A Summoner's Tale – MonoGame Tutorial Series

Chapter 5

Player Component

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called A Summoner's Tale. 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 web site: A Summoner's Tale. The source code for each tutorial will be available as well. I will be using Visual Studio 2013 Premium for the series. The code should compile on the 2013 Express version and Visual Studio 2015 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 add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

This tutorial will add a game component that will represent the player and some classes for animated sprites. I will be starting with the classes for animation first. Right click the TileEngine folder, select Add and then Class. Name this new class Animation. Here it the code.

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

namespace Avatars.TileEngine
{
    public class Animation
    {
        #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);

            animationClone.frameWidth = this.frameWidth;
            animationClone.frameHeight = this.frameHeight;
            animationClone.Reset();

            return animationClone;
        }

        #endregion
    }
}
```

This class implements frame animation that is similar to creating a cartoon with a sprite sheet. The way this works is you start with the first frame in the animation. After a period of time you switch to the next frame and repeat the process until you've displayed all frames then go back to the first frame.

There is an array of Rectangles that will represent each frame of the animation. The next member variable, framesPerSecond, determines how many frames will be displayed each second and determines if the animation is slow or fast. There are then two TimeSpan member variables. The first holds how long to display each frame and the second holds how much time has passed since the last frame change. There are then three integer fields that represent the current frame displayed in the animation, the width of the frames and the height of the frames respectively.

I have included a property to expose the framesPerSecond member variable. The getter just returns the number of frames. The setter though does some validation. The number of frames per second should not be less than 1 so if a value less than 1 gets passed in I instead set it to 1. Similarly, it is not often that you will have a sprite that has more than 60 frames so I cap that at 60 frames. Otherwise I set the framesPerSecond member variable to the value requested. Afterwards I set the frameLength member variable to be 1 divided by framesPerSecond. I cast that to a double so that a decimal value will be generated.

I then have a property that returns what the current rectangle for the animation is. There is also a

property for the current frame of the animation. The getter returns the frame while the setter clamps the value between 0 and the number of frames minus 1 because arrays are zero based. Next there are two get only properties for returning the width and height of the frames.

There is a public constructor that will be used for creating animations. It takes as parameters the number of frames, the width and height of each frame, and x offset and y offset. The last two are used in generating the rectangles for each frame.

Inside the constructor I create the array of rectangles first. Next I set the frameWidth and frameHeight member variables. Next is an array that creates the source rectangles. In this class I'm assuming that the animations are in a horizontal row. For this to work I need the first pixel in the row, which are the x offset and y offset values. Each new frame will have the same y offset but x will increase by the frame width for each frame. I then set FramesPerSecond to be 5, which is good for the sprites that I will be using. I then call a method Reset that resets the animation to use base values.

There is then a private constructor that takes as a parameter and Animation object. This constructor is used when we need a new animation object. This is because classes are reference values and when you assign one member to another member it is referencing this rather than creating a new copy. I also default the number of frames per second to 5.

The Update method as you will gather updates the animation. If takes as a parameter the current GameTime object from the game. This holds how much time has elapsed since the last update, or frame in the game. I increase the frameTime member variable with the elapsed time since the last frame. Next I check if frameTime is greater than or equal to the length each frame is displayed. If it is I reset the duration since the last frame back to 0 and move the current frame. I use a formula to do this using the modulus operator. Since this returns a number between 0 and the value minus 1 I add 1 to the current frame variable.

The Reset method just sets the currentFrame member variable to the first frame, 0. It then resets the elapsed time back to 0 as well.

I've included a method, Clone, that takes an existing Animation object and creates a copy of it. This was generated from the ICloneable interface so it is still in a region related to that. ICloneable is not supported in all flavours of the .NET Framework so I removed implementing the interface but kept the method.

What the Clone method does is create an new Animation object using the private constructor. It then sets the frameWidth and frameHeight member variables for the animation. Next it calls Reset to reset the remain members. Finally it returns an object that is a clone of the animation. It returns as object because that is the signature of the method from the ICloneable interface. If you are not implementing the interface you could return as Animation instead. I will leave that decision up to you.

The next class to add in will be an animated sprite class that uses the Animation class to determine which frame to draw during game play. Right click the TileEngine folder, select Add and then Class. Name this new class AnimatedSprite. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

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

namespace Avatars.TileEngine
{
    public enum AnimationKey
    {
        IdleLeft,
        IdleRight,
        IdleDown,
        IdleUp,
        WalkLeft,
        WalkRight,
        WalkDown,
        WalkUp,
        ThrowLeft,
        ThrowRight,
        DuckLeft,
        DuckRight,
        JumpLeft,
        JumpRight,
        Dieing,
    }

    public class AnimatedSprite
    {
        #region Field Region

        Dictionary<AnimationKey, Animation> animations;
        AnimationKey currentAnimation;
        bool isAnimating;

        Texture2D texture;
        public Vector2 Position;
        Vector2 velocity;
        float speed = 200.0f;

        #endregion

        #region Property Region

        public bool IsActive { get; set; }

        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, 400.0f); }
        }

        public Vector2 Velocity
        {
            get { return velocity; }
            set { velocity = value; }
        }

        #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 ResetAnimation()
        {
            animations[currentAnimation].Reset();
        }

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

        public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(
                texture,
                Position,
                animations[currentAnimation].CurrentFrameRect,
                Color.White);
        }

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

        #endregion
    }
}
```

I included an enumeration with a number of values associated with it for common animations that you will find. I won't be using them all in the tutorial series but I kept them in.

The first member variable is a Dictionary<AnimationKey, Animation> that holds the animations for the sprite. There is also a AnimationKey member that represents the current animation for the sprite. Next

is a member variable, isAnimating, the returns if the animation should be played or not. Next is the sprite sheet as a Texture2D. I decided there was no risk in assigning the sprite's position directly so I include its position as a public member variable. Next are the sprite's velocity and speed. They might sound redundant to you but they are not. Velocity is a normalized vector that holds the direction the sprite is travelling. Speed holds how far the sprite travels in that direction. You will see this in practice shortly.

There a number of properties to expose values to other classes. The first, IsActive, auto-implemented property that represents if the sprite is active or not. Next is CurrentAnimation that returns what animation is was last played. IsAnimating determines if the sprite is actually animationing or not. Width and Height are very important attributes of the sprite so I have properties that return the width and height of the sprite. The last two properties expose the speed and velocity of the sprite. I cap the speed of the sprite between 1 and 400. You might be thinking why 400. Our game plays in 1280 by 720 and moving an object 400 pixels a frame would hardly been seen on the screen. It is because to have the distance the sprite moves constant I take this value and multiply it by the elapsed time between frames. So, regardless if the game is playing at 30 frames per second or 1000 frames per second the sprite moves at the same rate each second.

There is just one constructor that takes a Texture2D which is the sprite sheet and a Dictionary<AnimationKey, Animation> which holds the animations that the sprite has associated with it. Inside the constructor I set the sprite sheet and create a new Dictionary<AnimationKey, Animation> for the animations. In a foreach loop I go over the keys in the dictionary that was passed in. I then add a copy of the animation to the dictionary with that key.

There are a few methods next. The first, ResetAnimation, just resets the current animation by calling the Reset method on the current animation. The Update method checks to see if the sprite is animating. If it is animation it calls the Update method of the animation. The Draw method draws the sprite at its position. This will be drawn relative to the camera for the tile map so you don't need to work about using the camera here. Finally is a method, LockToMap, that takes the size of the map as a point and keeps the sprite from moving off the map.

Before I get to the player component I want to add a little code to the Game1 class. This will include the animations that are going to be defined for the player. So, open the Game1 class and update it to the following.

```
using Avatars.Components;
using Avatars.GameStates;
using Avatars.StateManager;
using Avatars.TileEngine;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System.Collections.Generic;

namespace Avatars
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Dictionary<AnimationKey, Animation> playerAnimations = new Dictionary<AnimationKey,
Animation>();

        GameStateManager gameStateManager;
```

```
        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;
        IGamePlayState gamePlayState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

        public static Rectangle ScreenRectangle
        {
            get { return screenRectangle; }
        }

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        public IGamePlayState GamePlayState
        {
            get { return gamePlayState; }
        }

        public Dictionary<AnimationKey, Animation> PlayerAnimations
        {
            get { return playerAnimations; }
        }

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";

            screenRectangle = new Rectangle(0, 0, 1280, 720);

            graphics.PreferredBackBufferWidth = ScreenRectangle.Width;
            graphics.PreferredBackBufferHeight = ScreenRectangle.Height;

            gameStateManager = new GameStateManager(this);
            Components.Add(gameStateManager);

            this.IsMouseVisible = true;

            titleIntroState = new TitleIntroState(this);
            startMenuState = new MainMenuState(this);
            gamePlayState = new GamePlayState(this);

            gameStateManager.ChangeState((TitleIntroState)titleIntroState, PlayerIndex.One);
        }

        protected override void Initialize()
        {
            Components.Add(new Xin(this));

            Animation animation = new Animation(3, 32, 32, 0, 0);
            playerAnimations.Add(AnimationKey.WalkDown, animation);

            animation = new Animation(3, 32, 32, 0, 32);
```

```
            playerAnimations.Add(AnimationKey.WalkLeft, animation);

            animation = new Animation(3, 32, 32, 0, 64);
            playerAnimations.Add(AnimationKey.WalkRight, animation);

            animation = new Animation(3, 32, 32, 0, 96);
            playerAnimations.Add(AnimationKey.WalkUp, animation);


            base.Initialize();
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);

        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

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

The change here is I added a new member variable playerAnimations that defines the animations that are implemented for the player's sprite. I also added a read only property to expose the member variable. In the Initialize method I create the animations and add them to the dictionary. While we are at this point let's add the sprite sheets that I used to the solution.

First, download the sprite sheets from this location and extract them. Now, open the content manager. With the Content node selected click the Add New Folder icon in the toolbar. Name this new folder PlayerSprites. Now select the PlayerSprites folder and click the Add Existing Item button in the tool bar navigate to the sprite sheets that you just downloaded and add them to the folder. Before closing the content manager make sure that you build the content project.

Now to add in the component for the player. Right click the project in the solution explorer, select Add and then Class. Name this new class Player. Here is the code for the Player 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 Avatars.TileEngine;

namespace Avatars.PlayerComponents
{
    public class Player : DrawableGameComponent
    {
        #region Field Region

        private Game1 gameRef;
        private string name;
        private bool gender;
        private string mapName;
        private Point tile;
        private AnimatedSprite sprite;
        private Texture2D texture;
        private float speed = 180f;

        private Vector2 position;

        #endregion

        #region Property Region

        public Vector2 Position
        {
            get { return sprite.Position; }
            set { sprite.Position = value; }
        }

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

        public float Speed
        {
            get { return speed; }
            set { speed = value; }
        }

        #endregion

        #region Constructor Region

        private Player(Game game)
            : base(game)
        {
        }

        public Player(Game game, string name, bool gender, Texture2D texture)
            : base(game)
        {
            gameRef = (Game1)game;
            this.name = name;
            this.gender = gender;

            this.texture = texture;
            this.sprite = new AnimatedSprite(texture, gameRef.PlayerAnimations);
            this.sprite.CurrentAnimation = AnimationKey.WalkDown;
        }

        #endregion

        #region Method Region

        public void SavePlayer()
```

```
        {
        }

        public static Player Load(Game game)
        {
            Player player = new Player(game);

            return player;
        }

        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            base.LoadContent();
        }

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

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);

            sprite.Draw(gameTime, gameRef.SpriteBatch);
        }

        #endregion
    }
}
```

This class inherits from DrawableGameComponent so that it has an Initialize, LoadContent, Update and Draw method wired for us. For member variables there is a reference to the Game1 object so that we have access to properties from that class. Next is a string variable, name, for the name of the player. The gender member variable describes the player's gender. Male will be false and female will be true. The next member currently isn't used but will be in the future and is the name of the map the player is currently on. The next member, tile, is what tile the player is in. There are member variables for the player's sprite and texture for the sprite. I added a speed member and set it to 180, which seemed a good speed in my demo. There is also a position member variable. There are properties to expose the position, sprite and speed of the sprite.

There is a private constructor that requires a Game parameter that is required by the base class and just calls the base constructor. There is then a constructor that takes a Game, string, bool and Texture2D parameter. The represent the Game1 object, the name of the player, their selected gender and the texture for the sprite.

The constructor first sets the member variables to the values passed in. I then create an AnimatedSprite object using the Texture2D that was passed in and animations that we defined in the Game1 class. I then set the current animation to be the one for walking downward.

I included two methods above the ones that inheriting from DrawableGameComponent provides called SavePlayer and Load. SavePlayer will save the player so that we can load their progress when they return to the game. Load is a static method so that it can be called without needing and instance of the Player class already. Finally are the methods that we inherit from DrawableGameComponent. The

only one that does anything is Draw and it just draws the sprite.

The last thing that needs to be added before adding the player to the game play state is that I need to add a method to the camera class that will lock the camera to the player's sprite. Open the Camera class and the following method.

```
        public void LockToSprite(TileMap map, AnimatedSprite sprite, Rectangle viewport)
        {
            position.X = (sprite.Position.X + sprite.Width / 2)
                         - (viewport.Width / 2);
            position.Y = (sprite.Position.Y + sprite.Height / 2)
                         - (viewport.Height / 2);
            LockCamera(map, viewport);
        }
```

What is happening here is I'm setting the camera's position so that it is centered on the sprite. The map also will not start scrolling until the middle of the sprite is half way across the screen. Similarly when it gets to the right edge it will stop scrolling once the sprite is closer than half the width or height of the screen.

The last thing to do is update the game play state to add in the player component that we just created. To do that update the GamePlayState class to the following.

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

namespace Avatars.GameStates
{
    public interface IGamePlayState
    {
        void SetUpNewGame();
        void LoadExistingGame();
        void StartGame();
    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);
        TileMap map;
        Camera camera;
        Player player;

        public GamePlayState(Game game)
            : base(game)
        {
            game.Services.AddService(typeof(IGamePlayState), this);
        }

        public override void Initialize()
        {
```

```
            base.Initialize();
        }

        protected override void LoadContent()
        {
            Texture2D spriteSheet = content.Load<Texture2D>(@"PlayerSprites\maleplayer");
            player = new Player(GameRef, "Wesley", false, spriteSheet);
        }

        public override void Update(GameTime gameTime)
        {
            Vector2 motion = Vector2.Zero;

            if (Xin.KeyboardState.IsKeyDown(Keys.W) && Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
                motion.Y = -1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkLeft;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = -1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkRight;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
                motion.Y = 1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkLeft;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = 1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkRight;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W))
            {
                motion.Y = -1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkUp;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S))
            {
                motion.Y = 1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkDown;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkLeft;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                player.Sprite.CurrentAnimation = AnimationKey.WalkRight;
            }

            if (motion != Vector2.Zero)
            {
                motion.Normalize();
                motion *= (player.Speed * (float)gameTime.ElapsedGameTime.TotalSeconds);
```

```
                Vector2 newPosition = player.Sprite.Position + motion;

                player.Sprite.Position = newPosition;
                player.Sprite.IsAnimating = true;
                player.Sprite.LockToMap(new Point(map.WidthInPixels, map.HeightInPixels));
            }

            camera.LockToSprite(map, player.Sprite, Game1.ScreenRectangle);
            player.Sprite.Update(gameTime);

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);

            if (map != null && camera != null)
                map.Draw(gameTime, GameRef.SpriteBatch, camera);

            GameRef.SpriteBatch.Begin(
                SpriteSortMode.Deferred,
                BlendState.AlphaBlend,
                SamplerState.PointClamp,
                null,
                null,
                null,
                camera.Transformation);

            player.Sprite.Draw(gameTime, GameRef.SpriteBatch);

            GameRef.SpriteBatch.End();
        }

        public void SetUpNewGame()
        {
            Texture2D tiles = GameRef.Content.Load<Texture2D>(@"Tiles\tileset1");
            TileSet set = new TileSet(8, 8, 32, 32);
            set.Texture = tiles;

            TileLayer background = new TileLayer(200, 200);
            TileLayer edge = new TileLayer(200, 200);
            TileLayer building = new TileLayer(200, 200);
            TileLayer decor = new TileLayer(200, 200);

            map = new TileMap(set, background, edge, building, decor, "test-map");

            map.FillEdges();
            map.FillBuilding();
            map.FillDecoration();

            camera = new Camera();
        }

        public void LoadExistingGame()
        {
        }

        public void StartGame()
        {
        }
    }
}
```

The first change I made is that I added a using statement to bring the player component into scope in

this class. I then created a Player field to hold the player object. In the LoadContent method I loaded the sprite sheet for the player and create a new male player named Wesley.

The Update method is where most of the changes will occur. In each of the if statements that check to see which keys are depressed I update the animation for the player's sprite. Since I don't have diagonal animations I use WalkLeft for the left diagonals and WalkRight for the right diagonals. I find it “more realistic” than using the up or down animations. In the other cases I use the appropriate animation based on the direction.

In the if statement where I check for movement I multiply the motion vector by the Speed property of the player and the ElapsedGameTime as seconds. This will be a really low value because the average frame rate is 60 times per second (1 / 60) which is 0.016666666667. If frame was to take longer than that the sprite would be moved a little further instead.

The next step is that I assign a local variable newPosition to be the sprite's position plus the motion vector. I assign the sprite's position to this value. Why I did that is in the future we will be introducing collision detection between other objects so we need to make sure the position we are moving to is valid. If it is not valid I won't update the sprite's position to this value. Since the player is moving the sprite I set its IsAnimating property to true. Then I call LockToMap to make sure it does not go outside of the bounds of the map.

After all of those udpates I call the new LockToSprite method of the camera to lock it to the sprite's position. I also call the Update method of the player's sprite so that it will update, including the animation for the sprite.

In the Draw method after drawing the map I call the Begin method the same as drawing the map so that the sprite will be drawn relative to the camera's position. I then call the Draw method on the player's sprite.

If you build and run the game now when you get to the game play state you will see the sprite in the upper left hand corner of the screen. You can now use the WASD keys to move the sprite around the map and it will animate appropriately. It will also not go outside the bounds of the map and the screen moves as described.

I'm going to end the tutorial here because we covered a lot in this tutorial already. I'm not sure what we will implement in the next tutorial at this time. I'd like to try to implement a few more game play features before moving back to scaffolding/plumbing.

Please stay tuned for the next tutorial in this series. If you don't want to have to keep visiting the site to check for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news from Game Programming Adventures.

I wish you the best in your MonoGame Programming Adventures!
Jamie McMahon