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.

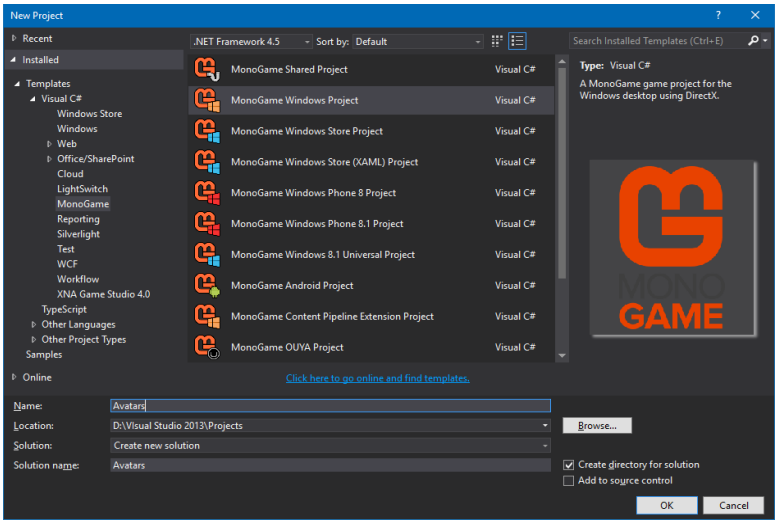
I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

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.



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