A Summoner's Tale – MonoGame Tutorial Series

Chapter 7

Characters

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 is about adding in characters for the player to interact with. So, let's get started. First, right click the Avatars project, select Add and then New Folder. Name this new folder CharacterComponents. Now right click the CharacterComponents folder, select Add and then New Item. From the list of items choose Interface. Name this new interface ICharacter. Here is the code for that interface.

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

namespace Avatars.CharacterComponents
{
    public interface ICharacter
    {
        string Name { get; }
        AnimatedSprite Sprite { get; }
        Avatar BattleAvatar { get; }
        Avatar GiveAvatar { get; }
        void SetConversation(string newConversation);
        void Update(GameTime gameTime);
        void Draw(GameTime gameTime, SpriteBatch spriteBatch);
    }
}
```

There are some using statements to bring components for MonoGame, the tile engine and avatars into scope. In the actual interface there are readonly properties for the name of the character, their

sprite, the avatar that they are currently battling with and an avatar that they will give to the player in certain conditions. I also added in a method that will set the activate conversation for the character. I also included an Update method that will be called to update the character and a draw method to draw the character.

Now I'm going to add the class for the character. Right click the CharacterComponents folder, select Add and then Class. Name this new class Character. Here is the code for the character class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Avatars.AvatarComponents;
using Avatars.TileEngine;

namespace Avatars.CharacterComponents
{
    public class Character : ICharacter
    {
        #region Constant

        public const float SpeakingRadius = 40f;

        #endregion

        #region Field Region

        private string name;
        private Avatar battleAvatar;
        private Avatar givingAvatar;
        private AnimatedSprite sprite;

        private string conversation;

        private static Game1 gameRef;
        private static Dictionary<AnimationKey, Animation> characterAnimations = new
Dictionary<AnimationKey, Animation>();

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

        public Avatar BattleAvatar
        {
            get { return battleAvatar; }
        }

        public Avatar GiveAvatar
        {
            get { return givingAvatar; }
        }
```

```
        public string Conversation
        {
            get { return conversation; }
        }

        #endregion

        #region Constructor Region

        private Character()
        {
        }

        #endregion

        #region Method Region

        private static void BuildAnimations()
        {
        }

        public static Character FromString(Game game, string characterString)
        {
            if (gameRef == null)
                gameRef = (Game1)game;

            if (characterAnimations.Count == 0)
                BuildAnimations();

            Character character = new Character();
            string[] parts = characterString.Split(',');

            character.name = parts[0];
            Texture2D texture = game.Content.Load<Texture2D>(@"CharacterSprites\" +
parts[1]);
            character.sprite = new AnimatedSprite(texture, gameRef.PlayerAnimations);

            AnimationKey key = AnimationKey.WalkDown;
            Enum.TryParse<AnimationKey>(parts[2], true, out key);

            character.sprite.CurrentAnimation = key;

            character.conversation = parts[3];

            return character;
        }

        public void SetConversation(string newConversation)
        {
            this.conversation = newConversation;
        }

        public static void Save(string characterName)
        {

        }

        public void Update(GameTime gameTime)
        {
            sprite.Update(gameTime);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            sprite.Draw(gameTime, spriteBatch);
        }
```

```
        #endregion
    }
}
```

As always there are using statements to bring classes in other namespaces into scope. The class itself implements the ICharacter interface that was defined earlier. I included a constant in this class, SpeakingRadius. This defines how close to the player and the character have to be in order to have a conversation. That will implemented in a future tutorial.

I included a few member variables in this class. The first four are for implementing the ICharacter interface. The next, conversation, represents what the current conversation is for the character. Next are two static fields. The first represents the Game1 class and the second is a dictionary for the animations for the character's sprite.

Next there are properties that implement the properties from ICharacter. They are all get only, or readonly depending on who you are speaking with. The last property, Conversation, will expose the current conversation the character and player are in.

Next up is a private constructor that takes no parameters. That is also no public constructor that can be used to create instances of the Character class. That will done using a static method that is up soon. I did this because I was using CSV files for storing characters rather than creating an editor and using the IntermediateSerializer to save content.

BuildAnimations is a method that will need to be expanded to create the animations for the character's sprite. I included it because it was used in my demo. Next is the FromString method that takes a Game parameter and a string parameter. Game is used for loading content and getting the animations that we created for the player's sprite. That is because currently the sprites that I will be using have the same layout and animations. It is entirely possible that you can have different animations and why I included the other method.

What the method does is first check to see if the static member field is set or not. If it is not set I set it. Similarly I check to see if there are animations for the sprite. If there are no animations I call BuildAnimations that would build those animations.

I then create an instance of the Character class using the private constructor. I then split the string on the comma. You can use other separators by using it in your file and updating the code.

The first part of the string is the character's name so I set that field to that array element. Next up is the sprite sheet for character. I then create the sprite using the texture and the animations from the Game1 class. The next part represents which animation to draw the sprite with. It should be down so I set that animation to walk down. I then try and parse the string value and get the AnimationKey from the string. The last part to get is the current conversation associated with the character. This method will be fleshed out more when I start with adding in avatars for the players.

The last methods implement the ICharacter interface methods. SetConverstion sets the current conversation for the character. Next the Update method calls the update method of the sprite and the draw method calls the draw method of the sprite.

In this class the character only has one avatar but in Pokemon the characters can have up to six Pokemon. How could you implement that in this class? Let me add a second class and I will develop

through the tutorial at the same time. Right click the CharacterComponents folder, select Add and then Class. Name this new class PCharacter. 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;
using Avatars.AvatarComponents;
using Avatars.TileEngine;

namespace Avatars.CharacterComponents
{
    public class Pcharacter : ICharacter
    {
        #region Constant

        public const float SpeakingRadius = 40f;
        public const int AvatarLimit = 6;

        #endregion

        #region Field Region

        private string name;
        private Avatar[] avatars = new Avatar[AvatarLimit];
        private int currentAvatar;
        private Avatar givingAvatar;
        private AnimatedSprite sprite;

        private string conversation;

        private static Game1 gameRef;
        private static Dictionary<AnimationKey, Animation> characterAnimations = new
Dictionary<AnimationKey, Animation>();

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

        public Avatar BattleAvatar
        {
            get { return avatars[currentAvatar]; }
        }

        public Avatar GiveAvatar
        {
            get { return givingAvatar; }
        }

        public string Conversation
        {
            get { return conversation; }
```

```
        }

        #endregion

        #region Constructor Region

        private PCharacter()
        {
        }

        #endregion

        #region Method Region

        private static void BuildAnimations()
        {
        }

        public static PCharacter FromString(Game game, string characterString)
        {
            if (gameRef == null)
                gameRef = (Game1)game;

            if (characterAnimations.Count == 0)
                BuildAnimations();

            PCharacter character = new PCharacter();
            string[] parts = characterString.Split(',');

            character.name = parts[0];
            Texture2D texture = game.Content.Load<Texture2D>(@"CharacterSprites\" +
parts[1]);
            character.sprite = new AnimatedSprite(texture, gameRef.PlayerAnimations);

            AnimationKey key = AnimationKey.WalkDown;
            Enum.TryParse<AnimationKey>(parts[2], true, out key);

            character.sprite.CurrentAnimation = key;

            character.conversation = parts[3];

            return character;
        }

        public void ChangeAvatar(int index)
        {
            if (index < 0 || index >= AvatarLimit)
            {
                currentAvatar = index;
            }
        }

        public void SetConversation(string newConversation)
        {
            this.conversation = newConversation;
        }

        public static void Save(string characterName)
        {

        }

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

```
        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            sprite.Draw(gameTime, spriteBatch);
        }

        #endregion
    }
}
```

What I did was add another constant, AvatarLimit, which is the maximum number of avatars a character can have at one time. I then added an array of Avatar objects with a length of AvatarLimit. I replaced the battleAvatar field with an integer field currentAvatar. For the BattleAvatar property I return the avatar at the currentAvatar index. I also added a method ChangeAvatar that would be used to switch the current avatar for another avatar. I would add that method to the ICharacter interface.

Let's add a class to manage the characters in the game like the other manager classes. Right click the CharacterComponents folder, select Add and then Class. Name this new class CharacterManager. Here is the code for that class.

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

namespace Avatars.CharacterComponents
{
    public sealed class CharacterManager
    {
        private static readonly CharacterManager instance = new CharacterManager();

        private Dictionary<string, ICharacter> characters = new Dictionary<string,
ICharacter>();

        public static CharacterManager Instance
        {
            get { return instance; }
        }

        private CharacterManager()
        {
        }

        public ICharacter GetCharacter(string name)
        {
            if (characters.ContainsKey(name))
                return characters[name];

            return null;
        }

        public void AddCharacter(string name, ICharacter character)
        {
            if (!characters.ContainsKey(name))
            {
                characters.Add(name, character);
            }
        }
    }
}
```

This is another singleton class because we only want one in the entire game. For that reason the class is marked as sealed. There is a member variable for the instance of the singleton and a dictionary that has a string as the key and an ICharacter as the value. Since I used ICharacter it is possible to add both Character and PCharacter objects to this dictionary. Instead of exposing the entire dictionary to external classes a provided methods to get a character and a method to add a character. Both methods do some simple validation before returning or adding a character.

Lets implement this into the game now. First, we will want a couple of images for characters. I've provided a few sample sprites at this link for this purpose. I also resized the player sprite sheets so that the sprites are 64x64 instead of 32x32.

Character Sprites
http://gameprogrammingadventures.org/monogame/downloads/CharacterSprites.zip

Now lets add these to the game. First, replace the player sprites in the project with the new sprites that you just downloaded. Open the MonoGame content manager so that we can add the new character sprites. First, select the Content node and click the Add New Folder button in the toolbar. Name this new folder CharacterSprites. Select the CharacterSprites folder and then click the Add Existing Item button. Navigate to the teacherone and teachertwo sprite sheets to add them to the folder. When prompted copy them to this folder. Save the project and rebuild, not just build, before closing the manager.

The next thing to do is update the make game class, Game1. What I want to do is add a field for the character manager to the class and a readonly property to expose the character manager. In the constructor I will get the instance. I also updated the Initialize method to adapt to the new sprite size. I replaced the 32 width and heights with 64. I also had to update the Y offset variables to accommodate the new size as well. Update the Game1 class to the following.

```
using Avatars.CharacterComponents;
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;
        CharacterManager characterManager;

        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 CharacterManager CharacterManager
        {
            get { return characterManager; }
        }

        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);

            characterManager = CharacterManager.Instance;
        }

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

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

            animation = new Animation(3, 64, 64, 0, 64);
            playerAnimations.Add(AnimationKey.WalkLeft, animation);
```

```
            animation = new Animation(3, 64, 64, 0, 128);
            playerAnimations.Add(AnimationKey.WalkRight, animation);

            animation = new Animation(3, 64, 64, 0, 192);
            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 next thing that I'm going to tackle is creating a few characters and get them drawing on the map. That will be done in the SetUpNewGame method. Update that code to the following.

```
        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();

            ICharacter teacherOne = Character.FromString(GameRef,
"Lance,teacherone,WalkDown,teacherone");
            ICharacter teacherTwo = PCharacter.FromString(GameRef,
"Marissa,teachertwo,WalkDown,tearchertwo");
```

```
            GameRef.CharacterManager.AddCharacter("teacherone", teacherOne);
            GameRef.CharacterManager.AddCharacter("teachertwo", teacherTwo);

            map.Characters.Add("teacherone", new Point(0, 4));
            map.Characters.Add("teachertwo", new Point(4, 0));

            camera = new Camera();
        }
```

The code creates a new Character and PCharacter using the respective FromString methods and assigns them to an ICharacter variable. Next I add them to the character manager so they are stored centrally. Finally I add them to the map.

The next step is to draw the characters. To do that we need to update the TileMap class. What I did was add a member variable for the character manager and get the instance in the constructor. I also added in a new method DrawCharacters that loops through all of the characters that were added to the map and draw them. I will go over DrawCharacters a bit more after you've seen the code. Update the TileMap class as follows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Avatars.CharacterComponents;

namespace Avatars.TileEngine
{
    public class TileMap
    {
        #region Field Region

        string mapName;
        TileLayer groundLayer;
        TileLayer edgeLayer;
        TileLayer buildingLayer;
        TileLayer decorationLayer;
        Dictionary<string, Point> characters;
        CharacterManager characterManager;

        [ContentSerializer]
        int mapWidth;

        [ContentSerializer]
        int mapHeight;

        TileSet tileSet;

        #endregion

        #region Property Region

        [ContentSerializer]
        public string MapName
        {
            get { return mapName; }
            private set { mapName = value; }
        }

        [ContentSerializer]
        public TileSet TileSet
```

```
        {
            get { return tileSet; }
            set { tileSet = value; }
        }

        [ContentSerializer]
        public TileLayer GroundLayer
        {
            get { return groundLayer; }
            set { groundLayer = value; }
        }

        [ContentSerializer]
        public TileLayer EdgeLayer
        {
            get { return edgeLayer; }
            set { edgeLayer = value; }
        }

        [ContentSerializer]
        public TileLayer BuildingLayer
        {
            get { return buildingLayer; }
            set { buildingLayer = value; }
        }

        [ContentSerializer]
        public Dictionary<string, Point> Characters
        {
            get { return characters; }
            private set { characters = value; }
        }

        public int MapWidth
        {
            get { return mapWidth; }
        }

        public int MapHeight
        {
            get { return mapHeight; }
        }

        public int WidthInPixels
        {
            get { return mapWidth * Engine.TileWidth; }
        }

        public int HeightInPixels
        {
            get { return mapHeight * Engine.TileHeight; }
        }

        #endregion

        #region Constructor Region

        private TileMap()
        {
        }

        private TileMap(TileSet tileSet, string mapName)
        {
            this.characters = new Dictionary<string, Point>();
            this.tileSet = tileSet;
            this.mapName = mapName;
```

```
            characterManager = CharacterManager.Instance;
        }

        public TileMap(
            TileSet tileSet,
            TileLayer groundLayer,
            TileLayer edgeLayer,
            TileLayer buildingLayer,
            TileLayer decorationLayer,
            string mapName)
            : this(tileSet, mapName)
        {
            this.groundLayer = groundLayer;
            this.edgeLayer = edgeLayer;
            this.buildingLayer = buildingLayer;
            this.decorationLayer = decorationLayer;

            mapWidth = groundLayer.Width;
            mapHeight = groundLayer.Height;
        }

        #endregion

        #region Method Region

        public void SetGroundTile(int x, int y, int index)
        {
            groundLayer.SetTile(x, y, index);
        }

        public int GetGroundTile(int x, int y)
        {
            return groundLayer.GetTile(x, y);
        }

        public void SetEdgeTile(int x, int y, int index)
        {
            edgeLayer.SetTile(x, y, index);
        }

        public int GetEdgeTile(int x, int y)
        {
            return edgeLayer.GetTile(x, y);
        }

        public void SetBuildingTile(int x, int y, int index)
        {
            buildingLayer.SetTile(x, y, index);
        }

        public int GetBuildingTile(int x, int y)
        {
            return buildingLayer.GetTile(x, y);
        }

        public void SetDecorationTile(int x, int y, int index)
        {
            decorationLayer.SetTile(x, y, index);
        }

        public int GetDecorationTile(int x, int y)
        {
            return decorationLayer.GetTile(x, y);
        }

        public void FillEdges()
```

```
        {
            for (int y = 0; y < mapHeight; y++)
            {
                for (int x = 0; x < mapWidth; x++)
                {
                    edgeLayer.SetTile(x, y, -1);
                }
            }
        }

        public void FillBuilding()
        {
            for (int y = 0; y < mapHeight; y++)
            {
                for (int x = 0; x < mapWidth; x++)
                {
                    buildingLayer.SetTile(x, y, -1);
                }
            }
        }

        public void FillDecoration()
        {
            for (int y = 0; y < mapHeight; y++)
            {
                for (int x = 0; x < mapWidth; x++)
                {
                    decorationLayer.SetTile(x, y, -1);
                }
            }
        }

        public void Update(GameTime gameTime)
        {
            if (groundLayer != null)
                groundLayer.Update(gameTime);

            if (edgeLayer != null)
                edgeLayer.Update(gameTime);

            if (buildingLayer != null)
                buildingLayer.Update(gameTime);

            if (decorationLayer != null)
                decorationLayer.Update(gameTime);

        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
        {
            if (groundLayer != null)
                groundLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            if (edgeLayer != null)
                edgeLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            if (buildingLayer != null)
                buildingLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            if (decorationLayer != null)
                decorationLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            DrawCharacters(gameTime, spriteBatch, camera);
        }

        public void DrawCharacters(GameTime gameTime, SpriteBatch spriteBatch, Camera
```

```
camera)
        {
            spriteBatch.Begin(
                SpriteSortMode.Deferred,
                BlendState.AlphaBlend,
                SamplerState.PointClamp,
                null,
                null,
                null,
                camera.Transformation);

            foreach (string s in characters.Keys)
            {
                ICharacter c = CharacterManager.Instance.GetCharacter(s);

                if (c != null)
                {
                    c.Sprite.Position.X = characters[s].X * Engine.TileWidth;
                    c.Sprite.Position.Y = characters[s].Y * Engine.TileHeight;

                    c.Sprite.Draw(gameTime, spriteBatch);
                }

            }

            spriteBatch.End();
        }

        #endregion
    }
}
```

The first thing the DrawCharacters method does is call the Begin method to start the sprite batch rendering. I loop through all of the keys in the characters member variable that holds the characters that have been added to the map. I then use the GetCharacter method of the CharacterManager class to get the character and assign it to ICharacter. If it is not null I set the position of the sprite and call it's Draw method.

This is a big part of why I use interfaces a lot. Since we defined a contract that a class implementing the interface must implement a draw method any object that is assigned to an ICharacter variable will have a Draw method with the same signature. That this method will draw a Character or PCharacter without having to include code changes. You can also do the same thing with inheritance as well. Have one base class and multiple classes that inherit from that class. The sub classes can then override the default behaviour of the parent class.

The last thing that I'm going to tackle in this tutorial is collision detection between the player and the characters. This will be done in the GamePlayState's Update method. Modify the Update method in GamePlayState to the following.

```
public override void Update(GameTime gameTime)
{
    Vector2 motion = Vector2.Zero;
    int cp = 8;

    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);

        Rectangle pRect = new Rectangle(
            (int)player.Sprite.Position.X + (int)motion.X + cp,
            (int)player.Sprite.Position.Y + (int)motion.Y + cp,
            Engine.TileWidth - cp,
            Engine.TileHeight - cp);

        foreach (string s in map.Characters.Keys)
        {
            ICharacter c = GameRef.CharacterManager.GetCharacter(s);
            Rectangle r = new Rectangle(
                (int)map.Characters[s].X * Engine.TileWidth + cp,
                (int)map.Characters[s].Y * Engine.TileHeight + cp,
                Engine.TileWidth - cp,
                Engine.TileHeight - cp);

            if (pRect.Intersects(r))
            {
                motion = Vector2.Zero;
                break;
            }
        }

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

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

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

        base.Update(gameTime);
    }
```

I included a local variable cp that stands for collision padding. This value is used to reduce the destination rectangles for sprites so that it is more inside the sprite. Since a lot of the sprite is white space it allows the player to get their sprite closer to other characters. When checking to see if the player is trying to move their sprite I create a rectangle based on where the player is trying to move the sprite to. I also add the padding to the X and Y because that will make those values inside the sprite and subtract it from the height and width for the same reason. In a foreach loop I iterate over all of the characters that have been added to the map. I then get the character using the character manager and assign it to an ICharacter variable, similarly as before. I then create its rectangle using the padding. If the player's rectangle and the character's rectangle intersect there is a collision between the two and I cancel the movement. That is why I added the motion to the player's position.

I also included a minor bug fix here. What was happening is if you moved the player it would animated as expected. If you stopped moving the player it will still animation which is the wrong behaviour. So, if there is no motion I set the IsAnimating property of the player's sprite to false.

That was a little longer than I had anticipated but a very important component of the game so I'm going to stop the tutorial at this point because everything is functioning as expected. In the next tutorial I will get started on being able to talk to characters by adding some conversation components to the game.

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. You can also follow my tutorials on Twitter at https://twitter.com/GPAAdmi77640534.

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