Forest Rush – 3.8.1

Part 2

Then There Was Light

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 two 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.

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 for future tutorials.

Infrastructure, Again

There will almost always be a need for infrastructure in the tutorials. However, I'm going to try and add just little bits at a time so that I can focus more on gameplay. Even then, much of what is going on is related to infrastructure.

So, let's get started. We have game states but no mechanism for changing states. For that reason, I am going to add the state manager. This is something that I use in all of my games. It is fairly evolved, and I hardly ever make changes to it. Right-click the ForestRushShared project, select Add and then Class. Name this new class StateManager. The code for the state management follows next.

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

namespace ForestRushShared

{
    public interface IStateManager
    {
        BaseGameState CurrentState { get; }

        event EventHandler StateChanged;

        void PushTopMost(BaseGameState state);
        void PushState(BaseGameState state);
        void ChangeState(BaseGameState state);
        void PopState();
        void PopTopMost();
        bool ContainsState(BaseGameState state);
    }

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

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

        private const int startDrawOrder = 500;
        private const int drawOrderInc = 50;
        private const int MaxDrawOrder = 5000;

        private int drawOrder;

        #endregion

        #region Event Handler Region

        public event EventHandler StateChanged;

        #endregion

        #region Property Region

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

        #endregion

        #region Constructor Region

        public StateManager(Game game)
            : base(game)
        {
            Game.Services.AddService(typeof(IStateManager), this);
            drawOrder = startDrawOrder;
        }
        #endregion

        #region Method Region

        public void PushTopMost(BaseGameState state)
        {
            drawOrder += MaxDrawOrder;
            state.DrawOrder = drawOrder;
            gameStates.Push(state);
            Game.Components.Add(state);
            StateChanged += state.StateChanged;
            OnStateChanged();
        }

        public void PushState(BaseGameState state)
        {
            drawOrder += drawOrderInc;
            state.DrawOrder = drawOrder;
            AddState(state);
            OnStateChanged();
        }

        private void AddState(BaseGameState state)
        {
            gameStates.Push(state);
            if (!Game.Components.Contains(state))
                Game.Components.Add(state);
            StateChanged += state.StateChanged;
        }

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

        public void PopTopMost()
        {
            if (gameStates.Count > 0)
            {
                RemoveState();
                drawOrder -= MaxDrawOrder;
                OnStateChanged();
            }
        }

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

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

        public void ChangeState(BaseGameState state)
        {
            while (gameStates.Count > 0)
            {
                RemoveState();
            }

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

            AddState(state);
            OnStateChanged();
        }

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

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

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

        #endregion
    }
}
```

First, there is an interface that the state manager will implement. What exactly is an interface? I like to think of them as contracts. A contract that the class implementing them must follow. In this case, it must implement a property that returns a BaseGameState that is the current state. In addition, there is an event that must be implemented, StateChanged. This event is fired whenever the game state changes.

Next, there is a method PushTopMost that pushes a state onto the stack that is the topmost state. What do I mean by all of that? Game states are managed on a stack. You can think of a stack like plates. To add a state, place it on the pile or push it. For the topmost item, I ensure its draw order is the highest. Theoretically, the state on top of the stack will always have the highest draw order. This method makes double sure. There is also a method, PushState, that is essentially the same as PushTopMost. ChangeState removes all states from the stack and pushes the state passed onto the stack. Next comes PopState or PopTopMost. They remove the state from off of the pile or pop it. The final method is ContainsState and checks to ensure a state is not on the stack.

The class inherits from GameComponent and implements the IStateManager interface. In the **Event** region is the code for the event that I created that will be triggered when there is a change in state or screens. You saw the handler for the event in the **GameState** class, **StateChange**. To manage the states, I used a generic **Stack<GameState>**. A stack, in computer science, is a last-in-first-out data structure.

There are three integer fields: **startDrawOrder**, **drawOrderInc**, and **drawOrder**. These fields determine the order in which game screens are drawn. For example, **DrawableGameComponent** has a property called **DrawOrder**. Components will be drawn in ascending order according to their **DrawOrder** property. The component with the lowest draw order will be drawn first. The component with the highest **DrawOrder** property will be drawn last. I chose 5000 as a good starting point. When a screen is added to the stack, I will set its **DrawOrder** property higher than the previous screen. The **drawOrderInc** field is how much to increase or decrease when a screen is added or removed. **drawOrder** holds the current value of the screen on top. Finally, there is a public property, **CurrentScreen**, that returns which screen is on top of the stack.

The constructor of this class sets the **drawOrder** field to the **startDrawOrder** field. By choosing the magic numbers 5000 and 100 for the amount to increase or decrease, there is room for many screens. Far more than you will probably have on the stack at once.

In the **Methods** region, there are three public and two private methods. The public methods are **PopState**, **PushState**, and **ChangeState**. The private methods are **RemoveState** and **AddState**. **PopState** is the method to call if you have a screen on top of the stack that you want to remove and go back to the previous screen. An excellent example is opening an options menu from the main menu.

You want to go back to the main menu from the options menu, so you pop the options menu off the stack. **PushState** is the method to call if you want to move to another state and keep the previous state.

From above, you would push the options screen on the stack so that you can return to the previous state when you are done. Finally, the last public method, **ChangeState**, is called when you wish to remove all other
states from the stack.

The **PopState** method checks to make sure there is a game state to pop off by checking the **Count** property of the **gameStates** stack. It calls the **RemoveState** method that removes the state that is on the top of the stack. It then decrements the **drawOrder** field so that when the following screen is added to the stack of screens, it will be drawn appropriately. It then checks if **OnStateChange** is not **null**. What this means is that it checks to see if the **OnStateChange** event is subscribed to. If the event is not subscribed to, the event handler code should not be executed. It then calls the event handler code, passing itself as the sender and null for the event arguments.

The **RemoveState** method first gets which state is on top of the stack. It then unsubscribes the state from the subscribers to the **OnStateChange** event. It then removes the screen from the components in the game. It then pops the current state off the stack.

The **PushState** method takes the state to be placed on the top of the stack as a parameter. It first increases the **drawOrder** field and sets it to the **DrawOrder** property of the component so it will have the highest **DrawOrder** property. It then calls the **AddState** method to add the screen to the stack. It then checks to make sure the **OnStateChange** event is subscribed to before calling the event handler code.

The **AddState** method pushes the new screen on the top of the stack. Next, it adds it to the list of components of the game. Then, finally, it subscribes to the new state to the **OnStateChange** event.

The first thing the **ChangeState** method does is remove all of the screens from the stack. It then sets the **DrawOrder** property of the new screen to the **startDrawOrder** value and **drawOrder** to the same value. It then calls the **AddScreen** method passing in the screen to be changed. It then will call the **OnStateChange** event handler if it is subscribed to.

With the state manager class created, let's add it to the game. First, replace the code of the Game1 class with the following.

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

namespace ForestRush
{
    public class Game1 : Game
    {
        public const int BaseWidth = 1280;
        public const int BaseHeight = 720;
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;
        private StateManager _stateManager;

        public GamePlayState GamePlayState { get; private set; }

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


            GamePlayState = new(this);
            _stateManager = new(this);

            Services.AddService<StateManager>(_stateManager);
            Components.Add(_stateManager);
        }

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

            _graphics.PreferredBackBufferWidth = BaseWidth;
            _graphics.PreferredBackBufferHeight = BaseHeight;
            _graphics.ApplyChanges();

            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();
            Components.Add(GamePlayState);
            _stateManager.PushState(GamePlayState);
        }

        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);
        }
    }
}
```

First, I added a few constants: BaseWidth and BaseHeight. They are the base height and width at which we are programming the game. With the state manager built, let's add it to the game. That will happen in the Game1 class. Replace the code of the Game1 class with the following. What we are going to do is the following. The game will take place on 1280 by 720 window. The rendering and mechanics will happen at the resolution. We will use a render target to draw at the resolution. Then, we will flip that to the player's desired resolution. We will see that in practice in another tutorial. I also added a field for the state manager.

In the constructor, I create an instance of the state manager. It is then added as a service to the collection of services so that we can retrieve it later. I then add it to the list of components so its Update method will be called automatically.

In the Initialize method, I set the preferred back buffer width to the base resolution and the preferred back buffer height to the base resolution. Then, I call ApplyChanges to apply the changes. So, what is a back buffer, and why should you care? Well, back buffers are used to prevent flickering. If you draw directly to the screen, you may render when a screen refresh occurs. To prevent that, you render to a back buffer, which is just a collection of integers that define the pixels to be rendered. Once you have rendered the back buffer, flip it to the window simultaneously.

In the LoadContent method, I call the Show method of the GamePlayState to display it. I then add it to the state manager. After that, the rest of the class runs as before.

If you build and run now, the player's sprite will appear and fall off the screen. The falling is desired, but it should remain on the screen. It would be cool if it could run and jump as well. To handle running well walking in this tutorial, we need to add the content for walking. In the ForestRushShared project, open the Content folder to reveal the Content.mgcb node. Double-click the Content.mgcb to open the MGCB Editor window. Open the PlayerSprite folder. Right-click the PlayerSprite folder, select Add and then New Folder. Name this new folder Walking. Right-click the Walking folder, select Add and then Existing Folder. Navigate to the Fallen_Angels_2 folder in the graphics downloaded earlier in the lesson. Go into the PNG folder and then the PNG Sequences folder. Select the Walking folder. When prompted, select the Add link option instead of copying the asset. Save and close the editor.

So, let's confine the sprite to the screen. To do that, we must check that its Position property remains on the screen. Sounds easy enough. Just clamp it between 0 and the base height. Then, replace the code for the Update method of the GamePlayState with the following.

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

    if ((_player.Sprite.Position + _motion).Y < Game1.BaseHeight)
    {
        _player.Sprite.Position += _motion;
    }

    base.Update(gameTime);
}
```

The sprite still falls off the screen if you build and run now. What did we do wrong? Well, we didn't take into account the height of the sprite. We need to subtract that from the base height in order to keep it on the screen. Replace the Update method with the following code.

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

    if ((_player.Sprite.Position + _motion).Y < Game1.BaseHeight - 128)
    {
        _player.Sprite.Position += _motion;
    }

    base.Update(gameTime);
}
```

It is the same as the previous version. The difference is that I subtract 128 from the screen height. I do 128 because that is the height and width I render. The texture itself is 900 pixels.
In order to move the sprite ourselves, we need to gather input. I'm going to add a very stripped-down

input handler. We will do most of the work there. Right-click the ForestRushedShared project, select Add and then Class. Name this new class Xin. Here is the code for that class. In case you're curious, why Xin? It comes from my XNA days, and I created it as XNA Input Handler. It was way too long, so it was abbreviated to Xin.

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

namespace ForestRushShared
{
    public class Xin : GameComponent
    {
        private static KeyboardState currentKeyboardState = Keyboard.GetState();
        private static KeyboardState previousKeyboardState = Keyboard.GetState();

        public static KeyboardState KeyboardState
        {
            get { return currentKeyboardState; }
        }

        public static KeyboardState PreviousKeyboardState
        {
            get { return previousKeyboardState; }
        }

        public static bool IsKeyDown(Keys key)
        {
            return currentKeyboardState.IsKeyDown(key);
        }

        public Xin(Game game)
            : base(game)
        {
            TouchPanel.EnableMouseTouchPoint = true;
        }

        public override void Update(GameTime gameTime)
        {
            Xin.previousKeyboardState = Xin.currentKeyboardState;
            Xin.currentKeyboardState = Keyboard.GetState();

            base.Update(gameTime);
        }

        public static void FlushInput()
        {
            currentKeyboardState = previousKeyboardState;
        }

        public static bool CheckKeyReleased(Keys key)
        {
            return currentKeyboardState.IsKeyUp(key) && previousKeyboardState.Is-
KeyDown(key);
        }
        public static bool IsKeyUp(Keys key)
        {
            return KeyboardState.IsKeyUp(key);
        }
    }
}
```

A short and sweet version for our present uses. I will be adding to it as we go. First, it inherits from GameComponent so it can be added to the list of components for the game and have its Update method called automatically. You don't have to worry about it!

There are two static fields currentKeyboardState and previousKeyboardState which hold the state of the keyboard in this frame of the game and the last frame of the game. Why the previous frame? Why static? Well, glad you asked. In order to tell if a key has been pressed rather than just down, you need to know the state in the last frame of the game to compare to. For example, to tell if a key has been released since the previous frame of the game, you check to see if it is up now but down in the last frame. You will see it in action in the CheckKeyReleased method.

There at two properties, KeyboadState and PreviousKeyboardState, that return the state of the keyboard in this frame of the game and the last frame of the game, respectively. I also included a method that crept up from where it should be IsKeyDown which lets us know if a key is down. Finally, we can ignore the line in the constructor about touch. I will get back to it when I add Android and iOS support.

In the Update method, I set the previousKeyboardState to the currentKeyboardState so that it holds the last frame and then update the current keyboard state. I call base.Update to make sure that any children will be updated. FlushInput sets the previous keyboard state to the current keyboard state, which makes them equal and calls to check if a key was released or pressed failed. With MonoGame 3.8, I needed help with this working. The CheckKeyReleased method checks to see if a key has been released, as described earlier. Finally, the IsKeyUp method checks to see if a key is up.

Let's move the sprite! First, we will handle lateral movement. Then, update the Update method of the GamePlayState class as follows.

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

    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
    {
        _motion.X = -1;
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
    {
        _motion.X = 1;
    }

    _motion.X *= Speed * (float)gameTime.ElapsedGameTime.TotalSeconds;
    _player.Sprite.Position += _motion;

    _player.Sprite.Position = new(
        MathHelper.Clamp(_player.Sprite.Position.X, 0, Game1.BaseWidth - 128),
        MathHelper.Clamp(_player.Sprite.Position.Y, 0, Game1.BaseHeight - 128));

    base.Update(gameTime);
}
```

What the new code does is check to see if either the A or D keys are pressed. If the A key is pressed, I tell the game that the player wants to move left. Similarly, if the D key is down, I tell the game that the player wants to move right. Next, I multiply the desired X movement by a new constant that I haven't added yet and the elapsed seconds. What that does is have the sprite move at the same rate, no matter how many frames the player is getting. I then update the sprite's position by the motion. I then use the MathHelper.Clamp method to restrict the X value of the sprite's position between zero and the width of the screen minus 128. I do something similar for the Y value of the sprite's position and the screen's height.

Two problems. One, you don't have the constant. Two, we aren't gathering input. Yes, we declared Xin, but it needs to be added to the list of game components. So, first, the constant, add this constant with the other fields and constants in the GamePlayState.

```
        private const float Speed = 128;
```

On to problem number two. That will be handled in the constructor of the Game1 class. Replace the constructor of the Game1 class with the following.

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

    Components.Add(new Xin(this));

    GamePlayState = new(this);
    _stateManager = new(this);

    Services.AddService<StateManager>(_stateManager);
    Components.Add(_stateManager);
}
```

Well, that's an improvement. There is vertical movement and lateral movement. The issue is that there is no walk animation. Worse, the sprite is always facing right. Let's fix that. Replace the Update method of the GamePlayState with this version.

```
public override void Update(GameTime gameTime)
{
    _player.Update(gameTime);
    _motion = new(0, 1);
    _motion.Y *= Gravity * (float)gameTime.ElapsedGameTime.TotalSeconds;
    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
    {
        _motion.X = -1;
        _player.Flip = true;

        if (_player.CurrentAnimation != "walking")
        {
            _player.ChangeAnimation("walking");
        }
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
    {
        _motion.X = 1;
        _player.Flip = false;

        if (_player.CurrentAnimation != "walking")
        {
            _player.ChangeAnimation("walking");
        }
    }

    _motion.X *= Speed * (float)gameTime.ElapsedGameTime.TotalSeconds;


    _player.Sprite.Position += _motion;

    _player.Sprite.Position = new(
        MathHelper.Clamp(_player.Sprite.Position.X, 0, Game1.BaseWidth - 128),
        MathHelper.Clamp(_player.Sprite.Position.Y, 0, Game1.BaseHeight - 128));

    base.Update(gameTime);
}
```

When I set the X movement to left or right, I check to see if the current animation is not walking. If not, I change the current animation to walking. If it is walking left, I set Flip to true. Otherwise, it is set to false.

There is one thing bugging me watching the sprite on the screen. The animation should be shorter. We need to shorten it by quite a bit. Replace the constructor of the FrameAnimatedSprite with this new version.

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

    _frameLength = TimeSpan.FromSeconds(1.0 / 20.0);
}
```

The sprite animates at a decent frame rate if you build and run now. There are two things left for this tutorial. One is that I promised you jumping, and two, the sprite moves much faster on diagonals. Before coding, let's talk a little about theory. When anything goes up, it has to break the bonds of gravity. It needs sufficient push/force or impulse. So, our sprite is being dragged down constantly. We need to apply not equal but greater force to move up. Then, gravity will take care of pulling it back down for us. The most frequent target for jumping is the space bar, so that we will go with that. First, add an impulse constant and field with the other fields and constants of the GamePlayState.

```
private const float Impulse = -640;
private float _impulse = 0.0f;

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

    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
    {
        _motion.X = -1;

        if (_player.CurrentAnimation != "walking")
        {
            _player.ChangeAnimation("walking");
            _player.Flip = true;
        }
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
    {
        _motion.X = 1;

        if (_player.CurrentAnimation != "walking")
        {
            _player.ChangeAnimation("walking");
            _player.Flip = false;
        }
    }

    if (Xin.CheckKeyReleased(Microsoft.Xna.Framework.Input.Keys.Space) && _impulse >= 0)
    {
        _impulse = Impulse;
        _player.ChangeAnimation("jumpstart");
    }

    if (_impulse < 0)
    {
        _motion.Y += _impulse * (float)gameTime.ElapsedGameTime.TotalSeconds;
        _impulse += Gravity * 2 * (float)gameTime.ElapsedGameTime.TotalSeconds;
    }
    else
    {
        _motion.Y += Gravity * (float)gameTime.ElapsedGameTime.TotalSeconds;
    }

    _motion.X *= Speed * (float)gameTime.ElapsedGameTime.TotalSeconds;


    _player.Sprite.Position += _motion;

    _player.Sprite.Position = new(
        MathHelper.Clamp(_player.Sprite.Position.X, 0, Game1.BaseWidth - 128),
        MathHelper.Clamp(_player.Sprite.Position.Y, 0, Game1.BaseHeight - 128));
    base.Update(gameTime);
}
```

So, there is a new constant, Impulse, that is, the initial trust when the player starts the jump. It is set to a value that will give a decent jump. There is also a field, _impulse, that is, the amount of thrust left in the jump. In the Update method, I check to see if the space bar has been released. If it has, I set the _impulse field to the initial thrust. Then, if _impulse is less than zero, I set _motion.Y to _impulse.Y times the amount of time since the last update, in seconds. And, I bleed off some of the thrust. Otherwise, I update _motion.Y to be Gravity times the elapsed time in seconds, as before.

If you build and run now, the player's sprite will jump if you release the space bar. It is buggy, though. The jump should happen when the space bar is down, not when it is released. That brings us back to handling input. Sometimes it is better to handle an event the moment that a key is down. Other times, it is better to handle releases. When a state changes, it is much better to handle jumps, attacks, firing a weapon, etc. It is better to handle things like clicking a button or selecting a menu item when the key has been released. So, let's add a new method to Xin for detecting a new press.

```
public static bool CheckKeyPressed(Keys key)
{
    return currentKeyboardState.IsKeyDown(key) && previousKeyboardState.IsKeyUp(key);
}
```

The new method checks to see if a key is now down but was up in the previous frame—the complete opposite of the released method. Now, let's bring it all together in the GamePlayState. This is what the entire method looks like.

We are close, but we are not quite there. There is just a matter of the animation. We need to handle the jump animation a bit better. What we want to do is a transition from the jumpstart to the jumploop. Then, ideally, we will transition to idle. That is done in the Update method of the Player class. Replace it with the following.

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

    Sprites[_currentAnimation].Update(gameTime);

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

    base.Update(gameTime);
}
```

So, it checks to see if the current animation is not walking and not running and that the current animation is at its end. Then, if the animation is jumpstart, it transitions to jumploop. So, if you build and run now, the player's sprite jumps pretty coolly.

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