Shadow Monsters – MonoGame Tutorial Series
Chapter 3
State Management

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called Shadow Monsters. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my blog: Shadow Monsters. The source code for each tutorial will be available as well. I will be using Visual Studio 2019 Community for the series. The code should compile on the 2013, 2015 and 2017 versions as well.

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

In this tutorial I will be covering state management. State management allows you to control the flow of the game and abstract and organize your code. In theory you could just throw everything into a few classes and manage state manually but a state manager makes it much cleaner. Question is what is a game state? A game state is a logical abstraction of the different modes in the game. For example, the player exploring the map is a state the same as the player viewing inventory or having a conversation with an NPC.

First, we need a class that represents a game state. Right click the ShadowMonsters project in the Solution Explorer, select Add and then Class. Name this new class GameState. Here is the code for that class.

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

namespace ShadowMonsters
{
    public interface IGameState
    {
        GameState Tag { get; }
    }

    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;

        #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
    }
}
```

There are using statements to bring the MonoGame classes that we will be using into scope. There is a public interface that has a single property Tag that is a GameState. This is used to know what state the state is. The class itself is abstract meaning that it cannot be instantiated it must be inherited and instantiated in the child class. It also inherits GameComponent and implements the interface that we declared earlier.

For fields there is a GameState field, tag, that is for the interface property Tag, Again, you could just use an auto-property but I prefer fields and exposing them with properties. There is an IStateManager field that we haven't added the code for yet. We will get to that shortly. There is a ContentManager field that will be used to load any content that the game state needs. There is also a List<GameComponent> for any components the state may need.

There are two public properties in this class. The first exposes the components field and the second implements the IGameState interface.

The constructor takes as a parameter a Game object, required for the base class, and calls the base class passing in the Game object. It sets the tag field to this, the current game state. Next I create a new List<GameComponent> and assign the content field to the

ContentManager of the Game class passed in. Finally I get the state manager object that will be registered in the state manager. The LoadContent method just calls the base method.

The Update method loops over all of the child components. If they are Enabled, a property of GameComponent, their Update method is called. Similarly, the Draw method loops over all of the components. If the component is a DrawableGameComponent and it is Visible, a property of DrawableGameComponent, I call the Draw method. Both methods call their base method. I call base.Draw before drawing the components or the base will be drawn over top of them.

There is a method, StateChanged, that will be called when the game state changes. If the state being changed is this state I call the Show method to show the game state. Otherwise I call the Hide method to hide the state. The Show method sets their Visible and Enabled properties to true. It then loops over all of the game components in the state and sets their Enabled property to true. If the component is a DrawableGameComponent I set the Visible property to true as well. Hide works the same way but sets the properties to false.

Now I will add the state manager that handles game state. Right click the ShadowMonsters project in the SolutionExplorer, select Add and then Class. Name this new class GameStateManager. Here is the code for that class.

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

namespace ShadowMonsters
{
    public interface IStateManager
    {
        GameState CurrentState { get; }

        event EventHandler StateChanged;

        void PushState(GameState state);
        void ChangeState(GameState state);
        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);
    drawOrder = startDrawOrder;
}

#endregion

#region Method Region

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

private void AddState(GameState state)
{
    gameStates.Push(state);
    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)
{
    while (gameStates.Count > 0)
    {
        RemoveState();
```

```
            }

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

            AddState(state);
            OnStateChanged();
        }

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

        protected internal virtual void OnStateChanged()
        {
            StateChanged?.Invoke(this, null);
        }

        #endregion
    }
}
```

Here is the interface we referenced in the last class. It has as a property the current game state. Next is an event, StateChnaged, that is called when the state changes. We will be implementing state management using a stack. States are pushed onto the stack as they are in use. When they are no longer in use they are popped off the stack. The current state is on the top of the stack. There are methods for pushing a state, popping a state and changing the state. There is also a method to check if a state exists on the stack.

The class itself inherits from GameComponent and implements the IStateManager interface. For fields it has a Stack<GameState> for the states that we talked about earlier. There are two constants, startDrawOrder and drawOrderInc. States are layered and will be drawn based on a field drawOrder, higher values will be drawn over top of lower values. The constants just define the base line and the increment between states. There is also a property that implements the property from the interface.

There is just one constructor. It takes as a parameter a Game object that is required by the base class. Here is where we register the state manager as a service using the interface. It is better to use an interface than the class because you don't always want everything in the class to be accessible. I also initialize the drawOrder field in the constructor.

The first method that I implement is the PushState method. This method increments the drawOrder field, calls a helper method AddState and OnStateChange to notify any subscribers to the event that there has been a state change. AddState pushes the state onto the stack and adds it to components of the game. Finally it subscribes to the StateChanged event.

The next interface method that I implement is the PopState method. What it does is check to see if there is a state to pop off of the stack. If there is I call a helper method RemoveState, decrement the drawOrder field and call OnStateChanged to notify the states that there has been a change in state. The RemoveState method peeks what state is on top of the stack. It

then unsubscribes from the StateChanged event, removes the state from the game components and pops the state off of the stack.

What the ChangeState method does is remove all states and pushes a state onto the stack. It also resets the drawOrder back to its base state. ContainsState checks if the state is on the stack or not. OnStateChange just invokes the delegate.

I want to add another game state that houses information stored by all other game states. Right click the ShadowMonsters project in the Solution Explorer, select Add and then New Folder. Name this new folder GameStates. Now right click the GameStates folder, select Add and then Class. Name this new class BaseGameState. Here is the code for that class.

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

namespace ShadowMonsters.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
    }
}
```

There are two protected fields in this class. The first is a Random that will be shared by all game states rather than having multiple Random objects. There is also a Game1 field that is a reference to the game. There will be multiple instances where this will be important. The

constructor just sets the gameRef field. The overrides of LoadContent, Update and Draw just call the respective base methods.

The last thing I'm going to cover in this tutorial is moving everything in the Game1 class into a game state. Right click the GameStates folder in the Solution Explorer, select Add and then Class. Name this new class GamePlayState. Here is the code for that class.

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

namespace ShadowMonsters.GameStates
{
    public class GamePlayState : BaseGameState
    {
        private readonly Engine engine = new Engine(new Rectangle(0, 0, 1280, 720));
        private TileMap map;
        private Dictionary<AnimationKey, Animation> animations = new Dictionary<AnimationKey,
Animation>();
        private AnimatedSprite sprite;
        private Vector2 motion;

        public GamePlayState(Game game) : base(game)
        {
        }

        protected override void LoadContent()
        {
            TileSet set = new TileSet();
            set.TextureNames.Add("tileset1");
            set.Textures.Add(content.Load<Texture2D>(@"Tiles\tileset16-outdoors"));

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

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

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

            engine.SetMap(map);

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

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

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

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

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

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

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

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

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

            sprite.Position = newPosition;
            motion = sprite.LockToMap(
                new Point(
```

```
                    map.WidthInPixels,
                    map.HeightInPixels),
                motion);
            }

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

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            engine.Draw(gameTime, GameRef.SpriteBatch);
            GameRef.SpriteBatch.Begin(
                SpriteSortMode.Deferred,
                BlendState.AlphaBlend,
                SamplerState.PointClamp,
                null,
                null,
                null,
                Engine.Camera.Transformation);
            sprite.Draw(gameTime, GameRef.SpriteBatch);
            GameRef.SpriteBatch.End();
        }
    }
}
```

The code is almost identical to what was in the Game class. The only differences are in the Draw method and the LoadContent method. Instead of having direct access to the SpriteBatch field I had to reference it using the GameRef field. In LoadContent I use the shared Random object of BaseGameState instead of the one from the Game1 class.

The last change will be in the Game1 class. Since the changes were extensive I will give you the code for the entire class.

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

namespace ShadowMonsters
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    public class Game1 : Game
    {
        public static Random Random = new Random();
        private readonly GraphicsDeviceManager graphics;
        private SpriteBatch spriteBatch;
        private readonly GamePlayState gamePlayState;
        private readonly GameStateManager stateManager;
```

```
public SpriteBatch SpriteBatch => spriteBatch;
public GamePlayState GamePlayState => gamePlayState;

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

    graphics.ApplyChanges();

    Content.RootDirectory = "Content";

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

    gamePlayState = new GamePlayState(this);

    stateManager.PushState(gamePlayState);
}

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

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

    // TODO: use this.Content to load your game content here
}

/// <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);
        }
    }
}
```

Firs thing I did was remove all of the fields that were specific to game play. I added in fields for the GamePlayState and the GameStateManager. There are properties to expose the spriteBatch field and the gamePlayState fields. In the constructor I instantiate the stateManager and gamePlayState fields. I also add the stateManager to the list of components for the game. Finally, I push the GamePlayState on top of the stack. I also removed all of the old code from the LoadContent, Update and Draw methods.

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

I wish you the best in your MonoGame Programming Adventures!