A Summoner's Tale – MonoGame Tutorial Series

Chapter 6

Avatars

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called A Summoner's Tale. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my web site: A Summoner's Tale. The source code for each tutorial will be available as well. I will be using Visual Studio 2013 Premium for the series. The code should compile on the 2013 Express version and Visual Studio 2015 versions as well.

I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

The key component for this game will be the avatars. Without the avatars the player is just wandering the map interacting with characters. There is no excitement for the player. Just as a refresher, I have substituted what I call avatars for Pokemon. In essence they are basically the same though. You find avatars, battle them against other avatars to increase your their power and level. They are elementally aligned just like in Pokemon as well with different attributes and moves. The difference is that you learn spells instead of capturing them. I will include a side tutorial on how you can change the game to capture avatars instead of learning to summon them from other characters or scrolls.

Since they are so integral to the game I am going to implement them now. So, right click the Avatars project, select Add and then New Folder. Name this new folder AvatarComponents. Avatars have moves that they use when battling other avatars. For that reason I'm going to add in an interface for moves. All moves will implement this interface. That allows us to have a collection of moves in the

avatar component. Right click the AvatarComponents folder, select Add and then Interface. Name this interface IMove. Here is the code for this interface.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Avatars.AvatarComponents
{
	public enum Target
	{
		Self, Enemy
	}

	public enum MoveType
	{
```



```
		Attack, Heal, Buff, Debuff, Status
	}

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

	public enum MoveElement
	{
		None, Dark, Earth, Fire, Light, Water, Wind
	}

	public interface IMove
	{
		string Name { get; }
		Target Target { get; }
		MoveType MoveType { get; }
		MoveElement MoveElement { get; }
		Status Status { get; }
		int UnlockedAt { get; set; }
		bool Unlocked { get; }
		int Duration { get; set; }
		int Attack { get; }
		int Defense { get; }
		int Speed { get; }
		int Health { get; }
		void Unlock();
		object Clone();
	}
}
```

First, a move targets something. It can either target the enemy/opponent or it can target that avatar

so there is an enumeration that defines this. Next is an enumeration that defines what type of move this is. For example, is it an attack, a buff for your avatar or a debuff on the enemy avatar. This can be extended if you want to include a different type of move like Abra's escape move from Pokemon. A move can affect the enemy avatar's status so I included an enumeration for that. As well I included an enumeration for what element the move is for. A basic move like a tackle has no element where as a fire based or water based attack does.

I use enumerations for these sorts of things because I don't want a list of strings or numbers to represent things like this. Also, since I'm using an enumeration it is easy to move them to classes and have a static property that returns the value.

Next is the interface that defines the different properties/methods that a move must implement. These are all common elements for the move. Moves must have a name so I included a property for that. They also require a target using the Target enumeration, a move type using the MoveType enumeration, an element using MoveElement enumeration and a status it can apply using the Status enumeration.

Moves will unlock at different levels so I included a property that exposes that as well as if the move has been learned/unlocked. A move may have a duration so I included that as a property. A move can affect the base attributes of an avatar which are Attack, Defence, Speed and Health. You can of course add other attributes but this is a good base. I then have a method that will be called to unlock/learn the move and a method to clone a move. I add clone so that I can create a master list of moves and just clone them and add them to the player's or opponent's avatars. It is always a good idea when you are using game objects that you might want multiple of to have a way to easily make a copy of that game object. For that reason you will find that I use Clone a lot in my games.

Next, I will add the class for avatars. Right click the AvatarComponents folder, select Add and then Class. Name this new class Avatar. This class contains a lot of code so I'm going to give it to you in pieces and explain it bit by bit rather than the whole things. First, these are the using statements that I used to bring required classes into scope.

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

I didn't implement avatars as content items in my demo that I build so I included the System.IO namespace so that I can easily read/write to and from disk. I also brought in some of the XNA/MonoGame names spaces into scope.

Next up is an enumeration that defines the elements and avatar can have. Outside of the Avatar class add the following enumeration.

```
public enum AvatarElement
{
	Dark, Earth, Fire, Light, Water, Wind
}
```

There are a number of fields required for an avatar. Inside of the Avatar class add the following fields.

```
#region Field Region

private static Random random = new Random();
private Texture2D texture;
private string name;
private AvatarElement element;
private int level;
private long experience;
private int costToBuy;
private int speed;
private int attack;
private int defense;
private int health;
private int currentHealth;
private List<IMove> effects;
private Dictionary<string, IMove> knownMoves;

#endregion
```

I included a static Random field in this class for random number generation for avatars. In my demo I just used a texture for the avatars so I included a Texture2D field for that. Avatars have a name so their is a field for that. They also have an element so that was added as well. They have a level and experience gained so that is there as well. They can be bought so I included a cost to buy field. Their base attributes are speed, attack, defense and health so there are fields for that. Health represents their maximum health so I added a field for their current health. The next is a List<IMove> called effects. If you look back at IMove I included status effects as well as buffs/debuffs for avatars. When one of these moves is applied to an avatar I add that move to the effects list. When it no longer affects the avatar is removed from the list. I also include a Dictionary<string, IMove> that holds the moves known by the avatar. Here I will be deviating a bit from Pokemen in that an avatar can know more than 4 moves and does not have to forget an old move to learn a new move. I will include a

side tutorial that explains how to implement the learning/forgetting moves if you want to stay more true to Pokemon.

These fields need to be exposed to other classes so I had to add a number of properties. Add the following properties just below the fields inside the Avatar class.

```
#region Property Region

public string Name
{
	get { return name; }
}

public int Level
{
	get { return level; }
	set { level = (int)MathHelper.Clamp(value, 1, 100); }
}

public long Experience
{
	get { return experience; }
}

public Texture2D Texture
{
	get { return texture; }
}

public Dictionary<string, IMove> KnownMoves
{
	get { return knownMoves; }
}

public AvatarElement Element
{
	get { return element; }
}

public List<IMove> Effects
{
	get { return effects; }
}

public static Random Random
{
	get { return random; }
}

public int BaseAttack
{
	get { return attack; }
}

public int BaseDefense
{
	get { return defense; }
}

public int BaseSpeed
{
	get { return speed; }
}
```

```
public int BaseHealth
{
        get { return health; }
}

public int CurrentHealth
{
        get { return currentHealth; }
}

public bool Alive
{
        get { return (currentHealth > 0); }
}

#endregion
```

Other than Level these are all read only properties. I included a set for Level because I was capping the avatar's level at 100. Typically I would have mad the set private so that it could only be adjusted inside of the class. The other interesting thing is that instead of naming the properties for Attack, Defense, Speed and Health I place Base before each of them. That is because the effects field can possibly affect the avatar's attributes. I also included an Alive field that that can be used to check if the avatar's health is less than one an is either unconscious or defeated.

I added a private constructor to this class. It just sets the level to 1 and initializes the dictionary and list for moves. Add the following constructor below the properties.

```
#region Constructor Region

private Avatar()
{
        level = 1;
        knownMoves = new Dictionary<string, IMove>();
        effects = new List<IMove>();
}

#endregion
```

The next thing I'm going to add is the code for resolving a move. Below the constructor add the following method.

```
        public void ResoleveMove(IMove move, Avatar target)
        {
                bool found = false;
                switch (move.Target)
                {
                        case Target.Self:
                                if (move.MoveType == MoveType.Buff)
                                {
                                        found = false;
                                        for (int i = 0; i < effects.Count; i++)
                                        {
```

```
                    if (effects[i].Name == move.Name)
                    {
                        effects[i].Duration += move.Duration;
                        found = true;
                    }
                }

                if (!found)
                    effects.Add((IMove)move.Clone());
```



```
                }
                else if (move.MoveType == MoveType.Heal)
                {
                    currentHealth += move.Health;
                    if (currentHealth > health)
                        currentHealth = health;
                }
                else if (move.MoveType == MoveType.Status)
                {
                }

                break;
            case Target.Enemy:
                if (move.MoveType == MoveType.Debuff)
                {
                    found = false;
                    for (int i = 0; i < target.Effects.Count; i++)
                    {
                        if (target.Effects[i].Name == move.Name)
                        {
                            target.Effects[i].Duration += move.Duration;
                            found = true;
                        }
                    }

                    if (!found)
                        target.Effects.Add((IMove)move.Clone());
                }
                else if (move.MoveType == MoveType.Attack)
                {
                    float modifier = GetMoveModifier(move.MoveElement, target.Element);

                    float tDamage = GetAttack() + move.Health * modifier -
target.GetDefense();

                    if (tDamage < 1f)
                        tDamage = 1f;

                    target.ApplyDamage((int)tDamage);
                }

                break;
        }
```

```
}
```

The method accepts an IMove parameter for the move to be applied and an Avatar parameter for the target. The local variable found is used to look to see if an existing effect is found or not. There is then a switch on the target for the move. This is part of the reason why I created an interface for moves. With the interface I have all the information needed to resolve the move without knowing anything about the move being applied. I just use the contract that was defined to apply the move.

The first case that I check in the switch is if the target is Self, or the current avatar being used. I then check to see if the move type is Buff, which increases the avatar's attribute in some way. If it is I set the found variable to false. I then loop through all of the active effects that have been added to the avatar. If I find a move that has the same name I increase the current duration of the effect that is being applied to current duration. Instead of this you might want to replace the current duration with the duration for the move. It is totally up to you how you want to apply this.

I then check if the move was found or not. If it wasn't found I add a clone of the move to the list of effects currently applied to the avatar. If the move is of type Heal I increase the avatar's current health by the health modifier for the move. Then if current health is above the maximum health I set

it to the maximum health. I included a check for the status of the move but didn't implement any code yet. That is because I was considering adding in more status types in my demo but never made it that far. For example, I could have included a status Invulnerable where the avatar could not be harmed by physical attacks.

Next I handle the enemy case, which is very similar to the self case. The biggest difference is that the move is applied to the enemy. It doesn't make sense to buff an enemy so I include the debuff case. This works the same as the buff case for self. I search to see if that is already there. If it is not there I add it to the list. For attacking I first call a method GetMoveModifier passing in the element for the move and the element of the target. This is where you apply logic for fire moves being strong against grass moves. I will get to that method shortly. I then call a method GetAttack that returns the avatar's attack value adding in any buffs or debuffs that have been applied. I add that to move.Health times the modifier which returns how much damage the moved does I then subtract the target's defense attribute. I then check if the damage done is less than 1 and if it is I set it to 1 because I implemented the rule that move always does 1 damage. You can also add in here if a move misses by including accuracy and such. I then call a method ApplyDamage on the target which will apply the damage. That method is yet to come.

The next method that I want to add is the one that gets if a move is effective against a certain type of avatar or not effective. Add the following method to the class.

```
public static float GetMoveModifier(MoveElement moveElement, AvatarElement avatarElement)
{
```

```
        float modifier = 1f;

        switch (moveElement)
        {
            case MoveElement.Dark:
                if (avatarElement == AvatarElement.Light)
                    modifier += .25f;
                else if (avatarElement == AvatarElement.Wind)
                    modifier -= .25f;
                break;
            case MoveElement.Earth:
                if (avatarElement == AvatarElement.Water)
                    modifier += .25f;
                else if (avatarElement == AvatarElement.Wind)
                    modifier -= .25f;
                break;
            case MoveElement.Fire:
                if (avatarElement == AvatarElement.Wind)
                    modifier += .25f;
                else if (avatarElement == AvatarElement.Water)
                    modifier -= .25f;
                break;
            case MoveElement.Light:
                if (avatarElement == AvatarElement.Dark)
                    modifier += .25f;
                else if (avatarElement == AvatarElement.Earth)
                    modifier -= .25f;
                break;
            case MoveElement.Water:
                if (avatarElement == AvatarElement.Fire)
                    modifier += .25f;
                else if (avatarElement == AvatarElement.Earth)
                    modifier -= .25f;
                break;
            case MoveElement.Wind:
                if (avatarElement == AvatarElement.Light)
                    modifier += .25f;
```



```
                else if (avatarElement == AvatarElement.Earth)
                    modifier -= .25f;
                break;

        }

        return modifier;
}
```

This method is really just a look up table. It takes the element of the move type and compares it to the avatar's type. What this does is first sets a local variable modifier to 1f, or normal damage. In each of the cases I first check to see if the move is effective. If it is effect the move will do 25% more damage than the base so I add .25f to the modifier. If it is not effective it will to 25% less damage so I subtract .25 from the modifier. For example, a Dark move is strong against a Light avatar and does

125% of its normal damage to a Light type avatar. Similarly, it is weak against a Wind avatar and does 75% of its normal damage. All of the other cases are similar where an element is strong against one type and weak against another. In this look up table you can add a lot more cases though, and elements. This was just a good start for my demo game.

I'm going to now add two small methods. The one that applies damage and one that updates the avatar each round of combat. Add the following two methods to the class.

```
public void ApplyDamage(int tDamage)
{
	currentHealth -= tDamage;
}

public void Update(GameTime gameTime)
{
	for (int i = 0; i < effects.Count; i++)
	{
		effects[i].Duration--;

		if (effects[i].Duration < 1)
		{
			effects.RemoveAt(i);
			i--;
		}
	}
}
```

Apply damage is trivial. It just reduces the avatar's currentHealth field by the damage being passed in. Update loops through the activate effects and reduces the Duration field by 1. If the duration is less than zero I remove the effect and decrease the loop variable by 1. I do that because there is one less element in the list.

What I am going to add next are four get method that get the current attack, defense, speed and maximum health of an avatar. They all work in the same way. They loop through each of the active effects. If the effect is a buff it adds the buff to the attribute and if it is a debuff it subtracts it. It then returns the attribute plus the modifier. Add the following four methods to the class after Update.

```
public int GetAttack()
{
	int attackMod = 0;

	foreach (IMove move in effects)
	{
		if (move.MoveType == MoveType.Buff)
```



```
			attackMod += move.Attack;
```

```
                if (move.MoveType == MoveType.Debuff)
                        attackMod -= move.Attack;
        }

        return attack + attackMod;
}

public int GetDefense()
{
        int defenseMod = 0;

        foreach (IMove move in effects)
        {
                if (move.MoveType == MoveType.Buff)
                        defenseMod += move.Defense;

                if (move.MoveType == MoveType.Debuff)
                        defenseMod -= move.Defense;
        }

        return defense + defenseMod;
}

public int GetSpeed()
{
        int speedMod = 0;

        foreach (IMove move in effects)
        {
                if (move.MoveType == MoveType.Buff)
                        speedMod += move.Speed;
                if (move.MoveType == MoveType.Debuff)
                        speedMod -= move.Speed;
        }

        return speed + speedMod;
}

public int GetHealth()
{
        int healthMod = 0;

        foreach (IMove move in effects)
        {
                if (move.MoveType == MoveType.Buff)
                        healthMod += move.Health;
                if (move.MoveType == MoveType.Debuff)
                        healthMod += move.Health;
        }

        return health + healthMod;
}
```

I'm going to add in a couple more methods now. One that will be called when combat starts. One that will be called when an avatar wins a battle and one with it loses a battle. Finally one that will be called to check if the avatar has levelled up. Add these methods after the get methods.

```
public void StartCombat()
{
        effects.Clear();
        currentHealth = health;
}
```

```
public long WinBattle(Avatar target)
{
	int levelDiff = target.Level - level;
	long expGained = 0;

	if (levelDiff <= -10)
	{
		expGained = 10;
	}
	else if (levelDiff <= -5)
	{
		expGained = (long)(100f * (float)Math.Pow(2, levelDiff));
	}
	else if (levelDiff <= 0)
	{
		expGained = (long)(50f * (float)Math.Pow(2, levelDiff));
	}
	else if (levelDiff <= 5)
	{
		expGained = (long)(5f * (float)Math.Pow(2, levelDiff));
	}
	else if (levelDiff <= 10)
	{
		expGained = (long)(10f * (float)Math.Pow(2, levelDiff));
	}
	else
	{
		expGained = (long)(50f * (float)Math.Pow(2, target.Level));
	}

	return expGained;
}

public long LoseBattle(Avatar target)
{
	return (long)((float)WinBattle(target) * .5f);
}

public bool CheckLevelUp()
{
	bool leveled = false;

	if (experience >= 50 * (1 + (long)Math.Pow(level, 2.5)))
	{
		leveled = true;
		level++;
	}

	return leveled;
}
```

In my game after each battle the avatar returns to its element plane. There it instantly heals all

damage and all status effects are removed. So, I clear the effects and reset the health in the StartCombat method. In WinBattle I decide how much experience the avatar gains for winning the battle. I first determine the level difference between the two avatars. If the player's avatar is more than ten levels higher than the opponent's avatar it gains 10 experience. If it is between 9 and 5 levels higher I use a formula to get an experience value. The way the formula works is using 2 to the power of the level difference. In this case the level difference is negative so it will return a fraction of the base. The next case is for between 0 and 4 levels higher. Again if it is negative it will return a fraction and if it is zero it will return 1. The other cases are similar when the opposing avatar was a higher level that the player's avatar. Technically the player should never win if the opposing avatar is more than 10 levels but I included it is a catch all. I know that in some games that if the opponent is

that much higher you gain zero experience for defeating it. If the player does lose a battle I still reward the avatar half the experience for losing the battle. This would actually be a very high value is they lost against a much stronger avatar and will need to be tweeked accordingly. The other method that I added is called CheckLevelUp and it checks to see if the avatar has levelled up or not. I use another formula for that that uses a power variable again. This makes it so that as the avatar's level grows the experience needed to grow increase as well. This seemed to work okay in my demo but may need to be tweeked a bit in a real game.

I also included a method for levelling up an avatar. Rather than automatically adjusting the attributes like in Pokemon I allow the players to assign points to the attribute of their choice. There is a switch that checks which attribute has been chosen and updates that attribute. If they choose health I multiple that value by 5. Here is the code for that method.

```
public void AssignPoint(string s, int p)
{
        switch (s)
        {
                case "Attack":
                        attack += p;
                        break;
                case "Defense":
                        defense += p;
                        break;
                case "Speed":
                        speed += p;
                        break;
                case "Health":
                        health += p * 5;
                        break;
        }
}
```

The last method that I'm going to add is a Clone method. This can be used to grab a copy of the

avatar from a master list of avatars when the player finds a new avatar. Here is a code for that method.

```
public object Clone()
{
        Avatar avatar = new Avatar();

        avatar.name = this.name;
        avatar.texture = this.texture;
        avatar.element = this.element;
        avatar.costToBuy = this.costToBuy;
        avatar.level = this.level;
        avatar.experience = this.experience;
        avatar.attack = this.attack;
        avatar.defense = this.defense;
        avatar.speed = this.speed;
        avatar.health = this.health;
        avatar.currentHealth = this.health;

        foreach (string s in this.knownMoves.Keys)
        {
                avatar.knownMoves.Add(s, this.knownMoves[s]);
        }

        return avatar;
}
```


Nothing hard about this method. It just sets the fields with the values from the current instance. To do the moves I use a foreach loop that loops over the list. I then return the new object.

I'm going to end this here as we've covered a lot and it is a very important component. I had wanted to implement some game play in this tutorial but most elements that I wanted to include use this component. For example, most NPCs have an avatar associated with them or work with avatars in some way, such as giving them to the player, training them or something similar. For battles they are definitely required. The next tutorial I will be adding in NPCs to the game and some of the components required for having conversations with NPCs.

Please stay tuned for the next tutorial in this series. If you don't want to have to keep visiting the site to check for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news from Game Programming Adventures. You can also follow my tutorials on Twitter at https://twitter.com/GPAAdmi77640534.

I wish you the best in your MonoGame Programming Adventures!
Jamie McMahon