EvenMoreModifiers Documentation

Jofairden

Oct 21, 2018

Table of contents

CHAPTER 1

Introductory

Welcome to the official documentation of Even More Modifiers! Please refer to the sidebar menu to navigate this documentation site.

This documentation is generously hosted by readthedocs.io Docs are written in rst (reStructuredText) and is processed with Sphinx python. (Markdown docs supported using MkDocs) You can view the docs on Github here Syntax highlighting is done by Pygments which internally uses 'lexers' to generate highlighting.

Navigate using the links below or use the sidebar.

1.1 Getting started

It is completely possible to use EMM in your own mod and start adding your own modifiers! On this page, we will cover how te set yourself up in order to do this.

1.1.1 Extracting the mod

Firstly, download EMM's .tmod file and place it in your mods folder. Now go in game, navigate to the mods menu and press the extra info button for EMM. From there, press the extract button.

1.1.2 Getting the .dll file

Now you will have extracted EMM, with that comes the .dll file you'll need for your project. The extraction process unpacks the mod and places its files in the Mod Reader folder. This is usually located in `%userprofile%\Documents\My Games\Terraria\ModLoader\Mod Reader`. In there, look for a folder named `Loot`. Inside that folder, grab the Windows.dll file and take it to some place you can easily navigate to. It is recommended to place it inside a `lib/` folder inside your mod root. Make sure to rename the file to `Loot.dll`.

FAQ: Why call it 'Loot'? **A**: EMM is internally named as 'Loot' by its previous owner, which cannot be changed.

1.1.3 Referencing the file

Now inside Visual Studio you can add the .dll as a reference which will give you access to everything EMM has to offer you. To do this, navigate to the references subject in your solution explorer, right click it and press 'Add Reference'

1.1.4 Usage requirements

Setting the dependency

Note: it is very important to consider if you want your mod to be an optional reference or forced. If you want EMM to be required for your mod, add EMM to your build.txt as dependency: `modReferences = Loot`. If you want EMM to be optional, add it to the weakReferences in build.txt: `weakReferences = Loot`. Additionally, if you make EMM a weak reference, **you must make 100% sure to not call any code referencing the EMM assembly if EMM is not loaded**. If you don't, the game will crash for people using your mod.

Registering your mod

In order to have content in your mod added to the EMM system, you need to actually register your mod and then your content. You do this inside your mod's Load() hook. If you made EMM a weakReference, make sure to only call this if EMM is loaded.

```
public override void Load()
{
    EMMLoader.RegisterMod(this);
    EMMLoader.SetupContent(this);
}
```

Or as a weakReference:

```
public override void Load()
{
    Mod mod = ModLoader.GetMod("Loot");
    if (mod != null)
    {
        EMMLoader.RegisterMod(this);
        EMMLoader.SetupContent(this);
    }
}
```

Note: if you want your mod to work on Mono (mac), you must put code that references the EMM assembly in a separate class. It could look something like this:

```
public static class EMMHandler
{
    public static void RegisterMod(Mod mod)
    {
        EMMLoader.RegisterMod(mod);
    }

    public static void SetupContent(Mod mod)
    {
        EMMLoader.SetupContent(mod);
    }
}
```

Which would change your Load() to:

```
public override void Load()
{
    EMMHandler.RegisterMod(this);
    EMMHandler.SetupContent(this);
}
```

Or if as a weakReference:

```
public override void Load()
{
    Mod mod = ModLoader.GetMod("Loot");
    if (mod != null)
    {
        EMMHandler.RegisterMod(this);
        EMMHandler.SetupContent(this);
    }
}
```

Note that, it is best practice to do this for any weakReference. You only have to do this for this part though *if you can make sure to not reference anything in your assembly yourself that references EMM's assembly, such as a modifier*.

After this, you're set! You can now start adding modifiers. To learn how to add modifiers, see the particular section that describes it.

Autoloading glowmask and shader assets

If you want to autoload your own assets, you'll need to add the following to your Load():

```
if (!Main.dedServ)
{
    EMMLoader.RegisterAssets(this, "MyFolder");
}
```

Note that again, it is best practice to add a new method to EMMHandler and call that if you want this to work on Mono.

Make sure to edit "MyFolder" to the folder you want it to register. More details can be found in the specific section for this subject.

1.2 Modifier

To make a new Modifier, simply make a new class that inherits the class `Modifier` in the `Loot.System` namespace.

1.2.1 Properties

Brief properties explanation

Modifiers have the following features:

Power rolls: modifiers can roll its power by 'magnitudes'. This happens when the modifier is rolled on the item, and these values are saved.

Weighted roll chances: modifiers can have different chances of being rolling (weighted rolling). The default weight is 1f. If you increase this number, your modifier will show up more often. Putting it to 2f will make it show up twice as often. Something like 10f is a really high roll chance. For very powerful modifiers, you want the roll chance to be low.

Rarity levels: the total rarity is decided by the sum of the rarity level of all modifiers on the item. The default is 1f. If you have more powerful modifiers, you will want to increase this number to signify the power of it.

Modifiers function like GlobalItems. This means you have access to pretty much all hooks available in GlobalItem (a few exceptions). You can assume the item in the context of a hook (the item parameter) is an item being influenced by that modifier. This is handled by EMM and out of your way, mostly consisting of boilerplate code.

Below I will detail all the properties of ModifierProperties

```
public float MinMagnitude { get; private set; }
public float MaxMagnitude { get; private set; }
public float MagnitudeStrength { get; private set; }
public float BasePower { get; private set; }
public float RarityLevel { get; private set; }
public float RollChance { get; private set; }
public int RoundPrecision { get; private set; }
public float Magnitude { get; private set; }
private float _power;
public float Power
{
    get { return _power; }
    private set
    {
        _power = value;
        RoundedPower = (float)Math.Round(value, RoundPrecision);
    }
}
public float RoundedPower
{
    get;
    private set;
}
```

Properties explanation in context

In short:

1. roll around a certain number (BasePower)
2. control the range in which the power will roll (MinMagnitude and MaxMagnitude)
3. influence the end result of the power (MagnitudeStrength)

The luck stat is something you can roll on gear which influences some of this

In depth:

MinMagnitude and **MaxMagnitude** define the range that **Magnitude** will roll in. For example 10f - 15f will make **Magnitude** roll between 10f and 15f. **MagnitudeStrength** influences the final result of that roll as a multiplier. For example setting this to 2 and rolling the upper bound of the previous example (15f) would result in a Magnitude of 30. **Power** is then calculated by **BasePower***Magnitude**, so a BasePower of 10 in the previous context would result in a Power of 300. When Power is set, it automatically sets **RoundedPower** as well. This property is primarily useful

for tooltips, where you'll likely want to display 230% instead of something like 230.568893%. The precision of the rounding can be adjusted with **RoundPrecision**, which defaults to 0. So by default it will prune all decimals. If you set it to 1, it will leave 1 decimal, at 2 it will leave 2, and so forth...

These Properties (and all other hooks) support inheritance. A good example of that can be found here and here.

Modifying properties

To modify the Properties, you override the GetModifierProperties method and return a new object. All properties have a default value, to set specific ones you can specify the parameter names like so:

```
public virtual ModifierProperties GetModifierProperties(Item item)
{
    return new ModifierProperties(roundPrecision: 1, minMagnitude: 15f, maxMagnitude:␣
↪30f);
}
```

Please note that you need to be careful when using inheritance! Always call base first and then modify whatever properties it returns:

A base class:

```
public abstract class EquipModifier : Modifier
{
    public override ModifierProperties GetModifierProperties(Item item)
    {
        return new ModifierProperties(magnitudeStrength: item.IsAccessory() ? .6f :␣
↪1f);
    }
}
```

If a class inherits from this base class, the properties should be handled like so:

```
public override ModifierProperties GetModifierProperties(Item item)
{
    return base.GetModifierProperties(item).Set(maxMagnitude: 15f);
}
```

1.2.2 Saving and loading data per modifier

You can save and load your own data per modifier, an example can be seen here. Essentially, saving and loading works exactly the same as you are used to. The handling of it is managed by EMM itself and out of your way.

Saving data

Please note that, inside the Save method the TagCompound already contains necessary save data for the modifier (which is automatically handled by EMM). Make sure that you don't clear the TC or make a new instance!

```
public override void Save(Item item, TagCompound tag)
{
    // If you use inheritance, always call base first
    base.Save(item, tag);
```

(continues on next page)

(continued from previous page)

```
    // Any saving here!
}
```

Loading data

```
public override void Load(Item item, TagCompound tag)
{
    // If you use inheritance, always call base first
    base.Load(item, tag);

    // Any loading here!
}
```

1.2.3 Tooltips

Tooltips by modifiers are automatically managed by EMM, and it does the following:

- Item name automatically modified if rarity overrides color and/or adds prefix/suffix
- Adds a rarity tag by name [Modifier:Name]
- Adds a modifier line (description) per line by name [Modifier:Description:i]

However, you can still modify the tooltips per modifier if you want to. The ModifyTooltips method is called after this automatic behaviour.

1.3 ModifierRarity

To make a new ModifierRarity, simply make a new class that inherits the class `ModifierRarity` in the `Loot.System` namespace.

1.3.1 Primary functions

The primary use of a rarity is to provide some representation of how strong a rolled item's modifiers are. For most rarities, you'll just want to be overriding these properties: (example is Common Rarity)

```
public class CommonRarity : ModifierRarity
{
    public override string Name => "Common";
    public override Color Color => Color.White;
    public override float RequiredRarityLevel => 0f;
}
```

A rarity is always rolled by its RequiredRarityLevel, and the highest matching rarity is always selected. A higher required level indicates a more powerful item, so the rarity should signify that.

1.3.2 Extra functions

A rarity can add a prefix and suffix to an item name, as well as change the color of it in the tooltip.

```
    public virtual Color? OverrideNameColor => null;
    public virtual string ItemPrefix => null;
    public virtual string ItemSuffix => null;
```

1.3.3 Saving and loading data

You can save and load data exactly like how you can do it for a Modifier (see above)

1.4 ModifierPool

To make a new ModifierPool, simply make a new class that inherits the class `ModifierPool` in the `Loot.System` namespace.

1.4.1 Concept

The idea of a pool is that it consists of a 'pool' of modifiers of which up to 4 can be rolled on an item. EMM itself uses a single pool that consists of all modifiers in the system.

Also note that, the modifiers on an item are actually saved as a ModifierPool: the Modifiers are inside this ModifierPool.

1.4.2 Primary use

The primary use of a pool should be to roll a 'themed' item. You could also make pools that are rare that consist only of modifiers that work well together, making it more likely the item becomes very potent with the rolled modifiers.

You'll want to populate the `Modifiers` array in your contructor. An example can be seen here

1.5 ModifierEffect

A ModifierEffect signifies the effect of a modifier on a player It should house the implementation, and delegation of such an effect Methods on effects can be delegated from ModPlayer

1.5.1 How to get a ModifierEffect

You can get an effect as follows:

```
ModifierPlayer.Player(player).GetEffect<MyModifierEffect>()
```

1.5.2 UsesEffectAttribute

The UsesEffect attribute is linking a Modifier to a ModifierEffect. This will make the ModifierEffect become activated if we have an item with this modifier.

Usage:

```
[UsesEffect(typeof(MyEffect))]
public class MyModifier : Modifier
```

```
[UsesEffect(typeof(FirstEffect), typeof(SecondEffect), typeof(ThirdEffect)]
public class MyModifier : Modifier
```

1.5.3 DelegationPrioritizationAttribute

This attribute is used to set a custom prioritization for a delegation It allows you to customize at which point your delegation is called in the chain The end result is a prioritization list as follows:

- First part: all delegations prioritized as DelegationPrioritization.Early, order by their level
- Second part: all delegations with no custom prioritization
- Third part: all delegations prioritized as DelegationPrioritization.Late, order by their level

To increase the force put into your prioritization, increase the delegation level

Example use-case

Consider a modifier adding a flat +50 damage, and another modifier giving +100% damage to the item. Without the prioritization options, it sometimes happens that the latter modifier is delegated before the former one. This means the multiplication happens before addition. Example:

```
Base 10 damage. * 100% = 20 damage, + 50 damage = 70 damage
```

```
Base 10 damage. + 50 = 60 damage * 100% = 120 damage
```

You can see how big of a difference this can make. So this is what this functionality is used for: **to prioritize when your delegation is executed in the chain**

How to use

You have two options (technically three): early in the chain or later in the chain.

To be in the middle, simply don't use this attribute.

Examples:

```
// a multiplicative effect wants to be last in the chain, hence 999
[AutoDelegation("OnModifyHitNPC")]
[DelegationPrioritization(DelegationPrioritization.Late, 999)]
```

```
// this particular example wants to happen early, for example mitigating damage␣
↪entirely
[AutoDelegation("OnPreHurt")]
[DelegationPrioritization(DelegationPrioritization.Early, 100)]
```

1.6 Utilities

EMM provides necessary utility functions for modifiers, rarities and pools. These can be called on a mod instance. The utils can be found in EMMUtils. The utils work primarily the same as the ones in tML. An example call to get one of your modifiers:

```
myMod.GetModifier<MyModifier>();
```

1.7 Advanced content creation

In the following section we will go over advanced content creation, which includes an explanation of the more complex systems that are in place that make this possible.

1.7.1 Automated event-based delegation

This mod features what I call 'automated event-based delegation'. The purpose of this system is simple: it allows you to run ModPlayer code, through a Modifier. What does this mean? Just imagine that you need to run ModPlayer code for some Modifier, this system makes that possible **in an easy-to-use yet efficient manner**.

To use this, you do not need to understand how the system works internally: you just need to know how to call the necessary features to have it work. This system is used with ModifierEffects. This is a class not yet covered, so we will do so now.

1.7.2 ModifierEffect & delegation

ModifierEffects live on the player, specifically a ModPlayer, part of EMM called the 'ModifierPlayer'. Do not make these more complicated as they are: they give you access to virtually all ModPlayer hooks similarly to how Modifier exposes most GlobalItem hooks to you. Instead, it does it in a different way. It does not inherit from ModPlayer, it is it's own class instead. This is where the event-based delegation comes into play, as it allows those hooks to actually be 'attached' to an existing ModPlayer hook. Here is what that looks like:

```
public class CursedEffect : ModifierEffect
{
    public int CurseCount;

    public override void ResetEffects(ModifierPlayer player)
    {
        CurseCount = 0;
    }

    [AutoDelegation("OnUpdateBadLifeRegen")]
    private void Curse(ModifierPlayer player)
    {
        if (CurseCount > 0 && !player.player.buffImmune[BuffID.Cursed])
        {
            if (player.player.lifeRegen > 0)
            {
                player.player.lifeRegen = 0;
            }
            player.player.lifeRegen -= 2 * CurseCount;
            player.player.lifeRegenTime = 0;
        }
    }
}
```

The above example showcases a 'CursedEffect', that inherits from ModifierEffect. Our Curse method is being delegated to the `ModPlayer.OnUpdateBadLifeRegen`, notice how the AutoDelegation attribute makes that possible. All you need to make sure is that your method signature matches the one of the hook you want to attach to, where the first parameter is *always* of type ModifierPlayer. Even if the hook normally has no parameters.

You are now correct if you say that the Curse method is being ran from within the OnUpdateBadLifeRegen hook.

The effect is 'attached' to a Modifier in the following way:

```
[UsesEffect(typeof(CursedEffect))]
public class CursedDamage : Modifier
```

Notice the UsesEffect attribute, in which you specify which effect should become active when an item has this modifier. Note that you can specify multiple effects, if you want:

```
[UsesEffect(typeof(EffectOne), typeof(EffectTwo), typeof(EffectThree))]
public class MyModifier : Modifier
```

1.8 The glowmask- and shader-entities system

EMM features a system called entities for glowmasks and shaders. This allows you to apply glowmasks and shaders to items based on its modifiers. (You quite literally attach one to the Modifier itself!)

1.8.1 How does it work?

In the background, EMM handles everything for you. An 'entity' lives on the modifier of an item, which can alter its appearance. Examples:

https://streamable.com/6kj8o

https://streamable.com/ezkm5

1.8.2 How to use it?

Shader entity

Inside your modifier class, simply override `GetShaderEntity()` and return an entity:

```
public override ShaderEntity GetShaderEntity(Item item)
{
    return new ShaderEntity(item,
        GameShaders.Armor.GetShaderIdFromItemId(ItemID.MirageDye),
        drawLayer: ShaderDrawLayer.Front,
        drawOffsetStyle: ShaderDrawOffsetStyle.Alternate,
        shaderDrawColor: Color.IndianRed);
}
```

Note that custom shaders are not supported at this time.

Read below to learn how to use a custom sprite.

Glowmask entity

Inside your modifier class, simply override `GetGlowmaskEntity()` and return an entity:

```
public override GlowmaskEntity GetGlowmaskEntity(Item item)
{
    return new GlowmaskEntity(item);
}
```

Read below to learn how to use a custom sprite.

1.8.3 Using custom assets

EMM has a custom system built in that can automagically load assets that entities can use. This means you can control which parts of the item is being applied a glowmask or shader. An example where only a specific part of the item is being applied a shader: https://streamable.com/ezkm5

How to load my own assets?

All you have to do is specify to register your assets in Load():

```
if (!Main.dedServ)
{
    EMMLoader.RegisterAssets(this, "MyFolder");
}
```

How does it work?

The above code will make it so that assets from the folder `<YourMod>/GraphicAssets` are added to the system. This path is relative to your mod, so if you enter `MyFolder/SubFolder/DeeperFolder` it will look in there. Currently only a single folder is supported, and you should put all assets in there.

Keynote: the assets will be removed from your mod's internal texture array. You can disable this by passing: `clearOwnTextures: false`

Naming rules

All assets can be named by either the item name or its ID. The ID (or type) is recommended for vanilla items. Currently you can only add assets for your own mod or vanilla. Example asset names:

- MyModItem_Glowmask.png –or– MyModItem_Glow.png
- MyModItem_Shader.png –or– MyModItem_Shad.png

As shown, names can be suffixed with either the full variant or shorthand of the asset type. This suffix is important or else the asset will not be added.

For vanilla items, it is recommended to name them by their type, it is however still possible to name them by their name. (EMM will try to resolve its type manually)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Brief overview of common classes

Below is given a brief description of common classes from EMM. To learn more about these classes, visit their respective guides.

CHAPTER 4

Modifier

Defines a modifier, has access to virtually any hook for items that can be used to affect the item. (this goes into effect when an item has that modifier on it)

CHAPTER 5

ModifierEffect

Effects live on the player, and they allow running code in ModPlayer hooks coming from a modifier. There is an auto delegation system behind the scenes that makes this possible.

CHAPTER 6

ModifierPool

So a pool is what modifiers are drawn from when an item rolls it's modifiers. Just consider it a collection of modifiers, because that's all it is. On the item itself, all modifiers are also stored as a ModifierPool

CHAPTER 7

ModifierRarity

The combined strength of modifiers on an item form a numeric value that can be matched with a rarity, which can affect the item's name, change color etc. If you understand these bits, you understand most core stuff of the mod besides a lot of back-end stuff making stuff possible