A Summoner's Tale – MonoGame Tutorial Series

Chapter 1

Getting Started

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 2015 versions as well.

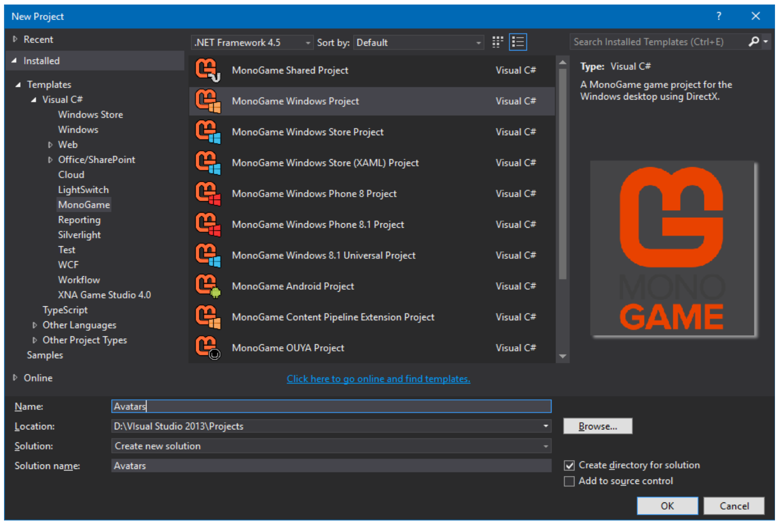
I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

First, let me give you a brief overview of the features of the finished game. This will be a 2D top down game similar to the Pokemon series of games. The player will be able to explore the world, interact with objects and non-player characters. They will battle against other players to gain experience. There will be a basic quest system as well.

The player character is a Summoner. Summoners are able to summon avatars from one of the elemental planes. There are six elemental planes: Light, Water, Air, Dark, Fire and Earth. There are many different types avatars and each avatar is attuned to a particular summoner. The summoner's avatars gain experience by fighting and defeating other avatars. The player can learn to summon other avatars by interacting with the non-player characters in the game or by finding rare spellbooks that contain the necessary rituals.

Combat between avatars will be the standard turn based combat. I hit you then you hit me and move to the next turn. Order will be based on the avatars' speed attribute. Each element is strong against one or more elements and weak against another.

To get started fire up Visual Studio and create a new MonoGame Windows Project. Call this new project Avatars. This is what my project looks like.



Type: Visual C#

A MonoGame game project for the Windows desktop using DirectX.

Name: Avatars
Location: D:\Visual Studio 2013\Projects
Solution: Create new solution
Solution name: Avatars
[✓] Create directory for solution
[ ] Add to source control

I always like adding in some "plumbing" before I start work on game play. Typically the first thing that I add is the game state manager. Right click the Avatars project, select Add and then New Folder. Name this folder StateManager. Now right click the StateManager folder, select Add and then Class. Name this class GameState. Right click the StateManager folder again, select Add and then Class. Name this class GameStateManager.

Open the GameState class and replace the code in that class with the following code. If you've not read one of my tutorials before I prefer to let you read the code and then explain what it is doing.

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

namespace Avatars.StateManager
{
    public interface IGameState
    {
        GameState Tag { get; }
        PlayerIndex? PlayerIndexInControl { get; set; }
    }

    public abstract partial class GameState : DrawableGameComponent, IGameState
    {
        #region Field Region

        protected GameState tag;
        protected readonly IStateManager manager;
        protected ContentManager content;
        protected readonly List<GameComponent> childComponents;

        protected PlayerIndex? indexInControl;

        public PlayerIndex? PlayerIndexInControl
        {
            get { return indexInControl; }
            set { indexInControl = value; }
        }

        #endregion
        #region Property Region

        public List<GameComponent> Components
        {
            get { return childComponents; }
        }

        public GameState Tag
        {
            get { return tag; }
        }

        #endregion

        #region Constructor Region

        public GameState(Game game)
            : base(game)
        {
            tag = this;

            childComponents = new List<GameComponent>();
            content = Game.Content;

            manager = (IStateManager)Game.Services.GetService(typeof(IStateManager));
        }

        #endregion

        #region Method Region

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

        public override void Update(GameTime gameTime)
        {
            foreach (GameComponent component in childComponents)
                if (component.Enabled)
                    component.Update(gameTime);

            base.Update(gameTime);
        }

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

            foreach (GameComponent component in childComponents)
                if (component is DrawableGameComponent &&
((DrawableGameComponent)component).Visible)
                    ((DrawableGameComponent)component).Draw(gameTime);
        }

        protected internal virtual void StateChanged(object sender, EventArgs e)
        {
            if (manager.CurrentState == tag)
                Show();
            else
                Hide();
        }

        public virtual void Show()
        {
            Enabled = true;
            Visible = true;

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = true;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = true;
            }
        }

        public virtual void Hide()
        {
            Enabled = false;
            Visible = false;

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = false;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = false;
            }
        }

        #endregion
    }
}
```

First, there is an interface that can be applied to other game states that are going to be implemented in the game. The two items that will need to implemented in any class that this are a Tag property and a PlayerIndexInControl. The first property is to make retrieving another state from the state manager in for use in other game states. PlayerIndexInControl was added to allow for the use of XBOX 360 controllers. I'm keeping it in because players can still connect controllers to their computers and use those for input rather than mouse and keyboard.

The GameState class is an abstract class that has methods that can be overridden in other game states and protected members that will be common to all game states. The GameState class derives from DrawableGameComponent. This adds LoadContent, Update and Draw methods to the game state so they can be called from other classes. It also implements the IGameState interface. In theory you could add the IGameState members to the abstract class. I just have always used this approach since I first discovered it.

There are a few fields in this class. The first is a field for the Tag property that needs to be implemented from IGameState. There is a readonly field of type IStateManager that gives us access to the GameStateManager that we will be implementing. I added a ContentManager field as well that will allow game states to load content. Next is a list of child game components. This allows us to add other GameComponents to the state. These components can be updated and rendered from the parent component. The last field is used to implement the PlayerIndexInControl property from the interface.

There are public properties to implement the two interface properties. There is also a property that will expose the child components of the GameState class. This will be useful if you need to add a component from one GameState to another GameState. An example is after creating the player object it can be passed from the character generator to the game play state in this way.

The constructor just initializes some of the fields. The interesting one is the way that the IStateManager is retrieved. The GameStateManager will register itself as a service. Once it is registered as a service it can be retrieved using GameSerivceContainer.Services method. This method takes the type of service to be retrieved as a parameter.

LoadContent method just calls the base.LoadContent method. The Update method loops over all of the child components in a foreach loop. It checks to see if the component is enabled. If it is enabled it will call the Update method of that component. Draw works pretty much the same as Update. It just checks if the component can be drawn and if it is visible. If both are true it will render the component.

Next is an event handler, similar to what you'd find in a WinForms project. It will be called to notify all game states that there is going to a change in the active component. It checks to see if this state is now the current state. If it is the current state at calls the Show method that sets the Visible and Enabled properties of the component to true. It then loops through the child components and enables them. It also checks to see if the component is drawable and if it is it will set the Visible property to true. If it is not the current state it calls the Hide method that works in the reverse of the Show method. It will set the Enabled and Visible property to false as well as setting the applicable property of child components to false as well.

That is it for the GameState class. Now open the GameStateManager class and replace the code with the following.

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

namespace Avatars.StateManager
{
    public interface IStateManager
    {
        GameState CurrentState { get; }

        event EventHandler StateChanged;

        void PushState(GameState state, PlayerIndex? index);
        void ChangeState(GameState state, PlayerIndex? index);
        void PopState();
        bool ContainsState(GameState state);
    }

    public class GameStateManager : GameComponent, IStateManager
    {
        #region Field Region

        private readonly Stack<GameState> gameStates = new Stack<GameState>();

        private const int startDrawOrder = 5000;
        private const int drawOrderInc = 50;
        private int drawOrder;

        #endregion

        #region Event Handler Region

        public event EventHandler StateChanged;

        #endregion
        #region Property Region

        public GameState CurrentState
        {
            get { return gameStates.Peek(); }
        }

        #endregion

        #region Constructor Region

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

        #endregion

        #region Method Region

        public void PushState(GameState state, PlayerIndex? index)
        {
            drawOrder += drawOrderInc;
            AddState(state, index);
            OnStateChanged();
        }

        private void AddState(GameState state, PlayerIndex? index)
        {
            gameStates.Push(state);
            state.PlayerIndexInControl = index;
            Game.Components.Add(state);
            StateChanged += state.StateChanged;
        }

        public void PopState()
        {
            if (gameStates.Count != 0)
            {
                RemoveState();
                drawOrder -= drawOrderInc;
                OnStateChanged();
            }
        }

        private void RemoveState()
        {
            GameState state = gameStates.Peek();

            StateChanged -= state.StateChanged;
            Game.Components.Remove(state);
            gameStates.Pop();
        }

        public void ChangeState(GameState state, PlayerIndex? index)
        {
            while (gameStates.Count > 0)
                RemoveState();

            drawOrder = startDrawOrder;
            state.DrawOrder = drawOrder;
            drawOrder += drawOrderInc;

            AddState(state, index);
            OnStateChanged();
        }

        public bool ContainsState(GameState state)
        {
            return gameStates.Contains(state);
        }

        protected internal virtual void OnStateChanged()
        {
            if (StateChanged != null)
                StateChanged(this, null);
        }

        #endregion
    }
}
```

The GameStateManager will track states using a stack. If you're not familiar with stacks a stack is a data structure that models a stack you find in the real world, like a stack of plates. Stacks implement what is called last in first out, or LIFO, model. Continuing with the stack of plates analogy when adding a plate you place it on the top of the stack and it will be the last plate added. When you go to get a plate you take the plate off the top of the stack. (We have an invisible barrier around the stack that prevents you from going and retrieving a plate other than the top plate.) I n code when you add an item to a stack you “push” it onto the stack. Similarly when you remove an item from the stack you “pop” it off the stack.

There is another interface in this class. This interface will be used when registering the state manager as a service. It has a property that returns the current game state. It also has an event that must be implemented. This event will trigger the event handler in all active games states, the ones on the stack. There are then methods that expose the functionality of the state manager. There is one to add a new state to the stack, PushState. Another, PopState, that will remove the current state off the stack. The ChangeState method removes all states off the stack and pushes the new state onto the stack. Finally, ContainsState is used to check if a state exists on the stack.

The GameStateManager inherits from GameComponent so it can be registered as a service and have its Update method called automatically. It also implements the interface that I was just speaking off. There is a read only member variable, gameStates, that is a Stack<GameState> that will be used in implementing the state manager. DrawableGameComponents have a DrawOrder associated with them. Components with higher values will be drawn before components with lower values. So, there is a field that is used for assigning game states a draw order. There are constants that hold the initial draw order of a component and an increment that will be added when a new state is pushed onto the stack or decremented when a state is popped off the stack.

There is also the event handler that needs to be implemented from the interface. If a component has subscribed to the event its internal handler will be called when this event is raised. The CurrentState is property uses the Peek method to return the state on the top of the stack.

The constructor is where we add the state manager as a service. That is done using the GameServiceContainer.AddService method. It works in opposite of the method that we used earlier to retrieve the service.

PushState receives the game state to be added and a PlayerIndex? parameter for the game controller in use. If you don't want to support controllers you can pass null whenever you require this parameter or leave it out entirely. It increments the draw order a calls a function AddState that will add the state to the stack. It then calls the OnStateChanged method that will raise the event for state change.

The AddState method pushes the state onto the stack. It updates the PlayerIndexInControl memeber of the state. Next it adds the state to the list of components for the game. Finally it subscribes the state to the StateChanged event.

PopState checks to see if there is a state on the stack. If there is not it calls RemoveState and decrements the draw order for the components. It also calls the OnStateChanged method to raise the state changed event.

RemoveState uses Peek to get the state that is on top of the stack. It unsubscribes the state from the event handler so it will no longer be notified. Next is removes the state form the list of game components. Finally it pops the state of the stack.

As I mentioned earlier ChangeState removes all states on the stack and then pushes the new state onto the stack. The first thing that it does is loop until there are no states on the stack and call the RemoveState method to remove the state from the stack. It then resets the drawOrder variable to its base value. It sets the draw order of the new state and increments the draw order again. It then calls AddState to push the state on the stack and calls the OnStateChanged to raise the state change event.

The method ContainsState just checks to see if state passed in is already on the stack of states. OnStateChanged checks to see if there are any subscribers to the StateChanged event. If there are it calls StateChanged(this, null) to raise the event. We're not too concerned about the sender or event args here so I passed the instance of the state manager and null for the event arguments.

The last things that I'm going to implement in this tutorial is the title screen. To do that please download the following two images, or replace them with two of your own, Summoner's Tale Images. In your solution expand the Content folder, right click Content.mgcb and select Open.

With the content builder that comes up right click on the Content folder, select Add and then New Folder. Name this new folder Fonts. Repeat the process but call the folder GameScreens. Now right click the Fonts folder, select Add and then New Item. Select Sprite Font Description from the list and name it InterfaceFont. Now right click the GameScreens folder, select Add and then Existing Item. Navigate to the two .png files from the step above and add them to the project. Save the item and then build to make sure that there are no problems and then close the content builder window. If your sprite font does not show up in the solution explorer you can select the Show All Files button at the top of the solution explorer. Now if you go to the content folder you should see a greyed out file. Just right click it and select Include in Project.

Now, right click the Avatars project, select Add and then New Folder. Name this new folder GameStates. Now right click this new folder, select Add and then Class. Name this class BaseGameState. Repeat the process and and name the class TitleIntroState.

Open the BaseGameState class and replace the code with the following.

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

namespace Avatars.GameStates
{
    public class BaseGameState : GameState
    {
        #region Field Region

        protected static Random random = new Random();

        protected Game1 GameRef;

        #endregion

        #region Constructor Region

        public BaseGameState(Game game)
            : base(game)
        {
            GameRef = (Game1)game;
        }

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

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

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

        #endregion
    }
}
```

This really just a skeleton class that can be used to allow for polymorphism of game states. What that means is if I derive TitleIntroState from BaseGameState I can assign an instance of TitleIntroState to a variable BaseGameState. Similarly this class inherits from GameState so I can assign any instance of a class that inherits from BaseGameState to a GameState. This allows us to use any class that inherits from BaseGameState in the state manager.

The two important things are that this class exposes a property GameRef that returns a reference to the current game object. This will come in handy many times when we need a specific item from the game object in another component. The other thing is there is a static Random instance variable that will be shared between all of the game state classes. You might consider setting a fixed seed when creating this variable for debugging. This way you will know what the results of the next random variable will be so that your code flows the same way each time. You will definitely want to stop this behaviour when you publish your game.

Now, open the TitleIntroState class and replace it with the following code.

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

namespace Avatars.GameStates
{
    public interface ITitleIntroState : IGameState
    {
    }

    public class TitleIntroState : BaseGameState, ITitleIntroState
    {
        #region Field Region

        Texture2D background;
        Rectangle backgroundDestination;
        SpriteFont font;
        TimeSpan elapsed;
        Vector2 position;
        string message;

        #endregion

        #region Constructor Region

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

        #endregion

        #region Method Region

        public override void Initialize()
        {
            backgroundDestination = Game1.ScreenRectangle;
            elapsed = TimeSpan.Zero;
            message = "PRESS SPACE TO CONTINUE";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            background = content.Load<Texture2D>(@"GameScreens\titlescreen");
            font = content.Load<SpriteFont>(@"Fonts\InterfaceFont");

            Vector2 size = font.MeasureString(message);
            position = new Vector2((Game1.ScreenRectangle.Width - size.X) / 2,
Game1.ScreenRectangle.Bottom - 50 - font.LineSpacing);

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            PlayerIndex index = PlayerIndex.One;
            elapsed += gameTime.ElapsedGameTime;

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(background, backgroundDestination, Color.White);

            Color color = new Color(1f, 1f, 1f) *
(float)Math.Abs(Math.Sin(elapsed.TotalSeconds * 2));

            GameRef.SpriteBatch.DrawString(font, message, position, color);
            GameRef.SpriteBatch.End();

            base.Draw(gameTime);
        }

        #endregion
    }
}
```

First, there is an interface that derives from the IGameState interface that we created earlier. This is just allowing us to hook into those items for use in the state manager. The interface will be used to register the component as a service that we can retrieve in another class if necessary.

This class class inherits from BaseGameState and implements the ITitleIntroState interface which makes us implement IGameState as well. There are member variables for the background image, the destination of the background image, the interface font, a TimeSpan that measures the amount of time that has passed. This will be used to have a message blink in and out. There is also a member variable for the message and its position.
The constructor just registers the game state as service that can be retrieved using its interface. If there was something that needed to be exposed to an external component that can be added to the interface and retrieved by getting the added service.

In the Initialize method I set the destination of the background image to a property that I haven't added to the Game1 class yet that describes the screen that we will be rendering into. I also initialize the elapsed time to zero and set the message that will be displayed.

In the LoadContent method I load the background for the titlescreen and the interface font. I then measure the size of the message that will flash. I then use the value to calculate where to display the message centered vertically and by the bottom off the screen.

In the Update method I the elapsed variable the that is incremented each pass through the loop. As I mentioned earlier I will be using this to have the message that will be displayed fade in and out.

In the Draw method I render the the background and the message. To do that I expose the SpriteBatch in the Game1 class as a read only property. I will get to that shortly. To make the message fade in and out I use Math.Sin. I do that because sine is a cyclic wave that repeats indefinitely between 1 and -1. I use Math.Abs to ensure that it is always positive. I multiply Color(1f,1f,1f) by this value so that it flashes in white. I then draw the message.

Open up the Game1 class now so that we can add all this plumbing/scaffolding to the game. Replace the Game1 class with the following code.

```
using Avatars.GameStates;
using Avatars.StateManager;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Avatars
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GameStateManager gameStateManager;
        ITitleIntroState titleIntroState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        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);

            titleIntroState = new TitleIntroState(this);

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

        /// <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

            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
        }

        /// <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

            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);
        }
    }
}
```

I added a field for the state manager and the title introduction screen. I also added a static field that will describe the bounds of the game screen. I added a public get only property that will return the SpriteBatch object for the game and a property that will expose the screen area rectangle as well. In the constructor I create a rectangle that will describe the screen area. I went low for resolution, 1280 by 720, to accommodate readers that are using laptops that default to the 1366 by 766 resolution or lower. You can easily change this for your game or make it dynamic so that the player can change the resolution for the game.

The next step is to set the PreferredBackBufferWidth and PreferredBackBufferHeight values of the GraphicsDeviceManager to match the height and width of the rectangle that describes the screen.

After that you need to create an instance of GameStateManager and add it to the Components collection so that it will be updated during the call to base.Update in the Update method. Next, I create a TitleIntroScreen object and assign it to ItitleIntroScreen. Finally I call the ChangeState method of the GameStateManager passing in the title introduction state and PlayerIndex.One.

If you run and build the project you should see the title introduction screen displaying with the message on how to proceed to the next screen.

I'm going to end the tutorial here as we've covered a lot already. Stay tuned for the next tutorial in the series. In that one I will be implementing a few new features that will be helpful for the rest of the game. I wish you the best in your MonoGame Programming Adventures!.

Jamie McMahon

A Summoner's Tale – MonoGame Tutorial Series

Chapter 2

More Plumbing/Scaffolding

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 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.

In the last tutorial I set up the base project with the state manager and a title/intro screen. In this tutorial I will be adding more plumbing/scaffolding to the game as there are some key components that are missing. I also discovered after loading up my project that there was a problem with the fonts that were added. I've fixed this in the project. First, I renamed InterfaceFont.interfacefont to InterfaceFont.spritefont. I also added a new font called GameFont.spritefont using the same process as in the last tutorial.

A quick note on fonts while I'm on the subject. Most fonts are commercial products and cannot be used in your game without paying royalties or buying the font out right. There are a lot of fonts that are available to game developers though. The two sources that I use are http://www.1001fonts.com and http://www.dafont.com. If you filter you will find many different licenses, including 100% free. I'd suggest that if you find a font and use it in your game that you mention to creator somewhere in your credits.

In the last tutorial we got the state manager and added a state and it is rendering. Other than that it doesn't do anything. In order for it to do something it will have to react to user input. I'm going to add a basic version of my input handling component that I wrote for use in XNA 3.0, a long time ago now, called Xin. It attempts to centralize the important code for handling input to try and reduce complexity in the rest of the game. It is pretty common for a developer to do this sort of abstraction process. Find a common task and create a class that wraps that task so that it can be used again in other projects.

First, right click your project and select Add and then New Folder. Name this new folder Components. Now, right click the Components folder that was just added, select Add and then Class. Name this new class Xin. The code for that class follows.

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

namespace Avatars.Components
{
    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 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();

            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;
        }
    }
}
```

You will see that there are a lot of static members in this class. I did that so instead of creating an instance of the class in each other class that it is used in you have a single point for handling input. When you need to do anything input related you just use Xin.member_name.

First, you will notice that there is an enumeration at the namespace level called MouseButtons. I added this because on the one thing that I've always found lacking in XNA and MonoGame because it derives from it is mouse support. What I mean is the keyboard and game pad states can be referenced using the Keys and Buttons enumerations where as for mouse input you have no such option.

This class then derives from GameComponent. I did that so that we can create an instance of Xin and add it to the list of components without creating a member variable in a class and forget about it, for all intents and purposes.

Next there are two variable for KeyboardState and MouseState. These variables hold the current state of the mouse and keyboard and the previous state of the mouse and the keyboard. You need both to check to see when a button or key is first pressed or when it is released. Otherwise you can't tell if a button was down since the last frame or if it was released since the last frame. This will make more sense in a bit.

I also added static readonly properties for both input types and their frame, whether they are the current frame of the game or last frame of the game. These provide raw access to input which is important in some instances.

The constructor is trivial really. It just calls the base constructor that requires a Game parameter that represents the game object the component will be added to. You could in theory require more than one Game object in a game but I've never had to use one.

The methods that follow are the real meet of the class and do most of the work. I will be extending these as time goes on but for the purposes of this tutorial they are sufficient. The first method, Update, is inherited from the GameComponet class. The way this works is if you create a component and add it to the list of components in a game their Update method is called automatically each frame, if the component is enabled. Inside the update method I first set the variables named previous to current. I then get the current state of the keyboard and mouse. This is how we will detect when a key is pressed or release, but not if it is just down or up. I then call base.Update which would call Update on other related components inside of this class.

The first static method is FlushInput. What this method does is set the current states to their previous states. Since they are now equal and of the methods that would check for new presses or releases will fail.

In CheckKeyReleased checks to see if a key that was down last frame is now up. Let me explain why I did this was as opposed to a key that was up in the previous frame is down in the current frame. What I've found checking it the second way, the key is now down, is that before the call to Xin.FlushInput happens sometimes the new press is caught in the next frame of the game. Checking for a release works around this phenomenon. To do this I use the IsDown and IsUp member methods of the KeyboardState class, which do as you might imagine.

The last, and most interesting, method is the CheckMouseReleased. I say that it is interesting is that compared to checking for keys, checking for mouse buttons is more complex. There are no built in methods of the input classes that check if a mouse button is down. All that there is are named properties that get a ButtonState enumeration that defines if the button is up or down. So, in order to check a button you retrieve the ButtonState and then check its value.

So, in the CheckMouseReleased button I have a switch statement that checks what parameter was passed into the method. Based on what button I have a statement that returns the current state of the button compared with the previous state of the button. If no other response is returned I then return false.

That is all for the Xin class. As you can see there is room for improvement, and you might be thinking of that already, which is good. It is always good to read code, see how it works and then try to implement it or another version of it. It is never a good idea to just copy and paste code without understanding it. That is a speech that I will spare you from.

The next item that I want to add is a menu component that will be used for the main menu and other menus in the game. Right click the Components folder, select Add and then Class. Name this new class MenuComponent. Here is the code for that class.

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

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Avatars.Components
{
    public class MenuComponent
    {
        #region Fields

        SpriteFont spriteFont;
        readonly List<string> menuItems = new List<string>();
        int selectedIndex = -1;
        bool mouseOver;

        int width;
        int height;

        Color normalColor = Color.White;
        Color hiliteColor = Color.Red;

        Texture2D texture;

        Vector2 position;

        #endregion Fields

        #region Properties

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

        public int Width
        {
            get { return width; }
        }

        public int Height
        {
            get { return height; }
        }

        public int SelectedIndex
        {
            get { return selectedIndex; }
            set
            {
                selectedIndex = (int)MathHelper.Clamp(
                        value,
                        0,
                        menuItems.Count - 1);
            }
        }

        public Color NormalColor
        {
            get { return normalColor; }
            set { normalColor = value; }
        }

        public Color HiliteColor
        {
            get { return hiliteColor; }
            set { hiliteColor = value; }
        }

        public bool MouseOver
        {
            get { return mouseOver; }
        }

        #endregion Properties
        #region Constructors

        public MenuComponent(SpriteFont spriteFont, Texture2D texture)
        {
            this.mouseOver = false;
            this.spriteFont = spriteFont;
            this.texture = texture;
        }

        public MenuComponent(SpriteFont spriteFont, Texture2D texture, string[] menuItems)
            : this(spriteFont, texture)
        {
            selectedIndex = 0;

            foreach (string s in menuItems)
            {
                this.menuItems.Add(s);
            }

            MeassureMenu();
        }

        #endregion Constructors

        #region Methods

        public void SetMenuItems(string[] items)
        {
            menuItems.Clear();
            menuItems.AddRange(items);
            MeassureMenu();

            selectedIndex = 0;
        }

        private void MeassureMenu()
        {
            width = texture.Width;
            height = 0;

            foreach (string s in menuItems)
            {
                Vector2 size = spriteFont.MeasureString(s);

                if (size.X > width)
                    width = (int)size.X;

                height += texture.Height + 50;
            }

            height -= 50;
        }

        public void Update(GameTime gameTime, PlayerIndex index)
        {
            Vector2 menuPosition = position;
            Point p = Xin.MouseState.Position;

            Rectangle buttonRect;
            mouseOver = false;

            for (int i = 0; i < menuItems.Count; i++)
            {
                buttonRect = new Rectangle((int)menuPosition.X, (int)menuPosition.Y,
texture.Width, texture.Height);
```

```
                if (buttonRect.Contains(p))
                {
                    selectedIndex = i;
                    mouseOver = true;
                }

                menuPosition.Y += texture.Height + 50;
            }

            if (!mouseOver && (Xin.CheckKeyReleased(Keys.Up)))
            {
                selectedIndex--;
                if (selectedIndex < 0)
                    selectedIndex = menuItems.Count - 1;
            }
            else if (!mouseOver && (Xin.CheckKeyReleased(Keys.Down)))
            {
                selectedIndex++;
                if (selectedIndex > menuItems.Count - 1)
                    selectedIndex = 0;
            }
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            Vector2 menuPosition = position;
            Color myColor;

            for (int i = 0; i < menuItems.Count; i++)
            {
                if (i == SelectedIndex)
                    myColor = HiliteColor;
                else
                    myColor = NormalColor;

                spriteBatch.Draw(texture, menuPosition, Color.White);

                Vector2 textSize = spriteFont.MeasureString(menuItems[i]);

                Vector2 textPosition = menuPosition + new Vector2((int)(texture.Width -
textSize.X) / 2, (int)(texture.Height - textSize.Y) / 2);
                spriteBatch.DrawString(spriteFont,
                    menuItems[i],
                    textPosition,
                    myColor);

                menuPosition.Y += texture.Height + 50;
            }
        }

        #endregion Methods

        #region Virtual Methods
        #endregion Virtual Methods

    }
}
```

Quite a bit of code but it is nothing that is overly complex. First, there are a number of member fields in the class. Since the menu needs to draw text I added a SpriteFont field. There is then a List<string> that will contain the individual menu items to be rendered. The selectedIndex field will return what menu item is currently selected. Next mouseOver is added to allow for mouse support. The width and height fields will be used to control where the menu is rendered. There are two Color fields that control what color menu items are rendered. There is one color for regular menu items and then a second for highlighted menu items. Texture is a field that will serve as a button background image and position controls where the menu it displayed.

I added some public properties that will expose some of the fields to other classes. You should be able to set the position of the menu items so I exposed the position member variable. Frequently you will need to know the width or height of the menu so I added properties for that as well. I added a property to return and set the selected menu item. You will see that in the set part I use MathHelper.Clamp to make sure the selected item is not outside of the available menu items. There are properties that allow you to get and set the menu colors as well. Finally there is a property that will return the mouseOver member variable.

You will notice that I did not add a property that returns the menu items. The reason is that when the menu items change I want to force a call to a method that will find the height and width of the menu. If I exposed the list it can be modified outside of the class and changes would not be picked up and could lead to a change in the was the menu is rendered.

I added two constructors. The first takes two parameters, the font the menu will be rendered with and the button texture. This one just initializes some of the fields that I added. The second constructor accepts an array of strings as well as the others. It will also call the first constructor to set some of the default values. Inside I set the selectedIndex member variable to 0, the first item, and then in a foreach loop I add each string to the menu items. I then call MeasureMenu to calculate the height and width of the menu.

The first method is called to change the menu items for the menu. It takes a string array that hold the menu items to be displayed. If clears any existing menu items, adds them to the list of menu items, calls the MeasureMenu method to calculate the width and height then finally sets the selectedIndex member variable to 0.

MeasureMenu is where I calculate the width and height of the menu. First, I set width to be the width of the texture. You technically do not want to menu items to be longer than the width of your button but you need to allow for this scenario. I reset the height to 0 as well. In a foreach loop I iterate over all of the menu items. I use the MeasureString member of the SpriteFont class to get the height and width of the string. If the X value is greater than the width than the current width I update that value. I then add the height of the button and some padding to the height member variable. I then subtract the padding as it is added to the last menu item.

In the Update method I handle updating the menu component. I added a local variables that hold the position that the menu is drawn and the mouse as a Point. I then have a Rectangle that will be the bounds of each menu item to determine if the mouse is over them or not. After the variables I set the mouseOver variable to false. This ensures that it is reset at the start of each frame of the game.

Next there is a for loop that loops over the menu items one by one. Inside I calculate the Rectangle that holds the bounds of the button. I use the local variable that is set to be the position of the menu and the height and width of the button texture. I check to see if the mouse is over the current button. If it is I set the selectedIndex member to be that menu and the mouseOver variable to true. I add the padding and the texture height to the current menu item position before moving to the next item.

The if statement that follows handles the user moving the selected menu item using the up and down keys. I do not change the position if the mouse is over a button as that is the button that should have focus. To move the selection up I decrement the selectedIndex member. If it is less than zero I set the menu item to last menu item, which is the Count member of the list of menu items minus one, since it is zero based. To move down I increment the selectedIndex member. I then check if it is outside the bounds of the list and if it is set it to zero.

The last method is the Draw method that will render the menu items. There are two local variables that hold the menu position and color that the items will be drawn in. You then loop over each of the menu items. I first set the color variable based on what the selected index is. I then draw the button image. I measure the menu item string. I then calculate its position relative to the button texture and center it. I then draw the menu item string. Finally I update the Y coordinate by adding the height of the texture plus the padding.

The last new class that I'm going to add is the start state that displays a menu to the user allowing them to pick options. Right click the GameStates folder, select Add and then Class. Name this new class MainMenuState. Here is the code for that class.

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

namespace Avatars.GameStates
{
    public interface IMainMenuState : IGameState
    {
    }

    public class MainMenuState : BaseGameState, IMainMenuState
    {
        #region Field Region

        Texture2D background;
        SpriteFont spriteFont;
        MenuComponent menuComponent;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

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

        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            spriteFont = Game.Content.Load<SpriteFont>(@"Fonts\InterfaceFont");
            background = Game.Content.Load<Texture2D>(@"GameScreens\menuscreen");

            Texture2D texture = Game.Content.Load<Texture2D>(@"Misc\wooden-button");

            string[] menuItems = { "NEW GAME", "CONTINUE", "OPTIONS", "EXIT" };

            menuComponent = new MenuComponent(spriteFont, texture, menuItems);

            Vector2 position = new Vector2();

            position.Y = 90;
            position.X = 1200 - menuComponent.Width;

            menuComponent.Postion = position;

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            menuComponent.Update(gameTime);

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter) ||
(menuComponent.MouseOver && Xin.CheckMouseReleased(MouseButtons.Left)))
            {
                if (menuComponent.SelectedIndex == 0)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 1)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 2)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 3)
                {
                    Game.Exit();
                }
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(background, Vector2.Zero, Color.White);

            GameRef.SpriteBatch.End();

            base.Draw(gameTime);

            GameRef.SpriteBatch.Begin();

            menuComponent.Draw(gameTime, GameRef.SpriteBatch);

            GameRef.SpriteBatch.End();

        }

        #endregion
    }
```

```
}
```

First, there are using statements to bring the Components namespace in this project as well as a few of the Xna.Framework namespace. There is an empty interface at the start of this class called IMainMenuState. One use for interfaces is to define a contract that other objects can use to interact with this object. An empty interface means that there are no members that will be exposed directly to other objects.

The class itself inherits from BaseGameState so it can be used in the state manager and it implements the ImainMenuSate interface. There are three private member variables: background, spriteFont and menuComponent. The background variable will hold the background image that will be render, spriteFont is for drawing any text and menuComponent will render our menu.

In the constructor I register the class as a service with the framework so it can be retrieved in other classes. This one won't be but I'm being consistent will all states.

In the LoadContent menu I load the font and background. I also load an image for the menu items. Download the Misc content items from this link and extract the file. Back in the project open the content manager by left clicking Content.mgcb and selecting Open. Select the Content node and create a new folder Misc. Right click the Misc folder, select Add and the Existing Item. Browse to the wooden-button.png file and add it. Now, right click the GameScreens folder under the Content folder. If you haven't added the menuscreen.png file to this folder do so now. Once all items are added open the Build menu and select Rebuild.

After loading the content I create a string that holds the menu items that will be displayed and create the menu component. I then position the menu, based on the image that I created for the background. The buttons should sit on top of the chains on the right side of the screen.

In the Update method I first call the Update method of the menu before checking if the space, enter or left mouse button have been released since the last frame. There is then a chain of if-else statements, one for each of the menu items. Inside the if statement I call Xin.FlushInput just to make sure there are no leftovers. The interesting part is that in the last index I call Game.Exit to close the game. It would be better if you put up a pop up asking if the player that they really want to end the game. I will implement that in a future tutorial.

The Draw method is pretty simple. It first renders the background, calls base.Draw and then renders the menu.

Now, I'm going to implement the title screen changing state to the menu state when space, enter or the left mouse button are released. First, update the using statements in the class to make sure all necessary entities are in scope for the class.

```
using System;
using Avatars.Components;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
```

Now, update the Update method to the following. You will get an error message on the state change because we have not yet updated the main game class yet. That is what we will tackle next.

```
public override void Update(GameTime gameTime)
{
    PlayerIndex? index = null;

    elapsed += gameTime.ElapsedGameTime;

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter) ||
Xin.CheckMouseReleased(MouseButtons.Left))
    {
        manager.ChangeState((MainMenuState)GameRef.StartMenuState, index);
    }

    base.Update(gameTime);
}
```

There is a nullable variable, index, for game pad support. I'm not including that at the moment but I will be in the future so I need to park this here. I then call the methods off the Xin class to check for the release of the space key, enter key or left mouse button. If any of those have been released I call the ChangeState method to change the state the menu state. You will have an error until we update the Game1 class.

We're in the home stretch now. All that is remaining is updating the Game1 class to handle this new logic. The changes were pretty extensive so I will paste the code for the entire class.

```
using Avatars.Components;
using Avatars.GameStates;
using Avatars.StateManager;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Avatars
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GameStateManager gameStateManager;

        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        public GameStateManager GameStateManager
        {
            get { return gameStateManager; }
        }

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        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);

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

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

            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);
        }
    }
}
```

So, I first removed all of the extra comments that are part of the template to shorten the class as they are not necessary for the project. I also added in a using statement to bring the Components namespace into scope. I then added a member variable of type IMainMenuState for the start menu for the game. I also added in some properties to expose the member variables to game objects that we will be creating.

In the constructor for the Game1 class I set the IsMouseVisible property to try so that the mouse cursor will be displayed. I also created the new MainMenuState object. Also, in the Initialize method I add an instance of Xin to the game components for the game.

Before building and running the game, first open the Content manager and build the content for the game. Once the content builds successfully then build and run the game. At this point the title screen will appear. Pressing either the space or enter key or clicking the left mouse button will move to the next scene. Once on the menu scene clicking the Exit button will close the game.

I'm going to end the tutorial here as we've covered a lot. The first two tutorials were a little dry in regard to game elements but at this point most of the plumbing/scaffolding is in place. Now we can move on to game elements. Stay tuned for the next tutorial in the series. In that one I will be implementing and new game state and some game play elements.

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 3

Tile Engine and Game Play State

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.

In this tutorial I will be adding the game state for the player exploring the map and the tile engine to render the map. Why do you need a tile engine though? You could just draw the map in an image editor, load that and draw it. The reason is memory and efficiency. Most of the map will be made up of the same background images. Creating a tile based on that and rendering just that saves a lot of memory. It is also more efficient to load a few hundred small images than one extremely large image. The same is true for rendering.

First, right click the Avatars project, select Add and then New Folder. Name this new folder TileEngine. Now right click the TileEngine folder, select Add and then Class. Name this new class TileSet. Here is the code for that class.

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

namespace Avatars.TileEngine
{
    public class TileSet
    {
        public int TilesWide = 8;
        public int TilesHigh = 8;
        public int TileWidth = 64;
        public int TileHeight = 64;

        #region Fields and Properties

        Texture2D image;
        string imageName;
        Rectangle[] sourceRectangles;

        #endregion

        #region Property Region

        [ContentSerializerIgnore]
        public Texture2D Texture
        {
            get { return image; }
            set { image = value; }
        }

        [ContentSerializer]
        public string TextureName
        {
            get { return imageName; }
            set { imageName = value; }
        }

        [ContentSerializerIgnore]
        public Rectangle[] SourceRectangles
        {
            get { return (Rectangle[])sourceRectangles.Clone(); }
        }

        #endregion

        #region Constructor Region

        public TileSet()
        {
            sourceRectangles = new Rectangle[TilesWide * TilesHigh];

            int tile = 0;

            for (int y = 0; y < TilesHigh; y++)
                for (int x = 0; x < TilesWide; x++)
                {
                    sourceRectangles[tile] = new Rectangle(
                        x * TileWidth,
                        y * TileHeight,
                        TileWidth,
                        TileHeight);
                    tile++;
                }
        }

        public TileSet(int tilesWide, int tilesHigh, int tileWidth, int tileHeight)
        {
            TilesWide = tilesWide;
            TilesHigh = tilesHigh;
            TileWidth = tileWidth;
            TileHeight = tileHeight;

            sourceRectangles = new Rectangle[TilesWide * TilesHigh];

            int tile = 0;

            for (int y = 0; y < TilesHigh; y++)
                for (int x = 0; x < TilesWide; x++)
                {
                    sourceRectangles[tile] = new Rectangle(
                        x * TileWidth,
                        y * TileHeight,
                        TileWidth,
                        TileHeight);
                    tile++;
                }
        }

        #endregion

        #region Method Region
        #endregion
    }
}
```

First question then is then “What is a tile set?”. A tile set is an image that is made up of the individual tiles that will be placed on the map. When rending the map you will pick out the individual tile using the image and a source rectangle. This source rectangle defines the X and Y coordinates of the tile and the height and width of the tile. You will see this in practice shortly.

I first have 4 public member variables in this class that hold the basics of the tile set. They are TilesWide for the number of tiles across the image, TilesHigh for the number of tiles down the image, TileWidth for the width of each tile and TileHeight for the height of each tile. There are private member variables for the texture for the image, the name of the image and the source rectangles that describe each tile.

You will see that I marked some of the properties with attributes. These attributes control if the member variable will be serialized or not. For serialization to work correctly you may need the actual XNA Framework installed, depending on the version of MonoGame that you are using. We will cross that bridge in the future when we get to that point. These attributes control if the property will be serialized or not when using IntermediateSerializer. Again, I'll explain this better when we actually use it.

There is a property that exposes the name of the texture and the actual texture. There is also a readonly property that exposes the source rectangles for the tile set.

I've included two constructors in this class. The parameterless one is meant for deserializing the tile set when importing it into the game. It uses the default values that I assigned to the member variables to create the source rectangles. It will also read these values from a serialized tile set and use those instead. The rectangles are calculate inside of a nested for loop. The X coordinate is found using the width of each tile and the index for the inner loop. The Y coordinate is found using the height of each tile and the index of the outer loop. It is important to remember that Y is the outer loop and X is in the inner loop or the source rectangles will come out rotated.

The next class that I'm going to add is for a basic 2D camera that will assist in rendering only the viewable portion of that map. By drawing only the visible portion of the map we are increasing the efficiency or the rendering process. For example, say we have a map that is 200 tiles by 200 tiles. That is 40000 tiles that would need to be drawn each frame of the game. If the screen can only display 40 by 30 plus one in each direction you can get by by drawing just 1200 tiles, actually a little more because we need to pad for the right and bottom edges. That is much quicker to render than the entire map as you can see.

Now, right click the TileEngine folder, select Add and then class. Name this new class Camera. Here is the code for the Camera class.

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

namespace Avatars.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 LockCamera(TileMap map, Rectangle viewport)
        {
            position.X = MathHelper.Clamp(position.X,
                0,
                map.WidthInPixels - viewport.Width);
            position.Y = MathHelper.Clamp(position.Y,
                0,
                map.HeightInPixels - viewport.Height);
        }
    }
}
```

There are two member variables in this class. They are position and speed. Position, as you've probably already gathered, is the position of the camera on the map. Speed is the speed at which the camera moves, in pixels. I also added properties that expose both of these member variables. In the Speed property I clamp the value between a minimum and maximum values, 1px and 16px. 4px would probably be better than 1px because that would be extremely slow.

There is a third property, Transformation, that returns a translation matrix. This matrix will be used to adjust where the map is drawn on the screen. You will notice that it uses -Position. This is because the camera's position is subtract from the map coordinates being drawn.

There are two public constructors. The first is parameterless and just sets the speed to a fixed value. The second takes as a parameter the position of the camera. It then sets the position of the camera using the value and sets the default speed of the camera.

There is also one method called LockCamera. What it does is clamp the X and Y coordinates between 0 and the width of the map minus the width of the viewport for width and 0 and the height of the map minus the height of the viewport for height. I subtract the height and width of the viewport so that we do not go past the right or bottom edge. Otherwise the camera would move to the width or height of the map and the default background color would show.

The next thing that I will add is a class that represents a layer in the map. Using layers allows you to separate duties. I typically include 4 layers on a map. The first layer is the base terrain for the map. The second layer is used for transition tiles. What I mean by this is that if you have a grass and a mud tile side by side there is an image that blends the two together. These can be made up of both tiles or one of the tiles with transparent areas. I then have a layer for solid objects, such as buildings, trees, and other objects. The last layer is for decorations. These tiles are used to add interest to the map. You of course are not limited to just these four layers but they are what I've included in this tutorial.

Now, right click the Tile Engine folder, select Add and then Class. Name this new class TileLayer. Here is the code for that class.

```
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;

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

        [ContentSerializer(CollectionItemName = "Tiles")]
        int[] tiles;

        int width;
        int height;

        Point cameraPoint;
        Point viewPoint;
        Point min;
        Point max;

        Rectangle destination;
        #endregion

        #region Property Region

        [ContentSerializerIgnore]
        public bool Enabled { get; set; }

        [ContentSerializerIgnore]
        public bool Visible { get; set; }

        [ContentSerializer]
        public int Width
        {
            get { return width; }
            private set { width = value; }
        }

        [ContentSerializer]
        public int Height
        {
            get { return height; }
            private set { height = value; }
        }

        #endregion

        #region Constructor Region

        private TileLayer()
        {
            Enabled = true;
            Visible = true;
        }

        public TileLayer(int[] tiles, int width, int height)
            : this()
        {
            this.tiles = (int[])tiles.Clone();
            this.width = width;
            this.height = height;
        }

        public TileLayer(int width, int height)
            : this()
        {
            tiles = new int[height * width];
            this.width = width;
            this.height = height;

            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    tiles[y * width + x] = 0;
                }
            }
        }

        public TileLayer(int width, int height, int fill)
            : this()
        {
            tiles = new int[height * width];
            this.width = width;
            this.height = height;
            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    tiles[y * width + x] = fill;
                }
            }
        }

        #endregion

        #region Method Region

        public int GetTile(int x, int y)
        {
            if (x < 0 || y < 0)
                return -1;

            if (x >= width || y >= height)
                return -1;

            return tiles[y * width + x];
        }

        public void SetTile(int x, int y, int tileIndex)
        {
            if (x < 0 || y < 0)
                return;

            if (x >= width || y >= height)
                return;

            tiles[y * width + x] = tileIndex;
        }

        public void Update(GameTime gameTime)
        {
            if (!Enabled)
                return;
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, TileSet tileSet, Camera
camera)
        {
            if (!Visible)
                return;

            cameraPoint = Engine.VectorToCell(camera.Position);
            viewPoint = Engine.VectorToCell(
                new Vector2(
                    (camera.Position.X + Engine.ViewportRectangle.Width),
                    (camera.Position.Y + Engine.ViewportRectangle.Height)));

            min.X = Math.Max(0, cameraPoint.X - 1);
            min.Y = Math.Max(0, cameraPoint.Y - 1);
            max.X = Math.Min(viewPoint.X + 1, Width);
            max.Y = Math.Min(viewPoint.Y + 1, Height);

            destination = new Rectangle(0, 0, Engine.TileWidth, Engine.TileHeight);
            int tile;

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

            for (int y = min.Y; y < max.Y; y++)
            {
                destination.Y = y * Engine.TileHeight;

                for (int x = min.X; x < max.X; x++)
                {
                    tile = GetTile(x, y);

                    if (tile == -1)
                        continue;

                    destination.X = x * Engine.TileWidth;

                    spriteBatch.Draw(
                        tileSet.Texture,
                        destination,
                        tileSet.SourceRectangles[tile],
                        Color.White);

                }
            }

            spriteBatch.End();
        }

        #endregion
    }
}
```

A few member variables here. First, is an array of integers that holds the tiles for the layer. Instead of creating a two dimensional array I created a one dimensional array. I will use a formula to find the tile at the requested X and Y coordinates. I will also use the convention that if a tile has a value of -1 then that tile will not be drawn. Next there are width and height member variables that describe the height and width of the map.

Next up for member variables are four Point variables: cameraPoint, viewPoint, min and max. I created these at the class level because I will be using them each time that the map is drawn. Doing it this way just means that we are not constantly creating and destroying variables each frame. It is not a great optimization but even minor optimizations will help in a game. I will discuss their purpose further when I get to rendering.

The last member variable is a Rectangle variable and did it this way for the same reason as the others. The thing to note though is that this will be used every time a tile is drawn as it represents the destination the tile will be drawn in screen space. This is more effective than the other optimization. The reason is we will be drawing 4 layers with approximately 800 tiles a layer 60 times per second. Compared to 4 layers 60 times per second.

There are two auto implemented properties Enabled and Visible that control if the layer is enabled and visible. I added Enabled because I'm considering adding in animated tiles and to support that I would require an Update method. Visible will determine if the layer should be drawn or not. I also added readonly properties to expose the width and height of the layer.

There are four constructors in this class. The first has no parameters and just sets the auto properties. It is also private and will not be called outside of the class. Its main purpose is for use with the

IntermediateSerializer class what is used for serializing and deserializing content. The other three constructors are used for creating layers. The one takes an array as well as height and width parameters, the second just takes height and width parameters and the third takes height, width and fill parameters. Each of them also calls this() to initialize the Visible and Enabled properties.

The constructor that takes an array as a parameter creates a clone of the array, which is a shallow copy of the array. It then sets the width and height parameters. The constructor that takes just the width and height of the layer first creates a new array that is height * width. Next there are nested loops where y is the outer loop and x is the inner loop. To calculate where a tile is places I use the formula y * width + x. So the first row will be from 0 to 1 * width - 1, the second from 1 * width to 2 * width – 1 and then 2 * width to 3 * width -1 and so on. This formula will has to be applied consistently or you will have very strange behaviour. The third works the same as the first but rather than setting the tile to a default value it sets the tile to the value passed in.

Next are two method GetTile and SetTile. Their name definitely describes their purpose as they get and set tiles respectively. GetTile returns a value where as SetTile receives and additional value. There are if statements in each method to check that the x and y values that are passed in are within the bounds of the map. Both also use the same formula as in the constructor to get and set the tile.

I included an Update method that accepts a GameTime parameter. This parameter holds the amount of time passed between frames and has some useful purposes. Currently it checks the Enabled property is false and if it is exits the method.

The Draw method is where the meat of this class is as it actually draws the tiles. You will get some errors here because it relies on some static methods for a class I have not implemented yet, Engine. Engine just holds some common properties for the tile engine.

First, I check to see if the layer is visible or not. If it is not visible I exit the function. I then set the cameraPoint and viewPoint vectors using Engine.VectorToCell. What this call does is takes a point in pixels and returns the tile that the pixel is in. These vectors will be used to determine where to start and stop drawing tiles. As I mentioned we will only be drawing the visible tiles. Actually it will be plus and minus one tile to account for the scenario where the pixel is not the X and Y coordinates of a tile. The viewPoint vector is found by taking the camera position and adding the size of the view port the map is being drawn to. The view port is typically the entire screen in these types of games but occasionally you will find a side bar or bottom bar that holds information that reduces the map space.

The min vector is clamped between 0 and the camera position minus 1 tile. Similarly max is clamped between the view port X and Y plus 1 tile. After that I create a new instance for the destination Rectangle member variable.

The call to SpriteBatch.Begin is interesting. I use SpriteSortMode.Deferred because I am drawing a lot of images within the batch. BlendState.AlphaBlend allows for tiles that have transparency with in them. This include fully transparent and partially transparent, if the exported image supports it. SamplerState.PointClamp is used to prevent strange behaviour when drawing the map. This includes line around the tiles when scrolling the map. This SamplerState is always recommended when using source rectangles when drawing.

Next are the next for loops that actually render the layer. The tiles are drawn from the top left corner across the screen then down to the bottom right corner. You can experiment with different loops to see how it affects the way the layer is rendered.

Outside of the inner loop I set the Y coordinate of the destination rectangle as it remains the same for the entire row. Inside the loop I call GetTile passing in the X and Y coordinates of the tile. If the tile is -1 I just move to the next iteration of the inner loop. I then set the X coordinate of the tile. Finally I draw the tile using the overload that requires a texture, source rectangle, destination rectangle and tint color.

That, at a very high level, is how a tile layer is rendered. There are likely a few more optimizations that could be made such as accessing the array directly rather than relying on GetTile. I just used that method to make sure that I calculate the tile in the one dimensional array correctly.

Now I'm going to add a class that represents the entire map for the game. Right click the TileEngine folder, select Add and then Class. Name this new class TileMap. Add the following code to the TileMap class.

```
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;

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

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

        [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;
        }

        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);
        }

        #endregion
    }
}
```

The class is pretty simple. It is has a member variable for the name of the map, member variables for the layers for the map, the height and width of the map along with a Dictionary<string, Point> that holds the names of NPCs on the map and their coordinates, in tiles. I will be getting to NPCs in a future tutorial. There is also a member variable for the tile set that is used for drawing the map.

There is a property for each of the member variables that exposes them to other classes. Most of these are public getters with private setters. They are also marked so that they will be serialized using the IntermediateSerializer.

There are two public get only properties that expose the width of the map in pixels and the height of the map in pixels. They are used in different places to determine if an object is inside the map or outside the map.

There are three constructors for this class. The first is a private constructor with no parameters. The no parameter constructor is required for deserializing objects. The second constructor takes as parameters the tile set for the map and the name of the map. It just initializes the character dictionary and member variables with the values passed in.

The third takes those parameters as well as four layers. It calls the two parameter constructor so that it will initialize those member variables rather than replicate the code in this constructor. It then sets the width and height members using the height and width of the ground layer.

There is a GetTile and SetTile method for each of the layers. They just provide the functionality to update the layers without exposing the layers themselves. This is done to promote good object-oriented programming. The user of the class does not need to know about the objects with in the class to work with them. They instead use the public interface that is exposed. I don't use the term as the keyword interface. I use it to mean a contract that is published to user so they can interact with the class.

There are also Fill methods for the layers that fill the building, edge and decoration layers with -1. This is useful when creating maps in the editor. I will be providing the editor as part of the series but will not be writing tutorials on how to create it.

Finally, the Update and Draw methods check to make sure the layers are not null and then call the Update and Draw method of that layer. We should probably be a little more diligent when calling the other methods that will through a null value exception when the layer does not have a value. I will leave that as an exercise for you to implement in your game.

Now I'm going to implement the the Engine class. So, right click the TileEngine folder, select Add and then Class. Name this new class Engine. Here is the code for that class.

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

namespace Avatars.TileEngine
{
    public class Engine
    {
        private static Rectangle viewPortRectangle;

        private static int tileWidth = 32;
        private static int tileHeight = 32;
        private TileMap map;

        private static float scrollSpeed = 500f;

        private static Camera camera;

        public static int TileWidth
        {
            get { return tileWidth; }
            set { tileWidth = value; }
        }

        public static int TileHeight
        {
            get { return tileHeight; }
            set { tileHeight = value; }
        }

        public TileMap Map
        {
            get { return map; }
        }

        public static Rectangle ViewportRectangle
        {
            get { return viewPortRectangle; }
            set { viewPortRectangle = value; }
        }


        public static Camera Camera
        {
            get { return camera; }
        }

        #region Constructors

        public Engine(Rectangle viewPort)
        {
            ViewportRectangle = viewPort;
            camera = new Camera();

            TileWidth = 64;
            TileHeight = 64;
        }

        public Engine(Rectangle viewPort, int tileWidth, int tileHeight)
            : this(viewPort)
        {
            TileWidth = tileWidth;
            TileHeight = tileHeight;
        }

        #endregion

        #region Methods

        public static Point VectorToCell(Vector2 position)
        {
            return new Point((int)position.X / tileWidth, (int)position.Y / tileHeight);
        }

        public void SetMap(TileMap newMap)
        {
            if (newMap == null)
            {
                throw new ArgumentNullException("newMap");
            }

            map = newMap;
        }

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

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            Map.Draw(gameTime, spriteBatch, camera);
        }

        #endregion
    }
}
```

For this I'm going on the premise that there is only one map and one instance of Engine at a time. For that reason I've included some public static members to expose values to other classes outside of the tile engine.

The first three member variables hold a Rectangle that describes the view port, the width of tiles on the screen and the height of tiles on the screen. The next member holds the map that is currently in use. The last two member variables hold scrollSpeed that determines how fast the map scrolls. You'll think that 500 pixels is really fast. This is just a multiplier though and not an actual speed. It is not used in this tutorial but will be in a future tutorial so I've left it in.

Static properties expose the tile width and height, the view port rectangle and the camera. There is also a property that exposes the map as well.

There are two constructors for this class. The first accepts a Rectangle that represents the visible area of the screen the map will be drawn to. It sets that value to the appropriate member variable and then sets the tile width and height on the screen to be 64 pixels.

The second constructor takes the same Rectangle for the screen space but also takes the width and height of the tiles on the screen. It calls the first constructor to set the view port and then sets the tile width and tile height member variables. That is done simply by dividing the X value by the tile width and the Y value by the tile height.

Next you will find the public static method VectorToCell that you saw earlier that takes a vector which represents a point on the map measured in pixels that returns what tile the pixel is in.

There is also a method called SetMap that accepts a TileMap parameter. It checks to see if it is null and if it is throws an exception as the map cannot be null. If it isn't null it sets the map member variable to be the map passed in.

The last two methods are the Update and Draw methods. The Update method takes a GameTime parameter and just calls the Update method of the map. The Draw method takes GameTime as well as a SpriteBatch object that will be used to render the map.

That wraps up the tile engine. It is rather plain but it will get our job done nicely. The last thing that I'm going to add today is a basic game play state. It won't do much as the tutorial is already pretty long but I will pick up with it in the next tutorial.

Now, right click the GameStates folder, select Add and then Class. Name this new class GamePlayState. Here is the code for that state.

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

namespace Avatars.GameStates
{
    public interface IGamePlayState
    {

    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);

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

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

        protected override void LoadContent()
        {
        }

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

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

It is a pretty stripped down version of a game state. I included the interface that this class will implement. Currently there is nothing in the interface but that will most definitely change in this for this class. I next included an Engine member variable that I initialize using the static ScreenRectangle property that is exposed by the Game1 class.

The constructor just registers this instance of the GamePlayState as a service using the interface that was included at the start of the class. Currently it does not do anything but will be required in the near future so I made sure to keep it in.

The constructor then registers the instance as a service with the game that can be retrieved as needed by other states. I also added method stubs for the main DrawableGameComponent methods, Initialize, Load, Update and Draw.

I'm going to end the tutorial here because we've covered a lot in this one. In the next tutorial I will wire the game play state to open from the main menu state and create an render a map. Please stay tuned for the next tutorial in this series. If you don't want to have to keep checking for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news for Game Programming Adventures

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 4

Exploring the Map

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.

In this tutorial I'm going to be working on the game play state that we added in the last tutorial. First, I'm going to cover wiring the game so that we can change from the menu state to the game state. I will then have the game play state render a map that we will code on the fly. Once the map is rendering I will add in scrolling the map.

To get started open up the Game1 class. Replace the fields, properties and constructor with the following.

```
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GameStateManager gameStateManager;

        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;
        IGamePlayState gamePlayState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        public IGamePlayState GamePlayState
        {
            get { return gamePlayState; }
        }

        public 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);
        }
```

The first change is that I've added a field for the game play state. I then added a property that exposes the IGamePlayState interface. Finally, in the constructor I create the state.

I'm going to make a few tweeks to the game play state. Open the GamePlayState file and update it to the following code.

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

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

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);
        TileMap map;
        Camera camera;
        public GamePlayState(Game game)
            : base(game)
        {
            game.Services.AddService(typeof(IGamePlayState), this);
        }

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

        protected override void LoadContent()
        {
        }

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

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

        public void SetUpNewGame()
        {
        }

        public void LoadExistingGame()
        {
        }

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

First, I added some using statements to bring a few of MonoGame/XNA classes into scope in this class. I then updated the interface to include three method signatures, SetUpNewGame, LoadExistingGame and StartGame. The first two are called to create a new game or load an existing game. The third will be called once the other two have finished to start a game. I added in two new fields, map and camera, for a TileMap and Camera respectively. I also implemented the interface method, SetUpNewGame, LoadExistingGame and StartGame.

Next, I'm going to add the transition from the menu to the game play. Open the MainMenuState file and replace the Update method with the following version.

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter) ||
(menuComponent.MouseOver && Xin.CheckMouseReleased(MouseButtons.Left)))
            {
                if (menuComponent.SelectedIndex == 0)
                {
                    Xin.FlushInput();

                    GameRef.GamePlayState.SetUpNewGame();
                    GameRef.GamePlayState.StartGame();
                    manager.PushState((GamePlayState)GameRef.GamePlayState,
PlayerIndexInControl);
                }
                else if (menuComponent.SelectedIndex == 1)
                {
                    Xin.FlushInput();

                    GameRef.GamePlayState.LoadExistingGame();
                    GameRef.GamePlayState.StartGame();
                    manager.PushState((GamePlayState)GameRef.GamePlayState,
PlayerIndexInControl);
                }
                else if (menuComponent.SelectedIndex == 2)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 3)
                {
                    Game.Exit();
                }
            }

            base.Update(gameTime);
        }
```

What changed is I updated the if statements where the selected index is 0 or if the selected index is 1. Since 0 is the new game option I call the SetUpNewGame and StartGame methods on the game play state. I then push the game play state onto the game state manager. Similarly, for the other state I call LoadExistingGame instead of SetUpNewGame. Otherwise, the code is the same for both.

What we are going to need next is a tile set. I've uploaded one to my site here. I ended up merging two sets into one. The building tiles were grabbed from a public domain tile set that I found on OpenGameArt.org here, http://opengameart.org/content/town-tiles. I recommend that you visit this site where your developing games. You can find some awesome art work with various licenses. You can even some of the assets as placeholders until you find exactly what it is you are looking for.

Once you've downloaded to the tile set open the content manager. First add a new folder to the content folder called Tiles. Now select the Tiles folder and select Add Existing Item. Navigate to the tileset1.png file and add it. Before closing the content manager make sure that you rebuild the content.

Now, go back to the GamePlayState.cs file in the solution. Update the Draw and SetUpNewGame methods as follows.

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

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

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

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

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

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

            camera = new Camera();
        }
```

In the Draw method I check if the map and camera parameters are both not null because they are required to draw the map. If they are not null I call the Draw method of the TileMap class passing in the GameTime parameter to this Draw method, the map member variable and camera member variable.

In the SetUpNewGame method I load the image for the tile set into the tiles variable. I then create a new tile set using the parameters 8, 8, 32 and 32 because the tile set is 8 tiles width, 8 tiles high with a tile width and height of 32 pixels. A quick note here. When you are creating assets like this it is best if your images have the same height and width with the same dimensions and they are a power of 2. This allows for loading on the GPU which speeds up rendering. After creating the tile set I assign the Texture property the tile set that I just loaded.

Next I create the four layers each map has with the same height and width. I then create a map object using the tile set, the four layers and call it test-map. After that I call the Fill methods to set all of the tiles for the layers above the background tiles to -1 so nothing will be drawn for those layers. Since we need a camera to draw a map I create a new instance of the camera as well.

If you build and run the game you will be shown the title screen. Dismissing the title screen will bring you to the menu. Now you can select the New Game option to be taken to the game play screen with a field of grass tiles being drawn. Since the destination tiles are bigger than the source tiles there is a bit of distortion in the rendered tiles. You can fix that by changing the engine to have width and height of 32 instead of 64.

Next I'm going to tackle scrolling the map. Still in the GamePlayState add the following using statement to bring the input manager into scope and and replace the Update method with the following.

```
using Avatars.Components;

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

            if (Xin.KeyboardState.IsKeyDown(Keys.W) && Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.A))
```

```
            {
                motion.X = -1;
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W))
            {
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S))
            {
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
            }

            if (motion != Vector2.Zero)
            {
                motion *= camera.Speed;
                camera.Position += motion;
                camera.LockCamera(map, Game1.ScreenRectangle);
            }

            base.Update(gameTime);
        }
```

I have a local variable, motion, that will hold what direction the player wants to try and move the map. I've implemented moving the map using the W, A, S and D keys. It allows for scrolling left, right, up, down and diagonals. That is why there are a series of if and else if statements, one for each of the directions. Before I cover the if statements a brief introduction to screen space. Screen space starts with the coordinates (0, 0) in the upper right corner increasing going down and right. To move up you subtract from the Y coordinate and add to the Y coordinate to move down. Similarly, to move left you subtract from the X coordinate and add to the X coordinate to move right.

The first direction that I check is up and left, the W and A keys. If that is true I set the X and Y value of motion to -1 and -1. Next up is W and D keys which is up and right so the X value 1 and the Y value is -1. Now down to the right is A and S with X set to -1 and Y set to 1. The last diagonal is S with D and the X and Y values of 1 and 1. For the cardinal directions: up, down, left and right. For those directions I check the W, S, A and D keys respectively. For W and S the X values are 0 and the Y values are -1 and 1 respectively. Similarly, A and D the Y values are 0 and the X values are -1 and 1 respectively.

Next there is an if statement that checks to see if the motion vector is not the zero vector. If it is not I multiply the motion vector by the camera speed and then add it to the position of the camera. Finally I call LockCamera method of the camera passing in the map object and the ScreenRectangle static property from the Game1 class.

If you build and run the game now once you reach the game play screen you will be able to move the map in all directions and the map will not scroll off the screen. You are going to notice a strange behavior on the diagonals. The map scrolls faster than in the cardinal directions. Why is that?

It has to do with vectors and their magnitudes. A two dimensional vector's magnitude is calculated by taking the square root of the sum of the squares, SQRT(X^2 + Y ^2). For a cardinal vector it will always evaluate to 1. For one of the diagonal vectors it evaluations as SQRT(2) which is greater than 1 so the map scrolls faster. How do you resolve this as the map should move at the same speed in all directions. The answer is you normalize the vector. What that means is the vector will still have the same direction but the magnitude of the vector will be 1. Fortunately the Vector2 class gives us a Normalize method and takes care of the for us. Update the if statement where I check for motion as follows.

```
            if (motion != Vector2.Zero)
            {
                motion.Normalize();
                motion *= camera.Speed;
                camera.Position += motion;
                camera.LockCamera(map, Game1.ScreenRectangle);
            }
```

So, the reason I check that the motion vector is not the zero vector is that the zero vector has no magnitude and you will be dividing by zero in the calculation and causing an exception to be thrown.

I'm going to end the tutorial here though. That is because we made good progress but the topics I want to cover now are fairly long and time consuming. I'd like to make sure that in each tutorial there is clearly defined progress now and there is something demonstrable at the end of each one.

In the next tutorial I'm going to add a component that will represent the player in the game with an animated sprite for the player. Please stay tuned for the next tutorial in this series. If you don't want to have to keep visiting the site to check for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news from Game Programming Adventures.

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 5

Player Component

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called A Summoner's Tale. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my web site: A Summoner's Tale. The source code for each tutorial will be available as well. I will be using Visual Studio 2013 Premium for the series. The code should compile on the 2013 Express version and Visual Studio 2015 versions as well.

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

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

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

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

        #endregion

        #region Property Region

        public int FramesPerSecond
        {
            get { return framesPerSecond; }
            set
            {
                if (value < 1)
                    framesPerSecond = 1;
                else if (value > 60)
                    framesPerSecond = 60;
                else
                    framesPerSecond = value;
                frameLength = TimeSpan.FromSeconds(1 / (double)framesPerSecond);
            }
        }

        public Rectangle CurrentFrameRect
        {
            get { return frames[currentFrame]; }
        }

        public int CurrentFrame
        {
            get { return currentFrame; }
            set
            {
                currentFrame = (int)MathHelper.Clamp(value, 0, frames.Length - 1);
            }
        }

        public int FrameWidth
        {
            get { return frameWidth; }
        }

        public int FrameHeight
        {
            get { return frameHeight; }
        }

        #endregion

        #region Constructor Region

        public Animation(int frameCount, int frameWidth, int frameHeight, int xOffset, int
yOffset)
        {
            frames = new Rectangle[frameCount];
            this.frameWidth = frameWidth;
            this.frameHeight = frameHeight;

            for (int i = 0; i < frameCount; i++)
            {
                frames[i] = new Rectangle(
                        xOffset + (frameWidth * i),
                        yOffset,
                        frameWidth,
                        frameHeight);
            }
            FramesPerSecond = 5;
            Reset();
        }

        private Animation(Animation animation)
        {
            this.frames = animation.frames;
            FramesPerSecond = 5;
        }

        #endregion
        #region Method Region

        public void Update(GameTime gameTime)
        {
            frameTimer += gameTime.ElapsedGameTime;

            if (frameTimer >= frameLength)
            {
                frameTimer = TimeSpan.Zero;
                currentFrame = (currentFrame + 1) % frames.Length;
            }
        }

        public void Reset()
        {
            currentFrame = 0;
            frameTimer = TimeSpan.Zero;
        }

        #endregion

        #region Interface Method Region

        public object Clone()
        {
            Animation animationClone = new Animation(this);

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

            return animationClone;
        }

        #endregion
    }
}
```

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

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

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

I then have a property that returns what the current rectangle for the animation is. There is also a property for the current frame of the animation. The getter returns the frame while the setter clamps the value between 0 and the number of frames minus 1 because arrays are zero based. Next there are two get only properties for returning the width and height of the frames.

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

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

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

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

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

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

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

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

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

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

    public class AnimatedSprite
    {
        #region Field Region

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

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

        #endregion

        #region Property Region

        public bool IsActive { get; set; }

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

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

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

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

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

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

        #endregion

        #region Constructor Region

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

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

        #endregion

        #region Method Region

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

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

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

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

        #endregion
    }
}
```

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

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

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

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

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

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

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

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

        GameStateManager gameStateManager;
        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;
        IGamePlayState gamePlayState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        public IGamePlayState GamePlayState
        {
            get { return gamePlayState; }
        }

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

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

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

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

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

            this.IsMouseVisible = true;

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

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

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

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

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

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

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


            base.Initialize();
        }

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

        }

        protected override void UnloadContent()
        {
        }

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

            base.Update(gameTime);
        }

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

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

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

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

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

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

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

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

        private Vector2 position;

        #endregion

        #region Property Region

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

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

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

        #endregion

        #region Constructor Region

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

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

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

        #endregion

        #region Method Region

        public void SavePlayer()
        {
        }

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

            return player;
        }

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

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

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

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

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

        #endregion
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

            if (motion != Vector2.Zero)
            {
                motion.Normalize();
                motion *= (player.Speed * (float)gameTime.ElapsedGameTime.TotalSeconds);
                Vector2 newPosition = player.Sprite.Position + motion;

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

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

            base.Update(gameTime);
        }

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

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

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

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

            GameRef.SpriteBatch.End();
        }

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

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

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

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

            camera = new Camera();
        }

        public void LoadExistingGame()
        {
        }

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

The first change I made is that I added a using statement to bring the player component into scope in this class. I then created a Player field to hold the player object. In the LoadContent method I loaded the sprite sheet for the player and create a new male player named Wesley.

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

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

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

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

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

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

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

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

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 6

Avatars

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.

The key component for this game will be the avatars. Without the avatars the player is just wandering the map interacting with characters. There is no excitement for the player. Just as a refresher, I have substituted what I call avatars for Pokemon. In essence they are basically the same though. You find avatars, battle them against other avatars to increase your their power and level. They are elementally aligned just like in Pokemon as well with different attributes and moves. The difference is that you learn spells instead of capturing them. I will include a side tutorial on how you can change the game to capture avatars instead of learning to summon them from other characters or scrolls.

Since they are so integral to the game I am going to implement them now. So, right click the Avatars project, select Add and then New Folder. Name this new folder AvatarComponents. Avatars have moves that they use when battling other avatars. For that reason I'm going to add in an interface for moves. All moves will implement this interface. That allows us to have a collection of moves in the avatar component. Right click the AvatarComponents folder, select Add and then Interface. Name this interface IMove. Here is the code for this interface.

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

namespace Avatars.AvatarComponents
{
    public enum Target
    {
        Self, Enemy
    }

    public enum MoveType
    {
        Attack, Heal, Buff, Debuff, Status
    }

    public enum Status
    {
        Normal, Sleep, Poison, Paralysis
    }

    public enum MoveElement
    {
        None, Dark, Earth, Fire, Light, Water, Wind
    }

    public interface IMove
    {
        string Name { get; }
        Target Target { get; }
        MoveType MoveType { get; }
        MoveElement MoveElement { get; }
        Status Status { get; }
        int UnlockedAt { get; set; }
        bool Unlocked { get; }
        int Duration { get; set; }
        int Attack { get; }
        int Defense { get; }
        int Speed { get; }
        int Health { get; }
        void Unlock();
        object Clone();
    }
}
```

First, a move targets something. It can either target the enemy/opponent or it can target that avatar so there is an enumeration that defines this. Next is an enumeration that defines what type of move this is. For example, is it an attack, a buff for your avatar or a debuff on the enemy avatar. This can be extended if you want to include a different type of move like Abra's escape move from Pokemon. A move can affect the enemy avatar's status so I included an enumeration for that. As well I included an enumeration for what element the move is for. A basic move like a tackle has no element where as a fire based or water based attack does.

I use enumerations for these sorts of things because I don't want a list of strings or numbers to represent things like this. Also, since I'm using an enumeration it is easy to move them to classes and have a static property that returns the value.

Next is the interface that defines the different properties/methods that a move must implement. These are all common elements for the move. Moves must have a name so I included a property for that. They also require a target using the Target enumeration, a move type using the MoveType enumeration, an element using MoveElement enumeration and a status it can apply using the Status enumeration.

Moves will unlock at different levels so I included a property that exposes that as well as if the move has been learned/unlocked. A move may have a duration so I included that as a property. A move can affect the base attributes of an avatar which are Attack, Defence, Speed and Health. You can of course add other attributes but this is a good base. I then have a method that will be called to unlock/learn the move and a method to clone a move. I add clone so that I can create a master list of moves and just clone them and add them to the player's or opponent's avatars. It is always a good idea when you are using game objects that you might want multiple of to have a way to easily make a copy of that game object. For that reason you will find that I use Clone a lot in my games.

Next, I will add the class for avatars. Right click the AvatarComponents folder, select Add and then Class. Name this new class Avatar. This class contains a lot of code so I'm going to give it to you in pieces and explain it bit by bit rather than the whole things. First, these are the using statements that I used to bring required classes into scope.

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

I didn't implement avatars as content items in my demo that I build so I included the System.IO namespace so that I can easily read/write to and from disk. I also brought in some of the XNA/MonoGame names spaces into scope.

Next up is an enumeration that defines the elements and avatar can have. Outside of the Avatar class add the following enumeration.

```
    public enum AvatarElement
    {
        Dark, Earth, Fire, Light, Water, Wind
    }
```

There are a number of fields required for an avatar. Inside of the Avatar class add the following fields.

```
        #region Field Region

        private static Random random = new Random();
        private Texture2D texture;
        private string name;
        private AvatarElement element;
        private int level;
        private long experience;
        private int costToBuy;
        private int speed;
        private int attack;
        private int defense;
        private int health;
        private int currentHealth;
        private List<IMove> effects;
        private Dictionary<string, IMove> knownMoves;

        #endregion
```

I included a static Random field in this class for random number generation for avatars. In my demo I just used a texture for the avatars so I included a Texture2D field for that. Avatars have a name so their is a field for that. They also have an element so that was added as well. They have a level and experience gained so that is there as well. They can be bought so I included a cost to buy field. Their base attributes are speed, attack, defense and health so there are fields for that. Health represents their maximum health so I added a field for their current health. The next is a List<IMove> called effects. If you look back at IMove I included status effects as well as buffs/debuffs for avatars. When one of these moves is applied to an avatar I add that move to the effects list. When it no longer affects the avatar is removed from the list. I also include a Dictionary<string, IMove> that holds the moves known by the avatar. Here I will be deviating a bit from Pokemen in that an avatar can know more than 4 moves and does not have to forget an old move to learn a new move. I will include a side tutorial that explains how to implement the learning/forgetting moves if you want to stay more true to Pokemon.

These fields need to be exposed to other classes so I had to add a number of properties. Add the following properties just below the fields inside the Avatar class.

```
#region Property Region

public string Name
{
    get { return name; }
}

public int Level
{
    get { return level; }
    set { level = (int)MathHelper.Clamp(value, 1, 100); }
}

public long Experience
{
    get { return experience; }
}

public Texture2D Texture
{
    get { return texture; }
}

public Dictionary<string, IMove> KnownMoves
{
    get { return knownMoves; }
}

public AvatarElement Element
{
    get { return element; }
}

public List<IMove> Effects
{
    get { return effects; }
}

public static Random Random
{
    get { return random; }
}

public int BaseAttack
{
    get { return attack; }
}

public int BaseDefense
{
    get { return defense; }
}

public int BaseSpeed
{
    get { return speed; }
}
```

```
public int BaseHealth
{
    get { return health; }
}

public int CurrentHealth
{
    get { return currentHealth; }
}

public bool Alive
{
    get { return (currentHealth > 0); }
}

#endregion
```

Other than Level these are all read only properties. I included a set for Level because I was capping the avatar's level at 100. Typically I would have mad the set private so that it could only be adjusted inside of the class. The other interesting thing is that instead of naming the properties for Attack, Defense, Speed and Health I place Base before each of them. That is because the effects field can possibly affect the avatar's attributes. I also included an Alive field that that can be used to check if the avatar's health is less than one an is either unconscious or defeated.

I added a private constructor to this class. It just sets the level to 1 and initializes the dictionary and list for moves. Add the following constructor below the properties.

```
#region Constructor Region

private Avatar()
{
    level = 1;
    knownMoves = new Dictionary<string, IMove>();
    effects = new List<IMove>();
}

#endregion
```

The next thing I'm going to add is the code for resolving a move. Below the constructor add the following method.

```
        public void ResoleveMove(IMove move, Avatar target)
        {
            bool found = false;
            switch (move.Target)
            {
                case Target.Self:
                    if (move.MoveType == MoveType.Buff)
                    {
                        found = false;
                        for (int i = 0; i < effects.Count; i++)
                        {
                            if (effects[i].Name == move.Name)
                            {
                                effects[i].Duration += move.Duration;
                                found = true;
                            }
                        }

                        if (!found)
                            effects.Add((IMove)move.Clone());
                    }
                    else if (move.MoveType == MoveType.Heal)
                    {
                        currentHealth += move.Health;
                        if (currentHealth > health)
                            currentHealth = health;
                    }
                    else if (move.MoveType == MoveType.Status)
                    {
                    }

                    break;
                case Target.Enemy:
                    if (move.MoveType == MoveType.Debuff)
                    {
                        found = false;
                        for (int i = 0; i < target.Effects.Count; i++)
                        {
                            if (target.Effects[i].Name == move.Name)
                            {
                                target.Effects[i].Duration += move.Duration;
                                found = true;
                            }
                        }

                        if (!found)
                            target.Effects.Add((IMove)move.Clone());
                    }
                    else if (move.MoveType == MoveType.Attack)
                    {
                        float modifier = GetMoveModifier(move.MoveElement, target.Element);

                        float tDamage = GetAttack() + move.Health * modifier -
target.GetDefense();

                        if (tDamage < 1f)
                            tDamage = 1f;

                        target.ApplyDamage((int)tDamage);
                    }

                    break;
            }
        }
```

The method accepts an IMove parameter for the move to be applied and an Avatar parameter for the target. The local variable found is used to look to see if an existing effect is found or not. There is then a switch on the target for the move. This is part of the reason why I created an interface for moves. With the interface I have all the information needed to resolve the move without knowing anything about the move being applied. I just use the contract that was defined to apply the move.

The first case that I check in the switch is if the target is Self, or the current avatar being used. I then check to see if the move type is Buff, which increases the avatar's attribute in some way. If it is I set the found variable to false. I then loop through all of the active effects that have been added to the avatar. If I find a move that has the same name I increase the current duration of the effect that is being applied to current duration. Instead of this you might want to replace the current duration with the duration for the move. It is totally up to you how you want to apply this.

I then check if the move was found or not. If it wasn't found I add a clone of the move to the list of effects currently applied to the avatar. If the move is of type Heal I increase the avatar's current health by the health modifier for the move. Then if current health is above the maximum health I set it to the maximum health. I included a check for the status of the move but didn't implement any code yet. That is because I was considering adding in more status types in my demo but never made it that far. For example, I could have included a status Invulnerable where the avatar could not be harmed by physical attacks.

Next I handle the enemy case, which is very similar to the self case. The biggest difference is that the move is applied to the enemy. It doesn't make sense to buff an enemy so I include the debuff case. This works the same as the buff case for self. I search to see if that is already there. If it is not there I add it to the list. For attacking I first call a method GetMoveModifier passing in the element for the move and the element of the target. This is where you apply logic for fire moves being strong against grass moves. I will get to that method shortly. I then call a method GetAttack that returns the avatar's attack value adding in any buffs or debuffs that have been applied. I add that to move.Health times the modifier which returns how much damage the moved does I then subtract the target's defense attribute. I then check if the damage done is less than 1 and if it is I set it to 1 because I implemented the rule that move always does 1 damage. You can also add in here if a move misses by including accuracy and such. I then call a method ApplyDamage on the target which will apply the damage. That method is yet to come.

The next method that I want to add is the one that gets if a move is effective against a certain type of avatar or not effective. Add the following method to the class.

```
public static float GetMoveModifier(MoveElement moveElement, AvatarElement avatarElement)
{
    float modifier = 1f;

    switch (moveElement)
    {
        case MoveElement.Dark:
            if (avatarElement == AvatarElement.Light)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Wind)
                modifier -= .25f;
            break;
        case MoveElement.Earth:
            if (avatarElement == AvatarElement.Water)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Wind)
                modifier -= .25f;
            break;
        case MoveElement.Fire:
            if (avatarElement == AvatarElement.Wind)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Water)
                modifier -= .25f;
            break;
        case MoveElement.Light:
            if (avatarElement == AvatarElement.Dark)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Earth)
                modifier -= .25f;
            break;
        case MoveElement.Water:
            if (avatarElement == AvatarElement.Fire)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Earth)
                modifier -= .25f;
            break;
        case MoveElement.Wind:
            if (avatarElement == AvatarElement.Light)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Earth)
                modifier -= .25f;
            break;

    }

    return modifier;
}
```

This method is really just a look up table. It takes the element of the move type and compares it to the avatar's type. What this does is first sets a local variable modifier to 1f, or normal damage. In each of the cases I first check to see if the move is effective. If it is effect the move will do 25% more damage than the base so I add .25f to the modifier. If it is not effective it will to 25% less damage so I subtract .25 from the modifier. For example, a Dark move is strong against a Light avatar and does 125% of its normal damage to a Light type avatar. Similarly, it is weak against a Wind avatar and does 75% of its normal damage. All of the other cases are similar where an element is strong against one type and weak against another. In this look up table you can add a lot more cases though, and elements. This was just a good start for my demo game.

I'm going to now add two small methods. The one that applies damage and one that updates the avatar each round of combat. Add the following two methods to the class.

```
public void ApplyDamage(int tDamage)
{
    currentHealth -= tDamage;
}

public void Update(GameTime gameTime)
{
    for (int i = 0; i < effects.Count; i++)
    {
        effects[i].Duration--;

        if (effects[i].Duration < 1)
        {
            effects.RemoveAt(i);
            i--;
        }
    }
}
```

Apply damage is trivial. It just reduces the avatar's currentHealth field by the damage being passed in. Update loops through the activate effects and reduces the Duration field by 1. If the duration is less than zero I remove the effect and decrease the loop variable by 1. I do that because there is one less element in the list.

What I am going to add next are four get method that get the current attack, defense, speed and maximum health of an avatar. They all work in the same way. They loop through each of the active effects. If the effect is a buff it adds the buff to the attribute and if it is a debuff it subtracts it. It then returns the attribute plus the modifier. Add the following four methods to the class after Update.

```
public int GetAttack()
{
    int attackMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
```

```
            attackMod += move.Attack;

        if (move.MoveType == MoveType.Debuff)
            attackMod -= move.Attack;
    }

    return attack + attackMod;
}

public int GetDefense()
{
    int defenseMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
            defenseMod += move.Defense;

        if (move.MoveType == MoveType.Debuff)
            defenseMod -= move.Defense;
    }

    return defense + defenseMod;
}

public int GetSpeed()
{
    int speedMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
            speedMod += move.Speed;
        if (move.MoveType == MoveType.Debuff)
            speedMod -= move.Speed;
    }

    return speed + speedMod;
}

public int GetHealth()
{
    int healthMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
            healthMod += move.Health;
        if (move.MoveType == MoveType.Debuff)
            healthMod += move.Health;
    }

    return health + healthMod;
}
```

I'm going to add in a couple more methods now. One that will be called when combat starts. One that will be called when an avatar wins a battle and one with it loses a battle. Finally one that will be called to check if the avatar has levelled up. Add these methods after the get methods.

```
public void StartCombat()
{
    effects.Clear();
    currentHealth = health;
}
```

```
public long WinBattle(Avatar target)
{
    int levelDiff = target.Level - level;
    long expGained = 0;

    if (levelDiff <= -10)
    {
        expGained = 10;
    }
    else if (levelDiff <= -5)
    {
        expGained = (long)(100f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 0)
    {
        expGained = (long)(50f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 5)
    {
        expGained = (long)(5f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 10)
    {
        expGained = (long)(10f * (float)Math.Pow(2, levelDiff));
    }
    else
    {
        expGained = (long)(50f * (float)Math.Pow(2, target.Level));
    }

    return expGained;
}

public long LoseBattle(Avatar target)
{
    return (long)((float)WinBattle(target) * .5f);
}

public bool CheckLevelUp()
{
    bool leveled = false;

    if (experience >= 50 * (1 + (long)Math.Pow(level, 2.5)))
    {
        leveled = true;
        level++;
    }

    return leveled;
}
```

In my game after each battle the avatar returns to its element plane. There it instantly heals all damage and all status effects are removed. So, I clear the effects and reset the health in the StartCombat method. In WinBattle I decide how much experience the avatar gains for winning the battle. I first determine the level difference between the two avatars. If the player's avatar is more than ten levels higher than the opponent's avatar it gains 10 experience. If it is between 9 and 5 levels higher I use a formula to get an experience value. The way the formula works is using 2 to the power of the level difference. In this case the level difference is negative so it will return a fraction of the base. The next case is for between 0 and 4 levels higher. Again if it is negative it will return a fraction and if it is zero it will return 1. The other cases are similar when the opposing avatar was a higher level that the player's avatar. Technically the player should never win if the opposing avatar is more than 10 levels but I included it is a catch all. I know that in some games that if the opponent is that much higher you gain zero experience for defeating it. If the player does lose a battle I still reward the avatar half the experience for losing the battle. This would actually be a very high value is they lost against a much stronger avatar and will need to be tweeked accordingly. The other method that I added is called CheckLevelUp and it checks to see if the avatar has levelled up or not. I use another formula for that that uses a power variable again. This makes it so that as the avatar's level grows the experience needed to grow increase as well. This seemed to work okay in my demo but may need to be tweeked a bit in a real game.

I also included a method for levelling up an avatar. Rather than automatically adjusting the attributes like in Pokemon I allow the players to assign points to the attribute of their choice. There is a switch that checks which attribute has been chosen and updates that attribute. If they choose health I multiple that value by 5. Here is the code for that method.

```
public void AssignPoint(string s, int p)
{
    switch (s)
    {
        case "Attack":
            attack += p;
            break;
        case "Defense":
            defense += p;
            break;
        case "Speed":
            speed += p;
            break;
        case "Health":
            health += p * 5;
            break;
    }
}
```

The last method that I'm going to add is a Clone method. This can be used to grab a copy of the avatar from a master list of avatars when the player finds a new avatar. Here is a code for that method.

```
public object Clone()
{
    Avatar avatar = new Avatar();

    avatar.name = this.name;
    avatar.texture = this.texture;
    avatar.element = this.element;
    avatar.costToBuy = this.costToBuy;
    avatar.level = this.level;
    avatar.experience = this.experience;
    avatar.attack = this.attack;
    avatar.defense = this.defense;
    avatar.speed = this.speed;
    avatar.health = this.health;
    avatar.currentHealth = this.health;

    foreach (string s in this.knownMoves.Keys)
    {
        avatar.knownMoves.Add(s, this.knownMoves[s]);
    }

    return avatar;
}
```

Nothing hard about this method. It just sets the fields with the values from the current instance. To do the moves I use a foreach loop that loops over the list. I then return the new object.

I'm going to end this here as we've covered a lot and it is a very important component. I had wanted to implement some game play in this tutorial but most elements that I wanted to include use this component. For example, most NPCs have an avatar associated with them or work with avatars in some way, such as giving them to the player, training them or something similar. For battles they are definitely required. The next tutorial I will be adding in NPCs to the game and some of the components required for having conversations with NPCs.

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

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 8

Conversations

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.

This tutorial is about attaching conversations to the characters that the player will with. At a very high level a conversation is a tree of nodes that can be traversed in different ways depending on the player's choices. In the game I called a node a GameScene. The scene contains the text to be displayed to the player and one or more SceneOptions. A SceneOption has a SceneAction which determines what action is taken if the player selects that action. A full conversation is made up a number of GameScenes.

So, let's get started. First, right click the Avatars project, select Add and then New Folder. Name this new folder ConversationComponents. Now right click the ConversationComponents folder, select Add and then class name this new class SceneOption. 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.ConversationComponents
{
    public enum ActionType
    {
        Talk,
        End,
        Change,
        Quest,
        Buy,
        Sell,
        GiveItems,
        GiveKey,
    }
    public class SceneAction
    {
        public ActionType Action;
        public string Parameter;
    }

    public class SceneOption
    {
        private string optionText;
        private string optionScene;
        private SceneAction optionAction;

        private SceneOption()
        {

        }

        public string OptionText
        {
            get { return optionText; }
            set { optionText = value; }
        }

        public string OptionScene
        {
            get { return optionScene; }
            set { optionScene = value; }
        }

        public SceneAction OptionAction
        {
            get { return optionAction; }
            set { optionAction = value; }
        }

        public SceneOption(string text, string scene, SceneAction action)
        {
            optionText = text;
            optionScene = scene;
            optionAction = action;
        }
    }
}
```

I added an enumeration called ActionType that defines what action to take when the player selects that option. Talk moves the conversation to another scene in the same conversation. End ends the conversation. Change changes the current conversation to another conversation. Quest gives the player a quest if the node is selected. The Buy and Sell options were added for speaking with shopkeepers. The GiveItem and GiveKey give an item or a key to the player. Next up is a really basic class, SceneAction. This holds the action to take and any parameters that will be used based on the action chosen.

The actual SceneOption class holds the details for the option. For that there are three private member variables. The optionText field holds what is drawn on the screen. Then optionScene is what scene to change to based on the option. Finally there is a SceneAction field that defines the action taken with any parameters. There are three public properties that expose these values to other classes. There is also a private constructor that takes no parameters that will be used for writing conversations and reading them back in. There is a second constructor that takes three parameters which are the text displayed, the scene being transitioned to and the action.

With the SceneOption in place I can now create the GameScene class. Right click the

ConversationComponents folder, select Add and then Class. Name this new class GameScene. This is the code for that class.

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

namespace Avatars.ConversationComponents
{
    public class GameScene
    {
        #region Field Region

        protected Game game;
        protected string text;
        private List<SceneOption> options;
        private int selectedIndex;
        private Color highLight;
        private Color normal;
        private Vector2 textPosition;
        private static Texture2D selected;
        private bool isMouseOver;

        private Vector2 menuPosition = new Vector2(50, 475);

        #endregion

        #region Property Region

        public string Text
        {
            get { return text; }
            set { text = value; }
        }

        public static Texture2D Selected
        {
            get { return selected; }
        }

        public List<SceneOption> Options
        {
            get { return options; }
            set { options = value; }
        }

        [ContentSerializerIgnore]
        public SceneAction OptionAction
        {
            get { return options[selectedIndex].OptionAction; }
        }

        public string OptionScene
        {
            get { return options[selectedIndex].OptionScene; }
        }

        public string OptionText
        {
            get { return options[selectedIndex].OptionText; }
        }

        public int SelectedIndex
        {
            get { return selectedIndex; }
        }

        public bool IsMouseOver
        {
            get { return isMouseOver; }
        }

        [ContentSerializerIgnore]
        public Color NormalColor
        {
            get { return normal; }
            set { normal = value; }
        }

        [ContentSerializerIgnore]
        public Color HighLightColor
        {
            get { return highLight; }
            set { highLight = value; }
        }

        public Vector2 MenuPosition
        {
            get { return menuPosition; }
        }

        #endregion

        #region Constructor Region

        private GameScene()
        {
            NormalColor = Color.Blue;
            HighLightColor = Color.Red;
        }

        public GameScene(string text, List<SceneOption> options)
        {
            this.text = text;
            this.options = options;
            textPosition = Vector2.Zero;
        }

        public GameScene(Game game, string text, List<SceneOption> options)
        {
            this.game = game;

            this.options = new List<SceneOption>();
            this.highLight = Color.Red;
            this.normal = Color.Black;

            this.options = options;
        }

        #endregion

        #region Method Region

        public void SetText(string text, SpriteFont font)
        {
            textPosition = new Vector2(450, 50);

            StringBuilder sb = new StringBuilder();
            float currentLength = 0f;

            if (font == null)
            {
                this.text = text;
                return;
            }

            string[] parts = text.Split(' ');

            foreach (string s in parts)
            {
                Vector2 size = font.MeasureString(s);

                if (currentLength + size.X < 500f)
                {
                    sb.Append(s);
                    sb.Append(" ");
                    currentLength += size.X;
                }
                else
                {
                    sb.Append("\n\r");
                    sb.Append(s);
                    sb.Append(" ");
                    currentLength = 0;
                }
            }

            this.text = sb.ToString();
        }

        public void Initialize()
        {
        }

        public void Update(GameTime gameTime, PlayerIndex index)
        {
            if (Xin.CheckKeyReleased(Keys.Down))
            {
                selectedIndex--;
                if (selectedIndex < 0)
                    selectedIndex = options.Count - 1;
            }
            else if (Xin.CheckKeyReleased(Keys.Down))
            {
                selectedIndex++;
                if (selectedIndex > options.Count - 1)
                    selectedIndex = 0;
            }
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Texture2D background,
SpriteFont font)
        {
            Vector2 selectedPosition = new Vector2();
            Color myColor;

            if (selected == null)
                selected = game.Content.Load<Texture2D>(@"Misc\selected");

            if (textPosition == Vector2.Zero)
```

```
                SetText(text, font);

            if (background != null)
                spriteBatch.Draw(background, Vector2.Zero, Color.White);

            spriteBatch.DrawString(font,
                text,
                textPosition,
                Color.White);

            Vector2 position = menuPosition;

            Rectangle optionRect = new Rectangle(0, (int)position.Y, 1280, 
font.LineSpacing);
            isMouseOver = false;

            for (int i = 0; i < options.Count; i++)
            {
                if (optionRect.Contains(Xin.MouseState.Position))
                {
                    selectedIndex = i;
                    isMouseOver = true;
                }

                if (i == SelectedIndex)
                {
                    myColor = HighLightColor;
                    selectedPosition.X = position.X - 35;
                    selectedPosition.Y = position.Y;

                    spriteBatch.Draw(selected, selectedPosition, Color.White);
                }
                else
                    myColor = NormalColor;

                spriteBatch.DrawString(font,
                    options[i].OptionText,
                    position,
                    myColor);

                position.Y += font.LineSpacing + 5;
                optionRect.Y += font.LineSpacing + 5;
            }
        }

        #endregion
    }
}
```

There is a lot going on in this class. I'll tackle member variables first. There is a Game type field that is the reference to the game. It is used for loading content using the content manager. Next is text and it is used in a method further on in the class so that the text for the scene wraps in the screen area. Next is a List<SceneOption> that is the options for the scene. The selectedIndex member is what scene option is currently selected. The two Color fields hold the color to draw unselected and selected options. There is also a Vector2 that controls where the scene text is rendered and a Texture2D selected that will be drawn beside the currently selected scene option. This one is static and will be shared by all instances of GameScene. The next field I included is isMouse over and will be used to test if the mouse is over an SceneOption. The last member variable is menuPosition and controls where the scene options are drawn.

Next are a number of properties to expose the member variables to other classes. The only thing out of the normal is that I've marked a few with attributes that define how the class is serialized using the IntermediateSerializer because I don't want some of the members serialized so that they are set at runtime rather than at build time.

Next up are the three constructors for this class. The first requires no parameters and is required to deserialize and load the exported XML content. The second is used in the editor to create scenes. The third is used in the game when creating scenes on the fly.

I added a method called SetText and it takes as parameters text and font. First, I set the position of where to draw the text. You will notice that the position is almost half way over to the right. This is because when I call Draw to draw the scene it accepted a Texture2D parameter called portrait that represented the portrait of the character the player is speaking to. I'm not implementing that for a while yet but I want it available if needed. After setting the position I create a StringBuilder that will be used to convert the single line of text to multiple lines of text. There is then a local variable, currentLength, that holds the length of the current line. I then check to make sure the font member variable is not null. It if is I just set the member variable to the parameter and exit the method.

I then use the Split method of the string class to split the string into parts on the space character. Next in a foreach loop I iterate over all of the parts. I then use MeasureString to determine the length of that word. If the length of the word is less than the maximum length of text on the screen I append the part to the string builder with a space and update the line length.

If the length is greater than the maximum length I append a carriage return, append the text and then append a space. I can do that because rendering text with DrawString allows for escape characters like \n and \r. I then reset currentLength to size.X. The last thing to do in this method is to set the text member variable to the string builder as a string.

Next there is an empty method, Initialize, that will be updated to initialize the scene if necessary. I included it now as I do use it my games an will be used in the future.

The Update method takes a GameTime parameter and a PlayerIndex parameter. The index parameter is the index of the current game pad. It can be excluded if you do not want to support game pads in your game. In the Update method I check to see if the player has requested to move the selected item up or down. I check if moving the item up or down exceeds the bounds of the list of options and if does I wrap to either the first or last item in the list.

The last thing to do is draw the scene. The Draw method takes a GameTime parameter, SpriteBatch parameter and a Texture2D for the background image and a SpriteFont to draw the text with. There are local variables that determine where to draw the selected item indicator, the speaker's portrait and the color to draw scene options with.

I check to see if selected texture is null. If it is null I load it. If textPosition is Vector2.Zero then the text has not been set so I set it. Next if the background texture is not null I draw the background.

After drawing the background I draw the speaker's text in white. You can include a property in the game scene for what color to draw the speaker's text in rather than hard coding it.

There is then a Vector2 local variable that I assign the position of the scene options to be drawn. I then create a rectangle based on the position that is the width of the screen. I then loop over all of the options for the player to choose from as a reply to the current scene. Inside that loop I check if the mouse is included in the rectangle. If it is I set the selectedIndex member variable to the current loop index. Next I check to see if the current loop index is the selectedIndex. If it is I draw the selected item texture to the left of that option and I set the local color variable to the highlight color. If it is not then the local color variable is set to the base option color. I then draw the scene option. At the end of the loop I update the Y value for the position to be the line spacing for the font plus 5 pixels.

Now I'm going to add in the class that represents a conversation. Right click the ConversationComponents folder, select Add and then Class. Name this new class Conversation. Here is the code for that class.

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

namespace Avatars.ConversationComponents
{
    public class Conversation
    {
        #region Field Region

        private string name;
        private string firstScene;
        private string currentScene;
        private Dictionary<string, GameScene> scenes;
        private string bsckgroundName;
        private Texture2D background;
        private string fontName;
        private SpriteFont spriteFont;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public string FirstScene
        {
            get { return firstScene; }
        }

        public GameScene CurrentScene
        {
            get { return scenes[currentScene]; }
        }

        public Dictionary<string, GameScene> Scenes
        {
            get { return scenes; }
        }

        public Texture2D Background
        {
            get { return background; }
        }

        public SpriteFont SpriteFont
        {
            get { return spriteFont; }
        }

        public string BackgroundName
        {
            get { return backgroundName; }
            set { backgroundName = value; }
        }

        public string FontName
        {
            get { return fontName; }
            set { fontName = value; }
        }

        #endregion

        #region Constructor Region

        public Conversation(string name, string firstScene, Texture2D background, SpriteFont 
font)
        {
            this.scenes = new Dictionary<string, GameScene>();
            this.name = name;
            this.firstScene = firstScene;
            this.background = background;
            this.spriteFont = font;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            CurrentScene.Update(gameTime, PlayerIndex.One);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            CurrentScene.Draw(gameTime, spriteBatch, background, spriteFont);
        }

        public void AddScene(string sceneName, GameScene scene)
        {
            if (!scenes.ContainsKey(sceneName))
                scenes.Add(sceneName, scene);
        }

        public GameScene GetScene(string sceneName)
        {
            if (scenes.ContainsKey(sceneName))
                return scenes[sceneName];

            return null;
        }

        public void StartConversation()
        {
            currentScene = firstScene;
        }

        public void ChangeScene(string sceneName)
        {
            currentScene = sceneName;
        }

        #endregion
    }
}
```

I added in member variables for the name of the conversation, the first scene of the conversation, the current scene of the conversation, a dictionary of scenes, a texture for the conversation and a font for the scene. I also include member variables for the name of the background for the conversation and the name of the font the text will be drawn with.

Next there are properties that expose the member variables. The ones for the name fields are both getters and setters. Most of the others are simple getters only. The one that is more than just a simple getter is the one that returns the current scene. Rather than returning the key for the scene I return the scene using the key.

The constructor for this class takes four parameters: name, firstScene, background and font. They represent the name of the conversation, the first scene to be displayed, the background for the conversation and the font the conversation is drawn with. I just set the fields with the parameters that are passed in.

The scene needs to have its Update method called so I included an Update method in this class. It calls the Update method of the current scene for the conversation. The conversation needs to be drawn so there is a Draw method for the conversation. It just calls the Draw method of the current scene.

You can just use the raw dictionary to add and retrieve scenes but I included a method for adding a scene and a method for getting a scene. The reason being is that I can validate the values and prevent the game from crashing if something unexpected is passed in.

The last two methods on this class are StartConversation and ChangeScene. StartConversation sets the currentScene member variable to the firstScene member variable. ChangeScene changes the scene to the parameter that is passed in.

The last thing that I'm going to add in this tutorial is a class that manages the conversations in the game. Right click the ConversationComponents folder, select Add and then Class. Name this new class ConversationManager. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Xml;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

```
namespace Avatars.ConversationComponents
{
    public class ConversationManager
    {
        #region Field Region

        private static Dictionary<string, Conversation> conversationList = new
Dictionary<string, Conversation>();

        #endregion

        #region Property Region

        public static Dictionary<string, Conversation> ConversationList
        {
            get { return conversationList; }
        }

        #endregion

        #region Constructor Region

        public ConversationManager()
        {
        }

        #endregion

        #region Method Region
        public static void AddConversation(string name, Conversation conversation)
        {
            if (!conversationList.ContainsKey(name))
                conversationList.Add(name, conversation);
        }

        public static Conversation GetConversation(string name)
        {
            if (conversationList.ContainsKey(name))
                return conversationList[name];

            return null;
        }

        public static bool ContainsConversation(string name)
        {
            return conversationList.ContainsKey(name);
        }

        public static void ToFile(string fileName)
        {
            XmlDocument xmlDoc = new XmlDocument();

            XmlElement root = xmlDoc.CreateElement("Conversations");
            xmlDoc.AppendChild(root);

            foreach (string s in ConversationManager.ConversationList.Keys)
            {
                Conversation c = ConversationManager.GetConversation(s);

                XmlElement conversation = xmlDoc.CreateElement("Conversation");

                XmlAttribute name = xmlDoc.CreateAttribute("Name");
                name.Value = s;
                conversation.Attributes.Append(name);
                XmlAttribute firstScene = xmlDoc.CreateAttribute("FirstScene");
                firstScene.Value = c.FirstScene;
                conversation.Attributes.Append(firstScene);

                XmlAttribute backgroundName = xmlDoc.CreateAttribute("BackgroundName");
                backgroundName.Value = c.BackgroundName;
                conversation.Attributes.Append(backgroundName);

                XmlAttribute fontName = xmlDoc.CreateAttribute("FontName");
                fontName.Value = c.FontName;
                conversation.Attributes.Append(fontName);

                foreach (string sc in c.Scenes.Keys)
                {
                    GameScene g = c.Scenes[sc];

                    XmlElement scene = xmlDoc.CreateElement("GameScene");

                    XmlAttribute sceneName = xmlDoc.CreateAttribute("Name");
                    sceneName.Value = sc;

                    scene.Attributes.Append(sceneName);

                    XmlElement text = xmlDoc.CreateElement("Text");
                    text.InnerText = c.Scenes[sc].Text;

                    foreach (SceneOption option in g.Options)
                    {
                        XmlElement sceneOption = xmlDoc.CreateElement("GameSceneOption");

                        XmlAttribute oText = xmlDoc.CreateAttribute("Text");
                        oText.Value = option.OptionText;
                        sceneOption.Attributes.Append(oText);

                        XmlAttribute oOption = xmlDoc.CreateAttribute("Option");
                        oOption.Value = option.OptionScene;
                        sceneOption.Attributes.Append(oOption);

                        XmlAttribute oAction = xmlDoc.CreateAttribute("Action");
                        oAction.Value = option.OptionAction.ToString();
                        sceneOption.Attributes.Append(oAction);

                        XmlAttribute oParam = xmlDoc.CreateAttribute("Parameter");
                        oParam.Value = option.OptionAction.Parameter;

                        scene.AppendChild(sceneOption);
                    }

                    conversation.AppendChild(scene);
                }

                root.AppendChild(conversation);
            }

            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;
            settings.Encoding = Encoding.UTF8;

            FileStream stream = new FileStream(fileName, FileMode.Create, FileAccess.Write);
            XmlWriter writer = XmlWriter.Create(stream, settings);
            xmlDoc.Save(writer);
        }

        public static void FromFile(string fileName, Game gameRef, bool editor = false)
        {
            XmlDocument xmlDoc = new XmlDocument();
            try
            {
                xmlDoc.Load(fileName);

                XmlNode root = xmlDoc.FirstChild;

                if (root.Name == "xml")
                    root = root.NextSibling;

                if (root.Name != "Conversations")
                    throw new Exception("Invalid conversation file!");

                foreach (XmlNode node in root.ChildNodes)
                {
                    if (node.Name == "#comment")
                        continue;

                    if (node.Name != "Conversation")
                        throw new Exception("Invalid conversation file!");

                    string conversationName = node.Attributes["Name"].Value;
                    string firstScene = node.Attributes["FirstScene"].Value;
                    string backgroundName = node.Attributes["BackgroundName"].Value;
                    string fontName = node.Attributes["FontName"].Value;

                    Texture2D background = gameRef.Content.Load<Texture2D>(@"Backgrounds\" +
backgroundName);
                    SpriteFont font = gameRef.Content.Load<SpriteFont>(@"Fonts\" +
fontName);

                    Conversation conversation = new Conversation(conversationName,
firstScene, background, font);
                    conversation.BackgroundName = backgroundName;
                    conversation.FontName = fontName;

                    foreach (XmlNode sceneNode in node.ChildNodes)
                    {
                        string text = "";
                        string optionText = "";
                        string optionScene = "";
                        string optionAction = "";
                        string optionParam = "";
                        string sceneName = "";

                        if (sceneNode.Name != "GameScene")
                            throw new Exception("Invalid conversation file!");

                        sceneName = sceneNode.Attributes["Name"].Value;

                        List<SceneOption> sceneOptions = new List<SceneOption>();

                        foreach (XmlNode innerNode in sceneNode.ChildNodes)
                        {
                            if (innerNode.Name == "Text")
                                text = innerNode.InnerText;

                            if (innerNode.Name == "GameSceneOption")
                            {
                                optionText = innerNode.Attributes["Text"].Value;
                                optionScene = innerNode.Attributes["Option"].Value;
                                optionAction = innerNode.Attributes["Action"].Value;
                                optionParam = innerNode.Attributes["Parameter"].Value;

                                SceneAction action = new SceneAction();
                                action.Parameter = optionParam;
                                action.Action = (ActionType)Enum.Parse(typeof(ActionType),
optionAction);

                                SceneOption option = new SceneOption(optionText,
optionScene, action);
                                sceneOptions.Add(option);
                            }
                        }

                        GameScene scene = null;

                        if (editor)
                            scene = new GameScene(text, sceneOptions);
                        else
                            scene = new GameScene(gameRef, text, sceneOptions);


                        conversation.AddScene(sceneName, scene);
                    }

                    conversationList.Add(conversationName, conversation);
                }
            }
            catch
            {
            }
            finally
            {
                xmlDoc = null;
            }
        }

        #endregion

        public static void ClearConversations()
        {
            conversationList = new Dictionary<string, Conversation>();
        }
    }
}
```

All of the members, other than the constructor, are all static. I did this because I am sharing this component with other classes. It is not really "best practices" to do this though. In this case it is for a demo and will suit our purposes. You would probably want to update the manager to be align with best object-oriented programming practices in a production game.

The only member variable holds the conversations in the manager. There is also a property that exposes the member variable. The constructor takes no parameters and does no actions. It was included for use in the future.

There are a number of static methods next. The first is AddConversation is and called to add a conversation to the manager. You can also use the static property and add the conversation that way. This just adds a little error checking to make sure a conversation with the given key does not already exists. This was added mostly for the editor so that if you try to add the same conversation twice you will get an error message.

The next static method is GetConversation and is used to retrieve a conversation from the manager. This could also be done with the property as well. I added it because it first checks to see that the conversation exists before trying to retrieve it. This would prevent a crash if the conversation was not present. You would have to handle the null value that is returned or that could crash the game.

ContainsConverstaion just checks to see if the give key is present in the dictionary and returns that back. This was also added to try and have better error checking before adding or retrieving a conversation.

Next is the method ToFile. This method is used to write the conversation manager to an XML document. I went this route to show one of the few ways that I use to read and write content without using the content manager. The method takes as a parameter the name of the file to write the conversation manager to. Creating XML documents through code is process of creating a root node and then appending children to that node. Those children can also have child nodes under them.

The first thing to do is to create a new XmlDocument. To that I add the root element for the document, Conversations. Next that is appended to the document. The next step is to loop over all of the conversations and create a node for them to append them to the root node of the document.

The first step is to get the conversation using the key from the foreach loop that iterates over the collection of conversations. Next I create a new element/node. I then create attributes for the element for the name, firstScene, backgroundName and fontName members. There is then another foreach loop to go over the scene collection.

Inside that loop I first get the scene using the key. I then create an element for that scene. I then create an attribute for the name of the scene and the text for the scene.

There is then another foreach loop to iterate over the options for that scene. I then create an element for that option. I then create attributes for the three scene properties, text, option and parameter. I then append that element to the scene element. Next the scene is appended to the conversation. The conversation is then appended to the root.

Now that the document is created it needs to be written out. I use the XmlWriter class for that. It requires and XmlWriterSettings parameter so I create that with standard XML attributes, indentation and encoding as UTF8. Next I create a FileStream that is required using Create and Write options. The Create option will create a new file if the file does not exist or overwrite the existing file if does exist. I then create the writer and write the file

The next static method is FromFile that will parse the XML document that was written out earlier. The first thing to do is to create an XmlDocument object. I then do everything inside of a try-catch-finally block to prevent crashes.

The XmlDocument class has a method Load that will load the document into that object. The object can then be parsed and the data be pulled out. I grab the root node of the document using the FirstChild property. If the name is xml then it is the header and we need to go down a level so I set the root node to its next sibling. I then compare that to Conversations. If it is not Conversations then the file is not in the format that we are expecting and I throw an exception.

I then iterate over all of the child nodes using the ChildNodes collection. I check to see if the name is comment, if it is I move onto the next node. Next I check to see if it is Conversation. If it is not conversation the document is not in the right format so I throw an exception. I then grab the name, first scene, background and font attributes. Next up I use the game object passed in to load the background texture and the sprite font. I then create a conversation using the attributes and assign the background and font properties.

Since that node should have child nodes I have a foreach loop that will iterate over them. Inside that loop I have some local variables to hold values that are required for scenes and scene options. If the name of the node is not GameScene I throw an exception. I then grab the Name attribute and assign it to the sceneName variable. Next I create a list of scene options that is required for creating the game scene. I then iterate over the child nodes. If the name is Text then I set the text variable to the inner text of that node. If it is GameSceneOption I assign the local variables to the attributes of the node. Afterwards I create an action and then the option using the text and the action. I then create a GameScene object and initialize it, so the compiler will not complain it has not been initialized. If we are in the editor I use the first constructor, otherwise is use the second constructor. I then add the scene to the conversation. The conversation is then added to the list of conversations.

I don't do anything in the catch but it is a good idea to handle it in some way. I will cover that in another tutorial. In the finally, which is always called, I set the xmlDoc to null.

There is one other static method, ClearConversations, that creates a new list of conversations. This could tax the garbage collector a bit so it might be best to use the Clear method to remove the elements instead.

I'm going to end the tutorial here as it is a lot to digest in one sitting. In the next tutorial I will cover updating the game to allow for conversations with other characters in 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 9

Conversations Continued

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.

In this tutorial I will be continuing on with having conversations with the characters in the game. First I will add the assets for conversations. Conversations require text, an object to indicate an item is selected and they require a background. You can download the items I used from this link.

After you have downloaded and extracted the content open the MonoGame content builder. Select the Fonts folder and then Add Existing Item button in the tool bar. Browse for the scenefont.spritefont and add it to that to the projection. Now select the Content node and select the New Folder button on the tool bar and name this folder Scenes. Select the Scenes folder and then click the Add Existing Item button in the tool bar. Browse for the scenebackground.png file and add it to the folder. Now, select the Misc folder, and from the toolbar select Add Exiting item. Navigate to the selected.png file and add it to that folder. From the toolbar hit the Save button and then the Build button to build the content. Now close the content builder.

Before we go much further I want to fix a bug that I found in the GameScene class that I created. In the constructors for the class not all of the fields were being assigned correctly. Update those constructors as follows.

```
private GameScene()
{
    NormalColor = Color.Blue;
    HighLightColor = Color.Red;
}

public GameScene(string text, List<SceneOption> options)
    : this()
{
    this.text = text;
    this.options = options;
    textPosition = Vector2.Zero;
}

public GameScene(Game game, string text, List<SceneOption> options)
    : this(text, options)
{
    this.game = game;
}
```

I also want to make an addition to the AnimatedSprite class. What I want to do is add a calculated property that will return the center of the sprite on the screen. This will be used to tell if two sprites are close together. All it does is take the Position of the sprite and add half the height and width to the position to get its center. Add the following property to the AnimatedSprite class.

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

The next thing is to add a state to handle conversations. Right click the GameStates folder, select Add and then Class. Name this new class ConversationState. The initial code for that class follows next.

```
using Avatars.CharacterComponents;
using Avatars.ConversationComponents;
using Avatars.PlayerComponents;
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.GameStates
{
    public interface IConversationState
    {
        void SetConversation(Player player, ICharacter character);
        void StartConversation();
    }

    public class ConversationState : BaseGameState, IConversationState
    {
        #region Field Region

        private Conversation conversation;
        private SpriteFont font;
        private Texture2D background;
        private Player player;
        private ICharacter speaker;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public ConversationState(Game game)
            : base(game)
        {
            game.Services.AddService(typeof(IConversationState), this);
        }
        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            font = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");
            background = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");
            base.LoadContent();
        }

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

            GameRef.SpriteBatch.Begin();
            conversation.Draw(gameTime, GameRef.SpriteBatch);
            GameRef.SpriteBatch.End();
        }

        public void SetConversation(Player player, ICharacter character)
        {
            this.player = player;
            speaker = character;

            if (ConversationManager.ConversationList.ContainsKey(character.Conversation))
                this.conversation =
ConversationManager.ConversationList[character.Conversation];
            else
                manager.PopState();
        }

        public void StartConversation()
        {
            conversation.StartConversation();
        }

        #endregion
    }
}
```

There are first a number of using statements to bring classes from other namespaces into scope for this class rather than need to explicitly reference them. I then added in an interface IconversationState. This interface will be implemented in the class and the class will register the interface as the service. The methods that must be implemented are SetConversation and StartConversation. SetConversation requires two parameters, the Player object and an ICharacter. Since I implemented Character and PCharacter from this interface either can be passed to this method. This method will initialize the conversation. The other method StartConversation will begin the conversation between the player and the character.

The class inherits from BaseGameState so that it can be used with the state manager and it implements the interface. There are a few private member variables. They hold the Conversation that is currently in progress, the SpriteFont to draw text with, the Texture2D for the conversation scene, the Player object and an ICharacter ther represents the character the player is talking with. I did not include any properties in this class to expose member variable. If you need to expose a member variable it would be best to include it in the interface, like the member variables, and implement them as part of the interface.

There is another reason why that this is important that I have not discussed yet. Suppose that you have created a library from your game code so that you can reuse in other games. Now, suppose you need to extend an object in the library, like adding a new ICharacter type. Rather than adding the new class to the library you can create a new class in the game and have it implement ICharacter. That means that anywhere in the library that you have used ICharacter as a type you can pass the new type from the game and you will not need to recompile the library to use this new object. In essence you are making a library that you can create plugins for. You could also publish this library for others to use in their games.

The constructor registers this instance of the object as a service that can be retrieved and consumed as a service in another class. This goes with my last comment. The class can be included in a library and then consumed in another project. The user will only know about what you've published in the interface. They can also extend it by implementing the interface in their own class(es).

In the LoadContent method I load in the font and the background into their respective member variables. The Draw method just calls the Draw method of the current conversation being displayed. SetConversation sets the player member to the value passed in and speaker to the value passed in. It then checks to see if the character has a conversation associated with them. If they do I set the conversation member variable to that conversation. If they don't have a conversation associated with them I pop the state off the stack and return control back to the calling class. The StartConversation method just calls the StartConversation method of the conversation. It would probably be a good idea to make sure that it is not null before doing this but I will leave that as an exercise.

One method is missing from this class, the Update method. That is because it is the brains of this class and it will need to be implemented in stages. First, add these two using statements with the others to bring some classes into scope in this class.

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

I added those because we will be using the input manager to test if the player has selected a scene option. Avatars.Components brings the input handler, Xin, into scope and the other brings in items like the Keys enumeration into scope.

Now, add this Update method to the class between LoadContent and Draw.

```
public override void Update(GameTime gameTime)
{
    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        switch (conversation.CurrentScene.OptionAction.Action)
        {
            case ActionType.Buy :
                break;
            case ActionType.Change :
                speaker.SetConversation(conversation.CurrentScene.OptionScene);
                manager.PopState();
                break;
            case ActionType.End :
                manager.PopState();
                break;
            case ActionType.GiveItems :
                break;
            case ActionType.GiveKey :
                break;
            case ActionType.Quest :
                break;
            case ActionType.Sell :
                break;
            case ActionType.Talk :
                conversation.ChangeScene(conversation.CurrentScene.OptionScene);
                break;
        }
    }
    conversation.Update(gameTime);
    base.Update(gameTime);
}
```

This current implementation checks to see if the space or enter keys have been released. If they have been released there is a switch statement on the currently selected scene option. I then included a case for each value in the enumeration for ActionType. For this tutorial I added code to handle three different actions: Change, End and Talk.

What Change does is changes the conversation for the speaker to an entirely new conversation. To do that I call the SetConversation method on the speaker passing in the OptionScene. The next step is to pop the state off the stack. You would use this when the player is speaking to a character and they choose an option that would end the conversation but the next time they talk to this character you want a different conversation.

End is a trivial case. All it does is pop the conversation state off the stack. As you've guessed this is used when the player stops talking to the character.

Change is a trivial case as well. What it does is call the ChangeScene method passing in the OptionScene of the current scene. This will be used when you want to move the conversation to another branch such as a continue option or answering yes or no to a question from the character.

The last thing to do is to call the Update method on the conversation. If you don't the player will not be able to change their response during the conversation.

Now, let's add this to the game. I'm going to post the code for the entire Game1 class, just to make it easier to follow the changes that are being made. What I have done though is first add a new field of type IConversationState. The next change was to create an instance of the new ConversationState class and assign it to that field. I didn't include a property to expose it because I will be demonstrating how to retrieve it as a service. Here is the code for the new Game1 class.

```
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;
        IConversationState conversationState;

        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);
            conversationState = new ConversationState(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 I want to do is to create a conversation for each of the characters that I added to the demo. To do that I added a new method to the ConversationManager class called CreateConversations. Add this method to that class.

```
public static void CreateConversations(Game gameRef)
{
    Texture2D sceneTexture = gameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");
    SpriteFont sceneFont = gameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");

    Conversation c = new Conversation("MarissaHello", "Hello", sceneTexture, sceneFont);
    c.BackgroundName = "scenebackground";
    c.FontName = "scenefont";

    List<SceneOption> options = new List<SceneOption>();
    SceneOption option = new SceneOption(
        "Good bye.",
        "",
        new SceneAction() { Action = ActionType.End, Parameter = "none" });
    options.Add(option);

    GameScene scene = new GameScene(
        gameRef,
        "Hello, my name is Marissa. I'm still learning about summoning avatars.",
        options);

    c.AddScene("Hello", scene);

    ConversationList.Add("MarissaHello", c);

    c = new Conversation("LanceHello", "Hello", sceneTexture, sceneFont);
    c.BackgroundName = "scenebackground";
    c.FontName = "scenefont";

    options = new List<SceneOption>();
    option = new SceneOption(
        "Yes",
        "ILikeFire",
        new SceneAction() { Action = ActionType.Talk, Parameter = "none" });
    options.Add(option);

    option = new SceneOption(
        "No",
        "IDislikeFire",
        new SceneAction() { Action = ActionType.Talk, Parameter = "none" });
    options.Add(option);

    scene = new GameScene(
        gameRef,
        "Fire avatars are my favorites. Do you like fire type avatars too?",
        options);

    c.AddScene("Hello", scene);

    options = new List<SceneOption>()
    option = new SceneOption(
                    "Good bye.",
                    "",
                    new SceneAction() { Action = ActionType.End, Parameter = "none" });
    options.Add(option);

    scene = new GameScene(
        gameRef,
        "That's cool. I wouldn't want to hug one though.",
        options);

    c.AddScene("ILikeFire", scene);

    options = new List<SceneOption>()
    option = new SceneOption(
                    "Good bye.",
                    "",
                    new SceneAction() { Action = ActionType.End, Parameter = "none" });
    options.Add(option);

    scene = new GameScene(
        gameRef,
        "Each to their own I guess.",
        options);

    c.AddScene("IDislikeFire", scene);

    conversationList.Add("LanceHello", c);
}
```

First thing of note is that this takes a Game type parameter. It is used to give us access to the content manager for importing the background for the scene and the font for the scene. That is exactly what I did at the start of this method.

The first conversation that I'm going to implement is the simplest type. It just one scene and the scene has one option. I first created a Conversation object with the parameters: “MarissaHello”, “Hello”, sceneTexture and sceneFont. The first is the name of the conversation, followed by the name of the first scene and the texture and font.

Next is a List<SceneOption> to hold the options for the scene I'm adding to the conversation. I then create a SceneOption object with the text Good bye, no scene to transition to and for the SceneAction is has ActionType.End and none for the parameter. If you recall from above this will just terminate the conversation and pop that state off the stack. This option is then added to the list.

Since a Conversation is made up of GameScene object I create one with the text to be displayed and the List<SceneOption> that I already created. The scene is then added to the conversation and the conversation is listed to the conversation list.

The other conversation constructed similarly. The difference is that the first scene has two options associated with it, a Yes option and a No option. These options use ActionType.Talk and change to the different branches, ILikeFire or IdislikeFire. Those two scene options display some text based on the player's choice.

The next thing to do for the demo is to create the conversations and then attach them to the characters. I did that in the SetUpNewGame in the GamePlayState class. Update that method as follows.

```
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();

    ConversationManager.CreateConversations(GameRef);

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

```
    teacherOne.SetConversation("LanceHello");
    teacherTwo.SetConversation("MarissaHello");

    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();
}
```

All I did was call the static CreateConversation method on the ConversationManager class. I also called the SetConversation method on the characters that I created and passed in the name of each conversation, LanceHello and MarissaHello.

The last update to have conversations working in the demo is to modify the Update method of the GamePlayState class to check if the space bar or enter key have been pressed and if they have check to see if the player is close to a character. If they are close to a character start a conversation with that character. Update that method as follows.

```
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);

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        foreach (string s in map.Characters.Keys)
        {
            ICharacter c = CharacterManager.Instance.GetCharacter(s);
            float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

            if (Math.Abs(distance) < 72f)
            {
                IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                manager.PushState(
                    (ConversationState)conversationState,
                    PlayerIndexInControl);

                conversationState.SetConversation(player, c);
                conversationState.StartConversation();
            }
        }
    }
    base.Update(gameTime);
}
```

What I did is add an if statement to check if either the space bar or enter key have been released since the last frame. I then iterate over all of the keys of the Characters property of the current map. I then grab the character with that name from the CharacterManager. I then use Vector2.Distance to get how far apart the two centers are. If the absolute value is less than 72 I get the IConversationState that was registered as a service. I then call the SetConversation method passing in the player object and the character object. Finally I start the conversation.

Now, I'm going to explain why I find the distance and compare it to 72. What I'm doing here is a type of collision detection called bounding circles rather than bounding boxes. What that means is I construct a circle around the objects and check to see if they collide by calculating the distance between their centers. I pad the circle so that the player just has to be close, not necessarily colliding with the character. I also like to call this type of collision detection proximity collision detection rather.

I'm going to end the tutorial here as it is a lot to digest in one sitting. In the next tutorial I will cover updating the game to allow for conversations with other characters in 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 10

Creating Avatars

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.

This tutorial will cover creating avatars and the moves that they can learn. The way that I implemented avatars in the demo that I created was as a CSV file. That data is read when the game loads and the avatars are then created from each line in the CSV file. This is what a sample avatar looked like in my demo.

```
Fire,Fire,100,1,12,8,10,50,Tackle:1,Block:1,Flare:2,None:100,None:100,None:100
```

The first value is the avatar's name, really creative wasn't I. The next value is the avatars element. The next value was how much it would cost to buy the avatar from a character. The next value is the avatar's level. The next four elements are the attack, defense, speed and health of the avatar. The next six parameters are the moves that the avatar knows or can learn. They consist of two values. The name of the move and the level at which it unlocks. I will implement this a little differently in the tutorial. I'm going to make the move list dynamic but still follow the same rule.

Before I implement this I want to add in a manager class to manage moves and avatars. Right click the AvatarComponents folder, select Add and then Class. Name this new class MoveManager. Here is the code.

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

namespace Avatars.AvatarComponents
{
    public static class MoveManager
    {
        #region Field Region
        private static Dictionary<string, IMove> allMoves = new Dictionary<string, IMove>();
        private static Random random = new Random();

        #endregion

        #region Property Region

        public static Random Random
        {
            get { return random; }
        }

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region

        public static void FillMoves()
        {
            //AddMove(new Tackle());
            //AddMove(new Block());
            //AddMove(new Haste());
            //AddMove(new Bless());
            //AddMove(new Curse());
            //AddMove(new Heal());
            //AddMove(new Flare());
            //AddMove(new Shock());
            //AddMove(new Gust());
            //AddMove(new Frostbite());
            //AddMove(new Shade());
            //AddMove(new Burst());
            //AddMove(new RockThrow());
        }

        public static IMove GetMove(string name)
        {
            if (allMoves.ContainsKey(name))
                return (IMove)allMoves[name].Clone();

            return null;
        }

        public static void AddMove(IMove move)
        {
            if (!allMoves.ContainsKey(move.Name))
                allMoves.Add(move.Name, move);
        }

        #endregion
    }
}
```

This is like most of the other managers that we've implemented through out the game so far. It consists of a static member variable to hold all of the objects that are being managed, in this case IMove. I included a Random member variable that will be shared by all of the moves as well. This just helps to make sure that the random number generation is consistent across all objects. You can also specify a seed here so that the same number series is generated each time that you run the game. It would be a good idea to do that when in debug mode by not when in release mode. Another difference in this class is that I don't exposes the moves with a property. I force the use of the get and add methods to get and add moves to the manager. I included the FillMoves function that I created with all of the moves commented out, because they haven't been implemented yet and will cause a compile time error since they could not be found.

GetMove is used to retrieve a move from the manager. Rather than just returning that object I return a clone of the object. If you don't anywhere that you modify that object it will affect all other variables that point to that object. This is because classes are reference types. That means that the variable does not contain the data, it points to a location in memory that contains the data.

The AddMove method works in reverse. It takes a move and checks to see if it has been added. If it has not yet been added it is added to the move collection.

The next thing that I want to add is a method to the Avatar class that will take a string in the format above and return an Avatar object. Add the following method to the bottom of the Avatar class.

```
public static Avatar FromString(string description, ContentManager content)
{
    Avatar avatar = new Avatar();
    string[] parts = description.Split(',');

    avatar.name = parts[0];
    avatar.texture = content.Load<Texture2D>(@"AvatarImages\" + parts[0]);
    avatar.element = (AvatarElement)Enum.Parse(typeof(AvatarElement), parts[1]);
    avatar.costToBuy = int.Parse(parts[2]);
    avatar.level = int.Parse(parts[3]);
    avatar.attack = int.Parse(parts[4]);
    avatar.defense = int.Parse(parts[5]);
    avatar.speed = int.Parse(parts[6]);
    avatar.health = int.Parse(parts[7]);
    avatar.currentHealth = avatar.health;

    avatar.knownMoves = new Dictionary<string, IMove>();

    for (int i = 8; i < parts.Length; i++)
    {
        string[] moveParts = parts[i].Split(':');

        if (moveParts[0] != "None")
        {
            IMove move = MoveManager.GetMove(moveParts[0]);
            move.UnlockedAt = int.Parse(moveParts[1]);

            if (move.UnlockedAt <= avatar.Level)
                move.Unlock();

            avatar.knownMoves.Add(move.Name, move);
        }
    }

    return avatar;
}
```

The method accepts two parameters. It accepts a string that describes the avatar and a ContentManager to load the avatar's image. It first creates a new Avatar object using the private constructor that was added to the class. I then call the Split method on the description to break it up into its individual parts.

The next series of lines assign the value of the Avatar object using the parts. The texture is assigned by loading the image using the content manager that was passed in. Up until I assign the currentHealth member variable I use the Parse method of the individual items to get the associated values. For the element I had to specify the type of the enumeration that holds the elements and the value. The currentHealth member is assigned the value of the health member.

The next step is to create the Dictionary that will hold the moves that the avatar knows. Next up is a loop that will loop through the remaining parts of the string and get the moves for the avatar. The first step is to split the string into parts base on the colon. If the first part of the string is None then skip it. I then use the GetMove method of the MoveManager to get the move. I then use the second part to determine what level the move unlocks at. If the avatar's level is greater than or equal to that I unlock the move. Finally the move is added to the dictionary of moves for the avatar. The avatar is then returned to the calling method.

As you saw from the MoveManager I had created several moves for the demo that I did. Since moves implement the IMove interface most of the code is identical. The differences are in the constructors and the Clone methods. I'm going to implement two moves in this tutorial. For the other moves I suggest that you download the source code for them.

What I am going to implement are a basic attack move, Tackle, and a basic block move, Block. Let's start with Tackle. Right click the AvatarComponents folder, select Add and then Class. Name this new class Tackle. Here is the code for that class.

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

namespace Avatars.AvatarComponents
{
    public class Tackle : IMove
    {
        #region Field Region

        private string name;
        private Target target;
        private MoveType moveType;
        private MoveElement moveElement;
        private Status status;
        private bool unlocked;
        private int unlockedAt;
        private int duration;
        private int attack;
        private int defense;
        private int speed;
        private int health;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public Target Target
        {
            get { return target; }
        }

        public MoveType MoveType
        {
            get { return moveType; }
        }

        public MoveElement MoveElement
        {
            get { return moveElement; }
        }

        public Status Status
        {
            get { return status; }
        }

        public int UnlockedAt
        {
            get { return unlockedAt; }
            set { unlockedAt = value; }
        }

        public bool Unlocked
        {
            get { return unlocked; }
        }

        public int Duration
        {
            get { return duration; }
            set { duration = value; }
        }

        public int Attack
        {
            get { return attack; }
        }

        public int Defense
        {
            get { return defense; }
        }

        public int Speed
        {
            get { return speed; }
        }

        public int Health
        {
            get { return health; }
        }

        #endregion

        #region Constructor region

        public Tackle()
        {
            name = "Tackle";
            target = Target.Enemy;
            moveType = MoveType.Attack;
            moveElement = MoveElement.None;
            status = Status.Normal;
            duration = 1;
            unlocked = false;
            attack = MoveManager.Random.Next(0, 0);
            defense = MoveManager.Random.Next(0, 0);
            speed = MoveManager.Random.Next(0, 0);
            health = MoveManager.Random.Next(10, 15);
        }

        #endregion

        #region Method Region

        public void Unlock()
        {
            unlocked = true;
        }

        public object Clone()
        {
            Tackle tackle = new Tackle();
            tackle.unlocked = this.unlocked;
            return tackle;
        }

        #endregion
    }
}
```

There are member variables to expose each of the properties that must be implemented for the IMove interface. I will run over what they are briefly. Name is the name that will be displayed when the avatar uses this move. Target is what the move targets. MoveType is what kind of move it is. MoveElement is the element associated with the move. Status determines if there is a status change for the move. Unlocked is if the move is unlocked or not. Duration is how long the move lasts. Moves that have a duration of 1 take place that round. Moves with a duration greater than 1 last that many rouds. Attack is applied to the target's attack attribute. Defense, Speed and Health are applied to their respective attributes.

The constructors set the properties for the move. For tackle, the name is set to Tackle, the target is an enemy, the type is attack, the element is none, the status is normal, the duration is 1, as explained earlier that means it is applied immediately and initially the move is locked. Next I up I assign attack, defense, speed and health random values. In this case an attack damages an opponent so I generate a number between 10 and 14 because the upper limit is exclusive for this method of the Random class. The Clone method creates a new Tackle object and assigns it's unlocked value to the object returned.

Next up I'm going to add in the Block class. This move is a buff for the avatar and increase's its defense score for a few rounds. Right click the AvatarComponents folder, select Add and then Class. Name this new class Block. Here is the code for that class.

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

namespace Avatars.AvatarComponents
{
    public class Block : IMove
    {
        #region Field Region

        private string name;
        private Target target;
        private MoveType moveType;
        private MoveElement moveElement;
        private Status status;
        private bool unlocked;
        private int unlockedAt;
        private int duration;
        private int attack;
        private int defense;
        private int speed;
        private int health;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public Target Target
        {
            get { return target; }
        }

        public MoveType MoveType
        {
            get { return moveType; }
        }

        public MoveElement MoveElement
        {
            get { return moveElement; }
        }

        public Status Status
        {
            get { return status; }
        }

        public int UnlockedAt
        {
            get { return unlockedAt; }
            set { unlockedAt = value; }
        }

        public bool Unlocked
        {
            get { return unlocked; }
        }

        public int Duration
        {
            get { return duration; }
            set { duration = value; }
        }

        public int Attack
        {
            get { return attack; }
        }

        public int Defense
        {
            get { return defense; }
        }

        public int Speed
        {
            get { return speed; }
        }

        public int Health
        {
            get { return health; }
        }

        #endregion

        #region Constructor Region

        public Block()
        {
            name = "Block";
            target = Target.Self;
            moveType = MoveType.Buff;
            moveElement = MoveElement.None;
            status = Status.Normal;
            unlocked = false;
            duration = 5;
            attack = MoveManager.Random.Next(0, 0);
            defense = MoveManager.Random.Next(2, 6);
            speed = MoveManager.Random.Next(0, 0);
            health = MoveManager.Random.Next(0, 0);
        }

        #endregion

        #region Method Region

        public void Unlock()
        {
            unlocked = true;
        }

        public object Clone()
        {
            Block block = new Block();
            block.unlocked = this.unlocked;
            return block;
        }

        #endregion
    }
}
```

Until you get to the constructor the code is identical to the Tackle move. Once you hit the constructor just a few of the properties change. The Name property changes to Block, the target to self, the type to buff, the duration to 5 and the random numbers generated. In this case Health is 0 and defense is between 2 and 5, because 6 is excluded from that range. The next difference is not until the Clone method where I created a Block object to return rather than a Tackle object.

At the time that I implemented this I was on a time restriction that I needed to finish the game for submission to the challenge I was building for. In hind sight this could have been better designed, similar to how I created avatars using strings. It would be idea to modify this so that you have perhaps one class that all moves share rather than a class for each move. I will leave that as an exercise, unless I get requests to demonstrate how you could do that.

So now I'm going to add some avatars to the game. I apologize for the lack of imagination when it comes to their names in advance. The first step will be to right click the Avatars project, select Add and then Folder. Name this new folder Data. Right click this new Data folder, select Add and then New Item. From the list of templates choose Textfile and name it Avatars.csv. Next, select the Avatars.csv file in the solution. In the properties window for the file change the Copy to Output Directory property to Copy always. This will ensure that the file is copied to the output directory for testing. Copy and paste the following lines into the Avatars.csv file.

```
Dark,Dark,100,1,9,12,10,50,Tackle:1,Block:1
Earth,Earth,100,1,10,10,9,60,Tackle:1,Block:1
Fire,Fire,100,1,12,8,10,50,Tackle:1,Block:1
Light,Light,100,1,12,9,10,50,Tackle:1,Block:1
Water,Water,100,1,9,12,10,50,Tackle:1,Block:1
Wind,Wind,100,1,10,10,12,50,Tackle:1,Block:1
```

All of the avatars start out basically the same. Each of them costs 100 gold to purchase and their levels start at 1.The next few properties are their attack, defense, speed and health. I tried to make each of them slightly different than the others. For example, in my description I said that earth avatars are strong but slow. For that reason I reduces their speed to 9 but increased their health to 60. I would recommend in your own games that you play test the different avatars to make sure that you don't end up with a “broken” avatar that your players will always use because they can't be beaten by other avatars.

The next step will be load these into the game. I did that by adding in a class to manage the avatars in the game. To this class I added a FromFile method that read the file line by line and created the avatars from each line. Let's add that to the game now. Right click the AvatarComponents folder, select Add and then Class. Name this new class AvatarManager. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Microsoft.Xna.Framework.Content;

namespace Avatars.AvatarComponents
{
    public static class AvatarManager
    {
        #region Field Region

        private static Dictionary<string, Avatar> avatarList = new Dictionary<string,
Avatar>();

        #endregion

        #region Property Region

        public static Dictionary<string, Avatar> AvatarList
        {
            get { return avatarList; }
        }

        #endregion

        #region Constructor Region

        #endregion

        #region Method Region
        public static void AddAvatar(string name, Avatar avatar)
        {
            if (!avatarList.ContainsKey(name))
                avatarList.Add(name, avatar);
        }

        public static Avatar GetAvatar(string name)
        {
            if (avatarList.ContainsKey(name))
                return (Avatar)avatarList[name].Clone();

            return null;
        }

        public static void FromFile(string fileName, ContentManager content)
        {
            using (Stream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read))
            {
                try
                {
                    using (TextReader reader = new StreamReader(stream))
                    {
                        try
                        {
                            string lineIn = "";

                            do
                            {
                                lineIn = reader.ReadLine();
                                if (lineIn != null)
                                {
                                    Avatar avatar = Avatar.FromString(lineIn, content);
                                    if (!avatarList.ContainsKey(avatar.Name))
                                        avatarList.Add(avatar.Name, avatar);
                                }
                            } while (lineIn != null);
                        }
                        catch
                        {
                        }
                        finally
                        {
                            if (reader != null)
                                reader.Close();
                        }
                    }
                }
                catch
                {
                }
                finally
                {
                    if (stream != null)
                        stream.Close();
                }
            }
        }

        #endregion
    }
}
```

This manager is similar to all other manager classes. It maintains a dictionary for the objects to be managed and has a mechanism to add and retrieve objects. What is different about this one is that it has a FromFile method that will open the file, read the avatars from the file and add them to the dictionary. The FromFile method accepts the name of the file to be read and ContentManager to load the images for the avatars.

In order to read from a file you need a Stream object. In this case I created a FileStream as my content was stored on disk. There are other types of streams that you can create in C, such as a NetworkStream that could be used to pull the file from a server if you were to implement your game on Android or iOS. This would prevent cheating where players modify your content locally. Once the stream is open I create a TextReader using the stream to read the text file. I will be loading each line into a variable, lineIn. Next up is a do-while loop that loops for as long as there is data in the file. Each pass through I try to read the next line of the file using ReadLine. If that is not null then data was read in. I then create an Avatar object using the FromString method that I showed earlier in the tutorial. Then if there is not already an avatar with that name in the collection I call the Add method to add the avatar to the collection.

There is one last piece of the puzzle missing. There are no images for the for the avatars so when you try to read them in the game will crash. For my demo I grabbed the six images from the YuGiOh card game. They will be good enough for this tutorial as well. You can find my images at this link.

Once you've downloaded and extracted the files open the MonoGame pipeline again. Select the Content folder then click the New Folder icon in the tool bar. Name this new folder AvatarImages. Now select the AvatarImages folder then click the Add Existing Item button in the ribbon. Add the images, dark.PNG, earth.PNG, fire.PNG, light.PNG, water.PNG and wind.PNG. Once the images have been added save the project, click Build from the menu and select Build to build the content project then close the window.

Now we can load the avatars into the game. First, in the MoveManager update the FillMoves folder to create the Tackle and Block moves. If you added the other moves you can create them as well. Now, open the Game1 class and update the LoadContent method to the following.

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

    AvatarComponents.MoveManager.FillMoves();
    AvatarComponents.AvatarManager.FromFile(@".\Data\avatars.csv", Content);
}
```

What this does is create the moves for the avatars and then loads the avatars from the file that we created. At this point you will be able to build and run the game with out problems.

I'm going to end the tutorial here as it is a lot to digest in one sitting. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 11

Battling Avatars

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.

So, we have the player, characters and avatars. The next step will be battling avatars. The first step will be to add to the player class the avatars that the player has captured/learned. I say captured as well as learned because I will be including a second player component that works more like Pokemon than my demo.

To get started open the Player class in the PlayerComponents folder. I updated this class to make the private member variables protected member variables. I also added in a field and accessor methods. Update that class to the following.

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

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

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

        protected Vector2 position;
        protected Dictionary<string, Avatar> avatars = new Dictionary<string, Avatar>();
        private string currentAvatar;

        #endregion

        #region Property Region

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

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

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

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

        #endregion

        #region Constructor Region

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

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

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

        #endregion

        #region Method Region

        public virtual void AddAvatar(string avatarName, Avatar avatar)
        {
            if (!avatars.ContainsKey(avatarName))
                avatars.Add(avatarName, avatar);
        }

        public virtual Avatar GetAvatar(string avatarName)
        {
            if (avatars.ContainsKey(avatarName))
```

```
                return avatars[avatarName];

            return null;
        }

        public virtual void SetAvatar(string avatarName)
        {
            if (avatars.ContainsKey(avatarName))
                currentAvatar = avatarName;
            else
                throw new IndexOutOfRangeException();
        }

        public void SavePlayer()
        {
        }

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

            return player;
        }

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

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

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

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

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

        #endregion
    }
}
```

The new field that I added is a Dictionary<string, Avatar> that holds all the avatars that the player owns. I also added a field currentAvatar that holds which is the avatar that will be used when combat first starts. I also added a virtual property that returns the avatar with that name. I then added a virtual method AddAvatar that checks if an avatar with that name already exist and if it doesn't adds it to the collection of avatars. The GetAvatar virtual method checks to see if an avatar with that name exists in the collection and if it does it returns it. Otherwise it returns null. I also added a SetAvatar method that will set the currentAvatar member to the value passed in if the collection of avatars contains the key passed in. If it does not exist in the collection I thrown an exception.

Why did I make some of the members virtual? I'm taking a slightly different approach with have different types of player components. Instead of using interfaces I'm using inheritance. This is really helpful to learn of you don't know it so that is why I included it in this tutorial. Any of the virtual members can be overridden in the class that inherits from the player with the same name and parameters but behave differently.

Another note, why do I throw exceptions instead of letting the game crash and handle the exception there? The first is throwing a specific exception gives me an idea of what the problem is and how to stop. The second reason is that eventually I start catching the exceptions and handle them. This helps prevent the player from injecting values into the game to cheat as well as find logic errors.

The next thing that I want to add is a second player class, called PPlayer. This class deals with avatars more like the player in Pokemon because they are limited to the number of avatars they have at one time. Right click the PlayerComponents folder in your solution, select Add and then Class. Name the new class PPlayer. Here is the code for that class.

```
using Avatars.AvatarComponents;
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.PlayerComponents
{
    public class PPlayer : Player
    {
        public const int MaxAvatars = 6;

        private List<Avatar> battleAvatars = new List<Avatar>();
        private int currentAvatar;

        public override Avatar CurrentAvatar
        {
            get { return battleAvatars[currentAvatar]; }
        }

        public PPlayer(Game game, string name, bool gender, Texture2D texture)
            : base(game, name, gender, texture)
        {
        }

        public void SetCurrentAvatar(int index)
        {
            if (index < 0 || index > MaxAvatars)
                throw new IndexOutOfRangeException();

            currentAvatar = index;
        }

        public Avatar GetBattleAvatar(int index)
        {
            if (index < 0 || index > MaxAvatars)
                throw new IndexOutOfRangeException();

            return battleAvatars[index];
        }

        public void AddBattleAvatar(Avatar avatar)
        {
            if (battleAvatars.Count >= MaxAvatars - 1)
                throw new OverflowException();
            battleAvatars.Add(avatar);
        }

        public void RemoveBattleAvatar(int index)
        {
            if (index >= battleAvatars.Count)
                throw new IndexOutOfRangeException();

            battleAvatars.RemoveAt(index);
        }
    }
}
```

There is a constant in this class that describes the maximum number of avatars in a party and it is set to 6. This could be set to just about any value but to be consistent with other classes I went with 6. was pretty good for demonstration purposes. Next up is a List<Avatar> which will hold the avatars for the party and an integer variable that holds the currently selected avatar that will be used when a battle starts. There is then a property where I override the behavior of the CurrentAvatar property. Instead of using the string value in the other class I use the integer value in this class.

In order to inherit from a class you need to call one of its constructors. The parent class has a private constructor and a public constructor. Since the values need to be set I added in the parameters to the list of arguments for the constructor to the signature for the constructor and the call the parent using base.

After that there are some methods for manipulating the battle avatars. SetCurrentAvatar is used to assign which avatar is the default for starting combat. It checks to see if the value passed in is out of bounds. If it is it throws an exception, otherwise it sets the value. GetBattleAvatar is used to get the actual avatar object. It also checks to make sure the value is in range and if it is throw an exceptions. Otherwise return the avatar at that particular index.

AddBattleAvatar is used to add an avatar to the player's battle avatar list. It checks to make sure that there is room for the avatar. Since the index is zero based if the count is greater than or equal to the maximum number of avatars minus one there is no room for it so throw an exception. Otherwise it is find to return the avatar. The RemoveBattleAvatar method checks that the index is with in the range of allowed indexes and if it is throws an exception. Otherwise it removes the avatar at the specified index.

I want to make an update to the AvatarManager method FromFile before moving onto the next step. What I want to do is change it so that the dictionary key is always in lower case rather than mixed case. To do that I used the method .ToLowerInvariant(). You could probably get away with just ToLower though. I included the Invariant part because I deal with localization on a daily basis and you may want to update your game to support multiple languages. Here is the update for that method.

```
public static void FromFile(string fileName, ContentManager content)
{
    using (Stream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read))
    {
        try
        {
            using (TextReader reader = new StreamReader(stream))
            {
                try
                {
                    string lineIn = "";
                    do
                    {
                        lineIn = reader.ReadLine();
                        if (lineIn != null)
                        {
                            Avatar avatar = Avatar.FromString(lineIn, content);
                            if (!avatarList.ContainsKey(avatar.Name.ToLowerInvariant()))
                                avatarList.Add(avatar.Name.ToLowerInvariant(), avatar);
                        }
                    } while (lineIn != null);
                }
                catch
                {
                }
                finally
                {
                    if (reader != null)
                        reader.Close();
                }
            }
        }
        catch
        {
        }
        finally
        {
            if (stream != null)
                stream.Close();
        }
    }
}
```

I'm now going to update the Character so that when a string is passed to the FromString method it will used the 5th value in the string to assign the current battleAvatar. Here is the update.

```
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];

    character.battleAvatar = AvatarManager.GetAvatar(parts[4].ToLowerInvariant());

    return character;
}
```

The change is the second last line of code. I call the GetAvatar method of the AvatarManager class passing in the 5th part of the string in lower case. I'm now going to make a similar change to the PCharacter class. It will add upto six avatars to the character's list of avatars. Update that method as follows.

```
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];

    for (int i = 4; i < 10 && i < parts.Length; i++)
        character.avatars[i - 4] = AvatarManager.GetAvatar(parts[i].ToLowerInvariant());

    return character;
}
```

What the code does is similar to what I did when building avatars. For avatars I would read their moves until there were no moves left in the string.

Next step is to assign the player and the characters avatars. I will do that in the SetUpNewGame method of the GamePlayState class. What I did was move creating the player form LoadContent to SetUpNewGame. Please update those two methods to the following.

```
protected override void LoadContent()
{
}

public void SetUpNewGame()
{
    Texture2D spriteSheet = content.Load<Texture2D>(@"PlayerSprites\maleplayer");

    player = new Player(GameRef, "Wesley", false, spriteSheet);
    player.AddAvatar("fire", AvatarManager.GetAvatar("fire"));
    player.SetAvatar("fire");

    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();

    ConversationManager.CreateConversations(GameRef);

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

    teacherOne.SetConversation("LanceHello");
    teacherTwo.SetConversation("MarissaHello");

    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 LoadContent now does nothing and is there in case it is need in the future. I moved creating the player to the top of the method. After creating the player I add a "fire" avatar to their avatar collection and set their current avatar to "fire". For the first character that is a Character I added another value to the string, water, that will set their avatar to be a "water" avatar. For the other character, that is a PCharacter, I added a "wind" and "earth" avatar to the character's collection.

The last thing that I'm going to add is the base battle state and show how to switch to it from the game play state. For that you will need two graphics. One is a border that holds the health bar and the actual health bar. You can download those images using this link. Once they are ready open the MonoGame content builder. Select the Misc folder and then click the Add Existing Item and browse to the avatarborder.png file and add id to the project. Repeat the process for the avatarhealth.png. Now save and build the project.

Next, right click the GameStates folder, select Add and then Class Name this new class BattleState. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Avatars.AvatarComponents;
using Avatars.ConversationComponents;
using Avatars.Components;

namespace Avatars.GameStates
{
    public interface IBattleState
    {
        void SetAvatars(Avatar player, Avatar enemy);
        void StartBattle();
    }

    public class BattleState : BaseGameState, IBattleState
    {
        #region Field Region

        private Avatar player;
        private Avatar enemy;
        private GameScene combatScene;
        private Texture2D combatBackground;
        private Rectangle playerRect;
        private Rectangle enemyRect;
        private Rectangle playerBorderRect;
        private Rectangle enemyBorderRect;
        private Rectangle playerMiniRect;
        private Rectangle enemyMiniRect;
        private Rectangle playerHealthRect;
        private Rectangle enemyHealthRect;
        private Rectangle healthSourceRect;
        private Vector2 playerName;
        private Vector2 enemyName;
        private float playerHealth;
        private float enemyHealth;
        private Texture2D avatarBorder;
        private Texture2D avatarHealth;
        private SpriteFont font;
        private SpriteFont avatarFont;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public BattleState(Game game)
            : base(game)
        {
            playerRect = new Rectangle(10, 90, 300, 300);
            enemyRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 10, 300, 300);

            playerBorderRect = new Rectangle(10, 10, 300, 75);
            enemyBorderRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 320, 300,
75);

            healthSourceRect = new Rectangle(10, 50, 290, 20);
            playerHealthRect = new Rectangle(playerBorderRect.X + 12, playerBorderRect.Y +
52, 286, 16);
            enemyHealthRect = new Rectangle(enemyBorderRect.X + 12, enemyBorderRect.Y + 52,
286, 16);

            playerMiniRect = new Rectangle(playerBorderRect.X + 11, playerBorderRect.Y + 11,
28, 28);
            enemyMiniRect = new Rectangle(enemyBorderRect.X + 11, enemyBorderRect.Y + 11,
28, 28);

            playerName = new Vector2(playerBorderRect.X + 55, playerBorderRect.Y + 5);
            enemyName = new Vector2(enemyBorderRect.X + 55, enemyBorderRect.Y + 5);
        }

        #endregion

        #region Method Region

        public override void Initialize()
        {
            base.Initialize();
        }
        protected override void LoadContent()
        {
            combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

            avatarFont = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");
            avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
            avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

            font = GameRef.Content.Load<SpriteFont>(@"Fonts\gamefont");

            combatScene = new GameScene(GameRef, "", new List<SceneOption>());

            base.LoadContent();
        }

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

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

            GameRef.SpriteBatch.Begin();

            combatScene.Draw(gameTime, GameRef.SpriteBatch, combatBackground, font);

            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);

            playerHealth = (float)player.CurrentHealth / (float)player.GetHealth();
            MathHelper.Clamp(playerHealth, 0f, 1f);
            playerHealthRect.Width = (int)(playerHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, playerHealthRect, healthSourceRect,
Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, enemyBorderRect, Color.White);

            enemyHealth = (float)enemy.CurrentHealth / (float)enemy.GetHealth();
            MathHelper.Clamp(enemyHealth, 0f, 1f);
            enemyHealthRect.Width = (int)(enemyHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, enemyHealthRect, healthSourceRect,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, player.Name, playerName,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, enemy.Name, enemyName, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerMiniRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyMiniRect, Color.White);

            GameRef.SpriteBatch.End();
        }

        public void SetAvatars(Avatar player, Avatar enemy)
        {
            this.player = player;
            this.enemy = enemy;

            player.StartCombat();
            enemy.StartCombat();
        }
        public void StartBattle()
        {
            player.StartCombat();
            enemy.StartCombat();
            playerHealth = 1f;
            enemyHealth = 1f;
        }

        #endregion
    }
}
```

First off, I included an interface for the state IBattleState. This is the contract that is provided to other states to interface with this state. The members that in includes are SetAvatars and StartBattle. The first is used to place the avatars on the screen where as the second is used to start a battle between the two avatars.

Most of the member variables in this class are for position elements and holding the image for the elements. There are also an Avatar field for the player and the enemy. There is also a GameScene that will be used by the player to select what move they want the avatar to use that round.

All the constructor does is position items based on the screen rectangle. The elements that are being places are the images for the avatars, the position of their bar that holds the health bar and the actual health bar.

In the LoadContent method I load in the background, two fonts that were already added to the project and the two textures that were just added. I also create a basic combat scene using the conversation components. This will be used to display options to the player and other messages during combat. Currently the Update screen just calls the base update method in order to update the components.

The Draw method first starts drawing the sprite batch. Since it is at the back the scene is drawn first. After that I draw the player's avatar and then the enemy's avatar. Then I draw the borders and the avatar's current health. To draw the current health I get the percentage of the avatar's health is left. I then clamp it between 0 and 1. To determine the width out I multiple the fraction of the remain health by the width of the texture. After drawing the bars I write the name of the avatar into the bar. Finally I then draw the avatar images.

The SetAvatars method is used to set the avatars. After assigning the local member variables to the corresponding parameters I call StartCombat to start combat between the two. In the StartBattle method I also call the StartCombat method and set two member variables to 1f.

The last thing that I'm going to cover is moving from the game play state to this new state. First, we need to update the Game1 class to create a base battle state that will be reused for all battles in the game. The change was I add a member variable for the state, a property to expose it and create it in the constructor. Here is the updated Game1 class.

```
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;
        IConversationState conversationState;
        IBattleState battleState;

        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 IBattleState BattleState
        {
            get { return battleState; }
        }

        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);
            conversationState = new ConversationState(this);
            battleState = new BattleState(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);

            AvatarComponents.MoveManager.FillMoves();
            AvatarComponents.AvatarManager.FromFile(@".\Data\avatars.csv", Content);
        }

        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);
        }
    }
}
```

Next up I'm going to update the Update method in the GamePlayScreen to switch states to the battle state if the player is close to the character and presses B. Change that method 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);

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        foreach (string s in map.Characters.Keys)
        {
            ICharacter c = CharacterManager.Instance.GetCharacter(s);
            float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

            if (Math.Abs(distance) < 72f)
            {
                IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                manager.PushState(
                    (ConversationState)conversationState,
                    PlayerIndexInControl);

                conversationState.SetConversation(player, c);
                conversationState.StartConversation();
            }
        }
    }

    if (Xin.CheckKeyReleased(Keys.B))
    {
        foreach (string s in map.Characters.Keys)
        {
            ICharacter c = CharacterManager.Instance.GetCharacter(s);
            float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

            if (Math.Abs(distance) < 72f)
            {
                GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                manager.PushState(
                    (BattleState)GameRef.BattleState,
                    PlayerIndexInControl);
            }
        }
    }
    base.Update(gameTime);
}
```

What this new code does is check to see if the B key was released. Then like when I checked for starting a conversation with a character I cycle through all of the characters. I get the character and get its distance to the player. If the distance is less than 72 I call the SetAvatars method of the battle state. I then push the battle state onto the stack.

You can now build and run the game. If you move the player close to either of the characters that were added you can press the B key and the game will switch to the battle state. Once you're in the battle state you're stuck there though. For now I'm going to add a quick escape from the battle stated. Change the Update method of the BattleState class to the following.

```
public override void Update(GameTime gameTime)
{
    PlayerIndex index = PlayerIndex.One;

    if (Xin.CheckKeyReleased(Keys.P))
        manager.PopState();

    base.Update(gameTime);
}
```

That is going to wrap up this tutorial. In the next tutorial I will get into the two avatars battling each other. I will also be working on preparing the game editor that I used for the demo I made so that you can play around with the framework up until now and see how all the pieces tie together.

I'm going to end the tutorial here as it is a lot to digest in one sitting. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 12

Battling Avatars Continued

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.

In this tutorial I'm going to pick up where I left off in the last tutorial in battling avatars. In this tutorial I will add in the next steps to actually battle the two avatars, such as selecting moves and having them applied to the target. I won't be covering the player and opponent having multiple avatars and will focus and just a single avatar. I will write a separate tutorial on how to add that functionality.

In the last tutorial I had added a game state called BattleState. This scene will display the available moves and allow the player to choose the move that they want. The options for this will need to be set each time the SetAvatars is called. Update that method to the following.

```
public void SetAvatars(Avatar player, Avatar enemy)
{
    this.player = player;
    this.enemy = enemy;

    player.StartCombat();
    enemy.StartCombat();

    List<SceneOption> moves = new List<SceneOption>();

    if (combatScene == null)
        LoadContent();

    foreach (string s in player.KnownMoves.Keys)
    {
        SceneOption option = new SceneOption(s, s, new SceneAction());
        moves.Add(option);
    }

    combatScene.Options = moves;
}
```

The change from last time is I built a List<SceneOption> that holds the moves for the avatar. I then check if the combatScene member variable is null and if it is I call LoadContent to create it. I had to do that because since I did not add the game component to the list of components in the game the LoadContent method is not automatically called.

In a foreach loop I loop over all of the keys in the KnownMoves dictionary. Inside I create a new scene options and add it to the list of moves. Before exiting the method I assign the scene options to the list that I just created.

I have updated the LoadContent method to handle the problem where it is not called automatically. All I did was have an if statement to make sure the combatScene is null before trying to load. Update the LoadContent method to the following.

```
protected override void LoadContent()
{
    if (combatScene == null)
    {
        combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

        avatarFont = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");
        avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
        avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

        font = GameRef.Content.Load<SpriteFont>(@"Fonts\gamefont");

        combatScene = new GameScene(GameRef, "", new List<SceneOption>());
    }

    base.LoadContent();
}
```

Next would be to wire the scene to perform the action the player selects and apply it to the enemy. First, I want to add in two game states. One state will be displayed when the battle is over. The other will be displayed while the moves of both avatars are being resolved for the current turn.

First, I am going to add the battle over state. Right click the GameScenes folder, select Add and then Class. Name the class BattleOverState. Here is the code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Avatars.AvatarComponents;
using Avatars.GameStates;
using Avatars.Components;

namespace Avatars.GameStates
{
    public interface IBattleOverState
    {
        void SetAvatars(Avatar player, Avatar enemy);
    }

    public class BattleOverState : BaseGameState, IBattleOverState
    {
        #region Field Region
        private Avatar player;
        private Avatar enemy;
        private Texture2D combatBackground;
        private Rectangle playerRect;
        private Rectangle enemyRect;
        private Rectangle playerBorderRect;
        private Rectangle enemyBorderRect;
        private Rectangle playerMiniRect;
        private Rectangle enemyMiniRect;
        private Rectangle playerHealthRect;
        private Rectangle enemyHealthRect;
        private Rectangle healthSourceRect;
        private Vector2 playerName;
        private Vector2 enemyName;
        private float playerHealth;
        private float enemyHealth;
        private Texture2D avatarBorder;
        private Texture2D avatarHealth;
        private SpriteFont avatarFont;
        private SpriteFont font;
        private string[] battleState;
        private Vector2 battlePosition;
        private bool levelUp;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public BattleOverState(Game game)
            : base(game)
        {
            battleState = new string[3];

            battleState[0] = "The battle was won!";
            battleState[1] = " gained ";
            battleState[2] = "Continue";

            battlePosition = new Vector2(25, 475);

            playerRect = new Rectangle(10, 90, 300, 300);
            enemyRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 10, 300, 300);

            playerBorderRect = new Rectangle(10, 10, 300, 75);
            enemyBorderRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 320, 300,
75);

            healthSourceRect = new Rectangle(10, 50, 290, 20);
            playerHealthRect = new Rectangle(playerBorderRect.X + 12, playerBorderRect.Y +
52, 286, 16);
            enemyHealthRect = new Rectangle(enemyBorderRect.X + 12, enemyBorderRect.Y + 52,
286, 16);

            playerMiniRect = new Rectangle(playerBorderRect.X + 11, playerBorderRect.Y + 11,
28, 28);
            enemyMiniRect = new Rectangle(enemyBorderRect.X + 11, enemyBorderRect.Y + 11,
28, 28);

            playerName = new Vector2(playerBorderRect.X + 55, playerBorderRect.Y + 5);
            enemyName = new Vector2(enemyBorderRect.X + 55, enemyBorderRect.Y + 5);
        }

        #endregion
        #region Method Region

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

        protected override void LoadContent()
        {
            combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

            avatarFont = GameRef.Content.Load<SpriteFont>(@"Fonts\GameFont");
            avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
            avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

            font = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            PlayerIndex index = PlayerIndex.One;

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                if (levelUp)
                {
                    this.Visible = true;
                }
                else
                {
                    manager.PopState();
                    manager.PopState();
                }
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            Vector2 position = battlePosition;

            base.Draw(gameTime);

            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(combatBackground, Vector2.Zero, Color.White);

            for (int i = 0; i < 2; i++)
            {
                GameRef.SpriteBatch.DrawString(font, battleState[i], position, Color.Black);
                position.Y += avatarFont.LineSpacing;
            }

            GameRef.SpriteBatch.DrawString(font, battleState[2], position, Color.Red);

            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);

            playerHealth = (float)player.CurrentHealth / (float)player.GetHealth();
            MathHelper.Clamp(playerHealth, 0f, 1f);
            playerHealthRect.Width = (int)(playerHealth * 286);
            GameRef.SpriteBatch.Draw(avatarHealth, playerHealthRect, healthSourceRect,
Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, enemyBorderRect, Color.White);

            enemyHealth = (float)enemy.CurrentHealth / (float)enemy.GetHealth();
            MathHelper.Clamp(enemyHealth, 0f, 1f);
            enemyHealthRect.Width = (int)(enemyHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, enemyHealthRect, healthSourceRect,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, player.Name, playerName,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, enemy.Name, enemyName, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerMiniRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyMiniRect, Color.White);

            GameRef.SpriteBatch.End();
        }

        public void SetAvatars(Avatar player, Avatar enemy)
        {
            levelUp = false;

            long expGained = 0;

            this.player = player;
            this.enemy = enemy;

            if (player.Alive)
            {
                expGained = player.WinBattle(enemy);
                battleState[0] = player.Name + " has won the battle!";
                battleState[1] = player.Name + " has gained " + expGained + " experience";

                if (player.CheckLevelUp())
                {
                    battleState[1] += " and gained a level!";

                    foreach (string s in player.KnownMoves.Keys)
                    {
                        if (player.KnownMoves[s].Unlocked == false && player.Level >=
player.KnownMoves[s].UnlockedAt)
                        {
                            player.KnownMoves[s].Unlock();
                            battleState[1] += " " + s + " was unlocked!";
                        }
                    }

                    levelUp = true;
                }
                else
                {
                    battleState[1] += ".";
                }
            }
            else
            {
                expGained = player.LoseBattle(enemy);

                battleState[0] = player.Name + " has lost the battle.";
                battleState[1] = player.Name + " has gained " + expGained + " experience";

                if (player.CheckLevelUp())
                {
                    battleState[1] += " and gained a level!";

                    foreach (string s in player.KnownMoves.Keys)
                    {
                        if (player.KnownMoves[s].Unlocked == false && player.Level >=
player.KnownMoves[s].UnlockedAt)
                        {
                            player.KnownMoves[s].Unlock();
                            battleState[1] += " " + s + " was unlocked!";
                        }
                    }

                    levelUp = true;
                }
                else
                {
                    battleState[1] += ".";
                }
            }
        }

        #endregion
    }
}
```

I included an interface for this class, IBattleOverState, that includes one method that must be implemented SetAvatars. This method is used just like the one in IBattleState for setting the battle avatars.

Like the BattleState this class has a lot of member variables for drawing and positioning graphical elements. There are member variables for both avatars as well. I included an array of strings called battleState. These strings are used to build and display the results of the battle to the player. I also included a member variable for positioning this element on the screen. The last member variable is a bool that measures if the avatar has leveled up or not.

The constructor just initializes member variables. I also create an array to hold some of the predefined strings and assign them values.

This time I didn't wrap loading the content into an if statement. The reason why is in the previous game state I was initializing values right after calling SetAvatar. In this case the load does not need to be done automatically and is deferred until after all creation has finished.

In the Update method I have the PlayerIndex variable that you have seen through out all of the game states so far. Next is an if statement where I check if the space key or enter key have been released. There is then an if statement that checks if the levelUp member variable is true or not. If it is I just set the Visible property of the game state to true. Later on in the tutorial I will be adding a level up state. If it was false I call PopState twice. The reason will be that multiple states will need to be popped of the stack to get back to the game play state.

Like in the other states the Draw method just positions all of the elements. What is different is that I also position the text displaying the battle state. First, I loop over all of the elements for the state and draw them at the desired position. I then update the Y position by the LineSpacing property for the font to move onto the next line.

In the SetAvatars method I determine what needs to be rendered by the scene. I check the Alive property of the player's avatar to determine if that avatar won the battle. If it did I call the WinBattle method passing in the enemy after to calculate how much experience the avatar gained for defeating this avatar. I then update the battle strings with these values. I next call the CheckLevelUp method on the player avatar to check if the avatar has levelled up or not. If it has I update the string being displayed. I then loop through all of the known moves and unlock any moves that unlock at the new level. I also append text to the display that the move was unlocked. Finally I set the levelUp member variable to true. If it did not level up I just append a period.

If the player's avatar lost I call the LostBattle method to assign it the experience it learned during the battle. I then constructor the strings that will display that the battle was lost. I also go over the same process of checking if the avatar levelled up or not.

Next up I want to add in the damage state. This state will resolve the selected moves for the avatars. Right click the GameStates folder, select Add and then Class. Name this new class DamageState. Here is the code for that class.

```
using Avatars.AvatarComponents;
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.GameStates
{
    public enum CurrentTurn
    {
        Players, Enemies
    }

    public interface IDamageState
    {
        void SetAvatars(Avatar player, Avatar enemy);
        void SetMoves(IMove playerMove, IMove enemyMove);
        void Start();
    }

    public class DamageState : BaseGameState, IDamageState
    {
        #region Field Region

        private CurrentTurn turn;
        private Texture2D combatBackground;
        private SpriteFont avatarFont;
        private SpriteFont font;
        private Rectangle playerRect;
        private Rectangle enemyRect;
        private TimeSpan cTimer;
        private TimeSpan dTimer;
        private Avatar player;
        private Avatar enemy;
        private IMove playerMove;
        private IMove enemyMove;
        private bool first;
        private Rectangle playerBorderRect;
        private Rectangle enemyBorderRect;
        private Rectangle playerMiniRect;
        private Rectangle enemyMiniRect;
        private Rectangle playerHealthRect;
        private Rectangle enemyHealthRect;
        private Rectangle healthSourceRect;
        private float playerHealth;
        private float enemyHealth;
        private Texture2D avatarBorder;
        private Texture2D avatarHealth;
        private Vector2 playerName;
        private Vector2 enemyName;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public DamageState(Game game)
            : base(game)
        {
            playerRect = new Rectangle(10, 90, 300, 300);
            enemyRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 10, 300, 300);

            playerBorderRect = new Rectangle(10, 10, 300, 75);
            enemyBorderRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 320, 300,
75);

            healthSourceRect = new Rectangle(10, 50, 290, 20);
            playerHealthRect = new Rectangle(playerBorderRect.X + 12, playerBorderRect.Y +
52, 286, 16);
            enemyHealthRect = new Rectangle(enemyBorderRect.X + 12, enemyBorderRect.Y + 52,
286, 16);

            playerMiniRect = new Rectangle(playerBorderRect.X + 11, playerBorderRect.Y + 11,
28, 28);
            enemyMiniRect = new Rectangle(enemyBorderRect.X + 11, enemyBorderRect.Y + 11,
28, 28);

            playerName = new Vector2(playerBorderRect.X + 55, playerBorderRect.Y + 5);
            enemyName = new Vector2(enemyBorderRect.X + 55, enemyBorderRect.Y + 5);
        }

        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

            avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
            avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

            avatarFont = Game.Content.Load<SpriteFont>(@"Fonts\GameFont");
            font = Game.Content.Load<SpriteFont>(@"Fonts\scenefont");

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            PlayerIndex index;
            if ((cTimer > TimeSpan.FromSeconds(4) || !enemy.Alive || !player.Alive) &&
dTimer > TimeSpan.FromSeconds(3))
            {
                if (!enemy.Alive || !player.Alive)
                {
                    manager.PopState();
                    manager.PushState((BattleOverState)GameRef.BattleOverState,
PlayerIndex.One);
                    GameRef.BattleOverState.SetAvatars(player, enemy);
                }
                else
                {
                    manager.PopState();
                }
            }
            else if (cTimer > TimeSpan.FromSeconds(2) && first && enemy.Alive &&
player.Alive)
            {
                first = false;
                dTimer = TimeSpan.Zero;
                if (turn == CurrentTurn.Players)
                {
                    turn = CurrentTurn.Enemies;
                    enemy.ResoleveMove(enemyMove, player);
                }
                else
                {
                    turn = CurrentTurn.Players;
                    player.ResoleveMove(playerMove, enemy);
                }
            }
            else if (cTimer == TimeSpan.Zero)
            {
                dTimer = TimeSpan.Zero;
                if (turn == CurrentTurn.Players)
                {
                    player.ResoleveMove(playerMove, enemy);
                }
                else
                {
                    enemy.ResoleveMove(enemyMove, player);
                }
            }

            cTimer += gameTime.ElapsedGameTime;
            dTimer += gameTime.ElapsedGameTime;

            base.Update(gameTime);
        }

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

            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(combatBackground, Vector2.Zero, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            Vector2 location = new Vector2(25, 475);

            if (turn == CurrentTurn.Players)
            {
                GameRef.SpriteBatch.DrawString(font, player.Name + " uses " +
playerMove.Name + ".", location, Color.Black);

                if (playerMove.Target == Target.Enemy && playerMove.MoveType ==
MoveType.Attack)
                {
                    location.Y += avatarFont.LineSpacing;

                    if (Avatar.GetMoveModifier(playerMove.MoveElement, enemy.Element) < 1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is not very effective.",
location, Color.Black);
                    }
                    else if (Avatar.GetMoveModifier(playerMove.MoveElement, enemy.Element) >
1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is super effective.",
location, Color.Black);
                    }
                }
            }
            else
            {
                GameRef.SpriteBatch.DrawString(font, "Enemy " + enemy.Name + " uses " +
enemyMove.Name + ".", location, Color.Black);

                if (enemyMove.Target == Target.Enemy && playerMove.MoveType ==
MoveType.Attack)
                {
                    location.Y += avatarFont.LineSpacing;

                    if (Avatar.GetMoveModifier(enemyMove.MoveElement, player.Element) < 1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is not very effective.",
location, Color.Black);
                    }
                    else if (Avatar.GetMoveModifier(enemyMove.MoveElement, player.Element) >
1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is super effective.",
location, Color.Black);
                    }
                }
            }

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);
            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);

            playerHealth = (float)player.CurrentHealth / (float)player.GetHealth();
            MathHelper.Clamp(playerHealth, 0f, 1f);
            playerHealthRect.Width = (int)(playerHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, playerHealthRect, healthSourceRect,
Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, enemyBorderRect, Color.White);

            enemyHealth = (float)enemy.CurrentHealth / (float)enemy.GetHealth();
            MathHelper.Clamp(enemyHealth, 0f, 1f);
            enemyHealthRect.Width = (int)(enemyHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, enemyHealthRect, healthSourceRect,
Color.White);
```

```
            GameRef.SpriteBatch.DrawString(avatarFont, player.Name, playerName,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, enemy.Name, enemyName, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerMiniRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyMiniRect, Color.White);

            GameRef.SpriteBatch.End();
        }

        public void SetAvatars(Avatar player, Avatar enemy)
        {
            this.player = player;
            this.enemy = enemy;

            if (player.GetSpeed() >= enemy.GetSpeed())
            {
                turn = CurrentTurn.Players;
            }
            else
            {
                turn = CurrentTurn.Enemies;
            }
        }

        public void SetMoves(IMove playerMove, IMove enemyMove)
        {
            this.playerMove = playerMove;
            this.enemyMove = enemyMove;
        }

        public void Start()
        {
            cTimer = TimeSpan.Zero;
            dTimer = TimeSpan.Zero;
            first = true;
        }

        #endregion
    }
}
```

There is first an enumeration CurrentTurn with values Player and Enemy. These values determine who's turn it is during this round of combat. There is then an interface IDamageState that defines the public members that can be called. They are SetAvatars which is the same as the other two methods, SetMoves which sets what moves each avatar is going to attempt to use and Start which starts the timers for this state.

Just like the other states there are a lot of member variables for positioning and drawing the visual elements. In hind sight it probably would have been better to make those variables protected in the BattleState class and inherit these two classes from that class instead of duplicating these variables in these other classes. There is also a member variable names turn that holds whose turn it is. The next new members are cTimer and dTimer. These are used to measure how much time has passed after a certain point. There are also IMove members that hold what move the avatars are going to use. Finally, first holds who goes first, the player or the enemy.

This constructor works like the other ones in that it just positions elements at certain points on the screen. The LoadContent method just loads the content for displaying the avatars and their properties.

The Update method is where the magic happens, so to speak. There is an if statement that checks a number of conditions. It checks if any of the following are true and if the time passed in dTimer is greater than 3. Those conditions are cTimer is greater than 4 seconds, the enemy avatar is not alive or the player avatar is not alive. Inside that if I check to see if either the player avatar or enemy avatar are not alive. If this is true I pop the current state off the stack, the damage state, and push the BattleOver state onto the stack. I then call the SetAvatars method passing in the player and enemy avatars. If they are both still alive I just pop this state off this stack which returns control back to the battle state.

There is then an if that check that cTimer is greater than 2 seconds, the first member variable is true and both avatars are in good health. In this case first means that if this is the first move resolved or the second move resolved. In this case I want to switch and resolve the second move. I set first to false. I then reset the duration of the dTimer memeber variable. I check the turn variable to see who's turn it is. If it is player I switch that member variable to be the enemies turn and ResolveMove to resolve the enemies move. Same in reverse for the else step.

The else if checks to see if cTimer is zero. That means that this is the first time that the Update method has been called. In that case I reset dTimer to 0 and call the ResolveMove method on whatever avatar's turn it is.

The last thing that I do is increment the two timer variables by adding in the ElapsedGameTime property of the gameTime parameter that is passed to Update. You will want to play with the duration of the timers to have them match what you are expecting. You could also play an animation when each avatar attacks. I will do just that in a future tutorial.

In the Draw method I draw the scene. Much of it will be familiar from the other two states. What is new is that I draw the out come of the last move resolved. First, I display what move the avatar has used. I then move to the next line and display if it wasn't effective or if it was super effective, based on the element of the move used and the element of the defending avatar.

The SetAvatars method assigns the member variables for the avatars to the values passed in. I then check if the player's speed is greater than or equal to the opponent's speed. If it is the first turn is the player's turn otherwise the enemy resolves its move first.

SetMoves just sets the moves to the values that are passed in. Start resets the two timers to zero and resets that first member variable to true so that both moves will be resolved.

The last thing to do is to is to have the Update method of the BattleState class drive the choices for the combat. Change the code of the Update method in BattleState to the following.

```
public override void Update(GameTime gameTime)
{
    PlayerIndex? index = PlayerIndex.One;

    if (Xin.CheckKeyReleased(Keys.P))
        manager.PopState();

    combatScene.Update(gameTime, index.Value);

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        manager.PushState((DamageState)GameRef.DamageState, index);
        GameRef.DamageState.SetAvatars(player, enemy);
        IMove enemyMove = null;

        do
        {
            int move = random.Next(0, enemy.KnownMoves.Count);
            int i = 0;

            foreach (string s in enemy.KnownMoves.Keys)
            {
                if (move == i)
                    enemyMove = (IMove)enemy.KnownMoves[s].Clone();
                i++;
            }

        } while (!enemyMove.Unlocked);


GameRef.DamageState.SetMoves((IMove)player.KnownMoves[combatScene.OptionText].Clone(),
enemyMove);
        GameRef.DamageState.Start();

        player.Update(gameTime);
        enemy.Update(gameTime);
    }

    Visible = true;

    base.Update(gameTime);
}
```

So, what is new that there is a check to see if the space bar or enter key have been released, meaning that the player has made their selection. If they have I push the damage state onto the stack of states. I then call the SetAvatars method passing in the two avatars. Next is a local variable of type IMove that will hold the move the enemy avatar will use. Follow that is a do while loop that generates a random number in the range of all known moves for the avatar. In a foreach loop I then iterate over the keys in the KnownMoves collection. If the selected move matches an variable that increments during each iteration of the loop I set enemyMove to be a clone of that move. I then check to see if that moves is unlocked or not. If it is not I repeat the process.

The next step is that I call the SetMoves method on the damage state to set the moves they are going to be applied. I also call the Start method to reset the timers for the state. Finally I call the Update method of the player and enemy avatars. This allows any effects that have a duration to count down and remove themselves. I also assign the Visible property of the state to true. This keeps this state visible while I draw the damage state.

You can now build and run the game. If you move the player next to one of the two avatars and press the B key you will be able to start a battle with the character's avatar and play through the battle.

I'm going to end the tutorial here as I like to keep the tutorials to a reasonable length so there is not a lot of new code to digest at once. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 13

Leveling Up

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.

What I am going to tackle first in this tutorial is to add the ability to level up avatars after a battle. There are two ways that you could handle an avatar levelling up. One ways is that you could increase the avatar's status automatically in code or you can give the player the ability to assign points to an avatar's stats. I personally like giving the player choices so I went with that route.

First, you will want to download the content that I used for this tutorial. You can download the content using this link A Summoner's Tale Content 13. First, let's add the background for the level up state. In the solution explorer expand the Content folder and then the GameScreens folder. Right click the GameScreens folder, select Add and then Existing Item. Add the levelup-menu.png to this folder. Now open the Content.mgcb by double clicking on it. Right click on the GameScreens folder select Add and then Existing Item. Add the levelup-menu.png that we just added to that folder. Hit <ctrl>+s to save the changes and then under the Build menu select Build to build the content.

The next thing that I want to do is add a new state for levelling up and avatar. Right click the GameStates folder, select Add and then Class. Name this class LevelUpState. Here is the code for that state.

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

namespace Avatars.GameStates
```

```
{
    public interface ILevelUpState
    {
        void SetAvatar(Avatar playerAvatar);
    }

    public class LevelUpState : BaseGameState, ILevelUpState
    {
        #region Field Region

        private Rectangle destination;
        private int points;
        private int selected;
        private SpriteFont font;
        private Avatar player;
        private Dictionary<string, int> attributes = new Dictionary<string, int>();
        private Dictionary<string, int> assignedTo = new Dictionary<string, int>();
        private Texture2D levelUpBackground;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public LevelUpState(Game game)
            : base(game)
        {
            attributes.Add("Attack", 0);
            attributes.Add("Defense", 0);
            attributes.Add("Speed", 0);
            attributes.Add("Health", 0);
            attributes.Add("Done", 0);

            foreach (string s in attributes.Keys)
                assignedTo.Add(s, 0);
        }

        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            levelUpBackground = GameRef.Content.Load<Texture2D>(
                @"GameScreens\levelup-menu");

            font = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");

            destination = new Rectangle(
                (Game1.ScreenRectangle.Width - levelUpBackground.Width) / 2,
                (Game1.ScreenRectangle.Height - levelUpBackground.Height) / 2,
                levelUpBackground.Width,
                levelUpBackground.Height);

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            PlayerIndex index = PlayerIndex.One;
            int i = 0;
            string attribute = "";

            if (Xin.CheckKeyReleased(Keys.Down))
            {
                selected++;

                if (selected >= attributes.Count)
                    selected = attributes.Count - 1;
            }
            else if (Xin.CheckKeyReleased(Keys.Up))
            {
                selected--;

                if (selected < 0)
                    selected = 0;
            }

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                if (selected == 4 && points == 0)
                {
                    foreach (string s in assignedTo.Keys)
                    {
                        player.AssignPoint(s, assignedTo[s]);
                    }

                    manager.PopState();
                    manager.PopState();
                    manager.PopState();
                    return;
                }
            }

            int increment = 1;

            if (Xin.CheckKeyReleased(Keys.Right) && points > 0)
            {
                foreach (string s in assignedTo.Keys)
                {
                    if (s == "Done")
                        return;

                    if (i == selected)
                    {
                        attribute = s;
                        break;
                    }

                    i++;
                }


                if (attribute == "Health")
                    increment *= 5;

                points--;
                assignedTo[attribute] += increment;

                if (points == 0)
                    selected = 4;
            }
            else if (Xin.CheckKeyReleased(Keys.Left) && points <= 3)
            {
                foreach (string s in assignedTo.Keys)
                {
                    if (s == "Done")
                        return;

                    if (i == selected)
                    {
                        attribute = s;
                        break;
                    }

                    i++;
                }

                if (assignedTo[attribute] != attributes[attribute])
                {
                    if (attribute == "Health")
                        increment *= 5;

                    points++;
                    assignedTo[attribute] -= increment;
                }
            }

            base.Update(gameTime);
        }

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


            GameRef.SpriteBatch.Begin();
            GameRef.SpriteBatch.Draw(levelUpBackground, destination, Color.White);

            Vector2 textPosition = new Vector2(destination.X + 5, destination.Y + 5);

            GameRef.SpriteBatch.DrawString(font, player.Name, textPosition, Color.Black);
            textPosition.Y += font.LineSpacing * 2;

            int i = 0;

            foreach (string s in attributes.Keys)
            {
                Color tint = Color.Black;

                if (i == selected)
                    tint = Color.Red;

                if (s != "Done")
                {
                    GameRef.SpriteBatch.DrawString(font, s + ":", textPosition, tint);
                    textPosition.X += 125;

                    GameRef.SpriteBatch.DrawString(font, attributes[s].ToString(),
textPosition, tint);
                    textPosition.X += 40;

                    GameRef.SpriteBatch.DrawString(font, assignedTo[s].ToString(),
textPosition, tint);
                    textPosition.X = destination.X + 5;

                    textPosition.Y += font.LineSpacing;
                }
                else
                {
                    GameRef.SpriteBatch.DrawString(font, "Done", textPosition, tint);
                    textPosition.Y += font.LineSpacing * 2;
                }
                i++;
            }

            GameRef.SpriteBatch.DrawString(font, points.ToString() + " point left.",
textPosition, Color.Black);
            GameRef.SpriteBatch.End();
        }

        public void SetAvatar(Avatar playerAvatar)
        {
            player = playerAvatar;

            attributes["Attack"] = player.BaseAttack;
            attributes["Defense"] = player.BaseDefense;
            attributes["Speed"] = player.BaseSpeed;
            attributes["Health"] = player.BaseHealth;

            assignedTo["Attack"] = player.BaseAttack;
            assignedTo["Defense"] = player.BaseDefense;
            assignedTo["Speed"] = player.BaseSpeed;
            assignedTo["Health"] = player.BaseHealth;

            points = 3;
            selected = 0;
        }

        #endregion
    }
}
```

First, there is an interface that I added for the state like the other game states. It has a single method in it, SetAvatar, that is used to pass the avatar that is to be levelled up to the state. The class then inherits from the BaseGameState abstract class so it can be used by the state manager and implements the interface that was just defined above.

For field in the class there is a Rectangle that will hold the destination of the level up state on the screen. Next there is a integer that holds the number of points that are available to be assigned to the avatar. There is also a field selected that holds the current option selected in the game state for assigning points. Since the state renders text there is a SpriteFont field for drawing text. There is also an Avatar type field that will hold the avatar that we will be updating. There are then two dictionaries that hold the attributes that the avatar currently has, attributes, and the assigned point, assignedTo. The last field that I included in this class is a field to hold the image for the level up state.

I added a single constructor to this game state. What it does is create the two dictionaries and initializes their values to 0. I also included a Done attribute that will be displayed with the other attributes that can be selected when all points have been assigned to.

In the LoadContent method I first load the image for the level up state into its field. I then load the font into its field as well. Next up I center the level up background image on the screen. Finally I call the LoadContent method on the base class to load any base content.

The way the level up state was designed to work is that there is the list of attributes and a Done option. When an attribute is selected if the player presses the left key the selected attribute will have an assigned point taken away and if the right key was selected a point will be added. Once all of the attribute points have been assigned they can choose the Done option to assign the points.

So, in the Update method I handle this logic. First, there are local variables to hold what the selected index and item are. I then check to see if the Down key was pressed. If it was pressed I increment the selected field of this class. I then check to see if that value is greater than or equal to the number of elements in the dictionary. If it is I reset selected back to the last element in the list. I do the same thing for the Up key but in reverse. I make sure that the value is never below 0.

After checking for up and down I check if the Space and Enter keys were pressed. If one of them was pressed I then check if the selected item is the last item, Done, and that there are no remaining points to be assigned. If those conditions are true I call the AssignPoint method on the player avatar to assign any points to that attribute. I then remove the states that were added to the stack.

There is then a local variable called increment. This value holds how many points are assigned based on the selected attribute. All attributes but health add 1 point to the selected attribute. The health attributes add 5 points to the avatar's health.

Now, I check to see if the Right key has been released and that there are free points to spend. If there are I loop through all of the keys in the assignedTo dictionary. If the selected option is Done then I exit the method. I then compare the variable i with the field selected. If they are the same I set attribute to be the current key. I then check if attribute is Health and if it is I multiply the value by 5. I then subtract 1 from the points that are available and increment that key in the dictionary. Finally, if there are no points left to assigned I set the selected attribute to the Done option.

I do the same thing in the case where I'm checking if the Left key was released if there are assigned points that can be moved. Next is the same loop that checks to see what attribute is currently selected and if it is Done exit the method. There is then an if statement that validates that the player can remove a point from that category.

The Draw method is pretty straight forward. First, draw the image for the background of the state. Next, create a local variable that determines where text will be positioned. Next I write the name of the avatar and then increment the position the next value will be written to. There is then a local variable i that is used for indexing items. In a foreach loop I go over the keys in the dictionary. I have a tint colour of Black that determines what colour to draw text in. If the current option is the selected option I change the tint colour to Red.

If the option is not the Done option I draw the attribute in the tint colour. I update the X value of the position to draw the individual parts and then reset it back to its default value and increment the Y spacing attribute. I then increase the i local variable and go onto the next iteration of the loop. If it is Done I draw Done and then increase the Y position so there is space between Done and the last bit of text to be drawn. The last bit of text to be draw is how many points are left to be assigned.

Finally there is a SetAvatar method that sets the Avatar being used in the LevelUpState. What it does is set the Avatar field player to be the avatar passed in. It then sets the values of the two dictionaries based off of that avatar. Finally it resets the points and selected fields.

Now that we have a LevelUpState class we need to add it to the game. Open the Game1.cs file. Where the other game states are add the following field.

```
        ILevelUpState levelUpState;
```

With the other game state properties add this property to expose the field to other classes.

```
        public ILevelUpState LevelUpState
        {
            get { return levelUpState; }
        }
```

Update the constructor to initialize the LevelUpState that we just created.

```
        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);
            conversationState = new ConversationState(this);
            battleState = new BattleState(this);
            battleOverState = new BattleOverState(this);
            damageState = new DamageState(this);
            levelUpState = new LevelUpState(this);

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

            characterManager = CharacterManager.Instance;
        }
```

The next step will be to call this state when a battle is over to see if the avatar needs to be levelled up. That will be done in the BattleOverState in the Update method. Update that method to the following.

```
        public override void Update(GameTime gameTime)
        {
            PlayerIndex? index = PlayerIndex.One;

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                if (levelUp)
                {
                    manager.PushState((LevelUpState)GameRef.LevelUpState,
PlayerIndexInControl);
                    GameRef.LevelUpState.SetAvatar(player);

                    this.Visible = true;
                }
                else
                {
                    manager.PopState();
                    manager.PopState();
                }
            }

            base.Update(gameTime);
        }
```

All that the new code does is if the field levelUp is true is push the level up state on top of the stack of game states and then sets the avatar that levelled up to the current avatar in use. What I've found is the Wind avatar almost always wins the battle against the Fire avatar in my testing, which is part of the reason you need to do a lot of testing of your game play elements. You might think that something works fine but in reality it is broken and the player will not be able to get past a certain point. In order to trigger an avatar level up I modified the CheckLevelUp method of the Avatar class. Update that method to the following code.

```
        public bool CheckLevelUp()
        {
            bool leveled = false;

            if (experience >= 50 * (1 + (long)Math.Pow((level - 1), 2.5)))
            {
                leveled = true;
                level++;
            }

            return leveled;
        }
```

All that this does is lower the amount of experience required to reach level 2 so fighting and losing will still cause the player's avatar to level up. I also want to update the AssignPoint method of the Avatar class because I'm controlling the way points are assigned in the LevelUpState instead of the Avatar class. Update that method to the following.

```
        public void AssignPoint(string s, int p)
        {
            switch (s)
            {
                case "Attack":
                    attack += p;
                    break;
                case "Defense":
                    defense += p;
                    break;
                case "Speed":
                    speed += p;
                    break;
                case "Health":
                    health += p;
                    break;
            }
        }
```

All that changes here is that when I update the health attribute it just uses the value passed in instead of 5 times the value passed in.

There is one minor issue still. You can continuously battle the other characters and train your avatar infinitely. There would be diminishing returns because as you level up when you battle you will gain less and less experience. It would be better if once you battled them you could not battle them again, like in Pokemon. It raises an issue with my story line. If you are fighting with summoned beings, how are you going to handle random encounters? I will leave the latter for another tutorial but I will cover limiting the number of times you can battle an NPC.

In order to do that I added a new property to the ICharacter interface, Battled. Update that interface to the following.

```
    public interface ICharacter
    {
        string Name { get; }
        bool Battled { get; set; }
        AnimatedSprite Sprite { get; }
        Avatar BattleAvatar { get; }
        Avatar GiveAvatar { get; }
        string Conversation { get; }
        void SetConversation(string newConversation);
        void Update(GameTime gameTime);
        void Draw(GameTime gameTime, SpriteBatch spriteBatch);
    }
```

The next step will be to update the Character class to include this update. In this case I'm just going to use an auto-property rather than having a field and work with the field using the property. Add the following line with the other properties in the Character class.

```
        public bool Battled { get; set; }
```

The last step will be that in the GamePlayState class when checking if the player triggered a battle check if the player has battled that character previously and if they have not go to the battle state. If they go to the battle state you need to update that property. Rather than paste the entire Update method I'm only pasting the if statement that checks if the B key was released.

```
            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in map.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                            PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }
```

I'm going to end the tutorial here as I like to keep the tutorials to a reasonable length so there is not a lot of new code to digest at once. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 14

Changing Maps

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.

In this tutorial I'm going to add the ability for the player to switch between maps. I will be adding a small building that the player can enter. To do this in my game I added in a new layer that I called a portal layer. I defined a portal as a tile that leads somewhere else. It does not have to lead to a different map but most often they will. So, you can also use a portal to move the player from one position on the map to a different position on the map, like a teleporter.

Let's get started. Open up your solution from the last time. Right click the TileEngine folder, select Add and then Class. Name this new class Portal. This class defines the properties and methods of a portal. Here is the code for this class.

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

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

        Point sourceTile;
        Point destinationTile;
        string destinationLevel;

        #endregion

        #region Property Region
        [ContentSerializer]
        public Point SourceTile
        {
            get { return sourceTile; }
            private set { sourceTile = value; }
        }

        [ContentSerializer]
        public Point DestinationTile
        {
            get { return destinationTile; }
            private set { destinationTile = value; }
        }

        [ContentSerializer]
        public string DestinationLevel
        {
            get { return destinationLevel; }
            private set { destinationLevel = value; }
        }

        #endregion

        #region Constructor Region

        private Portal()
        {
        }

        public Portal(Point sourceTile, Point destinationTile, string destinationLevel)
        {
            SourceTile = sourceTile;
            DestinationTile = destinationTile;
            DestinationLevel = destinationLevel;
        }

        #endregion
    }
}
```

So, a portal had three properties in my game. It had a Point that held the X and Y coordinates of where the portal was located on the map. It had another Point that held the X and Y coordinates of the destination tile for the portal. It also had a string variable the held the name of the map/level that the portal led to.

I added three variables to the class to hold those three properties: sourceTile, destinationTile, and destinationLevel. I then included three properties that exposed their values but could not be set outside of the class. The reason for doing this is two fold. First, you typically do not want to change the value of a portal. In some instances you may want a portal to change where it is located or where it leads to but usually this is not the norm. I also made the set accessors private because there are required to serialize and deserialize maps. It is also why I included a private constructor that takes no parameters and has no actions in it. Finally, there is a constructor that takes as parameters, the source tile, the destination tile and the destination level. It then assigns those values to the fields in the class. You will also so that I marked all of the properties with the ContentSerializer attribute so that they are serialized and deserialized using the Intermediate Serializer.

Now that there is a class that represents a Portal I now added a layer to the map called PortalLayer. It was a collection of Portal objects on the map. Keep in mind when developing tile maps your layers don't necessarily have to display graphics. They can also hold metadata about the map, in this case the portals on the map. Now, right click on the TileEngine folder, select Add and then Class. Name this new class PortalLayer.

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

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

        private Dictionary<Rectangle, Portal> portals;

        #endregion

        #region Property Region

        [ContentSerializer]
        public Dictionary<Rectangle, Portal> Portals
        {
            get { return portals; }
            private set { portals = value; }
        }

        #endregion

        #region Constructor Region

        public PortalLayer()
        {
            portals = new Dictionary<Rectangle, Portal>();
        }

        #endregion
    }
}
```

So, what we have here is a single field called portals, a property that exposes it for public read access but private write access called Portals and a constructor that creates the field. It is also marked with the ContentSerializer attribute so that it will be serialized and deserialized by the Intermediate Serializer.

Now, the portal layer needs to be integrated into the existing TileMap class. The changes were made so that it would not break any of the current functionality. Open the TileMap class and update the code field, property and constructor regions.

```
        #region Field Region

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

        [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 PortalLayer PortalLayer
        {
            get { return portalLayer; }
            private set { portalLayer = 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, PortalLayer portals = null)
        {
            this.characters = new Dictionary<string, Point>();
            this.tileSet = tileSet;
            this.mapName = mapName;
            characterManager = CharacterManager.Instance;

            portalLayer = portals != null ? portals : new PortalLayer();
        }

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

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

        #endregion
```

The change here is I added a field to hold the PortalLayer associated with the TileMap, a property to expose it externally as read and internally as write. I marked this with the ContentSeralizier property so that it will be serialized and deserialized. I add a PortalLayer parameter as an optional parameter that has the value of NULL. What this does is it makes it so that the change will not require you to find everywhere you create a TileMap object and add a new parameter to the call. In the second constructor I check to see if the portals parameter is not NULL. If it is not NULL I assign the value to the local variable, otherwise I create a new PortalLayer. If you know that an if statement is going to just set values by checking if an object has a value or not you can use the ? operator to do that. It is read **condition ? true action : false action**. It is a nice shorthand and cleans up the code a bit.

There is one other metadata class that I want to add. This class holds all of the maps for the game. I called this class World. Right click the TileEngine folder, select Add and then Class. Name this new class World. Here is the code for that class.

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

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

        private Dictionary<string, TileMap> maps;
        private string currentMapName;

        #endregion

        #region Property region

        [ContentSerializer]
        public Dictionary<string, TileMap> Maps
        {
            get { return maps; }
            private set { maps = value; }
        }

        [ContentSerializer]
        public string CurrentMapName
        {
            get { return currentMapName; }
            private set { currentMapName = value; }
        }

        public TileMap CurrentMap
        {
            get { return maps[currentMapName]; }
        }

        #endregion

        #region Constructor Region
        public World()
        {
            maps = new Dictionary<string, TileMap>();
        }

        #endregion

        #region Method Region

        public void AddMap(string mapName, TileMap map)
        {
            if (!maps.ContainsKey(mapName))
                maps.Add(mapName, map);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
        {
            CurrentMap.Draw(gameTime, spriteBatch, camera);
        }

        public void ChangeMap(string mapName, Rectangle portalLocation)
        {
            if (maps.ContainsKey(mapName))
            {
                currentMapName = mapName;
                return;
            }

            throw new Exception("Map name or portal name not found.");
        }

        #endregion
    }
}
```

This was a very simple class in my game that was responsible for managing all of the maps in the game. There were two fields in the class. One is a dictionary of the maps in the game and the other is a string that holds the name of the current map. There are properties marked with the ContentSerializer attribute so that the world can be serialized and deserialized. This allowed me to load all of the maps in my game at once. I also had an editor that worked with the maps. I will need to clean it up and update it a bit but I will eventually be posting the map editor that I used in this game. There are a few methods in this class. The one adds a map to the world, the second draws the current map.

There is also a method called ChangeMap. This is the method that will change the current map with the desired map, if it exists in the collection of maps. You probably should add a bit more validation here to make sure the destination is inside the map that you are switching to.

Now, this needs to be added to the GamePlayState to incorporate it into the existing game. I'm going to do this in stages because there are a lot of changes to the state to incorporate it into the game. First, in the GamePlayState, replace the TileMap map field with the field World world like this.

```
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);
        World world;
        Camera camera;
        Player player;
```

Now, the Update method needs to be modified because it used the map field to search for characters on the map. Change the Update method 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 world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = GameRef.CharacterManager.GetCharacter(s);
                    Rectangle r = new Rectangle(
                        (int)world.CurrentMap.Characters[s].X * Engine.TileWidth + cp,
                        (int)world.CurrentMap.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(world.CurrentMap.WidthInPixels,
world.CurrentMap.HeightInPixels));
            }
            else
            {
                player.Sprite.IsAnimating = false;
            }

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f)
                    {
                        IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                        manager.PushState(
                            (ConversationState)conversationState,
                             PlayerIndexInControl);

                        conversationState.SetConversation(player, c);
                        conversationState.StartConversation();
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);
                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                            PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }
            base.Update(gameTime);
        }
```

All that I did in this method was replace all the instances of map with world.CurrentMap. I did the exact same thing in the Draw method. You can update it to the following code.

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

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

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

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

            GameRef.SpriteBatch.End();
        }
```

The last change was to SetUpNewGame. In this method I initialized the world field, added a local variable to hold the map. I then created the map, added it to the world and changed the map to be this new map.

```
        public void SetUpNewGame()
        {
            Texture2D spriteSheet = content.Load<Texture2D>(@"PlayerSprites\maleplayer");
            TileMap map = null;
            world = new World();

            player = new Player(GameRef, "Wesley", false, spriteSheet);
            player.AddAvatar("fire", AvatarManager.GetAvatar("fire"));
            player.SetAvatar("fire");

            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();

            ConversationManager.CreateConversations(GameRef);

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

            teacherOne.SetConversation("LanceHello");
            teacherTwo.SetConversation("MarissaHello");

            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));

            map.PortalLayer.Portals.Add(Rectangle.Empty, new Portal(Point.Zero, Point.Zero,
"level1"));
            world.AddMap("level1", map);

            world.ChangeMap("level1", Rectangle.Empty);

            camera = new Camera();
        }
```

At this point I found a problem with the last tutorial. I missed implementing a member of the ICharacter interface for the PCharacter class so the game will not build. Add the following propery to the PCharacter class.

```
        public bool Battled
        {
            get
            {
                throw new NotImplementedException();
            }
            set
            {
                throw new NotImplementedException();
            }
        }
```

It will need to be fleshed out at some point but I'm going to do that in another tutorial. So, if you build and run the game you can start a new game and everything will work as expected. You can have a conversation with the NPCs and you can battle them. You still can't switch to another map. The first reason is that there is no portal to another map and second is that we haven't implemented that ability in the Update method. First, I will add the ability to switch maps in the Update method. Next I will update the SetUpNewGame method so that it creates two maps with a portal on each map that leads between the two maps.

I'm going to go with the default action, space bar or enter key, to activate an object, portal, initiate a conversation, etc. To do that I added the following changes to the Update method.

```
        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 world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = GameRef.CharacterManager.GetCharacter(s);
                    Rectangle r = new Rectangle(
                        (int)world.CurrentMap.Characters[s].X * Engine.TileWidth + cp,
                        (int)world.CurrentMap.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(world.CurrentMap.WidthInPixels,
world.CurrentMap.HeightInPixels));
            }
            else
            {
                player.Sprite.IsAnimating = false;
            }

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f)
                    {
                        IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                        manager.PushState(
                            (ConversationState)conversationState,
                             PlayerIndexInControl);

                        conversationState.SetConversation(player, c);
                        conversationState.StartConversation();
                    }
                }

                foreach (Rectangle r in world.CurrentMap.PortalLayer.Portals.Keys)
                {
                    Portal p = world.CurrentMap.PortalLayer.Portals[r];

                    float distance = Vector2.Distance(
                        player.Sprite.Center,
                        new Vector2(
                            r.X * Engine.TileWidth + Engine.TileWidth / 2,
                            r.Y * Engine.TileHeight + Engine.TileHeight / 2));

                    if (Math.Abs(distance) < 64f)
                    {
                        world.ChangeMap(p.DestinationLevel, new Rectangle(p.DestinationTile.X,
p.DestinationTile.Y, 32, 32));

                        player.Position = new Vector2(
                            p.DestinationTile.X * Engine.TileWidth,
                            p.DestinationTile.Y * Engine.TileHeight);
                        camera.LockToSprite(world.CurrentMap, player.Sprite,
Game1.ScreenRectangle);

                        return;
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                            PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }
            base.Update(gameTime);
        }
```

Inside the if statement that checks if Space/Enter have been pressed I added another foreach loop that loops over the keys in the portal layer of the current map. I first get the portal using the key. I then calculate the distance between the character and the portal as I did for conversations and battling avatars. You will notice that I'm multiplying X and Y coordinates by TileWidth and TileHeight properties of the engine to get positions in pixels instead of tiles. If the portal and player are close enough together I change the map by calling ChangeMap passing in the name of the level and the destination rectangle. I then update the player's position to be the position on the new map. Then I lock the camera onto the player on the map and exit the Update method because there is no point in processing anything else in the method at that point.

The last thing that I'm going to cover in this tutorial is creating a basic second map with a portal that leads back to the same position as the first map. I did that in the SetUpNewGame method as that is where I already created a map and added it to the world. Change that map to the following.

```
public void SetUpNewGame()
{
    Texture2D spriteSheet = content.Load<Texture2D>(@"PlayerSprites\maleplayer");
    TileMap map = null;
    world = new World();

    player = new Player(GameRef, "Wesley", false, spriteSheet);
    player.AddAvatar("fire", AvatarManager.GetAvatar("fire"));
    player.SetAvatar("fire");
    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();

    building.SetTile(4, 4, 18);

    ConversationManager.CreateConversations(GameRef);

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

    teacherOne.SetConversation("LanceHello");
    teacherTwo.SetConversation("MarissaHello");

    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));

    map.PortalLayer.Portals.Add(Rectangle.Empty, new Portal(Point.Zero, Point.Zero, "level1"));
    map.PortalLayer.Portals.Add(new Rectangle(4, 4, 32, 32), new Portal(new Point(4, 4), new
Point(10, 10), "inside"));

    world.AddMap("level1", map);
    world.ChangeMap("level1", Rectangle.Empty);

    background = new TileLayer(20, 20, 23);
    edge = new TileLayer(20, 20);
    building = new TileLayer(20, 20);
    decor = new TileLayer(20, 20);

    map = new TileMap(set, background, edge, building, decor, "inside");
    map.FillEdges();
    map.FillBuilding();
    map.FillDecoration();
    map.BuildingLayer.SetTile(9, 19, 18);

    map.PortalLayer.Portals.Add(new Rectangle(9, 19, 32, 32), new Portal(new Point(9, 19), new
Point(4, 4), "level1"));

    world.AddMap("inside", map);

    camera = new Camera();
}
```

What changed as adding the map to the world is that I created a new ground layer filling it with tile 23 from the tile set we are using that is a floor like tile (close enough for me.) I then create the other layers at the same size. I then create the map, call the FillEdge, FillBuilding and FillDecoration methods. I then set one tile to be a door tile. I add a new portal that leads back to the original portal. I then add the map to world. I also set a tile on the first map to be the same door tile to give the visual cue that the player can open the door.

If you build and run the game now and go to the door and press Space/Enter you should now be able to switch to the new map. You can then walk to the far right of the map and switch back to the original map.

I'm going to end the tutorial here as I like to keep the tutorials to a reasonable length so there is not a lot of new code to digest at once. 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!
Cynthia McMahon

A Summoner's Tale – MonoGame Tutorial Series

Chapter 15

Saving Game State

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.

When I asked on the blog for tutorial suggestions one of the comments asked how one would go about saving game state. I will cover adding that feature to the game in this tutorial. The first question that needs to be answered is what exactly needs to be save? The main objects will be the player and their avatars. Other objects would be conversation states, quest states and other events.

First, you must ask yourself the question, "What needs to be saved?". There are a variety of approaches that could be taken. Only save changed data relevant to the current game or use a shotgun approach and save the entire world. This is the easier of the two to implement so I've decided to go with that approach. Let's get started.

First, I want to update the ICharacter interface to add a Save method that any class that implements the interface must implement. Find the ICharacter interface and update the code to the following.

```
using Avatars.AvatarComponents;
using Avatars.ConversationComponents;
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;
using System.IO;

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

The Save method will require that a BinaryWriter object be passed into it. This writer will be used to actually save the character to disk. Why did I decide on using a BinaryWriter instead of a TextWriter or saving as XML? The reason is the other two formats are easy to read and manipulate. A binary file is harder for the player to decode and manipulate. Ideally you'd want to also encrypt the files as well but I will leave that as an exercise.

Next, I want to add the same method to the base Avatar class, but not implement it quite yet. Open the Avatar class and add the following method. Make sure to include the following using statement at the top of the file as well.

```
using System.IO;

        public bool Save(BinaryWriter writer)
         {
            return true;
        }
```

I will implement this method shortly. The last place to add a Save method will be the World class. Again, I'm just adding the stub for now and the rest will be filled out later. Here is the code, including the required using statement.

```
using System.IO;

        public bool Save(BinaryWriter writer)
         {
            return true;
        }
```

The last class that I want to add a Save method to is the Player class. Add the same method and using statements as before, repeated here.

```
using System.IO;

        public bool Save(BinaryWriter writer)
         {
            return true;
        }
```

The method stub needs to be added to the TileMap class as well. Add the method stub and using statement to TileMap.

```
using System.IO;
```

```
        public bool Save(BinaryWriter writer)
        {
            return true;
        }
```

The next step will be to trigger the save process. I did that by checking if the F1 key has been released and if it has trigger the save event for the world. Modify the Update method as follows. Make sure to add a using statement for System.IO at the beginning of the file.

```
using System.IO;
        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 world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = GameRef.CharacterManager.GetCharacter(s);
                    Rectangle r = new Rectangle(
                        (int)world.CurrentMap.Characters[s].X * Engine.TileWidth + cp,
                        (int)world.CurrentMap.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(world.CurrentMap.WidthInPixels,
world.CurrentMap.HeightInPixels));
            }
            else
            {
                player.Sprite.IsAnimating = false;
            }

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f)
                    {
                        IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                        manager.PushState(
                            (ConversationState)conversationState,
                            PlayerIndexInControl);

                        conversationState.SetConversation(player, c);
                        conversationState.StartConversation();
                    }
                }

                foreach (Rectangle r in world.CurrentMap.PortalLayer.Portals.Keys)
                {
                    Portal p = world.CurrentMap.PortalLayer.Portals[r];
                    float distance = Vector2.Distance(
                        player.Sprite.Center,
                        new Vector2(
                            r.X * Engine.TileWidth + Engine.TileWidth / 2,
                            r.Y * Engine.TileHeight + Engine.TileHeight / 2));

                    if (Math.Abs(distance) < 64f)
                    {
                        world.ChangeMap(p.DestinationLevel, new Rectangle(p.DestinationTile.X,
p.DestinationTile.Y, 32, 32));

                        player.Position = new Vector2(
                            p.DestinationTile.X * Engine.TileWidth,
                            p.DestinationTile.Y * Engine.TileHeight);
                        camera.LockToSprite(world.CurrentMap, player.Sprite, Game1.ScreenRectangle);

                        return;
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                            PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.F1))
            {
                FileStream stream = new FileStream("avatars.sav", FileMode.Create,
FileAccess.Write);
                BinaryWriter writer = new BinaryWriter(stream);
                world.Save(writer);
                player.Save(writer);
                writer.Close();
                stream.Close();
            }
            base.Update(gameTime);
        }
```

First, there is of course the check to see if the F1 key has been released this frame. If it has I create a FileStream to write the file to disk. The location of this file will be inside the debug folder, for now. It will create a new file even if there is an existing file and open it with write permissions. I then create the BinaryWriter passing in the FileStream. Next step is to call the Save method on the World object and the Save method of the Player object passing in the BinaryWriter object. I then close both the writer and the stream.

The first Save method that I will implement is on the World class. Modify that class to the following code.

```
public bool Save(BinaryWriter writer)
{
    writer.Write(currentMapName);

    foreach (string s in maps.Keys)
        maps[s].Save(writer);

    return true;
}
```

What is happening here is first I'm writing the name of the current map the player is in. Next I loop through all of the TileMaps and call their save method. It is still just returning true but eventually we will add in some error checking to make sure nothing unusual happens and that we can recover from it.

Now with the World done we'll tackle the TileMap class. Update the Save method of the TileMap class to the following.

```
public bool Save(BinaryWriter writer)
{
    foreach (string s in characters.Keys)
    {
        ICharacter c = CharacterManager.Instance.GetCharacter(s);
        c.Save(writer);
    }

    return true;
}
```

In this case what we do is iterate over the list of characters on the map. Then we use the CharacterManager object to get the character. Finally we call the Save method on the Character. I had to make two tweaks here. First, I added a new field called textureName that will as the name implies store the name of the texture for this character. Second, I set the value of the variable in the FromString method. Here is the updated code for the field and methods.

```
private string textureName;

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];
    character.textureName = parts[1];
    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];

    character.battleAvatar = AvatarManager.GetAvatar(parts[4].ToLowerInvariant());

    return character;
}


public bool Save(BinaryWriter writer)
{
    StringBuilder b = new StringBuilder();
    b.Append(name);
    b.Append(",");
    b.Append(textureName);
    b.Append(",");
    b.Append(sprite.CurrentAnimation);

    writer.Write(b.ToString());

    if (givingAvatar != null)
        givingAvatar.Save(writer);

    if (battleAvatar != null)
        battleAvatar.Save(writer);

    return true;
}
```

In the FromString method I assign the new field, characterTexture name to the name of the texture for the character. In the Save method I create a StringBuilder object and append the fields of the class in the order that are required by the FromString method, other than the Avatars. I then call the Write method of the BinaryWriter to write the character to disk. Here is where you'd want to encrypt the string in some way. The string will show up as text in the file even though it's a binary file. After calling the Write method I call the Save method of the two Avatar objects in this class. There are null checks to make sure there is something to be written.

I'm now going to do something similar in the PCharacter class. I added the same field and updated the FromString method. Add this field, update the FromString method and update the Save method to the following. I also write the Avatar objects separately from the character being written.

```
private string textureName;

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];
    character.textureName = parts[1];
    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];
    character.currentAvatar = int.Parse(parts[4]);

    for (int i = 5; i < 11 && i < parts.Length; i++)
        character.avatars[i - 5] = AvatarManager.GetAvatar(parts[i].ToLowerInvariant());

    return character;
}

public bool Save(BinaryWriter writer)
{
    StringBuilder b = new StringBuilder();

    b.Append(name);
    b.Append(",");
    b.Append(textureName);
    b.Append(",");
    b.Append(sprite.CurrentAnimation);
    b.Append(",");
    b.Append(conversation);
    b.Append(",");
    b.Append(currentAvatar);

    writer.Write(b.ToString()

    foreach (Avatar a in avatars)
    {
        if (a != null)
            a.Save(writer);
    }

    return true;
}
```

Very similar to the Character class. The differences are that I updated to the FromString method to shift the fields one part of the string. I also updated the loop where we iterate over the list of avatars to use the new index. Similarly, I use a string builder to create an object that can be written to disk. Once the player stats have been written the next step is to write out the avatars. To do that I look over all of the avatars in the array and call their Save methods passing in the writer so they can be saved to disk. Again, there is a null check to make sure that there is something to be written to disk. I still only return true because we haven't implemented that part yet.

Now it is time to implement the Save method of the Avatar class. Find that method and update it to the following.

```
public bool Save(BinaryWriter writer)
{
    StringBuilder b = new StringBuilder();

    b.Append(name);
    b.Append(",");
    b.Append(element);
    b.Append(",");
    b.Append(experience);
    b.Append(",");
    b.Append(costToBuy);
    b.Append(",");
    b.Append(level);
    b.Append(",");
    b.Append(attack);
    b.Append(",");
    b.Append(defense);
    b.Append(",");
    b.Append(speed);
    b.Append(",");
    b.Append(health);
    b.Append(",");
    b.Append(currentHealth);


    foreach (string s in knownMoves.Keys)
    {
        b.Append(",");
        b.Append(s);
    }

    writer.Write(b);

    return true;
}
```

Just like in the other methods there is StringBuilder that I use to build the string to written to disk. I then append the fields of the class one by one with a comma afterwards. Instead of writing a comma after currentHealth I go straight to a loop. In the loop I append the comma and then the key for the move. Finally, I write the StringBuilder to disk.

I'm going to end this tutorial here. In the next tutorial I will reverse the process and load a game from disk. Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so.

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!
Cynthia McMahon

A Summoner's Tale – MonoGame Tutorial Series

Chapter 1

Getting Started

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 2015 versions as well.

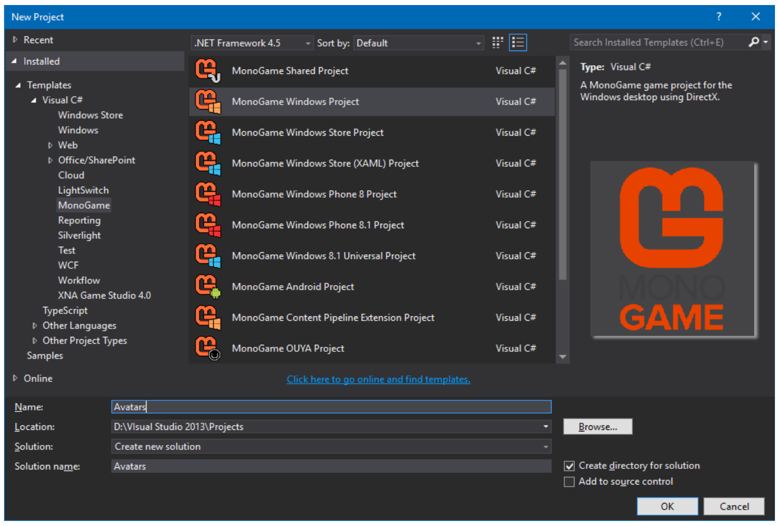
I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

First, let me give you a brief overview of the features of the finished game. This will be a 2D top down game similar to the Pokemon series of games. The player will be able to explore the world, interact with objects and non-player characters. They will battle against other players to gain experience. There will be a basic quest system as well.

The player character is a Summoner. Summoners are able to summon avatars from one of the elemental planes. There are six elemental planes: Light, Water, Air, Dark, Fire and Earth. There are many different types avatars and each avatar is attuned to a particular summoner. The summoner's avatars gain experience by fighting and defeating other avatars. The player can learn to summon other avatars by interacting with the non-player characters in the game or by finding rare spellbooks that contain the necessary rituals.

Combat between avatars will be the standard turn based combat. I hit you then you hit me and move to the next turn. Order will be based on the avatars' speed attribute. Each element is strong against one or more elements and weak against another.

To get started fire up Visual Studio and create a new MonoGame Windows Project. Call this new project Avatars. This is what my project looks like.


New Project
Recent
Installed
Templates
Visual C#
Windows Store
Windows
Web
Office/SharePoint
Cloud
LightSwitch
MonoGame
Reporting
Silverlight
Test
WCF
Workflow
XNA Game Studio 4.0
TypeScript
Other Languages
Other Project Types
Samples
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
MonoGame Shared Project Visual C#
MonoGame Windows Project Visual C#
MonoGame Windows Store Project Visual C#
MonoGame Windows Store (XAML) Project Visual C#
MonoGame Windows Phone 8 Project Visual C#
MonoGame Windows Phone 8.1 Project Visual C#
MonoGame Windows 8.1 Universal Project Visual C#
MonoGame Android Project Visual C#
MonoGame Content Pipeline Extension Project Visual C#
MonoGame OUYA Project Visual C#
Type: Visual C#
A MonoGame game project for the Windows desktop using DirectX.
MONO GAME
Click here to go online and find templates.
Browse...
OK
Cancel

Name: Avatars
Location: D:\Visual Studio 2013\Projects
Solution: Create new solution
Solution name: Avatars
☑ Create directory for solution
☐ Add to source control

I always like adding in some "plumbing" before I start work on game play. Typically the first thing that I add is the game state manager. Right click the Avatars project, select Add and then New Folder. Name this folder StateManager. Now right click the StateManager folder, select Add and then Class. Name this class GameState. Right click the StateManager folder again, select Add and then Class. Name this class GameStateManager.

Open the GameState class and replace the code in that class with the following code. If you've not read one of my tutorials before I prefer to let you read the code and then explain what it is doing.

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

namespace Avatars.StateManager
{
    public interface IGameState
    {
        GameState Tag { get; }
        PlayerIndex? PlayerIndexInControl { get; set; }
    }

    public abstract partial class GameState : DrawableGameComponent, IGameState
    {
        #region Field Region

        protected GameState tag;
        protected readonly IStateManager manager;
        protected ContentManager content;
        protected readonly List<GameComponent> childComponents;

        protected PlayerIndex? indexInControl;

        public PlayerIndex? PlayerIndexInControl
        {
            get { return indexInControl; }
            set { indexInControl = value; }
        }

        #endregion
        #region Property Region

        public List<GameComponent> Components
        {
            get { return childComponents; }
        }

        public GameState Tag
        {
            get { return tag; }
        }

        #endregion

        #region Constructor Region

        public GameState(Game game)
            : base(game)
        {
            tag = this;

            childComponents = new List<GameComponent>();
            content = Game.Content;

            manager = (IStateManager)Game.Services.GetService(typeof(IStateManager));
        }

        #endregion

        #region Method Region

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

        public override void Update(GameTime gameTime)
        {
            foreach (GameComponent component in childComponents)
                if (component.Enabled)
                    component.Update(gameTime);

            base.Update(gameTime);
        }

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

            foreach (GameComponent component in childComponents)
                if (component is DrawableGameComponent &&
((DrawableGameComponent)component).Visible)
                    ((DrawableGameComponent)component).Draw(gameTime);
        }

        protected internal virtual void StateChanged(object sender, EventArgs e)
        {
            if (manager.CurrentState == tag)
                Show();
            else
                Hide();
        }

        public virtual void Show()
        {
            Enabled = true;
            Visible = true;

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = true;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = true;
            }
        }

        public virtual void Hide()
        {
            Enabled = false;
            Visible = false;

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = false;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = false;
            }
        }

        #endregion
    }
}
```

First, there is an interface that can be applied to other game states that are going to be implemented in the game. The two items that will need to implemented in any class that this are a Tag property and a PlayerIndexInControl. The first property is to make retrieving another state from the state manager in for use in other game states. PlayerIndexInControl was added to allow for the use of XBOX 360 controllers. I'm keeping it in because players can still connect controllers to their computers and use those for input rather than mouse and keyboard.

The GameState class is an abstract class that has methods that can be overridden in other game states and protected members that will be common to all game states. The GameState class derives from DrawableGameComponent. This adds LoadContent, Update and Draw methods to the game state so they can be called from other classes. It also implements the IGameState interface. In theory you could add the IGameState members to the abstract class. I just have always used this approach since I first discovered it.

There are a few fields in this class. The first is a field for the Tag property that needs to be implemented from IGameState. There is a readonly field of type IStateManager that gives us access to the GameStateManager that we will be implementing. I added a ContentManager field as well that will allow game states to load content. Next is a list of child game components. This allows us to add other GameComponents to the state. These components can be updated and rendered from the parent component. The last field is used to implement the PlayerIndexInControl property from the interface.

There are public properties to implement the two interface properties. There is also a property that will expose the child components of the GameState class. This will be useful if you need to add a component from one GameState to another GameState. An example is after creating the player object it can be passed from the character generator to the game play state in this way.

The constructor just initializes some of the fields. The interesting one is the way that the IStateManager is retrieved. The GameStateManager will register itself as a service. Once it is registered as a service it can be retrieved using GameSerivceContainer.Services method. This method takes the type of service to be retrieved as a parameter.

LoadContent method just calls the base.LoadContent method. The Update method loops over all of the child components in a foreach loop. It checks to see if the component is enabled. If it is enabled it will call the Update method of that component. Draw works pretty much the same as Update. It just checks if the component can be drawn and if it is visible. If both are true it will render the component.

Next is an event handler, similar to what you'd find in a WinForms project. It will be called to notify all game states that there is going to a change in the active component. It checks to see if this state is now the current state. If it is the current state at calls the Show method that sets the Visible and Enabled properties of the component to true. It then loops through the child components and enables them. It also checks to see if the component is drawable and if it is it will set the Visible property to true. If it is not the current state it calls the Hide method that works in the reverse of the Show method. It will set the Enabled and Visible property to false as well as setting the applicable property of child components to false as well.

That is it for the GameState class. Now open the GameStateManager class and replace the code with the following.

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

namespace Avatars.StateManager
{
    public interface IStateManager
    {
        GameState CurrentState { get; }

        event EventHandler StateChanged;

        void PushState(GameState state, PlayerIndex? index);
        void ChangeState(GameState state, PlayerIndex? index);
        void PopState();
        bool ContainsState(GameState state);
    }

    public class GameStateManager : GameComponent, IStateManager
    {
        #region Field Region

        private readonly Stack<GameState> gameStates = new Stack<GameState>();

        private const int startDrawOrder = 5000;
        private const int drawOrderInc = 50;
        private int drawOrder;

        #endregion

        #region Event Handler Region

        public event EventHandler StateChanged;

        #endregion
        #region Property Region

        public GameState CurrentState
        {
            get { return gameStates.Peek(); }
        }

        #endregion

        #region Constructor Region

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

        #endregion

        #region Method Region

        public void PushState(GameState state, PlayerIndex? index)
        {
            drawOrder += drawOrderInc;
            AddState(state, index);
            OnStateChanged();
        }

        private void AddState(GameState state, PlayerIndex? index)
        {
            gameStates.Push(state);
            state.PlayerIndexInControl = index;
            Game.Components.Add(state);
            StateChanged += state.StateChanged;
        }

        public void PopState()
        {
            if (gameStates.Count != 0)
            {
                RemoveState();
                drawOrder -= drawOrderInc;
                OnStateChanged();
            }
        }

        private void RemoveState()
        {
            GameState state = gameStates.Peek();

            StateChanged -= state.StateChanged;
            Game.Components.Remove(state);
            gameStates.Pop();
        }

        public void ChangeState(GameState state, PlayerIndex? index)
        {
            while (gameStates.Count > 0)
                RemoveState();

            drawOrder = startDrawOrder;
            state.DrawOrder = drawOrder;
            drawOrder += drawOrderInc;

            AddState(state, index);
            OnStateChanged();
        }

        public bool ContainsState(GameState state)
        {
            return gameStates.Contains(state);
        }

        protected internal virtual void OnStateChanged()
        {
            if (StateChanged != null)
                StateChanged(this, null);
        }

        #endregion
    }
}
```

The GameStateManager will track states using a stack. If you're not familiar with stacks a stack is a data structure that models a stack you find in the real world, like a stack of plates. Stacks implement what is called last in first out, or LIFO, model. Continuing with the stack of plates analogy when adding a plate you place it on the top of the stack and it will be the last plate added. When you go to get a plate you take the plate off the top of the stack. (We have an invisible barrier around the stack that prevents you from going and retrieving a plate other than the top plate.) I n code when you add an item to a stack you “push” it onto the stack. Similarly when you remove an item from the stack you “pop” it off the stack.

There is another interface in this class. This interface will be used when registering the state manager as a service. It has a property that returns the current game state. It also has an event that must be implemented. This event will trigger the event handler in all active games states, the ones on the stack. There are then methods that expose the functionality of the state manager. There is one to add a new state to the stack, PushState. Another, PopState, that will remove the current state off the stack. The ChangeState method removes all states off the stack and pushes the new state onto the stack. Finally, ContainsState is used to check if a state exists on the stack.

The GameStateManager inherits from GameComponent so it can be registered as a service and have its Update method called automatically. It also implements the interface that I was just speaking off. There is a read only member variable, gameStates, that is a Stack<GameState> that will be used in implementing the state manager. DrawableGameComponents have a DrawOrder associated with them. Components with higher values will be drawn before components with lower values. So, there is a field that is used for assigning game states a draw order. There are constants that hold the initial draw order of a component and an increment that will be added when a new state is pushed onto the stack or decremented when a state is popped off the stack.

There is also the event handler that needs to be implemented from the interface. If a component has subscribed to the event its internal handler will be called when this event is raised. The CurrentState is property uses the Peek method to return the state on the top of the stack.

The constructor is where we add the state manager as a service. That is done using the GameServiceContainer.AddService method. It works in opposite of the method that we used earlier to retrieve the service.

PushState receives the game state to be added and a PlayerIndex? parameter for the game controller in use. If you don't want to support controllers you can pass null whenever you require this parameter or leave it out entirely. It increments the draw order a calls a function AddState that will add the state to the stack. It then calls the OnStateChanged method that will raise the event for state change.

The AddState method pushes the state onto the stack. It updates the PlayerIndexInControl memeber of the state. Next it adds the state to the list of components for the game. Finally it subscribes the state to the StateChanged event.

PopState checks to see if there is a state on the stack. If there is not it calls RemoveState and decrements the draw order for the components. It also calls the OnStateChanged method to raise the state changed event.

RemoveState uses Peek to get the state that is on top of the stack. It unsubscribes the state from the event handler so it will no longer be notified. Next is removes the state form the list of game components. Finally it pops the state of the stack.

As I mentioned earlier ChangeState removes all states on the stack and then pushes the new state onto the stack. The first thing that it does is loop until there are no states on the stack and call the RemoveState method to remove the state from the stack. It then resets the drawOrder variable to its base value. It sets the draw order of the new state and increments the draw order again. It then calls AddState to push the state on the stack and calls the OnStateChanged to raise the state change event.

The method ContainsState just checks to see if state passed in is already on the stack of states. OnStateChanged checks to see if there are any subscribers to the StateChanged event. If there are it calls StateChanged(this, null) to raise the event. We're not too concerned about the sender or event args here so I passed the instance of the state manager and null for the event arguments.

The last things that I'm going to implement in this tutorial is the title screen. To do that please download the following two images, or replace them with two of your own, Summoner's Tale Images. In your solution expand the Content folder, right click Content.mgcb and select Open.

With the content builder that comes up right click on the Content folder, select Add and then New Folder. Name this new folder Fonts. Repeat the process but call the folder GameScreens. Now right click the Fonts folder, select Add and then New Item. Select Sprite Font Description from the list and name it InterfaceFont. Now right click the GameScreens folder, select Add and then Existing Item. Navigate to the two .png files from the step above and add them to the project. Save the item and then build to make sure that there are no problems and then close the content builder window. If your sprite font does not show up in the solution explorer you can select the Show All Files button at the top of the solution explorer. Now if you go to the content folder you should see a greyed out file. Just right click it and select Include in Project.

Now, right click the Avatars project, select Add and then New Folder. Name this new folder GameStates. Now right click this new folder, select Add and then Class. Name this class BaseGameState. Repeat the process and and name the class TitleIntroState.

Open the BaseGameState class and replace the code with the following.

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

namespace Avatars.GameStates
{
    public class BaseGameState : GameState
    {
        #region Field Region

        protected static Random random = new Random();

        protected Game1 GameRef;

        #endregion

        #region Constructor Region

        public BaseGameState(Game game)
            : base(game)
        {
            GameRef = (Game1)game;
        }

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

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

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

        #endregion
    }
}
```

This really just a skeleton class that can be used to allow for polymorphism of game states. What that means is if I derive TitleIntroState from BaseGameState I can assign an instance of TitleIntroState to a variable BaseGameState. Similarly this class inherits from GameState so I can assign any instance of a class that inherits from BaseGameState to a GameState. This allows us to use any class that inherits from BaseGameState in the state manager.

The two important things are that this class exposes a property GameRef that returns a reference to the current game object. This will come in handy many times when we need a specific item from the game object in another component. The other thing is there is a static Random instance variable that will be shared between all of the game state classes. You might consider setting a fixed seed when creating this variable for debugging. This way you will know what the results of the next random variable will be so that your code flows the same way each time. You will definitely want to stop this behaviour when you publish your game.

Now, open the TitleIntroState class and replace it with the following code.

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

namespace Avatars.GameStates
{
    public interface ITitleIntroState : IGameState
    {
    }

    public class TitleIntroState : BaseGameState, ITitleIntroState
    {
        #region Field Region

        Texture2D background;
        Rectangle backgroundDestination;
        SpriteFont font;
        TimeSpan elapsed;
        Vector2 position;
        string message;

        #endregion

        #region Constructor Region

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

        #endregion

        #region Method Region

        public override void Initialize()
        {
            backgroundDestination = Game1.ScreenRectangle;
            elapsed = TimeSpan.Zero;
            message = "PRESS SPACE TO CONTINUE";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            background = content.Load<Texture2D>(@"GameScreens\titlescreen");
            font = content.Load<SpriteFont>(@"Fonts\InterfaceFont");

            Vector2 size = font.MeasureString(message);
            position = new Vector2((Game1.ScreenRectangle.Width - size.X) / 2,
Game1.ScreenRectangle.Bottom - 50 - font.LineSpacing);

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            PlayerIndex index = PlayerIndex.One;
            elapsed += gameTime.ElapsedGameTime;

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(background, backgroundDestination, Color.White);

            Color color = new Color(1f, 1f, 1f) *
(float)Math.Abs(Math.Sin(elapsed.TotalSeconds * 2));

            GameRef.SpriteBatch.DrawString(font, message, position, color);
            GameRef.SpriteBatch.End();

            base.Draw(gameTime);
        }

        #endregion
    }
}
```

First, there is an interface that derives from the IGameState interface that we created earlier. This is just allowing us to hook into those items for use in the state manager. The interface will be used to register the component as a service that we can retrieve in another class if necessary.

This class class inherits from BaseGameState and implements the ITitleIntroState interface which makes us implement IGameState as well. There are member variables for the background image, the destination of the background image, the interface font, a TimeSpan that measures the amount of time that has passed. This will be used to have a message blink in and out. There is also a member variable for the message and its position.
The constructor just registers the game state as service that can be retrieved using its interface. If there was something that needed to be exposed to an external component that can be added to the interface and retrieved by getting the added service.

In the Initialize method I set the destination of the background image to a property that I haven't added to the Game1 class yet that describes the screen that we will be rendering into. I also initialize the elapsed time to zero and set the message that will be displayed.

In the LoadContent method I load the background for the titlescreen and the interface font. I then measure the size of the message that will flash. I then use the value to calculate where to display the message centered vertically and by the bottom off the screen.

In the Update method I the elapsed variable the that is incremented each pass through the loop. As I mentioned earlier I will be using this to have the message that will be displayed fade in and out.

In the Draw method I render the the background and the message. To do that I expose the SpriteBatch in the Game1 class as a read only property. I will get to that shortly. To make the message fade in and out I use Math.Sin. I do that because sine is a cyclic wave that repeats indefinitely between 1 and -1. I use Math.Abs to ensure that it is always positive. I multiply Color(1f,1f,1f) by this value so that it flashes in white. I then draw the message.

Open up the Game1 class now so that we can add all this plumbing/scaffolding to the game. Replace the Game1 class with the following code.

```
using Avatars.GameStates;
using Avatars.StateManager;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Avatars
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GameStateManager gameStateManager;
        ITitleIntroState titleIntroState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        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);

            titleIntroState = new TitleIntroState(this);

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

        /// <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

            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
        }

        /// <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

            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);
        }
    }
}
```

I added a field for the state manager and the title introduction screen. I also added a static field that will describe the bounds of the game screen. I added a public get only property that will return the SpriteBatch object for the game and a property that will expose the screen area rectangle as well. In the constructor I create a rectangle that will describe the screen area. I went low for resolution, 1280 by 720, to accommodate readers that are using laptops that default to the 1366 by 766 resolution or lower. You can easily change this for your game or make it dynamic so that the player can change the resolution for the game.

The next step is to set the PreferredBackBufferWidth and PreferredBackBufferHeight values of the GraphicsDeviceManager to match the height and width of the rectangle that describes the screen.

After that you need to create an instance of GameStateManager and add it to the Components collection so that it will be updated during the call to base.Update in the Update method. Next, I create a TitleIntroScreen object and assign it to ItitleIntroScreen. Finally I call the ChangeState method of the GameStateManager passing in the title introduction state and PlayerIndex.One.

If you run and build the project you should see the title introduction screen displaying with the message on how to proceed to the next screen.

I'm going to end the tutorial here as we've covered a lot already. Stay tuned for the next tutorial in the series. In that one I will be implementing a few new features that will be helpful for the rest of the game. I wish you the best in your MonoGame Programming Adventures!.

Jamie McMahon

A Summoner's Tale – MonoGame Tutorial Series

Chapter 2

More Plumbing/Scaffolding

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 2015 versions as well.

In the last tutorial I set up the base project with the state manager and a title/intro screen. In this tutorial I will be adding more plumbing/scaffolding to the game as there are some key components that are missing. I also discovered after loading up my project that there was a problem with the fonts that were added. I've fixed this in the project. First, I renamed InterfaceFont.interfacefont to InterfaceFont.spritefont. I also added a new font called GameFont.spritefont using the same process as in the last tutorial.

A quick note on fonts while I'm on the subject. Most fonts are commercial products and cannot be used in your game without paying royalties or buying the font out right. There are a lot of fonts that are available to game developers though. The two sources that I use are http://www.1001fonts.com and http://www.dafont.com. If you filter you will find many different licenses, including 100% free. I'd suggest that if you find a font and use it in your game that you mention to creator somewhere in your credits.

In the last tutorial we got the state manager and added a state and it is rendering. Other than that it doesn't do anything. In order for it to do something it will have to react to user input. I'm going to add a basic version of my input handling component that I wrote for use in XNA 3.0, a long time ago now, called Xin. It attempts to centralize the important code for handling input to try and reduce complexity in the rest of the game. It is pretty common for a developer to do this sort of abstraction process. Find a common task and create a class that wraps that task so that it can be used again in other projects.

First, right click your project and select Add and then New Folder. Name this new folder Components. Now, right click the Components folder that was just added, select Add and then Class. Name this new class Xin. The code for that class follows.

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

namespace Avatars.Components
{
    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 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();

            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;
        }
    }
}
```

You will see that there are a lot of static members in this class. I did that so instead of creating an instance of the class in each other class that it is used in you have a single point for handling input. When you need to do anything input related you just use Xin.member_name.

First, you will notice that there is an enumeration at the namespace level called MouseButtons. I added this because on the one thing that I've always found lacking in XNA and MonoGame because it derives from it is mouse support. What I mean is the keyboard and game pad states can be referenced using the Keys and Buttons enumerations where as for mouse input you have no such option.

This class then derives from GameComponent. I did that so that we can create an instance of Xin and add it to the list of components without creating a member variable in a class and forget about it, for all intents and purposes.

Next there are two variable for KeyboardState and MouseState. These variables hold the current state of the mouse and keyboard and the previous state of the mouse and the keyboard. You need both to check to see when a button or key is first pressed or when it is released. Otherwise you can't tell if a button was down since the last frame or if it was released since the last frame. This will make more sense in a bit.

I also added static readonly properties for both input types and their frame, whether they are the current frame of the game or last frame of the game. These provide raw access to input which is important in some instances.

The constructor is trivial really. It just calls the base constructor that requires a Game parameter that represents the game object the component will be added to. You could in theory require more than one Game object in a game but I've never had to use one.

The methods that follow are the real meet of the class and do most of the work. I will be extending these as time goes on but for the purposes of this tutorial they are sufficient. The first method, Update, is inherited from the GameComponet class. The way this works is if you create a component and add it to the list of components in a game their Update method is called automatically each frame, if the component is enabled. Inside the update method I first set the variables named previous to current. I then get the current state of the keyboard and mouse. This is how we will detect when a key is pressed or release, but not if it is just down or up. I then call base.Update which would call Update on other related components inside of this class.

The first static method is FlushInput. What this method does is set the current states to their previous states. Since they are now equal and of the methods that would check for new presses or releases will fail.

In CheckKeyReleased checks to see if a key that was down last frame is now up. Let me explain why I did this was as opposed to a key that was up in the previous frame is down in the current frame. What I've found checking it the second way, the key is now down, is that before the call to Xin.FlushInput happens sometimes the new press is caught in the next frame of the game. Checking for a release works around this phenomenon. To do this I use the IsDown and IsUp member methods of the KeyboardState class, which do as you might imagine.

The last, and most interesting, method is the CheckMouseReleased. I say that it is interesting is that compared to checking for keys, checking for mouse buttons is more complex. There are no built in methods of the input classes that check if a mouse button is down. All that there is are named properties that get a ButtonState enumeration that defines if the button is up or down. So, in order to check a button you retrieve the ButtonState and then check its value.

So, in the CheckMouseReleased button I have a switch statement that checks what parameter was passed into the method. Based on what button I have a statement that returns the current state of the button compared with the previous state of the button. If no other response is returned I then return false.

That is all for the Xin class. As you can see there is room for improvement, and you might be thinking of that already, which is good. It is always good to read code, see how it works and then try to implement it or another version of it. It is never a good idea to just copy and paste code without understanding it. That is a speech that I will spare you from.

The next item that I want to add is a menu component that will be used for the main menu and other menus in the game. Right click the Components folder, select Add and then Class. Name this new class MenuComponent. Here is the code for that class.

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

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Avatars.Components
{
    public class MenuComponent
    {
        #region Fields

        SpriteFont spriteFont;
        readonly List<string> menuItems = new List<string>();
        int selectedIndex = -1;
        bool mouseOver;

        int width;
        int height;

        Color normalColor = Color.White;
        Color hiliteColor = Color.Red;

        Texture2D texture;

        Vector2 position;

        #endregion Fields

        #region Properties

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

        public int Width
        {
            get { return width; }
        }

        public int Height
        {
            get { return height; }
        }

        public int SelectedIndex
        {
            get { return selectedIndex; }
            set
            {
                selectedIndex = (int)MathHelper.Clamp(
                        value,
                        0,
                        menuItems.Count - 1);
            }
        }

        public Color NormalColor
        {
            get { return normalColor; }
            set { normalColor = value; }
        }

        public Color HiliteColor
        {
            get { return hiliteColor; }
            set { hiliteColor = value; }
        }

        public bool MouseOver
        {
            get { return mouseOver; }
        }

        #endregion Properties
        #region Constructors

        public MenuComponent(SpriteFont spriteFont, Texture2D texture)
        {
            this.mouseOver = false;
            this.spriteFont = spriteFont;
            this.texture = texture;
        }

        public MenuComponent(SpriteFont spriteFont, Texture2D texture, string[] menuItems)
            : this(spriteFont, texture)
        {
            selectedIndex = 0;

            foreach (string s in menuItems)
            {
                this.menuItems.Add(s);
            }

            MeassureMenu();
        }

        #endregion Constructors

        #region Methods

        public void SetMenuItems(string[] items)
        {
            menuItems.Clear();
            menuItems.AddRange(items);
            MeassureMenu();

            selectedIndex = 0;
        }

        private void MeassureMenu()
        {
            width = texture.Width;
            height = 0;

            foreach (string s in menuItems)
            {
                Vector2 size = spriteFont.MeasureString(s);

                if (size.X > width)
                    width = (int)size.X;

                height += texture.Height + 50;
            }

            height -= 50;
        }

        public void Update(GameTime gameTime, PlayerIndex index)
        {
            Vector2 menuPosition = position;
            Point p = Xin.MouseState.Position;

            Rectangle buttonRect;
            mouseOver = false;

            for (int i = 0; i < menuItems.Count; i++)
            {
                buttonRect = new Rectangle((int)menuPosition.X, (int)menuPosition.Y, 
texture.Width, texture.Height);
```

```
                if (buttonRect.Contains(p))
                {
                    selectedIndex = i;
                    mouseOver = true;
                }

                menuPosition.Y += texture.Height + 50;
            }

            if (!mouseOver && (Xin.CheckKeyReleased(Keys.Up)))
            {
                selectedIndex--;
                if (selectedIndex < 0)
                    selectedIndex = menuItems.Count - 1;
            }
            else if (!mouseOver && (Xin.CheckKeyReleased(Keys.Down)))
            {
                selectedIndex++;
                if (selectedIndex > menuItems.Count - 1)
                    selectedIndex = 0;
            }
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            Vector2 menuPosition = position;
            Color myColor;

            for (int i = 0; i < menuItems.Count; i++)
            {
                if (i == SelectedIndex)
                    myColor = HiliteColor;
                else
                    myColor = NormalColor;

                spriteBatch.Draw(texture, menuPosition, Color.White);

                Vector2 textSize = spriteFont.MeasureString(menuItems[i]);

                Vector2 textPosition = menuPosition + new Vector2((int)(texture.Width - 
textSize.X) / 2, (int)(texture.Height - textSize.Y) / 2);
                spriteBatch.DrawString(spriteFont,
                    menuItems[i],
                    textPosition,
                    myColor);

                menuPosition.Y += texture.Height + 50;
            }
        }

        #endregion Methods

        #region Virtual Methods
        #endregion Virtual Methods

    }
}
```

Quite a bit of code but it is nothing that is overly complex. First, there are a number of member fields in the class. Since the menu needs to draw text I added a SpriteFont field. There is then a List<string> that will contain the individual menu items to be rendered. The selectedIndex field will return what menu item is currently selected. Next mouseOver is added to allow for mouse support. The width and height fields will be used to control where the menu is rendered. There are two Color fields that control what color menu items are rendered. There is one color for regular menu items and then a second for highlighted menu items. Texture is a field that will serve as a button background image and position controls where the menu it displayed.

I added some public properties that will expose some of the fields to other classes. You should be able to set the position of the menu items so I exposed the position member variable. Frequently you will need to know the width or height of the menu so I added properties for that as well. I added a property to return and set the selected menu item. You will see that in the set part I use MathHelper.Clamp to make sure the selected item is not outside of the available menu items. There are properties that allow you to get and set the menu colors as well. Finally there is a property that will return the mouseOver member variable.

You will notice that I did not add a property that returns the menu items. The reason is that when the menu items change I want to force a call to a method that will find the height and width of the menu. If I exposed the list it can be modified outside of the class and changes would not be picked up and could lead to a change in the was the menu is rendered.

I added two constructors. The first takes two parameters, the font the menu will be rendered with and the button texture. This one just initializes some of the fields that I added. The second constructor accepts an array of strings as well as the others. It will also call the first constructor to set some of the default values. Inside I set the selectedIndex member variable to 0, the first item, and then in a foreach loop I add each string to the menu items. I then call MeasureMenu to calculate the height and width of the menu.

The first method is called to change the menu items for the menu. It takes a string array that hold the menu items to be displayed. If clears any existing menu items, adds them to the list of menu items, calls the MeasureMenu method to calculate the width and height then finally sets the selectedIndex member variable to 0.

MeasureMenu is where I calculate the width and height of the menu. First, I set width to be the width of the texture. You technically do not want to menu items to be longer than the width of your button but you need to allow for this scenario. I reset the height to 0 as well. In a foreach loop I iterate over all of the menu items. I use the MeasureString member of the SpriteFont class to get the height and width of the string. If the X value is greater than the width than the current width I update that value. I then add the height of the button and some padding to the height member variable. I then subtract the padding as it is added to the last menu item.

In the Update method I handle updating the menu component. I added a local variables that hold the position that the menu is drawn and the mouse as a Point. I then have a Rectangle that will be the bounds of each menu item to determine if the mouse is over them or not. After the variables I set the mouseOver variable to false. This ensures that it is reset at the start of each frame of the game.

Next there is a for loop that loops over the menu items one by one. Inside I calculate the Rectangle that holds the bounds of the button. I use the local variable that is set to be the position of the menu and the height and width of the button texture. I check to see if the mouse is over the current button. If it is I set the selectedIndex member to be that menu and the mouseOver variable to true. I add the padding and the texture height to the current menu item position before moving to the next item.

The if statement that follows handles the user moving the selected menu item using the up and down keys. I do not change the position if the mouse is over a button as that is the button that should have focus. To move the selection up I decrement the selectedIndex member. If it is less than zero I set the menu item to last menu item, which is the Count member of the list of menu items minus one, since it is zero based. To move down I increment the selectedIndex member. I then check if it is outside the bounds of the list and if it is set it to zero.

The last method is the Draw method that will render the menu items. There are two local variables that hold the menu position and color that the items will be drawn in. You then loop over each of the menu items. I first set the color variable based on what the selected index is. I then draw the button image. I measure the menu item string. I then calculate its position relative to the button texture and center it. I then draw the menu item string. Finally I update the Y coordinate by adding the height of the texture plus the padding.

The last new class that I'm going to add is the start state that displays a menu to the user allowing them to pick options. Right click the GameStates folder, select Add and then Class. Name this new class MainMenuState. Here is the code for that class.

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

namespace Avatars.GameStates
{
    public interface IMainMenuState : IGameState
    {
    }

    public class MainMenuState : BaseGameState, IMainMenuState
    {
        #region Field Region

        Texture2D background;
        SpriteFont spriteFont;
        MenuComponent menuComponent;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

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

        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            spriteFont = Game.Content.Load<SpriteFont>(@"Fonts\InterfaceFont");
            background = Game.Content.Load<Texture2D>(@"GameScreens\menuscreen");

            Texture2D texture = Game.Content.Load<Texture2D>(@"Misc\wooden-button");

            string[] menuItems = { "NEW GAME", "CONTINUE", "OPTIONS", "EXIT" };

            menuComponent = new MenuComponent(spriteFont, texture, menuItems);

            Vector2 position = new Vector2();

            position.Y = 90;
            position.X = 1200 - menuComponent.Width;

            menuComponent.Postion = position;

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            menuComponent.Update(gameTime);

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter) ||
(menuComponent.MouseOver && Xin.CheckMouseReleased(MouseButtons.Left)))
            {
                if (menuComponent.SelectedIndex == 0)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 1)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 2)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 3)
                {
                    Game.Exit();
                }
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(background, Vector2.Zero, Color.White);

            GameRef.SpriteBatch.End();

            base.Draw(gameTime);

            GameRef.SpriteBatch.Begin();

            menuComponent.Draw(gameTime, GameRef.SpriteBatch);

            GameRef.SpriteBatch.End();

        }

        #endregion
    }
```

```
}
```

First, there are using statements to bring the Components namespace in this project as well as a few of the Xna.Framework namespace. There is an empty interface at the start of this class called IMainMenuState. One use for interfaces is to define a contract that other objects can use to interact with this object. An empty interface means that there are no members that will be exposed directly to other objects.

The class itself inherits from BaseGameState so it can be used in the state manager and it implements the ImainMenuSate interface. There are three private member variables: background, spriteFont and menuComponent. The background variable will hold the background image that will be render, spriteFont is for drawing any text and menuComponent will render our menu.

In the constructor I register the class as a service with the framework so it can be retrieved in other classes. This one won't be but I'm being consistent will all states.

In the LoadContent menu I load the font and background. I also load an image for the menu items. Download the Misc content items from this link and extract the file. Back in the project open the content manager by left clicking Content.mgcb and selecting Open. Select the Content node and create a new folder Misc. Right click the Misc folder, select Add and the Existing Item. Browse to the wooden-button.png file and add it. Now, right click the GameScreens folder under the Content folder. If you haven't added the menuscreen.png file to this folder do so now. Once all items are added open the Build menu and select Rebuild.

After loading the content I create a string that holds the menu items that will be displayed and create the menu component. I then position the menu, based on the image that I created for the background. The buttons should sit on top of the chains on the right side of the screen.

In the Update method I first call the Update method of the menu before checking if the space, enter or left mouse button have been released since the last frame. There is then a chain of if-else statements, one for each of the menu items. Inside the if statement I call Xin.FlushInput just to make sure there are no leftovers. The interesting part is that in the last index I call Game.Exit to close the game. It would be better if you put up a pop up asking if the player that they really want to end the game. I will implement that in a future tutorial.

The Draw method is pretty simple. It first renders the background, calls base.Draw and then renders the menu.

Now, I'm going to implement the title screen changing state to the menu state when space, enter or the left mouse button are released. First, update the using statements in the class to make sure all necessary entities are in scope for the class.

```
using System;
using Avatars.Components;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
```

Now, update the Update method to the following. You will get an error message on the state change because we have not yet updated the main game class yet. That is what we will tackle next.

```
public override void Update(GameTime gameTime)
{
    PlayerIndex? index = null;

    elapsed += gameTime.ElapsedGameTime;

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter) ||
Xin.CheckMouseReleased(MouseButtons.Left))
    {
        manager.ChangeState((MainMenuState)GameRef.StartMenuState, index);
    }

    base.Update(gameTime);
}
```

There is a nullable variable, index, for game pad support. I'm not including that at the moment but I will be in the future so I need to park this here. I then call the methods off the Xin class to check for the release of the space key, enter key or left mouse button. If any of those have been released I call the ChangeState method to change the state the menu state. You will have an error until we update the Game1 class.

We're in the home stretch now. All that is remaining is updating the Game1 class to handle this new logic. The changes were pretty extensive so I will paste the code for the entire class.

```
using Avatars.Components;
using Avatars.GameStates;
using Avatars.StateManager;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Avatars
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GameStateManager gameStateManager;

        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        public GameStateManager GameStateManager
        {
            get { return gameStateManager; }
        }

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        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);

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

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

            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);
        }
    }
}
```

So, I first removed all of the extra comments that are part of the template to shorten the class as they are not necessary for the project. I also added in a using statement to bring the Components namespace into scope. I then added a member variable of type IMainMenuState for the start menu for the game. I also added in some properties to expose the member variables to game objects that we will be creating.

In the constructor for the Game1 class I set the IsMouseVisible property to try so that the mouse cursor will be displayed. I also created the new MainMenuState object. Also, in the Initialize method I add an instance of Xin to the game components for the game.

Before building and running the game, first open the Content manager and build the content for the game. Once the content builds successfully then build and run the game. At this point the title screen will appear. Pressing either the space or enter key or clicking the left mouse button will move to the next scene. Once on the menu scene clicking the Exit button will close the game.

I'm going to end the tutorial here as we've covered a lot. The first two tutorials were a little dry in regard to game elements but at this point most of the plumbing/scaffolding is in place. Now we can move on to game elements. Stay tuned for the next tutorial in the series. In that one I will be implementing and new game state and some game play elements.

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 3

Tile Engine and Game Play State

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.

In this tutorial I will be adding the game state for the player exploring the map and the tile engine to render the map. Why do you need a tile engine though? You could just draw the map in an image editor, load that and draw it. The reason is memory and efficiency. Most of the map will be made up of the same background images. Creating a tile based on that and rendering just that saves a lot of memory. It is also more efficient to load a few hundred small images than one extremely large image. The same is true for rendering.

First, right click the Avatars project, select Add and then New Folder. Name this new folder TileEngine. Now right click the TileEngine folder, select Add and then Class. Name this new class TileSet. Here is the code for that class.

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

namespace Avatars.TileEngine
{
    public class TileSet
    {
        public int TilesWide = 8;
        public int TilesHigh = 8;
        public int TileWidth = 64;
        public int TileHeight = 64;

        #region Fields and Properties

        Texture2D image;
        string imageName;
        Rectangle[] sourceRectangles;

        #endregion

        #region Property Region

        [ContentSerializerIgnore]
        public Texture2D Texture
        {
            get { return image; }
            set { image = value; }
        }

        [ContentSerializer]
        public string TextureName
        {
            get { return imageName; }
            set { imageName = value; }
        }

        [ContentSerializerIgnore]
        public Rectangle[] SourceRectangles
        {
            get { return (Rectangle[])sourceRectangles.Clone(); }
        }

        #endregion

        #region Constructor Region

        public TileSet()
        {
            sourceRectangles = new Rectangle[TilesWide * TilesHigh];

            int tile = 0;

            for (int y = 0; y < TilesHigh; y++)
                for (int x = 0; x < TilesWide; x++)
                {
                    sourceRectangles[tile] = new Rectangle(
                        x * TileWidth,
                        y * TileHeight,
                        TileWidth,
                        TileHeight);
                    tile++;
                }
        }

        public TileSet(int tilesWide, int tilesHigh, int tileWidth, int tileHeight)
        {
            TilesWide = tilesWide;
            TilesHigh = tilesHigh;
            TileWidth = tileWidth;
            TileHeight = tileHeight;

            sourceRectangles = new Rectangle[TilesWide * TilesHigh];

            int tile = 0;

            for (int y = 0; y < TilesHigh; y++)
                for (int x = 0; x < TilesWide; x++)
                {
                    sourceRectangles[tile] = new Rectangle(
                        x * TileWidth,
                        y * TileHeight,
                        TileWidth,
                        TileHeight);
                    tile++;
                }
        }

        #endregion

        #region Method Region
        #endregion
    }
}
```

First question then is then “What is a tile set?”. A tile set is an image that is made up of the individual tiles that will be placed on the map. When rending the map you will pick out the individual tile using the image and a source rectangle. This source rectangle defines the X and Y coordinates of the tile and the height and width of the tile. You will see this in practice shortly.

I first have 4 public member variables in this class that hold the basics of the tile set. They are TilesWide for the number of tiles across the image, TilesHigh for the number of tiles down the image, TileWidth for the width of each tile and TileHeight for the height of each tile. There are private member variables for the texture for the image, the name of the image and the source rectangles that describe each tile.

You will see that I marked some of the properties with attributes. These attributes control if the member variable will be serialized or not. For serialization to work correctly you may need the actual XNA Framework installed, depending on the version of MonoGame that you are using. We will cross that bridge in the future when we get to that point. These attributes control if the property will be serialized or not when using IntermediateSerializer. Again, I'll explain this better when we actually use it.

There is a property that exposes the name of the texture and the actual texture. There is also a readonly property that exposes the source rectangles for the tile set.

I've included two constructors in this class. The parameterless one is meant for deserializing the tile set when importing it into the game. It uses the default values that I assigned to the member variables to create the source rectangles. It will also read these values from a serialized tile set and use those instead. The rectangles are calculate inside of a nested for loop. The X coordinate is found using the width of each tile and the index for the inner loop. The Y coordinate is found using the height of each tile and the index of the outer loop. It is important to remember that Y is the outer loop and X is in the inner loop or the source rectangles will come out rotated.

The next class that I'm going to add is for a basic 2D camera that will assist in rendering only the viewable portion of that map. By drawing only the visible portion of the map we are increasing the efficiency or the rendering process. For example, say we have a map that is 200 tiles by 200 tiles. That is 40000 tiles that would need to be drawn each frame of the game. If the screen can only display 40 by 30 plus one in each direction you can get by by drawing just 1200 tiles, actually a little more because we need to pad for the right and bottom edges. That is much quicker to render than the entire map as you can see.

Now, right click the TileEngine folder, select Add and then class. Name this new class Camera. Here is the code for the Camera class.

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

namespace Avatars.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 LockCamera(TileMap map, Rectangle viewport)
        {
            position.X = MathHelper.Clamp(position.X,
                0,
                map.WidthInPixels - viewport.Width);
            position.Y = MathHelper.Clamp(position.Y,
                0,
                map.HeightInPixels - viewport.Height);
        }
    }
}
```

There are two member variables in this class. They are position and speed. Position, as you've probably already gathered, is the position of the camera on the map. Speed is the speed at which the camera moves, in pixels. I also added properties that expose both of these member variables. In the Speed property I clamp the value between a minimum and maximum values, 1px and 16px. 4px would probably be better than 1px because that would be extremely slow.

There is a third property, Transformation, that returns a translation matrix. This matrix will be used to adjust where the map is drawn on the screen. You will notice that it uses -Position. This is because the camera's position is subtract from the map coordinates being drawn.

There are two public constructors. The first is parameterless and just sets the speed to a fixed value. The second takes as a parameter the position of the camera. It then sets the position of the camera using the value and sets the default speed of the camera.

There is also one method called LockCamera. What it does is clamp the X and Y coordinates between 0 and the width of the map minus the width of the viewport for width and 0 and the height of the map minus the height of the viewport for height. I subtract the height and width of the viewport so that we do not go past the right or bottom edge. Otherwise the camera would move to the width or height of the map and the default background color would show.

The next thing that I will add is a class that represents a layer in the map. Using layers allows you to separate duties. I typically include 4 layers on a map. The first layer is the base terrain for the map. The second layer is used for transition tiles. What I mean by this is that if you have a grass and a mud tile side by side there is an image that blends the two together. These can be made up of both tiles or one of the tiles with transparent areas. I then have a layer for solid objects, such as buildings, trees, and other objects. The last layer is for decorations. These tiles are used to add interest to the map. You of course are not limited to just these four layers but they are what I've included in this tutorial.

Now, right click the Tile Engine folder, select Add and then Class. Name this new class TileLayer. Here is the code for that class.

```
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;

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

        [ContentSerializer(CollectionItemName = "Tiles")]
        int[] tiles;

        int width;
        int height;

        Point cameraPoint;
        Point viewPoint;
        Point min;
        Point max;

        Rectangle destination;
```

```
#endregion

#region Property Region

[ContentSerializerIgnore]
public bool Enabled { get; set; }

[ContentSerializerIgnore]
public bool Visible { get; set; }

[ContentSerializer]
public int Width
{
    get { return width; }
    private set { width = value; }
}

[ContentSerializer]
public int Height
{
    get { return height; }
    private set { height = value; }
}

#endregion

#region Constructor Region

private TileLayer()
{
    Enabled = true;
    Visible = true;
}

public TileLayer(int[] tiles, int width, int height)
    : this()
{
    this.tiles = (int[])tiles.Clone();
    this.width = width;
    this.height = height;
}

public TileLayer(int width, int height)
    : this()
{
    tiles = new int[height * width];
    this.width = width;
    this.height = height;

    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            tiles[y * width + x] = 0;
        }
    }
}

public TileLayer(int width, int height, int fill)
    : this()
{
    tiles = new int[height * width];
    this.width = width;
    this.height = height;
```

```
            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    tiles[y * width + x] = fill;
                }
            }
        }

        #endregion

        #region Method Region

        public int GetTile(int x, int y)
        {
            if (x < 0 || y < 0)
                return -1;

            if (x >= width || y >= height)
                return -1;

            return tiles[y * width + x];
        }

        public void SetTile(int x, int y, int tileIndex)
        {
            if (x < 0 || y < 0)
                return;

            if (x >= width || y >= height)
                return;

            tiles[y * width + x] = tileIndex;
        }

        public void Update(GameTime gameTime)
        {
            if (!Enabled)
                return;
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, TileSet tileSet, Camera 
camera)
        {
            if (!Visible)
                return;

            cameraPoint = Engine.VectorToCell(camera.Position);
            viewPoint = Engine.VectorToCell(
                new Vector2(
                    (camera.Position.X + Engine.ViewportRectangle.Width),
                    (camera.Position.Y + Engine.ViewportRectangle.Height)));

            min.X = Math.Max(0, cameraPoint.X - 1);
            min.Y = Math.Max(0, cameraPoint.Y - 1);
            max.X = Math.Min(viewPoint.X + 1, Width);
            max.Y = Math.Min(viewPoint.Y + 1, Height);

            destination = new Rectangle(0, 0, Engine.TileWidth, Engine.TileHeight);
            int tile;

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

            for (int y = min.Y; y < max.Y; y++)
            {
                destination.Y = y * Engine.TileHeight;

                for (int x = min.X; x < max.X; x++)
                {
                    tile = GetTile(x, y);

                    if (tile == -1)
                        continue;

                    destination.X = x * Engine.TileWidth;

                    spriteBatch.Draw(
                        tileSet.Texture,
                        destination,
                        tileSet.SourceRectangles[tile],
                        Color.White);

                }
            }

            spriteBatch.End();
        }

        #endregion
    }
}
```

A few member variables here. First, is an array of integers that holds the tiles for the layer. Instead of creating a two dimensional array I created a one dimensional array. I will use a formula to find the tile at the requested X and Y coordinates. I will also use the convention that if a tile has a value of -1 then that tile will not be drawn. Next there are width and height member variables that describe the height and width of the map.

Next up for member variables are four Point variables: cameraPoint, viewPoint, min and max. I created these at the class level because I will be using them each time that the map is drawn. Doing it this way just means that we are not constantly creating and destroying variables each frame. It is not a great optimization but even minor optimizations will help in a game. I will discuss their purpose further when I get to rendering.

The last member variable is a Rectangle variable and did it this way for the same reason as the others. The thing to note though is that this will be used every time a tile is drawn as it represents the destination the tile will be drawn in screen space. This is more effective than the other optimization. The reason is we will be drawing 4 layers with approximately 800 tiles a layer 60 times per second. Compared to 4 layers 60 times per second.

There are two auto implemented properties Enabled and Visible that control if the layer is enabled and visible. I added Enabled because I'm considering adding in animated tiles and to support that I would require an Update method. Visible will determine if the layer should be drawn or not. I also added readonly properties to expose the width and height of the layer.

There are four constructors in this class. The first has no parameters and just sets the auto properties. It is also private and will not be called outside of the class. Its main purpose is for use with the

IntermediateSerializer class what is used for serializing and deserializing content. The other three constructors are used for creating layers. The one takes an array as well as height and width parameters, the second just takes height and width parameters and the third takes height, width and fill parameters. Each of them also calls this() to initialize the Visible and Enabled properties.

The constructor that takes an array as a parameter creates a clone of the array, which is a shallow copy of the array. It then sets the width and height parameters. The constructor that takes just the width and height of the layer first creates a new array that is height * width. Next there are nested loops where y is the outer loop and x is the inner loop. To calculate where a tile is places I use the formula y * width + x. So the first row will be from 0 to 1 * width - 1, the second from 1 * width to 2 * width – 1 and then 2 * width to 3 * width -1 and so on. This formula will has to be applied consistently or you will have very strange behaviour. The third works the same as the first but rather than setting the tile to a default value it sets the tile to the value passed in.

Next are two method GetTile and SetTile. Their name definitely describes their purpose as they get and set tiles respectively. GetTile returns a value where as SetTile receives and additional value. There are if statements in each method to check that the x and y values that are passed in are within the bounds of the map. Both also use the same formula as in the constructor to get and set the tile.

I included an Update method that accepts a GameTime parameter. This parameter holds the amount of time passed between frames and has some useful purposes. Currently it checks the Enabled property is false and if it is exits the method.

The Draw method is where the meat of this class is as it actually draws the tiles. You will get some errors here because it relies on some static methods for a class I have not implemented yet, Engine. Engine just holds some common properties for the tile engine.

First, I check to see if the layer is visible or not. If it is not visible I exit the function. I then set the cameraPoint and viewPoint vectors using Engine.VectorToCell. What this call does is takes a point in pixels and returns the tile that the pixel is in. These vectors will be used to determine where to start and stop drawing tiles. As I mentioned we will only be drawing the visible tiles. Actually it will be plus and minus one tile to account for the scenario where the pixel is not the X and Y coordinates of a tile. The viewPoint vector is found by taking the camera position and adding the size of the view port the map is being drawn to. The view port is typically the entire screen in these types of games but occasionally you will find a side bar or bottom bar that holds information that reduces the map space.

The min vector is clamped between 0 and the camera position minus 1 tile. Similarly max is clamped between the view port X and Y plus 1 tile. After that I create a new instance for the destination Rectangle member variable.

The call to SpriteBatch.Begin is interesting. I use SpriteSortMode.Deferred because I am drawing a lot of images within the batch. BlendState.AlphaBlend allows for tiles that have transparency with in them. This include fully transparent and partially transparent, if the exported image supports it. SamplerState.PointClamp is used to prevent strange behaviour when drawing the map. This includes line around the tiles when scrolling the map. This SamplerState is always recommended when using source rectangles when drawing.

Next are the next for loops that actually render the layer. The tiles are drawn from the top left corner across the screen then down to the bottom right corner. You can experiment with different loops to see how it affects the way the layer is rendered.

Outside of the inner loop I set the Y coordinate of the destination rectangle as it remains the same for the entire row. Inside the loop I call GetTile passing in the X and Y coordinates of the tile. If the tile is -1 I just move to the next iteration of the inner loop. I then set the X coordinate of the tile. Finally I draw the tile using the overload that requires a texture, source rectangle, destination rectangle and tint color.

That, at a very high level, is how a tile layer is rendered. There are likely a few more optimizations that could be made such as accessing the array directly rather than relying on GetTile. I just used that method to make sure that I calculate the tile in the one dimensional array correctly.

Now I'm going to add a class that represents the entire map for the game. Right click the TileEngine folder, select Add and then Class. Name this new class TileMap. Add the following code to the TileMap class.

```
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;

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

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

        [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;
        }

        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);
        }

        #endregion
    }
}
```

The class is pretty simple. It is has a member variable for the name of the map, member variables for the layers for the map, the height and width of the map along with a Dictionary<string, Point> that holds the names of NPCs on the map and their coordinates, in tiles. I will be getting to NPCs in a future tutorial. There is also a member variable for the tile set that is used for drawing the map.

There is a property for each of the member variables that exposes them to other classes. Most of these are public getters with private setters. They are also marked so that they will be serialized using the IntermediateSerializer.

There are two public get only properties that expose the width of the map in pixels and the height of the map in pixels. They are used in different places to determine if an object is inside the map or outside the map.

There are three constructors for this class. The first is a private constructor with no parameters. The no parameter constructor is required for deserializing objects. The second constructor takes as parameters the tile set for the map and the name of the map. It just initializes the character dictionary and member variables with the values passed in.

The third takes those parameters as well as four layers. It calls the two parameter constructor so that it will initialize those member variables rather than replicate the code in this constructor. It then sets the width and height members using the height and width of the ground layer.

There is a GetTile and SetTile method for each of the layers. They just provide the functionality to update the layers without exposing the layers themselves. This is done to promote good object-oriented programming. The user of the class does not need to know about the objects with in the class to work with them. They instead use the public interface that is exposed. I don't use the term as the keyword interface. I use it to mean a contract that is published to user so they can interact with the class.

There are also Fill methods for the layers that fill the building, edge and decoration layers with -1. This is useful when creating maps in the editor. I will be providing the editor as part of the series but will not be writing tutorials on how to create it.

Finally, the Update and Draw methods check to make sure the layers are not null and then call the Update and Draw method of that layer. We should probably be a little more diligent when calling the other methods that will through a null value exception when the layer does not have a value. I will leave that as an exercise for you to implement in your game.

Now I'm going to implement the the Engine class. So, right click the TileEngine folder, select Add and then Class. Name this new class Engine. Here is the code for that class.

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

namespace Avatars.TileEngine
{
    public class Engine
    {
        private static Rectangle viewPortRectangle;

        private static int tileWidth = 32;
        private static int tileHeight = 32;
        private TileMap map;

        private static float scrollSpeed = 500f;

        private static Camera camera;

        public static int TileWidth
        {
            get { return tileWidth; }
            set { tileWidth = value; }
        }

        public static int TileHeight
        {
            get { return tileHeight; }
            set { tileHeight = value; }
        }

        public TileMap Map
        {
            get { return map; }
        }

        public static Rectangle ViewportRectangle
        {
            get { return viewPortRectangle; }
            set { viewPortRectangle = value; }
        }


        public static Camera Camera
        {
            get { return camera; }
        }

        #region Constructors

        public Engine(Rectangle viewPort)
        {
            ViewportRectangle = viewPort;
            camera = new Camera();

            TileWidth = 64;
            TileHeight = 64;
        }

        public Engine(Rectangle viewPort, int tileWidth, int tileHeight)
            : this(viewPort)
        {
            TileWidth = tileWidth;
            TileHeight = tileHeight;
        }

        #endregion

        #region Methods

        public static Point VectorToCell(Vector2 position)
        {
            return new Point((int)position.X / tileWidth, (int)position.Y / tileHeight);
        }

        public void SetMap(TileMap newMap)
        {
            if (newMap == null)
            {
                throw new ArgumentNullException("newMap");
            }

            map = newMap;
        }

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

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            Map.Draw(gameTime, spriteBatch, camera);
        }

        #endregion
    }
}
```

For this I'm going on the premise that there is only one map and one instance of Engine at a time. For that reason I've included some public static members to expose values to other classes outside of the tile engine.

The first three member variables hold a Rectangle that describes the view port, the width of tiles on the screen and the height of tiles on the screen. The next member holds the map that is currently in use. The last two member variables hold scrollSpeed that determines how fast the map scrolls. You'll think that 500 pixels is really fast. This is just a multiplier though and not an actual speed. It is not used in this tutorial but will be in a future tutorial so I've left it in.

Static properties expose the tile width and height, the view port rectangle and the camera. There is also a property that exposes the map as well.

There are two constructors for this class. The first accepts a Rectangle that represents the visible area of the screen the map will be drawn to. It sets that value to the appropriate member variable and then sets the tile width and height on the screen to be 64 pixels.

The second constructor takes the same Rectangle for the screen space but also takes the width and height of the tiles on the screen. It calls the first constructor to set the view port and then sets the tile width and tile height member variables. That is done simply by dividing the X value by the tile width and the Y value by the tile height.

Next you will find the public static method VectorToCell that you saw earlier that takes a vector which represents a point on the map measured in pixels that returns what tile the pixel is in.

There is also a method called SetMap that accepts a TileMap parameter. It checks to see if it is null and if it is throws an exception as the map cannot be null. If it isn't null it sets the map member variable to be the map passed in.

The last two methods are the Update and Draw methods. The Update method takes a GameTime parameter and just calls the Update method of the map. The Draw method takes GameTime as well as a SpriteBatch object that will be used to render the map.

That wraps up the tile engine. It is rather plain but it will get our job done nicely. The last thing that I'm going to add today is a basic game play state. It won't do much as the tutorial is already pretty long but I will pick up with it in the next tutorial.

Now, right click the GameStates folder, select Add and then Class. Name this new class GamePlayState. Here is the code for that state.

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

namespace Avatars.GameStates
{
    public interface IGamePlayState
    {

    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);

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

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

        protected override void LoadContent()
        {
        }

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

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

It is a pretty stripped down version of a game state. I included the interface that this class will implement. Currently there is nothing in the interface but that will most definitely change in this for this class. I next included an Engine member variable that I initialize using the static ScreenRectangle property that is exposed by the Game1 class.

The constructor just registers this instance of the GamePlayState as a service using the interface that was included at the start of the class. Currently it does not do anything but will be required in the near future so I made sure to keep it in.

The constructor then registers the instance as a service with the game that can be retrieved as needed by other states. I also added method stubs for the main DrawableGameComponent methods, Initialize, Load, Update and Draw.

I'm going to end the tutorial here because we've covered a lot in this one. In the next tutorial I will wire the game play state to open from the main menu state and create an render a map. Please stay tuned for the next tutorial in this series. If you don't want to have to keep checking for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news for Game Programming Adventures

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 4

Exploring the Map

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.

In this tutorial I'm going to be working on the game play state that we added in the last tutorial. First, I'm going to cover wiring the game so that we can change from the menu state to the game state. I will then have the game play state render a map that we will code on the fly. Once the map is rendering I will add in scrolling the map.

To get started open up the Game1 class. Replace the fields, properties and constructor with the following.

```
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GameStateManager gameStateManager;

        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;
        IGamePlayState gamePlayState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        public IGamePlayState GamePlayState
        {
            get { return gamePlayState; }
        }

        public 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);
        }
```

The first change is that I've added a field for the game play state. I then added a property that exposes the IGamePlayState interface. Finally, in the constructor I create the state.

I'm going to make a few tweeks to the game play state. Open the GamePlayState file and update it to the following code.

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

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

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);
        TileMap map;
        Camera camera;
        public GamePlayState(Game game)
            : base(game)
        {
            game.Services.AddService(typeof(IGamePlayState), this);
        }

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

        protected override void LoadContent()
        {
        }

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

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

        public void SetUpNewGame()
        {
        }

        public void LoadExistingGame()
        {
        }

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

First, I added some using statements to bring a few of MonoGame/XNA classes into scope in this class. I then updated the interface to include three method signatures, SetUpNewGame, LoadExistingGame and StartGame. The first two are called to create a new game or load an existing game. The third will be called once the other two have finished to start a game. I added in two new fields, map and camera, for a TileMap and Camera respectively. I also implemented the interface method, SetUpNewGame, LoadExistingGame and StartGame.

Next, I'm going to add the transition from the menu to the game play. Open the MainMenuState file and replace the Update method with the following version.

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter) ||
(menuComponent.MouseOver && Xin.CheckMouseReleased(MouseButtons.Left)))
            {
                if (menuComponent.SelectedIndex == 0)
                {
                    Xin.FlushInput();

                    GameRef.GamePlayState.SetUpNewGame();
                    GameRef.GamePlayState.StartGame();
                    manager.PushState((GamePlayState)GameRef.GamePlayState,
PlayerIndexInControl);
                }
                else if (menuComponent.SelectedIndex == 1)
                {
                    Xin.FlushInput();

                    GameRef.GamePlayState.LoadExistingGame();
                    GameRef.GamePlayState.StartGame();
                    manager.PushState((GamePlayState)GameRef.GamePlayState,
PlayerIndexInControl);
                }
                else if (menuComponent.SelectedIndex == 2)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 3)
                {
                    Game.Exit();
                }
            }

            base.Update(gameTime);
        }
```

What changed is I updated the if statements where the selected index is 0 or if the selected index is 1. Since 0 is the new game option I call the SetUpNewGame and StartGame methods on the game play state. I then push the game play state onto the game state manager. Similarly, for the other state I call LoadExistingGame instead of SetUpNewGame. Otherwise, the code is the same for both.

What we are going to need next is a tile set. I've uploaded one to my site here. I ended up merging two sets into one. The building tiles were grabbed from a public domain tile set that I found on OpenGameArt.org here, http://opengameart.org/content/town-tiles. I recommend that you visit this site where your developing games. You can find some awesome art work with various licenses. You can even some of the assets as placeholders until you find exactly what it is you are looking for.

Once you've downloaded to the tile set open the content manager. First add a new folder to the content folder called Tiles. Now select the Tiles folder and select Add Existing Item. Navigate to the tileset1.png file and add it. Before closing the content manager make sure that you rebuild the content.

Now, go back to the GamePlayState.cs file in the solution. Update the Draw and SetUpNewGame methods as follows.

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

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

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

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

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

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

            camera = new Camera();
        }
```

In the Draw method I check if the map and camera parameters are both not null because they are required to draw the map. If they are not null I call the Draw method of the TileMap class passing in the GameTime parameter to this Draw method, the map member variable and camera member variable.

In the SetUpNewGame method I load the image for the tile set into the tiles variable. I then create a new tile set using the parameters 8, 8, 32 and 32 because the tile set is 8 tiles width, 8 tiles high with a tile width and height of 32 pixels. A quick note here. When you are creating assets like this it is best if your images have the same height and width with the same dimensions and they are a power of 2. This allows for loading on the GPU which speeds up rendering. After creating the tile set I assign the Texture property the tile set that I just loaded.

Next I create the four layers each map has with the same height and width. I then create a map object using the tile set, the four layers and call it test-map. After that I call the Fill methods to set all of the tiles for the layers above the background tiles to -1 so nothing will be drawn for those layers. Since we need a camera to draw a map I create a new instance of the camera as well.

If you build and run the game you will be shown the title screen. Dismissing the title screen will bring you to the menu. Now you can select the New Game option to be taken to the game play screen with a field of grass tiles being drawn. Since the destination tiles are bigger than the source tiles there is a bit of distortion in the rendered tiles. You can fix that by changing the engine to have width and height of 32 instead of 64.

Next I'm going to tackle scrolling the map. Still in the GamePlayState add the following using statement to bring the input manager into scope and and replace the Update method with the following.

```
using Avatars.Components;

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

            if (Xin.KeyboardState.IsKeyDown(Keys.W) && Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.A))
```

```
            {
                motion.X = -1;
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W))
            {
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S))
            {
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
            }

            if (motion != Vector2.Zero)
            {
                motion *= camera.Speed;
                camera.Position += motion;
                camera.LockCamera(map, Game1.ScreenRectangle);
            }

            base.Update(gameTime);
        }
```

I have a local variable, motion, that will hold what direction the player wants to try and move the map. I've implemented moving the map using the W, A, S and D keys. It allows for scrolling left, right, up, down and diagonals. That is why there are a series of if and else if statements, one for each of the directions. Before I cover the if statements a brief introduction to screen space. Screen space starts with the coordinates (0, 0) in the upper right corner increasing going down and right. To move up you subtract from the Y coordinate and add to the Y coordinate to move down. Similarly, to move left you subtract from the X coordinate and add to the X coordinate to move right.

The first direction that I check is up and left, the W and A keys. If that is true I set the X and Y value of motion to -1 and -1. Next up is W and D keys which is up and right so the X value 1 and the Y value is -1. Now down to the right is A and S with X set to -1 and Y set to 1. The last diagonal is S with D and the X and Y values of 1 and 1. For the cardinal directions: up, down, left and right. For those directions I check the W, S, A and D keys respectively. For W and S the X values are 0 and the Y values are -1 and 1 respectively. Similarly, A and D the Y values are 0 and the X values are -1 and 1 respectively.

Next there is an if statement that checks to see if the motion vector is not the zero vector. If it is not I multiply the motion vector by the camera speed and then add it to the position of the camera. Finally I call LockCamera method of the camera passing in the map object and the ScreenRectangle static property from the Game1 class.

If you build and run the game now once you reach the game play screen you will be able to move the map in all directions and the map will not scroll off the screen. You are going to notice a strange behavior on the diagonals. The map scrolls faster than in the cardinal directions. Why is that?

It has to do with vectors and their magnitudes. A two dimensional vector's magnitude is calculated by taking the square root of the sum of the squares, SQRT(X^2 + Y ^2). For a cardinal vector it will always evaluate to 1. For one of the diagonal vectors it evaluations as SQRT(2) which is greater than 1 so the map scrolls faster. How do you resolve this as the map should move at the same speed in all directions. The answer is you normalize the vector. What that means is the vector will still have the same direction but the magnitude of the vector will be 1. Fortunately the Vector2 class gives us a Normalize method and takes care of the for us. Update the if statement where I check for motion as follows.

```
            if (motion != Vector2.Zero)
            {
                motion.Normalize();
                motion *= camera.Speed;
                camera.Position += motion;
                camera.LockCamera(map, Game1.ScreenRectangle);
            }
```

So, the reason I check that the motion vector is not the zero vector is that the zero vector has no magnitude and you will be dividing by zero in the calculation and causing an exception to be thrown.

I'm going to end the tutorial here though. That is because we made good progress but the topics I want to cover now are fairly long and time consuming. I'd like to make sure that in each tutorial there is clearly defined progress now and there is something demonstrable at the end of each one.

In the next tutorial I'm going to add a component that will represent the player in the game with an animated sprite for the player. Please stay tuned for the next tutorial in this series. If you don't want to have to keep visiting the site to check for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news from Game Programming Adventures.

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 5

Player Component

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called A Summoner's Tale. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my web site: A Summoner's Tale. The source code for each tutorial will be available as well. I will be using Visual Studio 2013 Premium for the series. The code should compile on the 2013 Express version and Visual Studio 2015 versions as well.

I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

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

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

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

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

        #endregion

        #region Property Region

        public int FramesPerSecond
        {
            get { return framesPerSecond; }
            set
            {
                if (value < 1)
                    framesPerSecond = 1;
                else if (value > 60)
                    framesPerSecond = 60;
                else
                    framesPerSecond = value;
                frameLength = TimeSpan.FromSeconds(1 / (double)framesPerSecond);
            }
        }

        public Rectangle CurrentFrameRect
        {
            get { return frames[currentFrame]; }
        }

        public int CurrentFrame
        {
            get { return currentFrame; }
            set
            {
                currentFrame = (int)MathHelper.Clamp(value, 0, frames.Length - 1);
            }
        }

        public int FrameWidth
        {
            get { return frameWidth; }
        }

        public int FrameHeight
        {
            get { return frameHeight; }
        }

        #endregion

        #region Constructor Region

        public Animation(int frameCount, int frameWidth, int frameHeight, int xOffset, int
yOffset)
        {
            frames = new Rectangle[frameCount];
            this.frameWidth = frameWidth;
            this.frameHeight = frameHeight;

            for (int i = 0; i < frameCount; i++)
            {
                frames[i] = new Rectangle(
                        xOffset + (frameWidth * i),
                        yOffset,
                        frameWidth,
                        frameHeight);
            }
            FramesPerSecond = 5;
            Reset();
        }

        private Animation(Animation animation)
        {
            this.frames = animation.frames;
            FramesPerSecond = 5;
        }

        #endregion
        #region Method Region

        public void Update(GameTime gameTime)
        {
            frameTimer += gameTime.ElapsedGameTime;

            if (frameTimer >= frameLength)
            {
                frameTimer = TimeSpan.Zero;
                currentFrame = (currentFrame + 1) % frames.Length;
            }
        }

        public void Reset()
        {
            currentFrame = 0;
            frameTimer = TimeSpan.Zero;
        }

        #endregion

        #region Interface Method Region

        public object Clone()
        {
            Animation animationClone = new Animation(this);

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

            return animationClone;
        }

        #endregion
    }
}
```

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

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

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

I then have a property that returns what the current rectangle for the animation is. There is also a property for the current frame of the animation. The getter returns the frame while the setter clamps the value between 0 and the number of frames minus 1 because arrays are zero based. Next there are two get only properties for returning the width and height of the frames.

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

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

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

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

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

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

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

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

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

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

    public class AnimatedSprite
    {
        #region Field Region

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

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

        #endregion

        #region Property Region

        public bool IsActive { get; set; }

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

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

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

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

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

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

        #endregion

        #region Constructor Region

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

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

        #endregion

        #region Method Region

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

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

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

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

        #endregion
    }
}
```

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

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

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

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

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

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

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

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

        GameStateManager gameStateManager;
        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;
        IGamePlayState gamePlayState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

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

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        public IGamePlayState GamePlayState
        {
            get { return gamePlayState; }
        }

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

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

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

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

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

            this.IsMouseVisible = true;

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

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

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

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

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

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

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


            base.Initialize();
        }

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

        }

        protected override void UnloadContent()
        {
        }

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

            base.Update(gameTime);
        }

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

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

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

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

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

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

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

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

        private Vector2 position;

        #endregion

        #region Property Region

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

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

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

        #endregion

        #region Constructor Region

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

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

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

        #endregion

        #region Method Region

        public void SavePlayer()
        {
        }

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

            return player;
        }

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

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

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

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

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

        #endregion
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

            if (motion != Vector2.Zero)
            {
                motion.Normalize();
                motion *= (player.Speed * (float)gameTime.ElapsedGameTime.TotalSeconds);
                Vector2 newPosition = player.Sprite.Position + motion;

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

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

            base.Update(gameTime);
        }

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

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

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

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

            GameRef.SpriteBatch.End();
        }

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

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

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

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

            camera = new Camera();
        }

        public void LoadExistingGame()
        {
        }

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

The first change I made is that I added a using statement to bring the player component into scope in this class. I then created a Player field to hold the player object. In the LoadContent method I loaded the sprite sheet for the player and create a new male player named Wesley.

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

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

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

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

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

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

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

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

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 6

Avatars

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.

The key component for this game will be the avatars. Without the avatars the player is just wandering the map interacting with characters. There is no excitement for the player. Just as a refresher, I have substituted what I call avatars for Pokemon. In essence they are basically the same though. You find avatars, battle them against other avatars to increase your their power and level. They are elementally aligned just like in Pokemon as well with different attributes and moves. The difference is that you learn spells instead of capturing them. I will include a side tutorial on how you can change the game to capture avatars instead of learning to summon them from other characters or scrolls.

Since they are so integral to the game I am going to implement them now. So, right click the Avatars project, select Add and then New Folder. Name this new folder AvatarComponents. Avatars have moves that they use when battling other avatars. For that reason I'm going to add in an interface for moves. All moves will implement this interface. That allows us to have a collection of moves in the avatar component. Right click the AvatarComponents folder, select Add and then Interface. Name this interface IMove. Here is the code for this interface.

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

namespace Avatars.AvatarComponents
{
    public enum Target
    {
        Self, Enemy
    }

    public enum MoveType
    {
        Attack, Heal, Buff, Debuff, Status
    }

    public enum Status
    {
        Normal, Sleep, Poison, Paralysis
    }

    public enum MoveElement
    {
        None, Dark, Earth, Fire, Light, Water, Wind
    }

    public interface IMove
    {
        string Name { get; }
        Target Target { get; }
        MoveType MoveType { get; }
        MoveElement MoveElement { get; }
        Status Status { get; }
        int UnlockedAt { get; set; }
        bool Unlocked { get; }
        int Duration { get; set; }
        int Attack { get; }
        int Defense { get; }
        int Speed { get; }
        int Health { get; }
        void Unlock();
        object Clone();
    }
}
```

First, a move targets something. It can either target the enemy/opponent or it can target that avatar so there is an enumeration that defines this. Next is an enumeration that defines what type of move this is. For example, is it an attack, a buff for your avatar or a debuff on the enemy avatar. This can be extended if you want to include a different type of move like Abra's escape move from Pokemon. A move can affect the enemy avatar's status so I included an enumeration for that. As well I included an enumeration for what element the move is for. A basic move like a tackle has no element where as a fire based or water based attack does.

I use enumerations for these sorts of things because I don't want a list of strings or numbers to represent things like this. Also, since I'm using an enumeration it is easy to move them to classes and have a static property that returns the value.

Next is the interface that defines the different properties/methods that a move must implement. These are all common elements for the move. Moves must have a name so I included a property for that. They also require a target using the Target enumeration, a move type using the MoveType enumeration, an element using MoveElement enumeration and a status it can apply using the Status enumeration.

Moves will unlock at different levels so I included a property that exposes that as well as if the move has been learned/unlocked. A move may have a duration so I included that as a property. A move can affect the base attributes of an avatar which are Attack, Defence, Speed and Health. You can of course add other attributes but this is a good base. I then have a method that will be called to unlock/learn the move and a method to clone a move. I add clone so that I can create a master list of moves and just clone them and add them to the player's or opponent's avatars. It is always a good idea when you are using game objects that you might want multiple of to have a way to easily make a copy of that game object. For that reason you will find that I use Clone a lot in my games.

Next, I will add the class for avatars. Right click the AvatarComponents folder, select Add and then Class. Name this new class Avatar. This class contains a lot of code so I'm going to give it to you in pieces and explain it bit by bit rather than the whole things. First, these are the using statements that I used to bring required classes into scope.

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

I didn't implement avatars as content items in my demo that I build so I included the System.IO namespace so that I can easily read/write to and from disk. I also brought in some of the XNA/MonoGame names spaces into scope.

Next up is an enumeration that defines the elements and avatar can have. Outside of the Avatar class add the following enumeration.

```
    public enum AvatarElement
    {
        Dark, Earth, Fire, Light, Water, Wind
    }
```

There are a number of fields required for an avatar. Inside of the Avatar class add the following fields.

```
        #region Field Region

        private static Random random = new Random();
        private Texture2D texture;
        private string name;
        private AvatarElement element;
        private int level;
        private long experience;
        private int costToBuy;
        private int speed;
        private int attack;
        private int defense;
        private int health;
        private int currentHealth;
        private List<IMove> effects;
        private Dictionary<string, IMove> knownMoves;

        #endregion
```

I included a static Random field in this class for random number generation for avatars. In my demo I just used a texture for the avatars so I included a Texture2D field for that. Avatars have a name so their is a field for that. They also have an element so that was added as well. They have a level and experience gained so that is there as well. They can be bought so I included a cost to buy field. Their base attributes are speed, attack, defense and health so there are fields for that. Health represents their maximum health so I added a field for their current health. The next is a List<IMove> called effects. If you look back at IMove I included status effects as well as buffs/debuffs for avatars. When one of these moves is applied to an avatar I add that move to the effects list. When it no longer affects the avatar is removed from the list. I also include a Dictionary<string, IMove> that holds the moves known by the avatar. Here I will be deviating a bit from Pokemen in that an avatar can know more than 4 moves and does not have to forget an old move to learn a new move. I will include a side tutorial that explains how to implement the learning/forgetting moves if you want to stay more true to Pokemon.

These fields need to be exposed to other classes so I had to add a number of properties. Add the following properties just below the fields inside the Avatar class.

```
#region Property Region

public string Name
{
    get { return name; }
}

public int Level
{
    get { return level; }
    set { level = (int)MathHelper.Clamp(value, 1, 100); }
}

public long Experience
{
    get { return experience; }
}

public Texture2D Texture
{
    get { return texture; }
}

public Dictionary<string, IMove> KnownMoves
{
    get { return knownMoves; }
}

public AvatarElement Element
{
    get { return element; }
}

public List<IMove> Effects
{
    get { return effects; }
}

public static Random Random
{
    get { return random; }
}

public int BaseAttack
{
    get { return attack; }
}

public int BaseDefense
{
    get { return defense; }
}

public int BaseSpeed
{
    get { return speed; }
}
```

```
public int BaseHealth
{
    get { return health; }
}

public int CurrentHealth
{
    get { return currentHealth; }
}

public bool Alive
{
    get { return (currentHealth > 0); }
}

#endregion
```

Other than Level these are all read only properties. I included a set for Level because I was capping the avatar's level at 100. Typically I would have mad the set private so that it could only be adjusted inside of the class. The other interesting thing is that instead of naming the properties for Attack, Defense, Speed and Health I place Base before each of them. That is because the effects field can possibly affect the avatar's attributes. I also included an Alive field that that can be used to check if the avatar's health is less than one an is either unconscious or defeated.

I added a private constructor to this class. It just sets the level to 1 and initializes the dictionary and list for moves. Add the following constructor below the properties.

```
#region Constructor Region

private Avatar()
{
    level = 1;
    knownMoves = new Dictionary<string, IMove>();
    effects = new List<IMove>();
}

#endregion
```

The next thing I'm going to add is the code for resolving a move. Below the constructor add the following method.

```
        public void ResoleveMove(IMove move, Avatar target)
        {
            bool found = false;
            switch (move.Target)
            {
                case Target.Self:
                    if (move.MoveType == MoveType.Buff)
                    {
                        found = false;
                        for (int i = 0; i < effects.Count; i++)
                        {
                            if (effects[i].Name == move.Name)
                            {
                                effects[i].Duration += move.Duration;
                                found = true;
                            }
                        }

                        if (!found)
                            effects.Add((IMove)move.Clone());
                    }
                    else if (move.MoveType == MoveType.Heal)
                    {
                        currentHealth += move.Health;
                        if (currentHealth > health)
                            currentHealth = health;
                    }
                    else if (move.MoveType == MoveType.Status)
                    {
                    }

                    break;
                case Target.Enemy:
                    if (move.MoveType == MoveType.Debuff)
                    {
                        found = false;
                        for (int i = 0; i < target.Effects.Count; i++)
                        {
                            if (target.Effects[i].Name == move.Name)
                            {
                                target.Effects[i].Duration += move.Duration;
                                found = true;
                            }
                        }

                        if (!found)
                            target.Effects.Add((IMove)move.Clone());
                    }
                    else if (move.MoveType == MoveType.Attack)
                    {
                        float modifier = GetMoveModifier(move.MoveElement, target.Element);

                        float tDamage = GetAttack() + move.Health * modifier -
target.GetDefense();

                        if (tDamage < 1f)
                            tDamage = 1f;

                        target.ApplyDamage((int)tDamage);
                    }

                    break;
            }
        }
```

The method accepts an IMove parameter for the move to be applied and an Avatar parameter for the target. The local variable found is used to look to see if an existing effect is found or not. There is then a switch on the target for the move. This is part of the reason why I created an interface for moves. With the interface I have all the information needed to resolve the move without knowing anything about the move being applied. I just use the contract that was defined to apply the move.

The first case that I check in the switch is if the target is Self, or the current avatar being used. I then check to see if the move type is Buff, which increases the avatar's attribute in some way. If it is I set the found variable to false. I then loop through all of the active effects that have been added to the avatar. If I find a move that has the same name I increase the current duration of the effect that is being applied to current duration. Instead of this you might want to replace the current duration with the duration for the move. It is totally up to you how you want to apply this.

I then check if the move was found or not. If it wasn't found I add a clone of the move to the list of effects currently applied to the avatar. If the move is of type Heal I increase the avatar's current health by the health modifier for the move. Then if current health is above the maximum health I set it to the maximum health. I included a check for the status of the move but didn't implement any code yet. That is because I was considering adding in more status types in my demo but never made it that far. For example, I could have included a status Invulnerable where the avatar could not be harmed by physical attacks.

Next I handle the enemy case, which is very similar to the self case. The biggest difference is that the move is applied to the enemy. It doesn't make sense to buff an enemy so I include the debuff case. This works the same as the buff case for self. I search to see if that is already there. If it is not there I add it to the list. For attacking I first call a method GetMoveModifier passing in the element for the move and the element of the target. This is where you apply logic for fire moves being strong against grass moves. I will get to that method shortly. I then call a method GetAttack that returns the avatar's attack value adding in any buffs or debuffs that have been applied. I add that to move.Health times the modifier which returns how much damage the moved does I then subtract the target's defense attribute. I then check if the damage done is less than 1 and if it is I set it to 1 because I implemented the rule that move always does 1 damage. You can also add in here if a move misses by including accuracy and such. I then call a method ApplyDamage on the target which will apply the damage. That method is yet to come.

The next method that I want to add is the one that gets if a move is effective against a certain type of avatar or not effective. Add the following method to the class.

```
public static float GetMoveModifier(MoveElement moveElement, AvatarElement avatarElement)
{
    float modifier = 1f;

    switch (moveElement)
    {
        case MoveElement.Dark:
            if (avatarElement == AvatarElement.Light)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Wind)
                modifier -= .25f;
            break;
        case MoveElement.Earth:
            if (avatarElement == AvatarElement.Water)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Wind)
                modifier -= .25f;
            break;
        case MoveElement.Fire:
            if (avatarElement == AvatarElement.Wind)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Water)
                modifier -= .25f;
            break;
        case MoveElement.Light:
            if (avatarElement == AvatarElement.Dark)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Earth)
                modifier -= .25f;
            break;
        case MoveElement.Water:
            if (avatarElement == AvatarElement.Fire)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Earth)
                modifier -= .25f;
            break;
        case MoveElement.Wind:
            if (avatarElement == AvatarElement.Light)
                modifier += .25f;
            else if (avatarElement == AvatarElement.Earth)
                modifier -= .25f;
            break;

    }

    return modifier;
}
```

This method is really just a look up table. It takes the element of the move type and compares it to the avatar's type. What this does is first sets a local variable modifier to 1f, or normal damage. In each of the cases I first check to see if the move is effective. If it is effect the move will do 25% more damage than the base so I add .25f to the modifier. If it is not effective it will to 25% less damage so I subtract .25 from the modifier. For example, a Dark move is strong against a Light avatar and does 125% of its normal damage to a Light type avatar. Similarly, it is weak against a Wind avatar and does 75% of its normal damage. All of the other cases are similar where an element is strong against one type and weak against another. In this look up table you can add a lot more cases though, and elements. This was just a good start for my demo game.

I'm going to now add two small methods. The one that applies damage and one that updates the avatar each round of combat. Add the following two methods to the class.

```
public void ApplyDamage(int tDamage)
{
    currentHealth -= tDamage;
}

public void Update(GameTime gameTime)
{
    for (int i = 0; i < effects.Count; i++)
    {
        effects[i].Duration--;

        if (effects[i].Duration < 1)
        {
            effects.RemoveAt(i);
            i--;
        }
    }
}
```

Apply damage is trivial. It just reduces the avatar's currentHealth field by the damage being passed in. Update loops through the activate effects and reduces the Duration field by 1. If the duration is less than zero I remove the effect and decrease the loop variable by 1. I do that because there is one less element in the list.

What I am going to add next are four get method that get the current attack, defense, speed and maximum health of an avatar. They all work in the same way. They loop through each of the active effects. If the effect is a buff it adds the buff to the attribute and if it is a debuff it subtracts it. It then returns the attribute plus the modifier. Add the following four methods to the class after Update.

```
public int GetAttack()
{
    int attackMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
```

```
            attackMod += move.Attack;

        if (move.MoveType == MoveType.Debuff)
            attackMod -= move.Attack;
    }

    return attack + attackMod;
}

public int GetDefense()
{
    int defenseMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
            defenseMod += move.Defense;

        if (move.MoveType == MoveType.Debuff)
            defenseMod -= move.Defense;
    }

    return defense + defenseMod;
}

public int GetSpeed()
{
    int speedMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
            speedMod += move.Speed;
        if (move.MoveType == MoveType.Debuff)
            speedMod -= move.Speed;
    }

    return speed + speedMod;
}

public int GetHealth()
{
    int healthMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
            healthMod += move.Health;
        if (move.MoveType == MoveType.Debuff)
            healthMod += move.Health;
    }

    return health + healthMod;
}
```

I'm going to add in a couple more methods now. One that will be called when combat starts. One that will be called when an avatar wins a battle and one with it loses a battle. Finally one that will be called to check if the avatar has levelled up. Add these methods after the get methods.

```
public void StartCombat()
{
    effects.Clear();
    currentHealth = health;
}
```

```
public long WinBattle(Avatar target)
{
    int levelDiff = target.Level - level;
    long expGained = 0;

    if (levelDiff <= -10)
    {
        expGained = 10;
    }
    else if (levelDiff <= -5)
    {
        expGained = (long)(100f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 0)
    {
        expGained = (long)(50f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 5)
    {
        expGained = (long)(5f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 10)
    {
        expGained = (long)(10f * (float)Math.Pow(2, levelDiff));
    }
    else
    {
        expGained = (long)(50f * (float)Math.Pow(2, target.Level));
    }

    return expGained;
}

public long LoseBattle(Avatar target)
{
    return (long)((float)WinBattle(target) * .5f);
}

public bool CheckLevelUp()
{
    bool leveled = false;

    if (experience >= 50 * (1 + (long)Math.Pow(level, 2.5)))
    {
        leveled = true;
        level++;
    }

    return leveled;
}
```

In my game after each battle the avatar returns to its element plane. There it instantly heals all damage and all status effects are removed. So, I clear the effects and reset the health in the StartCombat method. In WinBattle I decide how much experience the avatar gains for winning the battle. I first determine the level difference between the two avatars. If the player's avatar is more than ten levels higher than the opponent's avatar it gains 10 experience. If it is between 9 and 5 levels higher I use a formula to get an experience value. The way the formula works is using 2 to the power of the level difference. In this case the level difference is negative so it will return a fraction of the base. The next case is for between 0 and 4 levels higher. Again if it is negative it will return a fraction and if it is zero it will return 1. The other cases are similar when the opposing avatar was a higher level that the player's avatar. Technically the player should never win if the opposing avatar is more than 10 levels but I included it is a catch all. I know that in some games that if the opponent is that much higher you gain zero experience for defeating it. If the player does lose a battle I still reward the avatar half the experience for losing the battle. This would actually be a very high value is they lost against a much stronger avatar and will need to be tweeked accordingly. The other method that I added is called CheckLevelUp and it checks to see if the avatar has levelled up or not. I use another formula for that that uses a power variable again. This makes it so that as the avatar's level grows the experience needed to grow increase as well. This seemed to work okay in my demo but may need to be tweeked a bit in a real game.

I also included a method for levelling up an avatar. Rather than automatically adjusting the attributes like in Pokemon I allow the players to assign points to the attribute of their choice. There is a switch that checks which attribute has been chosen and updates that attribute. If they choose health I multiple that value by 5. Here is the code for that method.

```
public void AssignPoint(string s, int p)
{
    switch (s)
    {
        case "Attack":
            attack += p;
            break;
        case "Defense":
            defense += p;
            break;
        case "Speed":
            speed += p;
            break;
        case "Health":
            health += p * 5;
            break;
    }
}
```

The last method that I'm going to add is a Clone method. This can be used to grab a copy of the avatar from a master list of avatars when the player finds a new avatar. Here is a code for that method.

```
public object Clone()
{
    Avatar avatar = new Avatar();

    avatar.name = this.name;
    avatar.texture = this.texture;
    avatar.element = this.element;
    avatar.costToBuy = this.costToBuy;
    avatar.level = this.level;
    avatar.experience = this.experience;
    avatar.attack = this.attack;
    avatar.defense = this.defense;
    avatar.speed = this.speed;
    avatar.health = this.health;
    avatar.currentHealth = this.health;

    foreach (string s in this.knownMoves.Keys)
    {
        avatar.knownMoves.Add(s, this.knownMoves[s]);
    }

    return avatar;
}
```

Nothing hard about this method. It just sets the fields with the values from the current instance. To do the moves I use a foreach loop that loops over the list. I then return the new object.

I'm going to end this here as we've covered a lot and it is a very important component. I had wanted to implement some game play in this tutorial but most elements that I wanted to include use this component. For example, most NPCs have an avatar associated with them or work with avatars in some way, such as giving them to the player, training them or something similar. For battles they are definitely required. The next tutorial I will be adding in NPCs to the game and some of the components required for having conversations with NPCs.

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

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.

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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 8

Conversations

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 attaching conversations to the characters that the player will with. At a very high level a conversation is a tree of nodes that can be traversed in different ways depending on the player's choices. In the game I called a node a GameScene. The scene contains the text to be displayed to the player and one or more SceneOptions. A SceneOption has a SceneAction which determines what action is taken if the player selects that action. A full conversation is made up a number of GameScenes.

So, let's get started. First, right click the Avatars project, select Add and then New Folder. Name this new folder ConversationComponents. Now right click the ConversationComponents folder, select Add and then class name this new class SceneOption. 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.ConversationComponents
{
    public enum ActionType
    {
        Talk,
        End,
        Change,
        Quest,
        Buy,
        Sell,
        GiveItems,
        GiveKey,
    }
    public class SceneAction
    {
        public ActionType Action;
        public string Parameter;
    }

    public class SceneOption
    {
        private string optionText;
        private string optionScene;
        private SceneAction optionAction;

        private SceneOption()
        {

        }

        public string OptionText
        {
            get { return optionText; }
            set { optionText = value; }
        }

        public string OptionScene
        {
            get { return optionScene; }
            set { optionScene = value; }
        }

        public SceneAction OptionAction
        {
            get { return optionAction; }
            set { optionAction = value; }
        }

        public SceneOption(string text, string scene, SceneAction action)
        {
            optionText = text;
            optionScene = scene;
            optionAction = action;
        }
    }
}
```

I added an enumeration called ActionType that defines what action to take when the player selects that option. Talk moves the conversation to another scene in the same conversation. End ends the conversation. Change changes the current conversation to another conversation. Quest gives the player a quest if the node is selected. The Buy and Sell options were added for speaking with shopkeepers. The GiveItem and GiveKey give an item or a key to the player. Next up is a really basic class, SceneAction. This holds the action to take and any parameters that will be used based on the action chosen.

The actual SceneOption class holds the details for the option. For that there are three private member variables. The optionText field holds what is drawn on the screen. Then optionScene is what scene to change to based on the option. Finally there is a SceneAction field that defines the action taken with any parameters. There are three public properties that expose these values to other classes. There is also a private constructor that takes no parameters that will be used for writing conversations and reading them back in. There is a second constructor that takes three parameters which are the text displayed, the scene being transitioned to and the action.

With the SceneOption in place I can now create the GameScene class. Right click the

ConversationComponents folder, select Add and then Class. Name this new class GameScene. This is the code for that class.

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

namespace Avatars.ConversationComponents
{
    public class GameScene
    {
        #region Field Region

        protected Game game;
        protected string text;
        private List<SceneOption> options;
        private int selectedIndex;
        private Color highLight;
        private Color normal;
        private Vector2 textPosition;
        private static Texture2D selected;
        private bool isMouseOver;

        private Vector2 menuPosition = new Vector2(50, 475);

        #endregion

        #region Property Region

        public string Text
        {
            get { return text; }
            set { text = value; }
        }

        public static Texture2D Selected
        {
            get { return selected; }
        }

        public List<SceneOption> Options
        {
            get { return options; }
            set { options = value; }
        }

        [ContentSerializerIgnore]
        public SceneAction OptionAction
        {
            get { return options[selectedIndex].OptionAction; }
        }

        public string OptionScene
        {
            get { return options[selectedIndex].OptionScene; }
        }

        public string OptionText
        {
            get { return options[selectedIndex].OptionText; }
        }

        public int SelectedIndex
        {
            get { return selectedIndex; }
        }

        public bool IsMouseOver
        {
            get { return isMouseOver; }
        }

        [ContentSerializerIgnore]
        public Color NormalColor
        {
            get { return normal; }
            set { normal = value; }
        }

        [ContentSerializerIgnore]
        public Color HighLightColor
        {
            get { return highLight; }
            set { highLight = value; }
        }

        public Vector2 MenuPosition
        {
            get { return menuPosition; }
        }

        #endregion

        #region Constructor Region

        private GameScene()
        {
            NormalColor = Color.Blue;
            HighLightColor = Color.Red;
        }

        public GameScene(string text, List<SceneOption> options)
        {
            this.text = text;
            this.options = options;
            textPosition = Vector2.Zero;
        }

        public GameScene(Game game, string text, List<SceneOption> options)
        {
            this.game = game;

            this.options = new List<SceneOption>();
            this.highLight = Color.Red;
            this.normal = Color.Black;

            this.options = options;
        }

        #endregion

        #region Method Region

        public void SetText(string text, SpriteFont font)
        {
            textPosition = new Vector2(450, 50);

            StringBuilder sb = new StringBuilder();
            float currentLength = 0f;

            if (font == null)
            {
                this.text = text;
                return;
            }

            string[] parts = text.Split(' ');

            foreach (string s in parts)
            {
                Vector2 size = font.MeasureString(s);

                if (currentLength + size.X < 500f)
                {
                    sb.Append(s);
                    sb.Append(" ");
                    currentLength += size.X;
                }
                else
                {
                    sb.Append("\n\r");
                    sb.Append(s);
                    sb.Append(" ");
                    currentLength = 0;
                }
            }

            this.text = sb.ToString();
        }

        public void Initialize()
        {
        }

        public void Update(GameTime gameTime, PlayerIndex index)
        {
            if (Xin.CheckKeyReleased(Keys.Down))
            {
                selectedIndex--;
                if (selectedIndex < 0)
                    selectedIndex = options.Count - 1;
            }
            else if (Xin.CheckKeyReleased(Keys.Down))
            {
                selectedIndex++;
                if (selectedIndex > options.Count - 1)
                    selectedIndex = 0;
            }
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Texture2D background,
SpriteFont font)
        {
            Vector2 selectedPosition = new Vector2();
            Color myColor;

            if (selected == null)
                selected = game.Content.Load<Texture2D>(@"Misc\selected");

            if (textPosition == Vector2.Zero)
```

```
                SetText(text, font);

            if (background != null)
                spriteBatch.Draw(background, Vector2.Zero, Color.White);

            spriteBatch.DrawString(font,
                text,
                textPosition,
                Color.White);

            Vector2 position = menuPosition;

            Rectangle optionRect = new Rectangle(0, (int)position.Y, 1280,
font.LineSpacing);
            isMouseOver = false;

            for (int i = 0; i < options.Count; i++)
            {
                if (optionRect.Contains(Xin.MouseState.Position))
                {
                    selectedIndex = i;
                    isMouseOver = true;
                }

                if (i == SelectedIndex)
                {
                    myColor = HighLightColor;
                    selectedPosition.X = position.X - 35;
                    selectedPosition.Y = position.Y;

                    spriteBatch.Draw(selected, selectedPosition, Color.White);
                }
                else
                    myColor = NormalColor;

                spriteBatch.DrawString(font,
                    options[i].OptionText,
                    position,
                    myColor);

                position.Y += font.LineSpacing + 5;
                optionRect.Y += font.LineSpacing + 5;
            }
        }

        #endregion
    }
}
```

There is a lot going on in this class. I'll tackle member variables first. There is a Game type field that is the reference to the game. It is used for loading content using the content manager. Next is text and it is used in a method further on in the class so that the text for the scene wraps in the screen area. Next is a List<SceneOption> that is the options for the scene. The selectedIndex member is what scene option is currently selected. The two Color fields hold the color to draw unselected and selected options. There is also a Vector2 that controls where the scene text is rendered and a Texture2D selected that will be drawn beside the currently selected scene option. This one is static and will be shared by all instances of GameScene. The next field I included is isMouse over and will be used to test if the mouse is over an SceneOption. The last member variable is menuPosition and controls where the scene options are drawn.

Next are a number of properties to expose the member variables to other classes. The only thing out of the normal is that I've marked a few with attributes that define how the class is serialized using the IntermediateSerializer because I don't want some of the members serialized so that they are set at runtime rather than at build time.

Next up are the three constructors for this class. The first requires no parameters and is required to deserialize and load the exported XML content. The second is used in the editor to create scenes. The third is used in the game when creating scenes on the fly.

I added a method called SetText and it takes as parameters text and font. First, I set the position of where to draw the text. You will notice that the position is almost half way over to the right. This is because when I call Draw to draw the scene it accepted a Texture2D parameter called portrait that represented the portrait of the character the player is speaking to. I'm not implementing that for a while yet but I want it available if needed. After setting the position I create a StringBuilder that will be used to convert the single line of text to multiple lines of text. There is then a local variable, currentLength, that holds the length of the current line. I then check to make sure the font member variable is not null. It if is I just set the member variable to the parameter and exit the method.

I then use the Split method of the string class to split the string into parts on the space character. Next in a foreach loop I iterate over all of the parts. I then use MeasureString to determine the length of that word. If the length of the word is less than the maximum length of text on the screen I append the part to the string builder with a space and update the line length.

If the length is greater than the maximum length I append a carriage return, append the text and then append a space. I can do that because rendering text with DrawString allows for escape characters like \n and \r. I then reset currentLength to size.X. The last thing to do in this method is to set the text member variable to the string builder as a string.

Next there is an empty method, Initialize, that will be updated to initialize the scene if necessary. I included it now as I do use it my games an will be used in the future.

The Update method takes a GameTime parameter and a PlayerIndex parameter. The index parameter is the index of the current game pad. It can be excluded if you do not want to support game pads in your game. In the Update method I check to see if the player has requested to move the selected item up or down. I check if moving the item up or down exceeds the bounds of the list of options and if does I wrap to either the first or last item in the list.

The last thing to do is draw the scene. The Draw method takes a GameTime parameter, SpriteBatch parameter and a Texture2D for the background image and a SpriteFont to draw the text with. There are local variables that determine where to draw the selected item indicator, the speaker's portrait and the color to draw scene options with.

I check to see if selected texture is null. If it is null I load it. If textPosition is Vector2.Zero then the text has not been set so I set it. Next if the background texture is not null I draw the background.

After drawing the background I draw the speaker's text in white. You can include a property in the game scene for what color to draw the speaker's text in rather than hard coding it.

There is then a Vector2 local variable that I assign the position of the scene options to be drawn. I then create a rectangle based on the position that is the width of the screen. I then loop over all of the options for the player to choose from as a reply to the current scene. Inside that loop I check if the mouse is included in the rectangle. If it is I set the selectedIndex member variable to the current loop index. Next I check to see if the current loop index is the selectedIndex. If it is I draw the selected item texture to the left of that option and I set the local color variable to the highlight color. If it is not then the local color variable is set to the base option color. I then draw the scene option. At the end of the loop I update the Y value for the position to be the line spacing for the font plus 5 pixels.

Now I'm going to add in the class that represents a conversation. Right click the ConversationComponents folder, select Add and then Class. Name this new class Conversation. Here is the code for that class.

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

namespace Avatars.ConversationComponents
{
    public class Conversation
    {
        #region Field Region

        private string name;
        private string firstScene;
        private string currentScene;
        private Dictionary<string, GameScene> scenes;
        private string bsckgroundName;
        private Texture2D background;
        private string fontName;
        private SpriteFont spriteFont;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public string FirstScene
        {
            get { return firstScene; }
        }

        public GameScene CurrentScene
        {
            get { return scenes[currentScene]; }
        }

        public Dictionary<string, GameScene> Scenes
        {
            get { return scenes; }
        }

        public Texture2D Background
        {
            get { return background; }
        }

        public SpriteFont SpriteFont
        {
            get { return spriteFont; }
        }

        public string BackgroundName
        {
            get { return backgroundName; }
            set { backgroundName = value; }
        }

        public string FontName
        {
            get { return fontName; }
            set { fontName = value; }
        }

        #endregion

        #region Constructor Region

        public Conversation(string name, string firstScene, Texture2D background, SpriteFont
font)
        {
            this.scenes = new Dictionary<string, GameScene>();
            this.name = name;
            this.firstScene = firstScene;
            this.background = background;
            this.spriteFont = font;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            CurrentScene.Update(gameTime, PlayerIndex.One);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            CurrentScene.Draw(gameTime, spriteBatch, background, spriteFont);
        }

        public void AddScene(string sceneName, GameScene scene)
        {
            if (!scenes.ContainsKey(sceneName))
                scenes.Add(sceneName, scene);
        }

        public GameScene GetScene(string sceneName)
        {
            if (scenes.ContainsKey(sceneName))
                return scenes[sceneName];

            return null;
        }

        public void StartConversation()
        {
            currentScene = firstScene;
        }

        public void ChangeScene(string sceneName)
        {
            currentScene = sceneName;
        }

        #endregion
    }
}
```

I added in member variables for the name of the conversation, the first scene of the conversation, the current scene of the conversation, a dictionary of scenes, a texture for the conversation and a font for the scene. I also include member variables for the name of the background for the conversation and the name of the font the text will be drawn with.

Next there are properties that expose the member variables. The ones for the name fields are both getters and setters. Most of the others are simple getters only. The one that is more than just a simple getter is the one that returns the current scene. Rather than returning the key for the scene I return the scene using the key.

The constructor for this class takes four parameters: name, firstScene, background and font. They represent the name of the conversation, the first scene to be displayed, the background for the conversation and the font the conversation is drawn with. I just set the fields with the parameters that are passed in.

The scene needs to have its Update method called so I included an Update method in this class. It calls the Update method of the current scene for the conversation. The conversation needs to be drawn so there is a Draw method for the conversation. It just calls the Draw method of the current scene.

You can just use the raw dictionary to add and retrieve scenes but I included a method for adding a scene and a method for getting a scene. The reason being is that I can validate the values and prevent the game from crashing if something unexpected is passed in.

The last two methods on this class are StartConversation and ChangeScene. StartConversation sets the currentScene member variable to the firstScene member variable. ChangeScene changes the scene to the parameter that is passed in.

The last thing that I'm going to add in this tutorial is a class that manages the conversations in the game. Right click the ConversationComponents folder, select Add and then Class. Name this new class ConversationManager. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Xml;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

```
namespace Avatars.ConversationComponents
{
    public class ConversationManager
    {
        #region Field Region

        private static Dictionary<string, Conversation> conversationList = new
Dictionary<string, Conversation>();

        #endregion

        #region Property Region

        public static Dictionary<string, Conversation> ConversationList
        {
            get { return conversationList; }
        }

        #endregion

        #region Constructor Region

        public ConversationManager()
        {
        }

        #endregion

        #region Method Region
        public static void AddConversation(string name, Conversation conversation)
        {
            if (!conversationList.ContainsKey(name))
                conversationList.Add(name, conversation);
        }

        public static Conversation GetConversation(string name)
        {
            if (conversationList.ContainsKey(name))
                return conversationList[name];

            return null;
        }

        public static bool ContainsConversation(string name)
        {
            return conversationList.ContainsKey(name);
        }

        public static void ToFile(string fileName)
        {
            XmlDocument xmlDoc = new XmlDocument();

            XmlElement root = xmlDoc.CreateElement("Conversations");
            xmlDoc.AppendChild(root);

            foreach (string s in ConversationManager.ConversationList.Keys)
            {
                Conversation c = ConversationManager.GetConversation(s);

                XmlElement conversation = xmlDoc.CreateElement("Conversation");

                XmlAttribute name = xmlDoc.CreateAttribute("Name");
                name.Value = s;
                conversation.Attributes.Append(name);
                XmlAttribute firstScene = xmlDoc.CreateAttribute("FirstScene");
                firstScene.Value = c.FirstScene;
                conversation.Attributes.Append(firstScene);

                XmlAttribute backgroundName = xmlDoc.CreateAttribute("BackgroundName");
                backgroundName.Value = c.BackgroundName;
                conversation.Attributes.Append(backgroundName);

                XmlAttribute fontName = xmlDoc.CreateAttribute("FontName");
                fontName.Value = c.FontName;
                conversation.Attributes.Append(fontName);

                foreach (string sc in c.Scenes.Keys)
                {
                    GameScene g = c.Scenes[sc];

                    XmlElement scene = xmlDoc.CreateElement("GameScene");

                    XmlAttribute sceneName = xmlDoc.CreateAttribute("Name");
                    sceneName.Value = sc;

                    scene.Attributes.Append(sceneName);

                    XmlElement text = xmlDoc.CreateElement("Text");
                    text.InnerText = c.Scenes[sc].Text;

                    foreach (SceneOption option in g.Options)
                    {
                        XmlElement sceneOption = xmlDoc.CreateElement("GameSceneOption");

                        XmlAttribute oText = xmlDoc.CreateAttribute("Text");
                        oText.Value = option.OptionText;
                        sceneOption.Attributes.Append(oText);

                        XmlAttribute oOption = xmlDoc.CreateAttribute("Option");
                        oOption.Value = option.OptionScene;
                        sceneOption.Attributes.Append(oOption);

                        XmlAttribute oAction = xmlDoc.CreateAttribute("Action");
                        oAction.Value = option.OptionAction.ToString();
                        sceneOption.Attributes.Append(oAction);

                        XmlAttribute oParam = xmlDoc.CreateAttribute("Parameter");
                        oParam.Value = option.OptionAction.Parameter;

                        scene.AppendChild(sceneOption);
                    }

                    conversation.AppendChild(scene);
                }

                root.AppendChild(conversation);
            }

            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;
            settings.Encoding = Encoding.UTF8;

            FileStream stream = new FileStream(fileName, FileMode.Create, FileAccess.Write);
            XmlWriter writer = XmlWriter.Create(stream, settings);
            xmlDoc.Save(writer);
        }

        public static void FromFile(string fileName, Game gameRef, bool editor = false)
        {
            XmlDocument xmlDoc = new XmlDocument();
            try
            {
                xmlDoc.Load(fileName);

                XmlNode root = xmlDoc.FirstChild;

                if (root.Name == "xml")
                    root = root.NextSibling;

                if (root.Name != "Conversations")
                    throw new Exception("Invalid conversation file!");

                foreach (XmlNode node in root.ChildNodes)
                {
                    if (node.Name == "#comment")
                        continue;

                    if (node.Name != "Conversation")
                        throw new Exception("Invalid conversation file!");

                    string conversationName = node.Attributes["Name"].Value;
                    string firstScene = node.Attributes["FirstScene"].Value;
                    string backgroundName = node.Attributes["BackgroundName"].Value;
                    string fontName = node.Attributes["FontName"].Value;

                    Texture2D background = gameRef.Content.Load<Texture2D>(@"Backgrounds\" +
backgroundName);
                    SpriteFont font = gameRef.Content.Load<SpriteFont>(@"Fonts\" +
fontName);

                    Conversation conversation = new Conversation(conversationName,
firstScene, background, font);
                    conversation.BackgroundName = backgroundName;
                    conversation.FontName = fontName;

                    foreach (XmlNode sceneNode in node.ChildNodes)
                    {
                        string text = "";
                        string optionText = "";
                        string optionScene = "";
                        string optionAction = "";
                        string optionParam = "";
                        string sceneName = "";

                        if (sceneNode.Name != "GameScene")
                            throw new Exception("Invalid conversation file!");

                        sceneName = sceneNode.Attributes["Name"].Value;

                        List<SceneOption> sceneOptions = new List<SceneOption>();

                        foreach (XmlNode innerNode in sceneNode.ChildNodes)
                        {
                            if (innerNode.Name == "Text")
                                text = innerNode.InnerText;

                            if (innerNode.Name == "GameSceneOption")
                            {
                                optionText = innerNode.Attributes["Text"].Value;
                                optionScene = innerNode.Attributes["Option"].Value;
                                optionAction = innerNode.Attributes["Action"].Value;
                                optionParam = innerNode.Attributes["Parameter"].Value;

                                SceneAction action = new SceneAction();
                                action.Parameter = optionParam;
                                action.Action = (ActionType)Enum.Parse(typeof(ActionType),
optionAction);

                                SceneOption option = new SceneOption(optionText,
optionScene, action);
                                sceneOptions.Add(option);
                            }
                        }

                        GameScene scene = null;

                        if (editor)
                            scene = new GameScene(text, sceneOptions);
                        else
                            scene = new GameScene(gameRef, text, sceneOptions);


                        conversation.AddScene(sceneName, scene);
                    }

                    conversationList.Add(conversationName, conversation);
                }
            }
            catch
            {
            }
            finally
            {
                xmlDoc = null;
            }
        }

        #endregion

        public static void ClearConversations()
        {
            conversationList = new Dictionary<string, Conversation>();
        }
    }
}
```

All of the members, other than the constructor, are all static. I did this because I am sharing this component with other classes. It is not really "best practices" to do this though. In this case it is for a demo and will suit our purposes. You would probably want to update the manager to be align with best object-oriented programming practices in a production game.

The only member variable holds the conversations in the manager. There is also a property that exposes the member variable. The constructor takes no parameters and does no actions. It was included for use in the future.

There are a number of static methods next. The first is AddConversation is and called to add a conversation to the manager. You can also use the static property and add the conversation that way. This just adds a little error checking to make sure a conversation with the given key does not already exists. This was added mostly for the editor so that if you try to add the same conversation twice you will get an error message.

The next static method is GetConversation and is used to retrieve a conversation from the manager. This could also be done with the property as well. I added it because it first checks to see that the conversation exists before trying to retrieve it. This would prevent a crash if the conversation was not present. You would have to handle the null value that is returned or that could crash the game.

ContainsConverstaion just checks to see if the give key is present in the dictionary and returns that back. This was also added to try and have better error checking before adding or retrieving a conversation.

Next is the method ToFile. This method is used to write the conversation manager to an XML document. I went this route to show one of the few ways that I use to read and write content without using the content manager. The method takes as a parameter the name of the file to write the conversation manager to. Creating XML documents through code is process of creating a root node and then appending children to that node. Those children can also have child nodes under them.

The first thing to do is to create a new XmlDocument. To that I add the root element for the document, Conversations. Next that is appended to the document. The next step is to loop over all of the conversations and create a node for them to append them to the root node of the document.

The first step is to get the conversation using the key from the foreach loop that iterates over the collection of conversations. Next I create a new element/node. I then create attributes for the element for the name, firstScene, backgroundName and fontName members. There is then another foreach loop to go over the scene collection.

Inside that loop I first get the scene using the key. I then create an element for that scene. I then create an attribute for the name of the scene and the text for the scene.

There is then another foreach loop to iterate over the options for that scene. I then create an element for that option. I then create attributes for the three scene properties, text, option and parameter. I then append that element to the scene element. Next the scene is appended to the conversation. The conversation is then appended to the root.

Now that the document is created it needs to be written out. I use the XmlWriter class for that. It requires and XmlWriterSettings parameter so I create that with standard XML attributes, indentation and encoding as UTF8. Next I create a FileStream that is required using Create and Write options. The Create option will create a new file if the file does not exist or overwrite the existing file if does exist. I then create the writer and write the file

The next static method is FromFile that will parse the XML document that was written out earlier. The first thing to do is to create an XmlDocument object. I then do everything inside of a try-catch-finally block to prevent crashes.

The XmlDocument class has a method Load that will load the document into that object. The object can then be parsed and the data be pulled out. I grab the root node of the document using the FirstChild property. If the name is xml then it is the header and we need to go down a level so I set the root node to its next sibling. I then compare that to Conversations. If it is not Conversations then the file is not in the format that we are expecting and I throw an exception.

I then iterate over all of the child nodes using the ChildNodes collection. I check to see if the name is comment, if it is I move onto the next node. Next I check to see if it is Conversation. If it is not conversation the document is not in the right format so I throw an exception. I then grab the name, first scene, background and font attributes. Next up I use the game object passed in to load the background texture and the sprite font. I then create a conversation using the attributes and assign the background and font properties.

Since that node should have child nodes I have a foreach loop that will iterate over them. Inside that loop I have some local variables to hold values that are required for scenes and scene options. If the name of the node is not GameScene I throw an exception. I then grab the Name attribute and assign it to the sceneName variable. Next I create a list of scene options that is required for creating the game scene. I then iterate over the child nodes. If the name is Text then I set the text variable to the inner text of that node. If it is GameSceneOption I assign the local variables to the attributes of the node. Afterwards I create an action and then the option using the text and the action. I then create a GameScene object and initialize it, so the compiler will not complain it has not been initialized. If we are in the editor I use the first constructor, otherwise is use the second constructor. I then add the scene to the conversation. The conversation is then added to the list of conversations.

I don't do anything in the catch but it is a good idea to handle it in some way. I will cover that in another tutorial. In the finally, which is always called, I set the xmlDoc to null.

There is one other static method, ClearConversations, that creates a new list of conversations. This could tax the garbage collector a bit so it might be best to use the Clear method to remove the elements instead.

I'm going to end the tutorial here as it is a lot to digest in one sitting. In the next tutorial I will cover updating the game to allow for conversations with other characters in 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 9

Conversations Continued

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.

In this tutorial I will be continuing on with having conversations with the characters in the game. First I will add the assets for conversations. Conversations require text, an object to indicate an item is selected and they require a background. You can download the items I used from this link.

After you have downloaded and extracted the content open the MonoGame content builder. Select the Fonts folder and then Add Existing Item button in the tool bar. Browse for the scenefont.spritefont and add it to that to the projection. Now select the Content node and select the New Folder button on the tool bar and name this folder Scenes. Select the Scenes folder and then click the Add Existing Item button in the tool bar. Browse for the scenebackground.png file and add it to the folder. Now, select the Misc folder, and from the toolbar select Add Exiting item. Navigate to the selected.png file and add it to that folder. From the toolbar hit the Save button and then the Build button to build the content. Now close the content builder.

Before we go much further I want to fix a bug that I found in the GameScene class that I created. In the constructors for the class not all of the fields were being assigned correctly. Update those constructors as follows.

```
private GameScene()
{
    NormalColor = Color.Blue;
    HighLightColor = Color.Red;
}

public GameScene(string text, List<SceneOption> options)
    : this()
{
    this.text = text;
    this.options = options;
    textPosition = Vector2.Zero;
}

public GameScene(Game game, string text, List<SceneOption> options)
    : this(text, options)
{
    this.game = game;
}
```

I also want to make an addition to the AnimatedSprite class. What I want to do is add a calculated property that will return the center of the sprite on the screen. This will be used to tell if two sprites are close together. All it does is take the Position of the sprite and add half the height and width to the position to get its center. Add the following property to the AnimatedSprite class.

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

The next thing is to add a state to handle conversations. Right click the GameStates folder, select Add and then Class. Name this new class ConversationState. The initial code for that class follows next.

```
using Avatars.CharacterComponents;
using Avatars.ConversationComponents;
using Avatars.PlayerComponents;
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.GameStates
{
    public interface IConversationState
    {
        void SetConversation(Player player, ICharacter character);
        void StartConversation();
    }

    public class ConversationState : BaseGameState, IConversationState
    {
        #region Field Region

        private Conversation conversation;
        private SpriteFont font;
        private Texture2D background;
        private Player player;
        private ICharacter speaker;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public ConversationState(Game game)
            : base(game)
        {
            game.Services.AddService(typeof(IConversationState), this);
        }
        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            font = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");
            background = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");
            base.LoadContent();
        }

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

            GameRef.SpriteBatch.Begin();
            conversation.Draw(gameTime, GameRef.SpriteBatch);
            GameRef.SpriteBatch.End();
        }

        public void SetConversation(Player player, ICharacter character)
        {
            this.player = player;
            speaker = character;

            if (ConversationManager.ConversationList.ContainsKey(character.Conversation))
                this.conversation =
ConversationManager.ConversationList[character.Conversation];
            else
                manager.PopState();
        }

        public void StartConversation()
        {
            conversation.StartConversation();
        }

        #endregion
    }
}
```

There are first a number of using statements to bring classes from other namespaces into scope for this class rather than need to explicitly reference them. I then added in an interface IconversationState. This interface will be implemented in the class and the class will register the interface as the service. The methods that must be implemented are SetConversation and StartConversation. SetConversation requires two parameters, the Player object and an ICharacter. Since I implemented Character and PCharacter from this interface either can be passed to this method. This method will initialize the conversation. The other method StartConversation will begin the conversation between the player and the character.

The class inherits from BaseGameState so that it can be used with the state manager and it implements the interface. There are a few private member variables. They hold the Conversation that is currently in progress, the SpriteFont to draw text with, the Texture2D for the conversation scene, the Player object and an ICharacter ther represents the character the player is talking with. I did not include any properties in this class to expose member variable. If you need to expose a member variable it would be best to include it in the interface, like the member variables, and implement them as part of the interface.

There is another reason why that this is important that I have not discussed yet. Suppose that you have created a library from your game code so that you can reuse in other games. Now, suppose you need to extend an object in the library, like adding a new ICharacter type. Rather than adding the new class to the library you can create a new class in the game and have it implement ICharacter. That means that anywhere in the library that you have used ICharacter as a type you can pass the new type from the game and you will not need to recompile the library to use this new object. In essence you are making a library that you can create plugins for. You could also publish this library for others to use in their games.

The constructor registers this instance of the object as a service that can be retrieved and consumed as a service in another class. This goes with my last comment. The class can be included in a library and then consumed in another project. The user will only know about what you've published in the interface. They can also extend it by implementing the interface in their own class(es).

In the LoadContent method I load in the font and the background into their respective member variables. The Draw method just calls the Draw method of the current conversation being displayed. SetConversation sets the player member to the value passed in and speaker to the value passed in. It then checks to see if the character has a conversation associated with them. If they do I set the conversation member variable to that conversation. If they don't have a conversation associated with them I pop the state off the stack and return control back to the calling class. The StartConversation method just calls the StartConversation method of the conversation. It would probably be a good idea to make sure that it is not null before doing this but I will leave that as an exercise.

One method is missing from this class, the Update method. That is because it is the brains of this class and it will need to be implemented in stages. First, add these two using statements with the others to bring some classes into scope in this class.

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

I added those because we will be using the input manager to test if the player has selected a scene option. Avatars.Components brings the input handler, Xin, into scope and the other brings in items like the Keys enumeration into scope.

Now, add this Update method to the class between LoadContent and Draw.

```
public override void Update(GameTime gameTime)
{
    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        switch (conversation.CurrentScene.OptionAction.Action)
        {
            case ActionType.Buy :
                break;
            case ActionType.Change :
                speaker.SetConversation(conversation.CurrentScene.OptionScene);
                manager.PopState();
                break;
            case ActionType.End :
                manager.PopState();
                break;
            case ActionType.GiveItems :
                break;
            case ActionType.GiveKey :
                break;
            case ActionType.Quest :
                break;
            case ActionType.Sell :
                break;
            case ActionType.Talk :
                conversation.ChangeScene(conversation.CurrentScene.OptionScene);
                break;
        }
    }
    conversation.Update(gameTime);
    base.Update(gameTime);
}
```

This current implementation checks to see if the space or enter keys have been released. If they have been released there is a switch statement on the currently selected scene option. I then included a case for each value in the enumeration for ActionType. For this tutorial I added code to handle three different actions: Change, End and Talk.

What Change does is changes the conversation for the speaker to an entirely new conversation. To do that I call the SetConversation method on the speaker passing in the OptionScene. The next step is to pop the state off the stack. You would use this when the player is speaking to a character and they choose an option that would end the conversation but the next time they talk to this character you want a different conversation.

End is a trivial case. All it does is pop the conversation state off the stack. As you've guessed this is used when the player stops talking to the character.

Change is a trivial case as well. What it does is call the ChangeScene method passing in the OptionScene of the current scene. This will be used when you want to move the conversation to another branch such as a continue option or answering yes or no to a question from the character.

The last thing to do is to call the Update method on the conversation. If you don't the player will not be able to change their response during the conversation.

Now, let's add this to the game. I'm going to post the code for the entire Game1 class, just to make it easier to follow the changes that are being made. What I have done though is first add a new field of type IConversationState. The next change was to create an instance of the new ConversationState class and assign it to that field. I didn't include a property to expose it because I will be demonstrating how to retrieve it as a service. Here is the code for the new Game1 class.

```
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;
        IConversationState conversationState;

        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);
            conversationState = new ConversationState(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 I want to do is to create a conversation for each of the characters that I added to the demo. To do that I added a new method to the ConversationManager class called CreateConversations. Add this method to that class.

```
public static void CreateConversations(Game gameRef)
{
    Texture2D sceneTexture = gameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");
    SpriteFont sceneFont = gameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");

    Conversation c = new Conversation("MarissaHello", "Hello", sceneTexture, sceneFont);
    c.BackgroundName = "scenebackground";
    c.FontName = "scenefont";

    List<SceneOption> options = new List<SceneOption>();
    SceneOption option = new SceneOption(
        "Good bye.",
        "",
        new SceneAction() { Action = ActionType.End, Parameter = "none" });
    options.Add(option);

    GameScene scene = new GameScene(
        gameRef,
        "Hello, my name is Marissa. I'm still learning about summoning avatars.",
        options);

    c.AddScene("Hello", scene);

    ConversationList.Add("MarissaHello", c);

    c = new Conversation("LanceHello", "Hello", sceneTexture, sceneFont);
    c.BackgroundName = "scenebackground";
    c.FontName = "scenefont";

    options = new List<SceneOption>();
    option = new SceneOption(
        "Yes",
        "ILikeFire",
        new SceneAction() { Action = ActionType.Talk, Parameter = "none" });
    options.Add(option);

    option = new SceneOption(
        "No",
        "IDislikeFire",
        new SceneAction() { Action = ActionType.Talk, Parameter = "none" });
    options.Add(option);

    scene = new GameScene(
        gameRef,
        "Fire avatars are my favorites. Do you like fire type avatars too?",
        options);

    c.AddScene("Hello", scene);

    options = new List<SceneOption>()
    option = new SceneOption(
                "Good bye.",
                "",
                new SceneAction() { Action = ActionType.End, Parameter = "none" });
    options.Add(option);

    scene = new GameScene(
        gameRef,
        "That's cool. I wouldn't want to hug one though.",
        options);

    c.AddScene("ILikeFire", scene);

    options = new List<SceneOption>()
    option = new SceneOption(
                "Good bye.",
                "",
                new SceneAction() { Action = ActionType.End, Parameter = "none" });
    options.Add(option);

    scene = new GameScene(
        gameRef,
        "Each to their own I guess.",
        options);

    c.AddScene("IDislikeFire", scene);

    conversationList.Add("LanceHello", c);
}
```

First thing of note is that this takes a Game type parameter. It is used to give us access to the content manager for importing the background for the scene and the font for the scene. That is exactly what I did at the start of this method.

The first conversation that I'm going to implement is the simplest type. It just one scene and the scene has one option. I first created a Conversation object with the parameters: "MarissaHello", "Hello", sceneTexture and sceneFont. The first is the name of the conversation, followed by the name of the first scene and the texture and font.

Next is a List<SceneOption> to hold the options for the scene I'm adding to the conversation. I then create a SceneOption object with the text Good bye, no scene to transition to and for the SceneAction is has ActionType.End and none for the parameter. If you recall from above this will just terminate the conversation and pop that state off the stack. This option is then added to the list.

Since a Conversation is made up of GameScene object I create one with the text to be displayed and the List<SceneOption> that I already created. The scene is then added to the conversation and the conversation is listed to the conversation list.

The other conversation constructed similarly. The difference is that the first scene has two options associated with it, a Yes option and a No option. These options use ActionType.Talk and change to the different branches, ILikeFire or IdislikeFire. Those two scene options display some text based on the player's choice.

The next thing to do for the demo is to create the conversations and then attach them to the characters. I did that in the SetUpNewGame in the GamePlayState class. Update that method as follows.

```
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();

    ConversationManager.CreateConversations(GameRef);

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

```
    teacherOne.SetConversation("LanceHello");
    teacherTwo.SetConversation("MarissaHello");

    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();
}
```

All I did was call the static CreateConversation method on the ConversationManager class. I also called the SetConversation method on the characters that I created and passed in the name of each conversation, LanceHello and MarissaHello.

The last update to have conversations working in the demo is to modify the Update method of the GamePlayState class to check if the space bar or enter key have been pressed and if they have check to see if the player is close to a character. If they are close to a character start a conversation with that character. Update that method as follows.

```
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);

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        foreach (string s in map.Characters.Keys)
        {
            ICharacter c = CharacterManager.Instance.GetCharacter(s);
            float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

            if (Math.Abs(distance) < 72f)
            {
                IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                manager.PushState(
                    (ConversationState)conversationState,
                    PlayerIndexInControl);

                conversationState.SetConversation(player, c);
                conversationState.StartConversation();
                }
            }
        }
    base.Update(gameTime);
}
```

What I did is add an if statement to check if either the space bar or enter key have been released since the last frame. I then iterate over all of the keys of the Characters property of the current map. I then grab the character with that name from the CharacterManager. I then use Vector2.Distance to get how far apart the two centers are. If the absolute value is less than 72 I get the IConversationState that was registered as a service. I then call the SetConversation method passing in the player object and the character object. Finally I start the conversation.

Now, I'm going to explain why I find the distance and compare it to 72. What I'm doing here is a type of collision detection called bounding circles rather than bounding boxes. What that means is I construct a circle around the objects and check to see if they collide by calculating the distance between their centers. I pad the circle so that the player just has to be close, not necessarily colliding with the character. I also like to call this type of collision detection proximity collision detection rather.

I'm going to end the tutorial here as it is a lot to digest in one sitting. In the next tutorial I will cover updating the game to allow for conversations with other characters in 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 10

Creating Avatars

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.

This tutorial will cover creating avatars and the moves that they can learn. The way that I implemented avatars in the demo that I created was as a CSV file. That data is read when the game loads and the avatars are then created from each line in the CSV file. This is what a sample avatar looked like in my demo.

```
Fire,Fire,100,1,12,8,10,50,Tackle:1,Block:1,Flare:2,None:100,None:100,None:100
```

The first value is the avatar's name, really creative wasn't I. The next value is the avatars element. The next value was how much it would cost to buy the avatar from a character. The next value is the avatar's level. The next four elements are the attack, defense, speed and health of the avatar. The next six parameters are the moves that the avatar knows or can learn. They consist of two values. The name of the move and the level at which it unlocks. I will implement this a little differently in the tutorial. I'm going to make the move list dynamic but still follow the same rule.

Before I implement this I want to add in a manager class to manage moves and avatars. Right click the AvatarComponents folder, select Add and then Class. Name this new class MoveManager. Here is the code.

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

namespace Avatars.AvatarComponents
{
    public static class MoveManager
    {
        #region Field Region
        private static Dictionary<string, IMove> allMoves = new Dictionary<string, IMove>();
        private static Random random = new Random();

        #endregion

        #region Property Region

        public static Random Random
        {
            get { return random; }
        }

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region

        public static void FillMoves()
        {
            //AddMove(new Tackle());
            //AddMove(new Block());
            //AddMove(new Haste());
            //AddMove(new Bless());
            //AddMove(new Curse());
            //AddMove(new Heal());
            //AddMove(new Flare());
            //AddMove(new Shock());
            //AddMove(new Gust());
            //AddMove(new Frostbite());
            //AddMove(new Shade());
            //AddMove(new Burst());
            //AddMove(new RockThrow());
        }

        public static IMove GetMove(string name)
        {
            if (allMoves.ContainsKey(name))
                return (IMove)allMoves[name].Clone();

            return null;
        }

        public static void AddMove(IMove move)
        {
            if (!allMoves.ContainsKey(move.Name))
                allMoves.Add(move.Name, move);
        }

        #endregion
    }
}
```

This is like most of the other managers that we've implemented through out the game so far. It consists of a static member variable to hold all of the objects that are being managed, in this case IMove. I included a Random member variable that will be shared by all of the moves as well. This just helps to make sure that the random number generation is consistent across all objects. You can also specify a seed here so that the same number series is generated each time that you run the game. It would be a good idea to do that when in debug mode by not when in release mode. Another difference in this class is that I don't exposes the moves with a property. I force the use of the get and add methods to get and add moves to the manager. I included the FillMoves function that I created with all of the moves commented out, because they haven't been implemented yet and will cause a compile time error since they could not be found.

GetMove is used to retrieve a move from the manager. Rather than just returning that object I return a clone of the object. If you don't anywhere that you modify that object it will affect all other variables that point to that object. This is because classes are reference types. That means that the variable does not contain the data, it points to a location in memory that contains the data.

The AddMove method works in reverse. It takes a move and checks to see if it has been added. If it has not yet been added it is added to the move collection.

The next thing that I want to add is a method to the Avatar class that will take a string in the format above and return an Avatar object. Add the following method to the bottom of the Avatar class.

```
public static Avatar FromString(string description, ContentManager content)
{
    Avatar avatar = new Avatar();
    string[] parts = description.Split(',');

    avatar.name = parts[0];
    avatar.texture = content.Load<Texture2D>(@"AvatarImages\" + parts[0]);
    avatar.element = (AvatarElement)Enum.Parse(typeof(AvatarElement), parts[1]);
    avatar.costToBuy = int.Parse(parts[2]);
    avatar.level = int.Parse(parts[3]);
    avatar.attack = int.Parse(parts[4]);
    avatar.defense = int.Parse(parts[5]);
    avatar.speed = int.Parse(parts[6]);
    avatar.health = int.Parse(parts[7]);
    avatar.currentHealth = avatar.health;

    avatar.knownMoves = new Dictionary<string, IMove>();

    for (int i = 8; i < parts.Length; i++)
    {
        string[] moveParts = parts[i].Split(':');

        if (moveParts[0] != "None")
        {
            IMove move = MoveManager.GetMove(moveParts[0]);
            move.UnlockedAt = int.Parse(moveParts[1]);

            if (move.UnlockedAt <= avatar.Level)
                move.Unlock();

            avatar.knownMoves.Add(move.Name, move);
        }
    }

    return avatar;
}
```

The method accepts two parameters. It accepts a string that describes the avatar and a ContentManager to load the avatar's image. It first creates a new Avatar object using the private constructor that was added to the class. I then call the Split method on the description to break it up into its individual parts.

The next series of lines assign the value of the Avatar object using the parts. The texture is assigned by loading the image using the content manager that was passed in. Up until I assign the currentHealth member variable I use the Parse method of the individual items to get the associated values. For the element I had to specify the type of the enumeration that holds the elements and the value. The currentHealth member is assigned the value of the health member.

The next step is to create the Dictionary that will hold the moves that the avatar knows. Next up is a loop that will loop through the remaining parts of the string and get the moves for the avatar. The first step is to split the string into parts base on the colon. If the first part of the string is None then skip it. I then use the GetMove method of the MoveManager to get the move. I then use the second part to determine what level the move unlocks at. If the avatar's level is greater than or equal to that I unlock the move. Finally the move is added to the dictionary of moves for the avatar. The avatar is then returned to the calling method.

As you saw from the MoveManager I had created several moves for the demo that I did. Since moves implement the IMove interface most of the code is identical. The differences are in the constructors and the Clone methods. I'm going to implement two moves in this tutorial. For the other moves I suggest that you download the source code for them.

What I am going to implement are a basic attack move, Tackle, and a basic block move, Block. Let's start with Tackle. Right click the AvatarComponents folder, select Add and then Class. Name this new class Tackle. Here is the code for that class.

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

namespace Avatars.AvatarComponents
{
    public class Tackle : IMove
    {
        #region Field Region

        private string name;
        private Target target;
        private MoveType moveType;
        private MoveElement moveElement;
        private Status status;
        private bool unlocked;
        private int unlockedAt;
        private int duration;
        private int attack;
        private int defense;
        private int speed;
        private int health;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public Target Target
        {
            get { return target; }
        }

        public MoveType MoveType
        {
```

```
        get { return moveType; }
    }

    public MoveElement MoveElement
    {
        get { return moveElement; }
    }

    public Status Status
    {
        get { return status; }
    }

    public int UnlockedAt
    {
        get { return unlockedAt; }
        set { unlockedAt = value; }
    }

    public bool Unlocked
    {
        get { return unlocked; }
    }

    public int Duration
    {
        get { return duration; }
        set { duration = value; }
    }

    public int Attack
    {
        get { return attack; }
    }

    public int Defense
    {
        get { return defense; }
    }

    public int Speed
    {
        get { return speed; }
    }

    public int Health
    {
        get { return health; }
    }

    #endregion

    #region Constructor region

    public Tackle()
    {
        name = "Tackle";
        target = Target.Enemy;
        moveType = MoveType.Attack;
        moveElement = MoveElement.None;
        status = Status.Normal;
        duration = 1;
        unlocked = false;
        attack = MoveManager.Random.Next(0, 0);
        defense = MoveManager.Random.Next(0, 0);
        speed = MoveManager.Random.Next(0, 0);
```

```
            health = MoveManager.Random.Next(10, 15);
        }

        #endregion

        #region Method Region

        public void Unlock()
        {
            unlocked = true;
        }

        public object Clone()
        {
            Tackle tackle = new Tackle();
            tackle.unlocked = this.unlocked;
            return tackle;
        }

        #endregion
    }
}
```

There are member variables to expose each of the properties that must be implemented for the IMove interface. I will run over what they are briefly. Name is the name that will be displayed when the avatar uses this move. Target is what the move targets. MoveType is what kind of move it is. MoveElement is the element associated with the move. Status determines if there is a status change for the move. Unlocked is if the move is unlocked or not. Duration is how long the move lasts. Moves that have a duration of 1 take place that round. Moves with a duration greater than 1 last that many rouds. Attack is applied to the target's attack attribute. Defense, Speed and Health are applied to their respective attributes.

The constructors set the properties for the move. For tackle, the name is set to Tackle, the target is an enemy, the type is attack, the element is none, the status is normal, the duration is 1, as explained earlier that means it is applied immediately and initially the move is locked. Next I up I assign attack, defense, speed and health random values. In this case an attack damages an opponent so I generate a number between 10 and 14 because the upper limit is exclusive for this method of the Random class. The Clone method creates a new Tackle object and assigns it's unlocked value to the object returned.

Next up I'm going to add in the Block class. This move is a buff for the avatar and increase's its defense score for a few rounds. Right click the AvatarComponents folder, select Add and then Class. Name this new class Block. Here is the code for that class.

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

namespace Avatars.AvatarComponents
{
    public class Block : IMove
    {
        #region Field Region

        private string name;
        private Target target;
        private MoveType moveType;
        private MoveElement moveElement;
        private Status status;
        private bool unlocked;
        private int unlockedAt;
        private int duration;
        private int attack;
        private int defense;
        private int speed;
        private int health;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public Target Target
        {
            get { return target; }
        }

        public MoveType MoveType
        {
            get { return moveType; }
        }

        public MoveElement MoveElement
        {
            get { return moveElement; }
        }

        public Status Status
        {
            get { return status; }
        }

        public int UnlockedAt
        {
            get { return unlockedAt; }
            set { unlockedAt = value; }
        }

        public bool Unlocked
        {
            get { return unlocked; }
        }

        public int Duration
        {
            get { return duration; }
            set { duration = value; }
        }

        public int Attack
        {
            get { return attack; }
        }

        public int Defense
        {
            get { return defense; }
        }

        public int Speed
        {
            get { return speed; }
        }

        public int Health
        {
            get { return health; }
        }

        #endregion

        #region Constructor Region

        public Block()
        {
            name = "Block";
            target = Target.Self;
            moveType = MoveType.Buff;
            moveElement = MoveElement.None;
            status = Status.Normal;
            unlocked = false;
            duration = 5;
            attack = MoveManager.Random.Next(0, 0);
            defense = MoveManager.Random.Next(2, 6);
            speed = MoveManager.Random.Next(0, 0);
            health = MoveManager.Random.Next(0, 0);
        }

        #endregion

        #region Method Region

        public void Unlock()
        {
            unlocked = true;
        }

        public object Clone()
        {
            Block block = new Block();
            block.unlocked = this.unlocked;
            return block;
        }

        #endregion
    }
}
```

Until you get to the constructor the code is identical to the Tackle move. Once you hit the constructor just a few of the properties change. The Name property changes to Block, the target to self, the type to buff, the duration to 5 and the random numbers generated. In this case Health is 0 and defense is between 2 and 5, because 6 is excluded from that range. The next difference is not until the Clone method where I created a Block object to return rather than a Tackle object.

At the time that I implemented this I was on a time restriction that I needed to finish the game for submission to the challenge I was building for. In hind sight this could have been better designed, similar to how I created avatars using strings. It would be idea to modify this so that you have perhaps one class that all moves share rather than a class for each move. I will leave that as an exercise, unless I get requests to demonstrate how you could do that.

So now I'm going to add some avatars to the game. I apologize for the lack of imagination when it comes to their names in advance. The first step will be to right click the Avatars project, select Add and then Folder. Name this new folder Data. Right click this new Data folder, select Add and then New Item. From the list of templates choose Textfile and name it Avatars.csv. Next, select the Avatars.csv file in the solution. In the properties window for the file change the Copy to Output Directory property to Copy always. This will ensure that the file is copied to the output directory for testing. Copy and paste the following lines into the Avatars.csv file.

```
Dark,Dark,100,1,9,12,10,50,Tackle:1,Block:1
Earth,Earth,100,1,10,10,9,60,Tackle:1,Block:1
Fire,Fire,100,1,12,8,10,50,Tackle:1,Block:1
Light,Light,100,1,12,9,10,50,Tackle:1,Block:1
Water,Water,100,1,9,12,10,50,Tackle:1,Block:1
Wind,Wind,100,1,10,10,12,50,Tackle:1,Block:1
```

All of the avatars start out basically the same. Each of them costs 100 gold to purchase and their levels start at 1.The next few properties are their attack, defense, speed and health. I tried to make each of them slightly different than the others. For example, in my description I said that earth avatars are strong but slow. For that reason I reduces their speed to 9 but increased their health to 60. I would recommend in your own games that you play test the different avatars to make sure that you don't end up with a “broken” avatar that your players will always use because they can't be beaten by other avatars.

The next step will be load these into the game. I did that by adding in a class to manage the avatars in the game. To this class I added a FromFile method that read the file line by line and created the avatars from each line. Let's add that to the game now. Right click the AvatarComponents folder, select Add and then Class. Name this new class AvatarManager. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Microsoft.Xna.Framework.Content;

namespace Avatars.AvatarComponents
{
    public static class AvatarManager
    {
        #region Field Region

        private static Dictionary<string, Avatar> avatarList = new Dictionary<string,
Avatar>();

        #endregion

        #region Property Region

        public static Dictionary<string, Avatar> AvatarList
        {
            get { return avatarList; }
        }

        #endregion

        #region Constructor Region

        #endregion

        #region Method Region
        public static void AddAvatar(string name, Avatar avatar)
        {
            if (!avatarList.ContainsKey(name))
                avatarList.Add(name, avatar);
        }

        public static Avatar GetAvatar(string name)
        {
            if (avatarList.ContainsKey(name))
                return (Avatar)avatarList[name].Clone();

            return null;
        }

        public static void FromFile(string fileName, ContentManager content)
        {
            using (Stream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read))
            {
                try
                {
                    using (TextReader reader = new StreamReader(stream))
                    {
                        try
                        {
                            string lineIn = "";

                            do
                            {
                                lineIn = reader.ReadLine();
                                if (lineIn != null)
                                {
                                    Avatar avatar = Avatar.FromString(lineIn, content);
                                    if (!avatarList.ContainsKey(avatar.Name))
                                        avatarList.Add(avatar.Name, avatar);
                                }
                            } while (lineIn != null);
                        }
                        catch
                        {
                        }
                        finally
                        {
                            if (reader != null)
                                reader.Close();
                        }
                    }
                }
                catch
                {
                }
                finally
                {
                    if (stream != null)
                        stream.Close();
                }
            }
        }

        #endregion
    }
}
```

This manager is similar to all other manager classes. It maintains a dictionary for the objects to be managed and has a mechanism to add and retrieve objects. What is different about this one is that it has a FromFile method that will open the file, read the avatars from the file and add them to the dictionary. The FromFile method accepts the name of the file to be read and ContentManager to load the images for the avatars.

In order to read from a file you need a Stream object. In this case I created a FileStream as my content was stored on disk. There are other types of streams that you can create in C, such as a NetworkStream that could be used to pull the file from a server if you were to implement your game on Android or iOS. This would prevent cheating where players modify your content locally. Once the stream is open I create a TextReader using the stream to read the text file. I will be loading each line into a variable, lineIn. Next up is a do-while loop that loops for as long as there is data in the file. Each pass through I try to read the next line of the file using ReadLine. If that is not null then data was read in. I then create an Avatar object using the FromString method that I showed earlier in the tutorial. Then if there is not already an avatar with that name in the collection I call the Add method to add the avatar to the collection.

There is one last piece of the puzzle missing. There are no images for the for the avatars so when you try to read them in the game will crash. For my demo I grabbed the six images from the YuGiOh card game. They will be good enough for this tutorial as well. You can find my images at this link.

Once you've downloaded and extracted the files open the MonoGame pipeline again. Select the Content folder then click the New Folder icon in the tool bar. Name this new folder AvatarImages. Now select the AvatarImages folder then click the Add Existing Item button in the ribbon. Add the images, dark.PNG, earth.PNG, fire.PNG, light.PNG, water.PNG and wind.PNG. Once the images have been added save the project, click Build from the menu and select Build to build the content project then close the window.

Now we can load the avatars into the game. First, in the MoveManager update the FillMoves folder to create the Tackle and Block moves. If you added the other moves you can create them as well. Now, open the Game1 class and update the LoadContent method to the following.

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

    AvatarComponents.MoveManager.FillMoves();
    AvatarComponents.AvatarManager.FromFile(@".\Data\avatars.csv", Content);
}
```

What this does is create the moves for the avatars and then loads the avatars from the file that we created. At this point you will be able to build and run the game with out problems.

I'm going to end the tutorial here as it is a lot to digest in one sitting. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 11

Battling Avatars

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.

So, we have the player, characters and avatars. The next step will be battling avatars. The first step will be to add to the player class the avatars that the player has captured/learned. I say captured as well as learned because I will be including a second player component that works more like Pokemon than my demo.

To get started open the Player class in the PlayerComponents folder. I updated this class to make the private member variables protected member variables. I also added in a field and accessor methods. Update that class to the following.

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

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

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

        protected Vector2 position;
        protected Dictionary<string, Avatar> avatars = new Dictionary<string, Avatar>();
        private string currentAvatar;

        #endregion

        #region Property Region

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

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

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

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

        #endregion

        #region Constructor Region

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

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

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

        #endregion

        #region Method Region

        public virtual void AddAvatar(string avatarName, Avatar avatar)
        {
            if (!avatars.ContainsKey(avatarName))
                avatars.Add(avatarName, avatar);
        }

        public virtual Avatar GetAvatar(string avatarName)
        {
            if (avatars.ContainsKey(avatarName))
```

```
                return avatars[avatarName];

            return null;
        }

        public virtual void SetAvatar(string avatarName)
        {
            if (avatars.ContainsKey(avatarName))
                currentAvatar = avatarName;
            else
                throw new IndexOutOfRangeException();
        }

        public void SavePlayer()
        {
        }

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

            return player;
        }

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

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

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

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

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

        #endregion
    }
}
```

The new field that I added is a Dictionary<string, Avatar> that holds all the avatars that the player owns. I also added a field currentAvatar that holds which is the avatar that will be used when combat first starts. I also added a virtual property that returns the avatar with that name. I then added a virtual method AddAvatar that checks if an avatar with that name already exist and if it doesn't adds it to the collection of avatars. The GetAvatar virtual method checks to see if an avatar with that name exists in the collection and if it does it returns it. Otherwise it returns null. I also added a SetAvatar method that will set the currentAvatar member to the value passed in if the collection of avatars contains the key passed in. If it does not exist in the collection I thrown an exception.

Why did I make some of the members virtual? I'm taking a slightly different approach with have different types of player components. Instead of using interfaces I'm using inheritance. This is really helpful to learn of you don't know it so that is why I included it in this tutorial. Any of the virtual members can be overridden in the class that inherits from the player with the same name and parameters but behave differently.

Another note, why do I throw exceptions instead of letting the game crash and handle the exception there? The first is throwing a specific exception gives me an idea of what the problem is and how to stop. The second reason is that eventually I start catching the exceptions and handle them. This helps prevent the player from injecting values into the game to cheat as well as find logic errors.

The next thing that I want to add is a second player class, called PPlayer. This class deals with avatars more like the player in Pokemon because they are limited to the number of avatars they have at one time. Right click the PlayerComponents folder in your solution, select Add and then Class. Name the new class PPlayer. Here is the code for that class.

```
using Avatars.AvatarComponents;
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.PlayerComponents
{
    public class PPlayer : Player
    {
        public const int MaxAvatars = 6;

        private List<Avatar> battleAvatars = new List<Avatar>();
        private int currentAvatar;

        public override Avatar CurrentAvatar
        {
            get { return battleAvatars[currentAvatar]; }
        }

        public PPlayer(Game game, string name, bool gender, Texture2D texture)
            : base(game, name, gender, texture)
        {
        }

        public void SetCurrentAvatar(int index)
        {
            if (index < 0 || index > MaxAvatars)
                throw new IndexOutOfRangeException();

            currentAvatar = index;
        }

        public Avatar GetBattleAvatar(int index)
        {
            if (index < 0 || index > MaxAvatars)
                throw new IndexOutOfRangeException();

            return battleAvatars[index];
        }

        public void AddBattleAvatar(Avatar avatar)
        {
            if (battleAvatars.Count >= MaxAvatars - 1)
                throw new OverflowException();
            battleAvatars.Add(avatar);
        }

        public void RemoveBattleAvatar(int index)
        {
            if (index >= battleAvatars.Count)
                throw new IndexOutOfRangeException();

            battleAvatars.RemoveAt(index);
        }
    }
}
```

There is a constant in this class that describes the maximum number of avatars in a party and it is set to 6. This could be set to just about any value but to be consistent with other classes I went with 6. was pretty good for demonstration purposes. Next up is a List<Avatar> which will hold the avatars for the party and an integer variable that holds the currently selected avatar that will be used when a battle starts. There is then a property where I override the behavior of the CurrentAvatar property. Instead of using the string value in the other class I use the integer value in this class.

In order to inherit from a class you need to call one of its constructors. The parent class has a private constructor and a public constructor. Since the values need to be set I added in the parameters to the list of arguments for the constructor to the signature for the constructor and the call the parent using base.

After that there are some methods for manipulating the battle avatars. SetCurrentAvatar is used to assign which avatar is the default for starting combat. It checks to see if the value passed in is out of bounds. If it is it throws an exception, otherwise it sets the value. GetBattleAvatar is used to get the actual avatar object. It also checks to make sure the value is in range and if it is throw an exceptions. Otherwise return the avatar at that particular index.

AddBattleAvatar is used to add an avatar to the player's battle avatar list. It checks to make sure that there is room for the avatar. Since the index is zero based if the count is greater than or equal to the maximum number of avatars minus one there is no room for it so throw an exception. Otherwise it is find to return the avatar. The RemoveBattleAvatar method checks that the index is with in the range of allowed indexes and if it is throws an exception. Otherwise it removes the avatar at the specified index.

I want to make an update to the AvatarManager method FromFile before moving onto the next step. What I want to do is change it so that the dictionary key is always in lower case rather than mixed case. To do that I used the method .ToLowerInvariant(). You could probably get away with just ToLower though. I included the Invariant part because I deal with localization on a daily basis and you may want to update your game to support multiple languages. Here is the update for that method.

```
public static void FromFile(string fileName, ContentManager content)
{
    using (Stream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read))
    {
        try
        {
            using (TextReader reader = new StreamReader(stream))
            {
                try
                {
                    string lineIn = "";
                    do
                    {
                        lineIn = reader.ReadLine();
                        if (lineIn != null)
                        {
                            Avatar avatar = Avatar.FromString(lineIn, content);
                            if (!avatarList.ContainsKey(avatar.Name.ToLowerInvariant()))
                                avatarList.Add(avatar.Name.ToLowerInvariant(), avatar);
                        }
                    } while (lineIn != null);
                }
                catch
                {
                }
                finally
                {
                    if (reader != null)
                        reader.Close();
                }
            }
        }
        catch
        {
        }
        finally
        {
            if (stream != null)
                stream.Close();
        }
    }
}
```

I'm now going to update the Character so that when a string is passed to the FromString method it will used the 5th value in the string to assign the current battleAvatar. Here is the update.

```
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];

    character.battleAvatar = AvatarManager.GetAvatar(parts[4].ToLowerInvariant());

    return character;
}
```

The change is the second last line of code. I call the GetAvatar method of the AvatarManager class passing in the 5th part of the string in lower case. I'm now going to make a similar change to the PCharacter class. It will add upto six avatars to the character's list of avatars. Update that method as follows.

```
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];

    for (int i = 4; i < 10 && i < parts.Length; i++)
        character.avatars[i - 4] = AvatarManager.GetAvatar(parts[i].ToLowerInvariant());

    return character;
}
```

What the code does is similar to what I did when building avatars. For avatars I would read their moves until there were no moves left in the string.

Next step is to assign the player and the characters avatars. I will do that in the SetUpNewGame method of the GamePlayState class. What I did was move creating the player form LoadContent to SetUpNewGame. Please update those two methods to the following.

```
protected override void LoadContent()
{
}

public void SetUpNewGame()
{
    Texture2D spriteSheet = content.Load<Texture2D>(@"PlayerSprites\maleplayer");

    player = new Player(GameRef, "Wesley", false, spriteSheet);
    player.AddAvatar("fire", AvatarManager.GetAvatar("fire"));
    player.SetAvatar("fire");

    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();

    ConversationManager.CreateConversations(GameRef);

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

    teacherOne.SetConversation("LanceHello");
    teacherTwo.SetConversation("MarissaHello");

    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 LoadContent now does nothing and is there in case it is need in the future. I moved creating the player to the top of the method. After creating the player I add a "fire" avatar to their avatar collection and set their current avatar to "fire". For the first character that is a Character I added another value to the string, water, that will set their avatar to be a "water" avatar. For the other character, that is a PCharacter, I added a "wind" and "earth" avatar to the character's collection.

The last thing that I'm going to add is the base battle state and show how to switch to it from the game play state. For that you will need two graphics. One is a border that holds the health bar and the actual health bar. You can download those images using this link. Once they are ready open the MonoGame content builder. Select the Misc folder and then click the Add Existing Item and browse to the avatarborder.png file and add id to the project. Repeat the process for the avatarhealth.png. Now save and build the project.

Next, right click the GameStates folder, select Add and then Class Name this new class BattleState. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Avatars.AvatarComponents;
using Avatars.ConversationComponents;
using Avatars.Components;

namespace Avatars.GameStates
{
    public interface IBattleState
    {
        void SetAvatars(Avatar player, Avatar enemy);
        void StartBattle();
    }

    public class BattleState : BaseGameState, IBattleState
    {
        #region Field Region

        private Avatar player;
        private Avatar enemy;
        private GameScene combatScene;
        private Texture2D combatBackground;
        private Rectangle playerRect;
        private Rectangle enemyRect;
        private Rectangle playerBorderRect;
        private Rectangle enemyBorderRect;
        private Rectangle playerMiniRect;
        private Rectangle enemyMiniRect;
        private Rectangle playerHealthRect;
        private Rectangle enemyHealthRect;
        private Rectangle healthSourceRect;
        private Vector2 playerName;
        private Vector2 enemyName;
        private float playerHealth;
        private float enemyHealth;
        private Texture2D avatarBorder;
        private Texture2D avatarHealth;
        private SpriteFont font;
        private SpriteFont avatarFont;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public BattleState(Game game)
            : base(game)
        {
            playerRect = new Rectangle(10, 90, 300, 300);
            enemyRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 10, 300, 300);

            playerBorderRect = new Rectangle(10, 10, 300, 75);
            enemyBorderRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 320, 300,
75);

            healthSourceRect = new Rectangle(10, 50, 290, 20);
            playerHealthRect = new Rectangle(playerBorderRect.X + 12, playerBorderRect.Y +
52, 286, 16);
            enemyHealthRect = new Rectangle(enemyBorderRect.X + 12, enemyBorderRect.Y + 52,
286, 16);

            playerMiniRect = new Rectangle(playerBorderRect.X + 11, playerBorderRect.Y + 11,
28, 28);
            enemyMiniRect = new Rectangle(enemyBorderRect.X + 11, enemyBorderRect.Y + 11,
28, 28);

            playerName = new Vector2(playerBorderRect.X + 55, playerBorderRect.Y + 5);
            enemyName = new Vector2(enemyBorderRect.X + 55, enemyBorderRect.Y + 5);
        }

        #endregion

        #region Method Region

        public override void Initialize()
        {
            base.Initialize();
        }
        protected override void LoadContent()
        {
            combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

            avatarFont = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");
            avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
            avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

            font = GameRef.Content.Load<SpriteFont>(@"Fonts\gamefont");

            combatScene = new GameScene(GameRef, "", new List<SceneOption>());

            base.LoadContent();
        }

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

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

            GameRef.SpriteBatch.Begin();

            combatScene.Draw(gameTime, GameRef.SpriteBatch, combatBackground, font);

            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);

            playerHealth = (float)player.CurrentHealth / (float)player.GetHealth();
            MathHelper.Clamp(playerHealth, 0f, 1f);
            playerHealthRect.Width = (int)(playerHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, playerHealthRect, healthSourceRect,
Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, enemyBorderRect, Color.White);

            enemyHealth = (float)enemy.CurrentHealth / (float)enemy.GetHealth();
            MathHelper.Clamp(enemyHealth, 0f, 1f);
            enemyHealthRect.Width = (int)(enemyHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, enemyHealthRect, healthSourceRect,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, player.Name, playerName,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, enemy.Name, enemyName, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerMiniRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyMiniRect, Color.White);

            GameRef.SpriteBatch.End();
        }

        public void SetAvatars(Avatar player, Avatar enemy)
        {
            this.player = player;
            this.enemy = enemy;

            player.StartCombat();
            enemy.StartCombat();
        }
        public void StartBattle()
        {
            player.StartCombat();
            enemy.StartCombat();
            playerHealth = 1f;
            enemyHealth = 1f;
        }

        #endregion
    }
}
```

First off, I included an interface for the state IBattleState. This is the contract that is provided to other states to interface with this state. The members that in includes are SetAvatars and StartBattle. The first is used to place the avatars on the screen where as the second is used to start a battle between the two avatars.

Most of the member variables in this class are for position elements and holding the image for the elements. There are also an Avatar field for the player and the enemy. There is also a GameScene that will be used by the player to select what move they want the avatar to use that round.

All the constructor does is position items based on the screen rectangle. The elements that are being places are the images for the avatars, the position of their bar that holds the health bar and the actual health bar.

In the LoadContent method I load in the background, two fonts that were already added to the project and the two textures that were just added. I also create a basic combat scene using the conversation components. This will be used to display options to the player and other messages during combat. Currently the Update screen just calls the base update method in order to update the components.

The Draw method first starts drawing the sprite batch. Since it is at the back the scene is drawn first. After that I draw the player's avatar and then the enemy's avatar. Then I draw the borders and the avatar's current health. To draw the current health I get the percentage of the avatar's health is left. I then clamp it between 0 and 1. To determine the width out I multiple the fraction of the remain health by the width of the texture. After drawing the bars I write the name of the avatar into the bar. Finally I then draw the avatar images.

The SetAvatars method is used to set the avatars. After assigning the local member variables to the corresponding parameters I call StartCombat to start combat between the two. In the StartBattle method I also call the StartCombat method and set two member variables to 1f.

The last thing that I'm going to cover is moving from the game play state to this new state. First, we need to update the Game1 class to create a base battle state that will be reused for all battles in the game. The change was I add a member variable for the state, a property to expose it and create it in the constructor. Here is the updated Game1 class.

```
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;
        IConversationState conversationState;
        IBattleState battleState;

        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 IBattleState BattleState
        {
            get { return battleState; }
        }

        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);
            conversationState = new ConversationState(this);
            battleState = new BattleState(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);

            AvatarComponents.MoveManager.FillMoves();
            AvatarComponents.AvatarManager.FromFile(@".\Data\avatars.csv", Content);
        }

        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);
        }
    }
}
```

Next up I'm going to update the Update method in the GamePlayScreen to switch states to the battle state if the player is close to the character and presses B. Change that method 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);

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        foreach (string s in map.Characters.Keys)
        {
            ICharacter c = CharacterManager.Instance.GetCharacter(s);
            float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

            if (Math.Abs(distance) < 72f)
            {
                IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                manager.PushState(
                    (ConversationState)conversationState,
                    PlayerIndexInControl);

                conversationState.SetConversation(player, c);
                conversationState.StartConversation();
            }
        }
    }

    if (Xin.CheckKeyReleased(Keys.B))
    {
        foreach (string s in map.Characters.Keys)
        {
            ICharacter c = CharacterManager.Instance.GetCharacter(s);
            float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

            if (Math.Abs(distance) < 72f)
            {
                GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                manager.PushState(
                    (BattleState)GameRef.BattleState,
                    PlayerIndexInControl);
            }
        }
    }
    base.Update(gameTime);
}
```

What this new code does is check to see if the B key was released. Then like when I checked for starting a conversation with a character I cycle through all of the characters. I get the character and get its distance to the player. If the distance is less than 72 I call the SetAvatars method of the battle state. I then push the battle state onto the stack.

You can now build and run the game. If you move the player close to either of the characters that were added you can press the B key and the game will switch to the battle state. Once you're in the battle state you're stuck there though. For now I'm going to add a quick escape from the battle stated. Change the Update method of the BattleState class to the following.

```
public override void Update(GameTime gameTime)
{
    PlayerIndex index = PlayerIndex.One;

    if (Xin.CheckKeyReleased(Keys.P))
        manager.PopState();

    base.Update(gameTime);
}
```

That is going to wrap up this tutorial. In the next tutorial I will get into the two avatars battling each other. I will also be working on preparing the game editor that I used for the demo I made so that you can play around with the framework up until now and see how all the pieces tie together.

I'm going to end the tutorial here as it is a lot to digest in one sitting. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 12

Battling Avatars Continued

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.

In this tutorial I'm going to pick up where I left off in the last tutorial in battling avatars. In this tutorial I will add in the next steps to actually battle the two avatars, such as selecting moves and having them applied to the target. I won't be covering the player and opponent having multiple avatars and will focus and just a single avatar. I will write a separate tutorial on how to add that functionality.

In the last tutorial I had added a game state called BattleState. This scene will display the available moves and allow the player to choose the move that they want. The options for this will need to be set each time the SetAvatars is called. Update that method to the following.

```
public void SetAvatars(Avatar player, Avatar enemy)
{
    this.player = player;
    this.enemy = enemy;

    player.StartCombat();
    enemy.StartCombat();

    List<SceneOption> moves = new List<SceneOption>();

    if (combatScene == null)
        LoadContent();

    foreach (string s in player.KnownMoves.Keys)
    {
        SceneOption option = new SceneOption(s, s, new SceneAction());
        moves.Add(option);
    }

    combatScene.Options = moves;
}
```

The change from last time is I built a List<SceneOption> that holds the moves for the avatar. I then check if the combatScene member variable is null and if it is I call LoadContent to create it. I had to do that because since I did not add the game component to the list of components in the game the LoadContent method is not automatically called.

In a foreach loop I loop over all of the keys in the KnownMoves dictionary. Inside I create a new scene options and add it to the list of moves. Before exiting the method I assign the scene options to the list that I just created.

I have updated the LoadContent method to handle the problem where it is not called automatically. All I did was have an if statement to make sure the combatScene is null before trying to load. Update the LoadContent method to the following.

```
protected override void LoadContent()
{
    if (combatScene == null)
    {
        combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

        avatarFont = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");
        avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
        avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

        font = GameRef.Content.Load<SpriteFont>(@"Fonts\gamefont");

        combatScene = new GameScene(GameRef, "", new List<SceneOption>());
    }

    base.LoadContent();
}
```

Next would be to wire the scene to perform the action the player selects and apply it to the enemy. First, I want to add in two game states. One state will be displayed when the battle is over. The other will be displayed while the moves of both avatars are being resolved for the current turn.

First, I am going to add the battle over state. Right click the GameScenes folder, select Add and then Class. Name the class BattleOverState. Here is the code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Avatars.AvatarComponents;
using Avatars.GameStates;
using Avatars.Components;

namespace Avatars.GameStates
{
    public interface IBattleOverState
    {
        void SetAvatars(Avatar player, Avatar enemy);
    }

    public class BattleOverState : BaseGameState, IBattleOverState
    {
        #region Field Region
        private Avatar player;
        private Avatar enemy;
        private Texture2D combatBackground;
        private Rectangle playerRect;
        private Rectangle enemyRect;
        private Rectangle playerBorderRect;
        private Rectangle enemyBorderRect;
        private Rectangle playerMiniRect;
        private Rectangle enemyMiniRect;
        private Rectangle playerHealthRect;
        private Rectangle enemyHealthRect;
        private Rectangle healthSourceRect;
        private Vector2 playerName;
        private Vector2 enemyName;
        private float playerHealth;
        private float enemyHealth;
        private Texture2D avatarBorder;
        private Texture2D avatarHealth;
        private SpriteFont avatarFont;
        private SpriteFont font;
        private string[] battleState;
        private Vector2 battlePosition;
        private bool levelUp;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public BattleOverState(Game game)
            : base(game)
        {
            battleState = new string[3];

            battleState[0] = "The battle was won!";
            battleState[1] = " gained ";
            battleState[2] = "Continue";

            battlePosition = new Vector2(25, 475);

            playerRect = new Rectangle(10, 90, 300, 300);
            enemyRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 10, 300, 300);

            playerBorderRect = new Rectangle(10, 10, 300, 75);
            enemyBorderRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 320, 300,
75);

            healthSourceRect = new Rectangle(10, 50, 290, 20);
            playerHealthRect = new Rectangle(playerBorderRect.X + 12, playerBorderRect.Y +
52, 286, 16);
            enemyHealthRect = new Rectangle(enemyBorderRect.X + 12, enemyBorderRect.Y + 52,
286, 16);

            playerMiniRect = new Rectangle(playerBorderRect.X + 11, playerBorderRect.Y + 11,
28, 28);
            enemyMiniRect = new Rectangle(enemyBorderRect.X + 11, enemyBorderRect.Y + 11,
28, 28);

            playerName = new Vector2(playerBorderRect.X + 55, playerBorderRect.Y + 5);
            enemyName = new Vector2(enemyBorderRect.X + 55, enemyBorderRect.Y + 5);
        }

        #endregion
        #region Method Region

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

        protected override void LoadContent()
        {
            combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

            avatarFont = GameRef.Content.Load<SpriteFont>(@"Fonts\GameFont");
            avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
            avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

            font = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            PlayerIndex index = PlayerIndex.One;

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                if (levelUp)
                {
                    this.Visible = true;
                }
                else
                {
                    manager.PopState();
                    manager.PopState();
                }
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            Vector2 position = battlePosition;

            base.Draw(gameTime);

            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(combatBackground, Vector2.Zero, Color.White);

            for (int i = 0; i < 2; i++)
            {
                GameRef.SpriteBatch.DrawString(font, battleState[i], position, Color.Black);
                position.Y += avatarFont.LineSpacing;
            }

            GameRef.SpriteBatch.DrawString(font, battleState[2], position, Color.Red);

            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);

            playerHealth = (float)player.CurrentHealth / (float)player.GetHealth();
            MathHelper.Clamp(playerHealth, 0f, 1f);
            playerHealthRect.Width = (int)(playerHealth * 286);
            GameRef.SpriteBatch.Draw(avatarHealth, playerHealthRect, healthSourceRect,
Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, enemyBorderRect, Color.White);

            enemyHealth = (float)enemy.CurrentHealth / (float)enemy.GetHealth();
            MathHelper.Clamp(enemyHealth, 0f, 1f);
            enemyHealthRect.Width = (int)(enemyHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, enemyHealthRect, healthSourceRect,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, player.Name, playerName,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, enemy.Name, enemyName, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerMiniRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyMiniRect, Color.White);

            GameRef.SpriteBatch.End();
        }

        public void SetAvatars(Avatar player, Avatar enemy)
        {
            levelUp = false;

            long expGained = 0;

            this.player = player;
            this.enemy = enemy;

            if (player.Alive)
            {
                expGained = player.WinBattle(enemy);
                battleState[0] = player.Name + " has won the battle!";
                battleState[1] = player.Name + " has gained " + expGained + " experience";

                if (player.CheckLevelUp())
                {
                    battleState[1] += " and gained a level!";

                    foreach (string s in player.KnownMoves.Keys)
                    {
                        if (player.KnownMoves[s].Unlocked == false && player.Level >=
player.KnownMoves[s].UnlockedAt)
                        {
                            player.KnownMoves[s].Unlock();
                            battleState[1] += " " + s + " was unlocked!";
                        }
                    }

                    levelUp = true;
                }
                else
                {
                    battleState[1] += ".";
                }
            }
            else
            {
                expGained = player.LoseBattle(enemy);

                battleState[0] = player.Name + " has lost the battle.";
                battleState[1] = player.Name + " has gained " + expGained + " experience";

                if (player.CheckLevelUp())
                {
                    battleState[1] += " and gained a level!";

                    foreach (string s in player.KnownMoves.Keys)
                    {
                        if (player.KnownMoves[s].Unlocked == false && player.Level >=
player.KnownMoves[s].UnlockedAt)
                        {
                            player.KnownMoves[s].Unlock();
                            battleState[1] += " " + s + " was unlocked!";
                        }
                    }

                    levelUp = true;
                }
                else
                {
                    battleState[1] += ".";
                }
            }
        }

        #endregion
    }
}
```

I included an interface for this class, IBattleOverState, that includes one method that must be implemented SetAvatars. This method is used just like the one in IBattleState for setting the battle avatars.

Like the BattleState this class has a lot of member variables for drawing and positioning graphical elements. There are member variables for both avatars as well. I included an array of strings called battleState. These strings are used to build and display the results of the battle to the player. I also included a member variable for positioning this element on the screen. The last member variable is a bool that measures if the avatar has leveled up or not.

The constructor just initializes member variables. I also create an array to hold some of the predefined strings and assign them values.

This time I didn't wrap loading the content into an if statement. The reason why is in the previous game state I was initializing values right after calling SetAvatar. In this case the load does not need to be done automatically and is deferred until after all creation has finished.

In the Update method I have the PlayerIndex variable that you have seen through out all of the game states so far. Next is an if statement where I check if the space key or enter key have been released. There is then an if statement that checks if the levelUp member variable is true or not. If it is I just set the Visible property of the game state to true. Later on in the tutorial I will be adding a level up state. If it was false I call PopState twice. The reason will be that multiple states will need to be popped of the stack to get back to the game play state.

Like in the other states the Draw method just positions all of the elements. What is different is that I also position the text displaying the battle state. First, I loop over all of the elements for the state and draw them at the desired position. I then update the Y position by the LineSpacing property for the font to move onto the next line.

In the SetAvatars method I determine what needs to be rendered by the scene. I check the Alive property of the player's avatar to determine if that avatar won the battle. If it did I call the WinBattle method passing in the enemy after to calculate how much experience the avatar gained for defeating this avatar. I then update the battle strings with these values. I next call the CheckLevelUp method on the player avatar to check if the avatar has levelled up or not. If it has I update the string being displayed. I then loop through all of the known moves and unlock any moves that unlock at the new level. I also append text to the display that the move was unlocked. Finally I set the levelUp member variable to true. If it did not level up I just append a period.

If the player's avatar lost I call the LostBattle method to assign it the experience it learned during the battle. I then constructor the strings that will display that the battle was lost. I also go over the same process of checking if the avatar levelled up or not.

Next up I want to add in the damage state. This state will resolve the selected moves for the avatars. Right click the GameStates folder, select Add and then Class. Name this new class DamageState. Here is the code for that class.

```
using Avatars.AvatarComponents;
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.GameStates
{
    public enum CurrentTurn
    {
        Players, Enemies
    }

    public interface IDamageState
    {
        void SetAvatars(Avatar player, Avatar enemy);
        void SetMoves(IMove playerMove, IMove enemyMove);
        void Start();
    }

    public class DamageState : BaseGameState, IDamageState
    {
        #region Field Region

        private CurrentTurn turn;
        private Texture2D combatBackground;
        private SpriteFont avatarFont;
        private SpriteFont font;
        private Rectangle playerRect;
        private Rectangle enemyRect;
        private TimeSpan cTimer;
        private TimeSpan dTimer;
        private Avatar player;
        private Avatar enemy;
        private IMove playerMove;
        private IMove enemyMove;
        private bool first;
        private Rectangle playerBorderRect;
        private Rectangle enemyBorderRect;
        private Rectangle playerMiniRect;
        private Rectangle enemyMiniRect;
        private Rectangle playerHealthRect;
        private Rectangle enemyHealthRect;
        private Rectangle healthSourceRect;
        private float playerHealth;
        private float enemyHealth;
        private Texture2D avatarBorder;
        private Texture2D avatarHealth;
        private Vector2 playerName;
        private Vector2 enemyName;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public DamageState(Game game)
            : base(game)
        {
            playerRect = new Rectangle(10, 90, 300, 300);
            enemyRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 10, 300, 300);

            playerBorderRect = new Rectangle(10, 10, 300, 75);
            enemyBorderRect = new Rectangle(Game1.ScreenRectangle.Width - 310, 320, 300,
75);

            healthSourceRect = new Rectangle(10, 50, 290, 20);
            playerHealthRect = new Rectangle(playerBorderRect.X + 12, playerBorderRect.Y +
52, 286, 16);
            enemyHealthRect = new Rectangle(enemyBorderRect.X + 12, enemyBorderRect.Y + 52,
286, 16);

            playerMiniRect = new Rectangle(playerBorderRect.X + 11, playerBorderRect.Y + 11,
28, 28);
            enemyMiniRect = new Rectangle(enemyBorderRect.X + 11, enemyBorderRect.Y + 11,
28, 28);

            playerName = new Vector2(playerBorderRect.X + 55, playerBorderRect.Y + 5);
            enemyName = new Vector2(enemyBorderRect.X + 55, enemyBorderRect.Y + 5);
        }

        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            combatBackground = GameRef.Content.Load<Texture2D>(@"Scenes\scenebackground");

            avatarBorder = GameRef.Content.Load<Texture2D>(@"Misc\avatarborder");
            avatarHealth = GameRef.Content.Load<Texture2D>(@"Misc\avatarhealth");

            avatarFont = Game.Content.Load<SpriteFont>(@"Fonts\GameFont");
            font = Game.Content.Load<SpriteFont>(@"Fonts\scenefont");

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            PlayerIndex index;
            if ((cTimer > TimeSpan.FromSeconds(4) || !enemy.Alive || !player.Alive) &&
dTimer > TimeSpan.FromSeconds(3))
            {
                if (!enemy.Alive || !player.Alive)
                {
                    manager.PopState();
                    manager.PushState((BattleOverState)GameRef.BattleOverState,
PlayerIndex.One);
                    GameRef.BattleOverState.SetAvatars(player, enemy);
                }
                else
                {
                    manager.PopState();
                }
            }
            else if (cTimer > TimeSpan.FromSeconds(2) && first && enemy.Alive &&
player.Alive)
            {
                first = false;
                dTimer = TimeSpan.Zero;
                if (turn == CurrentTurn.Players)
                {
                    turn = CurrentTurn.Enemies;
                    enemy.ResoleveMove(enemyMove, player);
                }
                else
                {
                    turn = CurrentTurn.Players;
                    player.ResoleveMove(playerMove, enemy);
                }
            }
            else if (cTimer == TimeSpan.Zero)
            {
                dTimer = TimeSpan.Zero;
                if (turn == CurrentTurn.Players)
                {
                    player.ResoleveMove(playerMove, enemy);
                }
                else
                {
                    enemy.ResoleveMove(enemyMove, player);
                }
            }

            cTimer += gameTime.ElapsedGameTime;
            dTimer += gameTime.ElapsedGameTime;

            base.Update(gameTime);
        }

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

            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(combatBackground, Vector2.Zero, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            Vector2 location = new Vector2(25, 475);

            if (turn == CurrentTurn.Players)
            {
                GameRef.SpriteBatch.DrawString(font, player.Name + " uses " +
playerMove.Name + ".", location, Color.Black);

                if (playerMove.Target == Target.Enemy && playerMove.MoveType ==
MoveType.Attack)
                {
                    location.Y += avatarFont.LineSpacing;

                    if (Avatar.GetMoveModifier(playerMove.MoveElement, enemy.Element) < 1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is not very effective.",
location, Color.Black);
                    }
                    else if (Avatar.GetMoveModifier(playerMove.MoveElement, enemy.Element) >
1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is super effective.",
location, Color.Black);
                    }
                }
            }
            else
            {
                GameRef.SpriteBatch.DrawString(font, "Enemy " + enemy.Name + " uses " +
enemyMove.Name + ".", location, Color.Black);

                if (enemyMove.Target == Target.Enemy && playerMove.MoveType ==
MoveType.Attack)
                {
                    location.Y += avatarFont.LineSpacing;

                    if (Avatar.GetMoveModifier(enemyMove.MoveElement, player.Element) < 1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is not very effective.",
location, Color.Black);
                    }
                    else if (Avatar.GetMoveModifier(enemyMove.MoveElement, player.Element) >
1f)
                    {
                        GameRef.SpriteBatch.DrawString(font, "It is super effective.",
location, Color.Black);
                    }
                }
            }

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);
            GameRef.SpriteBatch.Draw(player.Texture, playerRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyRect, Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, playerBorderRect, Color.White);

            playerHealth = (float)player.CurrentHealth / (float)player.GetHealth();
            MathHelper.Clamp(playerHealth, 0f, 1f);
            playerHealthRect.Width = (int)(playerHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, playerHealthRect, healthSourceRect,
Color.White);

            GameRef.SpriteBatch.Draw(avatarBorder, enemyBorderRect, Color.White);

            enemyHealth = (float)enemy.CurrentHealth / (float)enemy.GetHealth();
            MathHelper.Clamp(enemyHealth, 0f, 1f);
            enemyHealthRect.Width = (int)(enemyHealth * 286);

            GameRef.SpriteBatch.Draw(avatarHealth, enemyHealthRect, healthSourceRect,
Color.White);
```

```
            GameRef.SpriteBatch.DrawString(avatarFont, player.Name, playerName,
Color.White);
            GameRef.SpriteBatch.DrawString(avatarFont, enemy.Name, enemyName, Color.White);

            GameRef.SpriteBatch.Draw(player.Texture, playerMiniRect, Color.White);
            GameRef.SpriteBatch.Draw(enemy.Texture, enemyMiniRect, Color.White);

            GameRef.SpriteBatch.End();
        }

        public void SetAvatars(Avatar player, Avatar enemy)
        {
            this.player = player;
            this.enemy = enemy;

            if (player.GetSpeed() >= enemy.GetSpeed())
            {
                turn = CurrentTurn.Players;
            }
            else
            {
                turn = CurrentTurn.Enemies;
            }
        }

        public void SetMoves(IMove playerMove, IMove enemyMove)
        {
            this.playerMove = playerMove;
            this.enemyMove = enemyMove;
        }

        public void Start()
        {
            cTimer = TimeSpan.Zero;
            dTimer = TimeSpan.Zero;
            first = true;
        }

        #endregion
    }
}
```

There is first an enumeration CurrentTurn with values Player and Enemy. These values determine who's turn it is during this round of combat. There is then an interface IDamageState that defines the public members that can be called. They are SetAvatars which is the same as the other two methods, SetMoves which sets what moves each avatar is going to attempt to use and Start which starts the timers for this state.

Just like the other states there are a lot of member variables for positioning and drawing the visual elements. In hind sight it probably would have been better to make those variables protected in the BattleState class and inherit these two classes from that class instead of duplicating these variables in these other classes. There is also a member variable names turn that holds whose turn it is. The next new members are cTimer and dTimer. These are used to measure how much time has passed after a certain point. There are also IMove members that hold what move the avatars are going to use. Finally, first holds who goes first, the player or the enemy.

This constructor works like the other ones in that it just positions elements at certain points on the screen. The LoadContent method just loads the content for displaying the avatars and their properties.

The Update method is where the magic happens, so to speak. There is an if statement that checks a number of conditions. It checks if any of the following are true and if the time passed in dTimer is greater than 3. Those conditions are cTimer is greater than 4 seconds, the enemy avatar is not alive or the player avatar is not alive. Inside that if I check to see if either the player avatar or enemy avatar are not alive. If this is true I pop the current state off the stack, the damage state, and push the BattleOver state onto the stack. I then call the SetAvatars method passing in the player and enemy avatars. If they are both still alive I just pop this state off this stack which returns control back to the battle state.

There is then an if that check that cTimer is greater than 2 seconds, the first member variable is true and both avatars are in good health. In this case first means that if this is the first move resolved or the second move resolved. In this case I want to switch and resolve the second move. I set first to false. I then reset the duration of the dTimer memeber variable. I check the turn variable to see who's turn it is. If it is player I switch that member variable to be the enemies turn and ResolveMove to resolve the enemies move. Same in reverse for the else step.

The else if checks to see if cTimer is zero. That means that this is the first time that the Update method has been called. In that case I reset dTimer to 0 and call the ResolveMove method on whatever avatar's turn it is.

The last thing that I do is increment the two timer variables by adding in the ElapsedGameTime property of the gameTime parameter that is passed to Update. You will want to play with the duration of the timers to have them match what you are expecting. You could also play an animation when each avatar attacks. I will do just that in a future tutorial.

In the Draw method I draw the scene. Much of it will be familiar from the other two states. What is new is that I draw the out come of the last move resolved. First, I display what move the avatar has used. I then move to the next line and display if it wasn't effective or if it was super effective, based on the element of the move used and the element of the defending avatar.

The SetAvatars method assigns the member variables for the avatars to the values passed in. I then check if the player's speed is greater than or equal to the opponent's speed. If it is the first turn is the player's turn otherwise the enemy resolves its move first.

SetMoves just sets the moves to the values that are passed in. Start resets the two timers to zero and resets that first member variable to true so that both moves will be resolved.

The last thing to do is to is to have the Update method of the BattleState class drive the choices for the combat. Change the code of the Update method in BattleState to the following.

```
public override void Update(GameTime gameTime)
{
    PlayerIndex? index = PlayerIndex.One;

    if (Xin.CheckKeyReleased(Keys.P))
        manager.PopState();

    combatScene.Update(gameTime, index.Value);

    if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
    {
        manager.PushState((DamageState)GameRef.DamageState, index);
        GameRef.DamageState.SetAvatars(player, enemy);
        IMove enemyMove = null;

        do
        {
            int move = random.Next(0, enemy.KnownMoves.Count);
            int i = 0;

            foreach (string s in enemy.KnownMoves.Keys)
            {
                if (move == i)
                    enemyMove = (IMove)enemy.KnownMoves[s].Clone();
                i++;
            }

        } while (!enemyMove.Unlocked);


GameRef.DamageState.SetMoves((IMove)player.KnownMoves[combatScene.OptionText].Clone(),
enemyMove);
        GameRef.DamageState.Start();

        player.Update(gameTime);
        enemy.Update(gameTime);
    }

    Visible = true;

    base.Update(gameTime);
}
```

So, what is new that there is a check to see if the space bar or enter key have been released, meaning that the player has made their selection. If they have I push the damage state onto the stack of states. I then call the SetAvatars method passing in the two avatars. Next is a local variable of type IMove that will hold the move the enemy avatar will use. Follow that is a do while loop that generates a random number in the range of all known moves for the avatar. In a foreach loop I then iterate over the keys in the KnownMoves collection. If the selected move matches an variable that increments during each iteration of the loop I set enemyMove to be a clone of that move. I then check to see if that moves is unlocked or not. If it is not I repeat the process.

The next step is that I call the SetMoves method on the damage state to set the moves they are going to be applied. I also call the Start method to reset the timers for the state. Finally I call the Update method of the player and enemy avatars. This allows any effects that have a duration to count down and remove themselves. I also assign the Visible property of the state to true. This keeps this state visible while I draw the damage state.

You can now build and run the game. If you move the player next to one of the two avatars and press the B key you will be able to start a battle with the character's avatar and play through the battle.

I'm going to end the tutorial here as I like to keep the tutorials to a reasonable length so there is not a lot of new code to digest at once. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 13

Leveling Up

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.

What I am going to tackle first in this tutorial is to add the ability to level up avatars after a battle. There are two ways that you could handle an avatar levelling up. One ways is that you could increase the avatar's status automatically in code or you can give the player the ability to assign points to an avatar's stats. I personally like giving the player choices so I went with that route.

First, you will want to download the content that I used for this tutorial. You can download the content using this link A Summoner's Tale Content 13. First, let's add the background for the level up state. In the solution explorer expand the Content folder and then the GameScreens folder. Right click the GameScreens folder, select Add and then Existing Item. Add the levelup-menu.png to this folder. Now open the Content.mgcb by double clicking on it. Right click on the GameScreens folder select Add and then Existing Item. Add the levelup-menu.png that we just added to that folder. Hit <ctrl>+s to save the changes and then under the Build menu select Build to build the content.

The next thing that I want to do is add a new state for levelling up and avatar. Right click the GameStates folder, select Add and then Class. Name this class LevelUpState. Here is the code for that state.

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

namespace Avatars.GameStates
```

```
{
    public interface ILevelUpState
    {
        void SetAvatar(Avatar playerAvatar);
    }

    public class LevelUpState : BaseGameState, ILevelUpState
    {
        #region Field Region

        private Rectangle destination;
        private int points;
        private int selected;
        private SpriteFont font;
        private Avatar player;
        private Dictionary<string, int> attributes = new Dictionary<string, int>();
        private Dictionary<string, int> assignedTo = new Dictionary<string, int>();
        private Texture2D levelUpBackground;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public LevelUpState(Game game)
            : base(game)
        {
            attributes.Add("Attack", 0);
            attributes.Add("Defense", 0);
            attributes.Add("Speed", 0);
            attributes.Add("Health", 0);
            attributes.Add("Done", 0);

            foreach (string s in attributes.Keys)
                assignedTo.Add(s, 0);
        }

        #endregion

        #region Method Region

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

        protected override void LoadContent()
        {
            levelUpBackground = GameRef.Content.Load<Texture2D>(
                @"GameScreens\levelup-menu");

            font = GameRef.Content.Load<SpriteFont>(@"Fonts\scenefont");

            destination = new Rectangle(
                (Game1.ScreenRectangle.Width - levelUpBackground.Width) / 2,
                (Game1.ScreenRectangle.Height - levelUpBackground.Height) / 2,
                levelUpBackground.Width,
                levelUpBackground.Height);

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
```

```
PlayerIndex index = PlayerIndex.One;
int i = 0;
string attribute = "";

if (Xin.CheckKeyReleased(Keys.Down))
{
    selected++;

    if (selected >= attributes.Count)
        selected = attributes.Count - 1;
}
else if (Xin.CheckKeyReleased(Keys.Up))
{
    selected--;

    if (selected < 0)
        selected = 0;
}

if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
{
    if (selected == 4 && points == 0)
    {
        foreach (string s in assignedTo.Keys)
        {
            player.AssignPoint(s, assignedTo[s]);
        }

        manager.PopState();
        manager.PopState();
        manager.PopState();
        return;
    }
}

int increment = 1;

if (Xin.CheckKeyReleased(Keys.Right) && points > 0)
{
    foreach (string s in assignedTo.Keys)
    {
        if (s == "Done")
            return;

        if (i == selected)
        {
            attribute = s;
            break;
        }

        i++;
    }


    if (attribute == "Health")
        increment *= 5;

    points--;
    assignedTo[attribute] += increment;

    if (points == 0)
        selected = 4;
}
else if (Xin.CheckKeyReleased(Keys.Left) && points <= 3)
{
    foreach (string s in assignedTo.Keys)
```

```
                {
                    if (s == "Done")
                        return;

                    if (i == selected)
                    {
                        attribute = s;
                        break;
                    }

                    i++;
                }

                if (assignedTo[attribute] != attributes[attribute])
                {
                    if (attribute == "Health")
                        increment *= 5;

                    points++;
                    assignedTo[attribute] -= increment;
                }
            }

            base.Update(gameTime);
        }

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


            GameRef.SpriteBatch.Begin();
            GameRef.SpriteBatch.Draw(levelUpBackground, destination, Color.White);

            Vector2 textPosition = new Vector2(destination.X + 5, destination.Y + 5);

            GameRef.SpriteBatch.DrawString(font, player.Name, textPosition, Color.Black);
            textPosition.Y += font.LineSpacing * 2;

            int i = 0;

            foreach (string s in attributes.Keys)
            {
                Color tint = Color.Black;

                if (i == selected)
                    tint = Color.Red;

                if (s != "Done")
                {
                    GameRef.SpriteBatch.DrawString(font, s + ":", textPosition, tint);
                    textPosition.X += 125;

                    GameRef.SpriteBatch.DrawString(font, attributes[s].ToString(),
textPosition, tint);
                    textPosition.X += 40;

                    GameRef.SpriteBatch.DrawString(font, assignedTo[s].ToString(),
textPosition, tint);
                    textPosition.X = destination.X + 5;

                    textPosition.Y += font.LineSpacing;
                }
                else
                {
                    GameRef.SpriteBatch.DrawString(font, "Done", textPosition, tint);
                    textPosition.Y += font.LineSpacing * 2;
                }
                i++;
            }

            GameRef.SpriteBatch.DrawString(font, points.ToString() + " point left.",
textPosition, Color.Black);
            GameRef.SpriteBatch.End();
        }

        public void SetAvatar(Avatar playerAvatar)
        {
            player = playerAvatar;

            attributes["Attack"] = player.BaseAttack;
            attributes["Defense"] = player.BaseDefense;
            attributes["Speed"] = player.BaseSpeed;
            attributes["Health"] = player.BaseHealth;

            assignedTo["Attack"] = player.BaseAttack;
            assignedTo["Defense"] = player.BaseDefense;
            assignedTo["Speed"] = player.BaseSpeed;
            assignedTo["Health"] = player.BaseHealth;

            points = 3;
            selected = 0;
        }

        #endregion
    }
}
```

First, there is an interface that I added for the state like the other game states. It has a single method in it, SetAvatar, that is used to pass the avatar that is to be levelled up to the state. The class then inherits from the BaseGameState abstract class so it can be used by the state manager and implements the interface that was just defined above.

For field in the class there is a Rectangle that will hold the destination of the level up state on the screen. Next there is a integer that holds the number of points that are available to be assigned to the avatar. There is also a field selected that holds the current option selected in the game state for assigning points. Since the state renders text there is a SpriteFont field for drawing text. There is also an Avatar type field that will hold the avatar that we will be updating. There are then two dictionaries that hold the attributes that the avatar currently has, attributes, and the assigned point, assignedTo. The last field that I included in this class is a field to hold the image for the level up state.

I added a single constructor to this game state. What it does is create the two dictionaries and initializes their values to 0. I also included a Done attribute that will be displayed with the other attributes that can be selected when all points have been assigned to.

In the LoadContent method I first load the image for the level up state into its field. I then load the font into its field as well. Next up I center the level up background image on the screen. Finally I call the LoadContent method on the base class to load any base content.

The way the level up state was designed to work is that there is the list of attributes and a Done option. When an attribute is selected if the player presses the left key the selected attribute will have an assigned point taken away and if the right key was selected a point will be added. Once all of the attribute points have been assigned they can choose the Done option to assign the points.

So, in the Update method I handle this logic. First, there are local variables to hold what the selected index and item are. I then check to see if the Down key was pressed. If it was pressed I increment the selected field of this class. I then check to see if that value is greater than or equal to the number of elements in the dictionary. If it is I reset selected back to the last element in the list. I do the same thing for the Up key but in reverse. I make sure that the value is never below 0.

After checking for up and down I check if the Space and Enter keys were pressed. If one of them was pressed I then check if the selected item is the last item, Done, and that there are no remaining points to be assigned. If those conditions are true I call the AssignPoint method on the player avatar to assign any points to that attribute. I then remove the states that were added to the stack.

There is then a local variable called increment. This value holds how many points are assigned based on the selected attribute. All attributes but health add 1 point to the selected attribute. The health attributes add 5 points to the avatar's health.

Now, I check to see if the Right key has been released and that there are free points to spend. If there are I loop through all of the keys in the assignedTo dictionary. If the selected option is Done then I exit the method. I then compare the variable i with the field selected. If they are the same I set attribute to be the current key. I then check if attribute is Health and if it is I multiply the value by 5. I then subtract 1 from the points that are available and increment that key in the dictionary. Finally, if there are no points left to assigned I set the selected attribute to the Done option.

I do the same thing in the case where I'm checking if the Left key was released if there are assigned points that can be moved. Next is the same loop that checks to see what attribute is currently selected and if it is Done exit the method. There is then an if statement that validates that the player can remove a point from that category.

The Draw method is pretty straight forward. First, draw the image for the background of the state. Next, create a local variable that determines where text will be positioned. Next I write the name of the avatar and then increment the position the next value will be written to. There is then a local variable i that is used for indexing items. In a foreach loop I go over the keys in the dictionary. I have a tint colour of Black that determines what colour to draw text in. If the current option is the selected option I change the tint colour to Red.

If the option is not the Done option I draw the attribute in the tint colour. I update the X value of the position to draw the individual parts and then reset it back to its default value and increment the Y spacing attribute. I then increase the i local variable and go onto the next iteration of the loop. If it is Done I draw Done and then increase the Y position so there is space between Done and the last bit of text to be drawn. The last bit of text to be draw is how many points are left to be assigned.

Finally there is a SetAvatar method that sets the Avatar being used in the LevelUpState. What it does is set the Avatar field player to be the avatar passed in. It then sets the values of the two dictionaries based off of that avatar. Finally it resets the points and selected fields.

Now that we have a LevelUpState class we need to add it to the game. Open the Game1.cs file. Where the other game states are add the following field.

```
        ILevelUpState levelUpState;
```

With the other game state properties add this property to expose the field to other classes.

```
        public ILevelUpState LevelUpState
        {
            get { return levelUpState; }
        }
```

Update the constructor to initialize the LevelUpState that we just created.

```
        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);
            conversationState = new ConversationState(this);
            battleState = new BattleState(this);
            battleOverState = new BattleOverState(this);
            damageState = new DamageState(this);
            levelUpState = new LevelUpState(this);

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

            characterManager = CharacterManager.Instance;
        }
```

The next step will be to call this state when a battle is over to see if the avatar needs to be levelled up. That will be done in the BattleOverState in the Update method. Update that method to the following.

```
        public override void Update(GameTime gameTime)
        {
            PlayerIndex? index = PlayerIndex.One;

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                if (levelUp)
                {
                    manager.PushState((LevelUpState)GameRef.LevelUpState, 
PlayerIndexInControl);
                    GameRef.LevelUpState.SetAvatar(player);

                    this.Visible = true;
                }
                else
                {
                    manager.PopState();
                    manager.PopState();
                }
            }

            base.Update(gameTime);
        }
```

All that the new code does is if the field levelUp is true is push the level up state on top of the stack of game states and then sets the avatar that levelled up to the current avatar in use. What I've found is the Wind avatar almost always wins the battle against the Fire avatar in my testing, which is part of the reason you need to do a lot of testing of your game play elements. You might think that something works fine but in reality it is broken and the player will not be able to get past a certain point. In order to trigger an avatar level up I modified the CheckLevelUp method of the Avatar class. Update that method to the following code.

```
        public bool CheckLevelUp()
        {
            bool leveled = false;

            if (experience >= 50 * (1 + (long)Math.Pow((level - 1), 2.5)))
            {
                leveled = true;
                level++;
            }

            return leveled;
        }
```

All that this does is lower the amount of experience required to reach level 2 so fighting and losing will still cause the player's avatar to level up. I also want to update the AssignPoint method of the Avatar class because I'm controlling the way points are assigned in the LevelUpState instead of the Avatar class. Update that method to the following.

```
        public void AssignPoint(string s, int p)
        {
            switch (s)
            {
                case "Attack":
                    attack += p;
                    break;
                case "Defense":
                    defense += p;
                    break;
                case "Speed":
                    speed += p;
                    break;
                case "Health":
                    health += p;
                    break;
            }
        }
```

All that changes here is that when I update the health attribute it just uses the value passed in instead of 5 times the value passed in.

There is one minor issue still. You can continuously battle the other characters and train your avatar infinitely. There would be diminishing returns because as you level up when you battle you will gain less and less experience. It would be better if once you battled them you could not battle them again, like in Pokemon. It raises an issue with my story line. If you are fighting with summoned beings, how are you going to handle random encounters? I will leave the latter for another tutorial but I will cover limiting the number of times you can battle an NPC.

In order to do that I added a new property to the ICharacter interface, Battled. Update that interface to the following.

```
    public interface ICharacter
    {
        string Name { get; }
        bool Battled { get; set; }
        AnimatedSprite Sprite { get; }
        Avatar BattleAvatar { get; }
        Avatar GiveAvatar { get; }
        string Conversation { get; }
        void SetConversation(string newConversation);
        void Update(GameTime gameTime);
        void Draw(GameTime gameTime, SpriteBatch spriteBatch);
    }
```

The next step will be to update the Character class to include this update. In this case I'm just going to use an auto-property rather than having a field and work with the field using the property. Add the following line with the other properties in the Character class.

```
        public bool Battled { get; set; }
```

The last step will be that in the GamePlayState class when checking if the player triggered a battle check if the player has battled that character previously and if they have not go to the battle state. If they go to the battle state you need to update that property. Rather than paste the entire Update method I'm only pasting the if statement that checks if the B key was released.

```
            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in map.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                            PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }
```

I'm going to end the tutorial here as I like to keep the tutorials to a reasonable length so there is not a lot of new code to digest at once. 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

A Summoner's Tale – MonoGame Tutorial Series

Chapter 14

Changing Maps

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.

In this tutorial I'm going to add the ability for the player to switch between maps. I will be adding a small building that the player can enter. To do this in my game I added in a new layer that I called a portal layer. I defined a portal as a tile that leads somewhere else. It does not have to lead to a different map but most often they will. So, you can also use a portal to move the player from one position on the map to a different position on the map, like a teleporter.

Let's get started. Open up your solution from the last time. Right click the TileEngine folder, select Add and then Class. Name this new class Portal. This class defines the properties and methods of a portal. Here is the code for this class.

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

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

        Point sourceTile;
        Point destinationTile;
        string destinationLevel;

        #endregion

        #region Property Region
        [ContentSerializer]
        public Point SourceTile
        {
            get { return sourceTile; }
            private set { sourceTile = value; }
        }

        [ContentSerializer]
        public Point DestinationTile
        {
            get { return destinationTile; }
            private set { destinationTile = value; }
        }

        [ContentSerializer]
        public string DestinationLevel
        {
            get { return destinationLevel; }
            private set { destinationLevel = value; }
        }

        #endregion

        #region Constructor Region

        private Portal()
        {
        }

        public Portal(Point sourceTile, Point destinationTile, string destinationLevel)
        {
            SourceTile = sourceTile;
            DestinationTile = destinationTile;
            DestinationLevel = destinationLevel;
        }

        #endregion
    }
}
```

So, a portal had three properties in my game. It had a Point that held the X and Y coordinates of where the portal was located on the map. It had another Point that held the X and Y coordinates of the destination tile for the portal. It also had a string variable the held the name of the map/level that the portal led to.

I added three variables to the class to hold those three properties: sourceTile, destinationTile, and destinationLevel. I then included three properties that exposed their values but could not be set outside of the class. The reason for doing this is two fold. First, you typically do not want to change the value of a portal. In some instances you may want a portal to change where it is located or where it leads to but usually this is not the norm. I also made the set accessors private because there are required to serialize and deserialize maps. It is also why I included a private constructor that takes no parameters and has no actions in it. Finally, there is a constructor that takes as parameters, the source tile, the destination tile and the destination level. It then assigns those values to the fields in the class. You will also so that I marked all of the properties with the ContentSerializer attribute so that they are serialized and deserialized using the Intermediate Serializer.

Now that there is a class that represents a Portal I now added a layer to the map called PortalLayer. It was a collection of Portal objects on the map. Keep in mind when developing tile maps your layers don't necessarily have to display graphics. They can also hold metadata about the map, in this case the portals on the map. Now, right click on the TileEngine folder, select Add and then Class. Name this new class PortalLayer.

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

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

        private Dictionary<Rectangle, Portal> portals;

        #endregion

        #region Property Region

        [ContentSerializer]
        public Dictionary<Rectangle, Portal> Portals
        {
            get { return portals; }
            private set { portals = value; }
        }

        #endregion

        #region Constructor Region

        public PortalLayer()
        {
            portals = new Dictionary<Rectangle, Portal>();
        }

        #endregion
    }
}
```

So, what we have here is a single field called portals, a property that exposes it for public read access but private write access called Portals and a constructor that creates the field. It is also marked with the ContentSerializer attribute so that it will be serialized and deserialized by the Intermediate Serializer.

Now, the portal layer needs to be integrated into the existing TileMap class. The changes were made so that it would not break any of the current functionality. Open the TileMap class and update the code field, property and constructor regions.

```
        #region Field Region

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

        [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 PortalLayer PortalLayer
        {
            get { return portalLayer; }
            private set { portalLayer = 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, PortalLayer portals = null)
        {
            this.characters = new Dictionary<string, Point>();
            this.tileSet = tileSet;
            this.mapName = mapName;
            characterManager = CharacterManager.Instance;

            portalLayer = portals != null ? portals : new PortalLayer();
        }

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

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

        #endregion
```

The change here is I added a field to hold the PortalLayer associated with the TileMap, a property to expose it externally as read and internally as write. I marked this with the ContentSeralizier property so that it will be serialized and deserialized. I add a PortalLayer parameter as an optional parameter that has the value of NULL. What this does is it makes it so that the change will not require you to find everywhere you create a TileMap object and add a new parameter to the call. In the second constructor I check to see if the portals parameter is not NULL. If it is not NULL I assign the value to the local variable, otherwise I create a new PortalLayer. If you know that an if statement is going to just set values by checking if an object has a value or not you can use the ? operator to do that. It is read **condition ? true action : false action**. It is a nice shorthand and cleans up the code a bit.

There is one other metadata class that I want to add. This class holds all of the maps for the game. I called this class World. Right click the TileEngine folder, select Add and then Class. Name this new class World. Here is the code for that class.

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

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

        private Dictionary<string, TileMap> maps;
        private string currentMapName;

        #endregion

        #region Property region

        [ContentSerializer]
        public Dictionary<string, TileMap> Maps
        {
            get { return maps; }
            private set { maps = value; }
        }

        [ContentSerializer]
        public string CurrentMapName
        {
            get { return currentMapName; }
            private set { currentMapName = value; }
        }

        public TileMap CurrentMap
        {
            get { return maps[currentMapName]; }
        }

        #endregion

        #region Constructor Region
        public World()
        {
            maps = new Dictionary<string, TileMap>();
        }

        #endregion

        #region Method Region

        public void AddMap(string mapName, TileMap map)
        {
            if (!maps.ContainsKey(mapName))
                maps.Add(mapName, map);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
        {
            CurrentMap.Draw(gameTime, spriteBatch, camera);
        }

        public void ChangeMap(string mapName, Rectangle portalLocation)
        {
            if (maps.ContainsKey(mapName))
            {
                currentMapName = mapName;
                return;
            }

            throw new Exception("Map name or portal name not found.");
        }

        #endregion
    }
}
```

This was a very simple class in my game that was responsible for managing all of the maps in the game. There were two fields in the class. One is a dictionary of the maps in the game and the other is a string that holds the name of the current map. There are properties marked with the ContentSerializer attribute so that the world can be serialized and deserialized. This allowed me to load all of the maps in my game at once. I also had an editor that worked with the maps. I will need to clean it up and update it a bit but I will eventually be posting the map editor that I used in this game. There are a few methods in this class. The one adds a map to the world, the second draws the current map.

There is also a method called ChangeMap. This is the method that will change the current map with the desired map, if it exists in the collection of maps. You probably should add a bit more validation here to make sure the destination is inside the map that you are switching to.

Now, this needs to be added to the GamePlayState to incorporate it into the existing game. I'm going to do this in stages because there are a lot of changes to the state to incorporate it into the game. First, in the GamePlayState, replace the TileMap map field with the field World world like this.

```
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);
        World world;
        Camera camera;
        Player player;
```

Now, the Update method needs to be modified because it used the map field to search for characters on the map. Change the Update method 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 world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = GameRef.CharacterManager.GetCharacter(s);
                    Rectangle r = new Rectangle(
                        (int)world.CurrentMap.Characters[s].X * Engine.TileWidth + cp,
                        (int)world.CurrentMap.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(world.CurrentMap.WidthInPixels,
world.CurrentMap.HeightInPixels));
            }
            else
            {
                player.Sprite.IsAnimating = false;
            }

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f)
                    {
                        IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                        manager.PushState(
                            (ConversationState)conversationState,
                            PlayerIndexInControl);

                        conversationState.SetConversation(player, c);
                        conversationState.StartConversation();
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);
                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                             PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }
            base.Update(gameTime);
        }
```

All that I did in this method was replace all the instances of map with world.CurrentMap. I did the exact same thing in the Draw method. You can update it to the following code.

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

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

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

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

            GameRef.SpriteBatch.End();
        }
```

The last change was to SetUpNewGame. In this method I initialized the world field, added a local variable to hold the map. I then created the map, added it to the world and changed the map to be this new map.

```
        public void SetUpNewGame()
        {
            Texture2D spriteSheet = content.Load<Texture2D>(@"PlayerSprites\maleplayer");
            TileMap map = null;
            world = new World();

            player = new Player(GameRef, "Wesley", false, spriteSheet);
            player.AddAvatar("fire", AvatarManager.GetAvatar("fire"));
            player.SetAvatar("fire");

            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();

            ConversationManager.CreateConversations(GameRef);

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

            teacherOne.SetConversation("LanceHello");
            teacherTwo.SetConversation("MarissaHello");

            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));

            map.PortalLayer.Portals.Add(Rectangle.Empty, new Portal(Point.Zero, Point.Zero, 
"level1"));
            world.AddMap("level1", map);

            world.ChangeMap("level1", Rectangle.Empty);

            camera = new Camera();
        }
```

At this point I found a problem with the last tutorial. I missed implementing a member of the ICharacter interface for the PCharacter class so the game will not build. Add the following propery to the PCharacter class.

```
        public bool Battled
        {
            get
            {
                throw new NotImplementedException();
            }
            set
            {
                throw new NotImplementedException();
            }
        }
```

It will need to be fleshed out at some point but I'm going to do that in another tutorial. So, if you build and run the game you can start a new game and everything will work as expected. You can have a conversation with the NPCs and you can battle them. You still can't switch to another map. The first reason is that there is no portal to another map and second is that we haven't implemented that ability in the Update method. First, I will add the ability to switch maps in the Update method. Next I will update the SetUpNewGame method so that it creates two maps with a portal on each map that leads between the two maps.

I'm going to go with the default action, space bar or enter key, to activate an object, portal, initiate a conversation, etc. To do that I added the following changes to the Update method.

```
        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 world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = GameRef.CharacterManager.GetCharacter(s);
                    Rectangle r = new Rectangle(
                        (int)world.CurrentMap.Characters[s].X * Engine.TileWidth + cp,
                        (int)world.CurrentMap.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(world.CurrentMap.WidthInPixels,
world.CurrentMap.HeightInPixels));
            }
            else
            {
                player.Sprite.IsAnimating = false;
            }

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f)
                    {
                        IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                        manager.PushState(
                            (ConversationState)conversationState,
                            PlayerIndexInControl);

                        conversationState.SetConversation(player, c);
                        conversationState.StartConversation();
                    }
                }

                foreach (Rectangle r in world.CurrentMap.PortalLayer.Portals.Keys)
                {
                    Portal p = world.CurrentMap.PortalLayer.Portals[r];

                    float distance = Vector2.Distance(
                        player.Sprite.Center,
                        new Vector2(
                            r.X * Engine.TileWidth + Engine.TileWidth / 2,
                            r.Y * Engine.TileHeight + Engine.TileHeight / 2));

                    if (Math.Abs(distance) < 64f)
                    {
                        world.ChangeMap(p.DestinationLevel, new Rectangle(p.DestinationTile.X,
p.DestinationTile.Y, 32, 32));

                        player.Position = new Vector2(
                            p.DestinationTile.X * Engine.TileWidth,
                            p.DestinationTile.Y * Engine.TileHeight);
                        camera.LockToSprite(world.CurrentMap, player.Sprite,
Game1.ScreenRectangle);

                        return;
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                            PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }
            base.Update(gameTime);
        }
```

Inside the if statement that checks if Space/Enter have been pressed I added another foreach loop that loops over the keys in the portal layer of the current map. I first get the portal using the key. I then calculate the distance between the character and the portal as I did for conversations and battling avatars. You will notice that I'm multiplying X and Y coordinates by TileWidth and TileHeight properties of the engine to get positions in pixels instead of tiles. If the portal and player are close enough together I change the map by calling ChangeMap passing in the name of the level and the destination rectangle. I then update the player's position to be the position on the new map. Then I lock the camera onto the player on the map and exit the Update method because there is no point in processing anything else in the method at that point.

The last thing that I'm going to cover in this tutorial is creating a basic second map with a portal that leads back to the same position as the first map. I did that in the SetUpNewGame method as that is where I already created a map and added it to the world. Change that map to the following.

```
public void SetUpNewGame()
{
    Texture2D spriteSheet = content.Load<Texture2D>(@"PlayerSprites\maleplayer");
    TileMap map = null;
    world = new World();

    player = new Player(GameRef, "Wesley", false, spriteSheet);
    player.AddAvatar("fire", AvatarManager.GetAvatar("fire"));
    player.SetAvatar("fire");
    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();

    building.SetTile(4, 4, 18);

    ConversationManager.CreateConversations(GameRef);

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

    teacherOne.SetConversation("LanceHello");
    teacherTwo.SetConversation("MarissaHello");

    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));

    map.PortalLayer.Portals.Add(Rectangle.Empty, new Portal(Point.Zero, Point.Zero, "level1"));
    map.PortalLayer.Portals.Add(new Rectangle(4, 4, 32, 32), new Portal(new Point(4, 4), new
Point(10, 10), "inside"));

    world.AddMap("level1", map);
    world.ChangeMap("level1", Rectangle.Empty);

    background = new TileLayer(20, 20, 23);
    edge = new TileLayer(20, 20);
    building = new TileLayer(20, 20);
    decor = new TileLayer(20, 20);

    map = new TileMap(set, background, edge, building, decor, "inside");
    map.FillEdges();
    map.FillBuilding();
    map.FillDecoration();
    map.BuildingLayer.SetTile(9, 19, 18);

    map.PortalLayer.Portals.Add(new Rectangle(9, 19, 32, 32), new Portal(new Point(9, 19), new
Point(4, 4), "level1"));

    world.AddMap("inside", map);

    camera = new Camera();
}
```

What changed as adding the map to the world is that I created a new ground layer filling it with tile 23 from the tile set we are using that is a floor like tile (close enough for me.) I then create the other layers at the same size. I then create the map, call the FillEdge, FillBuilding and FillDecoration methods. I then set one tile to be a door tile. I add a new portal that leads back to the original portal. I then add the map to world. I also set a tile on the first map to be the same door tile to give the visual cue that the player can open the door.

If you build and run the game now and go to the door and press Space/Enter you should now be able to switch to the new map. You can then walk to the far right of the map and switch back to the original map.

I'm going to end the tutorial here as I like to keep the tutorials to a reasonable length so there is not a lot of new code to digest at once. 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!
Cynthia McMahon

A Summoner's Tale – MonoGame Tutorial Series

Chapter 15

Saving Game State

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.

When I asked on the blog for tutorial suggestions one of the comments asked how one would go about saving game state. I will cover adding that feature to the game in this tutorial. The first question that needs to be answered is what exactly needs to be save? The main objects will be the player and their avatars. Other objects would be conversation states, quest states and other events.

First, you must ask yourself the question, "What needs to be saved?". There are a variety of approaches that could be taken. Only save changed data relevant to the current game or use a shotgun approach and save the entire world. This is the easier of the two to implement so I've decided to go with that approach. Let's get started.

First, I want to update the ICharacter interface to add a Save method that any class that implements the interface must implement. Find the ICharacter interface and update the code to the following.

```
using Avatars.AvatarComponents;
using Avatars.ConversationComponents;
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;
using System.IO;

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

The Save method will require that a BinaryWriter object be passed into it. This writer will be used to actually save the character to disk. Why did I decide on using a BinaryWriter instead of a TextWriter or saving as XML? The reason is the other two formats are easy to read and manipulate. A binary file is harder for the player to decode and manipulate. Ideally you'd want to also encrypt the files as well but I will leave that as an exercise.

Next, I want to add the same method to the base Avatar class, but not implement it quite yet. Open the Avatar class and add the following method. Make sure to include the following using statement at the top of the file as well.

```
using System.IO;

        public bool Save(BinaryWriter writer)
         {
            return true;
        }
```

I will implement this method shortly. The last place to add a Save method will be the World class. Again, I'm just adding the stub for now and the rest will be filled out later. Here is the code, including the required using statement.

```
using System.IO;

        public bool Save(BinaryWriter writer)
         {
            return true;
        }
```

The last class that I want to add a Save method to is the Player class. Add the same method and using statements as before, repeated here.

```
using System.IO;

        public bool Save(BinaryWriter writer)
         {
            return true;
        }
```

The method stub needs to be added to the TileMap class as well. Add the method stub and using statement to TileMap.

```
using System.IO;
```

```
        public bool Save(BinaryWriter writer)
        {
            return true;
        }
```

The next step will be to trigger the save process. I did that by checking if the F1 key has been released and if it has trigger the save event for the world. Modify the Update method as follows. Make sure to add a using statement for System.IO at the beginning of the file.

```
using System.IO;
        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 world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = GameRef.CharacterManager.GetCharacter(s);
                    Rectangle r = new Rectangle(
                        (int)world.CurrentMap.Characters[s].X * Engine.TileWidth + cp,
                        (int)world.CurrentMap.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(world.CurrentMap.WidthInPixels,
world.CurrentMap.HeightInPixels));
            }
            else
            {
                player.Sprite.IsAnimating = false;
            }

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f)
                    {
                        IConversationState conversationState =
(IConversationState)GameRef.Services.GetService(typeof(IConversationState));
                        manager.PushState(
                            (ConversationState)conversationState,
                            PlayerIndexInControl);

                        conversationState.SetConversation(player, c);
                        conversationState.StartConversation();
                    }
                }

                foreach (Rectangle r in world.CurrentMap.PortalLayer.Portals.Keys)
                {
                    Portal p = world.CurrentMap.PortalLayer.Portals[r];
                    float distance = Vector2.Distance(
                        player.Sprite.Center,
                        new Vector2(
                            r.X * Engine.TileWidth + Engine.TileWidth / 2,
                            r.Y * Engine.TileHeight + Engine.TileHeight / 2));

                    if (Math.Abs(distance) < 64f)
                    {
                        world.ChangeMap(p.DestinationLevel, new Rectangle(p.DestinationTile.X,
p.DestinationTile.Y, 32, 32));

                        player.Position = new Vector2(
                            p.DestinationTile.X * Engine.TileWidth,
                            p.DestinationTile.Y * Engine.TileHeight);
                        camera.LockToSprite(world.CurrentMap, player.Sprite, Game1.ScreenRectangle);

                        return;
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.B))
            {
                foreach (string s in world.CurrentMap.Characters.Keys)
                {
                    ICharacter c = CharacterManager.Instance.GetCharacter(s);
                    float distance = Vector2.Distance(player.Sprite.Center, c.Sprite.Center);

                    if (Math.Abs(distance) < 72f && !c.Battled)
                    {
                        GameRef.BattleState.SetAvatars(player.CurrentAvatar, c.BattleAvatar);
                        manager.PushState(
                            (BattleState)GameRef.BattleState,
                            PlayerIndexInControl);
                        c.Battled = true;
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.F1))
            {
                FileStream stream = new FileStream("avatars.sav", FileMode.Create,
FileAccess.Write);
                BinaryWriter writer = new BinaryWriter(stream);
                world.Save(writer);
                player.Save(writer);
                writer.Close();
                stream.Close();
            }
            base.Update(gameTime);
        }
```

First, there is of course the check to see if the F1 key has been released this frame. If it has I create a FileStream to write the file to disk. The location of this file will be inside the debug folder, for now. It will create a new file even if there is an existing file and open it with write permissions. I then create the BinaryWriter passing in the FileStream. Next step is to call the Save method on the World object and the Save method of the Player object passing in the BinaryWriter object. I then close both the writer and the stream.

The first Save method that I will implement is on the World class. Modify that class to the following code.

```
public bool Save(BinaryWriter writer)
{
    writer.Write(currentMapName);

    foreach (string s in maps.Keys)
        maps[s].Save(writer);

    return true;
}
```

What is happening here is first I'm writing the name of the current map the player is in. Next I loop through all of the TileMaps and call their save method. It is still just returning true but eventually we will add in some error checking to make sure nothing unusual happens and that we can recover from it.

Now with the World done we'll tackle the TileMap class. Update the Save method of the TileMap class to the following.

```
public bool Save(BinaryWriter writer)
{
    foreach (string s in characters.Keys)
    {
        ICharacter c = CharacterManager.Instance.GetCharacter(s);
        c.Save(writer);
    }

    return true;
}
```

In this case what we do is iterate over the list of characters on the map. Then we use the CharacterManager object to get the character. Finally we call the Save method on the Character. I had to make two tweaks here. First, I added a new field called textureName that will as the name implies store the name of the texture for this character. Second, I set the value of the variable in the FromString method. Here is the updated code for the field and methods.

```
private string textureName;

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];
    character.textureName = parts[1];
    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];

    character.battleAvatar = AvatarManager.GetAvatar(parts[4].ToLowerInvariant());

    return character;
}


public bool Save(BinaryWriter writer)
{
    StringBuilder b = new StringBuilder();
    b.Append(name);
    b.Append(",");
    b.Append(textureName);
    b.Append(",");
    b.Append(sprite.CurrentAnimation);

    writer.Write(b.ToString());

    if (givingAvatar != null)
        givingAvatar.Save(writer);

    if (battleAvatar != null)
        battleAvatar.Save(writer);

    return true;
}
```

In the FromString method I assign the new field, characterTexture name to the name of the texture for the character. In the Save method I create a StringBuilder object and append the fields of the class in the order that are required by the FromString method, other than the Avatars. I then call the Write method of the BinaryWriter to write the character to disk. Here is where you'd want to encrypt the string in some way. The string will show up as text in the file even though it's a binary file. After calling the Write method I call the Save method of the two Avatar objects in this class. There are null checks to make sure there is something to be written.

I'm now going to do something similar in the PCharacter class. I added the same field and updated the FromString method. Add this field, update the FromString method and update the Save method to the following. I also write the Avatar objects separately from the character being written.

```
private string textureName;

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];
    character.textureName = parts[1];
    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];
    character.currentAvatar = int.Parse(parts[4]);

    for (int i = 5; i < 11 && i < parts.Length; i++)
        character.avatars[i - 5] = AvatarManager.GetAvatar(parts[i].ToLowerInvariant());

    return character;
}

public bool Save(BinaryWriter writer)
{
    StringBuilder b = new StringBuilder();

    b.Append(name);
    b.Append(",");
    b.Append(textureName);
    b.Append(",");
    b.Append(sprite.CurrentAnimation);
    b.Append(",");
    b.Append(conversation);
    b.Append(",");
    b.Append(currentAvatar);

    writer.Write(b.ToString()

    foreach (Avatar a in avatars)
    {
        if (a != null)
            a.Save(writer);
    }

    return true;
}
```

Very similar to the Character class. The differences are that I updated to the FromString method to shift the fields one part of the string. I also updated the loop where we iterate over the list of avatars to use the new index. Similarly, I use a string builder to create an object that can be written to disk. Once the player stats have been written the next step is to write out the avatars. To do that I look over all of the avatars in the array and call their Save methods passing in the writer so they can be saved to disk. Again, there is a null check to make sure that there is something to be written to disk. I still only return true because we haven't implemented that part yet.

Now it is time to implement the Save method of the Avatar class. Find that method and update it to the following.

```
public bool Save(BinaryWriter writer)
{
    StringBuilder b = new StringBuilder();

    b.Append(name);
    b.Append(",");
    b.Append(element);
    b.Append(",");
    b.Append(experience);
    b.Append(",");
    b.Append(costToBuy);
    b.Append(",");
    b.Append(level);
    b.Append(",");
    b.Append(attack);
    b.Append(",");
    b.Append(defense);
    b.Append(",");
    b.Append(speed);
    b.Append(",");
    b.Append(health);
    b.Append(",");
    b.Append(currentHealth);


    foreach (string s in knownMoves.Keys)
    {
        b.Append(",");
        b.Append(s);
    }

    writer.Write(b);

    return true;
}
```

Just like in the other methods there is StringBuilder that I use to build the string to written to disk. I then append the fields of the class one by one with a comma afterwards. Instead of writing a comma after currentHealth I go straight to a loop. In the loop I append the comma and then the key for the move. Finally, I write the StringBuilder to disk.

I'm going to end this tutorial here. In the next tutorial I will reverse the process and load a game from disk. Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so.

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!
Cynthia McMahon