A Summoner's Tale
Chapter 7
Sprites

Welcome to my Summoner's Tale tutorial series on creating a Pokémon-inspired game with MonoGame. I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The code should work on previous versions of MonoGame and Visual Studio. I plan on creating the editors on macOS and Windows. I'm unfamiliar with Linux, so a few projects may not be done for that platform.

The tutorials will make more sense if they are read in order. You can find the list of tutorials on my web blog, A Summoner's Tale page. In addition to the PDFs, I will make the code for each tutorial available on GitHub here: https://github.com/Synammon/summoners-tale. It will also be included on the page that links to the tutorials.

Enough on GUI, for a while at least. In this tutorial, I will cover adding sprites to the game. To begin, you will need tilees. As chance would have it, I found some tiles that match our needs. They are by ArMM1998 and are from https://opengameart.org. I've downloaded them and added them to my Google Drive. You can find them at this link. As chance would have it, I have sprites from my Eyes of the Dragon series. Unfortunately, they've been around so long that I've forgotten their source. You can download them from my Google Drive here.

After downloading and extracting them, you need to add them to the project. So, in the SummonersTale project, open the Content folder and double-click the Content.mcgb file to open the content pipeline tool. Open the Tiles folder. Select the existing tiles, right-click them and select Exclude from project. Now, right-click the Tiles folder, select Add and then Existing Item. Browse where you extracted the files and add the Overworld.png, cave.png, and Inner.png files. Right-click the Content node, select Add and then New Folder. Name the new folder CharacterSprites. Right-click the CharacterSprites folder, choose Add and then Existing Item. Navigate to the folder where you extracted the character sprites and add the character.png item. Save the project and close the tool.

So, I couldn't find sprites like the ones I use in my game Shadow Monsters game. For my game, each animation is in its own file. In this case, animations are all in the same file. Eventually, I will be creating an editor to define animations in the file. You've had enough of editors for a while, so I will define them in code.

The first step is to update the tiles. Next, open the GamePlayState and replace it with the following code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Psilibrary.TileEngine;
using System;
```

```
using System.Collections.Generic;
using System.Reflection.Metadata;
using System.Text;

namespace SummonersTale.StateManagement
{
    public interface IGamePlayState
    {
        GameState GameState { get; }
    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        private TileMap _tileMap;
        private Camera _camera;

        public GameState GameState => this;

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

        public override void Initialize()
        {
            Engine.Reset(new(0, 0, 1280, 720), 32, 32);
            _camera = new();

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

            TileSheet sheet = new(content.Load<Texture2D>(@"Tiles/Overworld"),
"test", new(40, 36, 16, 16));
            TileSet set = new(sheet);

            TileLayer ground = new(100, 100, 0, 0);
            TileLayer edge = new(100, 100, -1, -1);
            TileLayer building = new(100, 100, -1, -1);
            TileLayer decore = new(100, 100, -1, -1);

            for (int i = 0; i < 1000; i++)
            {
                edge.SetTile(random.Next(0, 100), random.Next(0, 100), 0,
random.Next(0, 64));
            }

            _tileMap = new(set, ground, edge, building, decore, "test");

            base.LoadContent();
        }

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

            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
```

```
            {
                motion.X = -1;
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
            {
                motion.X = 1;
            }

            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.W))
            {
                motion.Y = -1;
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.S))
            {
                motion.Y = 1;
            }

            if (motion != Vector2.Zero)
                motion.Normalize();

            _camera.Position += motion * 240 *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            if (_camera.Position.X < 0)
            {
                _camera.Position = new(0, _camera.Position.Y);
            }

            if (_camera.Position.X > _tileMap.WidthInPixels - 1280)
            {
                _camera.Position = new(_tileMap.WidthInPixels - 1280,
_camera.Position.Y);
            }

            if (_camera.Position.Y < 0)
            {
                _camera.Position = new(_camera.Position.X, 0);
            }

            if (_camera.Position.Y > _tileMap.HeightInPixels - 720)
            {
                _camera.Position = new(_camera.Position.X, _tileMap.HeightInPixels -
720);
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            _tileMap.Draw(gameTime, SpriteBatch, _camera, false);
        }
    }
}
```

The changes are in the LoadContent method. I now load the new Overworld tile set. It is a 16 by 16 width and height tile set that is a 40 by 36 tile image. So, I pass in those values when I create the tile set. Instead of adding the random tiles to the ground layer, I add them to the edge layer.

Now, I will add classes for sprites. The first will be a base sprite class from which all other sprite classes will inherit. You guessed it. It is because I want to use polymorphism to manage all sprites in a manager. Right-click the Psilibrary project in the Solution Explorer, select Add and then New Folder. Name this new folder SpriteClasses. Next, right-click the SpriteClasses folder in the Psilibrary project, select Add and then Class. Name this new class, Sprite. The code for that class follows next. If you're curious why I am adding these classes to Psilibary rather than SummonersTale, it is because sprites are related to tile maps, and tile maps need to be in al library project, not a shared project.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Psilibrary.TileEngine;
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace SummonersTale.SpriteClasses
{
    public enum Facing { Up, Down, Left, Right }

    public abstract class Sprite
    {
        protected float _speed;
        protected Vector2 _velocity;

        public Facing Facing { get; set; }
        public string Name { get; set; }
        public Vector2 Position { get; set; }
        public Vector2 Size { get; set; }
        public Point Tile { get; set; }

        public int Width { get; set; }
        public float Height { get; set; }

        public float Speed
        {
            get { return _speed; }
            set { _speed = MathHelper.Clamp(_speed, 1.0f, 400.0f); }
        }

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

        public Vector2 Center
        {
            get { return Position + new Vector2(Width / 2, Height / 2); }
```

```
        }

        public Vector2 Origin
        {
            get { return new Vector2(Width / 2, Height / 2); }
        }

        public Sprite()
        {

        }

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

There is an enumeration here, Facing. This defines the direction that the sprite is facing. The class has two protected fields: _speed and _velocity. The _speed field is how many pixels the sprite travels per second. The _velocity field defines what direction the sprite is travelling. There are some properties in the class. Facing is what direction the sprite is facing. Name is the name of the sprite. Position is the position of the sprite. Size is the size of the sprite. Tile is which tile the sprite is in. Width and Height are the width and height of the sprite. Speed exposes the speed of the sprite. Velocity exposed the velocity of the sprite. The Center is the center of the sprite on the map. Finally, Origin is the center of the sprite. There is a base constructor that I may expand in the future. Finally, there are two abstract methods, Update and Draw, that classes inheriting from this class must implement.

The animation type that I will be implementing is frame animation. I mean that animation is a series of frames, much like animation in a cartoon. The frames are displayed in rapid succession. You start with the first frame, render it, move on to the second, render that frame and continue until you get to the last frame. When you get to the last frame, you go back to the first frame.

Before I get to the animated sprite, I need a class for animations. Right-click the SpriteClasses folder, select Add and then Class. Name this new class Animation. This is the code for that class.

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

namespace SummonersTale.SpriteClasses
{
    public class Animation
    {
        #region Field Region

        readonly 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(this)
            {
                frameWidth = this.frameWidth,
                frameHeight = this.frameHeight
            };
            animationClone.Reset();

            return animationClone;
        }

        #endregion
    }
}
```

There are some fields in the class. The first is an array of rectangles which describe the frames for the animation. The framesPerSecond field is how many frames should be rendered. I've found between 5 and 8 to be the best values. There is a TimeSpan which defines the length of a frame and a TimeSpan field which times how long since the last frame change. There are also fields for the height and width of the frames.

There are properties to expose the framesPerSecond field. I probably could have gotten away with using the Clamp method of MathHelper instead of using an if statement in the set part. After setting the field, I calculate the frame length by using the FromSeconds method of the TimeSpan class. There is a method to return the current rectangle as well as the current frame. Setting the current frame does use the Clamp method to clamp it in the valid range. There are then properties to expose the frame width and frame height.

The constructor takes as parameters the frame count, width and height. It also takes the x offset and y offset. These are used for sprite sheets where there are multiple animations on the same image. This animation assumes that the animations are in rows, not columns. After initializing the fields, I create the source rectangles. The Y coordinate is the easy part. It is just the y offset. The X coordinate is the current frame times the frame width plus the X offset. The width and height are the width and height passed in. I initialize the frame rate to 5 and call the Reset method that resets an animation back to zero. There is a second constructor that is used to clone an animation.

The Update method is where the magic takes place. The first step is to increase the frame timer by the elapsed game time. Following that, you check to see if the frame timer is greater than the frame length. If it is, set the frame timer back to zero. Next, increase the frame timer by one and then take the remainder of that calculation. The has the frames ranging between zero and the number of frames due to the way remainders work. The Reset method just resets the current frame to zero and the frame time to zero as well.

Initially, this class implemented the IClonable interface, which is why there is a region called Interface Method Region. It is easier to make a clone of a class than to initialize one sometimes. This one creates a new instance of the Animation class using the second constructor. It then sets the frameWidth and frameHeight fields. Finally, it calls the Reset method to reset the animation.

The next class to implement is the class for animated sprites. Right-click the SpriteClasses folder in thPsilibrary project, select Add and then Class. Name this new class, AnimatedSprite. Here is the code for that class.

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

```
namespace SummonersTale.SpriteClasses
{
    public class AnimatedSprite : Sprite
    {
        #region Field Region

        readonly Dictionary<string, Animation> animations;
        string currentAnimation;
        bool isAnimating;
        readonly Texture2D texture;

        #endregion

        #region Property Region

        public bool IsActive { get; set; }

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

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

        #endregion

        #region Constructor Region

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

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

        #endregion

        #region Method Region

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

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

```
        public override void Draw(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(
                texture,
                new Rectangle(
                    (int)Position.X,
                    (int)Position.Y,
                    Engine.TileWidth,
                    Engine.TileHeight),
                animations[currentAnimation].CurrentFrameRect,
                Color.White);
        }
}
```

So, there are a few assumptions about animated sprites. One is that all of the animations are on a single image. The other is that all the frames for a single image are on the same row. There is a Dictionary with a key type of string and a value of Animation. There is a string for what the current animation is and a bool for if the sprite is animating or not. Next there is a Texture2D for the image of the sprite. In another tutorial, I will demonstrate how to work with multiple images. There are properties to expose the active animation, if the sprite is active or not, and if it is animating.

The constructor takes a Texture2D and Dictionary argument. It sets the texture field to the value passed in. Then, it creates a new Dictionary. In a foreach loop, I loop over all of the keys in the animation Dictionary passed in. For each animation, I create a clone of it and add it to the sprite's animations. There is a simple method that I will mention, ResetAnimation. What this method does is call the Reset method of the current animation.

Because this method inherits from Sprite, it must implement an Update and Draw method. All the Update method does is check to see if the sprite is animating and if it is, call the current animation's Update method.

The Draw method assumes that it is called between Begin and End of the sprite batch object. It uses the overload that takes a texture, a destination rectangle and a tint colour. For the destination rectangle, I cast the position to an integer and used the height and width of the engine for those values.

Let's create a sprite in the gameplay state and render it on the map. Update the GamePlayState to the following code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Psilibrary.TileEngine;
using SummonersTale.SpriteClasses;
using System;
using System.Collections.Generic;
using System.Reflection.Metadata;
using System.Text;
```

```
namespace SummonersTale.StateManagement
{
    public interface IGamePlayState
    {
        GameState GameState { get; }
    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        private TileMap _tileMap;
        private Camera _camera;
        private AnimatedSprite sprite;

        public GameState GameState => this;

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

        public override void Initialize()
        {
            Engine.Reset(new(0, 0, 1280, 720), 32, 32);
            _camera = new();

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

            TileSheet sheet = new(content.Load<Texture2D>(@"Tiles/Overworld"),
"test", new(40, 36, 16, 16));
            TileSet set = new(sheet);

            TileLayer ground = new(100, 100, 0, 0);
            TileLayer edge = new(100, 100, -1, -1);
            TileLayer building = new(100, 100, -1, -1);
            TileLayer decore = new(100, 100, -1, -1);

            for (int i = 0; i < 1000; i++)
            {
                edge.SetTile(random.Next(0, 100), random.Next(0, 100), 0,
random.Next(0, 64));
            }

            _tileMap = new(set, ground, edge, building, decore, "test");

            Texture2D texture =
content.Load<Texture2D>(@"CharacterSprites/femalepriest");

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

            Animation animation = new(3, 32, 32, 0, 0) { CurrentFrame = 0,
FramesPerSecond = 8 };
            animations.Add("walkdown", animation);

            animation = new(3, 32, 32, 0, 32) { CurrentFrame = 0, FramesPerSecond =
8 };
```

```
            animations.Add("walkleft", animation);

            animation = new(3, 32, 32, 0, 64) { CurrentFrame = 0, FramesPerSecond =
8 };
            animations.Add("walkright", animation);

            animation = new(3, 32, 32, 0, 96) { CurrentFrame = 0, FramesPerSecond =
8 };
            animations.Add("walkup", animation);

            sprite = new(texture, animations)
            {
                CurrentAnimation = "walkdown",
                IsActive = true,
                IsAnimating = true,
            };

            base.LoadContent();
        }

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

            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
            {
                motion.X = -1;
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
            {
                motion.X = 1;
            }

            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.W))
            {
                motion.Y = -1;
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.S))
            {
                motion.Y = 1;
            }

            if (motion != Vector2.Zero)
                motion.Normalize();

            _camera.Position += motion * 240 *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            if (_camera.Position.X < 0)
            {
                _camera.Position = new(0, _camera.Position.Y);
            }

            if (_camera.Position.X > _tileMap.WidthInPixels - 1280)
            {
                _camera.Position = new(_tileMap.WidthInPixels - 1280,
_camera.Position.Y);
            }
```

```
            if (_camera.Position.Y < 0)
            {
                _camera.Position = new(_camera.Position.X, 0);
            }

            if (_camera.Position.Y > _tileMap.HeightInPixels - 720)
            {
                _camera.Position = new(_camera.Position.X, _tileMap.HeightInPixels -
720);
            }

            sprite.Update(gameTime);

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            _tileMap.Draw(gameTime, SpriteBatch, _camera, false);

            SpriteBatch.Begin();

            sprite.Draw(SpriteBatch);
            SpriteBatch.End();
        }
    }
}
```

The first change is the addition of an AnimatesSprite field for the player's character. Next, in the LoadContent method, I load the female priest texture because we all know girls rule. After loading the texture, I create a dictionary to hold the animations. The first animation that I create is the walk-down animation. It is 3 frames that are 32 by 32. The X and Y offsets are both 0 because the animations are on the top row. The animation is added to the dictionary of animations. The next animation is walk-left. The first three parameters are the same. The X offset is still zero because the row is on the left side of the image. The Y offset is 32 because it is on the second row, which is 32 pixels high. The next row is walk-right. The only different parameter is the Y offset, which is 64. Finally, the Y offset of the last parameter is 96. All four animations have a CurrentFrame of 0 and a FramesPerSecond of 8. It may be a little fast. Play around until you find a value that works for you. After creating the animations, I create the sprite passing as parameters the texture and the animations. I also initialize the CurrentAnimation to walk-down, IsActive to true and IsAnimating to true.

In the Update method, I call the update method of the sprite so it will animate. In the Draw method I call the Begin method of the SpriteBatch object after drawing the map. I then call Draw method of the sprite. Finally, I call the End method to stop rendering.

Well, that is all good. It still doesn't do much. It would be nice if the sprite moved around. Thankfully, that is relatively easy to do. We pretty much just replace the camera with the player sprite. Replace the Update method of the GamePlayState with the following code.

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

            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
            {
                motion.X = -1;

                if (sprite.CurrentAnimation != "walkleft")
                {
                    sprite.CurrentAnimation = "walkleft";
                    sprite.ResetAnimation();
                }
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
            {
                motion.X = 1;

                if (sprite.CurrentAnimation != "walkright")
                {
                    sprite.CurrentAnimation = "walkright";
                    sprite.ResetAnimation();
                }
            }

            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.W))
            {
                motion.Y = -1;

                if (sprite.CurrentAnimation != "walkup")
                {
                    sprite.CurrentAnimation = "walkup";
                    sprite.ResetAnimation();
                }
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.S))
            {
                motion.Y = 1;

                if (sprite.CurrentAnimation != "walkdown")
                {
                    sprite.CurrentAnimation = "walkdown";
                    sprite.ResetAnimation();
                }
            }

            if (motion != Vector2.Zero)
                motion.Normalize();

            sprite.Position += motion * 160 *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            if (sprite.Position.X < 0)
            {
                sprite.Position = new(0, sprite.Position.Y);
            }
```

```
            if (sprite.Position.X > _tileMap.WidthInPixels - 1280)
            {
                sprite.Position = new(_tileMap.WidthInPixels - 1280,
_camera.Position.Y);
            }

            if (sprite.Position.Y < 0)
            {
                sprite.Position = new(sprite.Position.X, 0);
            }

            if (sprite.Position.Y > _tileMap.HeightInPixels - 720)
            {
                sprite.Position = new(sprite.Position.X, _tileMap.HeightInPixels -
720);
            }

            sprite.Update(gameTime);

            base.Update(gameTime);
        }
```

What changed? Well, when I check to see if a key is down if the current animation is not in that direction, I set the animation to that direction and reset the animation. Then when I change the position of the camera, I change the position of the sprite. I then locked the sprite to the map as I did with the camera. Wait a minute. The camera doesn't follow the sprite, and the sprite moves off the right and bottom edges of the screen. That can be fixed by locking the camera on the sprite and the sprite on the map. Let's tackle sprites on the map first. What you want to do is add a new method to the AnimatedSprite class to lock the sprite to the map. Add this method to the AnimatedSprite class.

```
        public bool LockToMap(Point mapSize, ref Vector2 motion)
        {
            Position = new(
                MathHelper.Clamp(Position.X, 0, mapSize.X - Width),
                MathHelper.Clamp(Position.Y, 0, mapSize.Y - Height));

            if (Position.X == 0 && motion.X < 0 || Position.Y == 0 && motion.Y < 0)
            {
                motion = Vector2.Zero;
                return false;
            }

            if (Position.X == mapSize.X - Width && motion.X > 0)
            {
                motion = Vector2.Zero;
                return false;
            }

            if (Position.Y == mapSize.Y - Width && motion.Y > 0)
            {
                motion = Vector2.Zero;
                return false;
            }

            return true;
```

```
        }
```

The method returns a bool if the desired movement is negated. It takes a point that measures the map in pixels. I use the MathHelper.Clamp method to map the X and Y position between zero and the width minus the width of the sprite and the height minus the height of the sprite. If you don't deduct the height or the width of the sprite, the sprite will walk off the map. If the X property of the Position is X and the sprite wants to move left, or the Y property of the Position is Y, and the sprite wants to move up, the motion is set to zero, and false is returned. If the X property of the position is the width of the map in pixels minus the width of the sprite and the desired direction, I do the same as before, reset motion to zero and return false. Similarly, for down, I compare the height of the map minus the height of the sprite and if the sprite wants to move down. If all conditions pass, I return true.

Next, I need to update the Camera class. Replace the Camera class with the following code.

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

namespace Psilibrary.TileEngine
{
    public class Camera
    {
        #region Field Region

        Vector2 position;
        float speed;

        #endregion

        #region Property Region

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

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

        public Matrix Transformation
        {
            get
            {
                return Matrix.CreateTranslation(new Vector3(-Position, 0f));
            }
```

```
        }

        #endregion

        #region Constructor Region

        public Camera()
        {
            speed = 4f;
        }

        public Camera(Vector2 position)
        {
            speed = 4f;
            Position = position;
        }

        #endregion

        public void LockToSprite(AnimatedSprite sprite, TileMap map)
        {
            position.X = (sprite.Position.X + sprite.Width / 2)
                - (1280 / 2);

            position.Y = (sprite.Position.Y + sprite.Height / 2)
                - (720 / 2);

            LockCamera(map);
        }


        public void LockCamera(TileMap map)
        {
            position.X = MathHelper.Clamp(position.X,
                0,
                map.WidthInPixels - 1280);

            position.Y = MathHelper.Clamp(position.Y,
                0,
                map.HeightInPixels - 720);
        }

    }
}
```

I added two new methods. The first is LockToSprite that takes as parameters an AnimatedSprite and a TileMap as parameters. It sets the X position of the camera to the X position of the sprite passed in plus half the width of the sprite divided by two minus half the width of the window. What this does is once the sprite has reached half the width of the window, starts scrolling the map. The Y property of the sprite is similar to the X property of the sprite. It just uses the Y property and the height of the sprite. I then call a method LockCamera. The LockCamera method keeps the camera from scrolling off the map. It uses MathHelper.Clamp to keep the X property of the position between zero and the map width in pixels minus the width of the window. Y is between zero and map height in pixels minus the height of the window.

That just leaves updating the Update method of the GamePlayState. Replace the cod of that method with the following code.

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

    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
    {
        motion.X = -1;

        if (sprite.CurrentAnimation != "walkleft")
        {
            sprite.CurrentAnimation = "walkleft";
            sprite.ResetAnimation();
        }
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
    {
        motion.X = 1;

        if (sprite.CurrentAnimation != "walkright")
        {
            sprite.CurrentAnimation = "walkright";
            sprite.ResetAnimation();
        }
    }

    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.W))
    {
        motion.Y = -1;

        if (sprite.CurrentAnimation != "walkup")
        {
            sprite.CurrentAnimation = "walkup";
            sprite.ResetAnimation();
        }
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.S))
    {
        motion.Y = 1;

        if (sprite.CurrentAnimation != "walkdown")
        {
            sprite.CurrentAnimation = "walkdown";
            sprite.ResetAnimation();
        }
    }

    if (motion != Vector2.Zero)
        motion.Normalize();

    if (!sprite.LockToMap(new(99 * Engine.TileWidth, 99 * Engine.TileHeight), ref
motion)) return;

    sprite.Position += motion * 160 * (float)gameTime.ElapsedGameTime.TotalSeconds;
```

```
    _camera.LockToSprite(sprite, _tileMap);

    sprite.Update(gameTime);

    base.Update(gameTime);
}
```

What is new here is that there is a call to LockToMap on the sprite class passing in a point that is the number of rows in the map times the number of tiles wide the map is minus one times the width of a tile and the number of columns in the map minus one times the height of a tile. If the movement was cancelled, I exit the method. I then call the LockToSprite method on the Camera class, passing in the sprite and the _tileMap fields.

The last change for this tutorial is to right-click on the SummonersTaleGame project and select Set as Startup Project. If you build and run now, you should be presented with a title screen that will dismiss itself after five seconds. Then the game window will appear with the map and sprite. You can move the sprite around the screen, and the map will scroll. It will start scrolling if the sprite is half way across or down the screen.

I'm not going to dive any further into this tutorial. I think I've fed you enough for one day. So, that is going to be it for this tutorial. I covered a lot, and I don't want to overload you. I encourage you to keep visiting my blog for the latest news on my tutorials.

Good luck with your game programming adventures.
Cynthia