Eyes of the Dragon Tutorials
Part 4
Tile Engine – Part One

I'm writing these tutorials for the MonoGame 3.8 framework using Visual Studio 2019. 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 page of my web blog. I will be making each version of the project available on GitHub here. It will be included on the page that links to the tutorials.

In this tutorial I will be continuing on with the tile engine. At the moment the map is being drawn but you need to be able to scroll the map so the player can explore your world. A good way to control scrolling of the map is to use a 2D camera. The camera shows what the player is looking at in the world. Right click the **TileEngine** folder in the **XRpgLibrary** project, select **Add** and then **Class**. Name this new class **Camera**. This is the code for the **Camera** class.

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

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

        Vector2 position;
        float speed;
        float zoom;
        Rectangle viewportRectangle;

        #endregion
        #region Property Region

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

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

        public float Zoom
        {
            get { return zoom; }
        }
```

```
        #endregion

        #region Constructor Region

        public Camera(Rectangle viewportRect)
        {
            speed = 4f;
            zoom = 1f;
            viewportRectangle = viewportRect;
        }

        public Camera(Rectangle viewportRect, Vector2 position)
        {
            speed = 4f;
            zoom = 1f;
            viewportRectangle = viewportRect;
            Position = position;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            if (InputHandler.KeyDown(Keys.Left))
                position.X -= speed;
            else if (InputHandler.KeyDown(Keys.Right))
                position.X += speed;
            if (InputHandler.KeyDown(Keys.Up))
                position.Y -= speed;
            else if (InputHandler.KeyDown(Keys.Down))
                position.Y += speed;
        }

        #endregion
    }
}
```

This camera is a different from cameras that I've created in my other tutorials. It is going to allow the player to zoom in and out and it is responsible for determining its position in the world. I'm also thinking of allowing the player to be able to control the camera to view the world but when they move their character the camera will snap back to the character. If I do go that route I will also be implementing a "fog of war" that the player can only see areas they've discovered.

There are using statements to bring the MonoGame Framework and MonoGame Framework Graphics classes into scope. There are four fields in the class. The first, **position**, is the position of the camera on the map and is a **Vector2**. There are two float fields: **speed** and **zoom**. The **speed** field controls the speed at which the camera moves through the world. The **zoom** field controls the zoom level of the camera. The last field is a **Rectangle** field, **viewportRectangle**, that describes the view port that the tile engine will draw to.

There are three properties in the **Camera** class. The **Position** property returns the position of the camera in the world. There is a private set to the property. The **Speed** property exposes the **speed**

field.

The set part uses the **MathHelper.Clamp** method to clamp the speed between 1 and 16. The last property, **Zoom**, is used to return the zoom level of the camera.
There are two constructors for the **Camera** class. The first takes a **Rectangle** that describes the view port the tile engine will be drawn to. The second takes the same rectangle and **Vector2** for the position of the camera. They both set the speed of the camera to 4 pixels and the zoom to 1. A zoom of 1 is a map with no zoom at all. Less than one zooms in, showing more of the map. Greater than one zooms out, showing less of the map. Both set the rectangle for the screen as well. The constructor that takes a **Vector2** for the position sets the **position** field.

There is also a method **Update** that takes a **GameTime** parameter used to update the camera's position in the world. There are a couple if statements. The first checks to see if the left arrow key is down using the **InputHandler** class. If it is it decreases the **X** property of the camera's position by the camera's speed. In an else-if I check if the right arrow key is down. If it is I increase the camera's **X** property.

The reason you decrease to move the camera left is that the values of X increase as you move from left to right across the screen. There is another if statement that checks to see if the up arrow is down and decreases the **Y** property of the camera's position. In the else-if I check to see if the down key is down and increment the **Y** property of the camera's position. This is different than the X values because the values of Y increase as you move down the screen. The reason for this is because of the way memory for graphics is allocated.

I'm going to add a class to the game for the player. This class will be responsible for updating and drawing the player. Right click the **EyesOfTheDragon** project in the solution explorer, select **Add** and **New Folder**. Name this new folder **Components**. Right click the **Components** folder, select **Add** and then **Class**. Name this class **Player**. This is the code for the **Player** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MGRpgLibrary;
using MGRpgLibrary.TileEngine;

namespace EyesOfTheDragon.Components
{
    public class Player
    {
        #region Field Region

        Camera camera;
        Game1 gameRef;

        #endregion
        #region Property Region
```

```
        public Camera Camera
        {
            get { return camera; }
            set { camera = value; }
        }

        #endregion

        #region Constructor Region

        public Player(Game game)
        {
            gameRef = (Game1)game;
            camera = new Camera(gameRef.ScreenRectangle);
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            camera.Update(gameTime);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
        }

        #endregion
    }
}
```

This class will be developed more as the game progresses. There are using statements to bring some of the XNA Framework as well as our **XRpgLibrary** and **XRpgLibrary.TileEngine** name spaces. I added in two fields. The camera is tied to the player so there is a **Camera** field. There is a property to expose the camera as well. There is also a **Game1** field that will be a reference to the game. The constructor takes a **Game** parameter. It casts the parameter to **Game1** and assigns it to the **gameRef** field. It also creates a camera. The **Update** method takes a **GameTime** parameter that is frequently used by components. It also calls the **Update** method of the camera passing in the **gameTime** parameter. The **Draw** method a blank method that takes a **GameTime** parameter but takes a **SpriteBatch** parameter as well.
The next step is to add a **Player** field to the **GamePlayScreen** class so you will have access to the camera to scroll the map. You will need to add a using statement for the **Components** name space of your game. Also, initialize the player field in the constructor. Add this using statement and change the **Field** region and **Constructor** region to the following.

```
using EyesOfTheDragon.Components;

        #region Field Region

        Engine engine = new Engine(32, 32);
        Tileset tileset;
        TileMap map;
        Player player;
```

```
        #endregion
        #region Constructor Region

        public GamePlayScreen(Game game, GameStateManager manager)
            : base(game, manager)
        {
            player = new Player(game);
        }

        #endregion
```

You also need to call the **Update** method of the player in the **Update** method of the **GamePlayScreen**. Change the **Update** method of the **GamePlayScreen** to the following.

```
        public override void Update(GameTime gameTime)
        {
            player.Update(gameTime);

            base.Update(gameTime);
        }
```

In the first version of tutorial 4 I had neglected to post the code for the **TileMap** class. I'm including the full code for that class here now. I did go back and upgrade tutorial 4 if you're interested.

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

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

        List<Tileset> tilesets;
        List<MapLayer> mapLayers;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public TileMap(List<Tileset> tilesets, List<MapLayer> layers)
        {
            this.tilesets = tilesets;
            this.mapLayers = layers;
        }
        public TileMap(Tileset tileset, MapLayer layer)
        {
            tilesets = new List<Tileset>();
            tilesets.Add(tileset);
            mapLayers = new List<MapLayer>();
            mapLayers.Add(layer);
        }
```

```
        #endregion

        #region Method Region

        public void Draw(SpriteBatch spriteBatch)
        {
            Rectangle destination = new Rectangle(0, 0, Engine.TileWidth, Engine.TileHeight);
            Tile tile;

            foreach (MapLayer layer in mapLayers)
            {
                for (int y = 0; y < layer.Height; y++)
                {
                    destination.Y = y * Engine.TileHeight;

                    for (int x = 0; x < layer.Width; x++)
                    {
                        tile = layer.GetTile(x, y);

                        destination.X = x * Engine.TileWidth;

                        spriteBatch.Draw(
                            tilesets[tile.Tileset].Texture,
                            destination,
                            tilesets[tile.Tileset].SourceRectangles[tile.TileIndex],
                        Color.White);
                    }
                }
            }
        }

        #endregion
    }
}
```

You still need to update the **TileMap** class to use the camera. The best option is to pass your camera as a parameter to the **Draw** method of the **TileMap** class. You will also need to use the camera to control where the tiles are drawn. I'm also going to make a couple changes to make the drawing a little more efficient. Change the **Draw** method to the following.

```
        public void Draw(SpriteBatch spriteBatch, Camera camera)
        {
            Rectangle destination = new Rectangle(0, 0, Engine.TileWidth, Engine.TileHeight);
            Tile tile;
            foreach (MapLayer layer in mapLayers)
            {
                for (int y = 0; y < layer.Height; y++)
                {
                    destination.Y = y * Engine.TileHeight - (int)camera.Position.Y;
                    for (int x = 0; x < layer.Width; x++)
                    {
                        tile = layer.GetTile(x, y);
                        if (tile.TileIndex == -1 || tile.Tileset == -1)
                            continue;
                        destination.X = x * Engine.TileWidth - (int)camera.Position.X;
                        spriteBatch.Draw(
                        tilesets[tile.Tileset].Texture,
                        destination,
```

```
                    tilesets[tile.Tileset].SourceRectangles[tile.TileIndex],
                    Color.White);
            }
        }
    }
}
```

The first change is I added in a **Camera** parameter after the **SpriteBatch** parameter. This may be a little counter intuitive but to move the camera left you add its X value to the X coordinate of the destination rectangle and to move it right you subtract its X value from the X coordinate. Basically, you need to subtract the camera's X position from the X coordinate of the destination rectangle. You also subtract the camera's Y position form the Y coordinate of the destination rectangle.

As you can see, inside the outer for loop I cast the result of subtracting the camera's Y position from the destination rectangle's Y coordinate. Then in the inner for loop I cast the result of subtracting the camera's X position form the destination rectangle's X coordinate. In the inner loop I check to make sure that either the tile index or tile set for the tile is not -1. If it is I move onto the next iteration of the loop. If you run the game now the map will scroll with the cursor keys being pressed or the left thumb stick being pressed. Only problem is the map will move off the screen. To fix this you need to lock the camera so that it will not move off the screen.

You need to add a method to the **Camera** class to lock the camera and keep it from scrolling off the edges of the map. The easy parts are the left and top. To keep in from scrolling off the top you keep the **X** and **Y** values of the camera's position from being negative. To keep it from scroll off the right and bottom you need to know the width and height of the map in pixels and the width and height of the view port you are drawing to. It would be best to assign two fields in the **TileMap** class the width and height of the map in tiles. Then you can expose the width and height of the map in pixels using properties. Change the **Field** and **Property** regions to the following.

```
#region Field Region

List<Tileset> tilesets;
List<MapLayer> mapLayers;
static int mapWidth;
static int mapHeight;

#endregion

#region Property Region

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

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

#endregion
```

You still need to set the **mapHeight** and **mapWidth** fields. You will do that in the constructors. I'm not

going to have maps with different size layers. You could do it easily but drawing the layers directly rather than doing the rendering from the map. What I suggest is in the constructors for the **TileMap** class is add a check to make sure that the layers are the same size. Change the **Constructor** region of the **TileMap** class to the following.

```
#region Constructor Region

public TileMap(List<Tileset> tilesets, List<MapLayer> layers)
{
    this.tilesets = tilesets;
    this.mapLayers = layers;
    mapWidth = mapLayers[0].Width;
    mapHeight = mapLayers[0].Height;
    for (int i = 1; i < layers.Count; i++)
    {
        if (mapWidth != mapLayers[i].Width || mapHeight != mapLayers[i].Height)
            throw new Exception("Map layer size exception");
    }
}

public TileMap(Tileset tileset, MapLayer layer)
{
    tilesets = new List<Tileset>();
    tilesets.Add(tileset);
    mapLayers = new List<MapLayer>();
    mapLayers.Add(layer);
    mapWidth = mapLayers[0].Width;
    mapHeight = mapLayers[0].Height;
}

#endregion
```

The final change will be to add a method to the **Method** region of the **Camera** class, **LockCamera**, to lock the camera. You will also want to add a call to the **LockCamera** method in the **Update** method of the **Camera** class. Change the **Method** region of the **Camera** class to the following.

```
#region Method Region

public void Update(GameTime gameTime)
{
    if (InputHandler.KeyDown(Keys.Left))
        position.X -= speed;
    else if (InputHandler.KeyDown(Keys.Right))
        position.X += speed;
    if (InputHandler.KeyDown(Keys.Up))
        position.Y -= speed;
    else if (InputHandler.KeyDown(Keys.Down))
        position.Y += speed;
    LockCamera();
}

private void LockCamera()
{
    position.X = MathHelper.Clamp(position.X,
    0,
    TileMap.WidthInPixels - viewportRectangle.Width);
    position.Y = MathHelper.Clamp(position.Y,
```

```
            0,
            TileMap.HeightInPixels - viewportRectangle.Height);
        }

        #endregion
```

The **LockCamera** method uses the **Clamp** method of the **MathHelper** class to clamp the X value of the camera's position to the world to the width of the map in pixels minus the width of the view port. If you don't subtract the width of the view port the camera will keep on moving until it reaches the width of the map and the background color will show through. You also must make sure that the camera's position is never negative. You do the same for the keeping the map for scrolling off the top of the screen by making sure its Y coordinate is never less than zero and never greater than the height of the map in pixels minus the height of the view port.

So, if you run the game now and move to the game play screen you will see the map scroll but not off the edges of the screen. One thing you will notice is that if you move the map diagonally it will scroll faster than vertically or horizontally. This can be explained by the Pythagorean Theorem. If you move the map 8 pixels down and 8 pixels right in one frame you are moving the map the square root of (8 * 8 + 8 * 8) which is greater than 8 pixels. Fortunately MonoGame provides a good way to fix that.

Instead of moving the camera when the player wants to move you find out which direction the player wants to move. You can find that be creating a **Vector2** that will have a Y value of 1 if the player wants to move down or a value of -1 if the player wants to move up. Similarly, it will have an X value of 1 if the player wants to move right or a value of -1 if the player wants to move left. You take that value and normalize it. Normalizing a vector is the process of changing it to a vector of length 1. It will still be in the same direction though. You can multiply that vector by the speed you want to the camera to move and the camera will move at the same speed in all eight directions. You must check to make sure the vector is not the zero vector because it can't be normalized because it has no direction. You can change the **Update** method of the **Camera** class to the following.

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

            if (InputHandler.KeyDown(Keys.Left))
                motion.X = -speed;
            else if (InputHandler.KeyDown(Keys.Right))
                motion.X = speed;
            if (InputHandler.KeyDown(Keys.Up))
                motion.Y = -speed;
            else if (InputHandler.KeyDown(Keys.Down))
                motion.Y = speed;

            if (motion != Vector2.Zero)
                motion.Normalize();

            position += motion * speed;

            LockCamera();
        }
```

I think the next this I will do is demonstrate a tile map with multiple layers. The first thing I will do is add a new method to the **TileMap** class called **AddLayer**. This method will add a new layer to an existing map. You should make sure that the width and the height of the new layer is the same as width and height of the map. Add the following method to the **TileMap** class in the **Methods** region.

```
public void AddLayer(MapLayer layer)
{
    if (layer.Width != mapWidth && layer.Height != mapHeight)
        throw new Exception("Map layer size exception");

    mapLayers.Add(layer);
}
```

The next step is in the **LoadContent** method of the **GamePlayScreen** to create a map layer and add it to the list of layers. If you look at the tile set that I made there are several environmental tiles that are mostly transparent. I will create a new layer made up mostly of those tiles. The will break up the monotony of the single tile map that I made. What I will do is create 80 tiles and position them randomly on the map in a new layer. You can change the **LoadConent** method to the following.

```
protected override void LoadContent()
{
    Texture2D tilesetTexture = Game.Content.Load<Texture2D>(@"Tilesets\tileset1");
    tileset = new Tileset(tilesetTexture, 8, 8, 32, 32);

    MapLayer layer = new MapLayer(40, 40);

    for (int y = 0; y < layer.Height; y++)
    {
        for (int x = 0; x < layer.Width; x++)
        {
            Tile tile = new Tile(0, 0);
            layer.SetTile(x, y, tile);
        }
    }

    map = new TileMap(tileset, layer);

    MapLayer splatter = new MapLayer(40, 40);
    Random random = new Random();

    for (int i = 0; i < 80; i++)
    {
        int x = random.Next(0, 40);
        int y = random.Next(0, 40);
        int index = random.Next(2, 14);
        Tile tile = new Tile(index, 0);
        splatter.SetTile(x, y, tile);
    }

    map.AddLayer(splatter);

    base.LoadContent();
}
```

What the new code does is first create a new layer called **splatter**. I got this name from a friend of mine who is a game programmer. He calls tiles with a lot of transparency splatter tiles. He says he

splatters them on his maps to break up the boring monotony of fields. You can also use them to give buildings a weathered looked or burn holes, lots of different things. I then create an instance of the **Random** class to generate some random numbers. That is followed by a for loop that loops 80 times to add some of the splatter tiles to the layer. Inside of the for loop I first create a X coordinate for the tile between 0 and 40 and then a number in the same range for the Y coordinate. The next number is an integer between 3 and 14, the index of the splatter tiles in the tile set. I then create a new **Tile** object and call the **SetTile** method passing in the coordinates and the **Tile** object. After creating the layer I add it to the map using the **AddLayer** method I just created.

The next step would be to demonstrate using multiple tile sets on the same map. For that you will need two tile sets. Fortunately we downloaded and added two tile sets in the last tutorial. You can replace the **LoadContent** method with the following. You can also remove the field **tileset** from the fields of the **GamePlayScreen** as well.

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

    Texture2D tilesetTexture = Game.Content.Load<Texture2D>(@"Tilesets\tileset1");
    Tileset tileset1 = new Tileset(tilesetTexture, 8, 8, 32, 32);

    tilesetTexture = Game.Content.Load<Texture2D>(@"Tilesets\tileset2");

    Tileset tileset2 = new Tileset(tilesetTexture, 8, 8, 32, 32);

    List<Tileset> tilesets = new List<Tileset>();

    tilesets.Add(tileset1);
    tilesets.Add(tileset2);

    MapLayer layer = new MapLayer(40, 40);

    for (int y = 0; y < layer.Height; y++)
    {
        for (int x = 0; x < layer.Width; x++)
        {
            Tile tile = new Tile(0, 0);
            layer.SetTile(x, y, tile);
        }
    }

    MapLayer splatter = new MapLayer(40, 40);
    Random random = new Random();

    for (int i = 0; i < 80; i++)
    {
        int x = random.Next(0, 40);
        int y = random.Next(0, 40);
        int index = random.Next(2, 14);
        Tile tile = new Tile(index, 0);
        splatter.SetTile(x, y, tile);
    }

    splatter.SetTile(1, 0, new Tile(0, 1));
    splatter.SetTile(2, 0, new Tile(2, 1));
    splatter.SetTile(3, 0, new Tile(0, 1));
```

```
            List<MapLayer> mapLayers = new List<MapLayer>();

            mapLayers.Add(layer);
            mapLayers.Add(splatter);

            map = new TileMap(tilesets, mapLayers);
        }
```

I moved everything after the call to **base.LoadContent** so if you want to use controls, or a sprite font, you will have access to them. What this code does is first load in the texture for the first tile set and create a **Tileset** object, **tileset1**. It then loads in the texture for the second tile set and create another **Tileset** object, **tileset2**. I then created a **List<Tileset>** and added in the the two **Tileset** objects. Order is important here. If you mix up the order the tiles on your map will be mixed up. Then like in the old **LoadContent** method I create a **MapLayer** filled with tile index 0 and tile set 0, which is the grass tile. I then create a second layer like earlier called **splatter**. After doing call the **SetTile** method three times passing in coordinates (1, 0) to (3, 0) and using tiles from the second tile set. After creating both layers of the map I create a **List<MapLayer>** and add the two layers I created to the list. I then call the constructor of the **TileMap** class that takes a **List<Tileset>** and a **List<MapLayer>** as parameters. If you build and run now you will get your map with a little building with a door.

You need to update the **Draw** method of **GamePlayScreen**. The **Draw** method of **TileMap** now needs a **Camera** parameter. Update the **Draw** method of **GamePlayScreen** to the following.

```
        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin(
                SpriteSortMode.Immediate,
                BlendState.AlphaBlend,
                SamplerState.PointClamp,
                null,
                null,
                null,
                Matrix.Identity);

            map.Draw(GameRef.SpriteBatch, player.Camera);

            base.Draw(gameTime);

            GameRef.SpriteBatch.End();
        }
```

The tile engine is improving but there is still more work to do but it is a good beginning. I think this is enough for this tutorial. I'd like to try and keep them to a reasonable length so that you don'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.

Good luck in your game programming adventures!
Cynthia