Forest Rush – 3.8.1

Part 3

Not Walking on Air

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 three of a series of tutorials I plan to write on creating a platformer with MonoGame. While 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.

Tile Engine

There are two choices when it comes to rendering a level. First, you can create the level in software like Photoshop as one big huge image and render each frame. The other alternative is to use many small repeated images, or tiles, and draw those. I will be taking the latter approach. I will use a different method of having a two-dimensional array of tiles. Instead, I will have a dictionary of tiles with rectangles for keys. The rectangle's X and Y components will act as the X and Y coordinates of a 2D array. This will save memory and improve the number of times we must loop through the tiles.

Tile Frame

What we need to render is a class that represents the tiles. They are individual images that are logically combined into a tile frame or tile set. First, right-click the Psilibrary project in the Solution Explorer, select Add and then New Folder. Name this new folder TileEngine. Now, right-click the TileEngine folder, select Add and then Class. Name this new class TileFrames. Here is the code for that class.

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

namespace Psilibrary.TileEngine
{
    public class TileFrame
    {
        public string Name { get; set; }
        public Texture2D Texture { get; set; }

        private TileFrame()
        {

        }

        public TileFrame(string name, Texture2D texture)
        {
            Name = name;
            Texture = texture;
        }
    }

    public class TileFrames
    {
        public List<TileFrame> Frames { get; private set; } = new();

        public void LoadContent(ContentManager Content)
        {
            Frames.Clear();

            try
            {
                string folder = string.Format("{0}/Tiles", Content.RootDirectory);
                string root = Content.RootDirectory;

                foreach (var r in Directory.GetFiles(folder))
                {
                    string path = Path.GetFileNameWithoutExtension(r);

                    StringBuilder build = new StringBuilder()
                        .Append("Tiles/")
                        .Append(path);

                    TileFrame frame = new(
                        Path.GetFileNameWithoutExtension(r),
```

```
                        Content.Load<Texture2D>(build.ToString()));
                    Frames.Add(frame);
                }
            }
            catch(Exception ex)
            {
            }
        }
    }
}
```

There are two related classes in this file. The first, TileFrame, comprises a name and an image or texture. There is a private constructor that takes no parameters. It is required when we will start building maps in the editor and save them to disk and load them back in. There is also a public constructor that takes the name of the tile and the texture of the tile.

The second class is the logical combination of the tiles into one unit. It is a list of ti frames. It has a single method that reads all of the tiles in a folder into the list. The method requires a content manager to load the compiled resources. You need to get the folder the tiles are in to do that. You also need to know the root directory, which is the content manager's RootDiretctory property. You could just use that rather than assigning it to a variable. You then iterate over all of the files in the folder of interest. First, you get the name of the file without its extension. Next, you build the relative path to the XNB file without its extension. Now, you create a TileFrame using the name and the texture. Finally, the frame is added to the collection of frames.

Map Layer

Maps will be made up of layers. For that reason, I created an interface that will be used to combine the different layers into a map logically. The two layers I will implement in this tutorial are the tile and collision layers. I will add other layer types as we go.

Right-click the TileEngine folder in the Psilibrary project, select Add and then New Item. Scroll down until you find Interface. Name this new interface ILayer. This is the code for that interface.

```
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.TileEngine
{
    public interface ILayer
    {
        int Width { get; }
        int Height { get; }
        bool Enabled { get; set; }
        bool Visible { get; set; }
        void Update(GameTime gameTime);
        void Draw(SpriteBatch spriteBatch, Camera camera);
    }
}
```

It is a relatively straightforward interface. It is made up of four properties. The width of the layer, the height of the layer, if the layer is enabled and if the layer is visible. You also must implement two methods, Update and Draw.

Now, let's implement a tile layer. Right-click the TileEngine folder in the Psilibrary project, select Add and then class. Name this new class TileLayer. This is the code for that class.

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

namespace Psilibrary.TileEngine
{
    public class TileLayer : ILayer
    {
        #region Field Region

        [ContentSerializer]
        public Dictionary<Rectangle, int> Tiles { get; private set; } = new();

        private readonly TileFrames _tileSet = new();

        int width;
        int height;

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

        #endregion

        #region Property Region

        public bool Enabled { get; set; }

        public bool Visible { get; set; }

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

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

        #endregion
```

```
#region Constructor Region

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

public TileLayer(int width, int height, TileFrames frames)
    : this()
{
    Tiles = new();

    foreach (TileFrame frame in frames.Frames)
    {
        _tileSet.Frames.Add(frame);
    }

    this.width = width;
    this.height = height;
}

#endregion

#region Method Region

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

    if (x >= width || y >= height)
        return -1;
    Rectangle tile = new(x, y, Engine.TileWidth, Engine.TileHeight);

    if (Tiles.ContainsKey(tile))
        return Tiles[tile];

    return -1;
}

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

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

    Point location = new(x, y);

    if (Tiles.ContainsKey(Helpers.PointToTile(location)))
    {
        Tiles[Helpers.PointToTile(location)] = tile;
    }
    else
    {
        Tiles.Add(Helpers.PointToTile(location), tile);
    }
}
```

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

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

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

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

            foreach (Rectangle r in Tiles.Keys.Where(
                x => x.X >= min.X && x.X <= max.X && x.Y >= min.Y && x.Y <= max.Y))
            {
                spriteBatch.Draw(
                    _tileSet.Frames[Tiles[r]].Texture,
                    r,
                    null,
                    Color.White);
            }
        }

        #endregion
    }
}
```

The class implements the ILayer interface so it can be added to the list of layers on the map. As I mentioned, tiles are stored in a Dictionary<Rectangle, int> where Rectangle is where the tile should be rendered, and it is the tile's index in the tile frames. There is also a TileFrames object for the tiles on the layer. Next, there are width and height fields. The next four fields are used in rendering. They are a window of the map that is only what is visible on the screen, plus or minus one tile. Finally, some properties expose the width and height of the map.

There is a private constructor that initializes the Enabled and Visible properties to true. There is a second constructor that takes as parameters the width of the layer and the height of the layer as well as a TileFrames object. First, it initializes the Tiles property. It then loops over the frames passed in and adds them to this collection, copying it. It then sets the width and height fields to the width and height passed in.

There is a method that retrieves the tile at a given X and Y coordinate. If either the X or Y property is less than zero, I return -1—the same if they are greater than the width or height of the map. I then create a rectangle describing the tile we are trying to retrieve. If that key is in the dictionary, I return

the tile. Otherwise, I return minus one.

The SetTile method is used to set a tile. It takes as arguments the x coordinate, the y coordinate and the tile index. If the coordinates are less than zero, I exit the method. Then, if either coordinate is greater than or equal to the width or height of the layer, I exit the method. Currently, the Update method checks if the Enabled property is false and exits the method.

The Draw method is where the magic happens. It takes the required SpriteBatch and Camera parameters. If the layer is not visible, it exits the method. To determine where to start drawing, you take the camera position as a point using the Engine class that we will get to shortly—the same about the camera. Next, add the viewport's width and height to determine where to stop rendering. The minimum point is the maximum of zero, and the camera point is minus one. Why minus one? Well, it is for overflow. To find where to stop drawing, you take the minimum height and width of the map and the viewpoint plus one, again because of overflow. Next is a foreach loop that loops over the keys collection. There is a LINQ where clause on the loop. It checks if the tile is inside the minimum and maximum range.

Tile Map

Next, we need a class for the map. The map will have a foreground layer and a background layer. Sprites will be drawn between the background layer and the foreground layer. Right-click the TileEngine folder, select Add and then Class. Name this class TileMap. Here is the code of the class.

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

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

        string mapName;

        [ContentSerializer]
        public List<ILayer> Background { get; private set; } = new();

        [ContentSerializer]
        public List<ILayer> Foreground { get; private set; } = new();

        int mapWidth;
        int mapHeight;

        TileFrames tileSet;

        #endregion

        #region Property Region

        [ContentSerializer(Optional = true)]
        public string MapName
```

```
{
    get { return mapName; }
    private set { mapName = value; }
}

[ContentSerializer]
public TileFrames TileSet
{
    get { return tileSet; }
    set { tileSet = value; }
}

[ContentSerializer]
public int MapWidth
{
    get { return mapWidth; }
    private set { mapWidth = value; }
}

[ContentSerializer]
public int MapHeight
{
    get { return mapHeight; }
    private set { mapHeight = value; }
}

public int WidthInPixels
{
    get { return mapWidth * Engine.TileWidth; }
}

public int HeightInPixels
{
    get { return mapHeight * Engine.TileHeight; }
}

#endregion

#region Constructor Region

private TileMap()
{
}

private TileMap(TileFrames tileSet, string mapName)
    : this()
{
    this.tileSet = tileSet;
    this.mapName = mapName;
}

public TileMap(
    TileFrames tileSet,
    List<ILayer> backgroundLayers,
    List<ILayer> foregroundLayers,
    string mapName)
    : this(tileSet, mapName)
{
    int maxHeight = 0;
    int maxWidth = 0;
```

```
            foreach (var layer in backgroundLayers)
            {
                if (layer.Width > maxWidth) maxWidth = layer.Width;
                if (layer.Height > maxHeight) maxHeight = layer.Height;

                Background.Add(layer);
            }

            foreach (var layer in foregroundLayers)
            {
                if (layer.Width > maxWidth) maxWidth = layer.Width;
                if (layer.Height > maxHeight) maxHeight = layer.Height;

                Foreground.Add(layer);
            }

            mapWidth = maxWidth;
            mapHeight = maxHeight;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            foreach (ILayer layer in Background)
            {
                if (layer.Enabled)
                {
                    layer.Update(gameTime);
                }
            }

            foreach (ILayer layer in Foreground)
            {
                if (layer.Enabled)
                {
                    layer.Update(gameTime);
                }
            }
        }

        public void DrawBackground(GameTime gameTime, SpriteBatch spriteBatch, Camera
camera, bool debug = false)
        {
            if (WidthInPixels >= Engine.TargetWidth || debug)
            {
                spriteBatch.Begin(
                    SpriteSortMode.Deferred,
                    BlendState.AlphaBlend,
                    SamplerState.PointClamp,
                    null,
                    null,
                    null,
                    camera.Transformation);
            }
            else
            {
```

```
                Matrix m = Matrix.CreateTranslation(
                    new Vector3((Engine.TargetWidth) / 2, (Engine.TargetHeight -
HeightInPixels) / 2, 0));
                spriteBatch.Begin(
                    SpriteSortMode.Deferred,
                    BlendState.AlphaBlend,
                    SamplerState.PointClamp,
                    null,
                    null,
                    null,
                    m);
            }

            foreach (ILayer layer in Background)
            {
                if (layer.Visible)
                {
                    layer.Draw(spriteBatch, camera);
                }
            }

            spriteBatch.End();
        }

        public void DrawForeground(GameTime gameTime, SpriteBatch spriteBatch, Camera
camera, bool debug = false)
        {
            if (WidthInPixels >= Engine.TargetWidth || debug)
            {
                spriteBatch.Begin(
                    SpriteSortMode.Deferred,
                    BlendState.AlphaBlend,
                    SamplerState.PointClamp,
                    null,
                    null,
                    null,
                    camera.Transformation);
            }
            else
            {
                Matrix m = Matrix.CreateTranslation(
                    new Vector3((Engine.TargetWidth) / 2, (Engine.TargetHeight -
HeightInPixels) / 2, 0));
                spriteBatch.Begin(
                    SpriteSortMode.Deferred,
                    BlendState.AlphaBlend,
                    SamplerState.PointClamp,
                    null,
                    null,
                    null,
                    m);
            }

            foreach (ILayer layer in Foreground)
            {
                if (layer.Visible)
                {
                    layer.Draw(spriteBatch, camera);
                }
            }
```

```
            spriteBatch.End();
        }

        #endregion
    }
}
```

The map has a name field. It also has a list of foreground layers and background layers. It has a width and a height. The foreground and background fields are marked as ContentSerializer attributes so that they will be written using the intermittent serializer. What is the intermittent serializer? It takes a class and writes its public fields and properties to an XML document. The last field is the TileFrames field. There are properties to expose the private members. There are also properties to return the width and height of the map.

There are a few constructors in the class. The first is private with no parameters. The intermediate serializer requires it. The second was public, but I made it private for now. It takes for parameters the name of the map and the tile set. The last constructor takes a tile set, a list of background layers, foreground layers and the map name. It sets some local variables that hold the maximum height and width of the map. All layers start at the same origin (0, 0) but can have different heights and widths. This should work in theory, but I haven't tested it in practice yet.

The next step is to loop over all of the layers in the list of background layers passed in. If the layer width is greater than the maximum width, the maximum width is set to that layer's width. The same holds true for heights. The layer is then added to the list of background layers. There is a similar loop for the foreground. The mapWidth and mapHeight are then set to the maximum values.

In the Update method, I loop over the layers in the foreground and the background layers. If the layer is enabled, I call its Update method.

I split the rendering into two methods, one for the background and one for the foreground. They take a GameTime, SpriteBatch and Camera parameter and an optional debug parameter. Inside, it checks to see if the map's width in pixels is greater than or equal to the target width of the engine. If it is, I call Begin passing in parameters for the sort mode, blend state, sampler state, and several nulls because I want the default and a transformation matrix. What the…? I hear you asking. I want sprites sorted after End is called on the sprite batch, so I pass in deferred. I want to use transparency, so I use AlphaBlend. The next one prevents tearing of the tiles when they are rendered on partial pixels. How is that even possible? When you use transformation matrices and Vector2s, they have floating point values. So, things are rendered on partial pixels. Sampling the textures using PointClamp prevents bleeding into the neighbouring tiles. It is probably redundant because our tiles are in different textures. I'm used to rendering with tiles all in the same texture. Finally, the transformation matrix. In algebra, you can move, rotate, and scale points using matrices. You can do the same with MonoGame. So, the camera is the current view of the map. Rather than moving the map, you move the camera. Transforming it points to where to do the rendering. You must subtract the camera's position from where you want to start drawing. If you don't, the map will move in the opposite direction. If the map width is less than the screen width and the debug flag is false, I create a transformation matrix that centers the map on the screen, and I call Begin of the sprite batch using

the same parameters as before, other than the transformation matrix.
After all that, I loop over the layers in the background tiles. If the layer is Visible, I call its draw method passing in the sprite batch and camera. Finally, I call the End method of the sprite batch to do the drawing. I do the same for the foreground. You could optimize the update and render loops using LINQ instead of the if statements. With a game, it would be a good idea. I leave that to you as an exercise.

Camera

We've seen the camera mentioned a few times now. It is an abstraction that allows us to render the portion of the map that is visible to the player at the current point in time. First, add a class to the TileEngine folder called Camera. Next, replace the boilerplate code with the following.

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

namespace Psilibrary.TileEngine
{
    public class Camera
    {
        #region Field Region

        Vector2 position;
        float speed;

        #endregion

        #region Property Region

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

        public float Speed
        {
            get { return speed; }
            set { speed = (float)MathHelper.Clamp(speed, 1f, 16f); }
        }

        public Matrix Transformation
        {
            get
            {
                return Matrix.CreateTranslation(new Vector3(-Position, 0f));
            }
        }

        #endregion

        #region Constructor Region

        public Camera()
```

```
        {
            speed = 4f;
        }

        public Camera(Vector2 position)
        {
            speed = 4f;
            Position = position;
        }

        #endregion


        public void LockCamera(TileMap map)
        {
            position.X = MathHelper.Clamp(position.X,
                0,
                map.WidthInPixels - Engine.ViewportRectangle.Width);

            position.Y = MathHelper.Clamp(position.Y,
                0,
                map.HeightInPixels - Engine.ViewportRectangle.Height);
        }
    }
}
```

There are two fields in this class: position and speed. Speed is used when we want to move the camera independently of the player, and position is where we want to render the map. There are public properties to expose the values of the fields. There is also the transformation matrix. It is calculated by subtracting the camera's position and multiplying the position by minus one.

There are two constructors. The first initializes the speed property. Honestly, four is fast. It will do for now, however. The second constructor initializes the speed field and sets the position to the value passed.

There is one method in the class, LockCamera, that locks the camera to the map, keeping it from going off the map. What it does is use the MathHelper.Clamp method to bind the coordinates between minimum and maximum values. The minimum is zero for both coordinates. The maximum varies depending on the coordinate. For the X coordinate, you take the width of the map in pixels and subtract the width of the screen, not the width of the map in pixels. If you don't subtract the width of the screen, you will scroll off the edge of the map. The Y coordinate is the same, other than using the height of the map and screen.

Engine

The engine is a collection of methods and properties that define how the map is rendered. Add a new class Engine to the TileEngine folder. Replace the boilerplate code with the following.

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

namespace Psilibrary.TileEngine
{
    public static class Engine
```

```
{
    #region Field Region
    private static Rectangle viewPortRectangle;

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

    private static Camera camera;

    public const int TargetWidth = 1280;
    public const int TargetHeight = 720;

    #endregion

    #region Property Region

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

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

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


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

    #endregion

    #region Constructors

    static Engine()
    {
        ViewportRectangle = new Rectangle(0, 0, 1280, 720);
        camera = new Camera();

        TileWidth = 64;
        TileHeight = 64;
    }

    #endregion

    #region Methods

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

```
        }

        public static Vector2 VectorFromOrigin(Vector2 origin)
        {
            return new Vector2((int)origin.X / tileWidth * tileWidth, (int)origin.Y /
tileHeight * tileHeight);
        }

        public static void Reset(Rectangle rectangle, int x, int y)
        {
            Engine.viewPortRectangle = rectangle;
            Engine.TileWidth = x;
            Engine.TileHeight = y;
        }

        internal static Point PointToCell(Point p)
        {
            return new(p.X / TileWidth, p.Y / TileHeight);
        }

        #endregion
    }
}
```

This is a static class, so we do not create an instance directly. We access the static members. The first member is a rectangle that describes the screen. Next are integers that are the width and height of tiles on the screen. There is a Camera as well. Finally, are two constants that are the target width and height of the screen. We are going to target a base width and height of 1280 by 720 pixels. All calculations and rendering will be done using these values. If the game is being run at a higher resolution or lower resolution, it will be scaled to those values. There are properties to expose all of the fields. The constructor sets the fields and properties to default values.

There are a few methods. The first it VectorToCell. It takes a Vector2 and returns what cell it is in measured in tiles. The next is a rarely used method that got copied over. It creates a Vector2 based off the origin passed in. Reset is used to set the engine to new values. The PointToCell method is similar to the VectorToCell method. It just works with a point instead of a vector.

Collision Layer

The last tile engine class that I am going to implement today is a collision layer. The idea of a collision layer is that it tells where the player can and cannot go. Any game object, for that matter. Right-click the TileEngine folder, select Add and then Class. Name this new class CollisionLayer. Replace the boilerplate code with the following.

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

namespace Psilibrary.TileEngine
{
    public enum CollisionType
```

```
    {
        CollisionUp,
        CollisionDown,
        CollisionRight,
        CollisionLeft,
        CollisionAll,
        Damage,
        Death
    }

    public class CollisionLayer
    {
        [ContentSerializer]
        public Dictionary<Rectangle, CollisionType> Collisions { get; private set; } =
new();

        public CollisionLayer()
        {
        }
    }
}
```

There is an enumeration that defines the different types of collisions. There is one for each cardinal direction, plus all four directions. There is one that causes damage and one that causes death. There is a single property that has been marked with the ContentSerializer attribute, so it will be rendered to the XML document. There is an explicit constructor that is empty.

World

There is one last class that I am going to implement for the tile engine. That is a class that represents the world. The world is a collection of maps and other data. Right-click the TileEngine folder, select Add and then Class. Name this new class World. Here is the code for that class.

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

namespace Psilibrary.TileEngine
{
    public static class World
    {
        public static string CurrentMap { get; private set; }
        public static Dictionary<string, TileMap> Maps { get; private set; } = new();
        public static TileMap TileMap
        {
            get
            {
                if (Maps.ContainsKey(CurrentMap))
                    return Maps[CurrentMap];
                else
                    return null;
            }
        }
```

```
        static World()
        {
        }

        public static void ChangeMap(string mapName)
        {
            if (!Maps.ContainsKey(mapName)) return;

            CurrentMap = mapName;
        }

        public static void Load(ContentManager content, Game game)
        {
            TileFrames tileSet = new();
            tileSet.LoadContent(content);

            List<ILayer> layers = new();

            TileLayer groundLayer = new(100, 25, tileSet);

            for (int x = 0; x < 100; x++)
            {
                int y = 24;

                groundLayer.SetTile(x, y, 14);
            }

            layers.Add(groundLayer);

            List<ILayer> foreground = new();

            TileMap map = new(tileSet, layers, foreground, "Test");

            CollisionLayer collisionLayer = new();

            for (int x = 0; x < 100; x++)
            {

                collisionLayer.Collisions.Add(
                    new(
                        new(x * Engine.TileWidth, 24 * Engine.TileHeight),
                        new(Engine.TileWidth, Engine.TileHeight)),
                    CollisionType.CollisionDown);
            }

            World.Maps.Add("Test", map);
            World.ChangeMap("Test");
        }
    }
}
```

This is also a static class that you don't create an instance of. Instead, you access it through public members. It has three static properties. The first, CurrentMap, returns the name of the current map. The next its Maps and is a dictionary with string keys and tile map values. As you probably guessed, it holds all of our maps. Finally, there is a property that returns the current map.

There are two static methods. The first is used the change between maps. The other is used to load maps. Since we haven't created an editor, it just creates one programmatically. First, it creates a

TileFrames to hold the tiles. Then, it uses the LoadContent method to read in the tiles. Next, it creates a list of layers for the background. Following that, it creates a map that is twenty-five tiles high and one hundred tiles wide. It then creates a row of tiles at the bottom of the map. The layer is added to the list of layers for the background. I then create a layer for the foreground tiles that I haven't populated yet. Next, I create a map. I created a collision layer after that, but I didn't attach it to the map. The map is added to the list of maps, and I changed the map to the new map.

Rendering a Map

So, there is a problem. The ForestRushShared project needs to know about Psilibrary, but Psilibrary needs to know about ForestRushShared. Specifically, we need to be able to lock the camera to the sprite. One solution would be to move the FrameAnimatedSprite and related classes to Psilibrary simply. That is an option. However, I went a different route. I added a reference of Psilibrary to ForestRushShared, and I added an extension method for the Camera class to the ForestRushShared library. So, let's do that. Right-click the ForestRushShared project, select Add and then Class. Name this new class ExtensionMethods. Replace the boilerplate code of the ExtensionMethods class with the following.

```
using Psilibrary.SpriteClasses;
using Psilibrary.TileEngine;
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Text;

namespace ForestRushShared
{
    public static class ExtensionMethods
    {
        public static void LockToSprite(this Camera camera, FrameAnimatedSprite sprite,
TileMap map)
        {
                camera.Position = new((sprite.Position.X + 128 / 2)
                    - (Engine.TargetWidth / 2), (sprite.Position.Y + 128 / 2)
                    - (Engine.TargetHeight / 2));

                camera.LockCamera(map);
        }
    }
}
```

The LockToSprite method takes three parameters, a camera, a sprite and a map. The camera is prefixed with this, making it an extension method. Extension methods are methods that are called like you were calling them on the class. So, we can call camera.LockToSprite(…). It is the way we are going to resolve the circular dependency. We will be adding more extension methods as we go to add new functionality to existing classes.

What the method actually does is take the position of the sprite and add half of 128 to it, the subtract half the width of the screen. Wait, why 128? Why subtract half the width of the screen? What about the Y coordinate? Well, the Y coordinate is the Y coordinate of the sprite plus half of 128 minus half

the height of the screen. Why 128? Well, we are rendering our sprite at 128 by 128, so we use those for half the width and height of the sprite. I use those because the actual texture is ginormous. It is 900 pixels wide, and quite tall. I had to scale it to fit it on the screen. Finally, why subtract half the height and width of the screen? This makes the map scroll with the sprite. We only want it to scroll once it is half the way across the screen. The final thing the method does is call the LockToMap method to look the camera to the map.

There are a few pieces left to the puzzle. The easiest one is to add a reference to Psilibrary to the ForestRushShared project. Right-click the Dependencies node of ForestRushShared and select Add Reference. From the list, choose Project Reference. Add a reference to Psilibrary.

Next, we need to add a method to the FrameAnimatedSprite class to lock the sprite to the map. Add the following method to the FrameAnimatedSprite class.

```
        public void LockToMap(Point mapSize)
        {
            Position = new(
                MathHelper.Clamp(Position.X, 0, mapSize.X - 128),
                MathHelper.Clamp(Position.Y, 0, mapSize.Y - 128));
        }
```

This method clamps the X coordinate of the sprite between zero and the width of the map minus 128 and the Y coordinate between zero and the height of the map minus 128 pixels. I would normally use the height and width of the sprite, but it is huge, so I am using the height and width on the screen.

That leaves updating the position of the camera and rendering the map. That will take place on the GamePlayState. Let's tackle rendering first. Replace the Draw method of the GamePlayState with this new version.

```
public override void Draw(GameTime gameTime)
{
    World.TileMap.DrawBackground(gameTime, SpriteBatch, Engine.Camera);

    SpriteBatch.Begin(
        SpriteSortMode.Deferred,
        BlendState.AlphaBlend,
        SamplerState.PointClamp,
        null,
        null,
        null,
        Engine.Camera.Transformation);

    _player.Draw(gameTime);

    SpriteBatch.End();

    World.TileMap.DrawForeground(gameTime, SpriteBatch, Engine.Camera);
}
```

What I did was call rendering the background layer. I draw the player's sprite after a call to Begin passing the transformation matrix of the camera. I then call the draw method of the foreground layer.

That brings us to the Update method of the GamePlayState. Replace it with this new version.

```
public override void Update(GameTime gameTime)
{
    _player.Update(gameTime);
    World.TileMap.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.CheckKeyPressed(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.LockToMap(
        new Point(
            World.TileMap.WidthInPixels,
            World.TileMap.HeightInPixels));

    Engine.Camera.LockToSprite(_player.Sprite, World.TileMap);

    base.Update(gameTime);
```

```
}
```

It is mainly the same. What has changed is that I removed the code that locked the sprite to the screen. I replaced it with a call to the LockToMap method that I added previously. I then call the LockToSprite extension method. As you can see, it is just like calling a native method. If you build and run now, the sprite will fall from the top of the screen. It will appear to get stuck, but after a moment, the tiles at the bottom of the map will appear. The sprite will be stuck right in the middle. You can run and jump around the screen like before, but the camera will follow you.

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 considering 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