAnimation = AnimationKey.Down;
                motion.Y = 1;
            }

            if (InputHandler.KeyDown(Keys.A) ||
                InputHandler.ButtonDown(Buttons.LeftThumbstickLeft, PlayerIndex.One))
            {
                sprite.CurrentAnimation = AnimationKey.Left;
                motion.X = -1;
            }
            else if (InputHandler.KeyDown(Keys.D) ||
                InputHandler.ButtonDown(Buttons.LeftThumbstickRight, PlayerIndex.One))
            {
                sprite.CurrentAnimation = AnimationKey.Right;
                motion.X = 1;
            }

            if (motion != Vector2.Zero)
            {
                sprite.IsAnimating = true;

                motion.Normalize();

                sprite.Position += motion * sprite.Speed;
                sprite.LockToMap();
                if (player.Camera.CameraMode == CameraMode.Follow)
                    player.Camera.LockToSprite(sprite);
            }
            else
            {
                sprite.IsAnimating = false;
            }

            if (InputHandler.KeyReleased(Keys.F) ||
                InputHandler.ButtonReleased(Buttons.RightStick, PlayerIndex.One))
            {
                player.Camera.ToggleCameraMode();

                if (player.Camera.CameraMode == CameraMode.Follow)
                    player.Camera.LockToSprite(sprite);
            }

            if (player.Camera.CameraMode != CameraMode.Follow)
            {
                if (InputHandler.KeyReleased(Keys.C) ||
                    InputHandler.ButtonReleased(Buttons.LeftStick, PlayerIndex.One))
                {
                    player.Camera.LockToSprite(sprite);
                }
            }

            base.Update(gameTime);
        }
```

You want the sprite to move at a uniform speed so I set up a **Vector2** to hold the motion of the sprite so it can be normalized and multiplied by the constant speed. There is an if-else-if statement that checks if the **W** key is down, or the right thumb stick in the up direction. The else if checks for the **S** key and the right thumb stick in the down direction. I check for vertical motion before horizontal motion. The reason is it looks better if the sprite is moving diagonally to use the left and right animations rather then the up and down animations. If the sprite is moving up the **Y** component of the motion vector is set to -1 and the current animation of the sprite to up. Similarly, for the down direction test the **Y** component of the motion vector is set to 1 and the current animation is set to the down animation.

I didn't include the if-else-if for horizontal motion with the if-else-if for vertical motion to allow the sprite to move diagonally. This will make our life a pain down the road but it is what players expect now, free motion. The if-else-if works the same as before. The one for the **A** key and the left thumb stick left sets the **X** component of the motion vector to -1 and the animation to the left animation. For **D** and left thumb stick right, set the **X** component of the motion vector to 1 and the animation to right. The next if statement checks to see if there is motion to process. If there is, I set the **IsAnimating** property to true so the sprite will animate. I then normalize the vector to have motion uniform and update the sprite's position using the normalized motion vector and multiplying it by the sprite's speed.

I then call the **LockToMap** method of the sprite to make sure it doesn't travel off the map. If the camera is in follow mode you also want to call the **LockToSprite** method of the camera passing in the sprite. If there was no motion then **IsAnimating** is set to false so the sprite will no longer animate. The next if statement checks to see if the F key or the right thumb stick have been released since the last frame. If it has I call the **ToggleCameraMode** method to switch the mode of the camera. If the mode is now the follow mode you want to lock the camera to the sprite.

The last if checks to see if the mode of the camera is not follow as there is no need to snap to the sprite as the camera is already locked to the sprite. In that if I check to see if the C key or the left thumb stick have been been released since the last frame. If they have I call the **LockToSprite** method of the camera to snap the camera to the sprite's position.

Things are starting to come together but there is still a lot of work to be done. I think this is enough for this tutorial. I'd like to try and keep them to a reasonable length so that you don't have too much to digest at once. I encourage you to visit the news page of my site, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials.

Good luck in your game programming adventures!
Cynthia