Shadow Monsters – MonoGame Tutorial Series
Chapter 2
Player Character

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called Shadow Monsters. 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 blog: Shadow Monsters. The source code for each tutorial will be available as well. I will be using Visual Studio 2019 Community for the series. The code should compile on the 2013, 2015 and 2017 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 give credit to Cynthia McMahon and add a link to my site, https://mygameprogrammingadventures.blogspot,com. Screenshots of your project and/or a video of game play would be appreciated.

I also want to mention that I assume you have a basic understanding of C# and MonoGame. If you don't I recommend that you learn basic C# and work with MonoGame a little. Enough to know the basics of fields, properties, methods, classes and the MonoGame framework.

In this tutorial I will be adding a sprite for the player to control. To get started you will need some sprite sheets. You can download two public domain ones that I found on the internet using this link . Once you've downloaded them you will need to add them using the content pipeline tool. Double click the tool to open it. With the main node select choose Add New Folder. Name this new folder Sprites. Now select the Sprites folder and click the Add Existing Item button on the tool bar. Navigate to the sprite sheets and select them both.

I want to add a couple classes. Two are for animated sprites, three are for handling game state and one is for the player component itself. I will start with animated sprites. Right click the TileEngine folder in the solution explorer, select Add and then Class. Name this new class Animation. 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;

namespace ShadowMonsters.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
    }
}
```

There are a number of fields in this class. There is an array of rectangles that will hold the frames of an animation. When animating you loop over the frames one by one and draw them. Much like how cartoons were made. An image was drawn, then another slightly different and then another giving the illusion of movement. That is why this type of animation is called frame animation. The next field holds the number of frames to be drawn each second. The next two hold the length a frame will be drawn and the number of milliseconds until the next frame will be drawn. There is a field that holds the current frame that is being drawn as well as the height and width of each frame.

There are properties to expose their values and a few that allow for setting. Because we don't want the source rectangles to be overwritten that field is not exposed. FramesPerSecond is read and write. Since we want to make sure there is at least one animation we cap it there. Similarly, anything 60 frames a second or more is unrealistic. A lower number is probably a better idea but 60 is a good starting point. You'd probably want to adjust them limits in your own games. After capping the value I calculate the frameLength field. It is equal to one over the number of frames. So, at 60 you are drawing an image approximately once every sixteen milliseconds which is a little fast for the eye to see.

There are two constructors for this class. The first takes as parameters the number of frames for the animation, the width of each frame, the height of each frame, the X offset and the Y offset. The last two require some explanation. The sprites are store in sheets like below. X offset measures where each column begins. In this case it is all zeros because the sprite sheet is arranged in rows. The Y offset measures where each row begins. In this case it increments by 64 pixels. The Y offset for the first row is 0 and the second row is 64. The constructor then goes on to initialize values. The interesting code is creating the source rectangles. I loop for the number of frames. I create the source rectangles using the X offset plus the frame width times the current frame. It is because the sprite sheet is arranged in rows. Because it is in rows the Y offset is used by itself. If it wrapped to multiple rows or was arranged in columns the frame height would be used. There is also a private constructor that is used for cloning animations.

The Update method is where we decide when to switch frames and takes as a parameter a GameTime which is the measurement of the passage of time between calls to the Update method of the Game class. I update the frameTimer with the elapsed time. If it is greater than the frame length I reset it to zero and change the current frame. I use the modulos operator to cycle the current animation between zero and the number of frames minus one which is the same as the index of the array of source rectangles.

Next there is a short utility method reset which resets the animation back to the first frame. This is useful for when the sprite stops animating.

The last method is the Clone method. This is used to make a copy of the animation. This is useful because the sprites typically are laid out in the same format. Instead of calling the constructor you can just clone it with the same parameters. You could just call the constructor but I like being able to clone the animations. The method calls the private constructor passing in the current object. Next it sets the frame width and frame height fields. Finally it calls the Reset method to reset the animation.

Next up is adding an animated sprite. Right click the TileEngine folder in the Solution Explorer, 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 ShadowMonsters.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; }
}

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

#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,
        new Rectangle(
                    (int)Position.X,
                    (int)Position.Y,
                    Engine.TileWidth,
                    Engine.TileHeight),
                animations[currentAnimation].CurrentFrameRect,
                Color.White);
        }

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

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

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

            if (Position.Y == mapSize.Y - Height && motion.Y > 0)
                return Vector2.Zero;

            return motion;
        }

        #endregion
    }
}
```

The first thing is an enumeration that holds different animations. Really we are only interested in the walking animations. Yes, I know that is not how to spell dying but I prefer it for some strange reason.

Next up are some fields. There is a Dictionary<AnimationKey, Animation> that holds the animations. Next is what current animation is being played. Following that is a bool that says if the sprite is animating or not. There is also the texture for the sprite. I have a public field that is the position of the sprite on the mage. Finally there are velocity and speed fields. Velocity measures the direction the sprite is travelling and speed is how many pixels in that direction the sprite travels.

The properties expose the fields. There are also properties for the width and height of the sprite. I also included a property for the center of the sprite.
The constructor takes as parameters the texture of the sprite and a dictionary or animations. It sets the texture field and creates a new dictionary for the animations. I then loop over all of the animations and clone them.

There is a method that is used to reset the current animation. It just calls the reset method of the current animation. The Update method checks to see if the sprite is animating. If it is it calls the Update method of current animation. The Draw method just draws the sprite. It will be called between calls to Begin and End of the SpriteBatch.

LockToMap is the interesting method. As the name implies it keeps the sprite from going off the map. It takes as parameters a point the measures the size of the map, in pixels, and the motion the sprite is travelling. This will be used later on when I restrict the motion of the sprite to tiles. First, I clamp the position using MathHelper.Clamp between 0 and the height and width of the map minus the height and width of the sprite. Next there is an if statement that checks if the position is zero and if the motion the sprite is travelling in is left or up. If it is I return the zero vector. There are similar checks for the right side of the map and the sprite is moving right and the bottom of the map moving down. Finally, I return the original motion vector.

Before I cover adding a sprite to the game to control I want to add a utility class that handles input. Right click on the project in the Solution Explorer, select Add and then class. Name this new class Xin. Here is the code for that class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

namespace ShadowMonsters
{
    public enum MouseButtons
    {
        Left,
        Right,
        Center
    }

    public class Xin : GameComponent
    {
        private static KeyboardState currentKeyboardState = Keyboard.GetState();
        private static KeyboardState previousKeyboardState = Keyboard.GetState();

        private static MouseState currentMouseState = Mouse.GetState();
        private static MouseState previousMouseState = Mouse.GetState();

        public static MouseState MouseState
        {
            get { return currentMouseState; }
        }

        public static KeyboardState KeyboardState
        {
            get { return currentKeyboardState; }
        }

        public static KeyboardState PreviousKeyboardState
        {
            get { return previousKeyboardState; }
        }

        public static MouseState PreviousMouseState
        {
            get { return previousMouseState; }
        }

        public static Point MouseAsPoint { get; internal set; }

        public Xin(Game game)
            : base(game)
        {

        }
        public override void Update(GameTime gameTime)
        {
            Xin.previousKeyboardState = Xin.currentKeyboardState;
            Xin.currentKeyboardState = Keyboard.GetState();

            Xin.previousMouseState = Xin.currentMouseState;
            Xin.currentMouseState = Mouse.GetState();

            MouseAsPoint = new Point(currentMouseState.X, currentMouseState.Y);

            base.Update(gameTime);
        }

        public static void FlushInput()
        {
            currentMouseState = previousMouseState;
            currentKeyboardState = previousKeyboardState;
        }

        public static bool CheckKeyReleased(Keys key)
        {
            return currentKeyboardState.IsKeyUp(key) && previousKeyboardState.IsKeyDown(key);
        }

        public static bool CheckMouseReleased(MouseButtons button)
        {
            switch (button)
            {
                case MouseButtons.Left:
                    return (currentMouseState.LeftButton == ButtonState.Released) &&
(previousMouseState.LeftButton == ButtonState.Pressed);
                case MouseButtons.Right:
                    return (currentMouseState.RightButton == ButtonState.Released) &&
(previousMouseState.RightButton == ButtonState.Pressed);
                case MouseButtons.Center:
                    return (currentMouseState.MiddleButton == ButtonState.Released) &&
(previousMouseState.MiddleButton == ButtonState.Pressed);
            }

            return false;
        }
    }
}
```

I included an enumeration that defines the mouse buttons. I did this to make mouse input more like keyboard and game pad input. The class inherits from GameComponent. This allows it to be added to a list of components in the game and have the Update method called automatically by MonoGame. If it was a DrawableGameComponent then the Draw method would be called automatically as well as the Update method. We will be making more use of game components as the tutorial series continues.

There are static fields that hold the current state of the keyboard and the state of the keyboard in the previous frame of the game. The same is true for the mouse. The fields are static so that they can be referenced without having an instance of the class. They will be referenced as Xin.KeyboardState rather than having an instance of Xin and using that. There are read-only properties that expose these four fields. There is also an auto-property that exposes the position of the mouse cursor as a point.

GameComponents and DrawableGameComponents take a Game parameter so I had to add an empty constructor that just calls the base constructor.

In the Update method is where the magic takes place. First, I set the value of the previous keyboard state to the current keyboard field. Next I update the current keyboard field to be the current state of the keyboard. I do the same for the mouse. I also create a Point that represents the mouse cursor position.

There is a static method, FlushInput, that sets the previous mouse and keyboard to the current mouse and keyboard state. This will become more clear in a moment.

The next method is CheckKeyReleased. What it does is compare if a key that was down is now up. This is why FlushInput is important. Without it a key that was released could still be seen as released if we switch game components. I also check for released instead of pressed for similar reasons. If I was working on a real time game that I would do key pressed instead of key released.

The CheckMouseReleased method works the same way. The difference is the way the mouse works compared to the keyboard and why I created the enumeration. There is a switch on the button to be checked. I then check if the mouse button was released since the last frame of the game.

Time to change gears and work a bit on content before going much further. What I want to do is download some sprites by Antifareas at https://opengameart.org/content/antifareas-rpg-sprite-set-1-enlarged-w-transparent-background. Download and extract the sprites. Open the Content Pipeline Tool by expanding the Content folder and double clicking the Content.mgcb entry. Create a new folder called Sprites. To this folder add the mage_f.png and mage_m.png images. Save the project by pressing CTRL+S and then build by pressing F6. You can close the Content Pipeline Tool.

The next change is to the Game1 class. I made substantial changes to the Game1 class so I'm going to give you the code for the entire class.

```
using Avatars.TileEngine;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using ShadowMonsters.TileEngine;
using System;
using System.Collections.Generic;

namespace ShadowMonsters
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    public class Game1 : Game
    {
        public static Random Random = new Random();
        private readonly GraphicsDeviceManager graphics;
        private SpriteBatch spriteBatch;
        private readonly Engine engine = new Engine(new Rectangle(0, 0, 1280, 720));
        private TileMap map;
        private Dictionary<AnimationKey, Animation> animations = new Dictionary<AnimationKey, 
Animation>();
        private AnimatedSprite sprite;
        private Vector2 motion;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this)
            {
                PreferredBackBufferWidth = 1280,
                PreferredBackBufferHeight = 720
            };

            graphics.ApplyChanges();

            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here
            Components.Add(new Xin(this));
            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
            TileSet set = new TileSet();
            set.TextureNames.Add("tileset1");
            set.Textures.Add(Content.Load<Texture2D>(@"Tiles\tileset16-outdoors"));

            TileLayer groundLayer = new TileLayer(100, 100, 0, 1);
            TileLayer edgeLayer = new TileLayer(100, 100);
            TileLayer buildingLayer = new TileLayer(100, 100);
            TileLayer decorationLayer = new TileLayer(100, 100);

            for (int i = 0; i < 1000; i++)
            {
                decorationLayer.SetTile(Random.Next(0, 100), Random.Next(0, 100), 0, 
Random.Next(2, 4));
            }

            map = new TileMap(set, groundLayer, edgeLayer, buildingLayer, decorationLayer,
"level1");

            engine.SetMap(map);

            Animation animation = new Animation(3, 32, 36, 0, 0);
            animations.Add(AnimationKey.WalkUp, animation);

            animation = new Animation(3, 32, 36, 0, 36);
            animations.Add(AnimationKey.WalkRight, animation);

            animation = new Animation(3, 32, 36, 0, 72);
            animations.Add(AnimationKey.WalkDown, animation);

            animation = new Animation(3, 32, 36, 0, 108);
            animations.Add(AnimationKey.WalkLeft, animation);

            sprite = new AnimatedSprite(Content.Load<Texture2D>(@"Sprites\mage_f"), animations)
            {
                CurrentAnimation = AnimationKey.WalkDown
            };
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// game-specific content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            // TODO: Add your update logic here
            engine.Update(gameTime);
            motion = Vector2.Zero;

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

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

                Vector2 newPosition = sprite.Position + motion;
                newPosition.X = (int)newPosition.X;
                newPosition.Y = (int)newPosition.Y;

                sprite.Position = newPosition;
                motion = sprite.LockToMap(
                    new Point(
                        map.WidthInPixels,
                        map.HeightInPixels),
                    motion);
            }

            Engine.Camera.LockToSprite(map, sprite, new Rectangle(0, 0, 1280, 720));
            sprite.Update(gameTime);

            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
            engine.Draw(gameTime, spriteBatch);
            spriteBatch.Begin(
                SpriteSortMode.Deferred,
                BlendState.AlphaBlend,
                SamplerState.PointClamp,
                null,
                null,
                null,
                Engine.Camera.Transformation);
            sprite.Draw(gameTime, spriteBatch);
            spriteBatch.End();
        }
    }
}
```

I added a using statement for System.Collections.Generic to bring the Dictionary class into scope for this file. I then added in three new fields, a Dictionary to hold the animations for the sprite, an AnimatedSprite and a Vector2 that describes the motion of the sprite.

In the initialize method I create a new Xin component and add it to the list of components. This has the component update automatically so we can forget about it. In the LoadContent method I create the animations. The sprites are 32 pixels by 36 pixels and 3 sprites per row. The animations are in the order Up, Right, Down and Left. Once the animations are created I create the sprite.

In the Update method I set the motion vector to be the zero vector. I use Xin to check if the W key is pressed. If it is I set the Y value of motion to up, -1, the animation to the up animation and that the sprite is animating. If W is not pressed I check for the S key. I set the Y value of motion to be 1, or down, the animation to down and set the IsAnimating property to true. I do similar things for the A and D keys but with the X value. You will notice that I used elses when checking for direction to move. This means the sprite will only travel in cardinal directions and there is an order of precedence: up, down, left then right. If none of those keys are down I set the IsAnimating property to false.

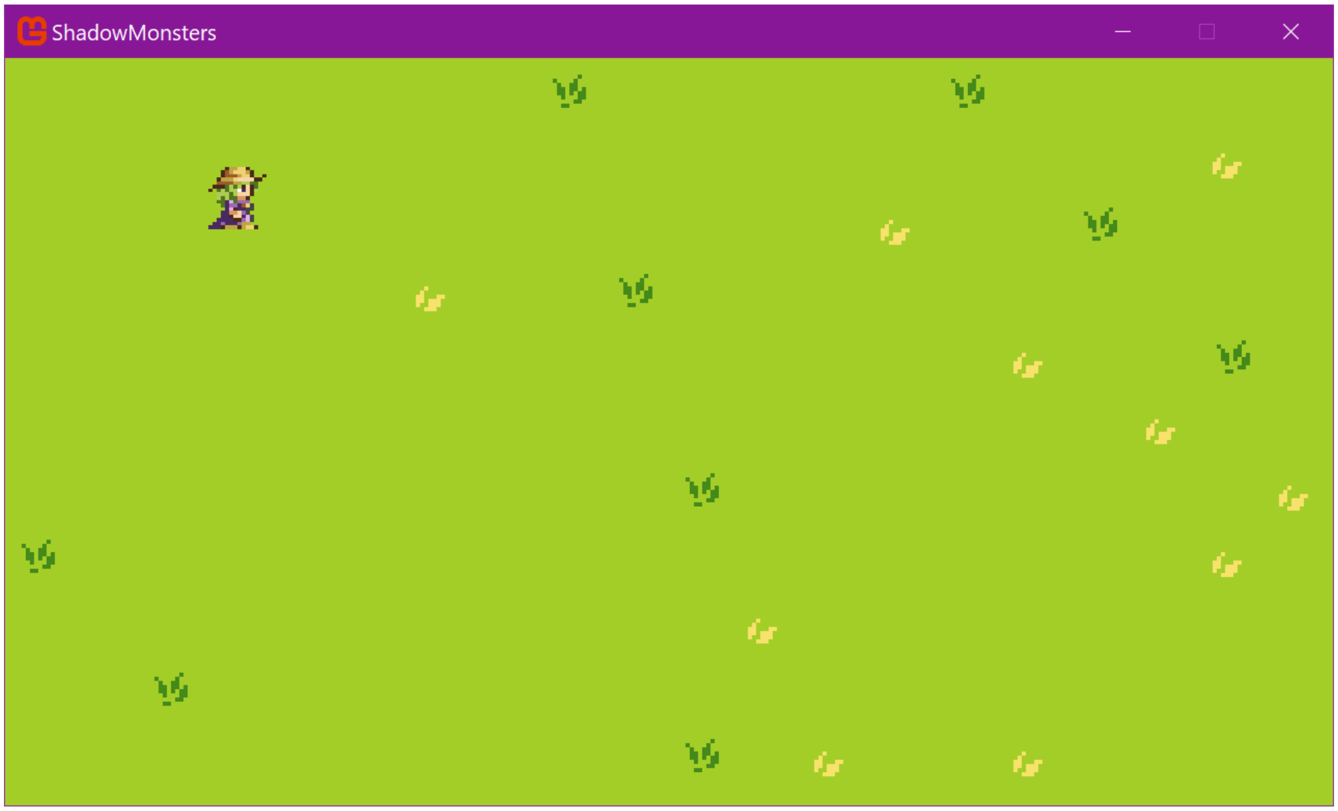
I then check to see if the sprite is moving by comparing it with the zero vector. If it is moving I normalize the vector. This is actually unnecessary because the motion vector will only ever have a length of one but I include it in case you decide you don't want to restrict movement to one direction at a time. Next I calculate the number of pixels to move the sprite using the speed and the elapsed game time since the last frame. Now I calculate the new position of the sprite based on the motion. I round it down to whole pixels rather than floating point pixels. Now I call the LockToMap method passing in the width and height of the map with the motion vector. I call a method I wrote on the Camera class that we will get to shortly that has the camera follow the sprite. Finally I call the Update method of the sprite.
In the Draw method I call the Begin method of SpriteBatch the same was as I did in the last tutorial using the transformation matrix. Now I call the Draw method of the sprite then End to stop drawing.

That leaves the LockToSprite method in the Camera class. Add the following code to the Method Region of the Camera class.

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

What this does is snap the camera to the position of the sprite minus half the height and width of the sprite. This has the camera scroll when the sprite gets to half way across or down the screen and then stop at the edges. It then calls the LockCamera method to make sure the camera does not go off the screen.

If you build and run the game now you should see something similar to the following. You can move the sprite around the screen using the W, A, S and D keys.



I will be starting work on the next tutorial shortly. Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so.

I wish you the best in your MonoGame Programming Adventures!