Eyes of the Dragon Tutorials 4.0

Part 6

Keep on Trekking

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 list of tutorials on the Eyes of the Dragon 4.0 page of my web blog. 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 six of a series of tutorials I plan to write on creating a role-playing game with MonoGame. I've worked on similar tutorials using XNA and the past version of MonoGame. In the process of writing more tutorials, I discovered better ways of doing some things and had to go back to fix things. I'm hoping in this series to avoid making those same 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, we have the game up and running. A blank map is being drawn, and a sprite for the player to control. That is all fine and good, but pretty useless. You can't do anything with it. So, I'm going to implement movement in this tutorial. I've thought about it a lot, actually, and I've decided to implement tile-based movement. What is the reason? Well, it makes collision-based detection against the environment a lot easier when it comes to objects.

Anyway, let's get started. Mobile platforms do not have keyboards or a mouse to gather input from. You have to rely on touch-based input. So, how are we going to handle movement? The answer is through a virtual D-pad. How are we going to create a virtual D-pad? Why through the use of some GUI elements? Namely, buttons. So, we need two things. We need a button class, and we need some buttons. Fortunately, we have easy access to both. For the button textures, I am going to use some buttons from OpenGameArt.org. They are by looneybits. You can download the textures from this link.

Okay, once you have downloaded and extracted the texture, they need to be added to the project. They will go into the SharedProject. So, in the SharedProject, open the Content folder and double-click the Content.mgcb node to open the MGCB Editor. Right-click on the root folder, select Add and then New Folder. Name this new folder GUI. Now, right-click the GUI folder, choose Add and then Existing Item. Navigate to where you extract the files, select all of them and them to the project. We probably won't be using all of them, but it is nice to have the option.

As I said, we need a button for the virtual D-pad. In essence, a button is just a link label with a background image attached to it. I will be separating them into two classes, however. So, right-click the Controls folder, select Add and then Class. Name this new class Button. The code for that class follows next.

```
    public enum ButtonRole { Accept, Cancel, Menu }
    public class Button : Control
    {
        #region

        public event EventHandler Click;
        public event EventHandler Down;

        #endregion
        #region Field Region

        private readonly Texture2D _background;
        float _frames;

        public ButtonRole Role { get; set; }
        public int Width { get { return _background.Width; } }
        public int Height { get { return _background.Height; } }

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public Button(Texture2D background, ButtonRole role)
        {
            Role = role;
            _background = background;
            Size = new(background.Width, background.Height);
            Text = "";
        }

        #endregion

        #region Method Region

        public override void Draw(SpriteBatch spriteBatch)
        {
            Rectangle destination = new(
            (int)Position.X,
            (int)Position.Y,
            (int)Size.X,
            (int)Size.Y);

            spriteBatch.Draw(_background, destination, Color.White);

            _spriteFont = ControlManager.SpriteFont;

            Vector2 size = _spriteFont.MeasureString(Text);
            Vector2 offset = new((Size.X - size.X) / 2, ((Size.Y - size.Y) / 2));

            spriteBatch.DrawString(_spriteFont, Text, Helper.NearestInt((Position + off-
set)), Color);
        }

        public override void HandleInput()
        {
            MouseState mouse = Mouse.GetState();
            Point position = new(mouse.X, mouse.Y);
```

```
Rectangle destination = new(
    (int)(Position.X),
    (int)(Position.Y),
    (int)(Size.X),
    (int)(Size.Y);

if ((Role == ButtonRole.Accept && Xin.WasKeyReleased(Keys.Enter)) ||
    (Role == ButtonRole.Accept && Xin.WasKeyReleased(Keys.Space)))
{
    OnClick();
    return;
}

if (Role == ButtonRole.Cancel && Xin.WasKeyReleased(Keys.Escape))
{
    OnClick();
    return;
}

if (Xin.WasMouseReleased(MouseButton.Left) && _frames >= 5)
{
    Rectangle r = destination.Scale(Settings.Scale);

    if (r.Contains(position))
    {
        OnClick();
        return;
    }
}

if (Xin.TouchReleased() && _frames >= 5)
{
    Rectangle rectangle= destination.Scale(Settings.Scale);

    if (rectangle.Contains(Xin.TouchReleasedAt))
    {
        OnClick();
        return;
    }
}

if (Xin.IsMouseDown(MouseButton.Left))
{
    Rectangle rectangle = destination.Scale(Settings.Scale);

    if (rectangle.Contains(Xin.MouseAsPoint))
    {
        OnDown();
        return;
    }
}

if (Xin.TouchLocation != new Vector2(-1, -1))
{
    Rectangle rectangle = destination.Scale(Settings.Scale);

    if (rectangle.Contains(Xin.TouchLocation))
    {
        OnDown();
        return;
```

```
                }
            }
        }

        private void OnDown()
        {
            Down?.Invoke(this, null);
        }

        private void OnClick()
        {
            Click?.Invoke(this, null);
        }

        public override void Update(GameTime gameTime)
        {
            _frames++;
            HandleInput();
        }

        public void Show()
        {
            _frames = 0;
        }

        #endregion
    }
```

Sorry for the wall of code. It is better than feeding it to you bit by bit. When it comes to the GamePlayState, I will provide it to you piece by piece. So, what is going on here?

Well, first off, there is an enumeration called ButtonRole. This enumeration defines if a button has a role. Roles are Accept, Cancel and Menu. If a button has the Accept role and the enter key is pressed, it will activate its Click event. Similarly, if a button has the Cancel role and the escape key is pressed, its Click event will be fired. The Menu role is for regular buttons; I should name it None.

Buttons have two events. The first event is Click, and it is fired when the button is clicked or tapped. The second event is Down, and it is triggered while the mouse is down over it or it is being pressed on a touch screen.

The _background field is the texture for the button. When I say texture, I could have said image. The terms are synonymous. The next field is _frames, and it counts the number of frames since a button was clicked. This is because, at times, input handling can stutter. It will repeat over a couple of frames. So, I added the to skip a few before triggering the Click event a second time.

There are a few properties. The first is Role and is the role of the button. The next are Width and Height, which are the width and height of the button, respectively.

The constructor has two arguments, the background texture and the role of the button. It initializes the Role property, the background field, the Size property and the Text property. I included the last one because if you didn't set it explicitly in your code, an exception would occur when you ran your game. So it was better to include it here.

In the Draw method, I create a destination rectangle for the button using the Position and Size properties. I do not want to use the Width and Height properties. That is because they are based on the size of the texture, and I want to be able to scale buttons. I then draw the background texture. Following that, I grab the font for the button. I measure the size of the Text property and then center it over the button. Finally, I render the text.

The first thing I do in the HandleInput method is grab the state of the mouse. I then create a destination rectangle for the button. Using the position and the size. It is important to use the size instead of the texture. Otherwise, you risk activating events that you don't mean to. I check to see if the Role of the button is the Accept role and if the Enter or Space keys have been released. If they have, I call the OnClick method to trigger the event handler and exit the method. Similarly, if the Escape key has been released and the Role of the button is Cancel, I call the OnClick method and exit the method. Next, I check to see if the mouse has been released and if at least five frames have passed to prevent stuttering. If the conditions I true, I call the Scale method of the Rectangle class passing in the scale difference from the Settings class. That method and Scale property I have not added yet. If the rectangle contains the mouse, I call the OnClick method and exit the method. Then if the touch has been released, I do the same thing. I pass in the property TouchReleasedAt, which I haven't added yet. If the mouse is down and it is inside the destination for the button, I call the OnDown method to trigger that event handler. Finally, if the TouchLocastion is over the button, I call the OnDown method.

The OnDown method checks to see if the Down event is subscribed to. If it is, it Invokes the event handler. The same is true for the OnClick method. It checks to see if the event is subscribed to, and if it is, it invokes the event handler.

So, there are a lot of broken things. I will address them one at a time. First, let's add the extension method and helper. Extension methods all you to extend the functionality of a class. In our case, we want to be able to scale rectangles in order to handle different resolutions. Also, I added a helper that rounds a Vector2 to the nearest integers, essentially a point. Add the following classes to the SharedProject.

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

namespace SharedProject
{
    public static class ExtensionMethods
    {
        public static Rectangle Scale(this Rectangle rect, Vector2 scale)
        {
            return new Rectangle(
                (int)(rect.X * scale.X),
                (int)(rect.Y * scale.Y),
                (int)(rect.Width * scale.X),
                (int)(rect.Height * scale.Y));
        }

    }
}

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

namespace SharedProject
{
    public static class Helper
    {
        public static Vector2 NearestInt(Vector2 vector2)
        {
            return new((int)vector2.X, (int)vector2.Y);
        }
    }
}
```

Next is the Scale property of the Settings class. To find what the scale ratio is in the settings class, you take the target width and divide it by the base width and the target height divided by the base height. Replace the Settings class with the following.

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

namespace SharedProject
{
    public class Settings
    {
        public const int BaseWidth = 1280;
        public const int BaseHeight = 720;

        public static Rectangle BaseRectangle { get { return new(0, 0, BaseWidth, Base-
Height); } }
        public static Rectangle TargetRectangle { get { return new(0, 0, TargetWidth,
TargetHeight); } }
        public static int TargetWidth { get; set; } = BaseWidth;
        public static int TargetHeight { get; set; } = BaseHeight;
        public static Vector2 Scale
        {
            get { return new((float)TargetWidth / BaseWidth, (float)TargetHeight / Base-
Height);}
        }
    }
}
```

Following that, I updated the ControlManager's Update method. I removed the code that handled GamePad input. Don't worry, I will add it back in when I have a gamepad to do testing with. Replace the Update method of the ControlManager with the following.

```
        public void Update(GameTime gameTime)
        {
            if (Count == 0)
                return;
            foreach (Control c in this)
            {
                if (c.Enabled)
                {
                    c.Update(gameTime);
                }

                if (c.HasFocus)
                {
                    c.HandleInput();
                }
            }

            if ((Xin.IsKeyDown(Keys.LeftShift) || Xin.IsKeyDown(Keys.RightShift))
                && Xin.WasKeyPressed(Keys.Tab))
            {
                PreviousControl();
            }

            if (Xin.WasKeyPressed(Keys.Tab))
            {
                NextControl();
            }
        }
```

Following that are the HandleInput methods of the Label and LinkLabel. I will start with the Label class, because it is trivial. I just removed the parameter for PlayerIndex. Replace that method with the following.

```
        public override void HandleInput()
        {
        }
```

For the Update method of the LinkLabel I did the same. Replace the HandleInput method of the LinkLabel class with the following code.

```
        public override void HandleInput()
        {
            if (!HasFocus)
            {
                return;
            }

            if (Xin.WasKeyReleased(Keys.Enter))
            {
                base.OnSelected(null);
                return;
            }

            if (Xin.WasMouseReleased(MouseButtons.Left))
            {
                size = SpriteFont.MeasureString(Text);

                Rectangle r = new(
                (int)Position.X,
                    (int)Position.Y,
                    (int)size.X,
                    (int)size.Y);

                if (r.Contains(Xin.MouseAsPoint))
                {
                    base.OnSelected(null);
                }
            }
        }
```

That leaves the Update method of the GameState class. All I did was remove passing the PlayerIndex argument to the control manager in the Update method. Replace the Update method of the control manager with the following.

```
        public override void Update(GameTime gameTime)
        {
            foreach (GameComponent component in childComponents)
            {
                if (component.Enabled)
                    component.Update(gameTime);
            }

            ControlManager.Update(gameTime);
            base.Update(gameTime);
        }
```

There is one last change to be made. That is to Xin. I added a property to detect touch releases. Add the following property to Xin.

```
        public static Vector2 TouchReleasedAt
        {
            get
            {
                Vector2 result = Vector2.Zero - Vector2.One;

                if (touchLocations.Count > 0)
                {
                    if (touchLocations[0].State == TouchLocationState.Released)
                    {
                        result = touchLocations[0].Position;
                    }
                }

                return result;
            }
        }
```

What it does is create a variable that is just outside of the screen to rule out a touch being released at exactly (0, 0). It then checks to see if there have been any touch events. If there have, it checks to see if the first event was a release. If it was, it returns the position. Finally, it returns the result.

Now, everything is in place to tackle moving the player's sprite and scrolling the map. It is a lot of code, so I will break up feeding it to you. First, we need to add some fields. We need a field for each of the buttons and some fields to track the motion of the sprite. Add the following fields to the

GamePlayState.

```
        private Button upButton, downButton, leftButton, rightButton;
        private bool inMotion = false;
        private Rectangle collision = new();
        private float speed;
        private Vector2 motion;
```

There are several new fields. First, there are four Button fields, one for each direction. Next, there is a field that tells if the sprite is in motion or not. Following that is a rectangle that the player's sprite will collide with once it has reached the end of the next tile. Finally, there is a new field, speed, that is, the speed of the sprites in pixels per second.

In the Initialize method, I set the base value of the speed field. I set the speed of the sprite to 96, which is 96 pixels per second. Okay, time for the LoadContent method. In the LoadContent method, I create the instances of the buttons, add an event handler, and add them to the control manager. Replace the LoadContent method with the following.

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

    renderTarget = new(GraphicsDevice, Settings.BaseWidth, Settings.BaseHeight);

    Texture2D texture = Game.Content.Load<Texture2D>(@"Tiles/tileset1");

    List<Tileset> tilesets = new()
    {
        new(texture, 8, 8, 32, 32),
    };

    TileLayer layer = new(100, 100);

    map = new("test", tilesets[0], layer);

    Dictionary<string, Animation> animations = new();

    Animation animation = new(3, 32, 32, 0, 0) { CurrentFrame = 0, FramesPerSecond = 8 };
    animations.Add("walkdown", animation);

    animation = new(3, 32, 32, 0, 32) { CurrentFrame = 0, FramesPerSecond = 8 };
    animations.Add("walkleft", animation);

    animation = new(3, 32, 32, 0, 64) { CurrentFrame = 0, FramesPerSecond = 8 };
    animations.Add("walkright", animation);

    animation = new(3, 32, 32, 0, 96) { CurrentFrame = 0, FramesPerSecond = 8 };
    animations.Add("walkup", animation);

    texture = Game.Content.Load<Texture2D>(@"PlayerSprites/femalepriest");

    sprite = new(texture, animations)
    {
        CurrentAnimation = "walkdown",
        IsActive = true,
        IsAnimating = true,
    };
    rightButton = new(Game.Content.Load<Texture2D>("GUI/g21245"), ButtonRole.Menu)
    {
        Position = new(80, Settings.BaseHeight - 80),
        Size = new(32, 32),
        Text = "",
        Color = Color.White,
    };

    rightButton.Down += RightButton_Down;
    ControlManager.Add(rightButton);

    upButton = new(Game.Content.Load<Texture2D>("GUI/g21263"), ButtonRole.Menu)
    {
        Position = new(48, Settings.BaseHeight - 48 - 64),
        Size = new(32, 32),
        Text = "",
        Color = Color.White,
    };

    upButton.Down += UpButton_Down;
    ControlManager.Add(upButton);

    downButton = new(Game.Content.Load<Texture2D>("GUI/g21272"), ButtonRole.Menu)
    {
        Position = new(48, Settings.BaseHeight - 48),
        Size = new(32, 32),
        Text = "",
        Color = Color.White,
    };

    downButton.Down += DownButton_Down;
    ControlManager.Add(downButton);

    leftButton = new(Game.Content.Load<Texture2D>("GUI/g22987"), ButtonRole.Menu)
    {
        Position = new(16, Settings.BaseHeight - 80),
        Size = new(32, 32),
        Text = "",
        Color = Color.White,
    };

    leftButton.Down += LeftButton_Down;

    ControlManager.Add(leftButton);
}
```

Taking the left button as an example, I load the texture that points left and assign the button the Menu role. I might change that to None at some point. I then positioned the button 16 pixels from the left and 80 pixels from the bottom. As I believe I mentioned earlier, they are 32 by 32 pixels in height and width. I explicitly set the text to the empty string and the text colour to white. I then wire the event handler and add it to the control manager. Other than the positions, the other buttons are the same.

In the event handlers, I check to see if the sprite is in motion, and if it is not, I call a method to move the sprite. I call the same method from the Update method. Add these eight methods to the class and change the update method to the following as well.

```
private void LeftButton_Down(object sender, EventArgs e)
{
    if (!inMotion)
    {
        MoveLeft();
    }
}

private void MoveLeft()
{
    motion = new(-1, 0);
    inMotion = true;
    sprite.CurrentAnimation = "walkleft";
    collision = new(
        (sprite.Tile.X - 2) * Engine.TileWidth,
        sprite.Tile.Y * Engine.TileHeight,
        Engine.TileWidth,
        Engine.TileHeight);
}

private void RightButton_Down(object sender, EventArgs e)
{
    if (!inMotion)
    {
        MoveRight();
    }
}

private void MoveRight()
{
    motion = new(1, 0);
    inMotion = true;
    sprite.CurrentAnimation = "walkright";
    collision = new(
        (sprite.Tile.X + 2) * Engine.TileWidth,
        sprite.Tile.Y * Engine.TileHeight,
        Engine.TileWidth,
        Engine.TileHeight);
}

private void DownButton_Down(object sender, EventArgs e)
{
    if (!inMotion)
    {
        MoveDown();
    }
}

private void MoveDown()
{
    motion = new(0, 1);
    Point newTile = sprite.Tile + new Point(0, 2);
    inMotion = true;
    sprite.CurrentAnimation = "walkdown";
    collision = new(
        newTile.X * Engine.TileWidth,
        newTile.Y * Engine.TileHeight,
        Engine.TileWidth,
        Engine.TileHeight);
}

private void UpButton_Down(object sender, EventArgs e)
{
    if (!inMotion)
    {
        MoveUp();
    }
}

private void MoveUp()
{
    motion = new(0, -1);
    inMotion = true;
    sprite.CurrentAnimation = "walkup";
    collision = new(
        sprite.Tile.X * Engine.TileWidth,
        (sprite.Tile.Y - 2) * Engine.TileHeight,
        Engine.TileWidth,
        Engine.TileHeight);
}

public override void Update(GameTime gameTime)
{
    ControlManager.Update(gameTime);

    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A) && !inMotion)
    {
        MoveLeft();
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D) && !inMotion)
    {
        MoveRight();
    }

    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.W) && !inMotion)
    {
        MoveUp();
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.S) && !inMotion)
    {
        MoveDown();
    }

    if (motion != Vector2.Zero)
    {
        motion.Normalize();
    }
    else
    {
        inMotion = false;
        return;
    }

    if (!sprite.LockToMap(new(99 * Engine.TileWidth, 99 * Engine.TileHeight), ref mo-
tion))
    {
        inMotion = false;
        return;
    }

    Vector2 newPosition = sprite.Position + motion * speed * (float)ga-
meTime.ElapsedGameTime.TotalSeconds;

    Rectangle nextPotition = new Rectangle(
        (int)newPosition.X,
        (int)newPosition.Y,
        Engine.TileWidth,
        Engine.TileHeight);

    if (nextPotition.Intersects(collision))
    {
        inMotion = false;
        motion = Vector2.Zero;
        sprite.Position = new((int)sprite.Position.X, (int)sprite.Position.Y);
        return;
    }
    sprite.Position = newPosition;
    sprite.Tile = Engine.VectorToCell(newPosition);

    camera.LockToSprite(sprite, map);

    sprite.Update(gameTime);

    base.Update(gameTime);
}
```

So, the event handlers are pretty straightforward. They check to see if the sprite is in motion already. If it is not, they call a move method in the appropriate direction. That method will also be called if the key for that direction is down in the Update method.

I will dissect the MoveLeft method and leave the others up to you. They are pretty much the same, the only difference is direction. In the MoveLeft method, I set the motion to (-1, 0), which is moving left across the screen. I set the inMotion property to true so that no further movement will take place until the sprite has stopped moving. I set the animation for the sprite to the walk-left animation. Here is an exciting piece. To test when to stop moving, I create a rectangle that is two tiles to the left of the current tile the sprite is in. So, the sprite will walk through the first tile and then stop when it reaches the outer edges of the collision rectangle. I've seen people calculating velocities, scaling those, calculating timing, and other methods for doing this. This method is much cleaner.

The Update method calls the Update method of the control manager. Then, in the checks, if a key is down, it also checks to make sure the sprite is not in motion. If the key is down and the sprite is not in motion, I call the appropriate Move method. Now, if the motion vector is the zero vector, I set inMotion to false and exit the method. I then calculate the new position of the sprite if it can move in that direction. I then create a rectangle based on its new position. If it collides with the collision rectangle, I set inMotion to false, the motion vector to the zero vector, and, most importantly, cast the position to the nearest integer. This doesn't matter so much on desktop, but it is vital on iOS and Android. If you don't cast them to integers, the floating point builds up, and eventually, you won't be able to move down or right. I then calculate the tile the player's sprite is in using the VectorToCell method.

So, there are two elements missing. There is a missing method on the Camera class, LockToSprite, that will attach the camera to a sprite. The other is the Draw method. Essentially what happens in the Draw method is that after calling End but before calling resetting the render target, you need to draw the control manager. You do it that way, or they will be locked to the camera and will scroll off the screen. Update the Draw method to the following code.

```
public override void Draw(GameTime gameTime)
{
    GraphicsDevice.SetRenderTarget(renderTarget);
    GraphicsDevice.Clear(Color.Black);

    SpriteBatch.Begin(SpriteSortMode.Immediate,
                      BlendState.AlphaBlend,
                      SamplerState.PointClamp,
                      null,
                      null,
                      null,
                      Matrix.Identity);

    map.Draw(gameTime, SpriteBatch, camera);
    sprite.Draw(SpriteBatch);

    SpriteBatch.End();

    SpriteBatch.Begin();

    base.Draw(gameTime);

    SpriteBatch.End();

    GraphicsDevice.SetRenderTarget(null);

    SpriteBatch.Begin(SpriteSortMode.Immediate,
                      BlendState.AlphaBlend,
                      SamplerState.PointClamp);

    SpriteBatch.Draw(renderTarget, Settings.TargetRectangle, Color.White);

    SpriteBatch.End();
}
```

That leaves the LockToSprite method on the Camera class. There is a bit of a problem there. The problem is that the Sprite and AnimatedSprite classes are in SharedProject and Camera is in RpgLibrary. Moving the sprite classes is too much work. So, what is the solution? Well, the solution is to create an interface in the RpgLibary that has the properties that we need. We then implement the interface in the base class. That way, any time we need a property of the sprite we can pass the interface. Don't have a property we need? Just add it to the interface.

So, right-click the RpgLibrary project, select Add and then New Folder. Name this new folder Sprites. Now, right-click the Sprites folder in the RpgLibrary, select Add and then New Item. Scroll down to the

Interface entry and select it. Name this new interface ISprite. Here is the code for that interface.

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

namespace RpgLibrary.Sprites
{
    public interface ISprite
    {
        Vector2 Position { get; set; }
        Vector2 Size { get; set; }
        int Width { get; }
        int Height { get; }
    }
}
```

Pretty straightforward. There are two get-and-set accessors. One for the position and one for the size. There are also get-only accessors for the width and height of the screen. That leaves two pieces of the puzzle to solve. The first is we need to apply the interface to the Sprite class in the SharedProject. The other is the LockToSprite method. First, the Sprite class. Update that code to the following.

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
using System;
using System.Collections.Generic;
using System.Text;
using RpgLibrary.Sprites;

namespace SharedProject.Sprites
{
    public enum Facing { Up, Down, Left, Right }

    public abstract class Sprite : ISprite
    {
        protected float _speed;
        protected Vector2 _velocity;

        public Facing Facing { get; set; }
        public string Name { get; set; }
        public Vector2 Position { get; set; }
        public Vector2 Size { get; set; }
        public Point Tile { get; set; }

        public int Width { get; set; }
        public int Height { get; set; }

        public float Speed
        {
            get { return _speed; }
            set { _speed = MathHelper.Clamp(_speed, 1.0f, 400.0f); }
        }

        public Vector2 Velocity
        {
            get { return _velocity; }
            set { _velocity = value; }
        }

        public Vector2 Center
        {
            get { return Position + new Vector2(Width / 2, Height / 2); }
        }

        public Vector2 Origin
        {
            get { return new Vector2(Width / 2, Height / 2); }
        }

        public Sprite()
        {

        }

        public abstract void Update(GameTime gameTime);
        public abstract void Draw(SpriteBatch spriteBatch);
    }
}
```

There were two changes. The first is implementing the ISprite interface. As part of implementing that interface, I had to adjust the Height property from float to int.

Closing in on the prize now. Add the following method to the Camera class.

```
public void LockToSprite(ISprite sprite, TileMap map)
{
    _position.X = (sprite.Position.X + sprite.Width / 2)
        - (Engine.ViewportRectangle.Width / 2);

    _position.Y = (sprite.Position.Y + sprite.Height / 2)
        - (Engine.ViewportRectangle.Height / 2);

    LockCamera(map);
}
```

If you build and run now, you can move the character around the map, and the camera follows it. Well, that is what is supposed to happen, but it doesn't. The thing is we are still using the Identity matrix when rendering the map. We need to build an actual transformation matrix, namely a translation matrix. What that means, is for each pixel we want to move it X pixels on the direction and Y pixels in the other. This is based off the camera, so I added to properties to the camera class. Update it as follows.

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

```
namespace RpgLibrary.TileEngine
{
    public class Camera
    {
        #region Field Region

        private Vector2 _position;
        private Rectangle _viewportRectangle;

        public Vector3 Translation { get { return new(-_position, 0); } }
        public Matrix Transformation { get { return Matrix.CreateTranslation(Transla-
tion); } }

        #endregion

        #region Property Region

        public Point PositionAsPoint
        {
            get { return new Point((int)_position.X, (int)_position.Y); }
        }

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

        public Rectangle ViewportRectangle
        {
            get
            {
                return new Rectangle(
                _viewportRectangle.X,
                _viewportRectangle.Y,
                _viewportRectangle.Width,
                _viewportRectangle.Height);
            }
        }

        #endregion

        #region Constructor Region

        public Camera(Rectangle viewportRect)
        {
            _viewportRectangle = viewportRect;
        }

        public Camera(Rectangle viewportRect, Vector2 position)
        {
            _viewportRectangle = viewportRect;
            Position = position;
        }

        #endregion

        #region Method Region
        public void SnapToPosition(Vector2 newPosition, TileMap map)
        {
            _position.X = newPosition.X - _viewportRectangle.Width / 2;
            _position.Y = newPosition.Y - _viewportRectangle.Height / 2;
            LockCamera(map);
        }

        public void LockToSprite(ISprite sprite, TileMap map)
        {
            _position.X = (sprite.Position.X + sprite.Width / 2)
                - (Engine.ViewportRectangle.Width / 2);

            _position.Y = (sprite.Position.Y + sprite.Height / 2)
                - (Engine.ViewportRectangle.Height / 2);

            LockCamera(map);
        }

        public void LockCamera(TileMap map)
        {
            if (map != null)
            {
                _position.X = MathHelper.Clamp(_position.X,
                    0,
                    map.WidthInPixels - _viewportRectangle.Width);
                _position.Y = MathHelper.Clamp(_position.Y,
                    0,
                    map.HeightInPixels - _viewportRectangle.Height);
            }
        }

        #endregion
    }
}
```

There are two new properties: Translation and Transformation. Translation is a Vector3 and is how many pixels in each direction we want to translate. It is counterintuitive, but you need to subtract the position of the camera. Transformation is an actual translation matrix. Now, let's fix the Draw method in the GamePlayState.

```
public override void Draw(GameTime gameTime)
{
    GraphicsDevice.SetRenderTarget(renderTarget);
    GraphicsDevice.Clear(Color.Black);

    SpriteBatch.Begin(SpriteSortMode.Immediate,
                      BlendState.AlphaBlend,
                      SamplerState.PointClamp,
                      null,
                      null,
                      null,
                      camera.Transformation);

    map.Draw(gameTime, SpriteBatch, camera);
    sprite.Draw(SpriteBatch);

    SpriteBatch.End();
    SpriteBatch.Begin();

    base.Draw(gameTime);

    SpriteBatch.End();

    GraphicsDevice.SetRenderTarget(null);

    SpriteBatch.Begin(SpriteSortMode.Immediate,
                      BlendState.AlphaBlend,
                      SamplerState.PointClamp);

    SpriteBatch.Draw(renderTarget, Settings.TargetRectangle, Color.White);

    SpriteBatch.End();
}
```

There, that's much better. When the player gets halfway across the screen in any direction, the camera will start moving with them. If they get halfway to the end, it stops moving.

If you build and run now, you will be prompted to tap to move on to the next screen. Once there, you will find the tile map with the sprite on top. I wanted to cover movement in this tutorial. So, I am going to end this here. There is more that I could pack in, but I will save it for the following tutorials. I don't want you to 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.

Good luck with your game programming adventures!
Cynthia