Forest Rush – 3.8.1

Part 1

In the Beginning…

I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The tutorials will make more sense if they are read in order. You can find the tutorials on the Forest Rush page of my web blog. In addition, I will be making the project available on GitHub here. It will be included on the page that links to the tutorials.

This is part one of a series of tutorials I plan to write on creating a platformer with MonoGame. In the process of writing more tutorials, I discover better ways of doing some things, and I have to go back to fix things. I'm hoping in this series to avoid many of these mistakes. Also, I am going to make the game cross-platform. In my previous tutorials, they were focused on Windows only. I want to open things up for macOS, Linux, Android, iOS, and any platform MonoGame supports.

So, let's get started. For these tutorials, I will be using resources from https://craftpix.net. They have graciously permitted me to use their free resources in my tutorials. They have some amazing resources, free and paid. I am using them in my projects. I recommend browsing their site for resources to use in your games. I will be using five resources in these tutorials. You can download them from the following links.

Tile Set
https://craftpix.net/freebies/free-medieval-ruins-cartoon-2d-tileset/?num=2&count=220&sq=tiles&pos=1

Chibis
https://craftpix.net/freebies/free-fallen-angel-chibi-2d-game-sprites/
https://craftpix.net/freebies/free-golems-chibi-2d-game-sprites/
https://craftpix.net/freebies/free-orc-ogre-and-goblin-chibi-2d-game-sprites/
https://craftpix.net/freebies/free-reaper-man-chibi-2d-game-sprites/

Download and save the resources. Next, I will add just a little infrastructure for the game. The first is a game state that will be the base of all our game states. This allows us to manage game states using polymorphism. If you are unfamiliar with the term, the base class can act as an inherited class. Don't worry. There is no test, and you will better understand seeing it in practice.

Infrastructure

To get started, we first need a solution to work with. Open Visual Studio and click the Create a new project tile. In the window that pops up, search for MonoGame Cross-Platform Desktop. Select that entry on the left side and click Next. Enter ForestRush for the name in this window and click the Create button.

Off to a great start. We need another project to hold our common code and compile our assets. Right-click the ForestRush solution in the Solution Explorer, select Add and then New Project. In the window that pops up, search for MonoGame Shared Library Project and click the Next button. Name this new project ForestRushShared and click Create. We need to tell our game about the new project so it can use it. Right-click the ForestRush project in the Solution Explorer, select Add and then Share Project Reference. Select the ForestRushShared project and click OK.

So, things are shaping up. But, before we start coding, there is one piece of housekeeping that I want to address in the ForestRushShared project, right-click the Game1.cs file and select Delete. Now it is time to add some code! First, I want to add a base game state that all of our game states will implement. Following this approach makes your game cleaner. Right-click the ForestRushShared project, and select Add and New Folder. Name this new Folder GameStates. Next, right-click the GameStates folder, select Add and then Class. Name this new class BaseGameState. When writing tutorials, I typically present the code and then describe it after you've had a chance to read it. So, here is the code for the BaseGameState.

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

namespace ForestRushShared
{
    public abstract partial class BaseGameState : DrawableGameComponent
    {
        #region Field Region

        protected ContentManager Content;
        protected readonly List<GameComponent> childComponents;

        #endregion

        #region Property Region

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

        public BaseGameState Tag => this;

        public SpriteBatch SpriteBatch
        {
            get; private set;
        }

        protected RenderTarget2D RenderTarget
        {
            get; private set;
        }

        #endregion
```

```
#region Constructor Region

public BaseGameState(Game game)
    : base(game)
{
    childComponents = new List<GameComponent>();
    Content = Game.Content;

    SpriteBatch = Game.Services.GetService<SpriteBatch>();
}

#endregion

#region Method Region

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

public override void Update(GameTime gameTime)
{
    foreach (GameComponent component in childComponents.Where(x => x.Enabled))
    {
        component.Update(gameTime);
    }

    base.Update(gameTime);
}

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

    foreach (DrawableGameComponent component in childComponents.Where(
        x => x is DrawableGameComponent &&
        (x as DrawableGameComponent).Visible).Cast<DrawableGameComponent>())
    {
        component.Draw(gameTime);
    }
}

protected internal virtual void StateChanged(object sender, EventArgs e)
{
}

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

    foreach (GameComponent component in childComponents)
    {
        component.Enabled = true;
        if (component is DrawableGameComponent component1)
            component1.Visible = true;
    }
}
```

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

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = false;
                if (component is DrawableGameComponent component1)
                    component1.Visible = false;
            }
        }

        #endregion
    }
}
```

This class was yanked out of my platformer, Eyes of Inashti. I have variations in all of my games as it changes and evolves. It is an abstract class, which means you cannot create an instance of it. However, it does inherit from DrawableGameComponent. There are two kinds of components: GameComponents and DrawableGameComponents. They are special classes that can be added to the list of game components and have their methods called implicitly by MonoGame instead of explicitly by you.

I have included a ContentManager field called Content. You can access it using Game.Content. However, I like having the same object in my game states as in the game itself. There is also a list of GameComponents that are the children of the state. This base state will call the Update and Draw methods of child components the same way as the Game class.

There are a few properties for the class. The first exposes the list of game components. They can be components of drawable components. This is where polymorphism comes into play. It is the ability of a base class, GameComponent, to act as a derived class, DrawableGameComponent. There is a property, Tag, that returns the current instance of the game state as a BaseGameState, another example of polymorphism. There is also a SpriteBatch and RenderTarget that will be used for rendering. When I handle the game in different resolutions, the render target will come into play. I'm just parking it for now.

The constructor requires a Game parameter, which will be our Game1 class because it inherits from the Game class. Yet another example of polymorphism. Okay, I will stop dragging that word around. It is just a feature of object-oriented programming that I use a lot. I mean every opportunity that comes up. It then grabs the content manager and assigns it to the Content field. I then grab the SpriteBatch object that has been registered as a service. That will be done shortly when I focus on the Game1 class.

There is an overload of the LoadContent method. This is where I will eventually be used to initialize the render target. So, basically, I parked it.

The Update method loops over all child components with the Enable property set to true. That is what the Where clause is saying. It then calls the component's update method the same way the game

would. It then calls the base.Update method to have the parent class call its update method. The Draw method calls the base.Draw method before rendering its children so they will appear on top. Next is an ugly for each loop. It filters out the game components that are DrawableGameComponents in a safe way that will not throw an exception. If it is visible, it calls its Draw method.

There is another parked method that will fire when the game state changes. I will not be getting into that in this tutorial. The Show method will be called when the state becomes visible. Similarly, the Hide method will be called when the state is hidden.

The Show method sets the Enabled and Visible properties of the state to true. It then loops over all of the child components. It then sets their Enabled property to true. If it is a DrawableGameComponent, it sets its Visible property to true. The Hide method works in reverse. Their Enabled and Visible properties are set to false.

One more state, and then that is it for infrastructure for a while. After all, you're interested in gameplay, not infrastructure. So, I'm going to add a state for gameplay. Right-click the GameStates folder, select Add and then Class. Name this new class GamePlayState. Here is the boilerplate code that will serve as a starting point for the game.

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

namespace ForestRushShared.GameStates
{
    public class GamePlayState : BaseGameState
    {
        public GamePlayState(Game game) : base(game)
        {
        }

        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);
        }

        public override void Hide()
        {
```

```
            base.Hide();
        }

        public override void Show()
        {
            base.Show();
        }
    }
}
```

For now, it is just a skeleton. Of course, we will be adding more to it as we go. But, for now, all it does is call the base methods of the parent class, BaseGameState.

Help Me! I'm Falling

As you can gather from the heading, I'm moving on to the sprite falling due to gravity. To start, we need sprites, which we have, and a sprite class, which we don't have. I will begin with the sprites. Extract the fallen angel file to a folder. Expand the Content folder in the ForestRushShared project, then double-click the Content.mgcb node to open the MGCB Editor. Right-click the Content node, select Add and then New Folder. Name the new folder PlayerSprite. Right-click the PlayerSprites folder, and select Add and New Folder. Name this new folder Idle. Right-click the PlayerSprite folder, select Add and then New Folder. Name this new folder JumpStart. Repeat the process and call the new folder JumpLoop.

That was a lot of work. It's not over, however. First, we need to add the actual images to the folders. So, right-click the Idle folder and select Add and then Existing Folder. Next, navigate to where you extract the sprites. Find the Fallen_Angles_2 folder. Drill down to the PNG folder, then the PNG Sequences, and select the Idle folder. When prompted, choose either Copy to your project or Add link. I decided to add a link. Why? I can use the assets from CraftPix.net, but you need to download them from their site. So, I can't include them in my archives or GitHub repository. It also saves space on the disk, having one copy instead of two.

So, typically when you deal with animation, the frames for the animation are all stored in the same image. In this case, each frame is in its own image. A little annoying, but we can work with it! It looks like animation is next.

So, right-click the ForestRushShared project in the Solution Explorer, select Add and then New Folder. Name this new folder SpriteClasses. Now, right-click the SpriteClasses folder, select Add and then Class. Name this new class FrameAnimatedSprite. Here is the code for that class.

```
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 Psilibrary.SpriteClasses
{
    public class FrameAnimatedSprite
```

```
    {
        private readonly List<Texture2D> _frames = new();
        protected int _currentFrame;
        private TimeSpan _frameTimer;
        private readonly TimeSpan _frameLength;

        public int Frames { get { return _frames.Count; } }
        public int CurrentFrame { get { return _currentFrame; } }
        public Texture2D CurrentTexture { get { return _frames[_currentFrame]; } }
        public Vector2 Position { get; set; }

        public int Width { get { return _frames[0].Width; } }
        public int Height { get { return _frames[0].Height; } }

        public Rectangle Bounds
        {
            get { return new((int)Position.X, (int)Position.Y, Width, Height); }
        }

        public FrameAnimatedSprite(List<Texture2D> frames)
        {
            foreach (var frame in frames)
            {
                _frames.Add(frame);
            }

            _frameLength = TimeSpan.FromSeconds(1 / 8.0);
        }

        public void Reset()
        {
            _currentFrame = 0;
            _frameTimer = TimeSpan.Zero;
        }

        public void Draw(SpriteBatch spriteBatch, Vector2 position, SpriteEffects flip)
        {
            Rectangle location = new(new((int)position.X, (int)position.Y), new(128,
128));
            spriteBatch.Draw(_frames[_currentFrame], location, null, Color.White, 0f,
Vector2.Zero, flip, 1f);
        }

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

            if (_frameTimer >= _frameLength)
            {
                _frameTimer = TimeSpan.Zero;
                _currentFrame++;
            }
        }
    }
}
```

There is a List<Texture2D> to hold the textures of the animated sprite. It is readonly because I don't want it reassigned outside of the constructor. The _currentFrame counts which frame is currently being rendered. The _frameTime measures the time that has passed since the last frame change. The

_frameLength field is how long a frame is rendered for. There is a property that exposes the number of frames. The CurrentFrame property exposes which frame is being rendered. CurrentTexture exposes what image is currently being drawn. The Position property is the position of the sprite in the world, not on the screen. It is important to remember that. Finally, the Width and Height properties expose the width and height of the sprite.

The constructor takes a single parameter: a List<Texture2D> which is the images for the animation. It loops over the frames collection and adds them to its frame collection. It then sets the frame length to eight frames per second. I find it a good duration for the animations.

The Reset method resets an animation to the beginning. It does that by setting the current frame to zero and the frame timer to zero.

The draw method takes three parameters: a SpriteBatch, a position, and a SpriteEffects. In the Draw method, I create a Rectangle that represents the position of the sprite in world space. That is the position of the sprite and 128 pixels wide and high. In the overload of the Draw method that I use, I pass in the image. This location was just calculated for the destination rectangle, null for the source rectangle because we are using the entire image, white for the tint colour. Hence, it is its original colour, a rotation of zero degrees, the origin of the upper left-hand corner, the flip effect and 1f for the layer depth. So why the flip? Well, the animations are only in one direction. If we want to draw facing the other way, and we do, we can use the flip effect.

That leaves the Update method. For that, we need to pass in a GameTime parameter. It increments the _frameTime by the amount of time that has been called since the last game update. If it is greater than the length of a frame, it sets the frame timer back to zero and increments the current frame.

Before we put falling into action, there are a few pieces we need to put in place in the Game1 class. We need to register the sprite batch as a service, and we need to create a GamePlayState object to hold our gameplay stuff. Replace the Game1 class in the ForestRush project with the following.

```
using ForestRushShared.GameStates;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace ForestRush
{
    public class Game1 : Game
    {
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;

        public GamePlayState GamePlayState { get; private set; }

        public Game1()
        {
            _graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
            IsMouseVisible = true;
```

```
            GamePlayState = new(this);
            Components.Add(GamePlayState);

        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);
            Services.AddService<SpriteBatch>(_spriteBatch);

            // TODO: use this.Content to load your game content here
            GamePlayState.Show();
        }

        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);
        }

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

            // TODO: Add your drawing code here

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

What I did was add a property to hold the GamePlayState. I initialize it in the constructor and add it to the list of components. Then, in the LoadContent, method, I register the _spriteBatch as a service. This means that it can be retrieved from the services and used elsewhere. It saves us from passing it around as a parameter. Finally, I call the Show method of the GamePlayState.

Before I turn my attention to the GamePlayState, I need to add a class that represents the player. It houses things like animations. Right-click the ForestRushShared project, select Add and then Class. Name this new class Player. Here is the code for that class.

```
using ForestRushShared.GameStates;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
```

```
using Psilibrary.SpriteClasses;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

namespace ForestRush
{
    public class TransitionEventArgs : EventArgs
    {
        public string Animation;
    }

    public class Player : DrawableGameComponent
    {
        public event EventHandler<TransitionEventArgs> Transition;
        public Dictionary<string, FrameAnimatedSprite> Sprites { get; private set; } =
new();
        protected string _currentAnimation;
        protected readonly SpriteBatch _spriteBatch;

        public bool Flip { get; set; }

        public FrameAnimatedSprite Sprite
        {
            get
            {
                if (Sprites.ContainsKey(_currentAnimation))
                    return Sprites[_currentAnimation];
                else
                    return null;
            }
        }

        public string CurrentAnimation { get { return _currentAnimation; } }

        public Player(Game game) : base(game)
        {
            _spriteBatch = Game.Services.GetService<SpriteBatch>();
            _currentAnimation = "idle";
        }

        public virtual void LoadContent(ContentManager Content)
        {
            Sprites.Clear();

            try
            {
                string folder = string.Format("{0}/PlayerSprite/", Content.RootDirec-
tory);
                foreach (var f in Directory.GetDirectories(folder))
                {
                    string root = f.Replace("Content/", "");
                    string animation = root.Replace("PlayerSprite/", "");

                    List<Texture2D> list = new();
                    foreach (var r in Directory.GetFiles(f))
                    {
                        string path = Path.GetFileNameWithoutExtension(r);
```

```
                        StringBuilder build = new StringBuilder()
                            .Append(root)
                            .Append('/')
                            .Append(path);

                        list.Add(Content.Load<Texture2D>(build.ToString()));
                    }

                    FrameAnimatedSprite sprite = new(list);
                    Sprites.Add(animation.ToLower(), sprite);
                }
            }
            catch (Exception ex)
            {
            }
        }

        public override void Update(GameTime gameTime)
        {
            if (Sprites.Count == 0)
            {
                LoadContent(Game.Content);
                return;
            }

            Sprites[_currentAnimation].Update(gameTime);

            if (Sprites[_currentAnimation].CurrentFrame >= Sprites[_currentAnima-
tion].Frames && _currentAnimation.ToLower() != "walk" && _currentAnimation.ToLower() !=
"run")
            {
                OnTransition("idle");
                ChangeAnimation("idle");
            }
            else if (Sprites[_currentAnimation].CurrentFrame >= Sprites[_currentAnima-
tion].Frames)
            {
                OnTransition(_currentAnimation);
                ChangeAnimation(_currentAnimation);
            }

            base.Update(gameTime);
        }

        private void OnTransition(string animation)
        {
            Transition?.Invoke(this, new() { Animation = animation });
        }
        public override void Draw(GameTime gameTime)
        {
            if (Sprites.Count == 0) return;

            if (Flip)
                Sprites[_currentAnimation].Draw(_spriteBatch, Sprite.Position, SpriteEf-
fects.FlipHorizontally);
            else
                Sprites[_currentAnimation].Draw(_spriteBatch, Sprite.Position, SpriteEf-
fects.None);
```

```
            base.Draw(gameTime);
        }

        public void ChangeAnimation(string animation)
        {
            if (Sprites.ContainsKey(animation))
            {
                Sprites[animation].Position = Sprites[_currentAnimation].Position;
                _currentAnimation = animation;
                Sprites[_currentAnimation].Reset();
            }
        }

        public void Reset()
        {
            if (Sprite != null)
            {
                Sprite.Position = Vector2.Zero;
            }
        }
    }
}
```

There is a second class inside this file, TransitionEventArgs. It is here because there is an event that triggers when the animation changes. It has a single field, Animation, which is the name of the animation that was transitioned. The class itself inherits from DrawableGameComponent so it his methods like LoadContent, Draw, and Update.

As I mentioned, there is an event called Transition. Next, is a Dictionary<string, FrameAnimatedSprite> that holds all of the animations for the player. In the event that it needs to be inherited, the next fields are protected. They are the current animation and the sprite batch. There is a bool property, Flip, that says if the sprite should be flipped or not. Following that is a property that returns either null or the current sprite. The final property returns what the current animation is.

The constructor retrieves the sprite batch the same way that we did in the BaseGameState. It also sets the current animation to the idle animation.

The LoadContent method is an interesting method. The first step is to clear all of the sprites. Then there is a try-catch block because we will be working with the file system. First, we get the full path for the PlayerSprite folder. I could have just concatenated the strings together, but it is preferred to use either string interpolation or a method like string.Format. The fewer string instantiations, the better. If you see me using string concatenation, slap my hand. Now I get all of the sub-folders and iterate over them. I get what root folder is by replacing the content folder with the empty string. I then get what animation it is by replacing PlayerSprites with the empty string. I create a new List<Texture2> to hold the textures.

The next step is to retrieve the list of files and iterate over them. Following that, I get what the file name without extension is. It was a bit of over kill, but I used a string build to make the path to the XNB file. I then load it and add it to the list. After I have all of the images, I create a sprite. Once I have the sprite, I add it to the dictionary of sprites using the path set to lower.

While it should never happen, I check to see if there are no sprites. If that is true, I call the LoadContent method to load them. After loading them, I exit the method. If there are sprites, I call the update method of the current sprite. Next, I check to see if the current frame of the current animation is greater than or equal to the number of frames and that the animation is not walking or running. In that case, I want to transition to idle. I do that by first calling the OnTransition method to trigger the Transition event. I then call ChangeAnimation to change the animation to idle. That includes going from idle to idle. If it was walking or running, we want to keep doing that, so we call OnTransition to go from walking to walking or running to running and reset the animation.

OnTransition takes a string parameter which is the animation being transitioned to. If the event is subscribed to, it is invoked, passing in this for the sender a TransitionEventHandler with the Animation property being the animation.

If there are no sprites, I exit the method. I should have done like I did in the Update method, though, and loaded them. If the Flip property is true, I call the Draw method of the FrameAnimatedSprite and pass FlipHoriziontally to flip the sprite. Otherwise, I call it with no sprite effects.

The ChangeAnimation method is called from outside the class to change the animation. It received as a parameter the animation to be changed to. If that key exists in the dictionary, the Position of the sprite is set to the current sprite. If you don't do this, there will be what appears to be random teleportation when switching animations. Really had me scratching my head when that would happen in my latest game, Eyes of Inashti. Then, the current animation is set to the animation passed in, and the Reset method is called to reset the timer and current animation.

The Reset method checks to see if there is a sprite to work with. If there is, it sets the position of the sprite to the zero vector.

So, now we can turn our attention to the GamePlayState. I probably should have saved adding it to now because there were big changes. Replace the GamePlayState with the following code.

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

namespace ForestRushShared.GameStates
{
    public class GamePlayState : BaseGameState
    {
        private Player _player;
        private const float Gravity = 256;
        private Vector2 _motion;
        public GamePlayState(Game game) : base(game)
        {
        }

        public override void Initialize()
        {
```

```
            base.Initialize();
        }

        protected override void LoadContent()
        {
            if (_player== null)
            {
                _player = new(Game);
                _player.LoadContent(Content);
            }
            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            _player.Update(gameTime);
            _motion = new(0, 1);
            _motion.Y *= Gravity * (float)gameTime.ElapsedGameTime.TotalSeconds;

            _player.Sprite.Position += _motion;

            base.Update(gameTime);
        }

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

            base.Draw(gameTime);
            _player.Draw(gameTime);

            SpriteBatch.End();
        }

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

        public override void Show()
        {
            base.Show();

            if (_player == null)
            {
                LoadContent();
            }

        }
    }
}
```

First, I added a new field, _player of type Player. I also added a constant Gravity which is the pull downward on all objects. Next, the field _motion is the desired movement of the player. Currently, it will only be from gravity.

In the LoadContent method, I check to see if the _player field is null. If it is, I create a new Player object and assign it to the field. I then call the LoadContent method passing in the ContentManager.

The Update method is where the magic happens! First, it calls Update on the player object to update the animation. Eight frames per second might be a little slow. I may bump it up to five. We'll see. Next, the desired motion is set to down. It is then multiplied by Gravity times the total number of seconds that have happened since the last frame. The player's position is then updated by the desired motion.

In the Draw method, I call Begin on the sprite batch to begin rendering. After the call to base.Draw to draw the children. I draw the player's sprite. I call End to begin rendering.

In the Show method, if the _player field is null, I call the LoadContent method to create the player object.

I did discover a but during testing. That is, the SpriteBatch will be null if you add the state to the list of components in the constructor. You need to do that in the LoadContent method. Replace the code in the Game1 class with the following.

```
using ForestRushShared.GameStates;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace ForestRush
{
    public class Game1 : Game
    {
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;

        public GamePlayState GamePlayState { get; private set; }

        public Game1()
        {
            _graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
            IsMouseVisible = true;

            GamePlayState = new(this);
        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);
            Services.AddService<SpriteBatch>(_spriteBatch);
            // TODO: use this.Content to load your game content here

            Components.Add(GamePlayState);
            GamePlayState.Show();
        }
```

```
        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);
        }

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

            // TODO: Add your drawing code here

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

If you build and run now, the player's sprite will appear and fall off the screen. That is going to be it for this tutorial. There is more that I could pack in, but I will save it for the following tutorials. It would be best if you didn't have too much to digest at once. I encourage you to visit the news page of my site, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials. Also, I'm thinking of reviving my newsletter so you will be informed of new stuff rather than having to keep looking for new things.

Good luck with your game programming adventures!
Cynthia