Eyes of the Dragon Tutorials 4.0

Part 17

Encounters-Part Three

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 seventeen 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.

The first thing I did was change AttributePair from a struct to a class. It turns out, having it as a value type rather than a reference type was causing the Adjust method to not remember changes. Replace the ICharacter interface with this version.

```
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 RpgLibrary.Characters
{
    public class AttributePair
    {
        public int Current;
        public int Maximum;

        public AttributePair()
        {
            Current = 10;
            Maximum = 10;
        }

        public AttributePair(int maximum)
        {
            Current = Maximum = maximum;
        }

        public void Adjust(int amount)
        {
```

```
            Current += amount;

            if (Current > Maximum)
            {
                Current = Maximum;
            }
        }
    }

    public interface ICharacter
    {
        string Name { get; }

        int Strength { get; set; }
        int Perception { get; set; }
        int Endurance { get; set; }
        int Charisma { get; set; }
        int Intellect { get; set; }
        int Agility { get; set; }
        int Luck { get; set; }

        AttributePair Health { get; set; }
        AttributePair Mana { get; set; }

        int Gold { get; set; }
        int Experience { get; set; }

        bool Enabled { get; set; }
        bool Visible { get; set; }
        Vector2 Position { get; set; }
        Point Tile { get; set; }
        void Update(GameTime gameTime);
        void Draw(SpriteBatch spriteBatch);
    }
}
```

As you can see, all I did was replace struct with class. Since it is now a reference type, changes made to one instance will affect all instances.

Now, it is time to get started on encounters. I am not going to create moves as strings. They will be hard coded rather than dynamic. Since I will want to use polymorphism, I will have an abstract base class that all moves will inherit from. So, right-click the SharedProject, select Add and then New Folder. Name this new folder Moves. Right-click the Moves folder, select Add and then Class. Name this new class Move. The code for that class follows next.

```
using Microsoft.Xna.Framework;
using RpgLibrary.Characters;
using System;
using System.Collections.Generic;
using System.Net.NetworkInformation;
using System.Reflection;
using System.Text;

namespace SharedProject.Moves
{
    public enum MoveType { Damage, Heal, Status }
    public enum TargetType { Health, Mana, Attribute }
```

```
public enum Status { Normal, Poison, Paralysis }

public abstract class Move : ICloneable
{
    public string Name { get; protected set; }
    public string Description { get; protected set; }
    public string Attribute { get; protected set; }
    public int Mana { get; protected set; }
    public int Range { get; protected set; }
    public Point Spread { get; protected set; }
    public MoveType MoveType { get; protected set; }
    public TargetType TargetType { get; protected set; }
    public Status Status { get; protected set; }

    protected Move(string name,
                   string description,
                   string attribute,
                   int mana,
                   int range,
                   Point spread,
                   MoveType moveType,
                   TargetType targetType,
                   Status status)
    {
        Name = name;
        Description = description;
        Attribute = attribute;
        Mana = mana;
        Range = range;
        Spread = spread;
        MoveType = moveType;
        TargetType = targetType;
        Status = status;
    }

    public virtual string Apply(ICharacter source, ICharacter target)
    {
        if (source.Mana.Current < Mana)
        {
            return $"Not enough mana to use {Name.ToLower()}.";
        }

        source.Mana.Adjust(-Mana);

        int adjustment = GetAdjustment(source, Attribute);
        int spread = (Helper.Random.Next(Spread.X, Spread.Y) + adjustment);

        if (MoveType == MoveType.Damage)
        {

            if (spread <= 0)
            {
                spread = 1;
            }

            target.Health.Adjust(-spread);
            return $"{source.Name} uses {Name.ToLower()}.";
        }
        else if (MoveType == MoveType.Heal)
        {
```

```
                target.Health.Adjust(spread);
                return $"{source.Name} uses {Name.ToLower()}.";
            }

            return $"Cannot use {Name} at this time.";
        }

        public static int GetAdjustment(ICharacter source, string attribute)
        {
            PropertyInfo info = source.GetType().GetProperty(attribute);

            if (info != null)
            {
                if (!int.TryParse(info.GetValue(source, null).ToString(), out int adjust-
ment))
                {
                    return 0;
                }

                switch (adjustment)
                {
                    case 0:
                    case 1:
                    case 2:
                        return -1;
                    case 3:
                    case 4:
                    case 5:
                        return 0;
                    case 6:
                    case 7:
                        return 1;
                    case 8:
                    case 9:
                        return 2;
                    default:
                        return 3;
                }
            }

            return 0;
        }

        public abstract object Clone();
    }
}
```

There is an enumeration, MoveType, that defines the type of move. Currently, moves are either Damage, Heal, or Status. The first two are if a move causes damage or heals respectively. The other is if the move affects the target's status, such as poison or paralysis. There is a second enumeration, TargetType that determines if the attack targets health, mana, or an attribute. The last enumeration is what status a move affects. The values are none, poisoned or paralyzed.

The class is abstract so it cannot be instansiated directly. It just allows us to group moves and use polymorphism. There are nine properties. They are all public get accessors with private set accessors. This allows us to assign them values inside child classes but not outside. They include the name of a move, its description, the mana cost, its range, the minimum and maximum range of the move, the

type of move, its target and its status.

There is a constructor that takes nine parameters, one for each property. It sets the properties to the values passed in. There is a virtual method, Apply, that takes as parameters the source for the move and the target of the move as an ICharacter. It checks to see if the current move's mana cost is less than the current mana. If it is, it returns that as a message. It then adjusts the mana based on the cost of the move. Next, it calls a method called GetAdjustment. That method gets a modifier based on an attribute. I then calculate the spread using the Random number generator of the Helper class and then the adjustment. Next, I check to see if the move is a damaging move. A move will always do at least one damage, so there is a check to see if the spread is less than or equal to zero. If it is, I set it to one. Next, I adjust the Health of the target by negative the spread that was calculated and return a string that the source used the move. If the move was a Heal move, I adjust the Health value positively and return a message. If nothing matched, I return a message that the move cannot be used at this time.

The GetAdjustment method finds a modifier to a move based on an attribute. The first step is to go the property using reflection, I just love reflection! If that property is not null, I use TryParse to parse the value to an integer. If that fails, I return zero. Otherwise, there is a switch. If the attribute is between zero and two, I return -1. If it is between three and five, I return zero. If it is six or seven, I return one. If it is eight or nine, I return two. Otherwise, I return three. If parse the property failed, I return zero. Eventually, I will be implementing the factory pattern for moves, so there is a Clone method that is abstract.

Now, let's create a new move. This will be a basic move that deals between one and four damage. It is assuming an attack without a weapon. Right-click the Moves folder, select Add, and then Class. Name this new class BasicAttack. This is the code for that class.

```
using Microsoft.Xna.Framework;
using RpgLibrary.Characters;
using SharedProject.Moves;
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Text;

namespace SharedProject.Mobs
{
    public class BasicAttack : Move
    {
        private BasicAttack(string name,
                            string description,
                            string attribute,
                            int mana,
                            int range,
                            Point spread,
                            MoveType moveType,
                            TargetType targetType,
                            Status status)
            : base(name, description, attribute, mana, range, spread, moveType, target-
Type, status)
        {
```

```
        }

        public override string Apply(ICharacter source, ICharacter target)
        {
            return base.Apply(source, target);
        }

        public override object Clone()
        {
            BasicAttack basicAttack = (BasicAttack)MemberwiseClone();
            return basicAttack;
        }

        public static Move CreateInstance()
        {
            Move move = new BasicAttack("basic attack",
                            "This move does basic attack without weapons.",
                            "Strength",
                            0,
                            1,
                            new(1, 5),
                            MoveType.Damage,
                            TargetType.Health,
                            Status.Normal);

            return move;
        }
    }
}
```

This class inherits from Move, so it gains access to all public, internal, and protected members. It must call the constructor that takes nine parameters. The Apply method calls the base Apply method. The Clone method creates a shallow copy of the move. This means that any changes you make to the clone affect the parent, so be careful! Finally, there is a static method, CreateInstance, that creates an instance of the move with some default values.

So, let's implement the basic attack in the game. First, we need to add a using statement for the moves.

```
using SharedProject.Moves;
```

Next, we need to change the DoAttack method and the HandleEnemies method of the EncounterState to the following.

```
private void DoAttack(Point direction)
{
    Point target = Player.Tile + direction;
    Rectangle destination = new(target, new(Engine.TileWidth, Engine.TileHeight));

    for (int i =  0; i < encounter.Enemies.Count; i++)
    {
        var enemy = encounter.Enemies[i];

        Point enemyTile = new((int)((Mob)enemy).AnimatedSprite.Position.X / Engine.Tile-
Width,
```

```
            (int)((Mob)enemy).AnimatedSprite.Position.Y / Engine.TileHeight);

        Rectangle enemyDestination = new(enemyTile, new(Engine.TileWidth, Engine.Tile-
Height));

        if (enemyDestination.Intersects(destination) && Helper.RollDie(((ICharac-
ter)Player),"Agility"))
        {
            if (!Helper.RollDie(enemy, "Agility"))
            {
                _messages.Enqueue("   Enemy was hit...");

                Move attack = BasicAttack.CreateInstance();
                attack.Apply(Player, enemy);
                // health.Current -= Helper.Random.Next(1, 7);
            }
            else
            {
                _messages.Enqueue("   Enemy dodges your attack...");
            }
        }
        else
        {
            _messages.Enqueue("   Your attack fell upon empty air...");
        }
    }
}

private void HandleEnemies()
{
    _turn = !_turn;
    _timer = 0;

    foreach (ICharacter c in encounter.Enemies)
    {
        Mob mob = c as Mob;
        Point distance = mob.AnimatedSprite.Tile - Player.Tile;

        if ((Math.Abs(distance.X) == 1 && Math.Abs(distance.Y) == 0) ||
            (Math.Abs(distance.Y) == 1 && Math.Abs(distance.X) == 0))
        {
            bool roll = Helper.RollDie(c, "Agility");

            if (roll)
            {
                if (!Helper.RollDie(Player, "Agility"))
                {
                    _messages.Enqueue($"The {c.Name} swings and hits...");

                    Move attack = BasicAttack.CreateInstance();
                    attack.Apply(c, Player);
                }
                else
                {
                    _messages.Enqueue($"You nimbly dodge the {c.Name}'s attack");
                }
            }
            else
            {
                _messages.Enqueue($"The {c.Name} swings and misses");
```

```
            }
        }
    }
}
```

When it comes time to calculate the damage of an attack I use the CreateInstance method of the BasicAttack class to generate an instance of BaseAttack. I then call the Apply method to apply the attack. In the DoAttack method, I pass the player as the source and the enemy as the target. Then, in the HandleEnemies method, I create an instance of BasicAttack and call Apply passing in the enemy as the source and the player as the target.

You can build and run again. The game will appear to behave as before. Under the hood, though, it is now using BasicAttack when it comes to attacking.

With this addition, there is a problem. When I created the string for the bat, I left out mana. This is a problem when we went from using a value type to using a reference type. It was not initialized when creating a bat, causing a null reference error when it came to using a basic attack. Replace the LoadContent method of the GamePlayState to the following.

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

    _conversationState = Game.Services.GetService<IConversationState>();
    _conversationManager = Game.Services.GetService<IConversationManager>();
    _encounterState = Game.Services.GetService<IEncounterState>();

    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/femalefighter");
```

```
    AnimatedSprite rio = new(texture, animations)
    {
        CurrentAnimation = "walkdown",
        IsAnimating = true,
    };

    CharacterLayer chars = new();

    chars.Characters.Add(
        new Villager(rio, new(10, 10))
        {
            Position = new(480, 480),
            Tile = new(10, 10),
            Visible= true,
            Enabled=true,
            Conversation="Rio"
        });

    map.AddLayer(chars);

    EncounterLayer encounters = new();

    Encounter encounter = new(Player);
    encounter.Enemies.Add(Mob.FromString("Name=Giant Bat,Strength=3,Agil-
ity=5,Health=21,Mana=21,Position=640:640,Tile=5:5,AnimatedSprite=32x32-bat-
sprite;down:32:0:32:32:3;right:32:32:32:32:3;up:32:64:32:32:3;left:32:96:32:32:3;deaddown
:0:0:32:32:1;deadright:0:32:32:32:1;deadup:0:64:32:32:1;deadleft:0:96:32:32:1;down",
Game.Content));

    encounters.Encounters.Add(((Mob)encounter.Enemies[0]).AnimatedSprite, encounter);
    map.AddLayer(encounters);

    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);
}
```

I am going to get started on mob movement. First, I added a ray tracing method to the tile map class that is used for "seeing" the player. Add this method to the TileMap class.

```
public static IEnumerable<Vector2> GetPointsOnLine(float x0, float y0, float x1, float
y1)
{
    bool steep = Math.Abs(y1 - y0) > Math.Abs(x1 - x0);
    if (steep)
    {
        float t;
        t = x0; // swap x0 and y0
        x0 = y0;
        y0 = t;
        t = x1; // swap x1 and y1
        x1 = y1;
        y1 = t;
    }
    if (x0 > x1)
    {
        float t;
        t = x0; // swap x0 and x1
        x0 = x1;
        x1 = t;
        t = y0; // swap y0 and y1
        y0 = y1;
        y1 = t;
    }
    float dx = x1 - x0;
    float dy = Math.Abs(y1 - y0);
    float error = dx / 2;
    float ystep = (y0 < y1) ? 1f : -1f;
    float y = y0;
    for (float x = x0; x <= x1; x += 1f)
    {
        yield return new Microsoft.Xna.Framework.Vector2((steep ? y : x), (steep ? x :
y));
        error -= dy;
        if (error < 0)
        {
            y += ystep;
```

```
            error += dx;
        }
    }
    yield break;
}
```

This the code for a line between two points. It then adds each point to a IEnumerable<Vector2>. It is a very basic line of sight algorithm. Next, in the encounter state, we will use this points to see if the enemy can move into a tile. There is one exception where it is not working. That is if the enemy wants to move into the player's tile. They can coexist in the same tile. I will handle that in a separate tutorial. First, replace the HandleEnemies method of the EncounterState with the following code.

```
private void HandleEnemies()
{
    _turn = !_turn;
    _timer = 0;

    foreach (ICharacter c in encounter.Enemies)
    {
        if (HandleEnemyMove(c, Player))
        {
            return;
        }

        Mob mob = c as Mob;
        Point distance = mob.AnimatedSprite.Tile - Player.Sprite.Tile;

        if ((Math.Abs(distance.X) == 1 && Math.Abs(distance.Y) == 0) ||
            (Math.Abs(distance.Y) == 1 && Math.Abs(distance.X) == 0))
        {
            bool roll = Helper.RollDie(c, "Agility");

            if (roll)
            {
                if (!Helper.RollDie(Player, "Agility"))
                {
                    _messages.Enqueue($"The {c.Name} swings and hits...");

                    Move attack = BasicAttack.CreateInstance();
                    attack.Apply(c, Player);
                }
                else
                {
                    _messages.Enqueue($"You nimbly dodge the {c.Name}'s attack");
                }
            }
            else
            {
                _messages.Enqueue($"The {c.Name} swings and misses");
            }
            return;
        }

    }
}
```

What the new code does is call a new method that checks for a collision. If there is a collision, movement is cancelled. Here is the method that checks for a collision. Collision with the player is a bit wonky, but I will address that in a future tutorial, when I move to ranged attacks.

```
private bool HandleEnemyMove(ICharacter c, Player player)
{
    Mob e = c as Mob;

    float distance = Vector2.Distance(player.Sprite.Position, e.AnimatedSprite.Position);
    CollisionLayer collisions = encounter.Map.Layers.FirstOrDefault(x => x is Colli-
sionLayer) as CollisionLayer;

    List<Vector2> points = TileMap.GetPointsOnLine(
        player.Sprite.Tile.X,
        player.Sprite.Tile.Y,
        e.AnimatedSprite.Tile.X,
        e.AnimatedSprite.Tile.Y).OrderByDescending(x => x.X).ThenBy(x => x.Y).ToList();

    bool blocked = false;

    foreach (Vector2 point in points)
    {
        if (collisions == null)
        {
            break;
        }

        foreach (Rectangle rectangle  in collisions.Collisions.Keys)
        {
            if (rectangle.Contains(point))
            {
                blocked = true;
                break;
            }
        }
    }

    if (blocked)
    {
        return false;
    }

    Point next = e.AnimatedSprite.Tile;

    if (e.AnimatedSprite.Tile.X < player.Sprite.Tile.X)
    {
        next.X++;
    }
    else if (e.AnimatedSprite.Tile.X > player.Sprite.Tile.X)
    {
        next.X--;
    }
    else if (e.AnimatedSprite.Tile.Y < player.Sprite.Tile.Y )
    {
        next.Y++;
    }
    else if (e.AnimatedSprite.Tile.Y > player.Sprite.Tile.Y)
    {
```

```
            next.Y--;
        }

        Point nextInPixels = new(next.X * Engine.TileWidth, next.Y * Engine.TileHeight);

        if (collisions != null)
        {
            foreach (Rectangle rectangle in collisions.Collisions.Keys)
            {
                if (rectangle.Contains(nextInPixels))
                {
                    return true;
                }
            }
        }

        Rectangle location = Helper.Destiation(Player.Sprite.Tile);

        if (!location.Intersects(Helper.Destiation(next)))
        {
            e.AnimatedSprite.Tile = next;
            e.AnimatedSprite.Position = new Vector2(e.AnimatedSprite.Tile.X * Engine.Tile-
Width, e.AnimatedSprite.Tile.Y * Engine.TileHeight);
            return false;
        }

        return false;
    }
```

The method takes as parameters, an ICharacter and a Player object. It casts the ICharacter to a Mob object. It calculates the distance between the mob and the player. This is a remnant from code that I used in a Rogue-Like game that I made where the player was centred on the map, and the mob would only go after the player if it was half the distance of the screen away. I kept in it, because it might be useful.

The first step is to grab the collision layer. I do that using a little LINQ. I then use the GetPointsOnLine method to draw a line between the mob and the player. Ideally, you would want to draw lines in all directions. We are going for simplicity, though. The points are ordered by the X coordinate and then the Y coordinate, and that is converted to a List. There is a local variable, blocked, that determines if the line of sight is blocked. The next step is to loop over the points on the line. If the collision layer is null, break out of the loop. Next, I loop over the keys in the collision layer. If the key contains the point, blocked is set to true, and I break out of the loop. If blocked, false is returned.

Next, I save the tile the mob is currently in. Movement is biased in the horizontal direction. If the mob's X coordinate is less than the player's coordinate, it moves one tile right. Similarly, if the mob's X coordinate is great than the player's X coordinate, it moves one tile left. I do something similar in the Y direction. I get a point, in pixels, that represents the next tile the mob wants to move into. If the collision layer is not null, I loop over the keys in the collision collection. If the key contains the next point as a pixel, it returns true.

Next, I call a method that I added to the Helper class that gets a destination rectangle based on a point. This rectangle is the player's location. If this rectangle does not intersect with the desired

position, I set the Tile and Position property based on the desired movement. I then return false. If all checks have failed, I return false.

That leads us to the Destination method of the Helper class. It just takes a point that represents a tile and returns a rectangle that describes the tile in world space. Add this method to the Helper class.

```
public static Rectangle Destiation(Point p)
{
    return new(new(p.X * Engine.TileWidth, p.Y * Engine.TileHeight),
                new(Engine.TileWidth, Engine.TileHeight));
}
```

There are two outstanding problems in this tutorial. First, which I've already discussed, is that the mob can move into the player's tile. The second is that the player does not need to be next to the mob in order to attack it. It can be on the other side of the screen and still attack it. I will address this latter problem. The former problem isn't offering an immediate solution, so I'm going to park it for now. Update the DoAttack method of the EncounterState class to the following.

```
private void DoAttack(Point direction)
{
    Point target = Player.Sprite.Tile + direction;
    Rectangle destination = Helper.Destiation(target);

    for (int i =  0; i < encounter.Enemies.Count; i++)
    {
        var enemy = encounter.Enemies[i];

        Point enemyTile = ((Mob)enemy).AnimatedSprite.Tile;

        Rectangle enemyDestination = Helper.Destiation(enemyTile);

        if (enemyDestination.Intersects(destination) && Helper.RollDie(((ICharac-
ter)Player),"Agility"))
        {
            if (!Helper.RollDie(enemy, "Agility"))
            {
                _messages.Enqueue("   Enemy was hit...");

                Move attack = BasicAttack.CreateInstance();
                attack.Apply(Player, enemy);
                // health.Current -= Helper.Random.Next(1, 7);
            }
            else
            {
                _messages.Enqueue("   Enemy dodges your attack...");
            }
        }
        else
        {
            _messages.Enqueue("   Your attack fell upon empty air...");
        }
    }
}
```

What the new code does is grab the destination of the player's attacking tile using the new

Destination method of the Helper class. It now uses this new method to grab the location of the mob using the Destination method. The rest of the method flows as before.

I'm going to park this tutorial here. There is more that I could pack in, but I will save it for the following tutorials. Also, it's been a while since I've published, and I'm sure you're eager for more. 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. I'm thinking of reviving my newsletter so you will be informed of new stuff rather than having to keep looking for new things.

Good luck with your game programming adventures!
Cynthia